ISO/IEC JTC1 SC22 WG21 N4617
Date: 2016-11-28
ISO/IEC DTS 19568
ISO/IEC JTC1 SC22 WG21
Secretariat: ANSI

Programming Languages — C++ Extensions for Library Fundamentals, Version 2

Langages de programmation — Extensions C++ pour la bibliothèque fondamentaux, version 2

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: Draft Technical Specification
Document stage: (40) Enquiry
Document language: E

Contents

1 General [general]

1.1 Scope [general.scope]

1 This technical specification describes extensions to the C++ Standard Library (1.2). These extensions are classes and functions that are likely to be used widely within a program and/or on the interface boundaries between libraries written by different organizations.

2 This technical specification is non-normative. Some of the library components in this technical specification may be considered for standardization in a future version of C++, but they are not currently part of any C++ standard. Some of the components in this technical specification may never be standardized, and others may be standardized in a substantially changed form.

3 The goal of this technical specification is to build more widespread existing practice for an expanded C++ standard library. It gives advice on extensions to those vendors who wish to provide them.

1.2 Normative references [general.references]

1 The following referenced document is indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

— ISO/IEC 14882:2014, *Programming Languages — C++*

2 ISO/IEC 14882:2014 is herein called the *C++ Standard*. References to clauses within the C++ Standard are written as "C++14 §3.2". The library described in ISO/IEC 14882:2014 clauses 17–30 is herein called the *C++ Standard Library*.

3 Unless otherwise specified, the whole of the C++ Standard's Library introduction (C++14 §17) is included into this Technical Specification by reference.

1.3 Namespaces, headers, and modifications to standard classes [general.namespaces]

1 Since the extensions described in this technical specification are experimental and not part of the C++ standard library, they should not be declared directly within namespace `std`. Unless otherwise specified, all components described in this technical specification either:

— modify an existing interface in the C++ Standard Library in-place,
— are declared in a namespace whose name appends `::experimental::fundamentals_v2` to a namespace defined in the C++ Standard Library, such as `std` or `std::chrono`, or
— are declared in a subnamespace of a namespace described in the previous bullet, whose name is not the same as an existing subnamespace of namespace `std`.

[*Example:* This TS does not define `std::experimental::fundamentals_v2::chrono` because the C++ Standard Library defines `std::chrono`. This TS does not define `std::pmr::experimental::fundamentals_v2` because the C++ Standard Library does not define `std::pmr`. — *end example*]

2 Each header described in this technical specification shall import the contents of `std::experimental::fundamentals_v2` into `std::experimental` as if by

```
namespace std {
  namespace experimental {
    inline namespace fundamentals_v2 {}
```



```
    }
  }
```

3 This technical specification also describes some experimental modifications to existing interfaces in the C++ Standard Library. These modifications are described by quoting the affected parts of the standard and using underlining to represent added text and ~~strike-through~~ to represent deleted text.

4 Unless otherwise specified, references to other entities described in this technical specification are assumed to be qualified with `std::experimental::fundamentals_v2::`, and references to entities described in the standard are assumed to be qualified with `std::`.

5 Extensions that are expected to eventually be added to an existing header `<meow>` are provided inside the `<experimental/meow>` header, which shall include the standard contents of `<meow>` as if by

```
#include <meow>
```

6 New headers are also provided in the `<experimental/>` directory, but without such an `#include`.

Table 1 — C++ library headers

`<experimental/algorithm>`	`<experimental/map>`	`<experimental/string>`
`<experimental/any>`	`<experimental/memory>`	`<experimental/string_view>`
`<experimental/array>`	`<experimental/memory_resource>`	`<experimental/system_error>`
`<experimental/chrono>`	`<experimental/optional>`	`<experimental/tuple>`
`<experimental/deque>`	`<experimental/propagate_const>`	`<experimental/type_traits>`
`<experimental/forward_list>`	`<experimental/random>`	`<experimental/unordered_map>`
`<experimental/functional>`	`<experimental/ratio>`	`<experimental/unordered_set>`
`<experimental/future>`	`<experimental/regex>`	`<experimental/utility>`
`<experimental/iterator>`	`<experimental/set>`	`<experimental/vector>`
`<experimental/list>`	`<experimental/source_location>`	

1.4 Terms and definitions [general.defns]

1 For the purposes of this document, the terms and definitions given in the C++ Standard and the following apply.

1.4.1 [general.defns.direct-non-list-init]
direct-non-list-initialization
A direct-initialization that is not list-initialization.

1.5 Future plans (Informative) [general.plans]

1 This section describes tentative plans for future versions of this technical specification and plans for moving content into future versions of the C++ Standard.

2 The C++ committee intends to release a new version of this technical specification approximately every year, containing the library extensions we hope to add to a near-future version of the C++ Standard. Future versions will define their contents in `std::experimental::fundamentals_v3`, `std::experimental::fundamentals_v4`, etc., with the most recent implemented version inlined into `std::experimental`.

3 When an extension defined in this or a future version of this technical specification represents enough existing practice, it will be moved into the next version of the C++ Standard by removing the `experimental::fundamentals_vN` segment of its namespace and by removing the `experimental/` prefix from its header's path.

1.6 Feature-testing recommendations (Informative) [general.feature.test]

1 For the sake of improved portability between partial implementations of various C++ standards, WG21 (the ISO technical committee for the C++ programming language) recommends that implementers and programmers follow the guidelines in this section concerning feature-test macros. [*Note:* WG21's SD-6 makes similar recommendations for the C++ Standard itself. — *end note*]

2 Implementers who provide a new standard feature should define a macro with the recommended name, in the same circumstances under which the feature is available (for example, taking into account relevant command-line options), to indicate the presence of support for that feature. Implementers should define that macro with the value specified in the most recent version of this technical specification that they have implemented. The recommended macro name is "`__cpp_lib_experimental_`" followed by the string in the "Macro Name Suffix" column.

3 Programmers who wish to determine whether a feature is available in an implementation should base that determination on the presence of the header (determined with `__has_include(<header/name>)`) and the state of the macro with the recommended name. (The absence of a tested feature may result in a program with decreased functionality, or the relevant functionality may be provided in a different way. A program that strictly depends on support for a feature can just try to use the feature unconditionally; presumably, on an implementation lacking necessary support, translation will fail.)

Table 2 — Significant features in this technical specification

Doc. No.	Title	Primary Section	Macro Name Suffix	Value	Header
N3915	apply() call a function with arguments from a tuple	3.2.2	`apply`	201402	`<experimental/tuple>`
N3932	Variable Templates For Type Traits	3.3.1	`type_trait_variable_templates`	201402	`<experimental/type_traits>`
N3866	Invocation type traits	3.3.2	`invocation_type`	201406	`<experimental/type_traits>`
P0013R1	Logical Operator Type Traits	3.3.3	`logical_traits`	201511	`<experimental/type_traits>`
N4502	The C++ Detection Idiom	3.3.4	`detect`	201505	`<experimental/type_traits>`
N4388	A Proposal to Add a Const-Propagating Wrapper to the Standard Library	3.7	`propagate_const`	201505	`<experimental/propagate_const>`
N3916	Type-erased allocator for `std::function`	4.2	`function_erased_allocator`	201406	`<experimental/functional>`
N3905	Extending `std::search` to use Additional Searching Algorithms	4.3	`boyer_moore_searching`	201411	`<experimental/functional>`
N4076	A proposal to add a generalized callable negator	4.4	`not_fn`	201406	`<experimental/functional>`
N3672, N3793	A utility class to represent optional objects	5	`optional`	201411	`<experimental/optional>`
N3804	Any Library Proposal	6	`any`	201411	`<experimental/any>`

Doc. No.	Title	Primary Section	Macro Name Suffix	Value	Header
N3921	`string_view`: a non-owning reference to a string	7	`string_view`	201411	`<experimental/string_view>`
N3920	Extending shared_ptr to Support Arrays	8.2	`shared_ptr_arrays`	201406	`<experimental/memory>`
N3916	Polymorphic Memory Resources	8.4	`memory_resources`	201402	`<experimental/memory_resource>`
N4282	The World's Dumbest Smart Pointer	8.12	`observer_ptr`	201411	`<experimental/memory>`
N4273	Uniform Container Erasure	9.1	`erase_if`	201411	`<experimental/vector>`
N4391	make_array	9.2.2	`make_array`	201505	`<experimental/array>`
N4257	Delimited iterators	10.2	`ostream_joiner`	201411	`<experimental/iterator>`
N3916	Type-erased allocator for `std::promise`	11.2	`promise_erased_allocator`	201406	`<experimental/future>`
N3916	Type-erased allocator for `std::packaged_task`	11.3	`packaged_task_erased_allocator`	201406	`<experimental/future>`
N3925	A `sample` Proposal	12.3	`sample`	201402	`<experimental/algorithm>`
N4061	Greatest Common Divisor and Least Common Multiple	13.1.2, 13.1.3	`gcd_lcm`	201411	`<experimental/numeric>`
N4531	`std::rand` replacement	13.2.2.1	`randint`	201511	`<experimental/random>`
N4519	Source-Code Information Capture	14.1	`source_location`	201505	`<experimental/source_location>`

2 Modifications to the C++ Standard Library [mods]

1 Implementations that conform to this technical specification shall behave as if the modifications contained in this section are made to the C++ Standard.

2.1 Uses-allocator construction [mods.allocator.uses]

1 The following changes to the `uses_allocator` trait and to the description of uses-allocator construction allow a `memory_resource` pointer act as an allocator in many circumstances. [*Note:* Existing programs that use standard allocators would be unaffected by this change. — *end note*]

20.7.7 uses_allocator [allocator.uses]

20.7.7.1 uses_allocator trait [allocator.uses.trait]

```
template <class T, class Alloc> struct uses_allocator;
```

Remarks: Automatically detects whether `T` has a nested `allocator_type` that is convertible from `Alloc`. Meets the BinaryTypeTrait requirements (C++14 §20.10.1). The implementation shall provide a definition that is derived from `true_type` if a type `T::allocator_type` exists and either `is_convertible_v<Alloc, T::allocator_type> != false` or `T::allocator_type` is an alias for `std::experimental::erased_type` (3.1.2), otherwise it shall be derived from `false_type`. A program may specialize this template to derive from `true_type` for a user-defined type `T` that does not have a nested `allocator_type` but nonetheless can be constructed with an allocator where either:

— the first argument of a constructor has type `allocator_arg_t` and the second argument has type `Alloc` or
— the last argument of a constructor has type `Alloc`.

20.7.7.2 uses-allocator construction [allocator.uses.construction]

Uses-allocator construction with allocator `Alloc` refers to the construction of an object `obj` of type `T`, using constructor arguments `v1, v2, ..., vN` of types `V1, V2, ..., VN`, respectively, and an allocator `alloc` of type `Alloc`, where `Alloc` either (1) meets the requirements of an allocator (C++14 §17.6.3.5), or (2) is a pointer type convertible to `std::experimental::pmr::memory_resource*` (8.5), according to the following rules:

3 General utilities library [utilities]

3.1 Utility components [utility]

3.1.1 Header `<experimental/utility>` synopsis [utility.synop]

```
#include <utility>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  // 3.1.2, Class erased_type
  struct erased_type { };

} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

3.1.2 Class `erased_type` [utility.erased.type]

1 `struct erased_type { };`

2 The `erased_type` `struct` is an empty `struct` that serves as a placeholder for a type `T` in situations where the actual type `T` is determined at runtime. For example, the nested type, `allocator_type`, is an alias for `erased_type` in classes that use *type-erased allocators* (see 8.3).

3.2 Tuples [tuple]

3.2.1 Header <experimental/tuple> synopsis [header.tuple.synop]

```
#include <tuple>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  // See C++14 §20.4.2.5, tuple helper classes
  template <class T> constexpr size_t tuple_size_v
    = tuple_size<T>::value;

  // 3.2.2, Calling a function with a tuple of arguments
  template <class F, class Tuple>
  constexpr decltype(auto) apply(F&& f, Tuple&& t);

} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```


3.2.2 Calling a function with a `tuple` of arguments [tuple.apply]

1
```
template <class F, class Tuple>
  constexpr decltype(auto) apply(F&& f, Tuple&& t);
```

2 *Effects:* Given the exposition only function

```
template <class F, class Tuple, size_t... I>
constexpr decltype(auto) apply_impl(  // exposition only
    F&& f, Tuple&& t, index_sequence<I...>) {
  return INVOKE(std::forward<F>(f), std::get<I>(std::forward<Tuple>(t))...);
}
```

Equivalent to

```
return apply_impl(std::forward<F>(f), std::forward<Tuple>(t),
    make_index_sequence<tuple_size_v<decay_t<Tuple>>>{});
```

3.3 Metaprogramming and type traits [meta]

3.3.1 Header <experimental/type_traits> synopsis [meta.type.synop]

```
#include <type_traits>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  // See C++14 §20.10.4.1, primary type categories
  template <class T> constexpr bool is_void_v
    = is_void<T>::value;
  template <class T> constexpr bool is_null_pointer_v
    = is_null_pointer<T>::value;
  template <class T> constexpr bool is_integral_v
    = is_integral<T>::value;
  template <class T> constexpr bool is_floating_point_v
    = is_floating_point<T>::value;
  template <class T> constexpr bool is_array_v
    = is_array<T>::value;
  template <class T> constexpr bool is_pointer_v
    = is_pointer<T>::value;
  template <class T> constexpr bool is_lvalue_reference_v
    = is_lvalue_reference<T>::value;
  template <class T> constexpr bool is_rvalue_reference_v
    = is_rvalue_reference<T>::value;
  template <class T> constexpr bool is_member_object_pointer_v
    = is_member_object_pointer<T>::value;
  template <class T> constexpr bool is_member_function_pointer_v
    = is_member_function_pointer<T>::value;
  template <class T> constexpr bool is_enum_v
    = is_enum<T>::value;
  template <class T> constexpr bool is_union_v
    = is_union<T>::value;
  template <class T> constexpr bool is_class_v
```



```
  = is_class<T>::value;
template <class T> constexpr bool is_function_v
  = is_function<T>::value;

// See C++14 §20.10.4.2, composite type categories
template <class T> constexpr bool is_reference_v
  = is_reference<T>::value;
template <class T> constexpr bool is_arithmetic_v
  = is_arithmetic<T>::value;
template <class T> constexpr bool is_fundamental_v
  = is_fundamental<T>::value;
template <class T> constexpr bool is_object_v
  = is_object<T>::value;
template <class T> constexpr bool is_scalar_v
  = is_scalar<T>::value;
template <class T> constexpr bool is_compound_v
  = is_compound<T>::value;
template <class T> constexpr bool is_member_pointer_v
  = is_member_pointer<T>::value;

// See C++14 §20.10.4.3, type properties
template <class T> constexpr bool is_const_v
  = is_const<T>::value;
template <class T> constexpr bool is_volatile_v
  = is_volatile<T>::value;
template <class T> constexpr bool is_trivial_v
  = is_trivial<T>::value;
template <class T> constexpr bool is_trivially_copyable_v
  = is_trivially_copyable<T>::value;
template <class T> constexpr bool is_standard_layout_v
  = is_standard_layout<T>::value;
template <class T> constexpr bool is_pod_v
  = is_pod<T>::value;
template <class T> constexpr bool is_literal_type_v
  = is_literal_type<T>::value;
template <class T> constexpr bool is_empty_v
  = is_empty<T>::value;
template <class T> constexpr bool is_polymorphic_v
  = is_polymorphic<T>::value;
template <class T> constexpr bool is_abstract_v
  = is_abstract<T>::value;
template <class T> constexpr bool is_final_v
  = is_final<T>::value;
template <class T> constexpr bool is_signed_v
  = is_signed<T>::value;
template <class T> constexpr bool is_unsigned_v
  = is_unsigned<T>::value;
template <class T, class... Args> constexpr bool is_constructible_v
  = is_constructible<T, Args...>::value;
template <class T> constexpr bool is_default_constructible_v
  = is_default_constructible<T>::value;
template <class T> constexpr bool is_copy_constructible_v
```



```
  = is_copy_constructible<T>::value;
template <class T> constexpr bool is_move_constructible_v
  = is_move_constructible<T>::value;
template <class T, class U> constexpr bool is_assignable_v
  = is_assignable<T, U>::value;
template <class T> constexpr bool is_copy_assignable_v
  = is_copy_assignable<T>::value;
template <class T> constexpr bool is_move_assignable_v
  = is_move_assignable<T>::value;
template <class T> constexpr bool is_destructible_v
  = is_destructible<T>::value;
template <class T, class... Args> constexpr bool is_trivially_constructible_v
  = is_trivially_constructible<T, Args...>::value;
template <class T> constexpr bool is_trivially_default_constructible_v
  = is_trivially_default_constructible<T>::value;
template <class T> constexpr bool is_trivially_copy_constructible_v
  = is_trivially_copy_constructible<T>::value;
template <class T> constexpr bool is_trivially_move_constructible_v
  = is_trivially_move_constructible<T>::value;
template <class T, class U> constexpr bool is_trivially_assignable_v
  = is_trivially_assignable<T, U>::value;
template <class T> constexpr bool is_trivially_copy_assignable_v
  = is_trivially_copy_assignable<T>::value;
template <class T> constexpr bool is_trivially_move_assignable_v
  = is_trivially_move_assignable<T>::value;
template <class T> constexpr bool is_trivially_destructible_v
  = is_trivially_destructible<T>::value;
template <class T, class... Args> constexpr bool is_nothrow_constructible_v
  = is_nothrow_constructible<T, Args...>::value;
template <class T> constexpr bool is_nothrow_default_constructible_v
  = is_nothrow_default_constructible<T>::value;
template <class T> constexpr bool is_nothrow_copy_constructible_v
  = is_nothrow_copy_constructible<T>::value;
template <class T> constexpr bool is_nothrow_move_constructible_v
  = is_nothrow_move_constructible<T>::value;
template <class T, class U> constexpr bool is_nothrow_assignable_v
  = is_nothrow_assignable<T, U>::value;
template <class T> constexpr bool is_nothrow_copy_assignable_v
  = is_nothrow_copy_assignable<T>::value;
template <class T> constexpr bool is_nothrow_move_assignable_v
  = is_nothrow_move_assignable<T>::value;
template <class T> constexpr bool is_nothrow_destructible_v
  = is_nothrow_destructible<T>::value;
template <class T> constexpr bool has_virtual_destructor_v
  = has_virtual_destructor<T>::value;

// See C++14 §20.10.5, type property queries
template <class T> constexpr size_t alignment_of_v
  = alignment_of<T>::value;
template <class T> constexpr size_t rank_v
  = rank<T>::value;
template <class T, unsigned I = 0> constexpr size_t extent_v
```

```
  = extent<T, I>::value;

// See C++14 §20.10.6, type relations
template <class T, class U> constexpr bool is_same_v
  = is_same<T, U>::value;
template <class Base, class Derived> constexpr bool is_base_of_v
  = is_base_of<Base, Derived>::value;
template <class From, class To> constexpr bool is_convertible_v
  = is_convertible<From, To>::value;

// 3.3.2, Other type transformations
template <class> class invocation_type; // not defined
template <class F, class... ArgTypes> class invocation_type<F(ArgTypes...)>;
template <class> class raw_invocation_type; // not defined
template <class F, class... ArgTypes> class raw_invocation_type<F(ArgTypes...)>;

template <class T>
  using invocation_type_t = typename invocation_type<T>::type;
template <class T>
  using raw_invocation_type_t = typename raw_invocation_type<T>::type;

// 3.3.3, Logical operator traits
template<class... B> struct conjunction;
template<class... B> constexpr bool conjunction_v = conjunction<B...>::value;
template<class... B> struct disjunction;
template<class... B> constexpr bool disjunction_v = disjunction<B...>::value;
template<class B> struct negation;
template<class B> constexpr bool negation_v = negation<B>::value;

// 3.3.4, Detection idiom
template <class...> using void_t = void;

struct nonesuch {
  nonesuch() = delete;
  ~nonesuch() = delete;
  nonesuch(nonesuch const&) = delete;

  void operator=(nonesuch const&) = delete;
};

template <template<class...> class Op, class... Args>
  using is_detected = see below;
template <template<class...> class Op, class... Args>
  constexpr bool is_detected_v = is_detected<Op, Args...>::value;
template <template<class...> class Op, class... Args>
  using detected_t = see below;
template <class Default, template<class...> class Op, class... Args>
  using detected_or = see below;
template <class Default, template<class...> class Op, class... Args>
  using detected_or_t = typename detected_or<Default, Op, Args...>::type;
template <class Expected, template<class...> class Op, class... Args>
  using is_detected_exact = is_same<Expected, detected_t<Op, Args...>>;
```



```
template <class Expected, template<class...> class Op, class... Args>
  constexpr bool is_detected_exact_v
    = is_detected_exact<Expected, Op, Args...>::value;
template <class To, template<class...> class Op, class... Args>
  using is_detected_convertible = is_convertible<detected_t<Op, Args...>, To>;
template <class To, template<class...> class Op, class... Args>
  constexpr bool is_detected_convertible_v
    = is_detected_convertible<To, Op, Args...>::value;

} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

3.3.2 Other type transformations [meta.trans.other]

1 This sub-clause contains templates that may be used to transform one type to another following some predefined rule.

2 Each of the templates in this subclause shall be a *TransformationTrait* (C++14 §20.10.1).

3 Within this section, define the *invocation parameters* of `INVOKE(f, t1, t2, ..., tN)` as follows, in which `T1` is the possibly *cv*-qualified type of `t1` and `U1` denotes `T1&` if `t1` is an lvalue or `T1&&` if `t1` is an rvalue:

- When `f` is a pointer to a member function of a class `T` the *invocation parameters* are `U1` followed by the parameters of `f` matched by `t2`, ..., `tN`.
- When `N == 1` and `f` is a pointer to member data of a class `T` the *invocation parameter* is `U1`.
- If `f` is a class object, the *invocation parameters* are the parameters matching `t1`, ..., `tN` of the best viable function (C++14 §13.3.3) for the arguments `t1`, ..., `tN` among the function call operators and surrogate call functions of `f`.
- In all other cases, the *invocation parameters* are the parameters of `f` matching `t1`, ... `tN`.

4 In all of the above cases, if an argument `tI` matches the ellipsis in the function's *parameter-declaration-clause*, the corresponding *invocation parameter* is defined to be the result of applying the default argument promotions (C++14 §5.2.2) to `tI`.

[*Example:* Assume `S` is defined as

```
struct S {
  int f(double const &) const;
  void operator()(int, int);
  void operator()(char const *, int i = 2, int j = 3);
  void operator()(...);
};
```

- The invocation parameters of `INVOKE(&S::f, S(), 3.5)` are `(S &&, double const &)`.
- The invocation parameters of `INVOKE(S(), 1, 2)` are `(int, int)`.
- The invocation parameters of `INVOKE(S(), "abc", 5)` are `(const char *, int)`. The defaulted parameter `j` does not correspond to an argument.
- The invocation parameters of `INVOKE(S(), locale(), 5)` are `(locale, int)`. Arguments corresponding to ellipsis maintain their types.

— *end example*]

Table 3 — Other type transformations

Template	Condition	Comments
template <class Fn, class... ArgTypes> struct raw_invocation_type< Fn(ArgTypes...)>;	Fn and all types in the parameter pack ArgTypes shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.	*see below*
template <class Fn, class... ArgTypes> struct invocation_type< Fn(ArgTypes...)>;	Fn and all types in the parameter pack ArgTypes shall be complete types, (possibly cv-qualified) void, or arrays of unknown bound.	*see below*

5 Access checking is performed as if in a context unrelated to Fn and ArgTypes. Only the validity of the immediate context of the expression is considered. [*Note:* The compilation of the expression can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note*]

6 The member raw_invocation_type<Fn(ArgTypes...)>::type shall be defined as follows. If the expression *INVOKE*(declval<Fn>(), declval<ArgTypes>()...) is ill-formed when treated as an unevaluated operand (C++14 §5), there shall be no member type. Otherwise:

— Let R denote result_of_t<Fn(ArgTypes...)>.
— Let the types Ti be the *invocation parameters* of *INVOKE*(declval<Fn>(), declval<ArgTypes>()...).
— Then the member type shall name the function type R(T1, T2, ...).

7 The member invocation_type<Fn(ArgTypes...)>::type shall be defined as follows. If raw_invocation_type<Fn(ArgTypes...)>::type does not exist, there shall be no member type. Otherwise:

— Let A1, A2, … denote ArgTypes...
— Let R(T1, T2, …) denote raw_invocation_type_t<Fn(ArgTypes...)>
— Then the member type shall name the function type R(U1, U2, …) where Ui is decay_t<Ai> if declval<Ai>() is an rvalue otherwise Ti.

3.3.3 Logical operator traits [meta.logical]

1 This subclause describes type traits for applying logical operators to other type traits.

