Pure compile-time functions and classes in the CPH MPL

Jyrki Katajainen

Department of Computer Science, University of Copenhagen
Universitetsparken 5, 2100 Copenhagen East, Denmark
jyrki@di.ku.dk

Abstract. In this paper, we document the facilities available at the CPH MPL, Copenhagen metaprogramming library written in C++. When designing this library, we had three adjectives in mind: small, neat, and tidy. In C++, metaprogramming is a subject where functional programming and imperative programming meet. The values and types created at compile time are immutable in nature, so traditionally the compile-time entities have been processed using techniques known from functional programming. The variadic templates and constant expressions added to C++ have moved metaprogramming closer to the imperative world. Now, it is not necessary to use heavy template-metaprogramming recursion, when a simple loop will do. All this has made the implementation of a metaprogramming library easier. The main goal of this paper is pedagogical: By reading this paper today, you can use metaprogramming in your programs tomorrow. You have to give a few extra options to your compiler and the tools are at your fingertips. But you should not be surprised if you get an error message like "internal compiler error".

Keywords. Software libraries; C++; templates; metaprogramming; compile-time computations

1. Introduction

Background

The CPH STL (Copenhagen standard template library) is a collection of generic algorithms and data structures in the very same spirit as the algorithms and the data-structures part of the C++ standard library. The goal of this project has been to develop the best possible implementations for different standard-library components. Many of the advanced algorithms and data structures are not provided by the standard library for a very good reason: It is not easy to make them practically efficient. We have analysed many components meticulously and performed experiments to understand the fundamental limits better. For more details, see the reports and papers available at the website `http://www.cphstl.dk`.

The CPH MPL (Copenhagen metaprogramming library) is a little brother of the CPH STL. This library contains the metaprogramming tools used in

the CPH STL. All programs included in both of these libraries are placed in the public domain. For more information on how to get the programs, see Section "Software availability" at the end of the paper.

The purpose of this paper is to document some of the basic tools available at the CPH MPL. The main motivation for developing the CPH STL and the CPH MPL has been educational. The tools described have been used in our teaching at the University of Copenhagen. In addition to documentation, we report the results of some simple benchmarks to advice the users about the applicability of the tools. Hopefully, the descriptions given here will inspire you to extend the library and to develop some similar—and even better—tools.

Pure compile-time functions and classes

In Pascal, an ordinal type specifies an ordered, countable collection of values. The collection must be bounded so there is a smallest possible value and a largest possible value. For example, a byte represents a subrange `0..255` of integers. The built-in function `ord` can be used to convert any ordinal value to its corresponding integer value. For a character, this function gives the integer value of that character in the ASCII encoding.

Let us now write the `ord` function in C++; we only consider the specialization for characters. One of the strengths (and weaknesses) of C++ is that things can be done in several different ways. For example, the `ord` function can be written in at least the following five ways; more variants would be obtained by omitting the **constexpr** specifier. (For the latest online documentation on constant expressions, we refer to [4].)

Non-template function:

```
unsigned char ord(char symbol) {
  return static_cast<unsigned char>(symbol);
}
```

Constant-expression function:

```
constexpr unsigned char ord(char symbol) {
  return static_cast<unsigned char>(symbol);
}
```

Constant-expression function template:

```
template<char symbol>
constexpr unsigned char ord() {
  return static_cast<unsigned char>(symbol);
}
```

Variable template:

```
template<char symbol>
constexpr unsigned char ord ← static_cast<unsigned char>(symbol);
```

Class template as a function—return value via a member:

```
template<char symbol>
class ord {
public:

  static constexpr unsigned char value ← static_cast<unsigned char
    >(symbol);
};
```

Thus, a function can take compile-time template arguments and run-time function arguments. For example, to call the above-mentioned functions to get the ordinal value of character 'a', you should write `ord('a')`, `ord('a')`, `ord<'a'>()`, `ord<'a'>`, or `ord<'a'>::value`. The last four forms can even give the ordinal value at compile time, so that—at the call place—the generated code simply contains the constant 97. It is probable that the compiler will propagate the constant further, so the outcome is no code at all.

In principle, `ord` is an associative array that can be accessed with a character and the function produces the associated value. As discussed in many textbooks, table-driven methods can be powerful (see, for example, [15, Chapter 18]). With the new compile-time tools, you can avoid writing separate programs to generate the tables. And you do not need to do the computations beforehand, but you can make them visible at the call place and rely on the fact that the compiler will do the necessary optimizations.

At compile time, a program operates on *values* and *types*; at run time, it operates on *objects*. An integer 97 can be a compile-time or a run-time entity, whereas a type is always a compile-time entity. To be technically correct and to cover the whole universe, I should have written non-types and types, but I deliberately try to avoid the bland term *non-type*. Also, parameter packs and templates can be passed as template parameters, but we will put the emphasis on values and types.

Based on the syntax only, it can be difficult to see what is computed at compile time and what is not. For example, the built-in compile-time type function **sizeof** can be used in two ways to query the size (measured in bytes) of a type expression: 1) For a type `T`, you can ask its size by using the normal function-call syntax **sizeof**`(T)` and 2) for an expression `e`, you can write **sizeof** `e` to get the size of the object representation of the type that would be returned by expression `e`.

When a function `f` computes its result for argument `x` at compile time, we prefer the syntax `f<x>` and, when it computes its result at run-time, we prefer the syntax `f(x)`. Though, unlike run-time functions, a compile-time function that takes no arguments is invoked without the angle brackets. Naturally, we have no problem with functions that take both compile-time and run-time arguments. As it is often the case, the compiler can deduce the compile-time arguments from the run-time arguments, so it may not be necessary to specify them explicitly.

A *type function* is a mapping from a type to a value or another type. Since

any type can be used as a template parameter, all type functions can be written using our preferable syntax. Because of the restrictions set for non-type template parameters, we cannot fully avoid the use of the **constexpr** functions when operating on values. (For the latest online documentation on template parameters and template arguments, we refer to [8].)

We call a function `f` a *pure compile-time function* if an invocation can be written in the form `f<...>`. Furthermore, we say that a class is a *pure compile-time class* if it only takes template arguments and has no run-time data associated with it. In this paper, we focus on the pure compile-time functions and classes in the CPH MPL.

Observe that even though, at the interface level, we restrict ourselves to pure compile-time functions and classes, internally we rely on the new **constexpr** extensions added to the language. Without the new additions, it would not be allowed to use loops in our metaprograms since a loop variable cannot be a template argument. With simple loops, it is sometimes possible to avoid the heavy template-metaprogramming recursion.

In the C++ standard library, the suffix `_v` is often added to the name of a type function, if it returns a value, and the suffix `_t`, if the result is a type. See, for example, the type functions defined in the header `<type_traits>`. Instead of doing this, we try to use the following naming convention:

Predicate. type → **bool**; name begins with `is_`
Example: `is_character`

Introspection. type → **bool**; name begins with `has_`
Example: `has_type_iterator`

Property. type → integer; name begins with a noun
Example: `width`

Constant. type × integer → constant; name begins with a determiner
Example: `some_trailing_ones`

Transformation. type → type; name begins with a verb
Example: `make_unsigned`; Is `count` in `count_if` a noun or a verb? It is a noun.

Relationship. type × type → **bool**; name begins with `is_`
Example: `is_safely_convertible`.

Related work

Below you get a sample of how other authors have defined the concepts *metaprogramming* and *metaprogram*:

[1, p. 2]. "If you dissect the word *metaprogram* literally, it means 'a program about a program'. A little less poetically, a metaprogram is a program that manipulates code."

[9, p. 11]. "Checking specifications, performing optimizations and weaving, and assembling implementation components all require some form of metaprogramming. Metaprograms are programs manipulating other programs or themselves."

[20, p. 781]. "Metaprogramming is a combination of 'meta' and programming: *a metaprogram is a compile-time computation yielding types or functions to be used at run time*."

In C++, you can do native-language metaprogramming in two ways: 1) You can write preprocessor macros to generate code, and 2) you can perform compile-time computations using templates and constant expressions. Thinking reflectively, what is available at the CPH MPL, I would define a *metaprogram* to be a description of a computation that yields values, types, or functions to be used when generating a runnable program. This definition does not differ much from Stroustrup's definition. In my opinion, the first two definitions above are too general to describe what we are doing.

There are several other metaprogramming libraries available: the most influential ones have been Loki [2] and Boost MPL [1]. So we are by no means the first movers in the field. After the addition of variadic templates and constant expressions to C++11, several new metaprogramming libraries were released in the Internet (see, for example, [10, 16]). Your search engine can also direct you to some interesting blogs that discuss the matter.

To be honest, the syntax used in the early tools was unbearable. With the recent additions to the language, the situation has improved a lot. The main goal of the CPH MPL was to show that a metaprogramming library can be simple.

2. Architectural overview

The metaprogramming facilities available at the CPH MPL were created in connection with the development of the width-specific class templates for manipulating integers [14]. Hence, the tools were primarily targeted for this application. At the moment, the tools are packaged in three header files:

`cphmpl/functions.h++`. The compile-time functions fall into two groups: value functions and type functions. A good example of a value function is `cphmpl::lg<n>` which computes the binary logarithm of number `n`. In the CPH STL, the most used type function is `cphmpl::width<T>` which computes the width of an integer type `T` (measured in bits). For example, the width of `cphstl::constant<-3>` is 2. The class template `cphstl::constant` is similar to `std::integral_constant`; it encapsulates a compile-time integer value as a type. The essential difference is that `cphstl::constant` knows its width—which is computed at compile time.

`cphmpl/lists.h++`. A compile-time list is a heterogeneous data structure which keeps its data in the template parameters. The lists have two incarnations: a value list `cphmpl::valuelist` can hold any mix of non-type template parameters and a type list `cphmpl::typelist` can hold any mix of types. Two examples of homogeneous value lists are a list of function pointers and a list of characters.

`cphmpl/concepts.h++`. The concepts are on the way to the language. We find the experimental support provided by `g++` quite satisfactory—

compared to the old alternatives. Since the language facilities are still evolving, we will not discuss our experimental concept definitions any further. Though, we will use concepts to check some simple syntactical restrictions. We want to emphasize that constraint- and concept-based overloading is a convenient feature that you have to be aware of. (For the latest online documentation on concepts, we refer to [5].)

In the forthcoming sections, we give a more detailed picture of the tools provided. In Section 3, we explain how some of the functions are implemented and can be used. In Section 4, we show how a list of characters can be used in the manipulation of string literals at compile time, and how a list of types can be used in the manipulation of template argument lists. In Section 5, we present the results of some simple experiments that illustrate how the compile-time lists perform in practice. Finally, in Section 6, we conclude the paper with a brief discussion on the future trends.

