Documenting the CPH MPL—release 2—by code

Jyrki Katajainen

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

Abstract. In C++20, the definitions of function templates and class templates can be supplemented by a set of constraints that template arguments have to fulfil before the templates will be instantiated. To specify such constraints, we need a rich set of type predicates, type attributes, and type functions that can be evaluated at compile time. Such computations are called metaprograms; they are executed by the compiler during the compilation when generating the run-time code.

In this document, we publish the source code of the CPH MPL, Copenhagen metaprogramming library (release 2), which provides a wide variety of metaprograms. We had a strong urge to release this package because

- we find the use of the tools convenient; maybe others also want to use them;
- the tools must be reviewed and used by others to get rid of some hard-to-find errors that are unavoidable for a package of this size; thus, any constructive user feedback would be valuable for us;
- the tools are used in some of our other packages, so the users of these packages will need this package, too;
- the tools have a clear developmental value; they show some shortcomings in the facilities available at the C++ standard library; possibly this will lead to improvements there as well;
- the tools have some pedagogical value; by reading the source code it should be easier for others to write their own metaprograms.

The CPH MPL was developed in parallel with the CPH STL class templates designed for the manipulation of multiple-precision integers. Basically, only tools that were needed by this application were programmed. The reader should have this limitation in mind.

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

1. Introduction

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. However, the variadic templates and constant expressions added to the language have moved metaprogramming closer to the imperative world. It is no more necessary to use heavy template-metaprogramming recursion, when a simple switch statement, or a loop, is sufficient.

In our research group, we took the first steps to create a usable toolbox for metaprogramming in 2013. However, the tools were first needed in 2017 in the development of the class templates designed for the manipulation of multiple-precision integers [4]. The released metaprogramming package [2] was named the CPH MPL—Copenhagen metaprogramming library.

In the second release of the CPH MPL, discussed in this write-up, we have made a complete rewrite of the type functions—now we write them procedurally with a few exceptions. Our way of writing the type functions is explained in a separate paper [3]. The purpose of this document is to serve as a reference for the users of this metaprogramming package. After changing the writing style of the type functions, we hope that the code is self-documenting.

The utilities available at the CPH MPL are packaged in three main files:

`cphmpl/lists.h++`. A *compile-time list* is a heterogeneous data structure which keeps its data in the template parameters—it does not store any non-static data. The lists appear in 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 integers (`cphmpl::intlist`) and a list of characters (`cphmpl::charlist`).

`cphmpl/functions.h++`. A *type function* is a mapping from a type to a value or another type. In the <`type_traits`> header, the suffix `_v` is added to the name of a type function, if it yields a value, and the suffix `_t`, if it yields a type. We instead use the following naming convention:

Classification: type → **bool**; name begins with `is_`
Example: `cphmpl::is_iterator`

Introspection: type → **bool**; the name begins with `has_`
Example: `cphmpl::traits::has_type_member_companion`

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

Property: type → integer; the name describes the computed property
Example: `cphmpl::width` computes the bit width of a bounded type

Generation: type → value; the name describes the generated value
Example: `cphmpl::max` computes the maximum value of a bounded arithmetic type

Association: type → type; the name describes the association
Example: `cphmpl::iterator`

Construction: type → type; the name describes the created type
Example: `cphmpl::make_unsigned`

`cphmpl/concepts.h++`. A *concept* is a Boolean predicate that can be evaluated at compile time. As of this writing, the GNU compiler supported the **requires** clause which could be used to check that the specified requirements are fulfilled by the template arguments before successfully instantiating a template. This enabled constraint-based function overloading and constraint-based partial specialization. However, the compiler was not shipped with the <`concepts`> header. Therefore, I looked at the draft of the C++ standard [October 2019] and implemented the defined concepts using the above-mentioned naming convention. Even some of the basic concepts have been changed since C++17, so, for example, there are two sets of iterator concepts.
Example: `cpp17::is_forward_iterator` and `cpp20::is_forward_iterator`.

In the first part of this document (Sections 2–5), an introduction to the utilities available is given. In Section 2, the implemented compile-time classes and their member functions are listed. This part of the CPH MPL has remained stable since the first release, but naturally the found errors have been corrected. The presentation is supplemented by a simple example showing how the lists can be used. In Section 3, a full reference of the type functions is given. This part of the CPH MPL has been completely rewritten. Most type functions are simple: they are constant-expression functions that consist of a switch statement and possibly use recursion. The implemented type functions are classified in accordance with the above-mentioned categories. Also, some statistics is provided on the frequency of use of different type functions in the main file of the CPH STL integer package. Of course, this application is biased towards the type functions designed for integers. Still, it gives some interesting information which type functions may be useful for other users as well. In Section 4, the implemented standard concepts are briefly discussed. Here the main problem was that the compilers were not fully standard-compliant—they could not be since the standard was not yet ratified [April 2020]. There was one specific type function that could not be handled by the GNU compiler, but this deficiency could be easily remedied. Finally, in Section 5, the report is concluded with a discussion on the future of the CPH MPL.

In the second part of the document (Appendix A), the source code of the full package is listed in transliterated form. Ultimately, the code is the most accurate way of documenting a software product. If you have any proposals how to improve the readability of the code—and the package in general, do not hesitate to contact the author.

2. Compile-time lists

The compile-time lists support array operations, but due to inefficiency these collections are called lists. This inefficiency bursts to the surface, for example, when appending a new element to the end of a list—the result is a new list (type) where only one element is different from the elements in the

```
namespace cphmpl {

  template<auto... Vcal>
  class valuelist {
  public:

    using self = cphmpl::valuelist<Vcal...>;
    using size_type = std::size_t;

    static constexpr std::size_t length = implementation-defined;
    static constexpr bool is_empty = implementation-defined;
    static constexpr auto front = implementation-defined;
    static constexpr auto back = implementation-defined;

    template<std::size_t index>
    static constexpr auto value = implementation-defined;

    template<auto element>
    using push_front = valuelist<element, Vcal...>;

    template<auto element>
    using push_back = valuelist<Vcal..., element>;

    using pop_front = implementation-defined;

    using pop_back = implementation-defined;

    template<template<auto> class P>
    using filter = implementation-defined;

    template<template<auto> class F>
    using transform = valuelist<F<Vcal>::value...>;

    template<template<auto> class P>
    static constexpr std::size_t find_if = implementation-defined;

    template<template<auto> class P>
    static constexpr std::size_t count_if = implementation-defined;

    template<auto element>
    static constexpr bool is_member = implementation-defined;
  };
}
```

Figure 1. The public members of `cphmpl::valuelist`; most implementation details have been omitted. On the basis of the names, it should be clear what the operations do. The class template `P` is a unary predicate (i.e. a function that maps an element to `bool`) yielding its return value via the variable member `value`. The class template `F` is a transformer (i.e. a function that maps an element to some other element) yielding its result in the same way. **Source:** file `cphmpl/valuelist.h++`

```
namespace cphmpl {

  template<typename... T>
  class typelist {
  public:

    using self = cphmpl::typelist<T...>;
    using size_type = std::size_t;

    static constexpr std::size_t length = implementation-defined;
    static constexpr bool is_empty = implementation-defined;
    using front = implementation-defined;
    using back = implementation-defined;

    template<std::size_t index>
    using type = implementation-defined;

    template<typename T>
    using push_front = typelist<T, T...>;

    template<typename T>
    using push_back = typelist<T..., T>;

    using pop_front = implementation-defined;

    using pop_back = implementation-defined;

    template<template<typename> class P>
    using filter = implementation-defined;

    template<template<typename> class F>
    using transform = typelist<typename F<T>::type...>;

    template<template<typename> class P>
    static constexpr std::size_t find_if = implementation-defined;

    template<template<typename> class P>
    static constexpr std::size_t count_if = implementation-defined;

    template<typename T>
    static constexpr bool is_member = implementation-defined;
  };
}
```

Figure 2. The public members of `cphmpl::typelist`; most implementation details have been omitted. Basically, the operations are the same as those provided for `cphmpl::valuelist` and they have the same meaning. The class template `P` is a unary predicate (i.e. a function that maps a type to `bool`) yielding its return value via the variable member `value`. The class template `F` is a transformer (i.e. a function that maps a type to some other type) yielding its result via the type member `type`. **Source:** file `cphmpl/typelist.h++`

original. If done naively, this may mean a huge amount of copying.

The forward declarations of the class templates `cphmpl::valuelist` and `cphmpl::typelist` are given below. Synopses of their public members are shown in Figures 1 and 2, respectively. In the transliterated code, we use calligraphic letters for parameter packs.

```
template<auto... Vcal>
class valuelist;

template<typename... 𝒯>
class typelist;
```

The definitions of the class templates `cphmpl::charlist` and `cphmpl::intlist` are given in their full form in Figure 3. They just inherit all the public members from `cphmpl::valuelist`.

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

  using self = cphmpl::charlist<𝒞...>;
  using value_type = char;
};

template<int... ℐ>
requires std::conjunction_v<std::is_same<int, decltype(ℐ)>...>
class intlist
  : public cphmpl::valuelist<ℐ...> {
public:

  using self = cphmpl::intlist<ℐ...>;
  using value_type = int;
};
```

Figure 3. Definitions of the class templates `cphmpl::charlist` and `cphmpl::intlist`. **Source:** files `cphmpl/charlist.h++` and `cphmpl/intlist.h++`

In addition to the functions provided inside the compile-time classes, in the file `cphmpl/lists.h++`, there are some generic predicates and functions that can be used when operating with compile-time lists (see Figure 4). Let `T` be the list under inspection and let `I` be an index referring to an element in the template parameter list. The functions `cphmpl::size<T>`, `cphmpl::get_type<T, I>`, and `cphmpl::get_value<T, I>` work for the types defined here, but also for other similar types like `std::tuple` and `std::pair`. The first two of these functions generalize the functions `std::tuple_size<T>` giving the size of tuple `T` and `std::tuple_element<T, I>` retrieving the type of the `I`'th element of tuple `T`.

Type classification: `cphmpl::is_valuelist<T>` → **bool**
Effect: Return **true** if `T` is `cphmpl::valuelist<...>` that keeps a collection of non-types in its template parameter list.

Type classification: `cphmpl::specifies_valuelist<T>` → **bool**
Effect: Return **true** if `T` is `cphmpl::valuelist<...>` or a class type that keeps a collection of non-types in its template parameter list.

Type classification: `cphmpl::is_typelist<T>` → **bool**
Effect: Return **true** if `T` is `cphmpl::typelist<...>` that keeps a collection of types in its template parameter list.

Type classification: `cphmpl::specifies_typelist<T>` → **bool**
Effect: Return **true** if `T` is a tuple, a typelist, or any other class type that keeps a collection of types in its template parameter list.

Type property: `cphmpl::size<T>` → **unsigned** integer
Requires: `cphmpl::specifies_valuelist<T>` **or** `cphmpl::specifies_typelist<T>`
Effect: Return the number of the template parameters taken by type `T`. The parameter list must contain a set of types or a set of non-types, but not a mixture of them.

Type association: `cphmpl::get_type<T, I>` → type
Requires: `cphmpl::specifies_valuelist<T>` **or** `cphmpl::specifies_typelist<T>`
Effect: Retrieve the type of the `I`'th element in the template parameter list of type `T`.

Value association: `cphmpl::get_value<T, I>` → constant
Requires: `cphmpl::specifies_valuelist<T>`
Effect: Retrieve the `I`'th element in the valuelist `T`.

Figure 4. Generic compile-time predicates and functions designed for the class templates `cphmpl::valuelist`, `cphmpl::typelist`, and other similar constructs. **Source:** file `cphmpl/lists.h++`

The full definitions of the compile-time classes and the generic compile-time functions can be found in Appendix A. In the implementation of the generic functions we have handled the two cases separately: when the template parameters are types and when they are non-types, but not a mixture of them. This is the reason why most of the other functions are defined inside the classes. This makes the code somewhat easier to read, but the code written in the template-metaprogramming style can be tedious to read and it can hide some subtle errors.

```
#include "cphmpl/lists.h++"
#include <cstdlib> // std::size_t std::byte
#include <iostream> // std streams
#include <string> // std::string
#include <utility> // std::pair

template<std::size_t n>
class print {
public:

  print() {
    std::cout << n << "\n";
  }
};

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

int main() {
  using hello = cphmpl::charlist<'h', 'e', 'l', 'l', 'o'>;
  static_assert(hello::length == 5);
  print<hello::length> chars;
  static_assert(hello::value<0> == 'h');
  static_assert(cphmpl::get_value<hello, 4> == 'o');
  using ello = hello::pop_front;
  using fello = ello::push_front<'f'>;
  using fellow = fello::push_back<'w'>;
  static_assert(fellow::length == 6);

  using T = cphmpl::typelist<char, unsigned char, signed char>;
  static_assert(std::is_same_v<T::type<0>, char>);
  static_assert(std::is_same_v<cphmpl::get_type<T, 2>, signed char>);
  print<T::length> types;
  static_assert(not T::is_member<std::byte>);

  using P = std::pair<unsigned int, std::string>;
  static_assert(std::is_same_v<cphmpl::get_type<P, 0>, unsigned int>);

  using L = cphmpl::intlist<0, 1, 2, 3>;
  print<L::length> ints;

  ignore_warning(chars);
  ignore_warning(types);
  ignore_warning(ints);

  return 0;
}
```

Figure 5. A simple program illustrating the use of the CPH MPL lists. **Source:** file `cphmpl/example.c++`

A simple example of the use of the CPH MPL compile-time lists is given in Figure 5. In principle, these sequences are like any arrays, but the syntax of some of the operations is exotic. The reader should remember that type expressions are read from right to left. Having this in mind, it should be easier to parse these expressions. For example, on line 35 of Figure 5, the expression **not** `T::is_member<std::byte>` should be read as "`std::byte` is **not** a member of the typelist `T`", which is **true**.

As shown below, when the example program in Figure 5 is run on a Linux terminal, it outputs three numbers, the lengths of the manipulated lists—and all the static assertions hold. In this case, the CPH MPL package was installed under the name `cphmpl` in the parent directory so, therefore, the parent directory was included in the compilation process.