```
template<class... B> struct conjunction : see below { };
```

2 The class template conjunction forms the logical conjunction of its template type arguments.

3 For a specialization conjunction<B1, …, BN> if there is a template type argument B*i* with B*i*::value == false then instantiating conjunction<B1, …, BN>::value does not require the instantiation of B*j*::value for *j* > *i*. [*Note:* This is analogous to the short-circuiting behavior of &&. — *end note*]

4 Every template type argument for which B*i*::value is instantiated shall be usable as a base class and shall have a static data member value which is convertible to bool, is not hidden, and is unambiguously available in the type.

5 The specialization conjunction<B1, …, BN> has a public and unambiguous base that is either

— the first type B*i* in the list true_type, B1, …, BN for which bool(B*i*::value) is false, or
— if there is no such B*i*, the last type in the list.

6 [*Note:* This means a specialization of conjunction does not necessarily inherit from either true_type or false_type. — *end note*]

7 The member names of the base class, other than conjunction and operator=, shall not be hidden and shall be unambiguously available in conjunction.

```
template<class... B> struct disjunction : see below { };
```


8 The class template `disjunction` forms the logical disjunction of its template type arguments.

9 For a specialization `disjunction<B1, …, BN>` if there is a template type argument `Bi` with `Bi::value != false` then instantiating `disjunction<B1, …, BN>::value` does not require the instantiation of `Bj::value` for $j > i$. [*Note:* This is analogous to the short-circuiting behavior of `||`. — *end note*]

10 Every template type argument for which `Bi::value` is instantiated shall be usable as a base class and shall have a static data member `value` which is convertible to `bool`, is not hidden, and is unambiguously available in the type.

11 The specialization `disjunction<B1, …, BN>` has a public and unambiguous base that is either

- — the first type `Bi` in the list `false_type, B1, …, BN` for which `bool(Bi::value)` is `true`, or,
- — if there is no such `Bi`, the last type in the list.

12 [*Note:* This means a specialization of `disjunction` does not necessarily inherit from either `true_type` or `false_type`. — *end note*]

13 The member names of the base class, other than `disjunction` and `operator=`, shall not be hidden and shall be unambiguously available in `disjunction`.

```
template<class B> struct negation : see below { };
```

14 The class template `negation` forms the logical negation of its template type argument. The type `negation<B>` is a UnaryTypeTrait with a BaseCharacteristic of `integral_constant<bool, !bool(B::value)>`.

3.3.4 Detection idiom [meta.detect]

```
template <class Default, class AlwaysVoid,
          template<class...> class Op, class... Args>
struct DETECTOR { // exposition only
  using value_t = false_type;
  using type = Default;
};

template <class Default, template<class...> class Op, class... Args>
struct DETECTOR<Default, void_t<Op<Args...>>, Op, Args...> { // exposition only
  using value_t = true_type;
  using type = Op<Args...>;
};

template <template<class...> class Op, class... Args>
  using is_detected = typename DETECTOR<nonesuch, void, Op, Args...>::value_t;

template <template<class...> class Op, class... Args>
  using detected_t = typename DETECTOR<nonesuch, void, Op, Args...>::type;

template <class Default, template<class...> class Op, class... Args>
  using detected_or = DETECTOR<Default, void, Op, Args...>;
```

[*Example:*

```
// archetypal helper alias for a copy assignment operation:
template <class T>
  using copy_assign_t = decltype(declval<T&>() = declval<T const &>());

// plausible implementation for the is_assignable type trait:
template <class T>
```



```
  using is_copy_assignable = is_detected<copy_assign_t, T>;

// plausible implementation for an augmented is_assignable type trait
// that also checks the return type:
template <class T>
  using is_canonical_copy_assignable = is_detected_exact<T&, copy_assign_t, T>;
```

— *end example*]
[*Example:*

```
// archetypal helper alias for a particular type member:
template <class T>
  using diff_t = typename T::difference_type;

// alias the type member, if it exists, otherwise alias ptrdiff_t:
template <class Ptr>
  using difference_type = detected_or_t<ptrdiff_t, diff_t, Ptr>;
```

— *end example*]

3.4 Compile-time rational arithmetic [ratio]

3.4.1 Header <experimental/ratio> synopsis [header.ratio.synop]

```
#include <ratio>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  // See C++14 §20.11.5, ratio comparison
  template <class R1, class R2> constexpr bool ratio_equal_v
    = ratio_equal<R1, R2>::value;
  template <class R1, class R2> constexpr bool ratio_not_equal_v
    = ratio_not_equal<R1, R2>::value;
  template <class R1, class R2> constexpr bool ratio_less_v
    = ratio_less<R1, R2>::value;
  template <class R1, class R2> constexpr bool ratio_less_equal_v
    = ratio_less_equal<R1, R2>::value;
  template <class R1, class R2> constexpr bool ratio_greater_v
    = ratio_greater<R1, R2>::value;
  template <class R1, class R2> constexpr bool ratio_greater_equal_v
    = ratio_greater_equal<R1, R2>::value;

} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```


3.5 Time utilities [time]

3.5.1 Header <experimental/chrono> synopsis [header.chrono.synop]

```
#include <chrono>

namespace std {
namespace chrono {
namespace experimental {
inline namespace fundamentals_v2 {

  // See C++14 §20.12.4, customization traits
  template <class Rep> constexpr bool treat_as_floating_point_v
    = treat_as_floating_point<Rep>::value;

} // namespace fundamentals_v2
} // namespace experimental
} // namespace chrono
} // namespace std
```

3.6 System error support [syserror]

3.6.1 Header <experimental/system_error> synopsis [header.system_error.synop]

```
#include <system_error>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  // See C++14 §19.5, System error support
  template <class T> constexpr bool is_error_code_enum_v
    = is_error_code_enum<T>::value;
  template <class T> constexpr bool is_error_condition_enum_v
    = is_error_condition_enum<T>::value;

} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

3.7 Class template `propagate_const` [propagate_const]

3.7.1 Class template `propagate_const general` [propagate_const.general]

1 `propagate_const` is a wrapper around a pointer-like object type `T` which treats the wrapped pointer as a pointer to `const` when the wrapper is accessed through a `const` access path.

3.7.2 Header `<experimental/propagate_const>` synopsis [propagate_const.synopsis]

```
namespace std {
  namespace experimental {
  inline namespace fundamentals_v2 {
    template <class T> class propagate_const {
    public:
      using element_type = remove_reference_t<decltype(*declval<T&>())>;

      // 3.7.4, propagate_const constructors
      constexpr propagate_const() = default;
      propagate_const(const propagate_const& p) = delete;
      constexpr propagate_const(propagate_const&& p) = default;
      template <class U>
        see below constexpr propagate_const(propagate_const<U>&& pu);
      template <class U>
        see below constexpr propagate_const(U&& u);

      // 3.7.5, propagate_const assignment
      propagate_const& operator=(const propagate_const& p) = delete;
      constexpr propagate_const& operator=(propagate_const&& p) = default;
      template <class U>
        constexpr propagate_const& operator=(propagate_const<U>&& pu);
      template <class U>
        constexpr propagate_const& operator=(U&& u);

      // 3.7.6, propagate_const const observers
      explicit constexpr operator bool() const;
      constexpr const element_type* operator->() const;
      constexpr operator const element_type*() const; // Not always defined
      constexpr const element_type& operator*() const;
      constexpr const element_type* get() const;

      // 3.7.7, propagate_const non-const observers
      constexpr element_type* operator->();
      constexpr operator element_type*(); // Not always defined
      constexpr element_type& operator*();
      constexpr element_type* get();

      // 3.7.8, propagate_const modifiers
      constexpr void swap(propagate_const& pt) noexcept(see below);

    private:
      T t_; //exposition only
    };

    // 3.7.9, propagate_const relational operators
    template <class T>
      constexpr bool operator==(const propagate_const<T>& pt, nullptr_t);
    template <class T>
      constexpr bool operator==(nullptr_t, const propagate_const<T>& pu);
```



```
template <class T>
  constexpr bool operator!=(const propagate_const<T>& pt, nullptr_t);
template <class T>
  constexpr bool operator!=(nullptr_t, const propagate_const<T>& pu);

template <class T, class U>
  constexpr bool operator==(const propagate_const<T>& pt, const propagate_const<U>& pu);
template <class T, class U>
  constexpr bool operator!=(const propagate_const<T>& pt, const propagate_const<U>& pu);
template <class T, class U>
  constexpr bool operator<(const propagate_const<T>& pt, const propagate_const<U>& pu);
template <class T, class U>
  constexpr bool operator>(const propagate_const<T>& pt, const propagate_const<U>& pu);
template <class T, class U>
  constexpr bool operator<=(const propagate_const<T>& pt, const propagate_const<U>& pu);
template <class T, class U>
  constexpr bool operator>=(const propagate_const<T>& pt, const propagate_const<U>& pu);

template <class T, class U>
  constexpr bool operator==(const propagate_const<T>& pt, const U& u);
template <class T, class U>
  constexpr bool operator!=(const propagate_const<T>& pt, const U& u);
template <class T, class U>
  constexpr bool operator<(const propagate_const<T>& pt, const U& u);
template <class T, class U>
  constexpr bool operator>(const propagate_const<T>& pt, const U& u);
template <class T, class U>
  constexpr bool operator<=(const propagate_const<T>& pt, const U& u);
template <class T, class U>
  constexpr bool operator>=(const propagate_const<T>& pt, const U& u);

template <class T, class U>
  constexpr bool operator==(const T& t, const propagate_const<U>& pu);
template <class T, class U>
  constexpr bool operator!=(const T& t, const propagate_const<U>& pu);
template <class T, class U>
  constexpr bool operator<(const T& t, const propagate_const<U>& pu);
template <class T, class U>
  constexpr bool operator>(const T& t, const propagate_const<U>& pu);
template <class T, class U>
  constexpr bool operator<=(const T& t, const propagate_const<U>& pu);
template <class T, class U>
  constexpr bool operator>=(const T& t, const propagate_const<U>& pu);

// 3.7.10, propagate_const specialized algorithms
template <class T>
  constexpr void swap(propagate_const<T>& pt, propagate_const<T>& pt2) noexcept(see below);

// 3.7.11, propagate_const underlying pointer access
template <class T>
  constexpr const T& get_underlying(const propagate_const<T>& pt) noexcept;
template <class T>
```



```
    constexpr T& get_underlying(propagate_const<T>& pt) noexcept;
  } // inline namespace fundamentals_v2
  } // namespace experimental

  // 3.7.12, propagate_const hash support
  template <class T> struct hash;
  template <class T>
    struct hash<experimental::fundamentals_v2::propagate_const<T>>;

  // 3.7.13, propagate_const comparison function objects
  template <class T> struct equal_to;
  template <class T>
    struct equal_to<experimental::fundamentals_v2::propagate_const<T>>;
  template <class T> struct not_equal_to;
  template <class T>
    struct not_equal_to<experimental::fundamentals_v2::propagate_const<T>>;
  template <class T> struct less;
  template <class T>
    struct less<experimental::fundamentals_v2::propagate_const<T>>;
  template <class T> struct greater;
  template <class T>
    struct greater<experimental::fundamentals_v2::propagate_const<T>>;
  template <class T> struct less_equal;
  template <class T>
    struct less_equal<experimental::fundamentals_v2::propagate_const<T>>;
  template <class T> struct greater_equal;
  template <class T>
    struct greater_equal<experimental::fundamentals_v2::propagate_const<T>>;
} // namespace std
```

3.7.3 `propagate_const` requirements on `T` [propagate_const.requirements]

1 `T` shall be an object pointer type or a class type for which `decltype(*declval<T&>())` is an lvalue reference; otherwise the program is ill-formed.

2 If `T` is an array type, reference type, pointer to function type or pointer to (possibly cv-qualified) `void`, then the program is ill-formed.

3 [*Note:* `propagate_const<const int*>` is well-formed — *end note*]

3.7.3.1 `propagate_const` requirements on class type `T` [propagate_const.class_type_requirements]

1 If `T` is class type then it shall satisfy the following requirements. In this sub-clause `t` denotes a non-`const` lvalue of type `T`, `ct` is a `const T&` bound to `t`, `element_type` denotes an object type.

2 `T` and `const T` shall be contextually convertible to `bool`.

3 If `T` is implicitly convertible to `element_type*`, `(element_type*)t == t.get()` shall be `true`.

4 If `const T` is implicitly convertible to `const element_type*`, `(const element_type*)ct == ct.get()` shall be `true`.

Table 4 — Requirements on class types `T`

Expression	Return type	Pre-conditions	Operational semantics
`t.get()`	`element_type*`		

ct.get()	const element_type* or element_type*		t.get() == ct.get().
*t	element_type&	t.get() != nullptr	*t refers to the same object as *(t.get())
*ct	const element_type& or element_type&	ct.get() != nullptr	*ct refers to the same object as *(ct.get())
t.operator->()	element_type*	t.get() != nullptr	t.operator->() == t.get()
ct.operator->()	const element_type* or element_type*	ct.get() != nullptr	ct.operator->() == ct.get()
(bool)t	bool		(bool)t is equivalent to t.get() != nullptr
(bool)ct	bool		(bool)ct is equivalent to ct.get() != nullptr

3.7.4 propagate_const constructors [propagate_const.ctor]

1 [*Note:* The following constructors are conditionally specified as explicit. This is typically implemented by declaring two such constructors, of which at most one participates in overload resolution. — *end note*]

2
```
template <class U>
  see below constexpr propagate_const(propagate_const<U>&& pu);
```

3 *Remarks:* This constructor shall not participate in overload resolution unless is_constructible_v<T, U&&>. The constructor is specified as explicit if and only if !is_convertible_v<U&&, T>.

4 *Effects:* Initializes t_ as if direct-non-list-initializing an object of type T with the expression std::move(pu.t_).

5
```
template <class U>
  see below constexpr propagate_const(U&& u);
```

6 *Remarks:* This constructor shall not participate in overload resolution unless is_constructible_v<T, U&&> and decay_t<U> is not a specialization of propagate_const. The constructor is specified as explicit if and only if !is_convertible_v<U&&, T>.

7 *Effects:* Initializes t_ as if direct-non-list-initializing an object of type T with the expression std::forward<U>(u).

3.7.5 propagate_const assignment [propagate_const.assignment]

1
```
template <class U>
  constexpr propagate_const& operator=(propagate_const<U>&& pu);
```

2 *Remarks:* This function shall not participate in overload resolution unless U is implicitly convertible to T.

3 *Effects:* t_ = std::move(pu.t_).

4 *Returns:* *this.

5
```
template <class U>
  constexpr propagate_const& operator=(U&& u);
```

6 *Remarks:* This function shall not participate in overload resolution unless U is implicitly convertible to T and decay_t<U> is not a specialization of propagate_const.

7 *Effects:* t_ = std::forward<U>(u).

8 *Returns:* *this.

3.7.6 `propagate_const` const observers [propagate_const.const_observers]

1 `explicit constexpr operator bool() const;`

2 *Returns:* `(bool)t_`.

3 `constexpr const element_type* operator->() const;`

4 *Requires:* `get() != nullptr`.

5 *Returns:* `get()`.

6 `constexpr operator const element_type*() const;`

7 *Returns:* `get()`.

8 *Remarks:* This function shall not participate in overload resolution unless `T` is an object pointer type or has an implicit conversion to `const element_type*`.

9 `constexpr const element_type& operator*() const;`

10 *Requires:* `get() != nullptr`.

11 *Returns:* `*get()`.

12 `constexpr const element_type* get() const;`

13 *Returns:* `t_` if `T` is an object pointer type, otherwise `t_.get()`.

3.7.7 `propagate_const` non-const observers [propagate_const.non_const_observers]

1 `constexpr element_type* operator->();`

2 *Requires:* `get() != nullptr`.

3 *Returns:* `get()`.

4 `constexpr operator element_type*();`

5 *Returns:* `get()`.

6 *Remarks:* This function shall not participate in overload resolution unless `T` is an object pointer type or has an implicit conversion to `element_type*`.

7 `constexpr element_type& operator*();`

8 *Requires:* `get() != nullptr`.

9 *Returns:* `*get()`.

10 `constexpr element_type* get();`

11 *Returns:* `t_` if `T` is an object pointer type, otherwise `t_.get()`.

3.7.8 `propagate_const` modifiers [propagate_const.modifiers]

1
```
constexpr void swap(propagate_const& pt) noexcept(see below);
```

2 The constant-expression in the exception-specification is `noexcept(swap(t_, pt.t_))`.

3 *Effects:* `swap(t_, pt.t_)`.

3.7.9 `propagate_const` relational operators [propagate_const.relational]

1
```
template <class T>
  constexpr bool operator==(const propagate_const<T>& pt, nullptr_t);
```

2 *Returns:* `pt.t_ == nullptr`.

3
```
template <class T>
  constexpr bool operator==(nullptr_t, const propagate_const<T>& pt);
```

4 *Returns:* `nullptr == pt.t_`.

5
```
template <class T>
  constexpr bool operator!=(const propagate_const<T>& pt, nullptr_t);
```

6 *Returns:* `pt.t_ != nullptr`.

7
```
template <class T>
  constexpr bool operator!=(nullptr_t, const propagate_const<T>& pt);
```

8 *Returns:* `nullptr != pt.t_`.

9
```
template <class T, class U>
  constexpr bool operator==(const propagate_const<T>& pt, const propagate_const<U>& pu);
```

10 *Returns:* `pt.t_ == pu.t_`.

11
```
template <class T, class U>
  constexpr bool operator!=(const propagate_const<T>& pt, const propagate_const<U>& pu);
```

12 *Returns:* `pt.t_ != pu.t_`.

13
```
template <class T, class U>
  constexpr bool operator<(const propagate_const<T>& pt, const propagate_const<U>& pu);
```

14 *Returns:* `pt.t_ < pu.t_`.

15
```
template <class T, class U>
  constexpr bool operator>(const propagate_const<T>& pt, const propagate_const<U>& pu);
```

16 *Returns:* `pt.t_ > pu.t_`.

17
```
template <class T, class U>
  constexpr bool operator<=(const propagate_const<T>& pt, const propagate_const<U>& pu);
```

18 *Returns:* `pt.t_ <= pu.t_`.

19
```
template <class T, class U>
  constexpr bool operator>=(const propagate_const<T>& pt, const propagate_const<U>& pu);
```

20 *Returns:* `pt.t_ >= pu.t_`.

21
```
template <class T, class U>
  constexpr bool operator==(const propagate_const<T>& pt, const U& u);
```

22 *Returns:* `pt.t_ == u`.

23
```
template <class T, class U>
  constexpr bool operator!=(const propagate_const<T>& pt, const U& u);
```

24 *Returns:* `pt.t_ != u`.

25
```
template <class T, class U>
  constexpr bool operator<(const propagate_const<T>& pt, const U& u);
```

26 *Returns:* `pt.t_ < u`.

27
```
template <class T, class U>
  constexpr bool operator>(const propagate_const<T>& pt, const U& u);
```

28 *Returns:* `pt.t_ > u`.

29
```
template <class T, class U>
  constexpr bool operator<=(const propagate_const<T>& pt, const U& u);
```

30 *Returns:* `pt.t_ <= u`.

31
```
template <class T, class U>
  constexpr bool operator>=(const propagate_const<T>& pt, const U& u);
```

32 *Returns:* `pt.t_ >= u`.

33
```
template <class T, class U>
  constexpr bool operator==(const T& t, const propagate_const<U>& pu);
```

34 *Returns:* `t == pu.t_`.

35
```
template <class T, class U>
  constexpr bool operator!=(const T& t, const propagate_const<U>& pu);
```

36 *Returns:* `t != pu.t_`.

37
```
template <class T, class U>
  constexpr bool operator<(const T& t, const propagate_const<U>& pu);
```

38 *Returns:* `t < pu.t_`.

39
```
template <class T, class U>
  constexpr bool operator>(const T& t, const propagate_const<U>& pu);
```

40 *Returns:* `t > pu.t_`.

41
```
template <class T, class U>
  constexpr bool operator<=(const T& t, const propagate_const<U>& pu);
```

42 *Returns:* `t <= pu.t_`.

43
```
template <class T, class U>
  constexpr bool operator>=(const T& t, const propagate_const<U>& pu);
```

44 *Returns:* `t >= pu.t_`.

3.7.10 `propagate_const` specialized algorithms [propagate_const.algorithms]

1
```
template <class T>
  constexpr void swap(propagate_const<T>& pt1, propagate_const<T>& pt2) noexcept(see below);
```

2 The constant-expression in the exception-specification is `noexcept(pt1.swap(pt2))`.

3 *Effects:* `pt1.swap(pt2)`.

3.7.11 `propagate_const` underlying pointer access [propagate_const.underlying]

1 Access to the underlying object pointer type is through free functions rather than member functions. These functions are intended to resemble cast operations to encourage caution when using them.

2
```
template <class T>
  constexpr const T& get_underlying(const propagate_const<T>& pt) noexcept;
```

3 *Returns:* a reference to the underlying object pointer type.

4
```
template <class T>
  constexpr T& get_underlying(propagate_const<T>& pt) noexcept;
```

5 *Returns:* a reference to the underlying object pointer type.

3.7.12 `propagate_const` hash support [propagate_const.hash]

1
```
template <class T>
  struct hash<experimental::fundamentals_v2::propagate_const<T>>;
```

2 For an object `p` of type `propagate_const<T>`, `hash<experimental::fundamentals_v2::propagate_const<T>>()(p)` shall evaluate to the same value as `hash<T>()(p.t_)`.

3 *Requires:* The specialization `hash<T>` shall be well-formed and well-defined, and shall meet the requirements of class template hash.

3.7.13 `propagate_const` comparison function objects [propagate_const.comparison_function_objects]

1
```
template <class T>
  struct equal_to<experimental::fundamentals_v2::propagate_const<T>>;
```

2 For objects `p, q` of type `propagate_const<T>`, `equal_to<experimental::fundamentals_v2::propagate_const<T>>()(p, q)` shall evaluate to the same value as `equal_to<T>()(p.t_, q.t_)`.

3 *Requires:* The specialization `equal_to<T>` shall be well-formed and well-defined.

4
```
template <class T>
  struct not_equal_to<experimental::fundamentals_v2::propagate_const<T>>;
```

5 For objects `p, q` of type `propagate_const<T>`, `not_equal_to<experimental::fundamentals_v2::propagate_const<T>>()(p, q)` shall evaluate to the same value as `not_equal_to<T>()(p.t_, q.t_)`.

6 *Requires:* The specialization `not_equal_to<T>` shall be well-formed and well-defined.

7
```
template <class T>
  struct less<experimental::fundamentals_v2::propagate_const<T>>;
```

8 For objects `p, q` of type `propagate_const<T>`, `less<experimental::fundamentals_v2::propagate_const<T>>()(p, q)` shall evaluate to the same value as `less<T>()(p.t_, q.t_)`.

9 *Requires:* The specialization `less<T>` shall be well-formed and well-defined.

10
```
template <class T>
  struct greater<experimental::fundamentals_v2::propagate_const<T>>;
```

11 For objects `p, q` of type `propagate_const<T>`, `greater<experimental::fundamentals_v2::propagate_const<T>>()(p, q)` shall evaluate to the same value as `greater<T>()(p.t_, q.t_)`.

12 *Requires:* The specialization `greater<T>` shall be well-formed and well-defined.

13
```
template <class T>
  struct less_equal<experimental::fundamentals_v2::propagate_const<T>>;
```

14 For objects `p, q` of type `propagate_const<T>`, `less_equal<experimental::fundamentals_v2::propagate_const<T>>()(p, q)` shall evaluate to the same value as `less_equal<T>()(p.t_, q.t_)`.

15 *Requires:* The specialization `less_equal<T>` shall be well-formed and well-defined.

16
```
template <class T>
  struct greater_equal<experimental::fundamentals_v2::propagate_const<T>>;
```

17 For objects `p, q` of type `propagate_const<T>`, `greater_equal<experimental::fundamentals_v2::propagate_const<T>>()(p, q)` shall evaluate to the same value as `greater_equal<T>()(p.t_, q.t_)`.

18 *Requires:* The specialization `greater_equal<T>` shall be well-formed and well-defined.

4 Function objects [func]

4.1 Header `<experimental/functional>` synopsis [header.functional.synop]