3. Functions

In this section, we go briefly through which kind of compile-time functions there are in the CPH MPL.

Mathematical functions

Let us use `lg` to denote the base-2 logarithm. The number of bits in the binary representation of a natural number is an often-needed quantity. For an integer $\texttt{n} > 0$, this number is $\lfloor \texttt{lg n} \rfloor + 1$. Hence, it would be practical to have a compile-time function that computes $\lfloor \texttt{lg n} \rfloor$.

The following two programs illustrate how this function could be written 1) using the traditional template-metaprogramming approach relying on template specialization and 2) using the new **constexpr** approach.

```
namespace deprecated {

  template<auto n>
  class lg {
  public:

    static constexpr auto value ← 1 + lg<n / 2>::value;
  };

  template<>
  class lg<1> {
  public:

    static constexpr auto value ← 0;
  };
}
```

```
namespace cphmpl {
```

```
2
3   namespace helper {
4
5     template<auto n>
6     constexpr auto lg() {
7       using N ← decltype(n);
8       N result ← 0;
9       N i ← n;
10      while (i > 1) {
11        i ← i / 2;
12        ++result;
13      }
14      return result;
15    }
16  }
17
18  template<auto n>
19  constexpr auto lg ← helper::lg<n>();
20 }
```

The latter was selected to the library for two reasons: 1) It gives a reasonable, albeit not the correct answer for `n` = 0. 2) It can be invoked using our preferable syntax: `cphmpl::lg<512>`.

Exercise 1. Extend the library with some other useful mathematical functions that perform the calculations at compile time. □

Introspection

In psychology [22], *introspection* means the examination of one's own conscious thoughts and feelings. Here we talk about type introspection—more specifically, *compile-time type introspection*. In particular, we are *not* interested in run-time properties of an object.

We need introspection in constraint-based overloading. As a concrete example, consider the integer class templates developed for the CPH STL. For built-in types we can query their properties using the template specializations `std::numeric_limits` in the header `<limits>`. However, these tools fall in short for the new integer types. Knowing this to be a problem, we programmed the new class templates so that the numbers know their own traits and limits. For the integer type `cphstl::ℕ<8>`, for instance, you can write `cphstl::ℕ<8>::max` to get the maximum value that can be kept in that type. The answer 255 will be generated by the compiler at the call place.

Hereafter any generic algorithm, that operates on integer type `T` and needs `T::max`, must treat built-in types and the `cphstl` integers differently. In this particular case, we need a tool to query whether a type has a static member variable `max` or not. More generally, it would be useful if we could query whether a type has a specific member—static variable, static function, variable, enumeration, type, function, or class.

One option was to extend `std::numeric_limits` by adding the necessary

specializations for the new types. Actually, we did this, but after writing 500 lines of some trivial—but error-prone—code, we came to the conclusion that this cannot be the right solution. Of course, the optimal solution would be to let the built-in types to know their traits and limits, too. However, this solution was not in our circle of influence, so we had to accept the fact that both kinds of integers are present.

For a long time, it has been know how introspection can be implemented (see, for example, [21, Chapter 15]). Also, several proposals can be found from the Internet. However, many of the proposed solutions are complicated and fragile. We found a usable solution from the GNU implementation of the standard library under the header `<type_traits>`. This solution relies on macros, so it was only used internally inside the library implementation.

Consider now the query whether a type has a static member variable `is_integer`. Below you see how this task can be accomplished with the tools being used in the current implementation of the standard library.

```
#define HAS_STATIC_VARIABLE(MEMBER) \
template<typename T, typename ← std::void_t<>> \
class has_static_variable_##MEMBER \
  : public std::false_type { \
}; \
 \
template<typename T> \
class has_static_variable_##MEMBER<T, std::void_t<decltype(T::
    MEMBER)>> \
  : public std::true_type { \
};

namespace deprecated {

  namespace helper {

    HAS_STATIC_VARIABLE(is_integer)

  }

  template<typename T>
  constexpr bool has_static_variable_is_integer ← helper::
    has_static_variable_is_integer<T>::value;
}
```

The operational procedure is similar for other kinds of members. This means that you have to type a lot for a single member.

As you can see, this code is in the namespace `deprecated`. This solution was rejected after we looked into the experimental support of concepts and constraints to be added to C++20. All the above complications are away with the concepts! Even though `cphmpl::has_static_variable_is_integer` is not a real concept, since we are just checking a syntactic restriction, this tool is perfect for constraint-based overloading that we desperately need.

```
namespace cphmpl {

  namespace helper {

    template<typename T>
    concept bool has_static_variable_is_integer ←
    requires {
      T::is_integer;
    };
  }
}
```

Predicates

The standard library offers several examples of unary type predicates in the header <type_traits>. However, some of the tools only work for the built-in types. Hence, it is sometimes necessary to extend the existing facilities. Below you see two such extensions taken from the CPH MPL. The first one of these examples illustrates how to use the cphmpl::typelist facility—we will look at this tool in more detail in Section 4. The second example shows how to rely on constraint-based overloading when selecting a suitable implementation for some particular type.

```
#include "cphmpl/typelist.h++"
#include <type_traits>

namespace cphmpl {

  namespace helper {

    template<typename T>
    constexpr bool is_character() {
      using character_types ← cphmpl::typelist<signed char,
        unsigned char, char, wchar_t, char16_t, char32_t>;
      using U ← typename std::remove_cv<T>::type;
      return character_types::is_member<U>;
    }
  }

  template<typename T>
  constexpr bool is_character ← helper::is_character<T>();
}
```

```
#include <limits>
#include <type_traits>

namespace cphmpl {

  namespace helper {

```

```
8       template<typename T>
9       concept bool has_static_variable_is_integer ←
10      requires {
11        T::is_integer;
12      };
13
14      template<typename T>
15      class is_integer_selector {
16      public:
17
18        static constexpr bool value ← false;
19      };
20
21      template<typename T>
22      requires std::is_fundamental_v<T>
23      class is_integer_selector<T> {
24      public:
25
26        static constexpr bool value ← std::numeric_limits<T>::
        is_integer;
27      };
28
29      template<typename T>
30      requires has_static_variable_is_integer<T>
31      class is_integer_selector<T> {
32      public:
33
34        static constexpr bool value ← T::is_integer;
35      };
36    }
37
38    template<typename T>
39    constexpr bool is_integer ← helper::is_integer_selector<T>::
        value;
40  }
```

Properties

All the `cphmpl::is_*` and `cphmpl::has_*` functions tell whether a type has some particular property. A bit more general type function produces an integer as its result. The built-in function **sizeof** belongs to this category. Since the function `cphmpl::width` is the most used type function in the CPH STL, we have to release its implementation. If there are bugs, let us know!

```
1  #include <limits>
2  #include <type_traits>
3
4  namespace cphmpl {
5
6    namespace helper {
7
```

```
8       template<typename T>
9       concept bool has_static_variable_width ←
10      requires {
11        T::width;
12      };
13
14      template<typename T>
15      class width_selector {
16      private:
17
18        using S ← decltype(sizeof(0));
19        using B ← unsigned char;
20        static constexpr S bits ← std::numeric_limits<B>::digits;
21
22      public:
23
24        static constexpr S value ← bits * sizeof(T);
25      };
26
27      template<typename T>
28      requires std::is_fundamental_v<T>
29      class width_selector<T> {
30      private:
31
32        using S ← decltype(sizeof(0));
33        static constexpr S sign ← std::numeric_limits<T>::is_signed;
34        static constexpr S digits ← std::numeric_limits<T>::digits;
35
36      public:
37
38        static constexpr S value ← sign + digits;
39      };
40
41      template<typename T>
42      requires has_static_variable_width<T>
43      class width_selector<T> {
44      private:
45
46        using S ← decltype(sizeof(0));
47
48      public:
49
50        static constexpr S value ← T::width;
51      };
52    }
53
54    template<typename T>
55    requires cphmpl::is_integer<T>
56    constexpr auto width ← helper::width_selector<T>::value;
57  }
```

Constants

In programs that manipulate individual bits in a word, one often needs some form of masks to zero out parts of that word. In this kind of code, some hexadecimal constants like `0x0000FFFF` are not at all rare. When such hard-coded masks are used, portability might become a problem. For example, the above constant is tuned for a computer having 32-bit words. On a computer having 64-bit words, the code does not work any more. Simply, there should be a more generic way of generating these masks. And because of the efficiency, the masks should be generated at compile time, if feasible.

The function `cphmpl::some_trailing_ones<T, count>` can be used to generate a mask that has `count` one bits at the end of the word. Assume that we operate on the words of type `W`. Now a word, for which the upper half is filled with zeros and the lower half is filled with ones, could be created as follows:

```
using S ← decltype(sizeof(W));
constexpr S w ← cphmpl::width<W>;
static_assert(cphmpl::is_power_of_two<w>);
constexpr W mask ← cphstl::some_trailing_ones<W, w / 2>;
```

For a 32-bit word, this generates the same code as the constant `0x0000FFFF`, but this code also works for a 64-bit word.

You find the implementation of `cphmpl::some_trailing_ones` below. The helpers `all_zeros<T>()`, `all_ones<T>()`, and `power_of_two<T, count>()` are described in the appendix.

```
namespace cphmpl {

  namespace helper {

    template<typename T, std::size_t count>
    requires cphmpl::is_unsigned<T>
    constexpr T some_trailing_ones() {
      if constexpr (count == 0) {
        return all_zeros<T>();
      }
      else if constexpr (count ≥ cphmpl::width<T>) {
        return all_ones<T>();
      }
      else {
        return power_of_two<T, count>() − 1;
      }
    }
  }

  template<typename T, std::size_t count>
  constexpr T some_trailing_ones ← helper::some_trailing_ones<T,
    count>();
}
```

Exercise 2. Propose some generic constants that would be suitable for general use and could be available at the CPH MPL. Provide a compile-time implementation for one such constant. □

Transformations

As an example of a type transformation, consider `std::make_unsigned`. It works fine for built-in types, but not for `cphstl` integer types. To support this type transformation, the `cphstl` integer types must know their *companion type*. For an unsigned integer type, its companion is the signed variant that is of the same width; and for a signed integer type, its companion is the corresponding unsigned type.

```
namespace cphmpl {

  namespace helper {

    template<typename T>
    concept bool has_type_companion ←
    requires {
      typename T::companion;
    };

    template<typename T>
    class make_unsigned_selector {
    public:

      using type ← T;
    };

    template<typename T>
    requires cphmpl::is_signed<T> and std::is_fundamental_v<T>
    class make_unsigned_selector<T> {
    private:

      using E ← typename std::remove_reference<T>::type;

    public:

      using type ← typename std::make_unsigned<E>::type;
    };

    template<typename T>
    requires cphmpl::is_signed<T> and has_type_companion<T>
    class make_unsigned_selector<T> {
    public:

      using type ← typename T::companion;
    };
  }

```

```
39    template<typename T>
40    using make_unsigned ← typename helper::make_unsigned_selector<T>
        ::type;
41  }
```

Relationships

Naturally, a type function can take many types as its arguments. In the standard library, a handful of such functions are available like `std::is_same` and `std::is_convertible`. We have used these as a model and created yet another useful binary predicate: `cphmpl::is_safely_convertible`. Below you see its implementation. This function is used in the CPH STL to avoid unsafe conversions between integers of different sign and width.

```
1   namespace cphmpl {
2
3     namespace helper {
4
5       template<typename S, typename T>
6       requires cphmpl::is_integer<S> and cphmpl::is_integer<T>
7       constexpr bool is_safely_convertible() {
8         if constexpr (is_unsigned<S> and is_signed<T>) {
9           return width<S> + 1 ≤ width<T>;
10        }
11        else if constexpr (is_signed<S> and is_unsigned<T>) {
12          return false;
13        }
14        else {
15          return width<S> ≤ width<T>;
16        }
17      }
18    }
19
20    template<typename S, typename T>
21    constexpr bool is_safely_convertible ← helper::
        is_safely_convertible<S, T>();
22  }
```

4. Lists

We name the compile-time data structures as *lists* because only their ends are directly accessible. To access other elements, a linear scan is involved. Thus, the manipulation of compile-time lists can be inefficient, but at compile time the lists are normally not long so some inefficiency is acceptable.