```
$ make -f test.mk example.test
g++-9 -std=c++2a -Wall -Wextra -x c++ -fconcepts -I.. example.c++
./a.out
5
3
4
```

3. Type functions

A full list of the type functions provided in the file `cphmpl/functions.h++` is given in Figures 6, 7, 8, 9. This material has been extracted from the documentation comments which are in the file (see Appendix A). The functions have been divided into four categories: (1) the basic predicates taken directly from the draft of the C++20 standard [1], (2) the type-classification and type-relationship predicates used in our own development, (3) the type-association and type-construction functions used in our own development, and (4) other miscellaneous functions needed in our own development. These functions operate on type `T`, and possibly on type `U`, which can be element types, reference types, iterator types, collection types, or callable types.

To investigate how different type functions are used in an application where metaprogramming is essential, we took the file `cphstl/integers.h++` from the CPH STL [April 2020], in which the class templates `cphstl::`$\mathbb{N}$, `cphstl::`$\mathbb{Z}$, and `cphstl::integer` were implemented. Both of the first two class templates had two partial specializations, one for one-word integers and another for multi-word integers. That is, in this file the 40+ integer operations were implemented five times—of course, as efficiently as possible.

In total, the examined file had 4 093 lines of code (LOC). In our LOC counts, all comments, lines only having a single parenthesis, debugging aids, and assertions are excluded. Of these lines, 568 contained one or more invocations of a CPH MPL function. This indicates that in modern software many lines of code are used to ensure compatibility of the components.

In Table 1, we show how many times the most-frequently-used type functions were invoked. In all, 22 different type functions were used.

Type relationship: `cphmpl::is_same<T, U>` → **bool**
Effect: Return **true** if `T` and `U` name the same type (taking into account **const**/**volatile** qualifications); otherwise, return **false**.

Type relationship: `cphmpl::is_convertible<T, U>` → **bool**
Effect: Return **true** if an expression of the type/value specified by `T` can be implicitly and explicitly converted to the type `U`, and if the two forms of conversions are equivalent.

Type relationship: `cphmpl::is_derived<T, U>` → **bool**
Effect: Return **true** if `U` is a class type that is either `T` or a **public** and unambiguous base of `T`, ignoring the **const**/**volatile** qualifiers.

Type classification: `cphmpl::is_dereferenceable<T>` → **bool**
Effect: Return **true** if an object of type `T` is dereferenceable.

Type classification: `cphmpl::is_destructible<T>` → **bool**
Effect: Return **true** if an object of type `T` can be safely destroyed at the end of its lifetime.

Type classification: `cphmpl::is_constructible<T, Args...>` → **bool**
Effect: Return **true** if an object of type `T` can be created using the given set of argument types `Args...`.

Type classification: `cphmpl::is_destructible<T>` → **bool**
Effect: Return **true** if an object of type `T` can be safely destroyed at the end of its lifetime.

Type classification: `is_default_constructible<T>` → **bool**
Effect: Return **true** if an object of `T` is an object or a reference type, and an object of the specified type can be created using an empty argument list.

Type classification: `is_default_initializable<T>` → **bool**
Effect: Return **true** if a variable of type `T` can be (1) value-initialized, (2) direct-list-initialized from an empty initializer list, and (3) default-initialized.

Type classification: `cphmpl::is_move_constructible<T>` → **bool**
Effect: Return **true** if `T` is a reference type, or if it is an object type where an object of that type can be constructed from an rvalue of that type in both direct- and copy-initialization contexts.

Type classification: `cphmpl::is_copy_constructible<T>` → **bool**
Effect: Return **true** if `T` is an lvalue reference type, or if it is a move-constructible object type where an object of that type can be constructed from an lvalue (possibly **const**) or **const** rvalue of that type in both direct- and copy-initialization contexts.

Type classification: `cphmpl::is_move_assignable<T>` → **bool**
Effect: Return **false** if `T` is not a referenceble object; otherwise, return `std::is_assignable_v<T&, T&&>`.

Figure 6. Some basic predicates used in the definitions of the other type functions. Most of these concepts (modulo naming) have been taken from the draft of the C++20 standard [1]. **Source:** file `cphmpl/functions.h++`

Type relationship: `cphmpl::has_common_reference<T, U>` → **bool**
Effect: Return **true** if `T` and `U` share a common reference type (computed by `std::common_reference_t`) to which both can be converted.

Type relationship: `cphmpl::is_assignable<T, U>` → **bool**
Effect: Return **true** if an expression of the type/value specified by `U` can be assigned to an lvalue expression whose type is specified by `T`.

Type classification: `cphmpl::is_swappable<T>` → **bool**
Effect: Return **false** if `T` is not referenceable; otherwise, return `std::is_swappable_with_v<T&, T&>`.

Type classification: `cphmpl::is_movable<T>` → **bool**
Effect: Return **true** if an object of type `T` can be moved (i.e. `T` supports move construction, move assignment, and lvalues of type `T` can be swapped).

Type classification: `cphmpl::is_copyable<T>` → **bool**
Effect: Return **true** if an object of type `T` can be moved and also copied (i.e. `T` supports copy construction and copy assignment).

Type classification: `cphmpl::is_boolean<T>` → **bool**
Effect: Return **true** if `T` is usable in Boolean contexts. For `is_boolean` to be satisfied, the logical operators must have their usual behaviour.

Type classification: `cphmpl::is_invocable<T, Args...>` → **bool**
Effect: Return **true** if a callable type `T` can be called with a set of argument types `Args...` using the function template `std::invoke`.

Type classification: `cphmpl::is_predicate<T, Args...>` → **bool**
Effect: Return **true** if a callable type `T` can be called with a set of argument types `Args...` and the type of the return value is usable in Boolean contexts.

Figure 6. (Cont.)

Type relationship: `cphmpl::is_safely_convertible<T, U>` → **bool**
Effect: Return **true** if if type `T` is safety convertible to type `U`.

Type classification: `cphmpl::is_constant<T>` → **bool**
Effect: Check if `T` is a class type that wraps a compile-time constant like `std::integral_constant`. A class type will qualify if it has the **static** variable `T::is_constant` which is set to **true**.

Type classification: `cphmpl::is_built_in_character<T>` → **bool**
Effect: Check if `T` is a member of the typelist {**char**, **wchar_t**, `char8_t`, **char16_t**, **char32_t**}, or if it is any **const**/**volatile**-qualified version of them.

Type classification: `cphmpl::is_character<T>` → **bool**
Effect: Check if `T` is a built-in character type or a class type that has the **static** variable `T::is_character` which is set to **true**.

Type classification: `cphmpl::is_built_in_integer<T>` → **bool**
Effect: Check if `T` is a member of the typelist {**bool**, **unsigned char**, **signed char**, **unsigned short int**, **signed short int**, **unsigned int**, **signed int**, **unsigned long int**, **signed long int**, **unsigned long long int**, **signed long long int**}, or if it is any **const**/**volatile**-qualified version of them.

Type classification: `cphmpl::is_gnu_extension<T>` → **bool**
Effect: Check if `T` is a member of the typelist {**unsigned** `__int128`, **signed** `__int128`}, or if it is any **const**/**volatile**-qualified version of them.

Type classification: `cphmpl::is_integer<T>` → **bool**
Effect: Check if `T` is a built-in integer type, an integral enumeration type, a GNU integer extension type, a class type that has the static variable `T::is_integer` which is set to true, or if it is any **const**/**volatile**-qualified version of them.

Type classification: `cphmpl::is_built_in_floating_point<T>` → **bool**
Effect: Check if `T` is a member of the typelist {**float**, **double**, **long double**}, or if it is any **const**/**volatile**-qualified version of them.

Type classification: `cphmpl::is_floating_point<T>` → **bool**
Effect: Check if `T` is a built-in floating-point type, a class type that has the **static** variable `T::is_floating_point` which is set to **true**, or if it is any **const**/**volatile**-qualified version of them.

Type classification: `cphmpl::is_arithmetic<T>` → **bool**
Effect: Return **true** if `T` is an integer type or a floating-point type.

Type classification: `cphmpl::is_unsigned<T>` → **bool**
Effect: Return **true** if `T` is an **unsigned** integer type. A class type will qualify if it has the **static** variable `T::is_signed` which is set to **false**.

Type classification: `cphmpl::is_signed<T>` → **bool**
Effect: Return **true** if `T` is a signed integer type. A class type will qualify if it has the **static** variable `T::is_signed` which is set to **true**.

Figure 7. The type-classification and type-relationship predicates used by us. **Source:** file `cphmpl/functions.h++`

Type classification: `cphmpl::is_bounded<T>` → **bool**
Effect: Return **true** if `T` is a type for which the value universe is finite. A class type will qualify if it has the **static** variable `T::is_bounded` which is set to **true**.

Type classification: `cphmpl::is_exact<T>` → **bool**
Effect: Return **true** if `T` is an arithmetic type for which the representation is exact. A class type will qualify if that it has the **static** variable `T::is_exact` which is set to **true**.

Type classification: `cphmpl::is_modulo<T>` → **bool**
Effect: Return **true** if `T` is an arithmetic type which handles overflows with modulo arithmetic. A class type will qualify if it has the **static** variable `T::is_modulo` which is set to **true**.

Type classification: `cphmpl::is_const_reference<T>` → **bool**
Effect: Return **true** if `T` is a reference type and the value referred to cannot be modified.

Type classification: `cphmpl::is_iterator<T>` → **bool**
Effect: Return **true** if `T` is an iterator type fulfilling the requirements specified in the C++20 standard.

Type classification: `cphmpl::is_const_iterator<T>` → **bool**
Effect: Return **true** if `T` is an iterator type and the value referred to cannot be modified.

Type classification: `cphmpl::is_tuple<T>` → **bool**
Effect: Return **true** if `T` specifies a tuple with some unspecified number of elements.

Type classification: `cphmpl::is_pair<T>` → **bool**
Effect: Return **true** if `T` specifies a two-element pair.

Type classification: `cphmpl::is_bitset<T>` → **bool**
Effect: Return **true** if `T` is similar to the bitset specified in the C++20 standard. A class type will qualify if it has the **static** variable `T::is_bitset` which is set to **true**.

Type classification: `cphmpl::is_container<T>` → **bool**
Effect: Return **true** if `T` fulfils the container requirements specified in the C++20 standard.

Type classification: `cphmpl::is_slice<T>` → **bool**
Effect: Return **true** if `T` specifies an iterator range, but `T` is not a container. Often a slice is specified by a pair of iterators.

Figure 7. (Cont.)

Type classification: `cphmpl::specifies_range<T>` → **bool**
Effect: Check if `T` has the type members `value_type`, `reference`, `const_reference`, `iterator`, `const_iterator`, `difference_type`, and `size_type`; and if `T` is accepted as an argument by the functions `std::begin`, `std::cbegin`, `std::end`, `std::cend`, `std::size`, and `std::empty`.

Type classification: `cphmpl::allows_bit_view<T>` → **bool**
Effect: Check if objects of type `T` as a whole can be seen as a string of bits. This is **true** for constants, integers, bitsets, and collections storing bounded values.

Figure 7. (Cont.)

Type association: `cphmpl::value<T>` → **typename**
Effect: Provide an alias for the type of the value(s) stored in an object specified by type `T`. For a class type the result is `T::value_type` if this type member exists.

Type association: `cphmpl::reference<T>` → **typename**
Effect: Provide an alias for the reference type referring to the value(s) specified by type `T`. For a class type the result is `T::reference` if this type member exists.

Type association: `cphmpl::const_reference<T>` → **typename**
Effect: Provide an alias for the const-reference type referring to the value(s) specified by type `T`. For a class type the result is `T::const_reference` if this type member exists.

Type association: `cphmpl::rvalue_reference<T>` → **typename**
Effect: Provide an alias for the rvalue-reference type referring to the value(s) specified by type `T`. For a class type the result is `T::rvalue_reference` if this type member exists.

Type association: `cphmpl::pointer<T>` → **typename**
Effect: Provide an alias for the pointer type pointing to the value(s) specified by type `T`. For a class type the result is `T::pointer` if this type member exists.

Type association: `cphmpl::const_pointer<T>` → **typename**
Effect: Provide an alias for the const-pointer type pointing to the value(s) specified by type `T`. For a class type the result is `T::const_pointer` if this type member exists.

Type association: `cphmpl::difference<T>` → **typename**
Effect: Provide an alias for the difference type to be used for storing the distance between two values specified by type `T`. For a class type the result is `T::difference_type` if this type member exists.

Type association: `cphmpl::cell<T>` → **typename**
Effect: Return an alias for the type of cells used for storing the values specified by type `T`. For a class type the result is `T::cell` or `node_type` if either one of these type members exists.

Type association: `cphmpl::key<T>` → **typename**
Effect: Return an alias for the type of keys in the value(s) specified by type `T`, i.e. the values are (key, mapped) pairs. For a class type the result is `T::key_type` if this type member exists.

Type association: `cphmpl::mapped<T>` → **typename**
Effect: Return an alias for the mapped type in the values specified by type `T`, i.e. the values are (key, mapped) pairs. For a class type the result is `T::mapped_type` if this type member exists.

Figure 8. The type-association and type-construction functions used by us. **Source:** file `cphmpl/functions.h++`

Type association: `cphmpl::iterator<T>` → **typename**
Effect: Return an alias for the iterator type associated with type `T`. For a class type the result is `T::iterator` if this type member exists.
Observe: This function provides iterators even for types like **int** and `std::bitset` that do not have any iterators themselves. You need the file `cphstl/iterators.h++` for this to work.

Type association: `cphmpl::const_iterator<T>` → **typename**
Effect: Return an alias for the const-iterator type associated with type `T`. For a class type the result is `T::const_iterator` if this type member exists.

Type association: `cphmpl::category<T>` → **typename**
Effect: Return the category of iterators associated with type `T`. For a class type the result is `T::iterator_category` if this type member exists.

Type construction: `cphmpl::make_unsigned<T>` → **typename**
Effect: Return the **unsigned** integer type corresponding to type `T`, with the same **const**/**volatile** qualifiers. This will work for a class type provided that it knows its **unsigned** companion specified by the type member `T::companion`.

Type construction: `cphmpl::make_signed<T>` → **typename**
Effect: Return the **signed** integer type corresponding to type `T`, with the same **const**/**volatile** qualifiers. This will work for a class type provided that it knows its **signed** companion specified by the type member `T::companion`.

Type construction: `cphmpl::twice_wider<T>` → **typename**
Effect: Provide an alias for the type that is twice wider than the bounded integer type `T`.

Figure 8. Cont.

Type property: `cphmpl::round_style<T>` → `std::float_round_style`
Effect: Return the rounding style used by the arithmetic type `T`. For integers, the value should be `std::round_toward_zero`.

Type property: `cphmpl::radix<T>` → `integer`
Effect: Return the base of the number system used in the representation of the arithmetic type `T`. This will work for a class type provided that it has the **static** variable `T::radix` which records the base.

Numeric predicate: `cphmpl::is_power_of_two<integer>` → **bool**
Effect: Return **true** if the integer is a positive power of two.

Numeric function: `cphmpl::lg<integer>` → **unsigned** integer
Effect: Return the whole-number binary logarithm of the absolute value $|\mathtt{x}|$ of an integer `x`, i.e. $\lfloor \log_2 |x| \rfloor$; return 0, if `x` is zero.

Type property: `cphmpl::width<T>` → **unsigned** integer
Effect: Return the number of bits in the value representation of an object of type `T`; this number is indeterminate if `T` is unbounded. The object representation can be larger. For example, there can be some padding bits due to alignment. This will work for a class type provided that it has the **static** variable `T::width` which records the bit width.

Type selection: `cphmpl::first_wide_enough<T, integer>` → **typename**
Effect: Return the first type in the list `T` of types whose width is larger than or equal to the given integer.

Value generation: `cphmpl::min<T>` → `T`
Effect: Return the minimum finite value representable by type `T`. This will work for a class type provided that it has the **static** member function `T::min()` which returns the desired value.

Value generation: `cphmpl::max<T>` → `T`
Effect: Return the maximum finite value representable by type `T`. This will work for a class type provided that it has the **static** member function `T::max()` which returns the desired value.

Figure 9. Other miscellaneous functions needed by us. **Source:** file `cphmpl/functions.h++`

Type function	Invocations
cphmpl::is_built_in_integer	108
cphmpl::width	104
cphmpl::specifies_range	88
std::is_same_v	82
cphmpl::is_gnu_extension	78
cphmpl::is_unsigned	77
cphmpl::value	66
cphmpl::is_integer	47
cphmpl::const_iterator	24
cphmpl::is_bounded	23

Table 1. The ten most frequently used type functions in the file cphstl::integers.h++

.

4. Concepts

It was an interesting exercise to take the concept specifications from the draft of the C++20 standard [1] and convert them to a runnable code. In this particular case, the compiler in use was GNU g++ (version 9.2.1). Since the compiler did not support some of the convenience features used in the specification, it was necessary to do the implementation conservatively:

- In the code the keyword **typename** in the template parameter lists was never replaced with a concept name, but the concepts were just used as Boolean predicates. Solely the **requires** clause was used to check that the given arguments will satisfy the specified requirements. Actually, this is our preferable way of using the concepts since the compliance requirements can be complicated: It may not be enough to check that the requirements are satisfied for one argument, but it can also be important to check the compatibility with other arguments.
- Sometimes the compiler refused to accept a **requires** expression in an **if constexpr** statement. To overcome this problem, we used partial specialization and relied on SFINAE [5, Section 8.4].
- The <iterator> header was not aware of the new iterator category std::contiguous_iterator so concepts related to those were omitted.

For C++20, the iterator concepts are different from those used for C++17. In the online documentation at https://en.cppreference.com/, the old iterators are called legacy iterators. However, the standard draft specified both sets of concepts and we support them under separate namespaces: the old concepts are in the **namespace** cpp17 and the new ones in the **namespace** cpp20. In this way both sets of concepts can be used simultaneously.

After doing some straightforward code transformations to the concepts specified in the draft, the compiler stopped with a serious error:

error: 'std::common_reference_t' is not a member of 'std';

The definition of this type function was in the file <type_traits_latest>, but even after including this file the error still occurred. To get over this hurdle, a separate patch file type_traits_latest was created and the definition of the missing type function was placed there. This solved the issue. Since it is expected that this error will be fixed in the next release of the compiler, this patch file is not included in the package.

Of course, I have to apologize the users for the fact that the names of the concepts are not exactly the same as those specified in the standard. The two main differences are: (1) The names of the type predicates have the prefix "is_" or "has_". (2) In the names of the type functions the suffixes "_as", "_for", "_from", "_t", "_to", "_v", and "_with" have been removed to make the names shorter. The hope is that this file will be short-lived, but it is convenient to have both the legacy and the new concepts in one place.

The best documentation for the concepts specified in the C++ standard is provided in the standard itself. However, the concepts are still under development; it is quite probable that there will be more changes. Thus, it is not the right time to make a big effort to document the facilities available in the file cphmpl/concepts.h++. This file is released in the form as it is—with no additional documentation.

5. Future

It must be emphasized that you use the tools released in this package at your own risk. Let us concretize this statement with an example. When I tested the value-generating functions cphmpl::min<T> and cphmpl::max<T> with a small ten-line program, I accidentally made a one-letter typographical error in the template argument; I wrote ZL instead of the correct XL. The consequence was that the compilation process ended with the error message

"internal compiler error: Segmentation fault".

I figured out that this destructive behaviour could be avoided by replacing the shorthand cphmpl::width<T> with cphmpl::traits::width<T>::value in the **requires** clauses, when doing integer comparisons. (An error in the implementation of cphmpl::width was not found until later.)

It must be admitted that the development of the CPH MPL has progressed in an ad-hoc manner. A feature was added to the package when it was needed. Any further development should be better planned. In principle, we had to generalize the utilities in the header files <limits> and <type_traits> so that the existing type functions can handle class types as well. The hooks needed to facilitate this are clearly documented in this report. The current draft of the C++20 standard explicitly forbids users and library developers to provide specializations for many of the type functions. It would be more productive if the hooks needed were standardized instead.

The next question is whether we can trust on the hooks provided by the developers. To check some semantic properties, we have to. Let us consider a clarifying example. According to both of the header files <limits> and

<type_traits>, bool is an integer type. However, the C++17 standard explicitly forbids the operator ++ for bool, so it does not support the same operations as, for example, int. In our code we trust on the developers. In particular, for an integer type we do not check that all the 40+ operations are actually supported; we just check that the hook T::is_integer is there.

Our hope is that most of the facilities discussed in this document will be added, in one form or another, to the C++ standard library. Already now cphmpl::intlist is redundant since the <utility> header offers the class template std::integer_sequence that can be used for the same purpose. As pointed out in the first write-up [2], the template-based collections have serious performance issues. Typelists are normally short so this is not a problem here. However, there should be a more direct way of conjoining templates and strings of characters—which has partially already happened.

There are some overlap between the type functions available in the files cphmpl/functions.h++ and cphmpl/concepts.h++. Hence, a reorganization might be necessary. However, we have deliberately decided not to do any major changes before receiving proper user feedback.

Acknowledgements

Thanks to Andreas Milton Maniotis for initiating the CPH MPL project and for commenting this report.

Software availability

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

Appendix A: Source code

Copyright notice

attributed to the original source. 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

2020-05-16

Contents

cphmpl/lists.h++

```
1  /*
2    Author: Jyrki Katajainen © 2017, 2020
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 /// compile-time classification functions
14
15 // is_valuelist
16
17 namespace cphmpl::traits {
18
19   template<typename T>
20   class is_valuelist {
21   public:
22
23     static constexpr bool value = false;
24   };
25
26   template<auto... Vcal>
27   class is_valuelist<cphmpl::valuelist<Vcal...>> {
28   public:
29
30     static constexpr bool value = true;
31   };
32
33   template<auto... Vcal, template<auto...> class Name>
34   class is_valuelist<Name<Vcal...>> {
35   public:
36
37     static constexpr bool value = true;
38   };
39 }
40
41 namespace cphmpl {
42
43   // Type classification: cphmpl::is_valuelist<T> → bool
44
45   // Effect: Return true if T is cphmpl::valuelist<...> or a class
46   // type that keeps a collection of non-types in its template
47   // parameter list.
48
49   template<typename T>
```

```
50    concept bool is_valuelist = traits::is_valuelist<std::remove_cv_t<T>>::value;
51  }
52
53  // specifies_valuelist
54
55  namespace cphmpl::traits {
56
57    template<typename T>
58    class specifies_valuelist {
59    public:
60
61      static constexpr bool value = false;
62    };
63
64    template<auto... Vcal, template<auto...> class Name>
65    class specifies_valuelist<Name<Vcal...>> {
66    public:
67
68      static constexpr bool value = true;
69    };
70  }
71
72  namespace cphmpl {
73
74    // Type classification: cphmpl::specifies_valuelist<T> → bool
75
76    // Effect: Return true if type T keeps a collection of non–types in
77    // its template parameter list.
78
79    template<typename T>
80    concept bool specifies_valuelist =
        ↪ traits::specifies_valuelist<std::remove_cv_t<T>>::value;
81  }
82
83  // is_typelist
84
85  namespace cphmpl::traits {
86
87    template<typename T>
88    class is_typelist {
89    public:
90
91      static constexpr bool value = false;
92    };
93
94    template<typename... 𝒳>
95    class is_typelist<cphmpl::typelist<𝒳...>> {
96    public:
97
98      static constexpr bool value = true;
99    };
100 }
```

```
101
102 namespace cphmpl {
103
104   // Type classification: cphmpl::is_typelist<T> → bool
105
106   // Effect: Return true if T is cphmpl::typelist<...> that keeps a
107   // collection of types in its template parameter list.
108
109   template<typename T>
110   concept bool is_typelist = traits::is_typelist<std::remove_cv_t<T>>::value;
111 }
112
113 // specifies_typelist
114
115 namespace cphmpl::traits {
116
117   template<typename T>
118   class specifies_typelist {
119   public:
120
121     static constexpr bool value = false;
122   };
123
124   template<typename... X, template <typename...> class Name>
125   class specifies_typelist<Name<X...>> {
126   public:
127
128     static constexpr bool value = true;
129   };
130 }
131
132 namespace cphmpl {
133
134   // Type classification: cphmpl::specifies_typelist<T> → bool
135
136   // Effect: Return true if T is a tuple, a typelist, or any other
137   // class type that keeps a collection of types in its template
138   // parameter list.
139
140   template<typename T>
141   concept bool specifies_typelist = cphmpl::traits::specifies_typelist<T>::value;
142 }
143
144 /// generic non-member functions
145
146 // size
147
148 namespace cphmpl::traits {
149
150   template<typename T>
151   class size {
152   public:
```

```
153
154     static constexpr std::size_t value = 0;
155   };
156
157   template<auto... Vcal, template<auto...> class Name>
158   class size<Name<Vcal...>> {
159   public:
160
161     static constexpr std::size_t value = sizeof...(Vcal);
162   };
163
164   template<typename... T, template<typename...> class Name>
165   class size<Name<T...>> {
166   public:
167
168     static constexpr std::size_t value = sizeof...(T);
169   };
170 }
171
172 namespace cphmpl {
173
174   // Type property: size<type> → std::size_t
175
176   // Effect: Return the number of the template parameters taken by the
177   // given type. The parameter list must specify a set of types or a
178   // set of non–types, but not a mixture of them.
179
180   template<typename T>
181   requires cphmpl::specifies_valuelist<T> or cphmpl::specifies_typelist<T>
182   static constexpr std::size_t size = cphmpl::traits::size<T>::value;
183 }
184
185 // get_type
186
187 namespace cphmpl::traits {
188
189   template<typename T, std::size_t I>
190   class get_type;
191
192   // base case
193
194   template<auto head, auto... Vcal, template<auto...> class Name>
195   class get_type<Name<head, Vcal...>, std::size_t(0)> {
196   public:
197
198     using type = decltype(head);
199   };
200
201   template<typename H, typename... T, template<typename...> class Name>
202   class get_type<Name<H, T...>, std::size_t(0)> {
203   public:
204
```

```
205     using type = H;
206   };
207
208   // special case for pairs
209
210   template<typename X, typename Y, template<typename, typename> class Name>
211   class get_type<Name<X, Y>, std::size_t(0)> {
212   public:
213
214     using type = X;
215   };
216
217   template<typename X, typename Y, template<typename, typename> class Name>
218   class get_type<Name<X, Y>, std::size_t(1)> {
219   public:
220
221     using type = Y;
222   };
223
224   // recursive case
225
226   template<auto head, auto... Vcal, template<auto...> class Name, std::size_t I>
227   class get_type<Name<head, Vcal...>, I> {
228   public:
229
230     using type = typename cphmpl::traits::get_type<Name<Vcal...>, I - 1>::type;
231   };
232
233   template<typename H, typename... T, template<typename...> class Name, std::size_t I>
234   class get_type<Name<H, T...>, I> {
235   private:
236
237     using Tail = Name<T...>;
238
239   public:
240
241     using type = typename cphmpl::traits::get_type<Tail, I - 1>::type;
242   };
243 }
244
245 namespace cphmpl {
246
247   // Compile-time function: get_type<T, std::size_t I> -> type
248
249   // Effect: Retrieve the type of the I'th element in the template
250   // parameter list of type T
251
252   template<typename T, std::size_t I>
253   requires cphmpl::specifies_valuelist<T> or cphmpl::specifies_typelist<T>
254   using get_type = typename cphmpl::traits::get_type<T, I>::type;
255 }
256
```

```
257 // get_value
258
259 namespace cphmpl {
260
261   // Compile-time function: get_value<T, std::size_t I> → constant
262
263   // Effect: Get the I'th element in the valuelist T.
264
265   template<typename T, std::size_t I>
266   requires cphmpl::is_valuelist<T>
267   static constexpr auto get_value = T:: template value<I>;
268 }
269
270 #endif
```

cphmpl/valuelist.h++

```
1  /*
2    Author: Jyrki Katajainen © 2017–2020
3  */
4
5  #ifndef __CPHMPL_VALUELIST__
6  #define __CPHMPL_VALUELIST__
7
8  #include <cstddef> // std::size_t
9  #include <type_traits> // std::conditional
10
11 namespace cphmpl {
12
13   template<auto... Vcal>
14   class valuelist {
15   private:
16
17     // forward declarations
18
19     template<typename, std::size_t>
20     class at_helper;
21
22     template<typename>
23     class pop_front_helper;
24
25     template<typename, typename>
26     class pop_back_helper;
27
28     template<typename, typename, template<auto> class>
29     class filter_helper;
30
31     template<typename, std::size_t, template<auto> class>
32     class find_if_helper;
33
34     template<typename, std::size_t, template<auto> class>
```