```
#include <functional>

namespace std {
  namespace experimental {
  inline namespace fundamentals_v2 {

    // See C++14 §20.9.9, Function object binders
    template <class T> constexpr bool is_bind_expression_v
      = is_bind_expression<T>::value;
    template <class T> constexpr int is_placeholder_v
      = is_placeholder<T>::value;

    // 4.2, Class template function
    template<class> class function; // undefined
    template<class R, class... ArgTypes> class function<R(ArgTypes...)>;

    template<class R, class... ArgTypes>
    void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);

    template<class R, class... ArgTypes>
    bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;
    template<class R, class... ArgTypes>
    bool operator==(nullptr_t, const function<R(ArgTypes...)>&) noexcept;
    template<class R, class... ArgTypes>
    bool operator!=(const function<R(ArgTypes...)>&, nullptr_t) noexcept;
    template<class R, class... ArgTypes>
    bool operator!=(nullptr_t, const function<R(ArgTypes...)>&) noexcept;

    // 4.3, Searchers
    template<class ForwardIterator, class BinaryPredicate = equal_to<>>
      class default_searcher;

    template<class RandomAccessIterator,
             class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
             class BinaryPredicate = equal_to<>>
      class boyer_moore_searcher;

    template<class RandomAccessIterator,
             class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
             class BinaryPredicate = equal_to<>>
      class boyer_moore_horspool_searcher;


    template<class ForwardIterator, class BinaryPredicate = equal_to<>>
    default_searcher<ForwardIterator, BinaryPredicate>
    make_default_searcher(ForwardIterator pat_first, ForwardIterator pat_last,
```



```
                            BinaryPredicate pred = BinaryPredicate());

      template<class RandomAccessIterator,
               class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
               class BinaryPredicate = equal_to<>>
      boyer_moore_searcher<RandomAccessIterator, Hash, BinaryPredicate>
      make_boyer_moore_searcher(
          RandomAccessIterator pat_first, RandomAccessIterator pat_last,
          Hash hf = Hash(), BinaryPredicate pred = BinaryPredicate());

      template<class RandomAccessIterator,
               class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
               class BinaryPredicate = equal_to<>>
      boyer_moore_horspool_searcher<RandomAccessIterator, Hash, BinaryPredicate>
      make_boyer_moore_horspool_searcher(
          RandomAccessIterator pat_first, RandomAccessIterator pat_last,
          Hash hf = Hash(), BinaryPredicate pred = BinaryPredicate());

      // 4.4, Function template not_fn
      template <class F> unspecified not_fn(F&& f);

    } // namespace fundamentals_v2
    } // namespace experimental

    template<class R, class... ArgTypes, class Alloc>
    struct uses_allocator<experimental::function<R(ArgTypes...)>, Alloc>;

  } // namespace std
```

4.2 Class template `function` [func.wrap.func]

1 The specification of all declarations within this sub-clause 4.2 and its sub-clauses are the same as the corresponding declarations, as specified in C++14 §20.9.11.2, unless explicitly specified otherwise. [*Note:* `std::experimental::function` uses `std::bad_function_call`, there is no additional type `std::experimental::bad_function_call` — *end note*] .

```
  namespace std {
    namespace experimental {
    inline namespace fundamentals_v2 {

      template<class> class function; // undefined

      template<class R, class... ArgTypes>
      class function<R(ArgTypes...)> {
      public:
        using result_type = R;
        using argument_type = T1;
        using first_argument_type T1;
        using second_argument_type = T2;

        using allocator_type = erased_type;
```



```
    function() noexcept;
    function(nullptr_t) noexcept;
    function(const function&);
    function(function&&);
    template<class F> function(F);
    template<class A> function(allocator_arg_t, const A&) noexcept;
    template<class A> function(allocator_arg_t, const A&,
      nullptr_t) noexcept;
    template<class A> function(allocator_arg_t, const A&,
      const function&);
    template<class A> function(allocator_arg_t, const A&,
      function&&);
    template<class F, class A> function(allocator_arg_t, const A&, F);

    function& operator=(const function&);
    function& operator=(function&&);
    function& operator=(nullptr_t) noexcept;
    template<class F> function& operator=(F&&);
    template<class F> function& operator=(reference_wrapper<F>);

    ~function();

    void swap(function&);

    explicit operator bool() const noexcept;

    R operator()(ArgTypes...) const;

    const type_info& target_type() const noexcept;
    template<class T> T* target() noexcept;
    template<class T> const T* target() const noexcept;

    pmr::memory_resource* get_memory_resource() const noexcept;
  };

  template <class R, class... ArgTypes>
  bool operator==(const function<R(ArgTypes...)>&, nullptr_t) noexcept;
  template <class R, class... ArgTypes>
  bool operator==(nullptr_t, const function<R(ArgTypes...)>&) noexcept;

  template <class R, class... ArgTypes>
  bool operator!=(const function<R(ArgTypes...)>&, nullptr_t) noexcept;
  template <class R, class... ArgTypes>
  bool operator!=(nullptr_t, const function<R(ArgTypes...)>&) noexcept;

  template <class R, class... ArgTypes>
  void swap(function<R(ArgTypes...)>&, function<R(ArgTypes...)>&);

} // namespace fundamentals_v2
} // namespace experimental

template <class R, class... ArgTypes, class Alloc>
```

```
    struct uses_allocator<experimental::function<R(ArgTypes...)>, Alloc>
      : true_type { };

} // namespace std
```

4.2.1 `function` construct/copy/destroy [func.wrap.func.con]

1 When a `function` constructor that takes a first argument of type `allocator_arg_t` is invoked, the second argument is treated as a *type-erased allocator* (8.3). If the constructor moves or makes a copy of a function object (C++14 §20.9), including an instance of the `experimental::function` class template, then that move or copy is performed by *using-allocator construction* with allocator `get_memory_resource()`.

2 In the following descriptions, let *`ALLOCATOR_OF`*`(f)` be the allocator specified in the construction of `function f`, or the value of `experimental::pmr::get_default_resource()` at the time of the construction of `f` if no allocator was specified.

3 `function& operator=(const function& f);`

4 *Effects:* `function(allocator_arg, ALLOCATOR_OF(*this), f).swap(*this);`

5 *Returns:* `*this`.

6 `function& operator=(function&& f);`

7 *Effects:* `function(allocator_arg, ALLOCATOR_OF(*this), std::move(f)).swap(*this);`

8 *Returns:* `*this`.

9 `function& operator=(nullptr_t) noexcept;`

10 *Effects:* If `*this != nullptr`, destroys the target of `this`.

11 *Postconditions:* `!(*this)`. The memory resource returned by `get_memory_resource()` after the assignment is equivalent to the memory resource before the assignment. [*Note:* the address returned by `get_memory_resource()` might change — *end note*]

12 *Returns:* `*this`.

13 `template<class F> function& operator=(F&& f);`

14 *Effects:* `function(allocator_arg, ALLOCATOR_OF(*this), std::forward<F>(f)).swap(*this);`

15 *Returns:* `*this`.

16 *Remarks:* This assignment operator shall not participate in overload resolution unless `declval<decay_t<F>&>()` is Callable (C++14 §20.9.11.2) for argument types `ArgTypes...` and return type `R`.

17 `template<class F> function& operator=(reference_wrapper<F> f);`

18 *Effects:* `function(allocator_arg, ALLOCATOR_OF(*this), f).swap(*this);`

19 *Returns:* `*this`.

4.2.2 `function` modifiers [func.wrap.func.mod]

1 `void swap(function& other);`

2 *Requires:* `*this->get_memory_resource() == *other.get_memory_resource()`.

3 *Effects:* Interchanges the targets of `*this` and `other`.

4 *Remarks:* The allocators of `*this` and `other` are not interchanged.

4.3 Searchers [func.searchers]

1 This sub-clause provides function object types (C++14 §20.9) for operations that search for a sequence [`pat_first`, `pat_last`) in another sequence [`first`, `last`) that is provided to the object's function call operator. The first sequence (the pattern to be searched for) is provided to the object's constructor, and the second (the sequence to be searched) is provided to the function call operator.

2 Each specialization of a class template specified in this sub-clause 4.3 shall meet the `CopyConstructible` and `CopyAssignable` requirements. Template parameters named `ForwardIterator`, `ForwardIterator1`, `ForwardIterator2`, `RandomAccessIterator`, `RandomAccessIterator1`, `RandomAccessIterator2`, and `BinaryPredicate` of templates specified in this sub-clause 4.3 shall meet the same requirements and semantics as specified in C++14 §25.1. Template parameters named `Hash` shall meet the requirements as specified in C++14 §17.6.3.4.

3 The Boyer-Moore searcher implements the Boyer-Moore search algorithm. The Boyer-Moore-Horspool searcher implements the Boyer-Moore-Horspool search algorithm. In general, the Boyer-Moore searcher will use more memory and give better run-time performance than Boyer-Moore-Horspool.

4.3.1 Class template `default_searcher` [func.searchers.default]

```
template<class ForwardIterator1, class BinaryPredicate = equal_to<>>
class default_searcher {
public:
  default_searcher(ForwardIterator1 pat_first, ForwardIterator1 pat_last,
                   BinaryPredicate pred = BinaryPredicate());

  template<class ForwardIterator2>
    pair<ForwardIterator2, ForwardIterator2>
      operator()(ForwardIterator2 first, ForwardIterator2 last) const;

private:
  ForwardIterator1 pat_first_; // exposition only
  ForwardIterator1 pat_last_;  // exposition only
  BinaryPredicate  pred_;      // exposition only
};
```

1 `default_searcher(ForwardIterator pat_first, ForwardIterator pat_last,`
`BinaryPredicate pred = BinaryPredicate());`

2 *Effects:* Constructs a `default_searcher` object, initializing `pat_first_` with `pat_first`, `pat_last_` with `pat_last`, and `pred_` with `pred`.

3 *Throws:* Any exception thrown by the copy constructor of `BinaryPredicate` or `ForwardIterator1`.

```
4 template<class ForwardIterator2>
    pair<ForwardIterator2, ForwardIterator2>
      operator()(ForwardIterator2 first, ForwardIterator2 last) const;
```

5 *Returns:* A pair of iterators `i` and `j` such that

- `i == std::search(first, last, pat_first_, pat_last_, pred_)`, and
- if `i == last`, then `j == last`, otherwise `j == next(i, distance(pat_first_, pat_last_))`.

4.3.1.1 `default_searcher` creation functions [func.searchers.default.creation]

```
1 template<class ForwardIterator, class BinaryPredicate = equal_to<>>
   default_searcher<ForwardIterator, BinaryPredicate>
   make_default_searcher(ForwardIterator pat_first, ForwardIterator pat_last,
                         BinaryPredicate pred = BinaryPredicate());
```

2 *Effects:* Equivalent to `return default_searcher<ForwardIterator, BinaryPredicate>( pat_first, pat_last, pred);`

4.3.2 Class template `boyer_moore_searcher` [func.searchers.boyer_moore]

```
template<class RandomAccessIterator1,
         class Hash = hash<typename iterator_traits<RandomAccessIterator1>::value_type>,
         class BinaryPredicate = equal_to<>>
class boyer_moore_searcher {
public:
  boyer_moore_searcher(RandomAccessIterator1 pat_first, RandomAccessIterator1 pat_last,
                       Hash hf = Hash(), BinaryPredicate pred = BinaryPredicate());

  template<class RandomAccessIterator2>
    pair<RandomAccessIterator2, RandomAccessIterator2>
      operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;

private:
  RandomAccessIterator1 pat_first_; // exposition only
  RandomAccessIterator1 pat_last_;  // exposition only
  Hash                  hash_;      // exposition only
  BinaryPredicate       pred_;      // exposition only
};
```

```
1 boyer_moore_searcher(RandomAccessIterator1 pat_first, RandomAccessIterator1 pat_last,
   Hash hf = Hash(),
   BinaryPredicate pred = BinaryPredicate());
```

2 *Requires:* The value type of `RandomAccessIterator1` shall meet the `DefaultConstructible`, `CopyConstructible`, and `CopyAssignable` requirements.

3 *Requires:* For any two values `A` and `B` of the type `iterator_traits<RandomAccessIterator1>::value_type`, if `pred(A,B)==true`, then `hf(A)==hf(B)` shall be true.

4 *Effects:* Constructs a `boyer_moore_searcher` object, initializing `pat_first_` with `pat_first`, `pat_last_` with `pat_last`, `hash_` with `hf`, and `pred_` with `pred`.

5 *Throws:* Any exception thrown by the copy constructor of `RandomAccessIterator1`, or by the default constructor, copy constructor, or the copy assignment operator of the value type of `RandomAccessIterator1`, or the copy constructor or `operator()` of `BinaryPredicate` or `Hash`. May throw `bad_alloc` if additional memory needed for internal data structures cannot be allocated.

6
```
template<class RandomAccessIterator2>
    pair<RandomAccessIterator2, RandomAccessIterator2>
      operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;
```

7 *Requires:* `RandomAccessIterator1` and `RandomAccessIterator2` shall have the same value type.

8 *Effects:* Finds a subsequence of equal values in a sequence.

9 *Returns:* A pair of iterators `i` and `j` such that

- `i` is the first iterator in the range [`first`, `last - (pat_last_ - pat_first_)`) such that for every non-negative integer `n` less than `pat_last_ - pat_first_` the following condition holds: `pred(*(i + n), *(pat_first_ + n)) != false`, and
- `j == next(i, distance(pat_first_, pat_last_))`.

Returns `make_pair(first,first)` if [`pat_first_`, `pat_last_`) is empty, otherwise returns `make_pair(last,last)` if no such iterator is found.

10 *Complexity:* At most `(last - first) * (pat_last_ - pat_first_)` applications of the predicate.

4.3.2.1 `boyer_moore_searcher` creation functions [func.searchers.boyer_moore.creation]

1
```
template<class RandomAccessIterator,
    class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
    class BinaryPredicate = equal_to<>>
  boyer_moore_searcher<RandomAccessIterator, Hash, BinaryPredicate>
  make_boyer_moore_searcher(RandomAccessIterator pat_first, RandomAccessIterator pat_last,
                            Hash hf = Hash(), BinaryPredicate pred = BinaryPredicate());
```

2 *Effects:* Equivalent to `return boyer_moore_searcher<RandomAccessIterator, Hash, BinaryPredicate>( pat_first, pat_last, hf, pred);`

4.3.3 Class template `boyer_moore_horspool_searcher` [func.searchers.boyer_moore_horspool]

```
template<class RandomAccessIterator1,
         class Hash = hash<typename iterator_traits<RandomAccessIterator1>::value_type>,
         class BinaryPredicate = equal_to<>>
class boyer_moore_horspool_searcher {
public:
  boyer_moore_horspool_searcher(RandomAccessIterator1 pat_first, RandomAccessIterator1 pat_last,
                                Hash hf = Hash(), BinaryPredicate pred = BinaryPredicate());

  template<class RandomAccessIterator2>
    pair<RandomAccessIterator2, RandomAccessIterator2>
      operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;

private:
  RandomAccessIterator1 pat_first_; // exposition only
  RandomAccessIterator1 pat_last_;  // exposition only
  Hash                  hash_;      // exposition only
  BinaryPredicate       pred_;      // exposition only
};
```


1
```
boyer_moore_horspool_searcher(
    RandomAccessIterator1 pat_first, RandomAccessIterator1 pat_last,
    Hash hf = Hash(), BinaryPredicate pred = BinaryPredicate());
```

2 *Requires:* The value type of `RandomAccessIterator1` shall meet the `DefaultConstructible`, `CopyConstructible`, and `CopyAssignable` requirements.

3 *Requires:* For any two values `A` and `B` of the type iterator_traits<RandomAccessIterator1>::value_type, if `pred(A,B)==true`, then `hf(A)==hf(B)` shall be true.

4 *Effects:* Constructs a `boyer_moore_horspool_searcher` object, initializing `pat_first_` with `pat_first`, `pat_last_` with `pat_last`, `hash_` with `hf`, and `pred_` with `pred`.

5 *Throws:* Any exception thrown by the copy constructor of `RandomAccessIterator1`, or by the default constructor, copy constructor, or the copy assignment operator of the value type of `RandomAccessIterator1` or the copy constructor or `operator()` of `BinaryPredicate` or `Hash`. May throw `bad_alloc` if additional memory needed for internal data structures cannot be allocated..

6
```
template<class RandomAccessIterator2>
      pair<RandomAccessIterator2, RandomAccessIterator2>
        operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;
```

7 *Requires:* `RandomAccessIterator1` and `RandomAccessIterator2` shall have the same value type.

8 *Effects:* Finds a subsequence of equal values in a sequence.

9 *Returns:* A pair of iterators `i` and `j` such that

— `i` is the first iterator in the range [`first`, `last - (pat_last_ - pat_first_)`) such that for every non-negative integer `n` less than `pat_last_ - pat_first_` the following condition holds:
`pred(*(i + n), *(pat_first_ + n)) != false`, and

— `j == next(i, distance(pat_first_, pat_last_))`.

Returns `make_pair(first,first)` if [`pat_first_`, `pat_last_`) is empty, otherwise returns `make_pair(last,last)` if no such iterator is found.

10 *Complexity:* At most `(last - first) * (pat_last_ - pat_first_)` applications of the predicate.

4.3.3.1 `boyer_moore_horspool_searcher` creation functions [func.searchers.boyer_moore_horspool.creation]

1
```
template<class RandomAccessIterator,
            class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,
            class BinaryPredicate = equal_to<>>
    boyer_moore_horspool_searcher<RandomAccessIterator, Hash, BinaryPredicate>
    make_boyer_moore_horspool_searcher(
        RandomAccessIterator pat_first, RandomAccessIterator pat_last,
        Hash hf = Hash(), BinaryPredicate pred = BinaryPredicate());
```

2 *Effects:* Equivalent to

```
return boyer_moore_horspool_searcher<RandomAccessIterator, Hash, BinaryPredicate>(
pat_first, pat_last, hf, pred);
```

4.4 Function template not_fn [func.not_fn]

1 template <class F> *unspecified* not_fn(F&& f);

2 In the text that follows:

- — FD is the type decay_t<F>,
- — fd is an lvalue of type FD constructed from std::forward<F>(f),
- — fn is a forwarding call wrapper created as a result of not_fn(f),

3 *Requires:* is_constructible<FD, F>::value shall be true. fd shall be a callable object (C++14 §20.9.1).

4 *Returns:* A forwarding call wrapper fn such that the expression fn(a1, a2, ..., aN) is equivalent to !*INVOKE*(fd, a1, a2, ..., aN) (C++14 §20.9.2).

5 *Throws:* Nothing unless the construction of fd throws an exception.

6 *Remarks:* The return type shall satisfy the requirements of MoveConstructible. If FD satisfies the requirements of CopyConstructible, then the return type shall satisfy the requirements of CopyConstructible. [*Note:* This implies that FD is MoveConstructible. — *end note*]

7 [*Note:* Function template not_fn can usually provide a better solution than using the negators not1 and not2 — *end note*]

5 Optional objects [optional]

5.1 In general [optional.general]

1 This subclause describes class template `optional` that represents *optional objects*. An *optional object for object types* is an object that contains the storage for another object and manages the lifetime of this contained object, if any. The contained object may be initialized after the optional object has been initialized, and may be destroyed before the optional object has been destroyed. The initialization state of the contained object is tracked by the optional object.

5.2 Header `<experimental/optional>` synopsis [optional.synop]

```
namespace std {
  namespace experimental {
  inline namespace fundamentals_v2 {

    // 5.3, optional for object types
    template <class T> class optional;

    // 5.4, In-place construction
    struct in_place_t{};
    constexpr in_place_t in_place{};

    // 5.5, No-value state indicator
    struct nullopt_t{see below};
    constexpr nullopt_t nullopt(unspecified);

    // 5.6, Class bad_optional_access
    class bad_optional_access;

    // 5.7, Relational operators
    template <class T>
      constexpr bool operator==(const optional<T>&, const optional<T>&);
    template <class T>
      constexpr bool operator!=(const optional<T>&, const optional<T>&);
    template <class T>
      constexpr bool operator<(const optional<T>&, const optional<T>&);
    template <class T>
      constexpr bool operator>(const optional<T>&, const optional<T>&);
    template <class T>
      constexpr bool operator<=(const optional<T>&, const optional<T>&);
    template <class T>
      constexpr bool operator>=(const optional<T>&, const optional<T>&);

    // 5.8, Comparison with nullopt
    template <class T> constexpr bool operator==(const optional<T>&, nullopt_t) noexcept;
    template <class T> constexpr bool operator==(nullopt_t, const optional<T>&) noexcept;
    template <class T> constexpr bool operator!=(const optional<T>&, nullopt_t) noexcept;
    template <class T> constexpr bool operator!=(nullopt_t, const optional<T>&) noexcept;
    template <class T> constexpr bool operator<(const optional<T>&, nullopt_t) noexcept;
```



```
    template <class T> constexpr bool operator<(nullopt_t, const optional<T>&) noexcept;
    template <class T> constexpr bool operator<=(const optional<T>&, nullopt_t) noexcept;
    template <class T> constexpr bool operator<=(nullopt_t, const optional<T>&) noexcept;
    template <class T> constexpr bool operator>(const optional<T>&, nullopt_t) noexcept;
    template <class T> constexpr bool operator>(nullopt_t, const optional<T>&) noexcept;
    template <class T> constexpr bool operator>=(const optional<T>&, nullopt_t) noexcept;
    template <class T> constexpr bool operator>=(nullopt_t, const optional<T>&) noexcept;