As compile-time constructs in general, a compile-time list is immutable; after the construction it cannot be changed. However, using the available static member variables and nested types, new values and type aliases can be created. Hence, for example, the operation `pop_front` creates a new list

and leaves the original unchanged. Iterators for such a list are *cursors*, each referring to a particular position on that list. For a list `L`, the cursors are of type `L::cursor`. For a cursor `i`, `L::at<i>` gives the `i`'th value of `L`, if `L` is a value list; and `L::get<i>` gives the `i`'th type of `L`, if `L` is a type list. `L::front` and `L::back` are synonyms for the first and the last elements, respectively.

Some operations take type functions—predicates or transformers—as their arguments. We write these type functions using the class-template-as-a-function syntax so that the return value is obtained via a member—**static constexpr** `value` if the function returns a value and nested `type` if the result is a type. With this traditional syntax, it was easier to get the compiler to accept these functions.

Now consider the standard list operation `filter` that takes a list `L` and a unary predicate `P` as its input, and produces another list as its output containing every element `X` of `L` for which `P<X>::value` is **true**. When the list contains values, the signature of the predicate is **template<auto> class**; when the list contains types, the signature is **template<typename> class**.

Because of the distinction between values and types, there are two kinds of lists: value lists and type lists. The public part of the class template `cphmpl::valuelist` is shown in Figure 1. The public part of the class template `cphmpl::typelist` is similar. Except the `at` and `get` operations, the other operations supported by the different list structures are the same. However, be aware of the differences in the signatures of the entities we operate on.

Let $\mathcal{P}$ be an unexpanded parameter pack[1], `L` and `L′` two lists, `X` some element, `F` a transformer that transforms `X` to `F<X>::value` or `F<X>::type`; and `P` a predicate, which returns its result via a static member `P<X>::value`. New values and type aliases can be created as follows:

- construction: $\mathcal{P} \rightarrow$ `L`: create a list having the elements of an expanded parameter pack $\mathcal{P}$**...** as its members.
- `L::length` → `integer`: a value denoting the number of elements in `L`.
- `L::is_empty` → **bool**: a Boolean value telling if `L` is empty or not.
- `L::at<i>` → `X` or `L::get<i>` → `X`: the `i`'th element of `L` (0-base indexing); undefined if the cursor `i` is out of bounds.
- `L::front` → `X`: the first element of `L`; undefined if `L` is empty.
- `L::back` → `X`: the last element of `L`; undefined if `L` is empty.
- `L::push_front<X>` → `L′`: a list that contains `X` as its front followed by all the elements of `L`.
- `L::pop_front` → `L′`: a list that contains all the elements of `L` except its front; undefined if `L` is empty.
- `L::push_back<X>` → `L′`: a list that contains all the elements of `L` followed by `X` as its back.
- `L::pop_back` → `L′`: a list that contains all the elements of `L` except its back; undefined if `L` is empty.
- `L::transform<F>` → `L′`: a list where, for a transformer `F`, every element `X` of `L` is mapped to `F<X>::value` or `F<X>::type`.

[1] In all transliterated programs, parameter packs are displayed using calligraphic letters.

```
1  namespace cphmpl {
2
3    template<auto... Ecal>
4    class valuelist {
5    public:
6
7      using self ← valuelist<Ecal...>;
8      using size_type ← decltype(sizeof(0));
9      using cursor ← decltype(sizeof(0));
10
11     static constexpr size_type length ← sizeof...(Ecal);
12     static constexpr bool is_empty ← false;
13     static constexpr auto front ← at_helper<self, 0>::value;
14     static constexpr auto back ← at_helper<self, length − 1>::
       value;
15
16     template<cursor index>
17     static constexpr auto at ← at_helper<self, index>::value;
18
19     template<auto element>
20     using push_front ← valuelist<element, Ecal...>;
21
22     template<auto element>
23     using push_back ← valuelist<Ecal..., element>;
24
25     using pop_front ← typename pop_front_helper<self>::type;
26
27     using pop_back ← typename pop_back_helper<self, valuelist<>>
       ::type;
28
29     template<template<auto> class P>
30     using filter ← typename filter_helper<self, valuelist<>, P>::
       type;
31
32     template<template<auto> class F>
33     using transform ← typename transform_helper<self, valuelist<>,
        F>::type;
34
35     template<template<auto> class P>
36     static constexpr cursor find_if ← find_if_helper<self, length,
       P>::value;
37
38     template<template<auto> class P>
39     static constexpr cursor count_if ← count_if_helper<self, 0, P>
       ::value;
40
41     template<auto element>
42     static constexpr bool is_member ← is_member_helper<self,
       element>::value;
43   };
```

Figure 1. The public part of class template `cphmpl::valuelist`; the private part declares the helper templates. The specialization for `cphmpl::valuelist<>` is not shown.

- `L::filter<P>` → `L′`: a list that, for a unary predicate `P`, contains every element `X` of `L` for which `P<X>::value` is **true**.
- `L::find_if<P>` → `cursor`: the position of the first element `X` in `L` for which `P<X>::value` is **true**; `L::length` if no such element exists.
- `L::count_if<P>` → `integer`: the number of elements in `L` for which `P<X>::value` is **true**.
- `L::is_member<X>` → **bool**: a Boolean value telling if `X` is in `L` or not.

Charlist

As a warmup, let us look at `cphmpl::charlist` which is a compile-time list of characters. The definition of this class is straightforward:

```
#include "cphmpl/valuelist.h++"
#include <type_traits>

namespace cphmpl {

  template<char... Ecal>
  requires std::conjunction_v<std::is_same<char, decltype(Ecal)>...>
  class charlist
    : public valuelist<Ecal...> {
  public:

    using self ← charlist<Ecal...>;
    using value_type ← char;
  };
}
```

In the literature, this data structure appears under the names *constant string*, ***constexpr** string*, or `char_sequence`. At this point, you may want to compare the above solution to those available in the Internet; there are many smart solutions out there (see, for example, [17, 19, 23]).

One significant feature is that a `charlist` can hold $10^{\texttt{i}}$ characters—for some integer `i`—at compile time, but according to my compiler its size is one byte. This sounds like a good deal! I wrote this class in this way since it was difficult to write the constructors for a compile-time class that actually stored some data. Therefore, `cphmpl::charlist` and other lists at the CPH MPL have no run-time data associated with them. Compared to some **constexpr** alternatives, these lists are unusable at run time since template arguments can only be compile-time constants.

Below you see the `cphmpl::charlist` class template in action:

```
#include "cphmpl/charlist.h++"

int main() {
  using hello ← cphmpl::charlist<'h', 'e', 'l', 'l', 'o'>;
  static_assert(hello::length == 5);
  hello instance;
```

```
7     static_assert(instance.front == 'h');
8     static_assert(instance.at<1> == 'e');
9     static_assert(instance.back == hello::back);
10    using ello ← hello::pop_front;
11    using fello ← ello::push_front<'f'>;
12    using fellow ← fello::push_back<'w'>;
13    static_assert(fellow::length == 6);
14    static_assert(fellow::is_member<'l'> == true);
15    static_assert(fellow::is_member<'x'> == false);
16
17    return 0;
18  }
```

The cphmpl::charlist class template has been used in the implementation of the user-defined literals provided by the CPH STL. For example, one can enter binary constants by typing "1001,1000,0011,1010"_2 and decimal constants by typing "1 000 000"_10. The idea of using commas and spaces to improve the readability of the literals is from Schurr [17]. In this context, it is significant that the exact width of the numbers is known at compile time. For further details, we refer to the companion paper [14].

Exercise 3.[13] Consider the library implementation of **operator** ""_2:

```
1  template<typename C, C... Ccal>
2  requires cphmpl::is_character<C>
3  constexpr auto operator ""_2() {
4    using L ← cphmpl::charlist<Ccal...>;
5    return charlist_to_integer<L, 2>();
6  }
```

(a) The first thing to do in the function charlist_to_integer<L, base>() is to clean up the list L by removing all commas and spaces. Sketch in pseudo-code—or in real code if you prefer—how this clean-up can be done. Your description should be detailed enough so that you can analyse the performance of the clean-up routine.

(b) Analyse the compilation time—as the function of the number of characters in L—when your clean-up routine is compiled by your compiler. Make your assumptions explicit.

(c) This topic is hot! Discuss briefly the significance of the **constexpr** extensions added to C++. Give also your prediction of the future in this matter. □

Intlist

Another important special case is a list of integers. The declaration of cphmpl::intlist is equally easy as that of cphmpl::charlist.

```
1  #include "cphmpl/valuelist.h++"
2  #include <type_traits>
3
4  namespace cphmpl {
```

```
5
6     template<int... Ecal>
7     requires std::conjunction_v<std::is_same<int, decltype(Ecal)>...
        >
8     class intlist
9       : public valuelist<Ecal...> {
10    public:
11
12      using self ← intlist<Ecal...>;
13      using value_type ← int;
14    };
15  }
```

The pattern, how to make a heterogeneous list into a homogeneous one, should now be obvious. Even if `cphmpl::intlist` seems to be a naive data structure, the standard library offers a similar tool: `std::integer_sequence`. Compared to `cphmpl::intlist`, it only provides one static member function `size()`, which returns the length of the parameter pack. Another difference is that `std::integer_sequence` has one extra template parameter `T` that specifies the type of the remaining template parameters. Unnecessarily, this extra parameter complicates the use of this class template. The alias template `std::index_sequence` has the same interface as `cphmpl::intlist`, but we encountered problems with its use, since it is just an alias, not a type.