```
35      class count_if_helper;
36
37      template<typename, auto>
38      class is_member_helper;
39
40    public:
41
42      using self = cphmpl::valuelist<Vcal...>;
43      using size_type = std::size_t;
44
45      static constexpr std::size_t length = sizeof...(Vcal);
46      static constexpr bool is_empty = false;
47      static constexpr auto front = at_helper<self, std::size_t(0)>::value;
48      static constexpr auto back = at_helper<self, length - 1>::value;
49
50      template<std::size_t index>
51      static constexpr auto value = at_helper<self, index>::value;
52
53      template<auto element>
54      using push_front = valuelist<element, Vcal...>;
55
56      template<auto element>
57      using push_back = valuelist<Vcal..., element>;
58
59      using pop_front = typename pop_front_helper<self>::type;
60
61      using pop_back = typename pop_back_helper<self, valuelist<>>::type;
62
63      template<template<auto> class P>
64      using filter = typename filter_helper<self, valuelist<>, P>::type;
65
66      template<template<auto> class F>
67      using transform = valuelist<F<Vcal>::value...>;
68
69      template<template<auto> class P>
70      static constexpr std::size_t find_if = find_if_helper<self, length, P>::value;
71
72      template<template<auto> class P>
73      static constexpr std::size_t count_if = count_if_helper<self, std::size_t(0),
        ↪ P>::value;
74
75      template<auto element>
76      static constexpr bool is_member = is_member_helper<self, element>::value;
77
78    private:
79
80      // at: 0-based indexing
81
82      template<typename L, std::size_t index>
83      class at_helper {
84      public:
85
```

```
86        static_assert(index < L::length, "index out of bounds");
87      };
88
89      template<auto head, auto... T>
90      class at_helper<valuelist<head, T...>, std::size_t(0)> {
91      public:
92
93        static constexpr auto value = head;
94      };
95
96      template<auto head, auto... T, std::size_t i>
97      requires (i > std::size_t(0))
98      class at_helper<valuelist<head, T...>, i> {
99      public:
100
101       static constexpr auto value = at_helper<valuelist<T...>, i - 1>::value;
102     };
103
104     // pop_front
105
106     template<typename>
107     class pop_front_helper {
108     public:
109
110       using type = valuelist<>;
111     };
112
113     template<auto head, auto... T>
114     class pop_front_helper<valuelist<head, T...>> {
115     public:
116
117       using type = valuelist<T...>;
118     };
119
120     // pop_back
121
122     template<typename, typename>
123     class pop_back_helper {
124     public:
125
126       using type = valuelist<>;
127     };
128
129     template<auto last, auto... X>
130     class pop_back_helper<valuelist<last>, valuelist<X...>> {
131     public:
132
133       using type = valuelist<X...>;
134     };
135
136     template<auto head, auto... T, auto... X>
137     class pop_back_helper<valuelist<head, T...>, valuelist<X...>> {
```

```
138   public:
139
140     using type = typename pop_back_helper<valuelist<T...>, valuelist<X...,
      ↪ head>>::type;
141   };
142
143   // filter: P is a unary predicate
144
145   template<typename, typename, template<auto> class>
146   class filter_helper;
147
148   template<typename L, template<auto> class P>
149   class filter_helper<valuelist<>, L, P> {
150   public:
151
152     using type = L;
153   };
154
155   template<auto head, auto... T, auto... X, template<auto> class P>
156   class filter_helper<valuelist<head, T...>, valuelist<X...>, P> {
157   public:
158
159     using type = typename std::conditional<(P<head>::value),
160     typename filter_helper<valuelist<T...>, valuelist<X..., head>, P>::type,
161     typename filter_helper<valuelist<T...>, valuelist<X...>, P>::type
162     >::type;
163   };
164
165   // find_if: P is a unary predicate
166
167   template<typename, std::size_t, template<auto> class>
168   class find_if_helper;
169
170   template<std::size_t n, template<auto> class P>
171   class find_if_helper<valuelist<>, n, P> {
172   public:
173
174     static constexpr std::size_t value = n;
175   };
176
177   template<auto head, auto... T, std::size_t n, template<auto> class P>
178   class find_if_helper<valuelist<head, T...>, n, P> {
179   public:
180
181     static constexpr std::size_t value = (P<head>::value) ?
182       n - valuelist<head, T...>::length :
183       find_if_helper<valuelist<T...>, n, P>::value;
184   };
185
186   // count_if: P is a unary predicate
187
188   template<typename, std::size_t, template<auto> class>
```

```
189     class count_if_helper;
190
191     template<std::size_t n, template<auto> class P>
192     class count_if_helper<valuelist<>, n, P> {
193     public:
194
195       static constexpr std::size_t value = n;
196     };
197
198     template<auto head, auto... T, std::size_t n, template<auto> class P>
199     class count_if_helper<valuelist<head, T...>, n, P> {
200     public:
201
202       static constexpr std::size_t value = (P<head>::value) ?
203         count_if_helper<valuelist<T...>, n + 1, P>::value :
204         count_if_helper<valuelist<T...>, n, P>::value;
205     };
206
207     // is_member
208
209     template<typename, auto>
210     class is_member_helper;
211
212     template<auto element>
213     class is_member_helper<valuelist<>, element> {
214     public:
215
216       static constexpr bool value = false;
217     };
218
219     template<auto head, auto... T, auto element>
220     class is_member_helper<valuelist<head, T...>, element> {
221     public:
222
223       static constexpr bool value = (head == element) ?
224         true : is_member_helper<valuelist<T...>, element>::value;
225     };
226   };
227
228   template<>
229   class valuelist<> {
230   public:
231
232     using self = cphmpl::valuelist<>;
233     using size_type = std::size_t;
234
235     static constexpr std::size_t length = 0;
236     static constexpr bool is_empty = true;
237
238     // static constexpr auto front = undefined;
239     // static constexpr auto back = undefined;
240
```

```
241       // template<std::size_t index>
242       // static constexpr auto value = undefined;
243
244       template<auto element>
245       using push_front = valuelist<element>;
246
247       template<auto element>
248       using push_back = valuelist<element>;
249
250       // using pop_front = undefined;
251       // using pop_back = undefined;
252
253       template<template<auto> class P>
254       using filter = self;
255
256       template<template<auto> class F>
257       using transform = self;
258
259       template<template<auto> class P>
260       static constexpr std::size_t find_if = 0;
261
262       template<template<auto> class P>
263       static constexpr std::size_t count_if = 0;
264
265       template<auto element>
266       static constexpr bool is_member = false;
267     };
268   }
269
270   #endif
```

cphmpl/typelist.h++

```
1   /*
2     Author: Jyrki Katajainen © 2017–2020
3   */
4
5   #ifndef __CPHMPL_TYPELIST__
6   #define __CPHMPL_TYPELIST__
7
8   #include <cstddef> // std::size_t
9   #include <tuple> // std::tuple std::tuple_element std::tuple_size_v
10  #include <type_traits> // std::conditional std::is_same
11
12  namespace cphmpl {
13
14    template<typename... T>
15    class typelist {
16    private:
17
18      // forward declarations
```

```
19
20      template<typename, std::size_t>
21      class at_helper;
22
23      template<typename>
24      class pop_front_helper;
25
26      template<typename, typename>
27      class pop_back_helper;
28
29      template<typename, typename, template<typename> class>
30      class filter_helper;
31
32      template<typename, std::size_t, template<typename> class>
33      class find_if_helper;
34
35      template<typename, std::size_t, template<typename> class>
36      class count_if_helper;
37
38      template<typename, typename>
39      class is_member_helper;
40
41    public:
42
43      using self = cphmpl::typelist<T...>;
44      using size_type = std::size_t;
45
46      static constexpr std::size_t length = sizeof...(T);
47      static constexpr bool is_empty = false;
48      using front = typename at_helper<self, std::size_t(0)>::type;
49      using back = typename at_helper<self, length - 1>::type;
50
51      template<std::size_t index>
52      using type = typename at_helper<self, index>::type;
53
54      template<typename T>
55      using push_front = typelist<T, T...>;
56
57      template<typename T>
58      using push_back = typelist<T..., T>;
59
60      using pop_front = typename pop_front_helper<self>::type;
61
62      using pop_back = typename pop_back_helper<self, typelist<>>::type;
63
64      template<template<typename> class P>
65      using filter = typename filter_helper<self, typelist<>, P>::type;
66
67      template<template<typename> class F>
68      using transform = typelist<typename F<T>::type...>;
69
70      template<template<typename> class P>
```

```
71      static constexpr std::size_t find_if = find_if_helper<self, length, P>::value;
72
73      template<template<typename> class P>
74      static constexpr std::size_t count_if = count_if_helper<self, 0, P>::value;
75
76      template<typename T>
77      static constexpr bool is_member = is_member_helper<self, T>::value;
78
79    private:
80
81      // at: 0-based indexing
82
83      template<typename, std::size_t>
84      class at_helper {
85        // using type = undefined;
86      };
87
88      template<typename H, typename... R>
89      class at_helper<typelist<H, R...>, std::size_t(0)> {
90      public:
91
92        using type = H;
93      };
94
95      template<typename H, typename... R, std::size_t index>
96      class at_helper<typelist<H, R...>, index> {
97      public:
98
99        using type = typename at_helper<typelist<R...>, index - 1>::type;
100     };
101
102     // pop_front
103
104     template<typename>
105     class pop_front_helper {
106     public:
107
108       using type = typelist<>;
109     };
110
111     template<typename H, typename... R>
112     class pop_front_helper<typelist<H, R...>> {
113     public:
114
115       using type = typelist<R...>;
116     };
117
118     // pop_back
119
120     template<typename, typename>
121     class pop_back_helper {
122     public:
```

```
123
124       using type = typelist<>;
125     };
126
127     template<typename H, typename... X>
128     class pop_back_helper<typelist<H>, typelist<X...>> {
129     public:
130
131       using type = typelist<X...>;
132     };
133
134     template<typename H, typename... R, typename... X>
135     class pop_back_helper<typelist<H, R...>, typelist<X...>> {
136     public:
137
138       using type = typename pop_back_helper<typelist<R...>, typelist<X...,
      ↪ H>>::type;
139     };
140
141     // filter: P is a unary predicate
142
143     template<typename, typename, template<typename> class>
144     class filter_helper;
145
146     template<typename L, template<typename> class P>
147     class filter_helper<typelist<>, L, P> {
148     public:
149
150       using type = L;
151     };
152
153     template<typename H, typename... R, typename... X, template<typename> class P>
154     class filter_helper<typelist<H, R...>, typelist<X...>, P> {
155     public:
156
157       using type = typename std::conditional<(P<H>::value),
158       typename filter_helper<typelist<R...>, typelist<X..., H>, P>::type,
159       typename filter_helper<typelist<R...>, typelist<X...>, P>::type
160       >::type;
161     };
162
163     // find_if: P is a unary predicate
164
165     template<typename, std::size_t, template<typename> class>
166     class find_if_helper;
167
168     template<std::size_t n, template<typename> class P>
169     class find_if_helper<typelist<>, n, P> {
170     public:
171
172       static constexpr std::size_t value = n;
173     };
```

```
174
175   template<typename H, typename... R, std::size_t n, template<typename> class P>
176   class find_if_helper<typelist<H, R...>, n, P> {
177   public:
178
179     static constexpr std::size_t value = (P<H>::value) ?
180       n - typelist<H, R...>::length :
181       find_if_helper<typelist<R...>, n, P>::value;
182   };
183
184   // count_if: P is a unary predicate
185
186   template<typename, std::size_t, template<typename> class>
187   class count_if_helper;
188
189   template<std::size_t n, template<typename> class P>
190   class count_if_helper<typelist<>, n, P> {
191   public:
192
193     static constexpr std::size_t value = n;
194   };
195
196   template<typename H, typename... R, std::size_t n, template<typename> class P>
197   class count_if_helper<typelist<H, R...>, n, P> {
198   public:
199
200     static constexpr std::size_t value = (P<H>::value) ?
201       count_if_helper<typelist<R...>, n + 1, P>::value :
202       count_if_helper<typelist<R...>, n, P>::value;
203   };
204
205   // is_member
206
207   template<typename, typename>
208   class is_member_helper;
209
210   template<typename T>
211   class is_member_helper<typelist<>, T> {
212   public:
213
214     static constexpr bool value = false;
215   };
216
217   template<typename H, typename... R, typename T>
218   class is_member_helper<typelist<H, R...>, T> {
219   public:
220
221     static constexpr bool value = (std::is_same<H, T>::value) ?
222       true : is_member_helper<typelist<R...>, T>::value;
223   };
224 };
225
```

```
226   template<>
227   class typelist<> {
228   public:
229
230     using self = cphmpl::typelist<>;
231     using size_type = std::size_t;
232
233     static constexpr std::size_t length = 0;
234     static constexpr bool is_empty = true;
235
236     // using front = undefined;
237     // using back = undefined;
238
239     // template<std::size_t index>
240     // using get = undefined;
241
242     template<typename T>
243     using push_front = typelist<T>;
244
245     template<typename T>
246     using push_back = typelist<T>;
247
248     // using pop_front = undefined;
249     // using pop_back = undefined;
250
251     template<template<typename> class P>
252     using filter = self;
253
254     template<template<typename> class F>
255     using transform = self;
256
257     template<template<typename> class P>
258     static constexpr std::size_t find_if = 0;
259
260     template<template<typename> class P>
261     static constexpr std::size_t count_if = 0;
262
263     template<typename T>
264     static constexpr bool is_member = false;
265   };
266 }
267
268 #endif
```

cphmpl/charlist.h++

```
1  /*
2    Author: Jyrki Katajainen © 2017, 2020
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... C>
14   requires std::conjunction_v<std::is_same<char, decltype(C)>...>
15   class charlist
16     : public valuelist<C...> {
17   public:
18
19     using self = cphmpl::charlist<C...>;
20     using value_type = char;
21   };
22 }
23
24 // some compile-time functions for the manipulation of characters
25
26 namespace cphmpl::helper {
27
28   template<char symbol>
29   constexpr bool is_digit() noexcept {
30     constexpr char digits[] = {'0','1','2','3','4','5','6','7','8','9'};
31     for (char c: digits) {
32       if (c == symbol) {
33         return true;
34       }
35     }
36     return false;
37   }
38 }
39
40 namespace cphmpl {
41
42   // Compile-time function: is_digit<symbol> -> bool
43
44   // Effect: Returns true if the given symbol is a digit
45
46   template<char symbol>
47   constexpr bool is_digit = helper::is_digit<symbol>();
48 }
49
50 namespace cphmpl::helper {
51
52   template<char symbol>
53   constexpr bool is_lower_case_letter() noexcept {
54     constexpr char lower_case_letters[] =
     ↪ {'a','b','c','d','e','f','g','h','i','j','k','l','m','n','o','p','q','r','s','t','u',
55     for (char c: lower_case_letters) {
56       if (c == symbol) {
```

```
57          return true;
58        }
59      }
60      return false;
61    }
62  }
63
64  namespace cphmpl {
65
66    // Compile-time function: is_lower_case_letter<symbol> → bool
67
68    // Effect: Returns true if the given symbol is a lower-case letter
69
70    template<char symbol>
71    constexpr bool is_lower_case_letter = helper::is_lower_case_letter<symbol>();
72  }
73
74  namespace cphmpl::helper {
75
76    template<char symbol>
77    constexpr bool is_upper_case_letter() noexcept {
78      constexpr char upper_case_letters[] =
      ↪ {'A','B','C','D','E','F','G','H','I','J','K','L','M','N','O','P','Q','R','S','T','U','
79      for (char c: upper_case_letters) {
80        if (c == symbol) {
81          return true;
82        }
83      }
84      return false;
85    }
86  }
87
88  namespace cphmpl {
89
90    // Compile-time function: is_upper_case_letter<symbol> → bool
91
92    // Effect: Returns true if the given symbol is an upper-case letter
93
94    template<char symbol>
95    constexpr bool is_upper_case_letter = helper::is_upper_case_letter<symbol>();
96  }
97
98  namespace cphmpl {
99
100   template<char symbol>
101   constexpr bool is_letter =
102     is_lower_case_letter<symbol> or is_upper_case_letter<symbol>;
103 }
104
105 namespace cphmpl::helper {
106
107   // ' '   (0x20)   space (SPC)
```

```
108   // '\t'  (0x09)  horizontal tab (TAB)
109   // '\n'  (0x0a)  newline (LF)
110   // '\v'  (0x0b)  vertical tab (VT)
111   // '\f'  (0x0c)  form feed (FF)
112   // '\r'  (0x0d)  carriage return (CR)
113
114   template<char symbol>
115   constexpr bool is_whitespace() noexcept {
116     constexpr char whitespace_characters[] = {' ','\t','\n','\v','\f','\r'};
117     for (char c: whitespace_characters) {
118       if (c == symbol) {
119         return true;
120       }
121     }
122     return false;
123   }
124 }
125
126 namespace cphmpl {
127
128   // Compile-time function: is_whitespace<symbol> -> bool
129
130   // Effect: Returns true if the given symbol is a whitespace character
131
132   template<char symbol>
133   constexpr bool is_whitespace = helper::is_whitespace<symbol>();
134 }
135
136 namespace cphmpl::helper {
137
138   template<char symbol>
139   constexpr bool is_comma() noexcept {
140     constexpr char comma = ',';
141     return symbol == comma;
142   }
143 }
144
145 namespace cphmpl {
146
147   // Compile-time function: is_comma<symbol> -> bool
148
149   // Effect: Returns true if the given symbol is a comma
150
151   template<char symbol>
152   constexpr bool is_comma = helper::is_comma<symbol>();
153 }
154
155 #endif
```

cphmpl/intlist.h++

```
/*
  Author: Jyrki Katajainen © 2017, 2020
*/

#ifndef __CPHMPL_INTLIST__
#define __CPHMPL_INTLIST__

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

namespace cphmpl {

  template<int... I>
  requires std::conjunction_v<std::is_same<int, decltype(I)>...>
  class intlist
    : public valuelist<I...> {
  public:

    using self = cphmpl::intlist<I...>;
    using value_type = int;
  };

  namespace helper {

    template<typename, typename>
    class merge;

    template<int... L, int... R>
    class merge<cphmpl::intlist<L...>, cphmpl::intlist<R...>>
      : public cphmpl::intlist<L..., (sizeof...(L) + R)...> {
    };

    template<std::size_t n>
    class make_indexlist
      : public merge<typename make_indexlist<n / 2>::self,
                     typename make_indexlist<n - n / 2>::self> {
    };

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

  template<int n>
  using make_indexlist = typename helper::make_indexlist<n>::self;
}

#endif
```

cphmpl/functions.h++

```
1  /*
2    Author: Jyrki Katajainen © 2017–2020
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 <array> // std::array
11 #include <bitset> // std::bitset
12 #include <climits> // CHAR_BIT
13 #include "cphmpl/lists.h++" // cphmpl compile–time lists
14 #include <cstddef> // std::size_t std::ptrdiff_t std::byte
15 #include <functional> // std::invoke
16 #include <iterator> // std::iterator_traits std::begin ...
17 #include <limits> // std::numeric_limits for built–in types
18 #include <memory> // std::pointer_traits
19 #include <type_traits> // std::is_same_v std::is_arithmetic_v ...
20 #include "type_traits_latest" // std::common_reference_t
21 #include <utility> // std::move std::declval
22
23 namespace cphstl { // forwarding
24
25   template<std::size_t>
26   class ℕ;
27
28   template<std::size_t>
29   class ℤ;
30
31   template<typename W>
32   class integer;
33
34   template<typename>
35   class singleton_iterator;
36
37   template<typename>
38   class rank_iterator;
39 }
40
41 namespace cphmpl::traits { // forwarding
42
43   template<typename T>
44   class width;
45 }
46
47 namespace cphmpl::functions {
48
49   template<typename T>
50   class capsule {
51   public:
52
```

```
53      capsule() {
54      }
55
56      using type = T;
57    };
58
59    template<typename T>
60    class is_std_array {
61    public:
62
63      static constexpr bool value = false;
64    };
65
66    template<typename T, std::size_t N>
67    class is_std_array<std::array<T, N>> {
68    public:
69
70      static constexpr bool value = true;
71      static constexpr std::size_t size = N;
72      using value_type = T;
73    };
74
75    template<typename T>
76    class is_std_bitset {
77    public:
78
79      static constexpr bool value = false;
80    };
81
82    template<std::size_t N>
83    class is_std_bitset<std::bitset<N>> {
84    public:
85
86      static constexpr bool value = true;
87      static constexpr std::size_t width = N;
88    };
89  }
90
91  namespace cphmpl {
92
93    constexpr std::size_t indeterminate = std::numeric_limits<std::size_t>::max();
94  }
95
96  // basic type predicates
97
98  namespace cphmpl {
99
100   // Type relationship: cphmpl::is_same<T, U> → bool
101
102   // Effect: Return true if T and U name the same type (taking into
103   // account const/volatile qualifications); otherwise, return false.
104
```