    // 5.9, Comparison with T
    template <class T> constexpr bool operator==(const optional<T>&, const T&);
    template <class T> constexpr bool operator==(const T&, const optional<T>&);
    template <class T> constexpr bool operator!=(const optional<T>&, const T&);
    template <class T> constexpr bool operator!=(const T&, const optional<T>&);
    template <class T> constexpr bool operator<(const optional<T>&, const T&);
    template <class T> constexpr bool operator<(const T&, const optional<T>&);
    template <class T> constexpr bool operator<=(const optional<T>&, const T&);
    template <class T> constexpr bool operator<=(const T&, const optional<T>&);
    template <class T> constexpr bool operator>(const optional<T>&, const T&);
    template <class T> constexpr bool operator>(const T&, const optional<T>&);
    template <class T> constexpr bool operator>=(const optional<T>&, const T&);
    template <class T> constexpr bool operator>=(const T&, const optional<T>&);

    // 5.10, Specialized algorithms
    template <class T> void swap(optional<T>&, optional<T>&) noexcept(see below);
    template <class T> constexpr optional<see below> make_optional(T&&);

  } // namespace fundamentals_v2
  } // namespace experimental

  // 5.11, Hash support
  template <class T> struct hash;
  template <class T> struct hash<experimental::optional<T>>;

} // namespace std
```

1 A program that necessitates the instantiation of template `optional` for a reference type, or for possibly cv-qualified types `in_place_t` or `nullopt_t` is ill-formed.

5.3 `optional` for object types [optional.object]

```
template <class T>
class optional
{
public:
  using value_type = T;

  // 5.3.1, Constructors
  constexpr optional() noexcept;
  constexpr optional(nullopt_t) noexcept;
  optional(const optional&);
  optional(optional&&) noexcept(see below);
  constexpr optional(const T&);
```

```
    constexpr optional(T&&);
    template <class... Args> constexpr explicit optional(in_place_t, Args&&...);
    template <class U, class... Args>
      constexpr explicit optional(in_place_t, initializer_list<U>, Args&&...);
    template <class U> constexpr optional(U&&);
    template <class U> optional(const optional<U>&);
    template <class U> optional(optional<U>&&);

    // 5.3.2, Destructor
    ~optional();

    // 5.3.3, Assignment
    optional& operator=(nullopt_t) noexcept;
    optional& operator=(const optional&);
    optional& operator=(optional&&) noexcept(see below);
    template <class U> optional& operator=(U&&);
    template <class U> optional& operator=(const optional<U>&);
    template <class U> optional& operator=(optional<U>&&);
    template <class... Args> void emplace(Args&&...);
    template <class U, class... Args>
      void emplace(initializer_list<U>, Args&&...);

    // 5.3.4, Swap
    void swap(optional&) noexcept(see below);

    // 5.3.5, Observers
    constexpr T const* operator ->() const;
    constexpr T* operator ->();
    constexpr T const& operator *() const &;
    constexpr T& operator *() &;
    constexpr T&& operator *() &&;
    constexpr const T&& operator *() const &&;
    constexpr explicit operator bool() const noexcept;
    constexpr T const& value() const &;
    constexpr T& value() &;
    constexpr T&& value() &&;
    constexpr const T&& value() const &&;
    template <class U> constexpr T value_or(U&&) const &;
    template <class U> constexpr T value_or(U&&) &&;

  private:
    T*   val;  // exposition only
  };
```

1 Any instance of `optional<T>` at any given time either contains a value or does not contain a value. When an instance of `optional<T>` *contains a value*, it means that an object of type `T`, referred to as the optional object's *contained value*, is allocated within the storage of the optional object. Implementations are not permitted to use additional storage, such as dynamic memory, to allocate its contained value. The contained value shall be allocated in a region of the `optional<T>` storage suitably aligned for the type `T`. It is implementation-defined whether over-aligned types are supported (C++14 §3.11). When an object of type `optional<T>` is contextually converted to `bool`, the conversion returns `true` if the object contains a value; otherwise the conversion returns `false`.

2 Member `val` is provided for exposition only. When an `optional<T>` object contains a value, `val` points to the contained value.

3 `T` shall be an object type and shall satisfy the requirements of `Destructible` (Table 24).

5.3.1 Constructors [optional.object.ctor]

1
```
constexpr optional() noexcept;
constexpr optional(nullopt_t) noexcept;
```

2 *Postconditions:* `*this` does not contain a value.

3 *Remarks:* No contained value is initialized. For every object type `T` these constructors shall be `constexpr` constructors (C++14 §7.1.5).

4
```
optional(const optional<T>& rhs);
```

5 *Requires:* `is_copy_constructible_v<T>` is `true`.

6 *Effects:* If *`rhs`* contains a value, initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `*rhs`.

7 *Postconditions:* `bool(rhs) == bool(*this)`.

8 *Throws:* Any exception thrown by the selected constructor of `T`.

9
```
optional(optional<T>&& rhs) noexcept(see below);
```

10 *Requires:* `is_move_constructible_v<T>` is `true`.

11 *Effects:* If *`rhs`* contains a value, initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `std::move(*rhs)`. The value of `bool(rhs)` is unchanged.

12 *Postconditions:* `bool(rhs) == bool(*this)`.

13 *Throws:* Any exception thrown by the selected constructor of `T`.

14 *Remarks:* The expression inside `noexcept` is equivalent to:

```
is_nothrow_move_constructible_v<T>
```

15
```
constexpr optional(const T& v);
```

16 *Requires:* `is_copy_constructible_v<T>` is `true`.

17 *Effects:* Initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression *`v`*.

18 *Postconditions:* `*this` contains a value.

19 *Throws:* Any exception thrown by the selected constructor of `T`.

20 *Remarks:* If `T`'s selected constructor is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

21 `constexpr optional(T&& v);`

22 *Requires:* `is_move_constructible_v<T>` is `true`.

23 *Effects:* Initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `std::move(v)`.

24 *Postconditions:* `*this` contains a value.

25 *Throws:* Any exception thrown by the selected constructor of `T`.

26 *Remarks:* If `T`'s selected constructor is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

27 `template <class... Args> constexpr explicit optional(in_place_t, Args&&... args);`

28 *Requires:* `is_constructible_v<T, Args&&...>` is `true`.

29 *Effects:* Initializes the contained value as if direct-non-list-initializing an object of type `T` with the arguments `std::forward<Args>(args)...`.

30 *Postconditions:* `*this` contains a value.

31 *Throws:* Any exception thrown by the selected constructor of `T`.

32 *Remarks:* If `T`'s constructor selected for the initialization is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

33
```
template <class U, class... Args>
   constexpr explicit optional(in_place_t, initializer_list<U> il, Args&&... args);
```

34 *Requires:* `is_constructible_v<T, initializer_list<U>&, Args&&...>` is `true`.

35 *Effects:* Initializes the contained value as if direct-non-list-initializing an object of type `T` with the arguments `il, std::forward<Args>(args)...`.

36 *Postconditions:* `*this` contains a value.

37 *Throws:* Any exception thrown by the selected constructor of `T`.

38 *Remarks:* The function shall not participate in overload resolution unless `is_constructible_v<T, initializer_list<U>&, Args&&...>` is `true`. If `T`'s constructor selected for the initialization is a `constexpr` constructor, this constructor shall be a `constexpr` constructor.

[*Note:* The following constructors are conditionally specified as `explicit`. This is typically implemented by declaring two such constructors, of which at most one participates in overload resolution. — *end note*]

39
```
template <class U>
  constexpr optional(U&& v);
```

40 *Effects:* Initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `std::forward<U>(v)`.

41 *Postconditions:* `*this` contains a value.

42 *Throws:* Any exception thrown by the selected constructor of `T`.

43 *Remarks:* If `T`'s selected constructor is a `constexpr` constructor, this constructor shall be a `constexpr` constructor. This constructor shall not participate in overload resolution unless `is_constructible_v<T, U&&>` is `true` and `decay_t<U>` is not the same type as `T`. The constructor is `explicit` if and only if `is_convertible_v<U&&, T>` is `false`.

44
```
template <class U>
  optional(const optional<U>& rhs);
```

45 *Effects:* If `rhs` contains a value, initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `*rhs`.

46 *Postconditions:* `bool(rhs) == bool(*this)`.

47 *Throws:* Any exception thrown by the selected constructor of `T`.

48 *Remarks:* This constructor shall not participate in overload resolution unless `is_constructible_v<T, const U&>` is `true`, `is_same<decay_t<U>, T>` is `false`, `is_constructible_v<T, const optional<U>&>` is `false` and `is_convertible_v<const optional<U>&, T>` is `false`. The constructor is `explicit` if and only if `is_convertible_v<const U&, T>` is `false`.

49
```
template <class U>
  optional(optional<U>&& rhs);
```

50 *Effects:* If `rhs` contains a value, initializes the contained value as if direct-non-list-initializing an object of type `T` with the expression `std::move(*rhs)`. `bool(rhs)` is unchanged.

51 *Postconditions:* `bool(rhs) == bool(*this)`.

52 *Throws:* Any exception thrown by the selected constructor of `T`.

53 *Remarks:* This constructor shall not participate in overload resolution unless `is_constructible_v<T, U&&>` is `true`, `is_same<decay_t<U>, T>` is `false`, `is_constructible_v<T, optional<U>&&>` is `false` and `is_convertible_v<optional<U>&&, T>` is `false` and `U` is not the same type as `T`. The constructor is `explicit` if and only if `is_convertible_v<U&&, T>` is `false`.

5.3.2 Destructor [optional.object.dtor]

1
```
~optional();
```

2 *Effects:* If `is_trivially_destructible_v<T> != true` and `*this` contains a value, calls `val->T::~T()`.

3 *Remarks:* If `is_trivially_destructible_v<T> == true` then this destructor shall be a trivial destructor.

5.3.3 Assignment [optional.object.assign]

1 `optional<T>& operator=(nullopt_t) noexcept;`

2 *Effects:* If `*this` contains a value, calls `val->T::~T()` to destroy the contained value; otherwise no effect.

3 *Returns:* `*this`.

4 *Postconditions:* `*this` does not contain a value.

5 `optional<T>& operator=(const optional<T>& rhs);`

6 *Requires:* `is_copy_constructible_v<T>` is `true` and `is_copy_assignable_v<T>` is `true`.

7 *Effects:*

Table 5 — `optional::operator=(const optional&)` effects

	`*this` contains a value	**`*this` does not contain a value**
`rhs` contains a value	assigns `*rhs` to the contained value	initializes the contained value as if direct-non-list-initializing an object of type `T` with `*rhs`
`rhs` does not contain a value	destroys the contained value by calling `val->T::~T()`	no effect

8 *Returns:* `*this`.

9 *Postconditions:* `bool(rhs) == bool(*this)`.

10 *Remarks:* If any exception is thrown, the result of the expression `bool(*this)` remains unchanged. If an exception is thrown during the call to `T`'s copy constructor, no effect. If an exception is thrown during the call to `T`'s copy assignment, the state of its contained value is as defined by the exception safety guarantee of `T`'s copy assignment.

11 `optional<T>& operator=(optional<T>&& rhs) noexcept(see below);`

12 *Requires:* `is_move_constructible_v<T>` is `true` and `is_move_assignable_v<T>` is `true`.

13 *Effects:* The result of the expression `bool(rhs)` remains unchanged.

Table 6 — `optional::operator=(optional&&)` effects

	`*this` contains a value	**`*this` does not contain a value**
`rhs` contains a value	assigns `std::move(*rhs)` to the contained value	initializes the contained value as if direct-non-list-initializing an object of type `T` with `std::move(*rhs)`
`rhs` does not contain a value	destroys the contained value by calling `val->T::~T()`	no effect

14 *Returns:* `*this`.

15 *Postconditions:* `bool(rhs) == bool(*this)`.

16 *Remarks:* The expression inside `noexcept` is equivalent to:

```
is_nothrow_move_assignable_v<T> && is_nothrow_move_constructible_v<T>
```

If any exception is thrown, the result of the expression `bool(*this)` remains unchanged. If an exception is thrown during the call to `T`'s move constructor, the state of `*rhs.val` is determined by the exception safety guarantee of `T`'s move constructor. If an exception is thrown during the call to `T`'s move assignment, the state of `*val` and `*rhs.val` is determined by the exception safety guarantee of `T`'s move assignment.

17 template <class U> optional<T>& operator=(U&& *v*);

18 *Requires:* is_constructible_v<T, U> is true and is_assignable_v<T&, U> is true.

19 *Effects:* If *this contains a value, assigns std::forward<U>(*v*) to the contained value; otherwise initializes the contained value as if direct-non-list-initializing object of type T with std::forward<U>(*v*).

20 *Returns:* *this.

21 *Postconditions:* *this contains a value.

22 *Remarks:* If any exception is thrown, the result of the expression bool(*this) remains unchanged. If an exception is thrown during the call to T's constructor, the state of *v* is determined by the exception safety guarantee of T's constructor. If an exception is thrown during the call to T's assignment, the state of **val* and *v* is determined by the exception safety guarantee of T's assignment.

The function shall not participate in overload resolution unless decay_t<U> is not nullopt_t and decay_t<U> is not a specialization of optional.

23 template <class U> optional<T>& operator=(const optional<U>& rhs);

24 *Requires:* is_constructible_v<T, const U&> is true and is_assignable_v<T&, const U&> is true.

25 *Effects:*

Table 7 — optional::operator=(const optional<U>&) effects

	***this contains a value**	***this does not contain a value**
rhs contains a value	assigns *rhs to the contained value	initializes the contained value as if direct-non-list-initializing an object of type T with *rhs
rhs does not contain a value	destroys the contained value by calling val->T::~T()	no effect

26 *Returns:* *this.

27 *Postconditions:* bool(rhs) == bool(*this).

28 *Remarks:* If any exception is thrown, the result of the expression bool(*this) remains unchanged. If an exception is thrown during the call to T's constructor, the state of *rhs.val is determined by the exception safety guarantee of T's constructor. If an exception is thrown during the call to T's assignment, the state of *val and *rhs.val is determined by the exception safety guarantee of T's assignment. The function shall not participate in overload resolution unless is_same_v<decay_t<U>, T> is false.

29 template <class U> optional<T>& operator=(optional<U>&& rhs);

30 *Requires:* is_constructible_v<T, U> is true and is_assignable_v<T&, U> is true.

31 *Effects:* The result of the expression bool(rhs) remains unchanged.

Table 8 — optional::operator=(optional<U>&&) effects

	***this contains a value**	***this does not contain a value**
rhs contains a value	assigns std::move(*rhs) to the contained value	initializes the contained value as if direct-non-list-initializing an object of type T with std::move(*rhs)
rhs does not contain a value	destroys the contained value by calling val->T::~T()	no effect

32 *Returns:* *this.

33 *Postconditions:* bool(rhs) == bool(*this).

34 *Remarks:* If any exception is thrown, the result of the expression bool(*this) remains unchanged. If an exception is thrown during the call to T's constructor, the state of *rhs.val is determined by the exception safety guarantee of T's constructor. If an exception is thrown during the call to T's assignment, the state of *val and *rhs.val is determined by the exception safety guarantee of T's assignment. The function shall not participate in overload resolution unless is_same_v<decay_t<U>, T> is false.

35 template <class... Args> void emplace(Args&&... *args*);

36 *Requires:* is_constructible_v<T, Args&&...> is true.

37 *Effects:* Calls *this = nullopt. Then initializes the contained value as if direct-non-list-initializing an object of type T with the arguments std::forward<Args>(*args*)....

38 *Postconditions:* *this contains a value.

39 *Throws:* Any exception thrown by the selected constructor of T.

40 *Remarks:* If an exception is thrown during the call to T's constructor, *this does not contain a value, and the previous **val* (if any) has been destroyed.

41 template <class U, class... Args> void emplace(initializer_list<U> *il*, Args&&... *args*);

42 *Effects:* Calls *this = nullopt. Then initializes the contained value as if direct-non-list-initializing an object of type T with the arguments *il*, std::forward<Args>(*args*)....

43 *Postconditions:* *this contains a value.

44 *Throws:* Any exception thrown by the selected constructor of T.

45 *Remarks:* If an exception is thrown during the call to T's constructor, *this does not contain a value, and the previous **val* (if any) has been destroyed.

The function shall not participate in overload resolution unless is_constructible_v<T, initializer_list<U>&, Args&&...> is true.

5.3.4 Swap [optional.object.swap]

1 `void swap(optional<T>& rhs) noexcept(see below);`

2 *Requires:* Lvalues of type `T` shall be swappable and `is_move_constructible_v<T>` is `true`.

3 *Effects:*

Table 9 — `optional::swap(optional&)` effects

	`*this` contains a value	**`*this` does not contain a value**
`rhs` contains a value	calls `swap(*(*this), *rhs)`	initializes the contained value of `*this` as if direct-non-list-initializing an object of type `T` with the expression `std::move(*rhs)`, followed by `rhs.val->T::~T()`; postcondition is that `*this` contains a value and `rhs` does not contain a value
`rhs` does not contain a value	initializes the contained value of `rhs` as if direct-non-list-initializing an object of type `T` with the expression `std::move(*(*this))`, followed by `val->T::~T()`; postcondition is that `*this` does not contain a value and `rhs` contains a value	no effect

4 *Throws:* Any exceptions that the expressions in the Effects element throw.

5 *Remarks:* The expression inside `noexcept` is equivalent to:

```
is_nothrow_move_constructible_v<T> && noexcept(swap(declval<T&>(), declval<T&>()))
```

If any exception is thrown, the results of the expressions `bool(*this)` and `bool(rhs)` remain unchanged. If an exception is thrown during the call to function `swap` the state of `*val` and `*rhs.val` is determined by the exception safety guarantee of `swap` for lvalues of `T`. If an exception is thrown during the call to `T`'s move constructor, the state of `*val` and `*rhs.val` is determined by the exception safety guarantee of `T`'s move constructor.

5.3.5 Observers [optional.object.observe]

1
```
constexpr T const* operator->() const;
constexpr T* operator->();
```

2 *Requires:* `*this` contains a value.

3 *Returns:* `val`.

4 *Throws:* Nothing.

5 *Remarks:* Unless `T` is a user-defined type with overloaded unary `operator&`, these functions shall be `constexpr` functions.

6
```
constexpr T const& operator*() const &;
constexpr T& operator*() &;
```

7 *Requires:* `*this` contains a value.

8 *Returns:* `*val`.

9 *Throws:* Nothing.

10 *Remarks:* These functions shall be `constexpr` functions.

11
```
constexpr T&& operator*() &&;
constexpr const T&& operator*() const &&;
```

12 *Requires:* `*this` contains a value.

13 *Effects:* Equivalent to `return std::move(*val);`

14
```
constexpr explicit operator bool() const noexcept;
```

15 *Returns:* `true` if and only if `*this` contains a value.

16 *Remarks:* This function shall be a `constexpr` function.

17
```
constexpr T const& value() const &;
constexpr T& value() &;
```

18 *Effects:* Equivalent to `return bool(*this) ? *val : throw bad_optional_access();`

19
```
constexpr T&& value() &&;
constexpr const T&& value() const &&;
```

20 *Effects:* Equivalent to `return bool(*this) ? std::move(*val) : throw bad_optional_access();`

21
```
template <class U> constexpr T value_or(U&& v) const &;
```

22 *Effects:* Equivalent to `return bool(*this) ? **this : static_cast<T>(std::forward<U>(v));`

23 *Remarks:* If `is_copy_constructible_v<T> && is_convertible_v<U&&, T>` is `false`, the program is ill-formed.

24
```
template <class U> constexpr T value_or(U&& v) &&;
```

25 *Effects:* Equivalent to `return bool(*this) ? std::move(**this) : static_cast<T>(std::forward<U>(v));`

26 *Remarks:* If `is_move_constructible_v<T> && is_convertible_v<U&&, T>` is `false`, the program is ill-formed.

5.4 In-place construction [optional.inplace]

1
```
struct in_place_t{};
constexpr in_place_t in_place{};
```

2 The struct `in_place_t` is an empty structure type used as a unique type to disambiguate constructor and function overloading. Specifically, `optional<T>` has a constructor with `in_place_t` as the first parameter followed by a parameter pack; this indicates that `T` should be constructed in-place (as if by a call to a placement new expression) with the forwarded pack expansion as arguments for the initialization of `T`.

5.5 No-value state indicator [optional.nullopt]

1
```
struct nullopt_t{see below};
constexpr nullopt_t nullopt(unspecified);
```

2 The struct `nullopt_t` is an empty structure type used as a unique type to indicate the state of not containing a value for `optional` objects. In particular, `optional<T>` has a constructor with `nullopt_t` as a single argument; this indicates that an optional object not containing a value shall be constructed.

3 Type `nullopt_t` shall not have a default constructor. It shall be a literal type. Constant `nullopt` shall be initialized with an argument of literal type.

5.6 Class `bad_optional_access` [optional.bad_optional_access]