The class template `std::index_sequence` has turned out to be a useful tool when operating with tuples. More specifically, it can be used to establish a mapping between the `i`'th index (cursor), the `i`'th type, and the `i`'th value. Utilizing this connection with the parameter expansion magic [6], a lot of code can be generated by just typing three dots `...`. If you do not agree, look at the **requires** clause in the above **class** declaration one more time.

Typelist

A type list is a tool to manipulate a collection of types. The world was made aware of the usefulness of this tool by Alexandrescu [2, Chapter 3]. His implementation followed the traditional functional style, where each type function took a type list as its first template parameter. After the introduction of variadic templates to C++, the type lists attracted renewed interest and several home-brewed implementations appeared in the Internet (see, for example, [10, 16]).

When writing the library code, we wanted to achieve two things: 1) As to a value list, a type list should be a pure compile-time data structure having no run-time data associated with it. 2) There should be no pollution in the namespace `cphmpl`. Hence, all the helper classes were implemented inside the main class. No private subnamespaces were introduced either.

You saw this class template in action when we checked whether the given type is a character type or not. As another simple example, consider the following program:

Processor. Intel® Core™ i7-6600U CPU @ 2.6 GHz (turbo-boost up to 3.6 GHz) × 4
Word size. 64 bits
Main memory. 8.038 GB
Operating system. Ubuntu 17.10
Linux kernel. 4.13.0-16-generic
Compiler. `g++` version 7.2.0
Compiler options. `-O3 -std=c++17 -x c++ -Wall -Wextra -fconcepts`
Performance analysis tool. `perf stat`

Figure 2. Hardware and software used in the tests.

```
#include "cphmpl/typelist.h++"
#include <iostream>
#include <string>

template<typename T>
void ignore_warning(T) {
}

int main () {
  using L = cphmpl::typelist<int, std::string, void*>;
  L::get<0> x = 0;
  L::get<1> s = std::string("This works!");
  std::cout << s << "\n";
  L::get<2> p = nullptr;
  ignore_warning(p);
  return x;
}
```

5. How well do the class templates perform?

To get an idea, how well the compiler can handle the created templates, some simple experiments were carried out with the class template `cphmpl::intlist`. Two kinds of tasks were considered:

1. Create a compile-time list `L` of `n` consecutive integers starting from zero. The same computation can be carried out by using the alias template `std::make_index_sequence` available at the standard library.
2. Apply the function `L::is_member` for a number that is larger than `n`. That is, we want to perform an unsuccessful search.

A summary of our test environment is given in Figure 2. Briefly, the test computer could be characterized as a standard laptop that one can buy from any well-stocked computer store. All the experiments were run under Linux and the programs were compiled using the GNU `g++` compiler.

According to any textbook on algorithms, both of these tasks should have linear time complexity since we do not exploit the knowledge that the numbers are sorted. In this context, we need to be more careful with our analysis. Let us see how `cphmpl::is_member` has been implemented:

```
template<typename, auto>
class is_member_helper;

template<auto element>
class is_member_helper<valuelist<>, element> {
public:

  static constexpr bool value ← false;
};

template<auto head, auto... T, auto element>
class is_member_helper<valuelist<head, T...>, element> {
public:

  static constexpr bool value ← (head == element) ?
    true : is_member_helper<valuelist<T...>, element>::value;
};
```

Let `V[0:n)` be a shorthand for `cphmpl::valuelist<0, 1,..., n − 1>`. First, the calling program instantiates `V[0:n)`. Then, the above recursive program instantiates `V[1:n)`, `V[2:n)`, and so on. What does an instantiation mean? Without knowing the details how the compiler handles a list of template arguments, a good guess is that, to get from `V[0:n)` to `V[1:n)`, the last `n − 1` values are to be copied from one place to another. To sum up, `n` instantiations are done and the cost of the `i`'th instantiation is proportional to `n − i`. This means $\Theta(n^2)$ compilation time (the number of word operations). Thus, there is not much hope to be able to carry out this task for $n = 10^6$.

First program: Generate the numbers beforehand

In the first experiment, I wrote a program that outputted a list of integers 0, 1,..., `n − 1` to a file named `numbers.i++`; here `i` comes from *include*. To accomplish the tasks, I wrote another program that 1) includes the generated numbers to create a compile-time data structure `cphmpl::intlist` and 2) then uses this data structure to perform one unsuccessful search. The driver below is legal C++ even if the use of #include in the middle is a bit untraditional.

```
#include "cphmpl/intlist.h++"
#include <limits>

int main() {
  using L ← cphmpl::intlist<
#include "numbers.i++"
    >;
  constexpr int outsider ← std::numeric_limits<int>::max();
```

```
9     static_assert(L::is_member<outsider> == false);
10    return 0;
11  }
```

I repeated the test for $\texttt{n} = 10^{\texttt{i}}$, for different values of $\texttt{i} \in \{1, 2, ...\}$, to see how big i can grow. For $\texttt{n} \leftarrow 1000$, the compilation was terminated:

```
../cphmpl/intlist.h++:13:12:   required from here
/usr/include/c++/7/type_traits:148:12: fatal error: template
    instantiation depth exceeds maximum of 900 (use -ftemplate-depth
    <- to increase the maximum)
     struct __and_<_B1, _B2, _B3, _Bn...>
            ^~~~~~~~~~~~~~~~~~~~~~~~~~~~~
```

I hope that you remember what I wrote about the three dots. The error message refers to the line where the **requires** clause is. Simply, the template recursion became too deep. Since we are in the production mode, we can temporarily comment out this check and just try again.

```
intlist.c++:9:18:   required from here
../cphmpl/valuelist.h++:101:29: fatal error: template instantiation
    depth exceeds maximum of 900 (use -ftemplate-depth<- to increase
    the maximum)
       static constexpr auto value <- at_helper<valuelist<T...>, i
     - 1>::value;
                             ^~~~~
```

Good news is that now we could create the data structure for $\texttt{n} \leftarrow 1000$, but the implementation of at is also recursive. There was nothing else to do than to increase the maximum recursion depth. This helped; for $\texttt{n} \leftarrow 1000$ the compilation succeeded, but I got a more serious problem for $\texttt{n} \leftarrow 10000$.

```
g++: internal compiler error: Killed (program cc1plus)
Please submit a full bug report,
with preprocessed source if appropriate.
See <file:///usr/share/doc/gcc-7/README.Bugs> for instructions.
```

Table 1 gives a summary of the results obtained for the three successful test runs. Both the compilation time and the number of page faults incurred during the compilation are reported. These numbers show that, even if the program is simple, there is a lot going on behind the scenes. In all three cases, the size of executable produced was the same: about 8 KB.

Table 1. Performance of the program taking a pregenerated list of integers. The compilation times are in seconds.

n	Construction		Search	
	time	page faults	time	page faults
10	0.30	7 359	0.02	177
100	0.31	$25 \cdot 10^3$	0.10	$7.5 \cdot 10^3$
1000	60	$1.6 \cdot 10^6$	595	$67 \cdot 10^3$

Let us look at the numbers in Table 1 more closely. For construction, the increase in the number of page faults indicates that the amount of memory used during the compilation is enormous. For unsuccessful search, the increase in the compilation time indicates that the guess of quadratic performance seems to hold. The conclusion is that, if you want to use the tools available at the CPH MPL for lists having much more than 100 elements, you should probably rethink your approach.

Exercise 4.** There are several avenues how the state of the art could be advanced: For example, you could 1) improve the programs available at the CPH MPL or 2) improve the current compilation technology. Elaborate one of the possible avenues experimentally. □

Second program: Incremental construction

After the first experiment we already have a pretty good picture were the limits are. In this second experiment, we wanted to create a `cphmpl::intlist` for integers $0, 1, ..., \mathtt{n}-1$ using the compile-time tools only. With the incremental approach such a list can be created by starting with an empty list and appending new elements to the current list one by one. The code below gives the details. The `main` program describes the driver used—we were just interested in the compilation time, no real computation is done.

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

namespace deprecated {

  namespace helper {

    template<typename, int>
    class append;

    template<int... L, int m>
    class append<cphmpl::intlist<L...>, m>
      : public cphmpl::intlist<L..., m> {
    };

    template<std::size_t n>
    class make_indexlist
      : public append<typename make_indexlist<n-1>::self, n-1> {
    };

    template<>
    class make_indexlist<0>
      : public cphmpl::intlist<> {
    };
  }

  template<std::size_t n>
  using make_indexlist ← typename helper::make_indexlist<n>::self;
```

```
29 }
30
31 #include "n.i++" // n comes from here
32
33 int main() {
34   using L ← deprecated::make_indexlist<n>;
35   static_assert(L::front == 0);
36   return 0;
37 }
```

The key trick used is on lines 18 and 28, showing how to convert a derived class into its parent class in the inheritance hierarchy at compile time. This idea is from an Internet posting at Stack Overflow by Xeo and others [18].

The results of the successful test runs are shown in Table 2. The most surprising result was that the compilation failed for `n` ← 1000. The conclusion to be drawn from this is that we cannot recommend the use of the CPH MPL—and wide use of metaprogramming in general—for production code. Hence, we expect that the library will predominantly be used for research and recreational purposes.

Table 2. Performance of the program relying on the incremental-construction paradigm to generate a list of indices from 0 to `n` − 1 at compile time. The compilation times are in seconds. Minus sign − means that the compilation failed.

`n`	Construction	
	time	page faults
10	0.19	9 872
100	9.4	$7 \cdot 10^5$
1000	−	−

Third program: Divide and conquer

The divide-and-conquer paradigm is widely applicable; it can also be used to create a compile-time list of integers $0, 1, ..., \mathtt{n}-1$. Our third program for this task—shown below—is a modified version of the solutions published at Stack Overflow [18]. The original version was posted by Xeo and the beauty of the code was explained to others by Tony D.