```
105  template<typename T, typename U>
106  concept bool is_same =
107    std::is_same_v<T, U> and
108    std::is_same_v<U, T>;
109
110  // Type relationship: cphmpl::is_convertible<T, U> --> bool
111
112  // Effect: Return true if an expression of the type/value specified
113  // by T can be implicitly and explicitly converted to the type U,
114  // and if the two forms of conversions are equivalent.
115
116  template<typename T, typename U>
117  concept bool is_convertible =
118    std::is_convertible_v<std::add_rvalue_reference_t<T>, U> and
119    requires(T (&from)()) {
120      static_cast<U>(from());
121    };
122
123  // Type relationship: cphmpl::is_derived<T, U> → bool
124
125  // Effect: Return true if U is a class type that is either T or a
126  // public and unambiguous base of T, ignoring const/volatile
127  // qualifiers.
128
129  template<typename T, typename U>
130  concept bool is_derived =
131    std::is_base_of_v<U, T> and
132    std::is_convertible_v<const volatile T*, const volatile U*>;
133
134  // Type classification: cphmpl::is_dereferenceable<T> → bool
135
136  // Effect: Return true if an object of type T is dereferenceable.
137
138  template<typename T>
139  concept bool is_dereferenceable =
140    requires(T& p) {
141      { *p } → auto&&;
142    };
143
144  // Type classification: cphmpl::is_destructible<T> → bool
145
146  // Effect: Return true if an object of type T can be safely
147  // destroyed at the end of its lifetime.
148
149  template<typename T>
150  concept bool is_destructible =
151    std::is_nothrow_destructible_v<T>;
152
153  // Type classification: cphmpl::is_constructible<T, Args...> → bool
154
155  // Effect: Return true if an object of type T can be created using
156  // the given set of argument types Args
```

```
157
158   template<typename T, typename... Args>
159   concept bool is_constructible =
160     is_destructible<T> and
161     std::is_constructible_v<T, Args...>;
162
163   // Type classification: cphmpl::is_default_constructible<T> → bool
164
165   // Effect: Return true if T is an object or a reference type, and an
166   // object of the specified type can be created using an empty argument list.
167
168   template<typename T>
169   concept bool is_default_constructible =
170     is_constructible<T>;
171
172   // Type classification: cphmpl::is_default_initializable<T> → bool
173
174   // Effect: Return true if a variable of type T can be (1)
175   // value-initialized, (2) direct-list-initialized from an empty
176   // initializer list, and (3) default-initialized.
177
178   template<class T>
179   concept bool is_default_initializable =
180     is_constructible<T> and
181     requires { T{}; } and
182     requires { ::new (static_cast<void*>(nullptr)) T; };
183
184   // Type classification: cphmpl::is_move_constructible<T> → bool
185
186   // Effect: Return true if T is a reference type, or if it is an
187   // object type where an object of that type can be constructed from
188   // an rvalue of that type in both direct- and copy-initialization
189   // contexts.
190
191   template<typename T>
192   concept bool is_move_constructible =
193     is_constructible<T, T> and
194     is_convertible<T, T>;
195
196   // Type classification: cphmpl::is_copy_constructible<T> → bool
197
198   // Effect: Return true if T is an lvalue reference type, or if it is
199   // a move-constructible object type where an object of that type can
200   // be constructed from an lvalue (possibly const) or const rvalue of
201   // that type in both direct- and copy-initialization contexts.
202
203   template<typename T>
204   concept bool is_copy_constructible =
205     is_move_constructible<T> and
206     is_constructible<T, T&> and
207     is_constructible<T, T const&> and
208     is_constructible<T, T const> and
```

```
209     is_convertible<T&, T> and
210     is_convertible<T const&, T> and
211     is_convertible<T const, T>;
212
213   // Type classification: cphmpl::is_move_assignable<T> → bool
214
215   // Effect: Return false if T is not a referenceable type;
216   // otherwise, return std::is_assignable_v<T&, T&&>.
217
218   template<typename T>
219   concept bool is_move_assignable =
220     std::is_move_assignable_v<T>;
221
222   // Type classification: cphmpl::is_copy_assignable<T> → bool
223
224   // Effect: Return false if T is not a referenceable type;
225   // otherwise, return std::is_assignable_v<T&, T const&>.
226
227   template<typename T>
228   concept bool is_copy_assignable =
229     std::is_copy_assignable_v<T>;
230
231   // Type relationship: cphmpl::has_common_reference<T, U> → bool
232
233   // Effect: Return true if T and U share a common reference type
234   // (computed by std::common_reference_t) to which both can be
235   // converted.
236
237   template<typename T, typename U>
238   concept bool has_common_reference =
239     is_same<std::common_reference_t<T, U>, std::common_reference_t<U, T>> and
240     is_convertible<T, std::common_reference_t<T, U>> and
241     is_convertible<U, std::common_reference_t<T, U>>;
242
243   // Type relationship: cphmpl::is_assignable<T, U> → bool
244
245   // Effect: Return true if an expression of the type/value specified
246   // by U can be assigned to an lvalue expression whose type is
247   // specified by T.
248
249   template<typename T, typename U>
250   concept bool is_assignable =
251     std::is_lvalue_reference_v<T> and
252     has_common_reference<std::remove_reference_t<T> const&,
253                          std::remove_reference_t<U> const&> and
254     requires(T lhs, U&& rhs) {
255       requires is_same<decltype((lhs = std::forward<U>(rhs))), T>;
256   };
257
258   // Type classification: cphmpl::is_equality_comparable<T> → bool
259
260   // Effect: Return true if two objects of type T can be compared for
```

```
261   // equality with each other using both == and !=.
262
263   template<typename T>
264   concept bool is_equality_comparable =
265     requires(std::remove_reference_t<T> const& t,
266              std::remove_reference_t<T> const& u) {
267       requires is_convertible<decltype((t == u)), bool>;
268       requires is_convertible<decltype((u == t)), bool>;
269       requires is_convertible<decltype((t != u)), bool>;
270       requires is_convertible<decltype((u != t)), bool>;
271     };
272
273   // Type classification: cphmpl::is_swappable<T> → bool
274
275   // Effect: Return false if T is not referenceable; otherwise, return
276   // std::is_swappable_with_v<T&, T&>.
277
278   template<typename T>
279   concept bool is_swappable =
280     std::is_swappable_v<T>;
281
282   // Type classification: cphmpl::is_movable<T> → bool
283
284   // Effect: Return true if an object of type T can be moved (i.e. T
285   // supports move construction, move assignment, and lvalues of type
286   // T can be swapped).
287
288   template<typename T>
289   concept bool is_movable =
290     std::is_object_v<T> and
291     is_move_constructible<T> and
292     is_assignable<T&, T> and
293     is_swappable<T>;
294
295   // Type classification: cphmpl::is_copyable<T> → bool
296
297   // Effect: Return true if an object of type T can be moved and also
298   // copied (i.e. T supports copy construction and copy assignment).
299
300   template<typename T>
301   concept bool is_copyable =
302     is_copy_constructible<T> and
303     is_movable<T> and
304     is_assignable<T&, const T&>;
305
306   // Type classification: cphmpl::is_boolean<T> → bool
307
308   // Effect: Return true if T is usable in Boolean contexts. For
309   // is_boolean to be satisfied, the logical operators must have their
310   // usual behaviour.
311
312   template<typename B>
```

```
313   concept bool is_boolean =
314     is_movable<std::remove_cvref_t<B>> and
315     requires(std::remove_reference_t<B> const& b1,
316              std::remove_reference_t<B> const& b2, bool const a) {
317       requires is_convertible<decltype((b1)), bool>;
318       requires is_convertible<decltype((! b1)), bool>;
319       requires is_same<decltype((b1 && b2)), bool>;
320       requires is_same<decltype((b1 && a)), bool>;
321       requires is_same<decltype((a && b2)), bool>;
322       requires is_same<decltype((b1 || b2)), bool>;
323       requires is_same<decltype((b1 || a)), bool>;
324       requires is_same<decltype((a || b2)), bool>;
325       requires is_convertible<decltype((b1 == b2)), bool>;
326       requires is_convertible<decltype((b1 == a)), bool>;
327       requires is_convertible<decltype((a == b2)), bool>;
328       requires is_convertible<decltype((b1 != b2)), bool>;
329       requires is_convertible<decltype((b1 != a)), bool>;
330       requires is_convertible<decltype((a != b2)), bool>;
331     };
332
333   // Type classification: cphmpl::is_invocable<T, Args...> → bool
334
335   // Effect: Return true if a callable type T can be called with a set
336   // of argument types Args... using the function template
337   // std::invoke.
338
339   template<typename T, typename... Args>
340   concept bool is_invocable =
341     requires(T&& f, Args&&... args) {
342       std::invoke(std::forward<T>(f), std::forward<Args>(args)...);
343   };
344
345   // Type classification: cphmpl::is_predicate<T, Args...> → bool
346
347   // Effect: Return true if a callable type T can be called with a set
348   // of argument types Args... and the type of the return value is
349   // usable in Boolean contexts.
350
351   template<typename T, typename... Args>
352   concept bool is_predicate =
353     is_invocable<T, Args...> and
354     is_boolean<std::invoke_result_t<T, Args...>>;
355 }
356
357 // is_constant
358
359 namespace cphmpl::traits {
360
361   template<typename T>
362   class is_constant {
363   public:
364
```

```
365       static constexpr bool value = false;
366     };
367
368     template<bool B>
369     class is_constant<std::bool_constant<B>> {
370     public:
371
372       static constexpr bool value = true;
373     };
374
375     template<typename T, T v>
376     class is_constant<std::integral_constant<T, v>> {
377     public:
378
379       static constexpr bool value = true;
380     };
381
382     template<typename T>
383     requires requires {T::is_constant;}
384     class is_constant<T> {
385     public:
386
387       static constexpr bool value = T::is_constant;
388     };
389   }
390
391   namespace cphmpl {
392
393     // Type classification: cphmpl::is_constant<T> → bool
394
395     // Effect: Check if T is a class type that wraps a compile–time
396     // constant like std::integral_constant. A class type will qualify
397     // if it has the static variable T::is_constant which is set to
398     // true.
399
400     template<typename T>
401     concept bool is_constant = traits::is_constant<T>::value;
402   }
403
404   // is_built_in_character
405
406   namespace cphmpl::functions {
407
408     template<typename T>
409     constexpr bool is_built_in_character() noexcept {
410       using character_types = cphmpl::typelist<char, wchar_t, char8_t, char16_t,
      ↪ char32_t>;
411       using U = std::remove_cv_t<T>;
412       return character_types::is_member<U>;
413     }
414   }
415
```

```
416 namespace cphmpl {
417
418   // Type classification: cphmpl::is_built_in_character<T> → bool
419
420   // Effect: Check if T is a member of the typelist {char, wchar_t,
421   // char8_t, char16_t, char32_t}, or if it is any
422   // const/volatile-qualified version of them.
423
424   template<typename T>
425   concept bool is_built_in_character = functions::is_built_in_character<T>();
426 }
427
428 // is_character
429
430 namespace cphmpl::traits {
431
432   template<typename T>
433   class has_static_variable_is_character { // compiler wants SFINAE
434   public:
435     static constexpr bool value = false;
436   };
437
438   template<typename T>
439   requires requires {T::is_character;}
440   class has_static_variable_is_character<T> {
441   public:
442     static constexpr bool value = true;
443   };
444 }
445
446 namespace cphmpl::functions {
447
448   template<typename T>
449   constexpr bool is_character() noexcept {
450     if constexpr (cphmpl::is_built_in_character<T>) {
451       return true;
452     }
453     else if constexpr (traits::has_static_variable_is_character<T>::value) {
454       return T::is_character;
455     }
456     else {
457       return false;
458     }
459   }
460 }
461
462 namespace cphmpl {
463
464   // Type classification: cphmpl::is_character<T> → bool
465
466   // Effect: Check if T is a built-in character type or a class type
467   // that has the static variable T::is_character which is set to
```

```
468   // true.
469
470   template<typename T>
471   concept bool is_character = functions::is_character<T>();
472 }
473
474 // is_gnu_extension
475
476 namespace cphmpl::functions {
477
478   template<typename T>
479   constexpr bool is_gnu_extension() noexcept {
480     using gnu_extension_types = cphmpl::typelist<__int128, unsigned __int128>;
481     using U = std::remove_cv_t<T>;
482     return gnu_extension_types::is_member<U>;
483   }
484 }
485
486 namespace cphmpl {
487
488   // Type classification: cphmpl::is_gnu_extension<T> → bool
489
490   // Effect: Check if T is a member of the typelist {unsigned
491   // __int128, signed __int128}, or any const/volatile-qualified
492   // version of them.
493
494   template<typename T>
495   concept bool is_gnu_extension = functions::is_gnu_extension<T>();
496 }
497
498 // is_built_in_integer
499
500 namespace cphmpl::functions {
501
502   template<typename T>
503   constexpr bool is_built_in_integer() noexcept {
504     using integer_types = cphmpl::typelist<bool, unsigned  char, signed char, unsigned
      ↪ short int, signed short int, unsigned  int, signed int, unsigned long int, signed
      ↪ long int, unsigned long long int, signed long long int>;
505     using U = std::remove_cv_t<T>;
506     return integer_types::is_member<U>;
507   }
508 }
509
510 namespace cphmpl {
511
512   // Type classification: cphmpl::is_built_in_integer<T> → bool
513
514   // Effect: Check if T is a member of the typelist {bool, unsigned
515   // char, signed char, unsigned short int, signed short int, unsigned
516   // int, signed int, unsigned long int, signed long int, unsigned
517   // long long int, signed long long int}, or if it is any
```

```
518   // const/volatile-qualified version of them.
519
520   template<typename T>
521   concept bool is_built_in_integer = functions::is_built_in_integer<T>();
522 }
523
524 // is_integer
525
526 namespace cphmpl::traits {
527
528   template<typename T>
529   class has_static_variable_is_integer { // compiler wants SFINAE
530   public:
531     static constexpr bool value = false;
532   };
533
534   template<typename T>
535   requires requires {T::is_integer;}
536   class has_static_variable_is_integer<T> {
537   public:
538     static constexpr bool value = true;
539   };
540 }
541
542 namespace cphmpl::functions {
543
544   template<typename T>
545   constexpr bool is_integer() noexcept {
546     if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
547       return is_integer<std::remove_cv_t<T>>();
548     }
549     else if constexpr (std::is_arithmetic_v<T>) {
550       return std::numeric_limits<T>::is_integer;
551     }
552     else if constexpr (std::is_enum_v<T>) {
553       return std::is_integral_v<T>;
554     }
555     else if constexpr (cphmpl::is_gnu_extension<T>) {
556       return true;
557     }
558     else if constexpr (traits::has_static_variable_is_integer<T>::value) {
559       return T::is_integer;
560     }
561     else {
562       return false;
563     }
564   }
565 }
566
567 namespace cphmpl {
568
569   // Type classification: cphmpl::is_integer<T> → bool
```

```
570
571   // Effect: Check if T is a built-in integer type, an integral
572   // enumeration type, a GNU integer extension type, a class type that
573   // has the static variable T::is_integer which is set to true, or
574   // if it is any const/volatile-qualified version of them.
575
576   template<typename T>
577   concept bool is_integer = functions::is_integer<T>();
578 }
579
580 // is_built_in_floating_point
581
582 namespace cphmpl::functions {
583
584   template<typename T>
585   constexpr bool is_built_in_floating_point() noexcept {
586     using floating_point_types = cphmpl::typelist<float, double, long double>;
587     using U = std::remove_cv_t<T>;
588     return floating_point_types::is_member<U>;
589   }
590 }
591
592 namespace cphmpl {
593
594   // Type classification: cphmpl::is_built_in_floating_point<T> → bool
595
596   // Effect: Check if T is a member of the typelist {float, double,
597   // long double}, or if it is any const/volatile-qualified version of
598   // them.
599
600   template<typename T>
601   concept bool is_built_in_floating_point = functions::is_built_in_floating_point<T>();
602 }
603
604 // is_floating_point
605
606 namespace cphmpl::traits {
607
608   template<typename T>
609   class has_static_variable_is_floating_point { // compiler wants SFINAE
610   public:
611     static constexpr bool value = false;
612   };
613
614   template<typename T>
615   requires requires {T::is_floating_point;}
616   class has_static_variable_is_floating_point<T> {
617   public:
618     static constexpr bool value = true;
619   };
620 }
621
```

```
622 namespace cphmpl::functions {
623
624   template<typename T>
625   constexpr bool is_floating_point() noexcept {
626     if constexpr (cphmpl::is_built_in_floating_point<T>) {
627       return true;
628     }
629     else if constexpr (traits::has_static_variable_is_floating_point<T>::value) {
630       return T::is_floating_point;
631     }
632     else {
633       return false;
634     }
635   }
636 }
637
638 namespace cphmpl {
639
640   // Type classification: cphmpl::is_floating_point<T> → bool
641
642   // Effect: Check if T is a built-in floating-point type, a class
643   // type that has the static variable T::is_floating_point which is
644   // set to true, or if it is any const/volatile-qualified version of
645   // them.
646
647   template<typename T>
648   concept bool is_floating_point = functions::is_floating_point<T>();
649 }
650
651 // is_arithmetic
652
653 namespace cphmpl {
654
655   // Type classification: cphmpl::is_arithmetic<T> → bool
656
657   // Effect: Return true if T is an integer type or a floating-point
658   // type.
659
660   template<typename T>
661   concept bool is_arithmetic = is_integer<T> or is_floating_point<T>;
662 }
663
664 // is_unsigned
665
666 namespace cphmpl::traits {
667
668   template<typename T>
669   class has_static_variable_is_signed { // compiler wants SFINAE
670   public:
671     static constexpr bool value = false;
672   };
673
```

```
674   template<typename T>
675   requires requires {T::is_signed;}
676   class has_static_variable_is_signed<T> {
677   public:
678     static constexpr bool value = true;
679   };
680 }
681
682 namespace cphmpl::functions {
683
684   template<typename T>
685   constexpr bool is_unsigned() noexcept {
686     if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
687       return is_unsigned<std::remove_cv_t<T>>();
688     }
689     else if constexpr (std::is_same_v<T, bool>) {
690       return true;
691     }
692     else if constexpr (std::is_arithmetic_v<T>) {
693       return not std::numeric_limits<T>::is_signed;
694     }
695     else if constexpr (std::is_same_v<T, unsigned __int128>) {
696       return true;
697     }
698     else if constexpr (traits::has_static_variable_is_signed<T>::value) {
699       return not T::is_signed;
700     }
701     else {
702       return false;
703     }
704   }
705 }
706
707 namespace cphmpl {
708
709   // Type classification: cphmpl::is_unsigned<T> → bool
710
711   // Effect: Return true if T is an unsigned integer type. A class
712   // type will qualify if it has the static variable T::is_signed
713   // which is set to false.
714
715   template<typename T>
716   concept bool is_unsigned = functions::is_unsigned<T>();
717 }
718
719 // is_signed
720
721 namespace cphmpl::functions {
722
723   template<typename T>
724   constexpr bool is_signed() noexcept {
725     if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
```

```
726       return is_signed<std::remove_cv_t<T>>();
727     }
728     else if constexpr (std::is_same_v<T, bool>) {
729       return false;
730     }
731     else if constexpr (std::is_arithmetic_v<T>) {
732       return std::numeric_limits<T>::is_signed;
733     }
734     else if constexpr (std::is_same_v<T, signed __int128>) {
735       return true;
736     }
737     else if constexpr (traits::has_static_variable_is_signed<T>::value) {
738       return T::is_signed;
739     }
740     else {
741       return false;
742     }
743   }
744 }
745
746 namespace cphmpl {
747
748   // Type classification: cphmpl::is_signed<T> → bool
749
750   // Effect: Return true if T is a signed integer type. A class type
751   // will qualify if it has the static variable T::is_signed which is
752   // set to true.
753
754   template<typename T>
755   concept bool is_signed = functions::is_signed<T>();
756 }
757
758 // is_bounded
759
760 namespace cphmpl::traits {
761
762   template<typename T>
763   class has_static_variable_is_bounded { // compiler wants SFINAE
764   public:
765     static constexpr bool value = false;
766   };
767
768   template<typename T>
769   requires requires {T::is_bounded;}
770   class has_static_variable_is_bounded<T> {
771   public:
772     static constexpr bool value = true;
773   };
774 }
775
776 namespace cphmpl::functions {
777
```

```
778   template<typename T>
779   constexpr bool is_bounded() noexcept {
780     if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
781       return is_bounded<std::remove_cv_t<T>>();
782     }
783     else if constexpr (std::is_arithmetic_v<T>) {
784       return std::numeric_limits<T>::is_bounded;
785     }
786     else if constexpr (std::is_enum_v<T>) {
787       return true;
788     }
789     else if constexpr (cphmpl::is_gnu_extension<T>) {
790       return true;
791     }
792     else if constexpr (traits::has_static_variable_is_bounded<T>::value) {
793       return T::is_bounded;
794     }
795     else if constexpr (is_std_array<T>::value and is_bounded<typename
      ↪ T::value_type>()) {
796       return true;
797     }
798     else if constexpr (is_std_bitset<T>::value) {
799       return true;
800     }
801     else if constexpr (std::is_array_v<T> and
      ↪ is_bounded<std::remove_all_extents_t<T>>()) {
802       return true;
803     }
804     else {
805       return false;
806     }
807   }
808 }
809
810 namespace cphmpl {
811
812   // Type classification: cphmpl::is_bounded<T> → bool
813
814   // Effect: Return true if T is a type for which the value universe
815   // is finite. A class type will qualify if it has the static
816   // variable T::is_bounded which is set to true.
817
818   template<typename T>
819   concept bool is_bounded = functions::is_bounded<T>();
820 }
821
822 // is_exact
823
824 namespace cphmpl::traits {
825
826   template<typename T>
827   class has_static_variable_is_exact { // compiler wants SFINAE
```