```
class bad_optional_access : public logic_error {
public:
  bad_optional_access();
};
```

1 The class `bad_optional_access` defines the type of objects thrown as exceptions to report the situation where an attempt is made to access the value of an optional object that does not contain a value.

2 `bad_optional_access();`

3 *Effects:* Constructs an object of class `bad_optional_access`.

4 *Postconditions:* `what()` returns an implementation-defined NTBS.

5.7 Relational operators [optional.relops]

1 `template <class T> constexpr bool operator==(const optional<T>& x, const optional<T>& y);`

2 *Requires:* `T` shall meet the requirements of `EqualityComparable`.

3 *Returns:* If `bool(x) != bool(y)`, `false`; otherwise if `bool(x) == false`, `true`; otherwise `*x == *y`.

4 *Remarks:* Specializations of this function template for which `*x == *y` is a core constant expression, shall be `constexpr` functions.

5 `template <class T> constexpr bool operator!=(const optional<T>& x, const optional<T>& y);`

6 *Returns:* `!(x == y)`.

7 `template <class T> constexpr bool operator<(const optional<T>& x, const optional<T>& y);`

8 *Requires:* `*x < *y` shall be well-formed and its result shall be convertible to `bool`.

9 *Returns:* If `!y`, `false`; otherwise, if `!x`, `true`; otherwise `*x < *y`.

10 *Remarks:* Specializations of this function template for which `*x < *y` is a core constant expression, shall be `constexpr` functions.

11 `template <class T> constexpr bool operator>(const optional<T>& x, const optional<T>& y);`

12 *Returns:* `y < x`.

13 `template <class T> constexpr bool operator<=(const optional<T>& x, const optional<T>& y);`

14 *Returns:* `!(y < x)`.

15 `template <class T> constexpr bool operator>=(const optional<T>& x, const optional<T>& y);`

16 *Returns:* `!(x < y)`.

5.8 Comparison with `nullopt` [optional.nullops]

1
```
template <class T> constexpr bool operator==(const optional<T>& x, nullopt_t) noexcept;
template <class T> constexpr bool operator==(nullopt_t, const optional<T>& x) noexcept;
```

2 *Returns:* `!x`.

3
```
template <class T> constexpr bool operator!=(const optional<T>& x, nullopt_t) noexcept;
template <class T> constexpr bool operator!=(nullopt_t, const optional<T>& x) noexcept;
```

4 *Returns:* `bool(x)`.

5
```
template <class T> constexpr bool operator<(const optional<T>& x, nullopt_t) noexcept;
```

6 *Returns:* `false`.

7
```
template <class T> constexpr bool operator<(nullopt_t, const optional<T>& x) noexcept;
```

8 *Returns:* `bool(x)`.

9
```
template <class T> constexpr bool operator<=(const optional<T>& x, nullopt_t) noexcept;
```

10 *Returns:* `!x`.

11
```
template <class T> constexpr bool operator<=(nullopt_t, const optional<T>& x) noexcept;
```

12 *Returns:* `true`.

13
```
template <class T> constexpr bool operator>(const optional<T>& x, nullopt_t) noexcept;
```

14 *Returns:* `bool(x)`.

15
```
template <class T> constexpr bool operator>(nullopt_t, const optional<T>& x) noexcept;
```

16 *Returns:* `false`.

17
```
template <class T> constexpr bool operator>=(const optional<T>& x, nullopt_t) noexcept;
```

18 *Returns:* `true`.

19
```
template <class T> constexpr bool operator>=(nullopt_t, const optional<T>& x) noexcept;
```

20 *Returns:* `!x`.

5.9 Comparison with `T` [optional.comp_with_t]

1
```
template <class T> constexpr bool operator==(const optional<T>& x, const T& v);
```

2 *Returns:* `bool(x) ? *x == v : false`.

3
```
template <class T> constexpr bool operator==(const T& v, const optional<T>& x);
```

4 *Returns:* `bool(x) ? v == *x : false`.

5
```
template <class T> constexpr bool operator!=(const optional<T>& x, const T& v);
```

6 *Returns:* `bool(x) ? !(*x == v) : true`.

7
```
template <class T> constexpr bool operator!=(const T& v, const optional<T>& x);
```

8 *Returns:* `bool(x) ? !(v == *x) : true`.

9
```
template <class T> constexpr bool operator<(const optional<T>& x, const T& v);
```

10 *Returns:* `bool(x) ? *x < v : true`.

11 `template <class T> constexpr bool operator<(const T& v, const optional<T>& x);`

12 *Returns:* `bool(x) ? v < *x : false`.

13 `template <class T> constexpr bool operator<=(const optional<T>& x, const T& v);`

14 *Returns:* `!(x > v)`.

15 `template <class T> constexpr bool operator<=(const T& v, const optional<T>& x);`

16 *Returns:* `!(v > x)`.

17 `template <class T> constexpr bool operator>(const optional<T>& x, const T& v);`

18 *Returns:* `bool(x) ? v < *x : false`.

19 `template <class T> constexpr bool operator>(const T& v, const optional<T>& x);`

20 *Returns:* `bool(x) ? *x < v : true`.

21 `template <class T> constexpr bool operator>=(const optional<T>& x, const T& v);`

22 *Returns:* `!(x < v)`.

23 `template <class T> constexpr bool operator>=(const T& v, const optional<T>& x);`

24 *Returns:* `!(v < x)`.

5.10 Specialized algorithms [optional.specalg]

1 `template <class T> void swap(optional<T>& x, optional<T>& y) noexcept(noexcept(x.swap(y)));`

2 *Effects:* Calls `x.swap(y)`.

3 `template <class T> constexpr optional<decay_t<T>> make_optional(T&& v);`

4 *Returns:* `optional<decay_t<T>>(std::forward<T>(v))`.

5.11 Hash support [optional.hash]

1 `template <class T> struct hash<experimental::optional<T>>;`

2 *Requires:* The template specialization `hash<T>` shall meet the requirements of class template `hash` (C++14 §20.9.12). The template specialization `hash<optional<T>>` shall meet the requirements of class template `hash`. For an object `o` of type `optional<T>`, if `bool(o) == true`, `hash<optional<T>>()(o)` shall evaluate to the same value as `hash<T>()(*o)`; otherwise it evaluates to an unspecified value.

6 Class `any` [any]

1 This section describes components that C++ programs may use to perform operations on objects of a discriminated type.

2 [*Note:* The discriminated type may contain values of different types but does not attempt conversion between them, i.e. `5` is held strictly as an `int` and is not implicitly convertible either to `"5"` or to `5.0`. This indifference to interpretation but awareness of type effectively allows safe, generic containers of single values, with no scope for surprises from ambiguous conversions. — *end note*]

6.1 Header <experimental/any> synopsis [any.synop]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  class bad_any_cast : public bad_cast
  {
  public:
    virtual const char* what() const noexcept;
  };

  class any
  {
  public:
    // 6.3.1, any construct/destruct
    any() noexcept;

    any(const any& other);
    any(any&& other) noexcept;

    template <class ValueType>
      any(ValueType&& value);

    ~any();

    // 6.3.2, any assignments
    any& operator=(const any& rhs);
    any& operator=(any&& rhs) noexcept;

    template <class ValueType>
      any& operator=(ValueType&& rhs);

    // 6.3.3, any modifiers
    void clear() noexcept;
    void swap(any& rhs) noexcept;

    // 6.3.4, any observers
    bool empty() const noexcept;
    const type_info& type() const noexcept;
  };
```



```
  // 6.4, Non-member functions
  void swap(any& x, any& y) noexcept;

  template<class ValueType>
    ValueType any_cast(const any& operand);
  template<class ValueType>
    ValueType any_cast(any& operand);
  template<class ValueType>
    ValueType any_cast(any&& operand);

  template<class ValueType>
    const ValueType* any_cast(const any* operand) noexcept;
  template<class ValueType>
    ValueType* any_cast(any* operand) noexcept;

} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

6.2 Class `bad_any_cast` [any.bad_any_cast]

1 Objects of type `bad_any_cast` are thrown by a failed `any_cast`.

6.3 Class `any` [any.class]

1 An object of class `any` stores an instance of any type that satisfies the constructor requirements or is empty, and this is referred to as the *state* of the class `any` object. The stored instance is called the *contained object*. Two states are equivalent if they are either both empty or if both are not empty and if the contained objects are equivalent.

2 The non-member `any_cast` functions provide type-safe access to the contained object.

3 Implementations should avoid the use of dynamically allocated memory for a small contained object. [*Example:* where the object constructed is holding only an int. — *end example*] Such small-object optimization shall only be applied to types `T` for which `is_nothrow_move_constructible_v<T>` is true.

6.3.1 `any` construct/destruct [any.cons]

1 `any() noexcept;`

2 *Postconditions:* `this->empty()`.

3 `any(const any& other);`

4 *Effects:* Constructs an object of type `any` with an equivalent state as `other`.

5 *Throws:* Any exceptions arising from calling the selected constructor of the contained object.

6 `any(any&& other) noexcept;`

7 *Effects:* Constructs an object of type `any` with a state equivalent to the original state of `other`.

8 *Postconditions:* `other` is left in a valid but otherwise unspecified state.

9
```
template<class ValueType>
  any(ValueType&& value);
```

10 Let `T` be equal to `decay_t<ValueType>`.

11 *Requires:* `T` shall satisfy the `CopyConstructible` requirements, except for the requirements for `MoveConstructible`. If `is_copy_constructible_v<T>` is false, the program is ill-formed.

12 *Effects:* If `is_constructible_v<T, ValueType&&>` is true, constructs an object of type `any` that contains an object of type `T` direct-initialized with `std::forward<ValueType>(value)`. Otherwise, constructs an object of type `any` that contains an object of type `T` direct-initialized with `value`.

13 *Remarks:* This constructor shall not participate in overload resolution if `decay_t<ValueType>` is the same type as `any`.

14 *Throws:* Any exception thrown by the selected constructor of `T`.

15
```
~any();
```

16 *Effects:* `clear()`.

6.3.2 `any` assignments [any.assign]

1
```
any& operator=(const any& rhs);
```

2 *Effects:* `any(rhs).swap(*this)`. No effects if an exception is thrown.

3 *Returns:* `*this`.

4 *Throws:* Any exceptions arising from the copy constructor of the contained object.

5
```
any& operator=(any&& rhs) noexcept;
```

6 *Effects:* `any(std::move(rhs)).swap(*this)`.

7 *Returns:* `*this`.

8 *Postconditions:* The state of `*this` is equivalent to the original state of `rhs` and `rhs` is left in a valid but otherwise unspecified state.

9
```
template<class ValueType>
  any& operator=(ValueType&& rhs);
```

10 Let `T` be equal to `decay_t<ValueType>`.

11 *Requires:* `T` shall satisfy the `CopyConstructible` requirements. If `is_copy_constructible_v<T>` is false, the program is ill-formed.

12 *Effects:* Constructs an object `tmp` of type `any` that contains an object of type `T` direct-initialized with `std::forward<ValueType>(rhs)`, and `tmp.swap(*this)`. No effects if an exception is thrown.

13 *Returns:* `*this`.

14 *Remarks:* This operator shall not participate in overload resolution if `decay_t<ValueType>` is the same type as `any`.

15 *Throws:* Any exception thrown by the selected constructor of `T`.

6.3.3 `any` modifiers [any.modifiers]

1 `void clear() noexcept;`

2 *Effects:* If not empty, destroys the contained object.

3 *Postconditions:* `empty() == true`.

4 `void swap(any& rhs) noexcept;`

5 *Effects:* Exchange the states of `*this` and `rhs`.

6.3.4 `any` observers [any.observers]

1 `bool empty() const noexcept;`

2 *Returns:* `true` if `*this` has no contained object, otherwise `false`.

3 `const type_info& type() const noexcept;`

4 *Returns:* If `*this` has a contained object of type T, `typeid(T)`; otherwise `typeid(void)`.

5 [*Note:* Useful for querying against types known either at compile time or only at runtime. — *end note*]

6.4 Non-member functions [any.nonmembers]

1 `void swap(any& x, any& y) noexcept;`

2 *Effects:* `x.swap(y)`.

3
```
template<class ValueType>
  ValueType any_cast(const any& operand);
template<class ValueType>
  ValueType any_cast(any& operand);
template<class ValueType>
  ValueType any_cast(any&& operand);
```

4 *Requires:* `is_reference_v<ValueType>` is true or `is_copy_constructible_v<ValueType>` is true. Otherwise the program is ill-formed.

5 *Returns:* For the first form, `*any_cast<add_const_t<remove_reference_t<ValueType>>>(&operand)`. For the second form, `*any_cast<remove_reference_t<ValueType>>(&operand)`. For the third form, if `is_move_constructible_v<ValueType>` is `true` and `is_lvalue_reference_v<ValueType>` is `false`, `std::move(*any_cast<remove_reference_t<ValueType>>(&operand))`, otherwise, `*any_cast<remove_reference_t<ValueType>>(&operand)`.

6 *Throws:* `bad_any_cast` if `operand.type() != typeid(remove_reference_t<ValueType>)`.

[*Example:*

```
any x(5);                                 // x holds int
assert(any_cast<int>(x) == 5);            // cast to value
any_cast<int&>(x) = 10;                   // cast to reference
assert(any_cast<int>(x) == 10);

x = "Meow";                               // x holds const char*
assert(strcmp(any_cast<const char*>(x), "Meow") == 0);
any_cast<const char*&>(x) = "Harry";
assert(strcmp(any_cast<const char*>(x), "Harry") == 0);

x = string("Meow");                       // x holds string
string s, s2("Jane");
s = move(any_cast<string&>(x));           // move from any
assert(s == "Meow");
any_cast<string&>(x) = move(s2);          // move to any
assert(any_cast<const string&>(x) == "Jane");

string cat("Meow");
const any y(cat);                         // const y holds string
assert(any_cast<const string&>(y) == cat);

any_cast<string&>(y);                     // error; cannot
                                          //  any_cast away const
```

— end example]

7
```
template<class ValueType>
    const ValueType* any_cast(const any* operand) noexcept;
template<class ValueType>
    ValueType* any_cast(any* operand) noexcept;
```

8 *Returns:* If `operand != nullptr && operand->type() == typeid(ValueType)`, a pointer to the object contained by `operand`, otherwise `nullptr`.

[*Example:*

```
bool is_string(const any& operand) {
  return any_cast<string>(&operand) != nullptr;
}
```

— *end example*]

7 `string_view` [string.view]

1 The class template `basic_string_view` describes an object that can refer to a constant contiguous sequence of char-like (C++14 §21.1) objects with the first element of the sequence at position zero. In the rest of this section, the type of the char-like objects held in a `basic_string_view` object is designated by `charT`.

2 [*Note:* The library provides implicit conversions from `const charT*` and `std::basic_string<charT, ...>` to `std::basic_string_view<charT, ...>` so that user code can accept just `std::basic_string_view<charT>` as a non-templated parameter wherever a sequence of characters is expected. User-defined types should define their own implicit conversions to `std::basic_string_view` in order to interoperate with these functions. — *end note*]

3 The complexity of `basic_string_view` member functions is O(1) unless otherwise specified.

7.1 Header `<experimental/string_view>` synopsis [string.view.synop]

```
namespace std {
  namespace experimental {
  inline namespace fundamentals_v2 {

    // 7.2, Class template basic_string_view
    template<class charT, class traits = char_traits<charT>>
        class basic_string_view;

    // 7.9, basic_string_view non-member comparison functions
    template<class charT, class traits>
    constexpr bool operator==(basic_string_view<charT, traits> x,
                              basic_string_view<charT, traits> y) noexcept;
    template<class charT, class traits>
    constexpr bool operator!=(basic_string_view<charT, traits> x,
                              basic_string_view<charT, traits> y) noexcept;
    template<class charT, class traits>
    constexpr bool operator< (basic_string_view<charT, traits> x,
                                 basic_string_view<charT, traits> y) noexcept;
    template<class charT, class traits>
    constexpr bool operator> (basic_string_view<charT, traits> x,
                              basic_string_view<charT, traits> y) noexcept;
    template<class charT, class traits>
    constexpr bool operator<=(basic_string_view<charT, traits> x,
                                 basic_string_view<charT, traits> y) noexcept;
    template<class charT, class traits>
    constexpr bool operator>=(basic_string_view<charT, traits> x,
                              basic_string_view<charT, traits> y) noexcept;
    // see below, sufficient additional overloads of comparison functions

    // 7.10, Inserters and extractors
    template<class charT, class traits>
      basic_ostream<charT, traits>&
        operator<<(basic_ostream<charT, traits>& os,
                   basic_string_view<charT, traits> str);

    // basic_string_view typedef names
```



```
    using string_view = basic_string_view<char>;
    using u16string_view = basic_string_view<char16_t>;
    using u32string_view = basic_string_view<char32_t>;
    using wstring_view = basic_string_view<wchar_t>;

  }  // namespace fundamentals_v2
  }  // namespace experimental

  // 7.11, Hash support
  template <class T> struct hash;
  template <> struct hash<experimental::string_view>;
  template <> struct hash<experimental::u16string_view>;
  template <> struct hash<experimental::u32string_view>;
  template <> struct hash<experimental::wstring_view>;

}  // namespace std
```

1 The function templates defined in C++14 §20.2.2 and C++14 §24.7 are available when `<experimental/string_view>` is included.

7.2 Class template `basic_string_view` [string.view.template]

```
template<class charT, class traits = char_traits<charT>>
class basic_string_view {
  public:
  // types
  using traits_type = traits;
  using value_type = charT;
  using pointer = charT*;
  using const_pointer = const charT*;
  using reference = charT&;
  using const_reference = const charT&;
  using const_iterator = implementation-defined; // See 7.4
  using iterator = const_iterator;[1]
  using const_reverse_iterator =
      reverse_iterator<const_iterator>;
  using reverse_iterator = const_reverse_iterator;
  using size_type = size_t;
  using difference_type = ptrdiff_t;
  static constexpr size_type npos = size_type(-1);

  // 7.3, basic_string_view constructors and assignment operators
  constexpr basic_string_view() noexcept;
  constexpr basic_string_view(const basic_string_view&) noexcept = default;
  basic_string_view& operator=(const basic_string_view&) noexcept = default;
  template<class Allocator>
  basic_string_view(const basic_string<charT, traits, Allocator>& str) noexcept;
  constexpr basic_string_view(const charT* str);
  constexpr basic_string_view(const charT* str, size_type len);

  // 7.4, basic_string_view iterator support
```

1. Because `basic_string_view` refers to a constant sequence, `iterator` and `const_iterator` are the same type.


```
constexpr const_iterator begin() const noexcept;
constexpr const_iterator end() const noexcept;
constexpr const_iterator cbegin() const noexcept;
constexpr const_iterator cend() const noexcept;
const_reverse_iterator rbegin() const noexcept;
const_reverse_iterator rend() const noexcept;
const_reverse_iterator crbegin() const noexcept;
const_reverse_iterator crend() const noexcept;

// 7.5, basic_string_view capacity
constexpr size_type size() const noexcept;
constexpr size_type length() const noexcept;
constexpr size_type max_size() const noexcept;
constexpr bool empty() const noexcept;

// 7.6, basic_string_view element access
constexpr const_reference operator[](size_type pos) const;
constexpr const_reference at(size_type pos) const;
constexpr const_reference front() const;
constexpr const_reference back() const;
constexpr const_pointer data() const noexcept;

// 7.7, basic_string_view modifiers
constexpr void remove_prefix(size_type n);
constexpr void remove_suffix(size_type n);
constexpr void swap(basic_string_view& s) noexcept;

// 7.8, basic_string_view string operations
template<class Allocator>
explicit operator basic_string<charT, traits, Allocator>() const;
template<class Allocator = allocator<charT> >
basic_string<charT, traits, Allocator> to_string(
  const Allocator& a = Allocator()) const;

size_type copy(charT* s, size_type n, size_type pos = 0) const;

constexpr basic_string_view substr(size_type pos = 0, size_type n = npos) const;
constexpr int compare(basic_string_view s) const noexcept;
constexpr int compare(size_type pos1, size_type n1, basic_string_view s) const;
constexpr int compare(size_type pos1, size_type n1,
                      basic_string_view s, size_type pos2, size_type n2) const;
constexpr int compare(const charT* s) const;
constexpr int compare(size_type pos1, size_type n1, const charT* s) const;
constexpr int compare(size_type pos1, size_type n1,
                      const charT* s, size_type n2) const;
constexpr size_type find(basic_string_view s, size_type pos = 0) const noexcept;
constexpr size_type find(charT c, size_type pos = 0) const noexcept;
constexpr size_type find(const charT* s, size_type pos, size_type n) const;
constexpr size_type find(const charT* s, size_type pos = 0) const;
constexpr size_type rfind(basic_string_view s, size_type pos = npos) const noexcept;
constexpr size_type rfind(charT c, size_type pos = npos) const noexcept;
constexpr size_type rfind(const charT* s, size_type pos, size_type n) const;
```



```
  constexpr size_type rfind(const charT* s, size_type pos = npos) const;
  constexpr size_type find_first_of(basic_string_view s, size_type pos = 0) const noexcept;
  constexpr size_type find_first_of(charT c, size_type pos = 0) const noexcept;
  constexpr size_type find_first_of(const charT* s, size_type pos, size_type n) const;
  constexpr size_type find_first_of(const charT* s, size_type pos = 0) const;
  constexpr size_type find_last_of(basic_string_view s, size_type pos = npos) const noexcept;
  constexpr size_type find_last_of(charT c, size_type pos = npos) const noexcept;
  constexpr size_type find_last_of(const charT* s, size_type pos, size_type n) const;
  constexpr size_type find_last_of(const charT* s, size_type pos = npos) const;
  constexpr size_type find_first_not_of(basic_string_view s, size_type pos = 0) const noexcept;
  constexpr size_type find_first_not_of(charT c, size_type pos = 0) const noexcept;
  constexpr size_type find_first_not_of(const charT* s, size_type pos, size_type n) const;
  constexpr size_type find_first_not_of(const charT* s, size_type pos = 0) const;
  constexpr size_type find_last_not_of(basic_string_view s, size_type pos = npos) const noexcept;
  constexpr size_type find_last_not_of(charT c, size_type pos = npos) const noexcept;
  constexpr size_type find_last_not_of(const charT* s, size_type pos, size_type n) const;
  constexpr size_type find_last_not_of(const charT* s, size_type pos = npos) const;