```
1  namespace cphmpl {
2
3    namespace helper {
4
5      template<typename, typename>
6      class merge;
7
8      template<int... L, int... Rcal>
9      class merge<cphmpl::intlist<L...>, cphmpl::intlist<Rcal...>>
10       : public cphmpl::intlist<L..., (sizeof...(L) + Rcal)...> {
11     };
```

```
12
13      template<std::size_t n>
14      class make_indexlist
15        : public merge<typename make_indexlist<n / 2>::self,
16                       typename make_indexlist<n - n / 2>::self> {
17      };
18
19      template<>
20      class make_indexlist<0>
21        : public cphmpl::intlist<> {
22      };
23    }
24
25    template<std::size_t n>
26    using make_indexlist ← typename helper::make_indexlist<n>::self;
27  }
```

This solution relies on the pack-expansion facility supported for parameter packs. This is used on line 10 to renumber the indices on the right half `Rcal`.

Because of the renumbering done at each `merge`, the compilation time follows the recurrence equation $T(\mathtt{n}) = T(\mathtt{n}\ /\ 2) + T(\mathtt{n} - \mathtt{n}\ /\ 2) + O(\mathtt{n})$ with the base case $T(0) = O(1)$. By the master theorem (see, e.g. [3, Chapter 4]), the solution of this recurrence is $O(\mathtt{n} \lg \mathtt{n})$. Moreover, the recursion depth is never larger than $\lg \mathtt{n} + O(1)$, so this program should work without increasing the maximum template-instantiation depth set.

In spite of beauty, for $\mathtt{n} \leftarrow 1000$, the outcome was the same as before: internal compiler error. The results of the passed tests are given in Table 3. It is a bit unsatisfactory to conclude that only the straightforward generative approach could handle a list of 1000 integers. The template arguments were generated in a loop. A more direct approach failed. Namely, when a loop variable `i` was used in the template context, the compiler responded:

```
error: the value of 'i' is not usable in a constant expression
```

Exercise 5. Implement a program that 1) carries out the same task as `is_member` for a list of sorted integers in a `cphmpl::intlist` and 2) relies on the divide-and-conquer paradigm. In addition, provide experimental data on its performance. Are the results consistent with your expectations? □

Table 3. Performance of the program relying on the divide-and-conquer paradigm to generate a list of indices from 0 to $\mathtt{n} - 1$ at compile time. The compilation times are in seconds. Minus sign − means that the compilation failed.

n	Construction	
	time	page faults
10	0.12	7 809
100	0.77	$35 \cdot 10^3$
1000	−	−

Exercise 6.** Go to a fantasy land and outline some new language features that would simplify and potentially speed up the metaprograms presented. □

6. Future trends

The class template `std::tuple` is a fixed-size collection of heterogeneous (run-time) values. The class template `cphmpl::typelist` is a tuple without the values, just the types. And the class template `cphmpl::valuelist` is a tuple of (compile-time) values without the types—the types are deduced by the compiler. It might be a good idea to unify the design of these components.

To be able to see, what will happen in the future, we have to go a little bit back in time. As an exercise, I tried to track the origins of `std::tuple`. One of the cornerstones is a technical report by Järvi written in 1999 [11]. After several revisions (see, for instance, [12]), the component was accepted to the C++ standard library in 2011 [7]. So it takes time before an idea gets from a laboratory to a regular programmer.

Recently, in the development of C++, the focus has been on the compile-time facilities. As I have understood, Järvi took some of his inspiration from Haskell. Perhaps, the next step is to take inspiration from TEX and try to improve the preprocessing facilities. Here—as I see it—there are some unused opportunities for real code generation.

When writing the components documented in this paper, there were some moments of irritation. I expect that others have come across the same issues, so there is hope that some of these will be fixed soon.

Restrictions on template parameters: I tried to have separate syntax for compile-time functions `f<...>` and run-time functions `f(...)`. I can clearly see the difficulties to be faced: Because of the restrictions set to the allowable template parameters, not all functions can be written in this form. Why should it be like this? Constant-expression functions solve the problem, but we do not get a clarifying syntax that would separate compile-time computations from run-time computations.

Unified syntax: Consider the specialization of `std::numeric_limits` for a built-in type `T`. Why is `std::numeric_limits<T>::max()` a function and `std::numeric_limits<T>::digits` a variable? Since both are constant expressions, `max` could be made a variable, too. More generally, a compile-time function taking no function arguments can always be refactored to a variable or a variable template. Thus, it would be convenient for a library user, if he or she knew that a read-only member can be accessed without <> if it takes no template arguments—or if these can be deduced—and without () if it takes no function arguments.

Introspection: One reason for the existence of the tools in the headers `<type_traits>`, `<climits>`, and `<limits>` is that built-in types do not know their traits and limits. Is it time to fix this? I would like to write `int::max` instead of `INT_MAX` or `std::numeric_limits<int>::max()`. As long as, built-in types are treated differently from other types, all

generic introspection tools in the header `"cphmpl/functions.h++"` must have separate specializations for built-in types.

Automatic conversions: I would hope that my compiler stopped doing automatic conversions when processing template arguments. For example, it should not be possible to declare a mixed-type `charlist`. Now the following type alias declaration was accepted by my compiler:

```
using mixed ← cphmpl::charlist<'w', 'h', 97UL, 116>;
```

I expected to get a compilation error, but instead I got a list of four characters: `'w'`, `'h'`, `'a'`, `'t'`. I could force the compilation to fail by giving an extra type parameter for the class template `cphmpl::charlist`, but this is not what I want.

Polymorphism: My original intention was to write a single heterogeneous compile-time list that accepts non-types and types. In this attempt, the following three problems were encountered: 1) The predicates and transformers have a different signature for values and types. 2) Some of the functions return a value for value lists and a type for type lists. 3) If a function returns a type, one has to use **using** and, if it returns a value, one has to use **constexpr**. It would be convenient if there was a unified way of performing immutable assignments at compile time. Instead of ←, we could use :← (read: *anchored assignment*) that is sometimes used to denote a definition—a defined entity cannot change. This way, `valuedef` would be consistent with the ancient **typedef**. What is the type of the result of an anchored type assignment?
Because of these complications, I decided to go away from the all-in-one design and kept the two kinds of lists separate. On the one hand, not much extra code is needed to create `charlist`, `intlist`, `longlist` et cetera from `valuelist`. On the other hand, I can reveal that the class template `cphmpl::typelist` is copy-paste code. Starting from `cphmpl::valuelist`, it was necessary to replace **auto** with **typename** in some critical places. A few additional changes were necessary because of the schism between values and types. Finally, the name of `at` was purposely changed to `get`. The hope is that this way we can catch the wrong use of the templates at compile time. To conclude, there is some kind of polymorphism that is not captured properly.

Immutability: An array of `n` characters takes `n`$+O(1)$ bytes of memory. As the performed experiments showed, a corresponding pure compile-time list of characters uses several orders of magnitude more space during compilation. For efficiency, purity is not a good thing. In applications, where the lists can become large, a compile-time array of characters should be used instead. What do you think about a compile-time array of types?

As to the evolution of the CPH MPL, we will add new facilities to the library when we need them. We will deliberately try to avoid developing facilities that no one uses.

Acknowledgements

First, I thank Andreas Milton Maniotis for his initiative to create the CPH MPL branch for our library project (e-mail dated 2013-06-20). Second, I thank Ask Neve Gamby, Torben Mogensen, and the students of my algorithm-engineering course (autumn 2017) for their comments that helped me to shape my ideas on the subject matter.

Software availability

For your convenience, some of the files of the CPH MPL are rendered in the appendix. 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` balls available via the website `http://www.cphstl.dk`.

Appendix: Source code

Copyright notice

Copyright © 2000–2017 by The authors and Performance Engineering Laboratory (University of Copenhagen)

The programs included in the CPH MPL are placed in the public domain. The files may be freely copied and distributed, provided that no changes whatsoever are made. Changes are permissible only if the modified files are given new names, different from the names of existing files in the CPH MPL, and only if the modified files are clearly identified as not being part of the library. Usage of the source code in derived works is otherwise unrestricted.

The authors have tried to produce correct and useful programs, but no warranty of any kind should be assumed.

Release date

2017-11-16

Contents

File	Page
`cphmpl/functions.h++`	29
`cphmpl/lists.h++`	38
`cphmpl/valuelist.h++`	38
`cphmpl/charlist.h++`	44
`cphmpl/intlist.h++`	46
`cphmpl/typelist.h++`	47

cphmpl/functions.h++

```
1  /*
2    Author: Jyrki Katajainen © 2017
3
4    This file contains some useful functions evaluated at compile time.
5  */
6
7  #ifndef _CPHMPL_FUNCTIONS_
8  #define _CPHMPL_FUNCTIONS_
9
10 #include "cphmpl/typelist.h++" // cphmpl::typelist
11 #include <cstddef> // std::size_t
12 #include <limits> // std::numeric_limits for built-in types
13 #include <type_traits> // std::fundamental ...
14
15 namespace cphmpl {
16
17   // is_character
18
19   namespace helper {
20
21     template<typename T>
22     constexpr bool is_character() {
23       using character_types ← cphmpl::typelist<signed char,
       unsigned char, char, wchar_t, char16_t, char32_t>;
24       using U ← typename std::remove_cv<T>::type;
25       return character_types::is_member<U>;
26     }
27   }
28
29   template<typename T>
30   constexpr bool is_character ← helper::is_character<T>();
31
32   // is_integer
33
34   namespace helper {
35
36     template<typename T>
37     concept bool has_static_variable_is_integer ←
38     requires {
39       T::is_integer;
40     };
41
42     template<typename T>
43     class is_integer_selector {
44     public:
45
46       static constexpr bool value ← false;
47     };
48
```

```
49      template<typename T>
50      requires std::is_fundamental_v<T>
51      class is_integer_selector<T> {
52      public:
53
54        static constexpr bool value ← std::numeric_limits<T>::
        is_integer;
55      };
56
57      template<typename T>
58      requires has_static_variable_is_integer<T>
59      class is_integer_selector<T> {
60      public:
61
62        static constexpr bool value ← T::is_integer;
63      };
64    }
65
66    template<typename T>
67    constexpr bool is_integer ← helper::is_integer_selector<T>::
        value;
68
69    // is_signed
70
71    namespace helper {
72
73      template<typename T>
74      concept bool has_static_variable_is_signed ←
75      requires {
76        T::is_signed;
77      };
78
79      template<typename T>
80      class is_signed_selector {
81      public:
82
83        static constexpr bool value ← false;
84      };
85
86      template<typename T>
87      requires std::is_fundamental_v<T>
88      class is_signed_selector<T> {
89      public:
90
91        static constexpr bool value ← std::numeric_limits<T>::
        is_signed;
92      };
93
94      template<typename T>
95      requires has_static_variable_is_signed<T>
96      class is_signed_selector<T> {
97      public:
```