```
828   public:
829     static constexpr bool value = false;
830   };
831
832   template<typename T>
833   requires requires {T::is_exact;}
834   class has_static_variable_is_exact<T> {
835   public:
836     static constexpr bool value = true;
837   };
838 }
839
840 namespace cphmpl::functions {
841
842   template<typename T>
843   constexpr bool is_exact() noexcept {
844     if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
845       return is_exact<std::remove_cv_t<T>>();
846     }
847     else if constexpr (std::is_arithmetic_v<T>) {
848       return std::numeric_limits<T>::is_exact;
849     }
850     else if constexpr (std::is_enum_v<T>) {
851       return true;
852     }
853     else if constexpr (cphmpl::is_gnu_extension<T>) {
854       return true;
855     }
856     else if constexpr (traits::has_static_variable_is_exact<T>::value) {
857       return T::is_exact;
858     }
859     else {
860       return false;
861     }
862   }
863 }
864
865 namespace cphmpl {
866
867   // Type classification: cphmpl::is_exact<T> → bool
868
869   // Effect: Return true if T is an arithmetic type for which the
870   // representation is exact. A class type will qualify if it has the
871   // static variable T::is_exact which is set to true.
872
873   template<typename T>
874   concept bool is_exact = functions::is_exact<T>();
875 }
876
877 // round_style
878
879 namespace cphmpl::traits {
```

```
880
881   template<typename T>
882   class has_static_variable_round_style { // compiler wants SFINAE
883   public:
884     static constexpr bool value = false;
885   };
886
887   template<typename T>
888   requires requires {T::round_style;}
889   class has_static_variable_round_style<T> {
890   public:
891     static constexpr bool value = true;
892   };
893 }
894
895 namespace cphmpl::functions {
896
897   template<typename T>
898   constexpr std::float_round_style round_style() noexcept {
899     if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
900       return round_style<std::remove_cv_t<T>>();
901     }
902     else if constexpr (std::is_arithmetic_v<T>) {
903       return std::numeric_limits<T>::round_style;
904     }
905     else if constexpr (cphmpl::is_gnu_extension<T>) {
906       return std::round_toward_zero;
907     }
908     else if constexpr (traits::has_static_variable_round_style<T>::value) {
909       return T::round_style;
910     }
911     else {
912       return std::round_indeterminate;
913     }
914   }
915 }
916
917 namespace cphmpl {
918
919   // Type property: cphmpl::round_style<T> → std::float_round_style
920
921   // Effect: Return the rounding style used by the arithmetic type T.
922   // For integers, the value should be std::round_toward_zero.
923
924   template<typename T>
925   requires cphmpl::is_arithmetic<T>
926   constexpr std::float_round_style round_style = functions::round_style<T>();
927 }
928
929 // is_modulo
930
931 namespace cphmpl::traits {
```

```
932
933   template<typename T>
934   class has_static_variable_is_modulo { // compiler wants SFINAE
935   public:
936     static constexpr bool value = false;
937   };
938
939   template<typename T>
940   requires requires {T::is_modulo;}
941   class has_static_variable_is_modulo<T> {
942   public:
943     static constexpr bool value = true;
944   };
945 }
946
947 namespace cphmpl::functions {
948
949   template<typename T>
950   constexpr bool is_modulo() noexcept {
951     if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
952       return is_modulo<std::remove_cv_t<T>>();
953     }
954     else if constexpr (std::is_arithmetic_v<T>) {
955       return std::numeric_limits<T>::is_modulo;
956     }
957     else if constexpr (std::is_same_v<T, unsigned __int128>) {
958       return true;
959     }
960     else if constexpr (traits::has_static_variable_is_modulo<T>::value) {
961       return T::is_modulo;
962     }
963     else {
964       return false;
965     }
966   }
967 }
968
969 namespace cphmpl {
970
971   // Type classification: cphmpl::is_modulo<T> → bool
972
973   // Effect: Return true if T is an aritmetic type which handles
974   // overflows with modulo arithmetic. A class type will qualify if it
975   // has the static variable T::is_modulo which is set to true.
976
977   template<typename T>
978   concept bool is_modulo = functions::is_modulo<T>();
979 }
980
981 // radix
982
983 namespace cphmpl::traits {
```

```
984
985   template<typename T>
986   class has_static_variable_radix { // compiler wants SFINAE
987   public:
988     static constexpr bool value = false;
989   };
990
991   template<typename T>
992   requires requires {T::radix;}
993   class has_static_variable_radix<T> {
994   public:
995     static constexpr bool value = true;
996   };
997 }
998
999 namespace cphmpl::functions {
1000
1001   template<typename T>
1002   constexpr auto radix() noexcept {
1003     if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
1004       return radix<std::remove_cv_t<T>>();
1005     }
1006     else if constexpr (std::is_arithmetic_v<T>) {
1007       return std::numeric_limits<T>::radix;
1008     }
1009     else if constexpr (cphmpl::is_gnu_extension<T>) {
1010       return std::size_t(2);
1011     }
1012     else if constexpr (traits::has_static_variable_radix<T>::value) {
1013       return T::radix;
1014     }
1015     else {
1016       return indeterminate;
1017     }
1018   }
1019 }
1020
1021 namespace cphmpl {
1022
1023   // Type property: cphmpl::radix<T> → integer
1024
1025   // Effect: Return the base of the number system used in the
1026   // representation of the arithmetic type T.  This will work for a
1027   // class type provided that it has the static variable T::radix
1028   // which records the base.
1029
1030   template<typename T>
1031   constexpr auto radix = functions::radix<T>();
1032 }
1033
1034 // is_iterator
1035
```

```
1036 namespace cphmpl {
1037
1038   template<typename I>
1039   concept bool is_input_or_output_iterator =
1040     is_default_constructible<I> and
1041     is_movable<I> and
1042     is_dereferenceable<I> and
1043     requires(I i) {
1044       typename std::iterator_traits<I>::difference_type;
1045       requires cphmpl::is_signed<typename std::iterator_traits<I>::difference_type>;
1046       requires is_same<decltype((++i)), I&>;
1047       i++;
1048     };
1049
1050   // Type classification: cphmpl::is_iterator<T> → bool
1051
1052   // Effect: Return true if T is an iterator type fulfilling the
1053   // requirements specified in the C++20 standard.
1054
1055   template<typename I>
1056   concept bool is_iterator =
1057     is_input_or_output_iterator<I> or std::is_pointer_v<I>;
1058 }
1059
1060 // is_tuple
1061
1062 namespace cphmpl::traits {
1063
1064   template<typename T>
1065   class is_tuple {
1066   public:
1067
1068     static constexpr bool value = false;
1069   };
1070
1071   template<typename X, typename Y>
1072   class is_tuple<std::pair<X, Y>> {
1073   public:
1074
1075     static constexpr bool value = true;
1076   };
1077
1078   template<typename... T>
1079   class is_tuple<std::tuple<T...>> {
1080   public:
1081
1082     static constexpr bool value = true;
1083   };
1084 }
1085
1086 namespace cphmpl {
1087
```

```
1088    // Type classification: cphmpl::is_tuple<T> → bool
1089
1090    // Effect: Return true if T specifies a tuple with some unspecified
1091    // number of elements.
1092
1093    template<typename T>
1094    concept bool is_tuple = traits::is_tuple<std::remove_cv_t<T>>::value;
1095  }
1096
1097  // is_pair
1098
1099  namespace cphmpl {
1100
1101    // Type classification: cphmpl::is_pair<T> → bool
1102
1103    // Effect: Return true if T specifies a two–element pair.
1104
1105    template<typename T>
1106    concept bool is_pair = is_tuple<T> and std::tuple_size<T>::value == 2;
1107  }
1108
1109  // is_bitset
1110
1111  namespace cphmpl::traits {
1112
1113    template<typename T>
1114    class is_bitset {
1115    public:
1116
1117      static constexpr bool value = false;
1118    };
1119
1120    template<std::size_t N>
1121    class is_bitset<std::bitset<N>> {
1122    public:
1123
1124      static constexpr bool value = true;
1125    };
1126
1127    template<typename T>
1128    requires requires {T::is_bitset;}
1129    class is_bitset<T> {
1130    public:
1131
1132      static constexpr bool value = T::is_bitset;
1133    };
1134  }
1135
1136  namespace cphmpl {
1137
1138    // Type classification: cphmpl::is_bitset<T> → bool
1139
```

```
1140    // Effect: Return true if T is similar to the bitset specified in
1141    // the C++20 standard. This will work for a class type provided that
1142    // it has the static variable T::is_bitset which is set to true.
1143
1144    template<typename T>
1145    concept bool is_bitset = traits::is_bitset<std::remove_cv_t<T>>::value;
1146  }
1147
1148  // specifies_range
1149
1150  namespace cphmpl {
1151
1152    // Type classification: cphmpl::specifies_range<T> → bool
1153
1154    // Effect: Check if T has the type members value_type, reference,
1155    // const_reference, iterator, const_iterator, difference_type, and
1156    // size_type; and if T is accepted as an argument by the functions
1157    // std::begin, std::cbegin, std::end, std::cend, std::size, and
1158    // std::empty.
1159
1160    template<typename X>
1161    concept bool specifies_range =
1162      requires(std::remove_const_t<X> t, std::add_const_t<X> c) {
1163        typename X::value_type;
1164        typename X::reference;
1165        typename X::const_reference;
1166        typename X::iterator;
1167        typename X::const_iterator;
1168        typename X::size_type;
1169        typename X::difference_type;
1170        std::begin(t);
1171        std::begin(c);
1172        std::end(t);
1173        std::end(c);
1174        std::cbegin(c);
1175        std::cend(c);
1176        std::size(c);
1177        std::empty(c);
1178      };
1179  }
1180
1181  // is_container
1182
1183  namespace cphmpl {
1184
1185    // Type classification: cphmpl::is_container<T> → bool
1186
1187    // Effect: Return true if T fulfils the container requirements
1188    // specified in the C++20 standard.
1189
1190    template<typename T>
1191    concept bool is_container =
```

```
1192      is_default_constructible<T> and
1193      is_move_constructible<T> and
1194      is_copy_constructible<T> and
1195      is_destructible<T> and
1196      is_move_assignable<T> and
1197      is_copy_assignable<T> and
1198      is_assignable<T&, T const&> and
1199      is_swappable<T> and
1200      is_equality_comparable<T> and
1201      requires (std::remove_const_t<T> a, std::remove_const_t<T> b, T const c) {
1202        typename T::value_type;
1203        typename T::reference;
1204        typename T::const_reference;
1205        typename T::iterator;
1206        typename T::const_iterator;
1207        typename T::difference_type;
1208        typename T::size_type;
1209        requires is_same<decltype((a.begin())), typename T::iterator>;
1210        requires is_same<decltype((c.begin())), typename T::const_iterator>;
1211        requires is_same<decltype((a.end())), typename T::iterator>;
1212        requires is_same<decltype((c.end())), typename T::const_iterator>;
1213        requires is_same<decltype((a.cbegin())), typename T::const_iterator>;
1214        requires is_same<decltype((a.cend())), typename T::const_iterator>;
1215        a.swap(b);
1216        requires is_same<decltype((a.size())), typename T::size_type>;
1217        requires is_same<decltype((a.max_size())), typename T::size_type>;
1218        requires is_convertible<decltype((a.empty())), bool>;
1219      };
1220  }
1221
1222  // is_slice
1223
1224  namespace cphmpl {
1225
1226    // Type classification: cphmpl::is_slice<T> → bool
1227
1228    // Effect: Return true if T specifies an iterator range, but T is not
1229    // a container. Often a slice is specified by a pair of iterators.
1230
1231    template<typename T>
1232    concept bool is_slice = specifies_range<T> and not is_container<T>;
1233  }
1234
1235  // value
1236
1237  namespace cphmpl::traits {
1238
1239    template<typename T>
1240    class has_type_member_value_type { // compiler wants SFINAE
1241    public:
1242      static constexpr bool value = false;
1243    };
```

```
1244
1245    template<typename T>
1246    requires requires {typename T::value_type;}
1247    class has_type_member_value_type<T> {
1248    public:
1249      static constexpr bool value = true;
1250    };
1251  }
1252
1253  namespace cphmpl::functions {
1254
1255    template<typename T>
1256    constexpr auto value() noexcept {
1257      if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
1258        return value<std::remove_cv_t<T>>();
1259      }
1260      else if constexpr (std::is_reference_v<T>) {
1261        return value<std::remove_reference_t<T>>();
1262      }
1263      else if constexpr (std::is_array_v<T>) {
1264        return value<std::remove_all_extents_t<T>>();
1265      }
1266      else if constexpr (cphmpl::is_iterator<T>) {
1267        return capsule<typename std::iterator_traits<T>::value_type>();
1268      }
1269      else if constexpr (cphmpl::is_bitset<T> and ! cphmpl::specifies_range<T>) {
1270        return capsule<bool>();
1271      }
1272      else if constexpr (traits::has_type_member_value_type<T>::value) {
1273        return capsule<typename T::value_type>();
1274      }
1275      else { // default
1276        return capsule<T>();
1277      }
1278    }
1279  }
1280
1281  namespace cphmpl {
1282
1283    // Type association: cphmpl::value<T> → typename
1284
1285    // Effect: Provide an alias for the type of the value(s) stored in
1286    // an object specified by type T. For a class type the result is
1287    // T::value_type if this type member exists.
1288
1289    template<typename T>
1290    using value = typename decltype(functions::value<T>())::type;
1291  }
1292
1293  // reference
1294
1295  namespace cphmpl::functions {
```

```
1296
1297    template<typename T>
1298    constexpr auto reference() noexcept {
1299      if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
1300        return reference<std::remove_cv_t<T>>();
1301      }
1302      else if constexpr (std::is_reference_v<T>) {
1303        return reference<std::remove_reference_t<T>>();
1304      }
1305      else if constexpr (cphmpl::is_iterator<T>) {
1306        using R = decltype(*std::declval<T&>());
1307        return capsule<R>();
1308      }
1309      else if constexpr (std::is_array_v<T>) {
1310        using R = std::add_lvalue_reference_t<cphmpl::value<T>>;
1311        return capsule<R>();
1312      }
1313      else if constexpr (requires {typename T::reference;}) {
1314        using R = typename T::reference;
1315        return capsule<R>();
1316      }
1317      else { // default
1318        return capsule<std::add_lvalue_reference_t<T>>();
1319      }
1320    }
1321  }
1322
1323  namespace cphmpl {
1324
1325    // Type association: cphmpl::reference<T> → typename
1326
1327    // Effect: Provide an alias for the reference type referring to the
1328    // value(s) specified by type T. For a class type the result is
1329    // T::reference if this type member exists.
1330
1331    template<typename T>
1332    using reference = typename decltype(functions::reference<T>())::type;
1333  }
1334
1335  // const_reference
1336
1337  namespace cphmpl::functions {
1338
1339    template<typename T>
1340    constexpr auto const_reference() noexcept {
1341      if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
1342        return const_reference<std::remove_cv_t<T>>();
1343      }
1344      else if constexpr (std::is_reference_v<T>) {
1345        return const_reference<std::remove_reference_t<T>>();
1346      }
1347      else if constexpr (cphmpl::is_iterator<T>) {
```

```
1348        using R = decltype(*std::declval<std::add_const_t<T>&>());
1349        return capsule<R>();
1350      }
1351      else if constexpr (std::is_array_v<T>) {
1352        using V = cphmpl::value<T>;
1353        using R = std::add_lvalue_reference_t<std::add_const_t<V>>;
1354        return capsule<R>();
1355      }
1356      else if constexpr (requires {typename T::const_reference;}) {
1357        using R = typename T::const_reference;
1358        return capsule<R>();
1359      }
1360      else if constexpr (cphmpl::is_bitset<T>) {
1361        return capsule<bool>();
1362      }
1363      else { // default
1364        return capsule<std::add_lvalue_reference_t<std::add_const_t<T>>>();
1365      }
1366    }
1367  }
1368
1369  namespace cphmpl {
1370
1371    // Type association: cphmpl::const_reference<T> → typename
1372
1373    // Effect: Provide an alias for the const–reference type referring
1374    // to the value(s) specified by type T. For a class type the result
1375    // is T::const_reference if this type member exists.
1376
1377    template<typename T>
1378    using const_reference = typename decltype(functions::const_reference<T>())::type;
1379  }
1380
1381  // rvalue_reference
1382
1383  namespace cphmpl::functions {
1384
1385    template<typename T>
1386    constexpr auto rvalue_reference() noexcept {
1387      if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
1388        return rvalue_reference<std::remove_cv_t<T>>();
1389      }
1390      else if constexpr (std::is_reference_v<T>) {
1391        return rvalue_reference<std::remove_reference_t<T>>();
1392      }
1393      else if constexpr (cphmpl::is_iterator<T>) {
1394        using R = decltype(std::move(*std::declval<T&>()));
1395        return capsule<R>();
1396      }
1397      else if constexpr (std::is_array_v<T>) {
1398        using V = cphmpl::value<T>;
1399        using R = std::add_lvalue_reference_t<V>;
```

```
1400        return capsule<R>();
1401      }
1402      else if constexpr (requires {typename T::rvalue_reference;}) {
1403        using R = typename T::rvalue_reference;
1404        return capsule<R>();
1405      }
1406      else if constexpr (requires {typename T::value_type;}) {
1407        using V = typename T::value_type;
1408        using R = std::add_rvalue_reference_t<V>;
1409        return capsule<R>();
1410      }
1411      else { // default
1412        return capsule<std::add_rvalue_reference_t<T>>();
1413      }
1414    }
1415  }
1416
1417  namespace cphmpl {
1418
1419    // Type association: cphmpl::rvalue_reference<T> → typename
1420
1421    // Effect: Provide an alias for the rvalue–reference type referring
1422    // to the value(s) specified by type T. For a class type the result
1423    // is T::rvalue_reference if this type member exists.
1424
1425    template<typename T>
1426    using rvalue_reference = typename decltype(functions::rvalue_reference<T>())::type;
1427  }
1428
1429  // pointer
1430
1431  namespace cphmpl::functions {
1432
1433    template<typename T>
1434    constexpr auto pointer() noexcept {
1435      if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
1436        return pointer<std::remove_cv_t<T>>();
1437      }
1438      else if constexpr (std::is_reference_v<T>) {
1439        return pointer<std::remove_reference_t<T>>();
1440      }
1441      else if constexpr (cphmpl::is_iterator<T>) {
1442        using P = decltype(std::declval<T&>().operator→());
1443        return capsule<P>();
1444      }
1445      else if constexpr (std::is_array_v<T>) {
1446        using P = std::add_pointer_t<cphmpl::value<T>>;
1447        return capsule<P>();
1448      }
1449      else if constexpr (requires {typename T::pointer;}) {
1450        using P = typename T::pointer;
1451        return capsule<P>();
```

```
1452      }
1453      else { // default
1454        return capsule<std::add_pointer_t<T>>();
1455      }
1456    }
1457  }
1458
1459  namespace cphmpl {
1460
1461    // Type association: cphmpl::pointer<T> → typename
1462
1463    // Effect: Provide an alias for the pointer type pointing to the
1464    // value(s) specified by type T. For a class type the result is
1465    // T::pointer if this type member exists.
1466
1467    template<typename T>
1468    using pointer = typename decltype(functions::pointer<T>())::type;
1469  }
1470
1471  // const_pointer
1472
1473  namespace cphmpl::functions {
1474
1475    template<typename T>
1476    constexpr auto const_pointer() noexcept {
1477      if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
1478        return const_pointer<std::remove_cv_t<T>>();
1479      }
1480      else if constexpr (std::is_reference_v<T>) {
1481        return const_pointer<std::remove_reference_t<T>>();
1482      }
1483      else if constexpr (cphmpl::is_iterator<T>) {
1484        using P = decltype(std::declval<std::add_pointer_t<T>&>().operator→());
1485        return capsule<P>();
1486      }
1487      else if constexpr (std::is_array_v<T>) {
1488        using P = std::add_pointer_t<std::add_const_t<cphmpl::value<T>>>;
1489        return capsule<P>();
1490      }
1491      else if constexpr (requires {typename T::const_pointer;}) {
1492        using P = typename T::const_pointer;
1493        return capsule<P>();
1494      }
1495      else { // default
1496        return capsule<std::add_pointer_t<std::add_const_t<T>>>();
1497      }
1498    }
1499  }
1500
1501  namespace cphmpl {
1502
1503    // Type association: cphmpl::const_pointer<T> → typename
```

```
1504
1505    // Effect: Provide an alias for the const-pointer type pointing to
1506    // the value(s) specified by type T. For a class type the result is
1507    // T::const_pointer if this type member exists.
1508
1509    template<typename T>
1510    using const_pointer = typename decltype(functions::const_pointer<T>())::type;
1511  }
1512
1513  // cell
1514
1515  namespace cphmpl::functions {
1516
1517    template<typename T>
1518    constexpr auto cell() noexcept {
1519      if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
1520        return cell<std::remove_cv_t<T>>();
1521      }
1522      else if constexpr (std::is_reference_v<T>) {
1523        return capsule<std::remove_reference_t<T>>();
1524      }
1525      else if constexpr (std::is_pointer_v<T>) {
1526        using R = typename std::pointer_traits<T>::element_type;
1527        return capsule<R>();
1528      }
1529      else if constexpr (cphmpl::is_iterator<T>) {
1530        using R = typename std::iterator_traits<T>::value_type;
1531        return capsule<R>();
1532      }
1533      else if constexpr (std::is_array_v<T>) {
1534        return capsule<cphmpl::value<T>>();
1535      }
1536      else if constexpr (requires {typename T::cell;}) {
1537        using R = typename T::cell;
1538        return capsule<R>();
1539      }
1540      else if constexpr (requires {typename T::node_type;}) {
1541        using R = typename T::node_type;
1542        return capsule<R>();
1543      }
1544      else { // default
1545        return capsule<T>();
1546      }
1547    }
1548  }
1549
1550  namespace cphmpl {
1551
1552    // Type association: cphmpl::cell<T> → typename
1553
1554    // Effect: Return an alias for the type of cells used for storing
1555    // the values specified by type T. For a class type the result is
```