 private:
  const_pointer data_;  // exposition only
  size_type     size_;  // exposition only
};
```

1 In every specialization `basic_string_view<charT, traits>`, the type `traits` shall satisfy the character traits requirements (C++14 §21.2), and the type `traits::char_type` shall name the same type as `charT`.

7.3 `basic_string_view` constructors and assignment operators [string.view.cons]

1 `constexpr basic_string_view() noexcept;`

2 *Effects:* Constructs an empty `basic_string_view`.

3 *Postconditions:* `size_ == 0` and `data_ == nullptr`.

4 `template<class Allocator>`
`basic_string_view(const basic_string<charT, traits, Allocator>& str) noexcept;`

5 *Effects:* Constructs a `basic_string_view`, with the postconditions in Table 10.

Table 10 — `basic_string_view(const basic_string&)` effects

Element	Value
`data_`	`str.data()`
`size_`	`str.size()`

6 `constexpr basic_string_view(const charT* str);`

7 *Requires:* [`str`, `str + traits::length(str)`) is a valid range.

8 *Effects:* Constructs a `basic_string_view`, with the postconditions in Table 11.

Table 11 — `basic_string_view(const charT*)` effects

Element	Value
`data_`	`str`
`size_`	`traits::length(str)`

9 *Complexity:* O(`traits::length(str)`)

10 `constexpr basic_string_view(const charT* str, size_type len);`

11 *Requires:* [`str`, `str + len`) is a valid range.

12 *Effects:* Constructs a `basic_string_view`, with the postconditions in Table 12.

Table 12 — `basic_string_view(const charT*, size_type)` effects

Element	Value
`data_`	`str`
`size_`	`len`

7.4 `basic_string_view` iterator support [string.view.iterators]

1 `using const_iterator = implementation-defined;`

2 A constant random-access iterator type such that, for a `const_iterator it`, if `&*(it+N)` is valid, then it is equal to `(&*it)+N`.

3 For a `basic_string_view str`, any operation that invalidates a pointer in the range [`str.data()`, `str.data()+str.size()`) invalidates pointers, iterators, and references returned from `str`'s methods.

4 All requirements on container iterators (C++14 §23.2) apply to `basic_string_view::const_iterator` as well.

5
```
constexpr const_iterator begin() const noexcept;
constexpr const_iterator cbegin() const noexcept;
```

6 *Returns:* An iterator such that `&*begin() == data_` if `!empty()`, or else an unspecified value such that [`begin()`, `end()`) is a valid range.

7
```
constexpr const_iterator end() const noexcept;
constexpr const_iterator cend() const noexcept;
```

8 *Returns:* `begin() + size()`.

9
```
const_reverse_iterator rbegin() const noexcept;
const_reverse_iterator crbegin() const noexcept;
```

10 *Returns:* `const_reverse_iterator(end())`.

11
```
const_reverse_iterator rend() const noexcept;
const_reverse_iterator crend() const noexcept;
```

12 *Returns:* `const_reverse_iterator(begin())`.

7.5 `basic_string_view` capacity [string.view.capacity]

1 `constexpr size_type size() const noexcept;`

2 *Returns:* `size_`.

3 `constexpr size_type length() const noexcept;`

4 *Returns:* `size_`.

5 `constexpr size_type max_size() const noexcept;`

6 *Returns:* The largest possible number of char-like objects that can be referred to by a `basic_string_view`.

7 `constexpr bool empty() const noexcept;`

8 *Returns:* `size_ == 0`.

7.6 `basic_string_view` element access [string.view.access]

1 `constexpr const_reference operator[](size_type pos) const;`

2 *Requires:* `pos < size()`.

3 *Returns:* `data_[pos]`.

4 *Throws:* Nothing.

5 [*Note:* Unlike `basic_string::operator[]`, `basic_string_view::operator[](size())` has undefined behavior instead of returning `charT()`. — *end note*]

6 `constexpr const_reference at(size_type pos) const;`

7 *Throws:* `out_of_range` if `pos >= size()`.

8 *Returns:* `data_[pos]`.

9 `constexpr const_reference front() const;`

10 *Requires:* `!empty()`

11 *Returns:* `data_[0]`.

12 *Throws:* Nothing.

13 `constexpr const_reference back() const;`

14 *Requires:* `!empty()`

15 *Returns:* `data_[size() - 1]`.

16 *Throws:* Nothing.

17
```
constexpr const_pointer data() const noexcept;
```

18 *Returns:* `data_`.

19 [*Note:* Unlike `basic_string::data()` and string literals, `data()` may return a pointer to a buffer that is not null-terminated. Therefore it is typically a mistake to pass `data()` to a routine that takes just a `const charT*` and expects a null-terminated string. — *end note*]

7.7 `basic_string_view` modifiers [string.view.modifiers]

1
```
constexpr void remove_prefix(size_type n);
```

2 *Requires:* `n <= size()`.

3 *Effects:* Equivalent to `data_ += n; size_ -= n;`

4
```
constexpr void remove_suffix(size_type n);
```

5 *Requires:* `n <= size()`.

6 *Effects:* Equivalent to `size_ -= n;`

7
```
constexpr void swap(basic_string_view& s) noexcept;
```

8 *Effects:* Exchanges the values of `*this` and `s`.

7.8 `basic_string_view` string operations [string.view.ops]

1
```
template<class Allocator>
  explicit[2] operator basic_string<
      charT, traits, Allocator>() const;
```

2 *Effects:* Equivalent to `return basic_string<charT, traits, Allocator>(begin(), end());`

3 *Complexity:* O(`size()`)

4 [*Note:* Users who want to control the allocator instance should call `to_string(allocator)`. — *end note*]

5
```
template<class Allocator = allocator<charT>>
  basic_string<charT, traits, Allocator> to_string(
      const Allocator& a = Allocator()) const;
```

6 *Returns:* `basic_string<charT, traits, Allocator>(begin(), end(), a)`.

7 *Complexity:* O(`size()`)

2. This conversion is explicit to avoid accidental O(N) operations on type mismatches.

8 `size_type copy(charT* s, size_type n, size_type pos = 0) const;`

9 Let `rlen` be the smaller of `n` and `size() - pos`.

10 *Throws:* `out_of_range` if `pos > size()`.

11 *Requires:* [`s`, `s + rlen`) is a valid range.

12 *Effects:* Equivalent to `std::copy_n(begin() + pos, rlen, s)`.

13 *Returns:* `rlen`.

14 *Complexity:* O(`rlen`)

15 `constexpr basic_string_view substr(size_type pos = 0, size_type n = npos) const;`

16 *Throws:* `out_of_range` if `pos > size()`.

17 *Effects:* Determines the effective length *`rlen`* of the string to reference as the smaller of `n` and `size() - pos`.

18 *Returns:* `basic_string_view(data()+pos, rlen)`.

19 `constexpr int compare(basic_string_view str) const noexcept;`

20 *Effects:* Determines the effective length *`rlen`* of the strings to compare as the smaller of `size()` and `str.size()`. The function then compares the two strings by calling `traits::compare(data(), str.data(), rlen)`.

21 *Complexity:* O(*`rlen`*)

22 *Returns:* The nonzero result if the result of the comparison is nonzero. Otherwise, returns a value as indicated in Table 13.

Table 13 — `compare()` results

Condition	Return Value
`size() < str.size()`	`< 0`
`size() == str.size()`	`0`
`size() > str.size()`	`> 0`

23 `constexpr int compare(size_type pos1, size_type n1, basic_string_view str) const;`

24 *Effects:* Equivalent to `return substr(pos1, n1).compare(str);`

25
```
constexpr int compare(size_type pos1, size_type n1, basic_string_view str,
                      size_type pos2, size_type n2) const;
```

26 *Effects:* Equivalent to `return substr(pos1, n1).compare(str.substr(pos2, n2));`

27 `constexpr int compare(const charT* s) const;`

28 *Effects:* Equivalent to `return compare(basic_string_view(s));`

29 `constexpr int compare(size_type pos1, size_type n1, const charT* s) const;`

30 *Effects:* Equivalent to `return substr(pos1, n1).compare(basic_string_view(s));`

31
```
constexpr int compare(size_type pos1, size_type n1,
                      const charT* s, size_type n2) const;
```

32 *Effects:* Equivalent to `return substr(pos1, n1).compare(basic_string_view(s, n2));`

7.8.1 Searching `basic_string_view` [string.view.find]

1 This section specifies the `basic_string_view` member functions named `find`, `rfind`, `find_first_of`, `find_last_of`, `find_first_not_of`, and `find_last_not_of`.

2 Member functions in this section have complexity O(`size() * str.size()`) at worst, although implementations are encouraged to do better.

3 Each member function of the form

```
constexpr return-type fx1(const charT* s, size_type pos);
```

is equivalent to `return fx1(basic_string_view(s), pos);`

4 Each member function of the form

```
constexpr return-type fx1(const charT* s, size_type pos, size_type n);
```

is equivalent to `return fx1(basic_string_view(s, n), pos);`

5 Each member function of the form

```
constexpr return-type fx2(charT c, size_type pos);
```

is equivalent to `return fx2(basic_string_view(&c, 1), pos);`

6 `constexpr size_type find(basic_string_view str, size_type pos = 0) const noexcept;`

7 *Effects:* Determines the lowest position `xpos`, if possible, such that the following conditions obtain:

- `pos <= xpos`
- `xpos + str.size() <= size()`
- `traits::eq(at(xpos+I), str.at(I))` for all elements `I` of the string referenced by `str`.

8 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

9 `constexpr size_type rfind(basic_string_view str, size_type pos = npos) const noexcept;`

10 *Effects:* Determines the highest position `xpos`, if possible, such that the following conditions obtain:

- `xpos <= pos`
- `xpos + str.size() <= size()`
- `traits::eq(at(xpos+I), str.at(I))` for all elements `I` of the string referenced by `str`.

11 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

12 `constexpr size_type find_first_of(basic_string_view str, size_type pos = 0) const noexcept;`

13 *Effects:* Determines the lowest position `xpos`, if possible, such that the following conditions obtain:

- `pos <= xpos`
- `xpos < size()`
- `traits::eq(at(xpos), str.at(I))` for some element `I` of the string referenced by `str`.

14 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

15 `constexpr size_type find_last_of(basic_string_view str, size_type pos = npos) const noexcept;`

16 *Effects:* Determines the highest position `xpos`, if possible, such that the following conditions obtain:

- `xpos <= pos`
- `xpos < size()`
- `traits::eq(at(xpos), str.at(I))` for some element `I` of the string referenced by `str`.

17 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

18 `constexpr size_type find_first_not_of(basic_string_view str, size_type pos = 0) const noexcept;`

19 *Effects:* Determines the lowest position `xpos`, if possible, such that the following conditions obtain:

- `pos <= xpos`
- `xpos < size()`
- `traits::eq(at(xpos), str.at(I))` for no element `I` of the string referenced by `str`.

20 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

21 `constexpr size_type find_last_not_of(basic_string_view str, size_type pos = npos) const noexcept;`

22 *Effects:* Determines the highest position `xpos`, if possible, such that the following conditions obtain:

- `xpos <= pos`
- `xpos < size()`
- `traits::eq(at(xpos), str.at(I))` for no element `I` of the string referenced by `str`.

23 *Returns:* `xpos` if the function can determine such a value for `xpos`. Otherwise, returns `npos`.

7.9 `basic_string_view` non-member comparison functions [string.view.comparison]

1 Let `S` be `basic_string_view<charT, traits>`, and `sv` be an instance of `S`. Implementations shall provide sufficient additional overloads marked `constexpr` and `noexcept` so that an object `t` with an implicit conversion to `S` can be compared according to Table 14.

Table 14 — Additional `basic_string_view` comparison overloads

Expression	Equivalent to
`t == sv`	`S(t) == sv`
`sv == t`	`sv == S(t)`
`t != sv`	`S(t) != sv`
`sv != t`	`sv != S(t)`
`t < sv`	`S(t) < sv`
`sv < t`	`sv < S(t)`
`t > sv`	`S(t) > sv`
`sv > t`	`sv > S(t)`
`t <= sv`	`S(t) <= sv`
`sv <= t`	`sv <= S(t)`
`t >= sv`	`S(t) >= sv`
`sv >= t`	`sv >= S(t)`

[*Example:* A sample conforming implementation for operator== would be:

```
template<class T> using __identity = decay_t<T>;
template<class charT, class traits>
constexpr bool operator==(
    basic_string_view<charT, traits> lhs,
    basic_string_view<charT, traits> rhs) noexcept {
  return lhs.compare(rhs) == 0;
}
template<class charT, class traits>
constexpr bool operator==(
    basic_string_view<charT, traits> lhs,
    __identity<basic_string_view<charT, traits>> rhs) noexcept {
```



```
    return lhs.compare(rhs) == 0;
  }
  template<class charT, class traits>
  constexpr bool operator==(
      __identity<basic_string_view<charT, traits>> lhs,
      basic_string_view<charT, traits> rhs) noexcept {
    return lhs.compare(rhs) == 0;
  }
```

— *end example*]

2
```
template<class charT, class traits>
     constexpr bool operator==(basic_string_view<charT, traits> lhs,
                               basic_string_view<charT, traits> rhs) noexcept;
```

3 *Returns:* `lhs.compare(rhs) == 0`.

4
```
template<class charT, class traits>
     constexpr bool operator!=(basic_string_view<charT, traits> lhs,
                               basic_string_view<charT, traits> rhs) noexcept;
```

5 *Returns:* `lhs.compare(rhs) != 0`.

6
```
template<class charT, class traits>
     constexpr bool operator< (basic_string_view<charT, traits> lhs,
                               basic_string_view<charT, traits> rhs) noexcept;
```

7 *Returns:* `lhs.compare(rhs) < 0`.

8
```
template<class charT, class traits>
     constexpr bool operator> (basic_string_view<charT, traits> lhs,
                               basic_string_view<charT, traits> rhs) noexcept;
```

9 *Returns:* `lhs.compare(rhs) > 0`.

10
```
template<class charT, class traits>
     constexpr bool operator<=(basic_string_view<charT, traits> lhs,
                               basic_string_view<charT, traits> rhs) noexcept;
```

11 *Returns:* `lhs.compare(rhs) <= 0`.

12
```
template<class charT, class traits>
     constexpr bool operator>=(basic_string_view<charT, traits> lhs,
                               basic_string_view<charT, traits> rhs) noexcept;
```

13 *Returns:* `lhs.compare(rhs) >= 0`.

7.10 Inserters and extractors [string.view.io]

1
```
template<class charT, class traits>
     basic_ostream<charT, traits>&
       operator<<(basic_ostream<charT, traits>& os,
                  basic_string_view<charT, traits> str);
```

2 *Effects:* Equivalent to `return os << str.to_string();`

7.11 Hash support [string.view.hash]

1
```
template <> struct hash<experimental::string_view>;
template <> struct hash<experimental::u16string_view>;
template <> struct hash<experimental::u32string_view>;
template <> struct hash<experimental::wstring_view>;
```

2 The template specializations shall meet the requirements of class template hash (C++14 §20.9.12).

8 Memory [memory]

8.1 Header <experimental/memory> synopsis [header.memory.synop]

```
#include <memory>

namespace std {
  namespace experimental {
  inline namespace fundamentals_v2 {

    // See C++14 §20.7.7, uses_allocator
    template <class T, class Alloc> constexpr bool uses_allocator_v
      = uses_allocator<T, Alloc>::value;

    // 8.2.1, Class template shared_ptr
    template<class T> class shared_ptr;

    // C++14 §20.8.2.2.6
    template<class T, class... Args> shared_ptr<T> make_shared(Args&&... args);
    template<class T, class A, class... Args>
      shared_ptr<T> allocate_shared(const A& a, Args&&... args);

    // C++14 §20.8.2.2.7
    template<class T, class U>
      bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
    template<class T, class U>
      bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
    template<class T, class U>
      bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
    template<class T, class U>
      bool operator>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
    template<class T, class U>
      bool operator<=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
    template<class T, class U>
      bool operator>=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
    template <class T>
      bool operator==(const shared_ptr<T>& a, nullptr_t) noexcept;
    template <class T>
      bool operator==(nullptr_t, const shared_ptr<T>& b) noexcept;
    template <class T>
      bool operator!=(const shared_ptr<T>& a, nullptr_t) noexcept;
    template <class T>
      bool operator!=(nullptr_t, const shared_ptr<T>& b) noexcept;
    template <class T>
      bool operator<(const shared_ptr<T>& a, nullptr_t) noexcept;
    template <class T>
      bool operator<(nullptr_t, const shared_ptr<T>& b) noexcept;
    template <class T>
      bool operator<=(const shared_ptr<T>& a, nullptr_t) noexcept;
    template <class T>
```



```
  bool operator<=(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator>(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator>(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator>=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator>=(nullptr_t, const shared_ptr<T>& b) noexcept;

// C++14 §20.8.2.2.8
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

// 8.2.1.3, shared_ptr casts
template<class T, class U>
  shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> reinterpret_pointer_cast(const shared_ptr<U>& r) noexcept;

// C++14 §20.8.2.2.10
template<class D, class T> D* get_deleter(const shared_ptr<T>& p) noexcept;

// C++14 §20.8.2.2.11
template<class E, class T, class Y>
  basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, const shared_ptr<Y>& p);

// 8.2.2, Class template weak_ptr
template<class T> class weak_ptr;

// C++14 §20.8.2.3.6
template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;

// C++14 §20.8.2.4
template<class T> class owner_less;

// C++14 §20.8.2.5
template<class T> class enable_shared_from_this;

// C++14 §20.8.2.6
template<class T>
  bool atomic_is_lock_free(const shared_ptr<T>* p);
template<class T>
  shared_ptr<T> atomic_load(const shared_ptr<T>* p);
template<class T>
  shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);
template<class T>
  void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);
template<class T>
```

```
  void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);
template<class T>
  shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
template<class T>
  shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r,
                                         memory_order mo);
template<class T>
  bool atomic_compare_exchange_weak(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
template<class T>
  bool atomic_compare_exchange_strong(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
template<class T>
  bool atomic_compare_exchange_weak_explicit(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
    memory_order success, memory_order failure);
template<class T>
  bool atomic_compare_exchange_strong_explicit(
    shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
    memory_order success, memory_order failure);

// 8.12, Non-owning pointers
template <class W> class observer_ptr;

// 8.12.6, observer_ptr specialized algorithms
template <class W>
void swap(observer_ptr<W>&, observer_ptr<W>&) noexcept;
template <class W>
observer_ptr<W> make_observer(W*) noexcept;
// (in)equality operators
template <class W1, class W2>
bool operator==(observer_ptr<W1>, observer_ptr<W2>);

template <class W1, class W2>
bool operator!=(observer_ptr<W1>, observer_ptr<W2>);
template <class W>
bool operator==(observer_ptr<W>, nullptr_t) noexcept;
template <class W>
bool operator!=(observer_ptr<W>, nullptr_t) noexcept;
template <class W>
bool operator==(nullptr_t, observer_ptr<W>) noexcept;
template <class W>
bool operator!=(nullptr_t, observer_ptr<W>) noexcept;
// ordering operators
template <class W1, class W2>
bool operator<(observer_ptr<W1>, observer_ptr<W2>);
template <class W1, class W2>
bool operator>(observer_ptr<W1>, observer_ptr<W2>);
template <class W1, class W2>
bool operator<=(observer_ptr<W1>, observer_ptr<W2>);
template <class W1, class W2>
bool operator>=(observer_ptr<W1>, observer_ptr<W2>);
```



```
  } // inline namespace fundamentals_v2
  } // namespace experimental

  // 8.2.1.4, shared_ptr hash support
  template<class T> struct hash<experimental::shared_ptr<T>>;

  // 8.12.7, observer_ptr hash support
  template <class T> struct hash;
  template <class T> struct hash<experimental::observer_ptr<T>>;

} // namespace std
```

8.2 Shared-ownership pointers [memory.smartptr]

1 The specification of all declarations within this sub-clause 8.2 and its sub-clauses are the same as the corresponding declarations, as specified in C++14 §20.8.2, unless explicitly specified otherwise.

8.2.1 Class template shared_ptr [memory.smartptr.shared]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  template<class T> class shared_ptr {
  public:
    using element_type = remove_extent_t<T>;
    // 8.2.1.1, shared_ptr constructors
    constexpr shared_ptr() noexcept;
    template<class Y> explicit shared_ptr(Y* p);
    template<class Y, class D> shared_ptr(Y* p, D d);
    template<class Y, class D, class A> shared_ptr(Y* p, D d, A a);
    template <class D> shared_ptr(nullptr_t p, D d)
    template <class D, class A> shared_ptr(nullptr_t p, D d, A a);
    template<class Y> shared_ptr(const shared_ptr<Y>& r, element_type* p) noexcept;
    shared_ptr(const shared_ptr& r) noexcept;
    template<class Y> shared_ptr(const shared_ptr<Y>& r) noexcept;
    shared_ptr(shared_ptr&& r) noexcept;
    template<class Y> shared_ptr(shared_ptr<Y>&& r) noexcept;
    template<class Y> explicit shared_ptr(const weak_ptr<Y>& r);
    template<class Y> shared_ptr(auto_ptr<Y>&& r);
    template <class Y, class D> shared_ptr(unique_ptr<Y, D>&& r);
    constexpr shared_ptr(nullptr_t) : shared_ptr() { }

    // C++14 §20.8.2.2.2
    ~shared_ptr();

    // C++14 §20.8.2.2.3
    shared_ptr& operator=(const shared_ptr& r) noexcept;
    template<class Y> shared_ptr& operator=(const shared_ptr<Y>& r) noexcept;
    shared_ptr& operator=(shared_ptr&& r) noexcept;
```



```
  template<class Y> shared_ptr& operator=(shared_ptr<Y>&& r) noexcept;
  template<class Y> shared_ptr& operator=(auto_ptr<Y>&& r);
  template <class Y, class D> shared_ptr& operator=(unique_ptr<Y, D>&& r);

  // C++14 §20.8.2.2.4
  void swap(shared_ptr& r) noexcept;
  void reset() noexcept;
  template<class Y> void reset(Y* p);
  template<class Y, class D> void reset(Y* p, D d);
  template<class Y, class D, class A> void reset(Y* p, D d, A a);

  // 8.2.1.2, shared_ptr observers
  element_type* get() const noexcept;
  T& operator*() const noexcept;
  T* operator->() const noexcept;
  element_type& operator[](ptrdiff_t i) const noexcept;
  long use_count() const noexcept;
  bool unique() const noexcept;
  explicit operator bool() const noexcept;
  template<class U> bool owner_before(shared_ptr<U> const& b) const;
  template<class U> bool owner_before(weak_ptr<U> const& b) const;
};

// C++14 §20.8.2.2.6
template<class T, class... Args> shared_ptr<T> make_shared(Args&&... args);
template<class T, class A, class... Args>
  shared_ptr<T> allocate_shared(const A& a, Args&&... args);

// C++14 §20.8.2.2.7
template<class T, class U>
  bool operator==(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator!=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator<(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator>(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator<=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template<class T, class U>
  bool operator>=(const shared_ptr<T>& a, const shared_ptr<U>& b) noexcept;
template <class T>
  bool operator==(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator==(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator!=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator!=(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator<(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
```



```
  bool operator<(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator<=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator<=(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator>(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator>(nullptr_t, const shared_ptr<T>& b) noexcept;
template <class T>
  bool operator>=(const shared_ptr<T>& a, nullptr_t) noexcept;
template <class T>
  bool operator>=(nullptr_t, const shared_ptr<T>& b) noexcept;

// C++14 §20.8.2.2.8
template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b) noexcept;

// 8.2.1.3, shared_ptr casts
template<class T, class U>
  shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;
template<class T, class U>
  shared_ptr<T> reinterpret_pointer_cast(const shared_ptr<U>& r) noexcept;

// C++14 §20.8.2.2.10
template<class D, class T> D* get_deleter(const shared_ptr<T>& p) noexcept;

// C++14 §20.8.2.2.11
template<class E, class T, class Y>
  basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, const shared_ptr<Y>& p);

// C++14 §20.8.2.4
template<class T> class owner_less;

// C++14 §20.8.2.5
template<class T> class enable_shared_from_this;

// C++14 §20.8.2.6
template<class T>
  bool atomic_is_lock_free(const shared_ptr<T>* p);
template<class T>
  shared_ptr<T> atomic_load(const shared_ptr<T>* p);
template<class T>
  shared_ptr<T> atomic_load_explicit(const shared_ptr<T>* p, memory_order mo);
template<class T>
  void atomic_store(shared_ptr<T>* p, shared_ptr<T> r);
template<class T>
  void atomic_store_explicit(shared_ptr<T>* p, shared_ptr<T> r, memory_order mo);
template<class T>
```



```
    shared_ptr<T> atomic_exchange(shared_ptr<T>* p, shared_ptr<T> r);
  template<class T>
    shared_ptr<T> atomic_exchange_explicit(shared_ptr<T>* p, shared_ptr<T> r,
                                           memory_order mo);
  template<class T>
    bool atomic_compare_exchange_weak(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
  template<class T>
    bool atomic_compare_exchange_strong(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w);
  template<class T>
    bool atomic_compare_exchange_weak_explicit(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
      memory_order success, memory_order failure);
  template<class T>
    bool atomic_compare_exchange_strong_explicit(
      shared_ptr<T>* p, shared_ptr<T>* v, shared_ptr<T> w,
      memory_order success, memory_order failure);


} // namespace fundamentals_v2
} // namespace experimental

  // 8.2.1.4, shared_ptr hash support
  template<class T> struct hash<experimental::shared_ptr<T>>;

} // namespace std
```

1 For the purposes of subclause 8.2, a pointer type `Y*` is said to be *compatible with* a pointer type `T*` when either `Y*` is convertible to `T*` or `Y` is `U[N]` and `T` is `U` *cv* `[]`.

8.2.1.1 `shared_ptr` constructors [memory.smartptr.shared.const]

1 `template<class Y> explicit shared_ptr(Y* p);`

2 *Requires:* `Y` shall be a complete type. The expression `delete[] p`, when `T` is an array type, or `delete p`, when `T` is not an array type, shall be well-formed, shall have well defined behavior, and shall not throw exceptions. When `T` is `U[N]`, `Y(*)[N]` shall be convertible to `T*`; when `T` is `U[]`, `Y(*)[]` shall be convertible to `T*`; otherwise, `Y*` shall be convertible to `T*`.

3 *Effects:* When `T` is not an array type, constructs a `shared_ptr` object that *owns* the pointer `p`. Otherwise, constructs a `shared_ptr` that *owns* `p` and a deleter of an unspecified type that calls `delete[] p`. If an exception is thrown, `delete p` is called when `T` is not an array type, `delete[] p` otherwise.

4 *Postconditions:* `use_count() == 1 && get() == p`.