```
98
99        static constexpr bool value ← T::is_signed;
100     };
101   }
102
103   template<typename T>
104   constexpr bool is_signed ← helper::is_signed_selector<T>::value;
105
106   // is_unsigned
107
108   template<typename T>
109   constexpr bool is_unsigned ← not is_signed<T>;
110
111   // is_power_of_two
112
113   namespace helper {
114
115     template<typename T>
116     constexpr bool is_power_of_two(T const& x) {
117       if (x == T(0)) {
118         return false;
119       }
120       T y ← x bitand (x - 1);
121       if (y == T(0)) {
122         return true;
123       }
124       return false;
125     }
126   }
127
128   template<std::size_t number>
129   constexpr bool is_power_of_two ← helper::is_power_of_two(number);
130
131   // has_type_value_type
132
133   template<typename T>
134   concept bool has_type_value_type ←
135   requires {
136     typename T::value_type;
137   };
138
139   // has_type_pointer
140
141   template<typename T>
142   concept bool has_type_pointer ←
143   requires {
144     typename T::pointer;
145   };
146
147   // has_type_iterator
148
149   template<typename T>
```

```
150  concept bool has_type_iterator ←
151  requires {
152    typename T::iterator;
153  };
154
155  // width
156
157  namespace helper {
158
159    template<typename T>
160    concept bool has_static_variable_width ←
161    requires {
162      T::width;
163    };
164
165    template<typename T>
166    class width_selector {
167    private:
168
169      using S ← decltype(sizeof(0));
170      using B ← unsigned char;
171      static constexpr S bits ← std::numeric_limits<B>::digits;
172
173    public:
174
175      static constexpr S value ← bits * sizeof(T);
176    };
177
178    template<typename T>
179    requires std::is_fundamental_v<T>
180    class width_selector<T> {
181    private:
182
183      using S ← decltype(sizeof(0));
184      static constexpr S sign ← std::numeric_limits<T>::is_signed;
185      static constexpr S digits ← std::numeric_limits<T>::digits;
186
187    public:
188
189      static constexpr S value ← sign + digits;
190    };
191
192    template<typename T>
193    requires has_static_variable_width<T>
194    class width_selector<T> {
195    private:
196
197      using S ← decltype(sizeof(0));
198
199    public:
200
201      static constexpr S value ← T::width;
```

```
202     };
203   }
204
205   template<typename T>
206   requires cphmpl::is_integer<T>
207   constexpr auto width ← helper::width_selector<T>::value;
208
209   // all_zeros
210
211   namespace helper {
212
213     template<typename T>
214     requires cphmpl::is_integer<T>
215     constexpr T all_zeros() {
216       return T();
217     }
218   }
219
220   template<typename T>
221   constexpr T all_zeros ← helper::all_zeros<T>();
222
223   namespace helper {
224
225     template<typename T>
226     requires cphmpl::is_integer<T>
227     constexpr T all_ones() {
228       return ~all_zeros<T>();
229     }
230   }
231
232   template<typename T>
233   constexpr T all_zeros ← helper::all_zeros<T>();
234
235   // lg
236
237   namespace helper {
238
239     template<auto n>
240     constexpr auto lg() {
241       using N ← decltype(n);
242       N result ← 0;
243       N i ← n;
244       while (i > 1) {
245         i ← i / 2;
246         ++result;
247       }
248       return result;
249     }
250   }
251
252   template<auto n>
253   constexpr auto lg ← helper::lg<n>();
```

```
254
255   // power
256
257   namespace helper {
258
259     template<std::size_t base, std::size_t exponent>
260     constexpr std::size_t power() {
261       if constexpr (exponent == 0) {
262         return 1;
263       }
264       else {
265         return base * power<base, exponent - 1>();
266       }
267     }
268   }
269
270   template<std::size_t base, std::size_t exponent>
271   constexpr std::size_t power ← helper::power<base, exponent>();
272
273   // power_of_two
274
275   namespace helper {
276
277     template<typename T, std::size_t exponent>
278     requires cphmpl::is_integer<T>
279     constexpr T power_of_two() {
280       return T(1) << exponent;
281     }
282   }
283
284   template<typename T, std::size_t exponent>
285   constexpr T power_of_two ← helper::power_of_two<T, exponent>();
286
287   // some_trailing_ones
288
289   namespace helper {
290
291     template<typename T, std::size_t count>
292     constexpr T some_trailing_ones() {
293       if constexpr (count == 0) {
294         return all_zeros<T>();
295       }
296       else if constexpr (count ≥ cphmpl::width<T>) {
297         return all_ones<T>();
298       }
299       else {
300         return power_of_two<T, count>() - 1;
301       }
302     }
303   }
304
305   template<typename T, std::size_t count>
```

```
306   constexpr T some_trailing_ones ← helper::some_trailing_ones<T,
        count>();
307
308   // some_leading_ones
309
310     namespace helper {
311
312     template<typename T, std::size_t count>
313     constexpr T some_leading_ones() {
314       if constexpr (count == 0) {
315         return all_zeros<T>();
316       }
317       else if constexpr (count ≥ cphmpl::width<T>) {
318         return all_ones<T>();
319       }
320       else {
321   constexpr std::size_t w ← cphmpl::width<T>;
322   constexpr T ones ← power_of_two<T, count>() − 1;
323   constexpr T result ← ones << (w − count);
324         return result;
325       }
326     }
327   }
328
329   template<typename T, std::size_t count>
330   constexpr T some_leading_ones ← helper::some_leading_ones<T,
        count>();
331
332   // is_unsigned
333
334   namespace helper {
335
336     template<typename T, T n> // without T, n will be converted
337     requires cphmpl::is_unsigned<decltype(n)>
338     constexpr auto leading_zeros() {
339       using N ← decltype(n);
340       using S ← decltype(sizeof(0));
341       constexpr S w ← cphmpl::width<N>;
342       if constexpr (n == 0) {
343         return w;
344       }
345       else {
346         return w − (1 + cphmpl::lg<n>);
347       }
348     }
349   }
350
351   template<typename T, T n>
352   static constexpr auto leading_zeros ← helper::leading_zeros<T, n
        >();
353
354   // make_unsigned
```

```
355
356   namespace helper {
357
358     template<typename T>
359     concept bool has_type_companion ←
360     requires {
361       typename T::companion;
362     };
363
364     template<typename T>
365     class make_unsigned_selector {
366     public:
367
368       using type ← T;
369     };
370
371     template<typename T>
372     requires cphmpl::is_signed<T> and std::is_fundamental_v<T>
373     class make_unsigned_selector<T> {
374     private:
375
376       using E ← typename std::remove_reference<T>::type;
377
378     public:
379
380       using type ← typename std::make_unsigned<E>::type;
381     };
382
383     template<typename T>
384     requires cphmpl::is_signed<T> and has_type_companion<T>
385     class make_unsigned_selector<T> {
386     public:
387
388       using type ← typename T::companion;
389     };
390   }
391
392   template<typename T>
393   using make_unsigned ← typename helper::make_unsigned_selector<T>
      ::type;
394
395   // make_signed
396
397   namespace helper {
398
399     template<typename T>
400     class make_signed_selector {
401     public:
402
403       using type ← T;
404     };
405
```

```
406     template<typename T>
407     requires std::is_fundamental_v<T>
408     class make_signed_selector<T> {
409     private:
410
411       using E ← typename std::remove_reference<T>::type;
412
413     public:
414
415       using type ← typename std::make_signed<E>::type;
416     };
417
418     template<typename T>
419     requires has_type_companion<T>
420     class make_signed_selector<T> {
421     public:
422
423       using type ← typename T::companion;
424     };
425   }
426
427   template<typename T>
428   using make_signed ← typename helper::make_signed_selector<T>::
        type;
429
430   // is_safely_convertible
431
432   namespace helper {
433
434     template<typename S, typename T>
435     requires cphmpl::is_integer<S> and cphmpl::is_integer<T>
436     constexpr bool is_safely_convertible() {
437       if constexpr (is_unsigned<S> and is_signed<T>) {
438         return width<S> + 1 ≤ width<T>;
439       }
440       else if constexpr (is_signed<S> and is_unsigned<T>) {
441         return false;
442       }
443       else {
444         return width<S> ≤ width<T>;
445       }
446     }
447   }
448
449   template<typename S, typename T>
450   constexpr bool is_safely_convertible ← helper::
        is_safely_convertible<S, T>();
451
452   // width comparisons
453
454   template<typename S, typename T>
455   constexpr bool wide_same ← (width<S> == width<T>);
```

```
456
457   template<typename S, typename T>
458   constexpr bool narrower ← (width<S> < width<T>);
459
460   template<typename S, typename T>
461   constexpr bool wider ← (width<S> > width<T>);
462
463   template<typename S, typename T>
464   constexpr bool narrower_equal ← (width<S> ≤ width<T>);
465
466   template<typename S, typename T>
467   constexpr bool wider_equal ← (width<S> ≥ width<T>);
468 }
469
470 #endif
```

cphmpl/lists.h++

```
1  /*
2    Author: Jyrki Katajainen © 2017
3  */
4
5  #ifndef _CPHMPL_LISTS_
6  #define _CPHMPL_LISTS_
7
8  #include "cphmpl/charlist.h++"
9  #include "cphmpl/intlist.h++"
10 #include "cphmpl/typelist.h++"
11 #include "cphmpl/valuelist.h++"
12
13 #endif
```

cphmpl/valuelist.h++

```
1  /*
2    Author: Jyrki Katajainen © 2017
3  */
4
5  #ifndef _CPHMPL_VALUELIST_
6  #define _CPHMPL_VALUELIST_
7
8  #include <type_traits> // std::conditional
9
10 namespace cphmpl {
11
12   template<auto... Ecal>
13   class valuelist {
14   private:
15
16     // forward declarations
```

```
17
18      template<typename, auto>
19      class at_helper;
20
21      template<typename>
22      class pop_front_helper;
23
24      template<typename, typename>
25      class pop_back_helper;
26
27      template<typename, typename, template<auto> class>
28      class filter_helper;
29
30      template<typename, typename, template<auto> class>
31      class transform_helper;
32
33      template<typename, auto, template<auto> class>
34      class find_if_helper;
35
36      template<typename, auto, template<auto> class>
37      class count_if_helper;
38
39      template<typename, auto>
40      class is_member_helper;
41
42    public:
43
44      using self ← valuelist<Ecal...>;
45      using size_type ← decltype(sizeof(0));
46      using cursor ← decltype(sizeof(0));
47
48      static constexpr size_type length ← sizeof...(Ecal);
49      static constexpr bool is_empty ← false;
50      static constexpr auto front ← at_helper<self, 0>::value;
51      static constexpr auto back ← at_helper<self, length − 1>::
        value;
52
53      template<cursor index>
54      static constexpr auto at ← at_helper<self, index>::value;
55
56      template<auto element>
57      using push_front ← valuelist<element, Ecal...>;
58
59      template<auto element>
60      using push_back ← valuelist<Ecal..., element>;
61
62      using pop_front ← typename pop_front_helper<self>::type;
63
64      using pop_back ← typename pop_back_helper<self, valuelist<>>
        ::type;
65
66      template<template<auto> class P>
```