```
1556    // T::cell or T::node_type if either one of these type members
1557    // exists.
1558
1559    template<typename T>
1560    using cell = typename decltype(functions::cell<T>())::type;
1561  }
1562
1563  // key
1564
1565  namespace cphmpl::functions {
1566
1567    template<typename T>
1568    constexpr auto key() noexcept {
1569      if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
1570        return key<std::remove_cv_t<T>>();
1571      }
1572      else if constexpr (std::is_reference_v<T>) {
1573        return key<std::remove_reference_t<T>>();
1574      }
1575      else if constexpr (std::is_pointer_v<T>) {
1576        return key<std::remove_pointer_t<T>>();
1577      }
1578      else if constexpr (cphmpl::is_iterator<T>) {
1579        if constexpr (requires {typename T::owner;}) {
1580          using Collection = typename T::owner;
1581          return key<Collection>();
1582        }
1583        else {
1584          using R = typename std::iterator_traits<T>::value_type;
1585          return key<R>();
1586        }
1587      }
1588      else if constexpr (std::is_array_v<T>) {
1589        return key<cphmpl::value<T>>();
1590      }
1591      else if constexpr (cphmpl::is_pair<T>) {
1592        using R = typename std::tuple_element<0, T>::type;
1593        return capsule<R>();
1594      }
1595      else if constexpr (requires {typename T::key_type;}) {
1596        using R = typename T::key_type;
1597        return capsule<R>();
1598      }
1599      else { // default
1600        return capsule<void>();
1601      }
1602    }
1603  }
1604
1605  namespace cphmpl {
1606
1607    // Type association: cphmpl::key<T> → typename
```

```
1608
1609    // Effect: Return an alias for the type of keys in the value(s)
1610    // specified by type T, i.e. the values are (key, mapped) pairs. For
1611    // a class type the result is T::key_type if this type member
1612    // exists.
1613
1614    template<typename T>
1615    using key = typename decltype(functions::key<T>())::type;
1616  }
1617
1618  // mapped
1619
1620  namespace cphmpl::functions {
1621
1622    template<typename T>
1623    constexpr auto mapped() noexcept {
1624      if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
1625        return mapped<std::remove_cv_t<T>>();
1626      }
1627      else if constexpr (std::is_reference_v<T>) {
1628        return mapped<std::remove_reference_t<T>>();
1629      }
1630      else if constexpr (std::is_pointer_v<T>) {
1631        return mapped<std::remove_pointer_t<T>>();
1632      }
1633      else if constexpr (cphmpl::is_iterator<T>) {
1634        if constexpr (requires {typename T::owner;}) {
1635          using Collection = typename T::owner;
1636          return mapped<Collection>();
1637        }
1638        else {
1639          using V = typename std::iterator_traits<T>::value_type;
1640          return mapped<V>();
1641        }
1642      }
1643      else if constexpr (std::is_array_v<T>) {
1644        return mapped<cphmpl::value<T>>();
1645      }
1646      else if constexpr (cphmpl::is_pair<T>) {
1647        using R = typename std::tuple_element<1, T>::type;
1648        return capsule<R>();
1649      }
1650      else if constexpr (requires {typename T::mapped_type;}) {
1651        using R = typename T::mapped_type;
1652        return capsule<R>();
1653      }
1654      else { // default
1655        return capsule<void>();
1656      }
1657    }
1658  }
1659
```

```
1660 namespace cphmpl {
1661
1662   // Type association: cphmpl::mapped<T> → typename
1663
1664   // Effect: Return an alias for the mapped type in the value(s)
1665   // specified by type T, i.e. the values are (key, mapped) pairs. For
1666   // a class type the result is T::mapped_type if this type member
1667   // exists.
1668
1669   template<typename T>
1670   using mapped = typename decltype(functions::mapped<T>())::type;
1671 }
1672
1673 // is_const_reference
1674
1675 namespace cphmpl {
1676
1677   // Type classification: cphmpl::is_const_reference<T> → bool
1678
1679   // Effect: Return true if T is a reference type and the value
1680   // referred to cannot be modified.
1681
1682   template<typename T>
1683   concept bool is_const_reference =
1684     std::is_reference_v<T> and
1685     std::is_const_v<std::remove_reference_t<T>>;
1686 }
1687
1688 // is_const_iterator
1689
1690 namespace cphmpl {
1691
1692   // Type classification: cphmpl::is_const_iterator<T> → bool
1693
1694   // Effect: Return true if T is an iterator type and the value
1695   // referred to cannot be modified.
1696
1697   template<typename T>
1698   concept bool is_const_iterator =
1699     is_iterator<T> and
1700     std::is_const_v<std::remove_pointer_t<typename
       ↪ std::iterator_traits<T>::pointer>>;
1701 }
1702
1703 // allows_bit_view
1704
1705 namespace cphmpl {
1706
1707   // Type classification: cphmpl::allows_bit_view<T> → bool
1708
1709   // Effect: Check if objects of type T as a whole can be seen as
1710   // a string of bits. This is true for constants, integers, bitsets,
```

```
1711    // and collections storing bounded values.
1712
1713    template<typename T>
1714    concept bool allows_bit_view = (
1715      cphmpl::is_constant<T> or
1716      cphmpl::is_integer<T> or
1717      cphmpl::is_bitset<T> or
1718      (cphmpl::specifies_range<T> and cphmpl::is_bounded<cphmpl::value<T>>));
1719  }
1720
1721  // iterator
1722
1723  namespace cphmpl::functions {
1724
1725    template<typename T>
1726    constexpr auto iterator() noexcept {
1727      if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
1728        return iterator<std::remove_cv_t<T>>();
1729      }
1730      else if constexpr (std::is_reference_v<T>) {
1731        return iterator<std::remove_reference_t<T>>();
1732      }
1733      else if constexpr (std::is_array_v<T>) {
1734        using V = std::remove_all_extents_t<T>;
1735        using P = std::add_pointer_t<V>;
1736        return capsule<P>();
1737      }
1738      else if constexpr (cphmpl::is_iterator<T> and not cphmpl::is_const_iterator<T>) {
1739        return capsule<T>();
1740      }
1741      else if constexpr (requires {typename T::iterator;}) {
1742        using R = typename T::iterator;
1743        return capsule<R>();
1744      }
1745      else if constexpr (cphmpl::is_bitset<T>) {
1746        using R = cphstl::rank_iterator<T>;
1747        return capsule<R>();
1748      }
1749      else { // default
1750        using V = std::remove_cv_t<T>;
1751        using R = cphstl::singleton_iterator<V>;
1752        return capsule<R>();
1753      }
1754    }
1755  }
1756
1757  namespace cphmpl {
1758
1759    // Type association: cphmpl::iterator<T> → typename
1760
1761    // Effect: Return an alias for the iterator type associated with
1762    // type T. For a class type the result is T::iterator if this type
```

```
1763    // member exists.
1764
1765    // Observe: This function provides iterators even for types like int
1766    // and std::bitset that do not have any iterators themselves. You
1767    // need the file cphstl/iterators.h++ for this to work.
1768
1769    template<typename T>
1770    using iterator = typename decltype(functions::iterator<T>())::type;
1771  }
1772
1773  // const_iterator
1774
1775  namespace cphmpl::traits {
1776
1777    template<typename T>
1778    class has_type_member_const_iterator { // compiler wants SFINAE
1779    public:
1780      static constexpr bool value = false;
1781    };
1782
1783    template<typename T>
1784    requires requires {typename T::const_iterator;}
1785    class has_type_member_const_iterator<T> {
1786    public:
1787      static constexpr bool value = true;
1788    };
1789  }
1790
1791  namespace cphmpl::functions {
1792
1793    template<typename T>
1794    constexpr auto const_iterator() noexcept {
1795      if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
1796        return const_iterator<std::remove_cv_t<T>>();
1797      }
1798      else if constexpr (std::is_reference_v<T>) {
1799        return const_iterator<std::remove_reference_t<T>>();
1800      }
1801      else if constexpr (std::is_pointer_v<T>) {
1802        using V = std::remove_pointer_t<T>;
1803        using C = std::add_const_t<V>;
1804        using P = std::add_pointer_t<C>;
1805        return capsule<P>();
1806      }
1807      else if constexpr (std::is_array_v<T>) {
1808        using V = std::remove_all_extents_t<T>;
1809        using C = std::add_const_t<V>;
1810        using P = std::add_pointer_t<C>;
1811        return capsule<P>();
1812      }
1813      else if constexpr (cphmpl::is_const_iterator<T>) {
1814        return capsule<T>();
```

```
1815      }
1816      else if constexpr (traits::has_type_member_const_iterator<T>::value) {
1817        using R = typename T::const_iterator;
1818        return capsule<R>();
1819      }
1820      else if constexpr (cphmpl::is_bitset<T>) {
1821        using C = std::add_const_t<T>;
1822        using R = cphstl::rank_iterator<C>;
1823        return capsule<R>();
1824      }
1825      else { // default
1826        using C = std::add_const_t<T>;
1827        using R = cphstl::singleton_iterator<C>;
1828        return capsule<R>();
1829      }
1830    }
1831  }
1832
1833  namespace cphmpl {
1834
1835    // Type association: cphmpl::const_iterator<T> → typename
1836
1837    // Effect: Return an alias for the const-iterator type associated
1838    // with type T. For a class type the result is T::const_iterator if
1839    // this type member exists.
1840
1841    template<typename T>
1842    using const_iterator = typename decltype(functions::const_iterator<T>())::type;
1843  }
1844
1845  // category
1846
1847  namespace cphmpl::functions {
1848
1849    template<typename T>
1850    constexpr auto category() noexcept {
1851      if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
1852        return category<std::remove_cv_t<T>>();
1853      }
1854      else if constexpr (std::is_reference_v<T>) {
1855        return category<std::remove_reference_t<T>>();
1856      }
1857      else if constexpr (std::is_pointer_v<T>) {
1858        using R = typename std::iterator_traits<T>::iterator_category;
1859        return capsule<R>();
1860      }
1861      else if constexpr (std::is_array_v<T>) {
1862        using V = std::remove_all_extents_t<T>;
1863        using P = std::add_pointer_t<V>;
1864        using R = typename std::iterator_traits<P>::iterator_category;
1865        return capsule<R>();
1866      }
```

```
1867      else if constexpr (cphmpl::is_iterator<T>) {
1868        using R = typename std::iterator_traits<T>::iterator_category;
1869        return capsule<R>();
1870      }
1871      else if constexpr (requires {typename T::iterator_category;}) {
1872        using R = typename T::iterator_category;
1873        return capsule<R>();
1874      }
1875      else if constexpr (requires {typename T::iterator;}) {
1876        using I = typename T::iterator;
1877        using R = typename std::iterator_traits<I>::iterator_category;
1878        return capsule<R>();
1879      }
1880      else if constexpr (cphmpl::is_bitset<T>) {
1881        using I = cphstl::rank_iterator<T>;
1882        using R = typename std::iterator_traits<I>::iterator_category;
1883        return capsule<R>();
1884      }
1885      else { // default
1886        return capsule<void>();
1887      }
1888    }
1889  }
1890
1891  namespace cphmpl {
1892
1893    // Type association: cphmpl::category<T> → typename
1894
1895    // Effect: Return the category of iterators associated with type T.
1896    // For a class type the result is T::iterator_category if this type
1897    // member exists.
1898
1899    template<typename T>
1900    using category = typename decltype(functions::category<T>())::type;
1901  }
1902
1903  // make_unsigned
1904
1905  namespace cphmpl::traits {
1906
1907    template<typename T>
1908    class has_type_member_companion { // compiler wants SFINAE
1909    public:
1910      static constexpr bool value = false;
1911    };
1912
1913    template<typename T>
1914    requires requires {typename T::companion;}
1915    class has_type_member_companion<T> {
1916    public:
1917      static constexpr bool value = true;
1918    };
```

```
1919 }
1920
1921 namespace cphmpl::functions {
1922
1923   template<typename T>
1924   constexpr auto make_unsigned() noexcept {
1925     using U = std::remove_cv_t<T>;
1926     if constexpr (cphmpl::is_unsigned<T>) {
1927       return capsule<T>();
1928     }
1929     else if constexpr (cphmpl::is_built_in_integer<T>) {
1930       return capsule<std::make_unsigned_t<T>>();
1931     }
1932     else if constexpr (std::is_enum_v<T>) {
1933       return capsule<std::make_unsigned_t<T>>();
1934     }
1935     else if constexpr (cphmpl::is_built_in_character<T>) {
1936       return capsule<std::make_unsigned_t<T>>();
1937     }
1938     else if constexpr (std::is_same_v<U, signed __int128>) {
1939       using X = unsigned __int128;
1940       using Y = std::conditional_t<std::is_const_v<T>, std::add_const_t<X>, X>;
1941       using Z = std::conditional_t<std::is_volatile_v<T>, std::add_volatile_t<Y>,
      ↪ Y>;
1942       return capsule<Z>();
1943     }
1944     else if constexpr (traits::has_type_member_companion<T>::value) {
1945       using X = typename T::companion;
1946       using Y = std::conditional_t<std::is_const_v<T>, std::add_const_t<X>, X>;
1947       using Z = std::conditional_t<std::is_volatile_v<T>, std::add_volatile_t<Y>,
      ↪ Y>;
1948       return capsule<Z>();
1949     }
1950     else { // default
1951       return capsule<void>();
1952     }
1953   }
1954 }
1955
1956 namespace cphmpl {
1957
1958   // Type construction: cphmpl::make_unsigned<T> → typename
1959
1960   // Effect: Return the unsigned integer type corresponding to type T,
1961   // with the same const/volatile qualifiers.  This will work for a
1962   // class type provided that it knows its unsigned companion
1963   // specified by the type member T::companion.
1964
1965   template<typename T>
1966   using make_unsigned = typename decltype(functions::make_unsigned<T>())::type;
1967 }
1968
```

```
1969 // make_signed
1970
1971 namespace cphmpl::functions {
1972
1973   template<typename T>
1974   constexpr auto make_signed() noexcept {
1975     using U = std::remove_cv_t<T>;
1976     if constexpr (cphmpl::is_signed<T>) {
1977       return capsule<T>();
1978     }
1979     else if constexpr (cphmpl::is_built_in_integer<T>) {
1980       return capsule<std::make_signed_t<T>>();
1981     }
1982     else if constexpr (std::is_enum_v<T>) {
1983       return capsule<std::make_signed_t<T>>();
1984     }
1985     else if constexpr (cphmpl::is_built_in_character<T>) {
1986       return capsule<std::make_signed_t<T>>();
1987     }
1988     else if constexpr (std::is_same_v<U, unsigned __int128>) {
1989       using X = signed __int128;
1990       using Y = std::conditional_t<std::is_const_v<T>, std::add_const_t<X>, X>;
1991       using Z = std::conditional_t<std::is_volatile_v<T>, std::add_volatile_t<Y>,
    ↪ Y>;
1992       return capsule<Z>();
1993     }
1994     else if constexpr (traits::has_type_member_companion<T>::value) {
1995       using X = typename T::companion;
1996       using Y = std::conditional_t<std::is_const_v<T>, std::add_const_t<X>, X>;
1997       using Z = std::conditional_t<std::is_volatile_v<T>, std::add_volatile_t<Y>,
    ↪ Y>;
1998       return capsule<Z>();
1999     }
2000     else { // default
2001       return capsule<void>();
2002     }
2003   }
2004 }
2005
2006 namespace cphmpl {
2007
2008   // Type construction: cphmpl::make_signed<T> → typename
2009
2010   // Effect: Return the signed integer type corresponding to type T,
2011   // with the same const/volatile qualifiers. This will work for a
2012   // class type provided that it knows its signed companion specified
2013   // by the type member T::companion.
2014
2015   template<typename T>
2016   using make_signed = typename decltype(functions::make_signed<T>())::type;
2017 }
2018
```

```
2019 // is_power_of_two
2020
2021 namespace cphmpl::functions {
2022
2023   template<auto n>
2024   requires (n ≥ 0)
2025   constexpr bool is_power_of_two() noexcept {
2026     if constexpr (n == 0) {
2027       return false;
2028     }
2029     constexpr auto test = n bitand (n - 1);
2030     if constexpr (test == 0) { // only one 1–bit?
2031       return true;
2032     }
2033     return false;
2034   }
2035 }
2036
2037 namespace cphmpl {
2038
2039   // Numeric predicate: is_power_of_two<integer> → bool
2040
2041   // Effect: Return true if the integer is a positive power of two
2042
2043   template<auto n>
2044   constexpr bool is_power_of_two = functions::is_power_of_two<n>();
2045 }
2046
2047 // lg
2048
2049 namespace cphmpl::functions {
2050
2051   template<typename T>
2052   constexpr std::size_t default_width() noexcept {
2053     constexpr std::size_t bits_per_byte = CHAR_BIT;
2054     constexpr std::size_t width = bits_per_byte * sizeof(T);
2055     return width;
2056   }
2057
2058   template<typename L, unsigned long int b, unsigned long int i = 0>
2059   requires cphmpl::is_typelist<L>
2060   constexpr std::size_t internal_first_wide_enough() {
2061     if constexpr (i == L::length) {
2062       return L::length;
2063     }
2064     else if constexpr (default_width<typename L:: template type<i>>() ≥ b) {
2065       return i;
2066     }
2067     else {
2068       return internal_first_wide_enough<L, b, i + 1>();
2069     }
2070   }
```

```
2071
2072    template<typename integer>
2073    constexpr auto lg(integer x) {
2074      using natural = cphmpl::make_unsigned<integer>;
2075      natural i = (x < 0) ? natural(- x) : natural(x);
2076      natural result = 0;
2077      while (i > 1) {
2078        i = i / 2;
2079        ++result;
2080      }
2081      return result;
2082    }
2083  }
2084
2085  namespace cphmpl::traits {
2086
2087    using uints = cphmpl::typelist<bool, unsigned char, unsigned short int, unsigned int,
       ↪ unsigned long int, unsigned long long int>;
2088
2089    template<auto x>
2090    class lg {
2091    private:
2092
2093      static constexpr std::size_t result = cphmpl::functions::lg(x);
2094      static constexpr std::size_t rep = cphmpl::functions::lg(result) + 1;
2095
2096    public:
2097
2098      using type =
       ↪ traits::uints::type<functions::internal_first_wide_enough<traits::uints,
       ↪ rep>()>;
2099      static constexpr type value = result;
2100    };
2101
2102    // Observe: The return type is the smallest unsigned built-in
2103    // integer type that can store the result.
2104  }
2105
2106  namespace cphmpl {
2107
2108    // Numeric function: lg<integer> → unsigned integer
2109
2110    // Effect: Return the whole-number binary logarithm of the absolute
2111    // value |x| of an integer x, i.e. ⌊log2 |x|⌋;
2112    // return 0, if x is zero.
2113
2114    template<auto x>
2115    constexpr std::size_t lg = cphmpl::functions::lg(x);
2116  }
2117
2118  // width
2119
```

```
2120 namespace cphmpl::traits {
2121
2122   template<typename T>
2123   class has_static_variable_width { // compiler wants SFINAE
2124   public:
2125     static constexpr bool value = false;
2126   };
2127
2128   template<typename T>
2129   requires requires {T::width;}
2130   class has_static_variable_width<T> {
2131   public:
2132     static constexpr bool value = true;
2133   };
2134 }
2135
2136 namespace cphmpl::functions {
2137
2138   template<typename T, std::size_t i = 0, std::size_t product = 1>
2139   requires std::is_array_v<T>
2140   constexpr std::size_t total_size() {
2141     if constexpr (i == std::rank_v<T>) {
2142       return product;
2143     }
2144     else {
2145       constexpr std::size_t up_to_i = product * std::extent_v<T, i>;
2146       return total_size<T, i + 1, up_to_i>();
2147     }
2148   }
2149
2150   template<typename T>
2151   constexpr auto width() noexcept {
2152     if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
2153       return width<std::remove_cv_t<T>>();
2154     }
2155     else if constexpr (std::is_reference_v<T>) {
2156       constexpr std::size_t result = CHAR_BIT * sizeof(T);
2157       return result;
2158     }
2159     else if constexpr (std::is_pointer_v<T>) {
2160       constexpr std::size_t result = CHAR_BIT * sizeof(T);
2161       return result;
2162     }
2163     else if constexpr (std::is_arithmetic_v<T>) {
2164       constexpr std::size_t sign = std::numeric_limits<T>::is_signed;
2165       constexpr std::size_t digits = std::numeric_limits<T>::digits;
2166       constexpr std::size_t result = sign + digits;
2167       return result;
2168     }
2169     else if constexpr (cphmpl::is_gnu_extension<T>) {
2170       constexpr std::size_t result = CHAR_BIT * sizeof(T);
2171       return result;
```