5 *Throws:* `bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

6
```
template<class Y, class D> shared_ptr(Y* p, D d);
template<class Y, class D, class A> shared_ptr(Y* p, D d, A a);
template <class D> shared_ptr(nullptr_t p, D d);
template <class D, class A> shared_ptr(nullptr_t p, D d, A a);
```

7 *Requires:* `D` shall be `CopyConstructible`. The copy constructor and destructor of `D` shall not throw exceptions. The expression `d(p)` shall be well formed, shall have well defined behavior, and shall not throw exceptions. `A` shall be an allocator (C++14 §17.6.3.5). The copy constructor and destructor of `A` shall not throw exceptions. When `T` is `U[N]`, `Y(*)[N]` shall be convertible to `T*`; when `T` is `U[]`, `Y(*)[]` shall be convertible to `T*`; otherwise, `Y*` shall be convertible to `T*`.

8 *Effects:* Constructs a `shared_ptr` object that *owns* the object `p` and the deleter `d`. The second and fourth constructors shall use a copy of `a` to allocate memory for internal use. If an exception is thrown, `d(p)` is called.

9 *Postconditions:* `use_count() == 1 && get() == p`.

10 *Throws:* `bad_alloc`, or an implementation-defined exception when a resource other than memory could not be obtained.

11
```
template<class Y> shared_ptr(const shared_ptr<Y>& r, element_type* p) noexcept;
```

12 *Effects:* Constructs a `shared_ptr` instance that stores `p` and *shares ownership* with `r`.

13 *Postconditions:* `get() == p && use_count() == r.use_count()`.

14 [*Note:* To avoid the possibility of a dangling pointer, the user of this constructor must ensure that `p` remains valid at least until the ownership group of `r` is destroyed. — *end note*]

15 [*Note:* This constructor allows creation of an *empty* `shared_ptr` instance with a non-null stored pointer. — *end note*]

16
```
shared_ptr(const shared_ptr& r) noexcept;
template<class Y> shared_ptr(const shared_ptr<Y>& r) noexcept;
```

17 *Requires:* The second constructor shall not participate in the overload resolution unless `Y*` is *compatible with* `T*`.

18 *Effects:* If `r` is *empty*, constructs an *empty* `shared_ptr` object; otherwise, constructs a `shared_ptr` object that *shares ownership* with `r`.

19 *Postconditions:* `get() == r.get() && use_count() == r.use_count()`.

20
```
shared_ptr(shared_ptr&& r) noexcept;
template<class Y> shared_ptr(shared_ptr<Y>&& r) noexcept;
```

21 *Remarks:* The second constructor shall not participate in overload resolution unless `Y*` is *compatible with* `T*`.

22 *Effects:* Move-constructs a `shared_ptr` instance from `r`.

23 *Postconditions:* `*this` shall contain the old value of `r`. `r` shall be *empty*. `r.get() == 0`.

24 `template<class Y> explicit shared_ptr(const weak_ptr<Y>& r);`

25 *Requires:* `Y*` shall be *compatible with* `T*`.

26 *Effects:* Constructs a `shared_ptr` object that *shares ownership* with `r` and stores a copy of the pointer stored in `r`. If an exception is thrown, the constructor has no effect.

27 *Postconditions:* `use_count() == r.use_count()`.

28 *Throws:* `bad_weak_ptr` when `r.expired()`.

29 `template <class Y, class D> shared_ptr(unique_ptr<Y, D>&& r);`

30 *Remarks:* This constructor shall not participate in overload resolution unless `Y*` is *compatible with* `T*`.

31 *Effects:* Equivalent to `shared_ptr(r.release(), r.get_deleter())` when `D` is not a reference type, otherwise `shared_ptr(r.release(), ref(r.get_deleter()))`. If an exception is thrown, the constructor has no effect.

8.2.1.2 `shared_ptr` observers [memory.smartptr.shared.obs]

1 `element_type* get() const noexcept;`

2 *Returns:* The stored pointer.

3 `T& operator*() const noexcept;`

4 *Requires:* `get() != 0`.

5 *Returns:* `*get()`.

6 *Remarks:* When `T` is an array type or (possibly cv-qualified) `void`, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well formed.

7 `T* operator->() const noexcept;`

8 *Requires:* `get() != 0`.

9 *Returns:* `get()`.

10 *Remarks:* When `T` is an array type, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well formed.

11 `element_type& operator[](ptrdiff_t i) const noexcept;`

12 *Requires:* `get() != 0 && i >= 0`. If `T` is `U[N]`, `i < N`.

13 *Returns:* `get()[i]`.

14 *Remarks:* When `T` is not an array type, it is unspecified whether this member function is declared. If it is declared, it is unspecified what its return type is, except that the declaration (although not necessarily the definition) of the function shall be well formed.

8.2.1.3 `shared_ptr` casts [memory.smartptr.shared.cast]

1 `template<class T, class U> shared_ptr<T> static_pointer_cast(const shared_ptr<U>& r) noexcept;`

2 *Requires:* The expression `static_cast<T*>((U*)0)` shall be well formed.

3 *Returns:* `shared_ptr<T>(r, static_cast<typename shared_ptr<T>::element_type*>(r.get()))`.

4 [*Note:* The similar expression `shared_ptr<T>(static_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice. — *end note*]

5 `template<class T, class U> shared_ptr<T> dynamic_pointer_cast(const shared_ptr<U>& r) noexcept;`

6 *Requires:* The expression `dynamic_cast<T*>((U*)0)` shall be well formed.

7 *Returns:*

- When `dynamic_cast<typename shared_ptr<T>::element_type*>(r.get())` returns a nonzero value `p`, `shared_ptr<T>(r, p)`;
- Otherwise, `shared_ptr<T>()`.

8 [*Note:* The similar expression `shared_ptr<T>(dynamic_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice. — *end note*]

9 `template<class T, class U> shared_ptr<T> const_pointer_cast(const shared_ptr<U>& r) noexcept;`

10 *Requires:* The expression `const_cast<T*>((U*)0)` shall be well formed.

11 *Returns:* `shared_ptr<T>(r, const_cast<typename shared_ptr<T>::element_type*>(r.get()))`.

12 [*Note:* The similar expression `shared_ptr<T>(const_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice. — *end note*]

13 `template<class T, class U> shared_ptr<T> reinterpret_pointer_cast(const shared_ptr<U>& r) noexcept;`

14 *Requires:* The expression `reinterpret_cast<T*>((U*)0)` shall be well formed.

15 *Returns:* `shared_ptr<T>(r, reinterpret_cast<typename shared_ptr<T>::element_type*>(r.get()))`.

8.2.1.4 `shared_ptr` hash support [memory.smartptr.shared.hash]

1 `template <class T> struct hash<experimental::shared_ptr<T>>;`

2 The template specialization shall meet the requirements of class template `hash` (C++14 §20.9.12). For an object `p` of type `experimental::shared_ptr<T>`, `hash<experimental::shared_ptr<T>>()(p)` shall evaluate to the same value as `hash<typename experimental::shared_ptr<T>::element_type*>()(p.get())`.

8.2.2 Class template weak_ptr [memory.smartptr.weak]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  template<class T> class weak_ptr {
  public:
    using element_type = remove_extent_t<T>;
```

```
    // 8.2.2.1, weak_ptr constructors
    constexpr weak_ptr() noexcept;
    template<class Y> weak_ptr(shared_ptr<Y> const& r) noexcept;
    weak_ptr(weak_ptr const& r) noexcept;
    template<class Y> weak_ptr(weak_ptr<Y> const& r) noexcept;
    weak_ptr(weak_ptr&& r) noexcept;
    template<class Y> weak_ptr(weak_ptr<Y>&& r) noexcept;

    // C++14 §20.8.2.3.2
    ~weak_ptr();

    // C++14 §20.8.2.3.3
    weak_ptr& operator=(weak_ptr const& r) noexcept;
    template<class Y> weak_ptr& operator=(weak_ptr<Y> const& r) noexcept;
    template<class Y> weak_ptr& operator=(shared_ptr<Y> const& r) noexcept;
    weak_ptr& operator=(weak_ptr&& r) noexcept;
    template<class Y> weak_ptr& operator=(weak_ptr<Y>&& r) noexcept;

    // C++14 §20.8.2.3.4
    void swap(weak_ptr& r) noexcept;
    void reset() noexcept;

    // C++14 §20.8.2.3.5
    long use_count() const noexcept;
    bool expired() const noexcept;
    shared_ptr<T> lock() const noexcept;
    template<class U> bool owner_before(shared_ptr<U> const& b) const;
    template<class U> bool owner_before(weak_ptr<U> const& b) const;
  };

  // C++14 §20.8.2.3.6
  template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b) noexcept;

} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

8.2.2.1 `weak_ptr` constructors [memory.smartptr.weak.const]

1
```
weak_ptr(const weak_ptr& r) noexcept;
template<class Y> weak_ptr(const weak_ptr<Y>& r) noexcept;
template<class Y> weak_ptr(const shared_ptr<Y>& r) noexcept;
```

2 *Remarks:* The second and third constructors shall not participate in the overload resolution unless `Y*` is *compatible with* `T*`.

3 *Effects:* If `r` is *empty*, constructs an *empty* `weak_ptr` object; otherwise, constructs a `weak_ptr` object that *shares ownership* with `r` and stores a copy of the pointer stored in `r`.

4 *Postconditions:* `use_count() == r.use_count()`.

5 `weak_ptr(weak_ptr&& r) noexcept;`
`template<class Y> weak_ptr(weak_ptr<Y>&& r) noexcept;`

6 *Remarks:* The second constructor shall not participate in overload resolution unless `Y*` is *compatible with* `T*`.

7 *Effects:* Move-constructs a `weak_ptr` instance from `r`.

8 *Postconditions:* `*this` shall contain the old value of `r`. `r` shall be *empty*. `r.use_count() == 0`.

8.3 Type-erased allocator [memory.type.erased.allocator]

1 A *type-erased allocator* is an allocator or memory resource, `alloc`, used to allocate internal data structures for an object `X` of type `C`, but where `C` is not dependent on the type of `alloc`. Once `alloc` has been supplied to `X` (typically as a constructor argument), `alloc` can be retrieved from `X` only as a pointer `rptr` of static type `std::experimental::pmr::memory_resource*` (8.5). The process by which `rptr` is computed from `alloc` depends on the type of `alloc` as described in Table 15:

Table 15 — Computed `memory_resource` for type-erased allocator

If the type of `alloc` is	then the value of `rptr` is
non-existent — no `alloc` specified	The value of `experimental::pmr::get_default_resource()` at the time of construction.
`nullptr_t`	The value of `experimental::pmr::get_default_resource()` at the time of construction.
a pointer type convertible to `pmr::memory_resource*`	`static_cast<experimental::pmr::memory_resource*>(alloc)`
`pmr::polymorphic_allocator<U>`	`alloc.resource()`
any other type meeting the Allocator requirements (C++14 §17.6.3.5)	a pointer to a value of type `experimental::pmr::resource_adaptor<A>` where `A` is the type of `alloc`. `rptr` remains valid only for the lifetime of `X`.
None of the above	The program is ill-formed.

2 Additionally, class `C` shall meet the following requirements:

- `C::allocator_type` shall be identical to `std::experimental::erased_type`.
- `X.get_memory_resource()` returns `rptr`.

8.4 Header `<experimental/memory_resource>` synopsis [memory.resource.synop]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
namespace pmr {

  class memory_resource;

  bool operator==(const memory_resource& a,
                  const memory_resource& b) noexcept;
  bool operator!=(const memory_resource& a,
                  const memory_resource& b) noexcept;

  template <class Tp> class polymorphic_allocator;

  template <class T1, class T2>
  bool operator==(const polymorphic_allocator<T1>& a,
```

```
                 const polymorphic_allocator<T2>& b) noexcept;
template <class T1, class T2>
bool operator!=(const polymorphic_allocator<T1>& a,
                 const polymorphic_allocator<T2>& b) noexcept;

// The name resource_adaptor_imp is for exposition only.
template <class Allocator> class resource_adaptor_imp;

template <class Allocator>
  using resource_adaptor = resource_adaptor_imp<
    typename allocator_traits<Allocator>::template rebind_alloc<char>>;

// Global memory resources
memory_resource* new_delete_resource() noexcept;
memory_resource* null_memory_resource() noexcept;

// The default memory resource
memory_resource* set_default_resource(memory_resource* r) noexcept;
memory_resource* get_default_resource() noexcept;

// Standard memory resources
struct pool_options;
class synchronized_pool_resource;
class unsynchronized_pool_resource;
class monotonic_buffer_resource;

} // namespace pmr
} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

8.5 Class `memory_resource` [memory.resource]

8.5.1 Class `memory_resource` overview [memory.resource.overview]

1 The `memory_resource` class is an abstract interface to an unbounded set of classes encapsulating memory resources.

```
class memory_resource {
  // For exposition only
  static constexpr size_t max_align = alignof(max_align_t);

public:
  virtual ~memory_resource();

  void* allocate(size_t bytes, size_t alignment = max_align);
  void deallocate(void* p, size_t bytes,
                  size_t alignment = max_align);

  bool is_equal(const memory_resource& other) const noexcept;

protected:
  virtual void* do_allocate(size_t bytes, size_t alignment) = 0;
```

```
  virtual void do_deallocate(void* p, size_t bytes,
                             size_t alignment) = 0;

  virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;
};
```

8.5.2 `memory_resource` public member functions [memory.resource.public]

1 `~memory_resource();`

2 *Effects:* Destroys this memory_resource.

3 `void* allocate(size_t bytes, size_t alignment = max_align);`

4 *Effects:* Equivalent to `return do_allocate(bytes, alignment);`

5 `void deallocate(void* p, size_t bytes, size_t alignment = max_align);`

6 *Effects:* Equivalent to `do_deallocate(p, bytes, alignment);`

7 `bool is_equal(const memory_resource& other) const noexcept;`

8 *Effects:* Equivalent to `return do_is_equal(other);`

8.5.3 `memory_resource` protected virtual member functions [memory.resource.priv]

1 `virtual void* do_allocate(size_t bytes, size_t alignment) = 0;`

2 *Requires:* Alignment shall be a power of two.

3 *Returns:* A derived class shall implement this function to return a pointer to allocated storage (C++14 §3.7.4.2) with a size of at least `bytes`. The returned storage is aligned to the specified alignment, if such alignment is supported; otherwise it is aligned to `max_align`.

4 *Throws:* A derived class implementation shall throw an appropriate exception if it is unable to allocate memory with the requested size and alignment.

5 `virtual void do_deallocate(void* p, size_t bytes, size_t alignment) = 0;`

6 *Requires:* `p` shall have been returned from a prior call to `allocate(bytes, alignment)` on a memory resource equal to `*this`, and the storage at `p` shall not yet have been deallocated.

7 *Effects:* A derived class shall implement this function to dispose of allocated storage.

8 *Throws:* Nothing.

9 `virtual bool do_is_equal(const memory_resource& other) const noexcept = 0;`

10 *Returns:* A derived class shall implement this function to return `true` if memory allocated from this can be deallocated from other and vice-versa; otherwise it shall return false. [*Note:* The most-derived type of other might not match the type of this. For a derived class, D, a typical implementation of this function will compute `dynamic_cast<const D*>(&other)` and go no further (i.e., return `false`) if it returns `nullptr`. — *end note*]

8.5.4 `memory_resource` equality [memory.resource.eq]

1 `bool operator==(const memory_resource& a, const memory_resource& b) noexcept;`

2 *Returns:* `&a == &b || a.is_equal(b)`.

3 `bool operator!=(const memory_resource& a, const memory_resource& b) noexcept;`

4 *Returns:* `!(a == b)`.

8.6 Class template `polymorphic_allocator` [memory.polymorphic.allocator.class]

8.6.1 Class template `polymorphic_allocator` overview [memory.polymorphic.allocator.overview]

1 A specialization of class template `pmr::polymorphic_allocator` conforms to the `Allocator` requirements (C++14 §17.6.3.5). Constructed with different memory resources, different instances of the same specialization of `pmr::polymorphic_allocator` can exhibit entirely different allocation behavior. This runtime polymorphism allows objects that use `polymorphic_allocator` to behave as if they used different allocator types at run time even though they use the same static allocator type.

```
template <class Tp>
class polymorphic_allocator {
  memory_resource* m_resource; // For exposition only

public:
  using value_type = Tp;

  polymorphic_allocator() noexcept;
  polymorphic_allocator(memory_resource* r);

  polymorphic_allocator(const polymorphic_allocator& other) = default;

  template <class U>
    polymorphic_allocator(const polymorphic_allocator<U>& other) noexcept;

  polymorphic_allocator&
    operator=(const polymorphic_allocator& rhs) = default;

  Tp* allocate(size_t n);
  void deallocate(Tp* p, size_t n);

  template <class T, class... Args>
    void construct(T* p, Args&&... args);

  // Specializations for pair using piecewise construction
  template <class T1, class T2, class... Args1, class... Args2>
    void construct(pair<T1,T2>* p, piecewise_construct_t,
                   tuple<Args1...> x, tuple<Args2...> y);
  template <class T1, class T2>
    void construct(pair<T1,T2>* p);
  template <class T1, class T2, class U, class V>
    void construct(pair<T1,T2>* p, U&& x, V&& y);
  template <class T1, class T2, class U, class V>
```



```
    void construct(pair<T1,T2>* p, const std::pair<U, V>& pr);
  template <class T1, class T2, class U, class V>
    void construct(pair<T1,T2>* p, pair<U, V>&& pr);

  template <class T>
    void destroy(T* p);

  // Return a default-constructed allocator (no allocator propagation)
  polymorphic_allocator select_on_container_copy_construction() const;

  memory_resource* resource() const;
};
```

8.6.2 `polymorphic_allocator` constructors [memory.polymorphic.allocator.ctor]

1 `polymorphic_allocator() noexcept;`

2 *Effects:* Sets `m_resource` to `get_default_resource()`.

3 `polymorphic_allocator(memory_resource* r);`

4 *Requires:* `r` is non-null.

5 *Effects:* Sets `m_resource` to `r`.

6 *Throws:* Nothing.

7 *Notes:* This constructor provides an implicit conversion from `memory_resource*`.

8
```
template <class U>
   polymorphic_allocator(const polymorphic_allocator<U>& other) noexcept;
```

9 *Effects:* Sets `m_resource` to `other.resource()`.

8.6.3 `polymorphic_allocator` member functions [memory.polymorphic.allocator.mem]

1 `Tp* allocate(size_t n);`

2 *Returns:* Equivalent to `return static_cast<Tp*>(m_resource->allocate(n * sizeof(Tp), alignof(Tp)));`

3 `void deallocate(Tp* p, size_t n);`

4 *Requires:* `p` was allocated from a memory resource, `x`, equal to `*m_resource`, using `x.allocate(n * sizeof(Tp), alignof(Tp))`.

5 *Effects:* Equivalent to `m_resource->deallocate(p, n * sizeof(Tp), alignof(Tp))`.

6 *Throws:* Nothing.

7
```
template <class T, class... Args>
  void construct(T* p, Args&&... args);
```

8 *Requires:* *Uses-allocator construction* of `T` with allocator `this->resource()` (see 2.1) and constructor arguments `std::forward<Args>(args)...` is well-formed. [*Note: uses-allocator construction* is always well formed for types that do not use allocators. — *end note*]

9 *Effects:* Construct a `T` object at `p` by *uses-allocator construction* with allocator `this->resource()` (2.1) and constructor arguments `std::forward<Args>(args)...`.

10 *Throws:* Nothing unless the constructor for `T` throws.

11
```
template <class T1, class T2, class... Args1, class... Args2>
  void construct(pair<T1,T2>* p, piecewise_construct_t,
                 tuple<Args1...> x, tuple<Args2...> y);
```

12 *Effects:* Let `xprime` be a `tuple` constructed from `x` according to the appropriate rule from the following list. [*Note:* The following description can be summarized as constructing a `std::pair<T1,T2>` object at `p` as if by separate *uses-allocator construction* with allocator `this->resource()` (2.1) of `p->first` using the elements of `x` and `p->second` using the elements of `y`. — *end note*]

- If `uses_allocator_v<T1,memory_resource*>` is `false` and `is_constructible_v<T,Args1...>` is `true`, then `xprime` is `x`.
- Otherwise, if `uses_allocator_v<T1,memory_resource*>` is `true` and `is_constructible_v<T1,allocator_arg_t,memory_resource*,Args1...>` is `true`, then `xprime` is `tuple_cat(make_tuple(allocator_arg, this->resource()), std::move(x))`.
- Otherwise, if `uses_allocator_v<T1,memory_resource*>` is `true` and `is_constructible_v<T1,Args1...,memory_resource*>` is `true`, then `xprime` is `tuple_cat(std::move(x), make_tuple(this->resource()))`.
- Otherwise the program is ill formed.

and let `yprime` be a tuple constructed from `y` according to the appropriate rule from the following list:

- If `uses_allocator_v<T2,memory_resource*>` is `false` and `is_constructible_v<T,Args2...>` is `true`, then `yprime` is `y`.
- Otherwise, if `uses_allocator_v<T2,memory_resource*>` is `true` and `is_constructible_v<T2,allocator_arg_t,memory_resource*,Args2...>` is `true`, then `yprime` is `tuple_cat(make_tuple(allocator_arg, this->resource()), std::move(y))`.
- Otherwise, if `uses_allocator_v<T2,memory_resource*>` is `true` and `is_constructible_v<T2,Args2...,memory_resource*>` is `true`, then `yprime` is `tuple_cat(std::move(y), make_tuple(this->resource()))`.
- Otherwise the program is ill formed.

then this function constructs a `std::pair<T1,T2>` object at `p` using constructor arguments `piecewise_construct, xprime, yprime`.

13
```
template <class T1, class T2>
  void construct(std::pair<T1,T2>* p);
```

14 *Effects:* Equivalent to `this->construct(p, piecewise_construct, tuple<>(), tuple<>());`

15
```
template <class T1, class T2, class U, class V>
  void construct(std::pair<T1,T2>* p, U&& x, V&& y);
```

16 *Effects:* Equivalent to `this->construct(p, piecewise_construct, forward_as_tuple(std::forward<U>(x)), forward_as_tuple(std::forward<V>(y)));`

17
```
template <class T1, class T2, class U, class V>
  void construct(std::pair<T1,T2>* p, const std::pair<U, V>& pr);
```

18 *Effects:* Equivalent to `this->construct(p, piecewise_construct, forward_as_tuple(pr.first), forward_as_tuple(pr.second));`

19
```
template <class T1, class T2, class U, class V>
  void construct(std::pair<T1,T2>* p, std::pair<U, V>&& pr);
```

20 *Effects:* Equivalent to `this->construct(p, piecewise_construct, forward_as_tuple(std::forward<U>(pr.first)), forward_as_tuple(std::forward<V>(pr.second)));`

21
```
template <class T>
  void destroy(T* p);
```

22 *Effects:* `p->~T()`.

23
```
polymorphic_allocator select_on_container_copy_construction() const;
```

24 *Returns:* `polymorphic_allocator()`.

25
```
memory_resource* resource() const;
```

26 *Returns:* `m_resource`.

8.6.4 polymorphic_allocator equality [memory.polymorphic.allocator.eq]

1
```
template <class T1, class T2>
  bool operator==(const polymorphic_allocator<T1>& a,
                  const polymorphic_allocator<T2>& b) noexcept;
```

2 *Returns:* `*a.resource() == *b.resource()`.

3
```
template <class T1, class T2>
  bool operator!=(const polymorphic_allocator<T1>& a,
                  const polymorphic_allocator<T2>& b) noexcept;
```

4 *Returns:* `! (a == b)`.