```
67      using filter ← typename filter_helper<self, valuelist<>, P>::
        type;
68
69      template<template<auto> class F>
70      using transform ← typename transform_helper<self, valuelist<>,
         F>::type;
71
72      template<template<auto> class P>
73      static constexpr cursor find_if ← find_if_helper<self, length,
        P>::value;
74
75      template<template<auto> class P>
76      static constexpr cursor count_if ← count_if_helper<self, 0, P>
        ::value;
77
78      template<auto element>
79      static constexpr bool is_member ← is_member_helper<self,
        element>::value;
80
81    private:
82
83      // at: 0–based indexing
84
85      template<typename L, auto index>
86      class at_helper {
87      public:
88        // index out of bounds
89      };
90
91      template<auto head, auto... T>
92      class at_helper<valuelist<head, T...>, 0> {
93      public:
94
95        static constexpr auto value ← head;
96      };
97
98      template<auto head, auto... T, auto i>
99      class at_helper<valuelist<head, T...>, i> {
100     public:
101
102       static constexpr auto value ← at_helper<valuelist<T...>, i
        − 1>::value;
103     };
104
105     // pop_front
106
107     template<typename>
108     class pop_front_helper {
109     public:
110
111       using type ← valuelist<>;
112     };
```

```
113
114     template<auto head, auto... T>
115     class pop_front_helper<valuelist<head, T...>> {
116     public:
117
118       using type ← valuelist<T...>;
119     };
120
121     // pop_back
122
123     template<typename, typename>
124     class pop_back_helper {
125     public:
126
127       using type ← valuelist<>;
128     };
129
130     template<auto last, auto... X>
131     class pop_back_helper<valuelist<last>, valuelist<X...>> {
132     public:
133
134       using type ← valuelist<X...>;
135     };
136
137     template<auto head, auto... T, auto... X>
138     class pop_back_helper<valuelist<head, T...>, valuelist<X...
        >> {
139     public:
140
141       using type ← typename pop_back_helper<valuelist<T...>,
        valuelist<X..., head>>::type;
142     };
143
144     // filter: P is a unary predicate
145
146     template<typename, typename, template<auto> class>
147     class filter_helper;
148
149     template<typename L, template<auto> class P>
150     class filter_helper<valuelist<>, L, P> {
151     public:
152
153       using type ← L;
154     };
155
156     template<auto head, auto... T, auto... X, template<auto> class
         P>
157     class filter_helper<valuelist<head, T...>, valuelist<X...>,
        P> {
158     public:
159
160       using type ← typename std::conditional<(P<head>::value),
```

```
161       typename filter_helper<valuelist<T...>, valuelist<X..., head>, P>::type,
162       typename filter_helper<valuelist<T...>, valuelist<X...>, P>::type
163       >::type;
164   };
165
166   // transform: F is a transformer
167
168   template<typename, typename, template<auto> class>
169   class transform_helper;
170
171   template<auto... X, template<auto> class F>
172   class transform_helper<valuelist<>, valuelist<X...>, F> {
173   public:
174
175     using type ← valuelist<X...>;
176   };
177
178   template<auto head, auto... T, auto... X, template<auto> class F>
179   class transform_helper<valuelist<head, T...>, valuelist<X...>, F> {
180   public:
181
182     using type ← typename transform_helper<valuelist<T...>, valuelist<F<head>::value, X...>, F>::type;
183   };
184
185   // find_if: P is a unary predicate
186
187   template<typename, auto, template<auto> class>
188   class find_if_helper;
189
190   template<auto n, template<auto> class P>
191   class find_if_helper<valuelist<>, n, P> {
192   public:
193
194     static constexpr cursor value ← n;
195   };
196
197   template<auto head, auto... T, auto n, template<auto> class P>
198   class find_if_helper<valuelist<head, T...>, n, P> {
199   public:
200
201     static constexpr cursor value ← (P<head>::value) ?
202       n − valuelist<head, T...>::length :
203       find_if_helper<valuelist<T...>, n, P>::value;
204   };
205
206   // count_if: P is a unary predicate
```

```
207
208   template<typename, auto, template<auto> class>
209   class count_if_helper;
210
211   template<auto n, template<auto> class P>
212   class count_if_helper<valuelist<>, n, P> {
213   public:
214
215     static constexpr size_type value ← n;
216   };
217
218   template<auto head, auto... T, auto n, template<auto> class P
      >
219   class count_if_helper<valuelist<head, T...>, n, P> {
220   public:
221
222     static constexpr size_type value ← (P<head>::value) ?
223       count_if_helper<valuelist<T...>, n + 1, P>::value :
224       count_if_helper<valuelist<T...>, n, P>::value;
225   };
226
227   // is_member
228
229   template<typename, auto>
230   class is_member_helper;
231
232   template<auto element>
233   class is_member_helper<valuelist<>, element> {
234   public:
235
236     static constexpr bool value ← false;
237   };
238
239   template<auto head, auto... T, auto element>
240   class is_member_helper<valuelist<head, T...>, element> {
241   public:
242
243     static constexpr bool value ← (head == element) ?
244       true : is_member_helper<valuelist<T...>, element>::value;
245   };
246 };
247
248 template<>
249 class valuelist<> {
250 public:
251
252   using self ← valuelist<>;
253   using size_type ← decltype(sizeof(0));
254   using cursor ← decltype(sizeof(0));
255
256   static constexpr size_type length ← 0;
257   static constexpr bool is_empty ← true;
```

```
258
259     // static constexpr auto front ← undefined;
260     // static constexpr auto back ← undefined;
261
262     // template<cursor index>
263     // static constexpr auto at ← undefined;
264
265     template<auto element>
266     using push_front ← valuelist<element>;
267
268     template<auto element>
269     using push_back ← valuelist<element>;
270
271     // using pop_front ← undefined;
272     // using pop_back ← undefined;
273
274     template<template<auto> class P>
275     using filter ← self;
276
277     template<template<auto> class F>
278     using transform ← self;
279
280     template<template<auto> class P>
281     static constexpr cursor find_if ← 0;
282
283     template<template<auto> class P>
284     static constexpr cursor count_if ← 0;
285
286     template<auto element>
287     static constexpr bool is_member ← false;
288   };
289 }
290
291 #endif
```

cphmpl/charlist.h++

```
1  /*
2    Author: Jyrki Katajainen © 2017
3  */
4
5  #ifndef _CPHMPL_CHARLIST_
6  #define _CPHMPL_CHARLIST_
7
8  #include "cphmpl/valuelist.h++"
9  #include <type_traits>
10
11 namespace cphmpl {
12
13   template<char... Ecal>
```

```
14  requires std::conjunction_v<std::is_same<char, decltype(Ecal)>..
      .>
15  class charlist
16    : public valuelist<Ecal...> {
17  public:
18
19    using self ← charlist<Ecal...>;
20    using value_type ← char;
21  };
22
23  // specific char tools
24
25  using digits ← charlist<'0','1','2','3','4','5','6','7','8','9'>
      ;
26
27  using lower_case_letters ← charlist<'a','b','c','d','e','f','g','
      h','i','j','k','l','m','n','o','p','q','r','s','t','u','v','w','x
      ','y','z'>;
28
29  using upper_case_letters ← charlist<'A','B','C','D','E','F','G','
      H','I','J','K','L','M','N','O','P','Q','R','S','T','U','V','W','X
      ','Y','Z'>;
30
31  // ' '   (0x20)  space (SPC)
32  // '\t'  (0x09)  horizontal tab (TAB)
33  // '\n'  (0x0a)  newline (LF)
34  // '\v'  (0x0b)  vertical tab (VT)
35  // '\f'  (0x0c)  form feed (FF)
36  // '\r'  (0x0d)  carriage return (CR)
37
38  using space ← charlist<' ','\t','\n','\v','\f','\r'>;
39
40  using comma ← charlist<','>;
41
42  template<char symbol>
43  class is_digit {
44  public:
45
46    static constexpr bool value ← digits::is_member<symbol>;
47  };
48
49  template<char symbol>
50  class is_lower_case_letter {
51  public:
52
53    static constexpr bool value ← lower_case_letters::is_member<
      symbol>;
54  };
55
56  template<char symbol>
57  class is_upper_case_letter {
58  public:
```

```
59
60      static constexpr bool value ← upper_case_letters::is_member<
        symbol>;
61    };
62
63    template<char symbol>
64    class is_letter {
65    public:
66
67      static constexpr bool value ← is_lower_case_letter<symbol>::
        value or is_upper_case_letter<symbol>::value;
68    };
69
70    template<char symbol>
71    class is_space {
72    public:
73
74      static constexpr bool value ← space::is_member<symbol>;
75    };
76
77    template<char symbol>
78    class is_comma {
79    public:
80
81      static constexpr bool value ← comma::is_member<symbol>;
82    };
83  }
84
85  #endif
```

cphmpl/intlist.h++

```
1   /*
2     Author: Jyrki Katajainen © 2017
3   */
4
5   #ifndef _CPHMPL_INTLIST_
6   #define _CPHMPL_INTLIST_
7
8   #include "cphmpl/valuelist.h++"
9   #include <cstddef> // std::size_t
10
11  namespace cphmpl {
12
13    template<int... Ecal>
14    requires std::conjunction_v<std::is_same<int, decltype(Ecal)>...
        >
15    class intlist
16      : public valuelist<Ecal...> {
17    public:
18
```

```
19      using self ← intlist<Ecal...>;
20      using value_type ← int;
21    };
22
23    namespace helper {
24
25      template<typename, typename>
26      class merge;
27
28      template<int... L, int... Rcal>
29      class merge<cphmpl::intlist<L...>, cphmpl::intlist<Rcal...>>
30        : public cphmpl::intlist<L..., (sizeof...(L) + Rcal)...> {
31      };
32
33      template<std::size_t n>
34      class make_indexlist
35        : public merge<typename make_indexlist<n / 2>::self,
36           typename make_indexlist<n - n / 2>::self> {
37      };
38
39      template<>
40      class make_indexlist<0>
41        : public cphmpl::intlist<> {
42      };
43    }
44
45    template<int n>
46    using make_indexlist ← typename helper::make_indexlist<n>::self;
47  }
48
49  #endif
```

cphmpl/typelist.h++

```
1   /*
2     Author: Jyrki Katajainen © 2017
3   */
4
5   #ifndef _CPHMPL_TYPELIST_
6   #define _CPHMPL_TYPELIST_
7
8   #include <type_traits> // std::conditional std::is_same
9
10  namespace cphmpl {
11
12    template<typename... P>
13    class typelist {
14    private:
15
16      // forward declarations
17
```