```
2172      }
2173      else if constexpr (traits::has_static_variable_width<T>::value) {
2174        return T::width;
2175      }
2176      else if constexpr (is_std_bitset<T>::value) {
2177        constexpr std::size_t result = is_std_bitset<T>::width;
2178        return result;
2179      }
2180      else if constexpr (is_std_array<T>::value) {
2181        using V = typename is_std_array<T>::value_type;
2182        constexpr std::size_t result = is_std_array<T>::size * width<V>();
2183        return result;
2184      }
2185      else if constexpr (std::is_array_v<T>) {
2186        using V = std::remove_all_extents_t<T>;
2187        constexpr std::size_t result = total_size<T>() * width<V>();
2188        return result;
2189      }
2190      else { // default
2191        return indeterminate;
2192      }
2193    }
2194  }
2195
2196  namespace cphmpl::traits {
2197
2198    template<typename T>
2199    class width {
2200    private:
2201
2202      static constexpr std::size_t result = cphmpl::functions::width<T>();
2203      static constexpr std::size_t rep = cphmpl::functions::lg(result) + 1;
2204
2205    public:
2206
2207      using type =
          ↪ traits::uints::type<functions::internal_first_wide_enough<traits::uints,
          ↪ rep>()>;
2208      static constexpr type value = result;
2209    };
2210
2211    // Observe: The return type is the smallest unsigned built-in
2212    // integer type that can store the result.
2213  }
2214
2215  namespace cphmpl {
2216
2217    // Type property: cphmpl::width<T> → unsigned integer
2218
2219    // Effect: Return the number of bits in the value representation of
2220    // type T; this number is inderminate if T is unbounded. The object
2221    // representation can be larger. For example, there can be some
```

```
2222    // padding bits due to alignment. This will work for a class type
2223    // provided that it has the static variable T::width which records
2224    // the bit width.
2225
2226    template<typename T>
2227    constexpr std::size_t width = cphmpl::functions::width<T>();
2228  }
2229
2230  // difference
2231
2232  namespace cphmpl::traits {
2233
2234    template<typename T>
2235    class is_object_pointer {
2236    public:
2237      static constexpr bool value = false;
2238    };
2239
2240    template<typename T>
2241    requires std::is_object_v<T>
2242    class is_object_pointer<T*> {
2243    public:
2244      static constexpr bool value = true;
2245    };
2246
2247    template<typename T>
2248    class has_type_member_difference_type { // compiler wants SFINAE
2249    public:
2250      static constexpr bool value = false;
2251    };
2252
2253    template<typename T>
2254    requires requires {typename T::difference_type;}
2255    class has_type_member_difference_type<T> {
2256    public:
2257      static constexpr bool value = true;
2258    };
2259  }
2260
2261  namespace cphmpl::functions {
2262
2263    template<typename T>
2264    constexpr auto difference() noexcept {
2265      if constexpr (std::is_const_v<T> or std::is_volatile_v<T>) {
2266        return difference<std::remove_cv_t<T>>();
2267      }
2268      else if constexpr (std::is_reference_v<T>) {
2269        return capsule<std::ptrdiff_t>();
2270      }
2271      else if constexpr (traits::is_object_pointer<T>::value) {
2272        return capsule<std::ptrdiff_t>();
2273      }
```

```
2274      else if constexpr (cphmpl::is_iterator<T>) {
2275        using R = typename std::iterator_traits<T>::difference_type;
2276        return capsule<R>();
2277      }
2278      else if constexpr (traits::has_type_member_difference_type<T>::value) {
2279        using R = typename T::difference_type;
2280        return capsule<R>();
2281      }
2282      else if constexpr (requires(T const& a, T const& b)
          ↪ {std::is_integral_v<decltype((a - b))>;}) {
2283        using D = decltype(std::declval<T>() - std::declval<T>());
2284        constexpr std::size_t w = cphmpl::width<D>;
2285        using R = cphstl::ℤ<w + 2>;
2286        return capsule<R>();
2287      }
2288      else { // default
2289        using R = cphstl::integer<unsigned long long int>;
2290        return capsule<R>();
2291      }
2292    }
2293  }
2294
2295  namespace cphmpl {
2296
2297    // Type association: cphmpl::difference<T> → typename
2298
2299    // Effect: Provide an alias for the difference type to be used for
2300    // storing the distance between two values specified by type T. For
2301    // a class type the result is T::difference_type if this type member
2302    // exists.
2303
2304    template<typename T>
2305    using difference = typename decltype(functions::difference<T>())::type;
2306  }
2307
2308  // first_wide_enough
2309
2310  namespace cphmpl::functions {
2311
2312    template<typename L, std::size_t bit_width>
2313    requires cphmpl::is_typelist<L>
2314    constexpr auto first_wide_enough() {
2315      if constexpr (cphmpl::size<L> == 0) {
2316        return capsule<void>();
2317      }
2318      else {
2319        using head = typename L::front;
2320        using tail = typename L::pop_front;
2321        constexpr std::size_t present = cphmpl::width<head>;
2322        if constexpr (present ≥ bit_width) {
2323          return capsule<head>();
2324        }
```

```
2325        else {
2326          return first_wide_enough<tail, bit_width>();
2327        }
2328      }
2329    }
2330  }
2331
2332  namespace cphmpl {
2333
2334    // Type selection: cphmpl::first_wide_enough<typelist, integer> → typename
2335
2336    // Effect: Return the first type in the given list of types whose
2337    // width is larger than or equal to the given integer.
2338
2339    template<typename L, std::size_t width>
2340    requires cphmpl::is_typelist<L>
2341    using first_wide_enough = typename decltype(functions::first_wide_enough<L,
       ↪ width>())::type;
2342  }
2343
2344  // twice_wider
2345
2346  namespace cphmpl::traits {
2347
2348    template<typename T>
2349    requires cphmpl::is_unsigned<T>
2350    class twice_wider {
2351    public:
2352
2353      using type = std::array<T, 2>;
2354    };
2355
2356    template<unsigned long int b>
2357    class twice_wider<cphstl::ℕ<b>> {
2358    public:
2359
2360      using type = cphstl::ℕ<2 * b>;
2361    };
2362
2363    template<>
2364    class twice_wider<bool> {
2365    private:
2366
2367    public:
2368
2369      using type = cphstl::ℕ<2>;
2370    };
2371
2372    template<>
2373    class twice_wider<unsigned char> {
2374    private:
2375
```

```
2376      using size_type = unsigned long int;
2377      using U = unsigned char;
2378      static constexpr size_type w = cphmpl::width<U>;
2379
2380    public:
2381
2382      using type =
          ↪ traits::uints::type<functions::internal_first_wide_enough<traits::uints, 2 *
          ↪ w>()>;
2383    };
2384
2385    template<>
2386    class twice_wider<unsigned short int> {
2387    private:
2388
2389      using size_type = unsigned long int;
2390      using U = unsigned short int;
2391      static constexpr size_type w = cphmpl::width<U>;
2392
2393    public:
2394
2395      using type =
          ↪ traits::uints::type<functions::internal_first_wide_enough<traits::uints, 2 *
          ↪ w>()>;
2396    };
2397
2398    template<>
2399    class twice_wider<unsigned int> {
2400    private:
2401
2402      using size_type = unsigned long int;
2403      using U = unsigned int;
2404      static constexpr size_type w = cphmpl::width<U>;
2405
2406    public:
2407
2408      using type =
          ↪ traits::uints::type<functions::internal_first_wide_enough<traits::uints, 2 *
          ↪ w>()>;
2409    };
2410  }
2411
2412  namespace cphmpl {
2413
2414    // Type construction: cphmpl::twice_wider<T> → typename
2415
2416    // Effect: Provide an alias for the type that is twice wider than
2417    // the bounded integer type T.
2418
2419    template<typename T>
2420    using twice_wider = typename traits::twice_wider<T>::type;
2421  }
```

```
2422
2423 // min
2424
2425 namespace cphmpl::traits {
2426
2427   template<typename T>
2428   class has_static_member_function_min {
2429   public:
2430     static constexpr bool value = false;
2431   };
2432
2433   template<typename T>
2434   requires requires {T::min();}
2435   class has_static_member_function_min<T> {
2436   public:
2437     static constexpr bool value = true;
2438   };
2439 }
2440
2441 namespace cphmpl::functions {
2442
2443   template<typename T>
2444   static constexpr T lowest() noexcept {
2445     if constexpr (std::is_arithmetic_v<T>) {
2446       return std::numeric_limits<T>::min();
2447     }
2448     else if constexpr (cphmpl::is_gnu_extension<T>) {
2449       if constexpr (cphmpl::is_unsigned<T>) {
2450         return T();
2451       }
2452       else {
2453         constexpr std::size_t w = cphmpl::width<T>;
2454         using U = cphmpl::make_unsigned<T>;
2455         constexpr U min = U(1) << (w - 1);
2456         return T(min);
2457       }
2458     }
2459     else if constexpr (traits::has_static_member_function_min<T>::value) {
2460       return T::min();
2461     }
2462   }
2463 }
2464
2465 namespace cphmpl {
2466
2467   // Value generation: cphmpl::min<T> → T
2468
2469   // Effect: Return the minimum finite value representable by type T.
2470   // This will work for a class type provided that it has the static
2471   // member function T::min() which returns the desired value.
2472
2473   template<typename T>
```

```
2474    requires cphmpl::is_bounded<T> or (not cphmpl::is_bounded<T> and
          ↪ cphmpl::is_unsigned<T>)
2475    static constexpr T min = functions::lowest<T>();
2476  }
2477
2478  // max
2479
2480  namespace cphmpl::traits {
2481
2482    template<typename T>
2483    class has_static_member_function_max {
2484    public:
2485      static constexpr bool value = false;
2486    };
2487
2488    template<typename T>
2489    requires requires {T::max();}
2490    class has_static_member_function_max<T> {
2491    public:
2492      static constexpr bool value = true;
2493    };
2494  }
2495
2496  namespace cphmpl::functions {
2497
2498    template<typename T>
2499    static constexpr T highest() noexcept {
2500      if constexpr (std::is_arithmetic_v<T>) {
2501        return std::numeric_limits<T>::max();
2502      }
2503      else if constexpr (cphmpl::is_gnu_extension<T>) {
2504        if constexpr (cphmpl::is_unsigned<T>) {
2505          return compl T();
2506        }
2507        else {
2508          constexpr std::size_t w = cphmpl::width<T>;
2509          using U = cphmpl::make_unsigned<T>;
2510          constexpr U min = U(1) << (w - 1);
2511          constexpr T max = compl min;
2512          return max;
2513        }
2514      }
2515      else if constexpr (traits::has_static_member_function_max<T>::value) {
2516        return T::max();
2517      }
2518    }
2519  }
2520
2521  namespace cphmpl {
2522
2523    // Value generation: cphmpl::max<T> → T
2524
```

```
2525    // Effect: Return the maximum finite value representable by type T.
2526    // This will work for a class type provided that it has the static
2527    // member function T::max() which returns the desired value.
2528
2529    template<typename T>
2530    requires cphmpl::is_bounded<T>
2531    static constexpr T max = functions::highest<T>();
2532  }
2533
2534  // is_safely_convertible
2535
2536  namespace cphmpl::functions {
2537
2538    template<typename S, typename T>
2539    requires cphmpl::is_integer<S> and cphmpl::is_integer<T>
2540    constexpr bool is_safely_convertible() noexcept {
2541      if constexpr (std::is_same_v<S, T>) {
2542        return true;
2543      }
2544      else if constexpr (not cphmpl::is_bounded<T>) {
2545        return true;
2546      }
2547      else if constexpr (not cphmpl::is_bounded<S> and cphmpl::is_bounded<T>) {
2548        return false;
2549      }
2550      else if constexpr (cphmpl::is_bounded<S> and cphmpl::is_bounded<T>) {
2551        if constexpr (cphmpl::is_unsigned<S> and cphmpl::is_signed<T>) {
2552          return cphmpl::width<S> + 1 ≤ cphmpl::width<T>;
2553        }
2554        else if constexpr (cphmpl::is_signed<S> and cphmpl::is_unsigned<T>) {
2555          return false;
2556        }
2557        else {
2558          return (cphmpl::width<S> ≤ cphmpl::width<T>);
2559        }
2560      }
2561    }
2562  }
2563
2564  namespace cphmpl {
2565
2566    // Type relationship: cphmpl::is_safely_convertible<S, T> → bool
2567
2568    // Effect: Return true if type S is safety convertible to type T.
2569
2570    template<typename S, typename T>
2571    concept bool is_safely_convertible = functions::is_safely_convertible<S, T>();
2572  }
2573
2574  #endif
```

cphmpl/concepts.h++

```
/*
  Author: Jyrki Katajainen © 2019

  The type predicates imitate the concepts defined in the draft of
  the C++ standard [October 2019]. The names just have the prefix
  "is_" or "has_". Also, the suffixes "_as", "_for", "_from", "_t",
  "_to", "_v", and "_with" have been removed from the names to make
  them shorter.

  Example:
  - type function std::common_reference_t in <type_traits_latest>
  - concept std::common_reference_with in <concepts>
  - type predicate cpp20::has_common_reference in "cphmpl/concepts.h++"
*/

#ifndef __CPP_CONCEPTS__
#define __CPP_CONCEPTS__

#include <functional> // std::invoke
#include <iterator> // std::iterator_traits std::begin std::end
#include <type_traits> // std::is_same_v ...
#include "type_traits_latest" // std::common_reference_t
#include <utility> // std::pair std::swap std::declval

namespace cpp17 { ///

  template<typename T, typename U>
  concept is_same =
    std::is_same_v<T, U> and
    std::is_same_v<U, T>;

  template<typename F, typename T>
  concept bool is_convertible = /* implicitly */
    std::is_convertible_v<typename std::add_rvalue_reference_t<F>, T>;

  template<typename T>
  concept bool is_dereferenceable =
    requires(T& p) {
      { *p } -> auto&&;
    };

  template<typename T>
  concept bool is_equality_comparable =
    requires(T a, T const b) {
      requires is_convertible<decltype((a == a)), bool>;
      requires is_convertible<decltype((a == b)), bool>;
      requires is_convertible<decltype((b == a)), bool>;
      requires is_convertible<decltype((b == b)), bool>;
    };
```

```
50
51  template<typename T>
52  concept bool supports_inequality =
53    requires(T a, T const b) {
54      requires is_convertible<decltype((a != a)), bool>;
55      requires is_convertible<decltype((a != b)), bool>;
56      requires is_convertible<decltype((b != a)), bool>;
57      requires is_convertible<decltype((b != b)), bool>;
58    };
59
60  template<typename T>
61  concept bool is_less_than_comparable =
62    requires(T a, T const b) {
63      requires is_convertible<decltype((a < a)), bool>;
64      requires is_convertible<decltype((a < b)), bool>;
65      requires is_convertible<decltype((b < a)), bool>;
66      requires is_convertible<decltype((b < b)), bool>;
67    };
68
69  template<typename Derived, typename Base>
70  concept bool is_derived =
71    std::is_base_of_v<Base, Derived> and
72    std::is_convertible_v<const volatile Derived*, const volatile Base*>;
73
74  template<typename T>
75  concept bool is_default_constructible =
76    std::is_default_constructible_v<T>;
77
78  template<typename T>
79  concept bool is_move_constructible =
80    std::is_move_constructible_v<T>;
81
82  template<typename T>
83  concept bool is_copy_constructible =
84    std::is_copy_constructible_v<T>;
85
86  template<typename T, typename U>
87  concept bool is_assignable =
88    std::is_assignable_v<T, U>;
89
90  template<typename T>
91  concept bool is_move_assignable =
92    std::is_move_assignable_v<T>;
93
94  template<typename T>
95  concept bool is_copy_assignable =
96    std::is_copy_assignable_v<T>;
97
98  template<typename T>
99  concept bool is_destructible =
100   std::is_nothrow_destructible_v<T>;
101
```

```
102  template<typename T>
103  concept bool is_swappable =
104    std::is_swappable_v<T>;
105
106  template<typename I>
107  using iter_difference = typename std::iterator_traits<I>::difference_type;
108
109  template<typename I>
110  concept bool is_iterator =
111    is_copy_constructible<I> and
112    is_copy_assignable<I> and
113    is_destructible<I> and
114    is_dereferenceable<I> and
115    is_swappable<I> and
116    (std::is_signed_v<iter_difference<I>> or
117     std::is_void_v<iter_difference<I>>) and
118    requires(I i) {
119      requires is_same<decltype((++i)), I&>;
120      i++;
121    };
122
123  template<typename I>
124  requires is_iterator<I>
125  using iter_value = typename std::iterator_traits<I>::value_type;
126
127  template<typename T>
128  requires is_dereferenceable<T>
129  using iter_reference = decltype(*std::declval<T&>());
130
131  template<typename T>
132  requires is_dereferenceable<T>
133  using iter_rvalue_reference = decltype(std::move(*std::declval<T&>()));
134
135  template<typename I>
136  requires is_iterator<I>
137  using iter_category = typename std::iterator_traits<I>::iterator_category;
138
139  template<typename I>
140  concept bool is_input_iterator =
141    is_iterator<I> and
142    is_derived<iter_category<I>, std::input_iterator_tag> and
143    is_equality_comparable<I> and
144    supports_inequality<I> and
145    requires(I i) {
146      typename iter_value<I>;
147      requires is_convertible<decltype((*i)), iter_value<I>>;
148      requires is_convertible<decltype((*i++)), iter_value<I>>;
149    };
150
151  template<typename I>
152  concept bool is_output_iterator =
153    is_iterator<I> and
```

```
154     (is_derived<iter_category<I>, std::forward_iterator_tag>
155      or is_same<iter_category<I>, std::output_iterator_tag>) and
156     requires(I i, iter_value<I> o) {
157       *i = o;
158       requires is_same<decltype((++i)), I&>;
159       requires is_convertible<decltype((i++)), I const&>;
160       *i++ = o;
161     };
162
163   template<typename I>
164   concept bool is_const_iterator =
165     is_iterator<I> and
166     std::is_const_v<std::remove_reference_t<iter_reference<I>>>;
167
168   template<typename I>
169   concept bool is_forward_iterator =
170     is_input_iterator<I> and
171     is_derived<iter_category<I>, std::forward_iterator_tag> and
172     is_default_constructible<I> and
173     ((is_const_iterator<I> and is_same<std::add_const_t<iter_value<I>>&,
      ↪ iter_reference<I>>) or (not is_const_iterator<I> and
      ↪ is_same<iter_value<I>&, iter_reference<I>>)) and
174     requires(I i) {
175       requires is_convertible<decltype((++i)), I const&>;
176       requires is_same<decltype((*i++)), iter_reference<I>>;
177     };
178
179   template<typename I>
180   concept bool is_bidirectional_iterator =
181     is_forward_iterator<I> and
182     is_derived<iter_category<I>, std::bidirectional_iterator_tag> and
183     requires(I i) {
184       requires is_same<decltype((--i)), I&>;
185       requires is_convertible<decltype((i--)), I const&>;
186       requires is_same<decltype((*i--)), iter_reference<I>>;
187     };
188
189   template<typename I>
190   concept bool is_random_access_iterator =
191     is_bidirectional_iterator<I> and
192     is_convertible<iter_category<I>, std::random_access_iterator_tag> and
193     requires(I i, I j, iter_difference<I> d) {
194       requires is_same<decltype((i += d)), I&>;
195       requires is_same<decltype((i -= d)), I&>;
196       requires is_same<decltype((i + d)), I>;
197       requires is_same<decltype((d + i)), I>;
198       requires is_same<decltype((i - d)), I>;
199       requires is_same<decltype((i - j)), iter_difference<I>>;
200       requires is_convertible<decltype((i[d])), iter_reference<I>>;
201       requires is_convertible<decltype((i < j)), bool>;
202       requires is_convertible<decltype((i > j)), bool>;
203       requires is_convertible<decltype((i ≤ j)), bool>;
```

```
204       requires is_convertible<decltype((i ≥ j)), bool>;
205     };
206
207     template<typename F, typename A, typename R = void>
208     concept bool is_unary_function =
209       requires(F function, A argument) {
210         requires is_same<decltype((function(argument))), R>;
211     };
212
213     template<typename F, typename A>
214     concept bool is_unary_predicate =
215       requires(F function, A argument) {
216         requires is_convertible<decltype((function(argument))), bool>;
217     };
218
219     template<typename F, typename A, typename B, typename R = void>
220     concept bool is_binary_function =
221       requires(F function, A a, B b) {
222         requires is_same<decltype((function(a, b))), R>;
223       };
224
225     template<typename F, typename A, typename B>
226     concept bool is_binary_predicate =
227       requires(F function, A a, B b) {
228         requires is_convertible<decltype((function(a, b))), bool>;
229       };
230   }
231
232   namespace cpp20 { ///
233
234     template<typename T>
235     concept bool integral = std::is_integral_v<T>;
236
237     template<typename T, typename U>
238     concept bool is_same =
239       std::is_same_v<T, U> and
240       std::is_same_v<U, T>;
241
242     template<typename F, typename T>
243     concept bool is_convertible =
244       std::is_convertible_v<F, T> and
245       requires(F (&from)()) {
246         static_cast<T>(from());
247       };
248
249     template<typename Derived, typename Base>
250     concept bool is_derived =
251       std::is_base_of_v<Base, Derived> and
252       std::is_convertible_v<const volatile Derived*, const volatile Base*>;
253
254     template<typename T>
255     concept bool is_dereferenceable =
```