```
18      template<typename, auto>
19      class get_helper;
20
21      template<typename>
22      class pop_front_helper;
23
24      template<typename, typename>
25      class pop_back_helper;
26
27      template<typename, typename, template<typename> class>
28      class filter_helper;
29
30      template<typename, typename, template<typename> class>
31      class transform_helper;
32
33      template<typename, auto, template<typename> class>
34      class find_if_helper;
35
36      template<typename, auto, template<typename> class>
37      class count_if_helper;
38
39      template<typename, typename>
40      class is_member_helper;
41
42    public:
43
44      using self ← typelist<P...>;
45      using size_type ← decltype(sizeof(0));
46      using cursor ← decltype(sizeof(0));
47
48      static constexpr size_type length ← sizeof...(P);
49      static constexpr bool is_empty ← false;
50      using front ← typename get_helper<self, 0>::type;
51      using back ← typename get_helper<self, length − 1>::type;
52
53      template<cursor index>
54      using get ← typename get_helper<self, index>::type;
55
56      template<typename T>
57      using push_front ← typelist<T, P...>;
58
59      template<typename T>
60      using push_back ← typelist<P..., T>;
61
62      using pop_front ← typename pop_front_helper<self>::type;
63
64      using pop_back ← typename pop_back_helper<self, typelist<>>::
        type;
65
66      template<template<typename> class P>
67      using filter ← typename filter_helper<self, typelist<>, P>::
        type;
```

```
68
69      template<template<typename> class F>
70      using transform ← typename transform_helper<self, typelist<>,
        F>::type;
71
72      template<template<typename> class P>
73      static constexpr cursor find_if ← find_if_helper<self, length,
        P>::value;
74
75      template<template<typename> class P>
76      static constexpr cursor count_if ← count_if_helper<self, 0, P>
        ::value;
77
78      template<typename T>
79      static constexpr bool is_member ← is_member_helper<self, T>::
        value;
80
81    private:
82
83      // at: 0–based indexing
84
85      template<typename, auto>
86      class get_helper {
87        // using type ← undefined;
88      };
89
90      template<typename H, typename... T>
91      class get_helper<typelist<H, T...>, 0> {
92      public:
93
94        using type ← H;
95      };
96
97      template<typename H, typename... T, auto index>
98      class get_helper<typelist<H, T...>, index> {
99      public:
100
101       using type ← typename get_helper<typelist<T...>, index − 1
        >::type;
102     };
103
104     // pop_front
105
106     template<typename>
107     class pop_front_helper {
108     public:
109
110       using type ← typelist<>;
111     };
112
113     template<typename H, typename... T>
114     class pop_front_helper<typelist<H, T...>> {
```

```
115     public:
116
117       using type ← typelist<T...>;
118     };
119
120     // pop_back
121
122     template<typename, typename>
123     class pop_back_helper {
124     public:
125
126       using type ← typelist<>;
127     };
128
129     template<typename H, typename... X>
130     class pop_back_helper<typelist<H>, typelist<X...>> {
131     public:
132
133       using type ← typelist<X...>;
134     };
135
136     template<typename H, typename... T, typename... X>
137     class pop_back_helper<typelist<H, T...>, typelist<X...>> {
138     public:
139
140       using type ← typename pop_back_helper<typelist<T...>,
        typelist<X..., H>>::type;
141     };
142
143     // filter: P is a unary predicate
144
145     template<typename, typename, template<typename> class>
146     class filter_helper;
147
148     template<typename L, template<typename> class P>
149     class filter_helper<typelist<>, L, P> {
150     public:
151
152       using type ← L;
153     };
154
155     template<typename H, typename... T, typename... X, template<
        typename> class P>
156     class filter_helper<typelist<H, T...>, typelist<X...>, P> {
157     public:
158
159       using type ← typename std::conditional<(P<H>::value),
160       typename filter_helper<typelist<T...>, typelist<X..., H>,
        P>::type,
161       typename filter_helper<typelist<T...>, typelist<X...>, P>
        ::type
162       >::type;
```

```
163     };
164
165     // transform: F is a transformer
166
167     template<typename, typename, template<typename> class>
168     class transform_helper;
169
170     template<typename... X, template<typename> class F>
171     class transform_helper<typelist<>, typelist<X...>, F> {
172     public:
173
174       using type ← typelist<X...>;
175     };
176
177     template<typename H, typename... T, typename... X, template<
        typename> class F>
178     class transform_helper<typelist<H, T...>, typelist<X...>, F
        > {
179     public:
180
181       using type ← typename transform_helper<typelist<T...>,
        typelist<typename F<H>::type, X...>, F>::type;
182     };
183
184     // find_if: P is a unary predicate
185
186     template<typename, auto, template<typename> class>
187     class find_if_helper;
188
189     template<auto n, template<typename> class P>
190     class find_if_helper<typelist<>, n, P> {
191     public:
192
193       static constexpr cursor value ← n;
194     };
195
196     template<typename H, typename... T, auto n, template<typename>
         class P>
197     class find_if_helper<typelist<H, T...>, n, P> {
198     public:
199
200       static constexpr cursor value ← (P<H>::value) ?
201         n - typelist<H, T...>::length :
202         find_if_helper<typelist<T...>, n, P>::value;
203     };
204
205     // count_if: P is a unary predicate
206
207     template<typename, auto, template<typename> class>
208     class count_if_helper;
209
210     template<auto n, template<typename> class P>
```

```
211     class count_if_helper<typelist<>, n, P> {
212     public:
213
214       static constexpr size_type value ← n;
215     };
216
217     template<typename H, typename... T, auto n, template<typename>
         class P>
218     class count_if_helper<typelist<H, T...>, n, P> {
219     public:
220
221       static constexpr size_type value ← (P<H>::value) ?
222         count_if_helper<typelist<T...>, n + 1, P>::value :
223         count_if_helper<typelist<T...>, n, P>::value;
224     };
225
226     // is_member
227
228     template<typename, typename>
229     class is_member_helper;
230
231     template<typename T>
232     class is_member_helper<typelist<>, T> {
233     public:
234
235       static constexpr bool value ← false;
236     };
237
238     template<typename H, typename... T, typename T>
239     class is_member_helper<typelist<H, T...>, T> {
240     public:
241
242       static constexpr bool value ← (std::is_same<H, T>::value) ?
243         true : is_member_helper<typelist<T...>, T>::value;
244     };
245   };
246
247   template<>
248   class typelist<> {
249   public:
250
251     using self ← typelist<>;
252     using size_type ← decltype(sizeof(0));
253     using cursor ← decltype(sizeof(0));
254
255     static constexpr size_type length ← 0;
256     static constexpr bool is_empty ← true;
257
258     // using front ← undefined;
259     // using back ← undefined;
260
261     // template<cursor index>
```

```
262     // using get ← undefined;
263
264     template<typename T>
265     using push_front ← typelist<T>;
266
267     template<typename T>
268     using push_back ← typelist<T>;
269
270     // using pop_front ← undefined;
271     // using pop_back ← undefined;
272
273     template<template<typename> class P>
274     using filter ← self;
275
276     template<template<typename> class F>
277     using transform ← self;
278
279     template<template<typename> class P>
280     static constexpr cursor find_if ← 0;
281
282     template<template<typename> class P>
283     static constexpr cursor count_if ← 0;
284
285     template<typename T>
286     static constexpr bool is_member ← false;
287   };
288 }
289
290 #endif
```

References

[1] D. Abrahams and A. Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*, Pearson Education, Inc. (2005).

[2] A. Alexandrescu, *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley (2001).

[3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd edition, The MIT Press (2009).

[4] cppreference.com, Constant expressions, C++ documentation (2016). Retrieved from `http://en.cppreference.com/w/cpp/language/constant_expression`.

[5] cppreference.com, Constraints and concepts, C++ documentation (2017). Retrieved from `http://en.cppreference.com/w/cpp/language/constraints`.

[6] cppreference.com, Parameter pack, C++ documentation (2017). Retrieved from `http://en.cppreference.com/w/cpp/language/parameter_pack`.

[7] cppreference.com, `std::tuple`, C++ documentation (2017). Retrieved from `http://en.cppreference.com/w/cpp/utility/tuple`.

[8] cppreference.com, Template parameters and template arguments, C++ documentation (2017). Retrieved from `http://en.cppreference.com/w/cpp/language/template_parameters`.

[9] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley (2000).

[10] P. Dimov, Simple C++11 metaprogramming—With variadic templates, parameter packs and template aliases, Blog post (2015). Retrieved from `http://www.pdimov.com/cpp2/simple_cxx11_metaprogramming.html`.

[11] J. Järvi, Tuples and multiple return values in C++, TUCS technical report **249**, Turku Centre for Computer Science (1999).

[12] J. Järvi, The Boost tuple library, Boost documentation (2001). Retrieved from `http://www.boost.org/doc/libs/1_61_0/libs/tuple/doc/tuple_users_guide.html`.

[13] J. Katajainen, Algorithm engineering, Final test, Department of Computer Science, University of Copenhagen (2017).

[14] J. Katajainen, Width-specific templates $\mathbb{N}$<b> and $\mathbb{Z}$<b> for integer types in C++: Going beyond C, CPH STL report **2017-1**, Department of Computer Science, University of Copenhagen (2017).

[15] S. McConnell, *Code Complete*, 2nd edition, Microsoft Press (2004).

[16] E. Niebler, Tiny metaprogramming library, Blog post (2014). Retrieved from `http://ericniebler.com/2014/11/13/tiny-metaprogramming-library/`.

[17] S. Schurr, New tools for class and library authors, Presentation slides (2012). Retrieved from `https://github.com/boostcon/cppnow_presentations_2012/blob/master/wed/schurr_cpp11_tools_for_class_authors.pdf`.

[18] Stack Exchange, Inc., Implementation C++14 `make_integer_sequence`, Questions & answers (2013). Retrieved from `https://stackoverflow.com/questions/17424477/implementation-c14-make-integer-sequence`.

[19] Stack Exchange, Inc., Why isn't the compiler evaluating a constexpr function during compilation when every information is given?, Questions & answers (2015). Retrieved from `https://stackoverflow.com/questions/32329451/why-isnt-the-compiler-evaluating-a-constexpr-function-during-compilation-when-e`.

[20] B. Stroustrup, *The C++ Programming Language*, 4th edition, Pearson Education, Inc. (2013).

[21] D. Vandevoorde and N. M. Josuttis, *C++ Templates: The Complete Guide*, Pearson Education, Inc. (2003).

[22] Wikimedia Foundation, Inc., Introspection, Wikipedia article (2017). Retrieved from `https://en.wikipedia.org/wiki/Introspection`.

[23] S. Zubkov, What is the correct and efficient way to define string constants in modern C++?, Questions & answers (2016). Retrieved from `https://www.quora.com/What-is-the-correct-and-efficient-way-to-define-string-constants-in-Modern-C++`.