```
256     requires(T& p) {
257       { *p } -> auto&&;
258     };
259
260   template<typename I>
261   requires is_dereferenceable<I>
262   using dereference = decltype(*std::declval<I&>());
263
264   template<typename T, typename U>
265   concept bool has_common_reference =
266     is_same<std::common_reference_t<T, U>, std::common_reference_t<U, T>> and
267     is_convertible<T, std::common_reference_t<T, U>> and
268     is_convertible<U, std::common_reference_t<T, U>>;
269
270   template<typename L, typename R>
271   concept bool is_assignable =
272     std::is_lvalue_reference_v<L> and
273     has_common_reference<std::remove_reference_t<L> const&,
274                          std::remove_reference_t<R> const&> and
275     requires(L lhs, R&& rhs) {
276       requires is_same<decltype((lhs = std::forward<R>(rhs))), L>;
277   };
278
279   template<typename T>
280   concept bool is_swappable =
281     requires(T& a, T& b) {
282       std::swap(a, b);
283     };
284
285   template<typename T, typename U>
286   concept bool is_swappable_with =
287     has_common_reference<std::remove_reference_t<T> const&,
288                          std::remove_reference_t<U> const&> and
289     requires(T&& t, U&& u) {
290       std::swap(std::forward<T>(t), std::forward<T>(t));
291       std::swap(std::forward<U>(u), std::forward<U>(u));
292       std::swap(std::forward<T>(t), std::forward<U>(u));
293       std::swap(std::forward<U>(u), std::forward<T>(t));
294   };
295
296   template<typename T>
297   concept bool is_destructible = std::is_nothrow_destructible_v<T>;
298
299   template<typename T, typename... Args>
300   concept bool is_constructible =
301     is_destructible<T> and
302     std::is_constructible_v<T, Args...>;
303
304   template<typename T>
305   concept bool is_default_constructible = is_constructible<T>;
306
307   template<class T>
```

```
308  concept bool is_default_initializable =
309    is_constructible<T> and
310    requires { T{}; } and
311    requires { ::new (static_cast<void*>(nullptr)) T; };
312
313  template<typename T>
314  concept bool is_move_constructible =
315    is_constructible<T, T> and
316    is_convertible<T, T>;
317
318  template<typename T>
319  concept bool is_copy_constructible =
320    is_move_constructible<T> and
321    is_constructible<T, T&> and
322    is_constructible<T, T const&> and
323    is_constructible<T, T const> and
324    is_convertible<T&, T> and
325    is_convertible<T const&, T> and
326    is_convertible<T const, T>;
327
328  template<typename T>
329  concept bool is_movable =
330    std::is_object_v<T> and
331    is_move_constructible<T> and
332    is_assignable<T&, T> and
333    is_swappable<T>;
334
335  template<typename T>
336  concept bool is_copyable =
337    is_copy_constructible<T> and
338    is_movable<T> and
339    is_assignable<T&, const T&>;
340
341  template<typename B>
342  concept bool is_boolean =
343    is_movable<std::remove_cvref_t<B>> and
344    requires(std::remove_reference_t<B> const& b1,
345             std::remove_reference_t<B> const& b2, bool const a) {
346      requires is_convertible<decltype((b1)), bool>;
347      requires is_convertible<decltype((! b1)), bool>;
348      requires is_same<decltype((b1 && b2)), bool>;
349      requires is_same<decltype((b1 && a)), bool>;
350      requires is_same<decltype((a && b2)), bool>;
351      requires is_same<decltype((b1 || b2)), bool>;
352      requires is_same<decltype((b1 || a)), bool>;
353      requires is_same<decltype((a || b2)), bool>;
354      requires is_convertible<decltype((b1 == b2)), bool>;
355      requires is_convertible<decltype((b1 == a)), bool>;
356      requires is_convertible<decltype((a == b2)), bool>;
357      requires is_convertible<decltype((b1 != b2)), bool>;
358      requires is_convertible<decltype((b1 != a)), bool>;
359      requires is_convertible<decltype((a != b2)), bool>;
```

```
360     };
361 }
362
363 namespace cpp20::helper {
364
365   template<typename T, typename U>
366   concept is_equality_comparable =
367     requires(std::remove_reference_t<T> const& t,
368              std::remove_reference_t<U> const& u) {
369       requires is_boolean<decltype((t == u))>;
370       requires is_boolean<decltype((u == t))>;
371       requires is_boolean<decltype((t != u))>;
372       requires is_boolean<decltype((u != t))>;
373     };
374 }
375
376 namespace cpp20 {
377
378   template<typename T>
379   concept bool is_equality_comparable =
380     helper::is_equality_comparable<T, T>;
381
382   template<typename T, typename U>
383   concept bool is_equality_comparable_with =
384     is_equality_comparable<T> and
385     is_equality_comparable<U> and
386     has_common_reference<std::remove_reference_t<T> const&,
387                          std::remove_reference_t<U> const&> and
388     is_equality_comparable<
389     std::common_reference_t<std::remove_reference_t<T> const&,
390                             std::remove_reference_t<U> const&>> and
391     helper::is_equality_comparable<T, U>;
392
393   template<typename T>
394   concept bool is_semiregular =
395     is_copyable<T> and
396     is_default_constructible<T>;
397
398   template<typename T>
399   concept bool is_regular =
400     is_semiregular<T> and
401     is_equality_comparable<T>;
402
403   template<typename T>
404   concept bool is_totally_ordered =
405     is_equality_comparable<T> and
406     requires(std::remove_reference_t<T> const& t,
407              std::remove_reference_t<T> const& u) {
408       requires is_boolean<decltype((t < u))>;
409       requires is_boolean<decltype((t > u))>;
410       requires is_boolean<decltype((t <= u))>;
411       requires is_boolean<decltype((t >= u))>;
```

```
412     };
413
414   template<typename T, typename U>
415   concept bool is_totally_ordered_with =
416     is_totally_ordered<T> and
417     is_totally_ordered<U> and
418     has_common_reference<std::remove_reference_t<T> const&,
419                          std::remove_reference_t<U>const&> and
420     is_totally_ordered<std::common_reference_t<
421                          std::remove_reference_t<T> const&,
422                          std::remove_reference_t<U> const&>> and
423     is_equality_comparable_with<T, U> and
424     requires(std::remove_reference_t<T> const& t,
425              std::remove_reference_t<U> const& u) {
426       requires is_boolean<decltype((t < u))>;
427       requires is_boolean<decltype((t > u))>;
428       requires is_boolean<decltype((t ≤ u))>;
429       requires is_boolean<decltype((t ≥ u))>;
430       requires is_boolean<decltype((u < t))>;
431       requires is_boolean<decltype((u > t))>;
432       requires is_boolean<decltype((u ≤ t))>;
433       requires is_boolean<decltype((u ≥ t))>;
434     };
435
436   template<typename I>
437   concept bool is_input_or_output_iterator =
438     is_default_constructible<I> and
439     is_movable<I> and
440     is_dereferenceable<I> and
441     requires(I i) {
442       typename std::iterator_traits<I>::difference_type;
443       requires std::is_signed_v<typename std::iterator_traits<I>::difference_type>;
444       // §23.3.4.4 is_signed_integer_like
445       requires is_same<decltype((++i)), I&>;
446       i++;
447     };
448
449   template<typename I>
450   concept bool is_iterator =
451     is_input_or_output_iterator<I> or std::is_pointer_v<I>;
452
453   template<typename S, typename I>
454   concept bool is_sentinel =
455     is_copyable<S> and
456     is_default_constructible<S> and
457     is_input_or_output_iterator<I> and
458     is_equality_comparable_with<S, I>;
459
460   template<typename T>
461   requires is_iterator<T>
462   using iter_value = typename std::iterator_traits<T>::value_type;
463
```

```
464  template<typename T>
465  requires is_dereferenceable<T>
466  using iter_reference = decltype(*std::declval<T&>());
467
468  template<typename T>
469  requires is_dereferenceable<T>
470  using iter_rvalue_reference = decltype(std::move(*std::declval<T&>()));
471
472  template<typename T>
473  requires is_iterator<T>
474  using iter_difference = typename std::iterator_traits<T>::difference_type;
475
476  template<typename T>
477  requires is_iterator<T>
478  using iter_category = typename std::iterator_traits<T>::iterator_category;
479
480  template<typename S, typename I>
481  inline constexpr bool disable_sized_sentinel = false;
482
483  template<typename S, typename I>
484  concept bool is_sized_sentinel =
485    is_sentinel<S, I> and
486    not disable_sized_sentinel<std::remove_cv_t<S>, std::remove_cv_t<I>> and
487    requires(I const& i, S const& s) {
488      requires is_same<decltype((s - i)), iter_difference<I>>;
489      requires is_same<decltype((i - s)), iter_difference<I>>;
490    };
491
492  template<typename T>
493  concept bool is_incrementable =
494  requires(T i) {
495    ++i;
496    i++;
497  };
498
499  template<typename T>
500  concept bool is_decrementable =
501  requires(T i) {
502    --i;
503    i--;
504  };
505
506  template<typename I>
507  concept is_readable =
508    requires {
509      typename iter_value<I>;
510      typename iter_reference<I>;
511      typename iter_rvalue_reference<I>;
512    } and
513    has_common_reference<iter_reference<I>&&, iter_value<I>&> and
514    has_common_reference<iter_reference<I>&&, iter_rvalue_reference<I>&&> and
515    has_common_reference<iter_rvalue_reference<I>&&, iter_value<I> const&>;
```

```
516
517   template<typename Out, typename V>
518   concept bool is_writable =
519     requires(Out&& o, V&& t) {
520       *o = std::forward<V>(t);
521       *std::forward<Out>(o) = std::forward<V>(t);
522       const_cast<const iter_reference<Out>&&>(*o) = std::forward<V>(t);
523       const_cast<const iter_reference<Out>&&>(*std::forward<Out>(o)) =
      ↪ std::forward<V>(t);
524     };
525
526   template<typename I>
527   concept bool is_input_iterator =
528     is_input_or_output_iterator<I> and
529     is_readable<I> and
530     requires { typename iter_category<I>; } and
531     is_derived<iter_category<I>, std::input_iterator_tag>;
532
533   template<typename I, typename V>
534   concept bool is_output_iterator =
535     is_input_or_output_iterator<I> and
536     is_writable<I, V> and
537     requires(I i, V&& t) {
538       *i++ = std::forward<V>(t);
539     };
540
541   template<typename I>
542   concept bool is_forward_iterator =
543     is_input_iterator<I> and
544     is_derived<iter_category<I>, std::forward_iterator_tag> and
545     is_incrementable<I> and
546     is_sentinel<I, I>;
547
548   template<typename I>
549   concept bool is_bidirectional_iterator =
550     is_forward_iterator<I> and
551     is_derived<iter_category<I>, std::bidirectional_iterator_tag> and
552     requires(I i) {
553       requires is_same<decltype((--i)), I&>;
554       requires is_same<decltype((i--)), I>;
555     };
556
557   template<typename I>
558   concept bool is_random_access_iterator =
559     is_bidirectional_iterator<I> and
560     is_derived<iter_category<I>, std::random_access_iterator_tag> and
561     is_totally_ordered<I> and
562     is_sized_sentinel<I, I> and
563     requires(I i, I const j, iter_difference<I> const d) {
564       requires is_same<decltype((i += d)), I&>;
565       requires is_same<decltype((j + d)), I>;
566       requires is_same<decltype((d + j)), I>;
```

```
567       requires is_same<decltype((i -= d)), I&>;
568       requires is_same<decltype((j - d)), I>;
569       requires is_same<decltype((j[d])), dereference<I>>;
570     };
571
572   template<typename X>
573   concept bool specifies_range =
574     requires(X&& t) {
575       typename X::value_type;
576       typename X::reference;
577       typename X::const_reference;
578       typename X::iterator;
579       typename X::const_iterator;
580       typename X::difference_type;
581       typename X::size_type;
582       std::begin(t);
583       std::end(t);
584       std::cbegin(t);
585       std::cend(t);
586       std::size(t);
587       std::empty(t);
588     };
589
590   template<typename R>
591   using range_reference = typename std::remove_reference_t<R>::reference;
592
593   template<typename R>
594   using range_category = typename std::iterator_traits<typename
        ↪ std::remove_reference_t<R>::iterator>::iterator_category;
595
596   template<typename R>
597   concept bool is_output_range =
598     specifies_range<std::remove_reference_t<R>> and
599     is_derived<range_category<R>, std::output_iterator_tag>;
600
601   template<typename R>
602   concept bool is_input_range =
603     specifies_range<std::remove_reference_t<R>> and
604     is_derived<range_category<R>, std::input_iterator_tag>;
605
606   template<typename R>
607   concept bool is_forward_range =
608     is_input_range<R> and
609     is_derived<range_category<R>, std::forward_iterator_tag>;
610
611   template<typename R>
612   concept bool is_bidirectional_range =
613     is_forward_range<R> and
614     is_derived<range_category<R>, std::bidirectional_iterator_tag>;
615
616   template<typename R>
617   concept bool is_random_access_range =
```

```
618     is_bidirectional_range<R> and
619     is_derived<range_category<R>, std::random_access_iterator_tag>;
620
621   template<typename In, typename Out>
622   concept bool is_indirectly_movable =
623     is_readable<In> and
624     is_writable<Out, iter_rvalue_reference<In>>;
625
626   template<typename In, typename Out>
627   concept bool is_indirectly_movable_storable =
628     is_indirectly_movable<In, Out> and
629     is_writable<Out, iter_value<In>> and
630     is_movable<iter_value<In>> and
631     is_constructible<iter_value<In>, iter_rvalue_reference<In>> and
632     is_assignable<iter_value<In>&, iter_rvalue_reference<In>>;
633
634   template<typename In, typename Out>
635   concept bool is_indirectly_copyable =
636     is_readable<In> and
637     is_writable<Out, iter_reference<In>>;
638
639   template<typename In, typename Out>
640   concept bool is_indirectly_copyable_storable =
641     is_indirectly_copyable<In, Out> and
642     is_writable<Out, std::add_const_t<iter_value<In>>&> and
643     is_copyable<iter_value<In>> and
644     is_constructible<iter_value<In>, iter_reference<In>> and
645     is_assignable<iter_value<In>&, iter_reference<In>>;
646
647   template<typename I, typename J = I>
648   concept bool is_indirectly_swappable =
649     is_readable<I> and
650     is_readable<J> and
651     requires(I& i, J& j) {
652       std::iter_swap(i, i);
653       std::iter_swap(j, j);
654       std::iter_swap(i, j);
655       std::iter_swap(j, i);
656     };
657
658   template<typename I>
659   concept bool is_permutable =
660     is_forward_iterator<I> and
661     is_indirectly_movable_storable<I, I> and
662     is_indirectly_swappable<I, I>;
663 }
664
665 namespace cpp20 {
666
667   template<typename I, typename J>
668   requires
669     is_forward_iterator<I> and
```

```
670     is_forward_iterator<J> and
671     is_swappable_with<iter_value<I>, iter_value<J>>
672   constexpr void iter_swap(I a, J b) {
673     using std::swap;
674     swap(*a, *b);
675   }
676 }
677
678 namespace cpp20 {
679
680   template<typename F, typename... Args>
681   concept bool is_invocable =
682     requires(F&& f, Args&&... args) {
683       std::invoke(std::forward<F>(f), std::forward<Args>(args)...);
684   };
685
686   template<typename F, typename... Args>
687   concept bool is_predicate =
688     is_invocable<F, Args...> and
689     is_boolean<std::invoke_result_t<F, Args...>>;
690
691   template<typename R, typename T, typename U>
692   concept bool is_relation =
693     is_predicate<R, T, T> and
694     is_predicate<R, U, U> and
695     is_predicate<R, T, U> and
696     is_predicate<R, U, T>;
697
698   template<typename I>
699   requires is_readable<I>
700   using iter_common_reference =
701     std::common_reference_t<iter_reference<I>, iter_value<I>&>;
702
703   template<typename F, typename I>
704   concept bool is_indirectly_unary_invocable =
705     is_readable<I> and
706     is_copy_constructible<F> and
707     is_invocable<F&, iter_value<I>&> and
708     is_invocable<F&, iter_reference<I>> and
709     is_invocable<F&, iter_common_reference<I>> and
710     has_common_reference<std::invoke_result_t<F&, iter_value<I>&>,
711                          std::invoke_result_t<F&, iter_reference<I>>>;
712
713   template<typename F, typename... Is>
714   requires (is_readable<Is> and ...) and is_invocable<F, iter_reference<Is>...>
715   using indirect_result = std::invoke_result_t<F, iter_reference<Is>...>;
716
717   template<typename I, typename Proj>
718   requires is_readable<I> and is_indirectly_unary_invocable<Proj, I>
719   class projected {
720   public:
721
```

```
722     using self = projected<I, Proj>;
723     using value_type = std::remove_cvref_t<indirect_result<Proj&, I>>;
724     using reference = std::add_lvalue_reference_t<value_type>;
725     using difference_type = typename std::iterator_traits<I>::difference_type;
726     using pointer = typename std::iterator_traits<I>::pointer;
727     using iterator_category = typename std::iterator_traits<I>::iterator_category;
728
729     indirect_result<Proj&, I> operator*() const;
730     self& operator++();
731     self operator++(int);
732   };
733
734   template<typename F, typename I, typename J = I>
735   concept bool is_indirect_relation =
736     is_readable<I> and
737     is_readable<J> and
738     is_copy_constructible<F> and
739     is_relation<F&, iter_value<I>&, iter_value<J>&> and
740     is_relation<F&, iter_value<I>&, iter_reference<J>> and
741     is_relation<F&, iter_reference<I>, iter_value<J>&> and
742     is_relation<F&, iter_reference<I>, iter_reference<J>> and
743     is_relation<F&, iter_common_reference<I>, iter_common_reference<J>>;
744 }
745
746 namespace cpp20 {
747
748   template<typename R>
749   requires specifies_range<R>
750   using range_iterator = typename std::remove_reference_t<R>::iterator;
751
752   class dangling {
753   };
754
755   template<typename R>
756   requires specifies_range<R>
757   using safe_iterator_t =
758     typename std::conditional_t<is_forward_iterator<range_iterator<R>>,
759     range_iterator<R>, dangling>;
760
761   class identity {
762   public:
763
764     template<typename T>
765     constexpr T&& operator()(T&& v) const noexcept {
766       return std::forward<T>(v);
767     }
768   };
769
770   class less {
771   public:
772
773     template<typename T, typename U>
```

```
774       requires is_totally_ordered_with<T, U>
775       constexpr bool operator()(T&& t, U &&u) const {
776         return std::forward<T>(t) < std::forward<U>(u);
777       }
778     };
779
780     class greater {
781     public:
782
783       template<typename T, typename U>
784       requires is_totally_ordered_with<T, U>
785       constexpr bool operator()(T&& t, U &&u) const {
786         return std::forward<T>(t) > std::forward<U>(u);
787       }
788     };
789   }
790
791   namespace cpp20 {
792
793     template<typename C, typename I, typename P = identity>
794     concept bool is_indirectly_comparable =
795       is_indirect_relation<C, projected<I, P>>;
796
797     template<typename I, typename C = less, typename P = identity>
798     concept bool is_sortable =
799       is_permutable<I> and
800       is_indirectly_comparable<C, I, P>;
801   }
802
803   #endif
```

cphmpl/example.c++

```
1  #include "cphmpl/lists.h++"
2  #include <cstdlib> // std::size_t std::byte
3  #include <iostream> // std streams
4  #include <string> // std::string
5  #include <utility> // std::pair
6
7  template<std::size_t n>
8  class print {
9  public:
10
11   print() {
12     std::cout << n << "\n";
13   }
14 };
15
16 template<typename T>
17 void ignore_warning(T) {
18 }
19
```

```
20 int main() {
21   using hello = cphmpl::charlist<'h', 'e', 'l', 'l', 'o'>;
22   static_assert(hello::length == 5);
23   print<hello::length> chars;
24   static_assert(hello::value<0> == 'h');
25   static_assert(cphmpl::get_value<hello, 4> == 'o');
26   using ello = hello::pop_front;
27   using fello = ello::push_front<'f'>;
28   using fellow = fello::push_back<'w'>;
29   static_assert(fellow::length == 6);
30
31   using T = cphmpl::typelist<char, unsigned char, signed char>;
32   static_assert(std::is_same_v<T::type<0>, char>);
33   static_assert(std::is_same_v<cphmpl::get_type<T, 2>, signed char>);
34   print<T::length> types;
35   static_assert(not T::is_member<std::byte>);
36
37   using P = std::pair<unsigned int, std::string>;
38   static_assert(std::is_same_v<cphmpl::get_type<P, 0>, unsigned int>);
39
40   using L = cphmpl::intlist<0, 1, 2, 3>;
41   print<L::length> ints;
42
43   ignore_warning(chars);
44   ignore_warning(types);
45   ignore_warning(ints);
46
47   return 0;
48 }
```

cphmpl/test.mk

```
1  TESTFLAGS=-std=c++2a -Wall -Wextra -x c++ -fconcepts
2  IFLAGS = -I..
3  CXX = g++-9
4
5  .PHONY: remove-all-tabs clean
6
7  drivers:= $(wildcard *.c++)
8  bases:= $(basename $(drivers))
9  tests:= $(addsuffix .test, $(bases))
10
11 $(tests): %.test : %.c++
12         $(CXX) $(TESTFLAGS) $(IFLAGS) $*.c++
13         ./a.out
14         @rm -f ./a.out
15
16 remove-all-tabs:
17         sed -i~ -e 's/\t/        /g' *.*++
18
19 clean:
20         @rm -rf *.out temp *~ || true
```

References

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

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

[3] J. Katajainen, Writing C++ metafunctions procedurally, CPH STL report **2020-2**, Department of Computer Science, University of Copenhagen (2020). `http://hjemmesider.diku.dk/~jyrki/Myris/Kat2020bR.html`

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

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