8.7 template alias `resource_adaptor` [memory.resource.adaptor]

8.7.1 `resource_adaptor` [memory.resource.adaptor.overview]

1 An instance of `resource_adaptor<Allocator>` is an adaptor that wraps a `memory_resource` interface around `Allocator`. In order that `resource_adaptor<X<T>>` and `resource_adaptor<X<U>>` are the same type for any allocator template `X` and types `T` and `U`, `resource_adaptor<Allocator>` is rendered as an alias to a class template such that `Allocator` is rebound to a `char` value type in every specialization of the class template. The requirements on this class template are defined below. The name *`resource_adaptor_imp`* is for exposition only and is not normative, but the definitions of the members of that class, whatever its name, are normative. In addition to the `Allocator` requirements (C++14 §17.6.3.5), the parameter to `resource_adaptor` shall meet the following additional requirements:

- `typename allocator_traits<Allocator>::pointer` shall be identical to `typename allocator_traits<Allocator>::value_type*`.
- `typename allocator_traits<Allocator>::const_pointer` shall be identical to `typename allocator_traits<Allocator>::value_type const*`.
- `typename allocator_traits<Allocator>::void_pointer` shall be identical to `void*`.
- `typename allocator_traits<Allocator>::const_void_pointer` shall be identical to `void const*`.

```
// The name resource_adaptor_imp is for exposition only.
template <class Allocator>
class resource_adaptor_imp : public memory_resource {
  // for exposition only
  Allocator m_alloc;

public:
  using allocator_type = Allocator;

  resource_adaptor_imp() = default;
  resource_adaptor_imp(const resource_adaptor_imp&) = default;
  resource_adaptor_imp(resource_adaptor_imp&&) = default;

  explicit resource_adaptor_imp(const Allocator& a2);
  explicit resource_adaptor_imp(Allocator&& a2);

  resource_adaptor_imp& operator=(const resource_adaptor_imp&) = default;

  allocator_type get_allocator() const { return m_alloc; }

protected:
  virtual void* do_allocate(size_t bytes, size_t alignment);
  virtual void do_deallocate(void* p, size_t bytes, size_t alignment);

  virtual bool do_is_equal(const memory_resource& other) const noexcept;
};

template <class Allocator>
  using resource_adaptor = typename resource_adaptor_imp<
    typename allocator_traits<Allocator>::template rebind_alloc<char>>;
```

8.7.2 `resource_adaptor_imp` constructors [memory.resource.adaptor.ctor]

1 `explicit resource_adaptor_imp(const Allocator& a2);`

2 *Effects:* Initializes `m_alloc` with `a2`.

3 `explicit resource_adaptor_imp(Allocator&& a2);`

4 *Effects:* Initializes `m_alloc` with `std::move(a2)`.

8.7.3 `resource_adaptor_imp` member functions [memory.resource.adaptor.mem]

1 `void* do_allocate(size_t bytes, size_t alignment);`

2 *Returns:* Allocated memory obtained by calling `m_alloc.allocate`. The size and alignment of the allocated memory shall meet the requirements for a class derived from `memory_resource` (8.5).

3 `void do_deallocate(void* p, size_t bytes, size_t alignment);`

4 *Requires:* `p` was previously allocated using `A.allocate`, where `A == m_alloc`, and not subsequently deallocated.

5 *Effects:* Returns memory to the allocator using `m_alloc.deallocate()`.

6 `bool do_is_equal(const memory_resource& other) const noexcept;`

7 Let `p` be `dynamic_cast<const resource_adaptor_imp*>(&other)`.

8 *Returns:* `false` if `p` is null, otherwise the value of `m_alloc == p->m_alloc`.

8.8 Access to program-wide `memory_resource` objects [memory.resource.global]

1 `memory_resource* new_delete_resource() noexcept;`

2 *Returns:* A pointer to a static-duration object of a type derived from `memory_resource` that can serve as a resource for allocating memory using `::operator new` and `::operator delete`. The same value is returned every time this function is called. For return value `p` and memory resource `r`, `p->is_equal(r)` returns `&r == p`.

3 `memory_resource* null_memory_resource() noexcept;`

4 *Returns:* A pointer to a static-duration object of a type derived from `memory_resource` for which `allocate()` always throws `bad_alloc` and for which `deallocate()` has no effect. The same value is returned every time this function is called. For return value `p` and memory resource `r`, `p->is_equal(r)` returns `&r == p`.

5 The *default memory resource pointer* is a pointer to a memory resource that is used by certain facilities when an explicit memory resource is not supplied through the interface. Its initial value is the return value of `new_delete_resource()`.

6 `memory_resource* set_default_resource(memory_resource* r) noexcept;`

7 *Effects:* If `r` is non-null, sets the value of the default memory resource pointer to `r`, otherwise sets the default memory resource pointer to `new_delete_resource()`.

8 *Returns:* The previous value of the default memory resource pointer.

9 *Remarks:* Calling the `set_default_resource` and `get_default_resource` functions shall not incur a data race. A call to the `set_default_resource` function shall synchronize with subsequent calls to the `set_default_resource` and `get_default_resource` functions.

10 `memory_resource* get_default_resource() noexcept;`

11 *Returns:* The current value of the default memory resource pointer.

8.9 Pool resource classes [memory.resource.pool]

8.9.1 Classes `synchronized_pool_resource` and `unsynchronized_pool_resource` [memory.resource.pool.overview]

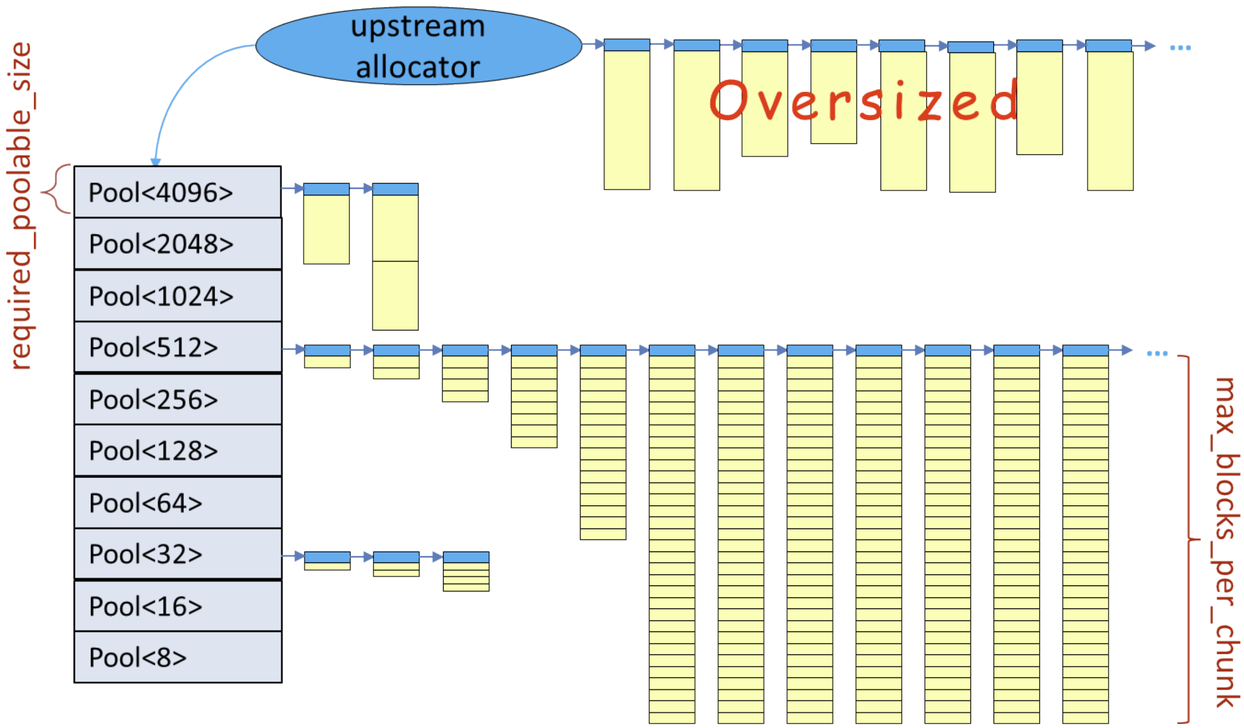
1 The `synchronized_pool_resource` and `unsynchronized_pool_resource` classes (collectively, *pool resource classes*) are general-purpose memory resources having the following qualities:

- Each resource *owns* the allocated memory, and frees it on destruction – even if `deallocate` has not been called for some of the allocated blocks.
- A pool resource (see Figure 1) consists of a collection of *pools*, serving requests for different block sizes. Each individual pool manages a collection of *chunks* that are in turn divided into blocks of uniform size, returned via calls to `do_allocate`. Each call to `do_allocate(size, alignment)` is dispatched to the pool serving the smallest blocks accommodating at least `size` bytes.
- When a particular pool is exhausted, allocating a block from that pool results in the allocation of an additional chunk of memory from the *upstream allocator* (supplied at construction), thus replenishing the pool. With each successive replenishment, the chunk size obtained increases geometrically. [*Note:* By allocating memory in chunks, the pooling strategy increases the chance that consecutive allocations will be close together in memory. — *end note*]

- Allocation requests that exceed the largest block size of any pool are fulfilled directly from the upstream allocator.
- A `pool_options` struct may be passed to the pool resource constructors to tune the largest block size and the maximum chunk size.

[*Example:* Figure 1 shows a possible data structure that implements a pool resource.

Figure 1 — pool resource



— *end example*]

2 A `synchronized_pool_resource` may be accessed from multiple threads without external synchronization and may have thread-specific pools to reduce synchronization costs. An `unsynchronized_pool_resource` class may not be accessed from multiple threads simultaneously and thus avoids the cost of synchronization entirely in single-threaded applications.

```
struct pool_options {
  size_t max_blocks_per_chunk = 0;
  size_t largest_required_pool_block = 0;
};

class synchronized_pool_resource : public memory_resource {
public:
  synchronized_pool_resource(const pool_options& opts, memory_resource* upstream);

  synchronized_pool_resource()
      : synchronized_pool_resource(pool_options(), get_default_resource()) { }
  explicit synchronized_pool_resource(memory_resource* upstream)
      : synchronized_pool_resource(pool_options(), upstream) { }
  explicit synchronized_pool_resource(const pool_options& opts)
      : synchronized_pool_resource(opts, get_default_resource()) { }

  synchronized_pool_resource(
```



```
      const synchronized_pool_resource&) = delete;
  virtual ~synchronized_pool_resource();

  synchronized_pool_resource& operator=(
      const synchronized_pool_resource&) = delete;

  void release();
  memory_resource* upstream_resource() const;
  pool_options options() const;

protected:
  virtual void* do_allocate(size_t bytes, size_t alignment);
  virtual void do_deallocate(void* p, size_t bytes, size_t alignment);

  virtual bool do_is_equal(const memory_resource& other) const noexcept;
};

class unsynchronized_pool_resource : public memory_resource {
public:
  unsynchronized_pool_resource(const pool_options& opts, memory_resource* upstream);

  unsynchronized_pool_resource()
      : unsynchronized_pool_resource(pool_options(), get_default_resource()) { }
  explicit unsynchronized_pool_resource(memory_resource* upstream)
      : unsynchronized_pool_resource(pool_options(), upstream) { }
  explicit unsynchronized_pool_resource(const pool_options& opts)
      : unsynchronized_pool_resource(opts, get_default_resource()) { }

  unsynchronized_pool_resource(
      const unsynchronized_pool_resource&) = delete;
  virtual ~unsynchronized_pool_resource();

  unsynchronized_pool_resource& operator=(
      const unsynchronized_pool_resource&) = delete;

  void release();
  memory_resource* upstream_resource() const;
  pool_options options() const;

protected:
  virtual void* do_allocate(size_t bytes, size_t alignment);
  virtual void do_deallocate(void* p, size_t bytes, size_t alignment);

  virtual bool do_is_equal(const memory_resource& other) const noexcept;
};
```

8.9.2 `pool_options` data members [memory.resource.pool.options]

1 The members of `pool_options` comprise a set of constructor options for pool resources. The effect of each option on the pool resource behavior is described below:

2 `size_t max_blocks_per_chunk;`

3 The maximum number of blocks that will be allocated at once from the upstream memory resource to replenish a pool. If the value of `max_blocks_per_chunk` is zero or is greater than an implementation-defined limit, that limit is used instead. The implementation may choose to use a smaller value than is specified in this field and may use different values for different pools.

4 `size_t largest_required_pool_block;`

5 The largest allocation size that is required to be fulfilled using the pooling mechanism. Attempts to allocate a single block larger than this threshold will be allocated directly from the upstream memory resource. If `largest_required_pool_block` is zero or is greater than an implementation-defined limit, that limit is used instead. The implementation may choose a pass-through threshold larger than specified in this field.

8.9.3 pool resource constructors and destructors [memory.resource.pool.ctor]

1
```
synchronized_pool_resource(const pool_options& opts, memory_resource* upstream);
unsynchronized_pool_resource(const pool_options& opts, memory_resource* upstream);
```

2 *Requires:* `upstream` is the address of a valid memory resource.

3 *Effects:* Constructs a pool resource object that will obtain memory from `upstream` whenever the pool resource is unable to satisfy a memory request from its own internal data structures. The resulting object will hold a copy of `upstream`, but will not own the resource to which `upstream` points. [*Note:* The intention is that calls to `upstream->allocate()` will be substantially fewer than calls to `this->allocate()` in most cases. — *end note*] The behavior of the pooling mechanism is tuned according to the value of the opts argument.

4 *Throws:* Nothing unless `upstream->allocate()` throws. It is unspecified if or under what conditions this constructor calls `upstream->allocate()`.

5
```
virtual ~synchronized_pool_resource();
virtual ~unsynchronized_pool_resource();
```

6 *Effects:* Calls `this->release()`.

8.9.4 pool resource members [memory.resource.pool.mem]

1 `void release();`

2 *Effects:* Calls `upstream_resource()->deallocate()` as necessary to release all allocated memory. [*Note:* memory is released back to `upstream_resource()` even if `deallocate` has not been called for some of the allocated blocks. — *end note*]

3 `memory_resource* upstream_resource() const;`

4 *Returns:* The value of the `upstream` argument provided to the constructor of this object.

5 `pool_options options() const;`

6 *Returns:* The options that control the pooling behavior of this resource. The values in the returned struct may differ from those supplied to the pool resource constructor in that values of zero will be replaced with implementation-defined defaults and sizes may be rounded to unspecified granularity.

7 `virtual void* do_allocate(size_t bytes, size_t alignment);`

8 *Returns:* A pointer to allocated storage (C++14 §3.7.4.2) with a size of at least `bytes`. The size and alignment of the allocated memory shall meet the requirements for a class derived from `memory_resource` (8.5).

9 *Effects:* If the pool selected for a block of size `bytes` is unable to satisfy the memory request from its own internal data structures, it will call `upstream_resource()->allocate()` to obtain more memory. If `bytes` is larger than that which the largest pool can handle, then memory will be allocated using `upstream_resource()->allocate()`.

10 *Throws:* Nothing unless `upstream_resource()->allocate()` throws.

11 `virtual void do_deallocate(void* p, size_t bytes, size_t alignment);`

12 *Effects:* Return the memory at `p` to the pool. It is unspecified if or under what circumstances this operation will result in a call to `upstream_resource()->deallocate()`.

13 *Throws:* Nothing.

14 `virtual bool unsynchronized_pool_resource::do_is_equal(const memory_resource& other) const noexcept;`

15 *Returns:* `this == dynamic_cast<const unsynchronized_pool_resource*>(&other)`.

16 `virtual bool synchronized_pool_resource::do_is_equal(const memory_resource& other) const noexcept;`

17 *Returns:* `this == dynamic_cast<const synchronized_pool_resource*>(&other)`.

8.10 Class `monotonic_buffer_resource` [memory.resource.monotonic.buffer]

8.10.1 Class `monotonic_buffer_resource` overview [memory.resource.monotonic.buffer.overview]

1 A `monotonic_buffer_resource` is a special-purpose memory resource intended for very fast memory allocations in situations where memory is used to build up a few objects and then is released all at once when the memory resource object is destroyed. It has the following qualities:

- A call to `deallocate` has no effect, thus the amount of memory consumed increases monotonically until the resource is destroyed.
- The program can supply an initial buffer, which the allocator uses to satisfy memory requests.
- When the initial buffer (if any) is exhausted, it obtains additional buffers from an *upstream* memory resource supplied at construction. Each additional buffer is larger than the previous one, following a geometric progression.
- It is intended for access from one thread of control at a time. Specifically, calls to `allocate` and `deallocate` do not synchronize with one another.
- It *owns* the allocated memory and frees it on destruction, even if `deallocate` has not been called for some of the allocated blocks.

```
class monotonic_buffer_resource : public memory_resource {
  memory_resource* upstream_rsrc; // exposition only
  void* current_buffer; // exposition only
  size_t next_buffer_size; // exposition only

public:
  explicit monotonic_buffer_resource(memory_resource* upstream);
  monotonic_buffer_resource(size_t initial_size,
                            memory_resource* upstream);
  monotonic_buffer_resource(void* buffer, size_t buffer_size,
```



```
                                  memory_resource* upstream);

    monotonic_buffer_resource()
        : monotonic_buffer_resource(get_default_resource()) { }
    explicit monotonic_buffer_resource(size_t initial_size)
        : monotonic_buffer_resource(initial_size,
                                    get_default_resource()) { }
    monotonic_buffer_resource(void* buffer, size_t buffer_size)
        : monotonic_buffer_resource(buffer, buffer_size,
                                    get_default_resource()) { }

    monotonic_buffer_resource(const monotonic_buffer_resource&) = delete;

    virtual ~monotonic_buffer_resource();

    monotonic_buffer_resource operator=(
        const monotonic_buffer_resource&) = delete;

    void release();
    memory_resource* upstream_resource() const;

  protected:
    virtual void* do_allocate(size_t bytes, size_t alignment);
    virtual void do_deallocate(void* p, size_t bytes,
                               size_t alignment);

    virtual bool do_is_equal(const memory_resource& other) const noexcept;
  };
```

8.10.2 `monotonic_buffer_resource` constructor and destructor [memory.resource.monotonic.buffer.ctor]

1
```
explicit monotonic_buffer_resource(memory_resource* upstream);
monotonic_buffer_resource(size_t initial_size, memory_resource* upstream);
```

2 *Requires:* `upstream` shall be the address of a valid memory resource. `initial_size`, if specified, shall be greater than zero.

3 *Effects:* Sets `upstream_rsrc` to `upstream` and `current_buffer` to `nullptr`. If `initial_size` is specified, sets `next_buffer_size` to at least `initial_size`; otherwise sets `next_buffer_size` to an implementation-defined size.

4
```
monotonic_buffer_resource(void* buffer, size_t buffer_size, memory_resource* upstream);
```

5 *Requires:* `upstream` shall be the address of a valid memory resource. `buffer_size` shall be no larger than the number of bytes in `buffer`.

6 *Effects:* Sets `upstream_rsrc` to `upstream`, `current_buffer` to `buffer`, and `next_buffer_size` to `buffer_size` (but not less than 1), then increases `next_buffer_size` by an implementation-defined growth factor (which need not be integral).

7
```
~monotonic_buffer_resource();
```

8 *Effects:* Calls `this->release()`.

8.10.3 monotonic_buffer_resource members [memory.resource.monotonic.buffer.mem]

1 `void release();`

2 *Effects:* Calls `upstream_rsrc->deallocate()` as necessary to release all allocated memory.

3 [*Note:* memory is released back to `upstream_rsrc` even if some blocks that were allocated from `this` have not been deallocated from `this`. — *end note*]

4 `memory_resource* upstream_resource() const;`

5 *Returns:* The value of `upstream_rsrc`.

6 `void* do_allocate(size_t bytes, size_t alignment);`

7 *Returns:* A pointer to allocated storage (C++14 §3.7.4.2) with a size of at least `bytes`. The size and alignment of the allocated memory shall meet the requirements for a class derived from `memory_resource` (8.5).

8 *Effects:* If the unused space in `current_buffer` can fit a block with the specified `bytes` and `alignment`, then allocate the return block from `current_buffer`; otherwise set `current_buffer` to `upstream_rsrc->allocate(n, m)`, where `n` is not less than `max(bytes, next_buffer_size)` and `m` is not less than `alignment`, and increase `next_buffer_size` by an implementation-defined growth factor (which need not be integral), then allocate the return block from the newly-allocated `current_buffer`.

9 *Throws:* Nothing unless `upstream_rsrc->allocate()` throws.

10 `void do_deallocate(void* p, size_t bytes, size_t alignment);`

11 *Effects:* None.

12 *Throws:* Nothing.

13 *Remarks:* Memory used by this resource increases monotonically until its destruction.

14 `bool do_is_equal(const memory_resource& other) const noexcept;`

15 *Returns:* `this == dynamic_cast<const monotonic_buffer_resource*>(&other)`.

8.11 Alias templates using polymorphic memory resources [memory.resource.aliases]

8.11.1 Header <experimental/string> synopsis [header.string.synop]

```
#include <string>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
namespace pmr {

  // basic_string using polymorphic allocator in namespace pmr
  template <class charT, class traits = char_traits<charT>>
   using basic_string =
     std::basic_string<charT, traits, polymorphic_allocator<charT>>;

  // basic_string typedef names using polymorphic allocator in namespace
```



```
  // std::experimental::pmr
  using string = basic_string<char>;
  using u16string = basic_string<char16_t>;
  using u32string = basic_string<char32_t>;
  using wstring =  basic_string<wchar_t>;

} // namespace pmr
} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

8.11.2 Header <experimental/deque> synopsis [header.deque.synop]

```
#include <deque>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
namespace pmr {

  template <class T>
  using deque = std::deque<T,polymorphic_allocator<T>>;

} // namespace pmr
} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

8.11.3 Header <experimental/forward_list> synopsis [header.forward_list.synop]

```
#include <forward_list>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
namespace pmr {

  template <class T>
  using forward_list =
    std::forward_list<T,polymorphic_allocator<T>>;

} // namespace pmr
} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

8.11.4 Header <experimental/list> synopsis [header.list.synop]

```
#include <list>

namespace std {
```



```
namespace experimental {
inline namespace fundamentals_v2 {
namespace pmr {

  template <class T>
  using list = std::list<T,polymorphic_allocator<T>>;

} // namespace pmr
} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

8.11.5 Header <experimental/vector> synopsis [header.vector.synop]

```
#include <vector>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
namespace pmr {

  template <class T>
  using vector = std::vector<T,polymorphic_allocator<T>>;

} // namespace pmr
} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

8.11.6 Header <experimental/map> synopsis [header.map.synop]

```
#include <map>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
namespace pmr {

  template <class Key, class T, class Compare = less<Key>>
  using map = std::map<Key, T, Compare,
                       polymorphic_allocator<pair<const Key,T>>>;

  template <class Key, class T, class Compare = less<Key>>
  using multimap = std::multimap<Key, T, Compare,
                                 polymorphic_allocator<pair<const Key,T>>>;

} // namespace pmr
} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```


8.11.7 Header <experimental/set> synopsis [header.set.synop]

```
#include <set>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
namespace pmr {

  template <class Key, class Compare = less<Key>>
  using set = std::set<Key, Compare,
                       polymorphic_allocator<Key>>;

  template <class Key, class Compare = less<Key>>
  using multiset = std::multiset<Key, Compare,
                                 polymorphic_allocator<Key>>;

} // namespace pmr
} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

8.11.8 Header <experimental/unordered_map> synopsis [header.unordered_map.synop]

```
#include <unordered_map>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
namespace pmr {

  template <class Key, class T,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>>
  using unordered_map =
    std::unordered_map<Key, T, Hash, Pred,
                       polymorphic_allocator<pair<const Key,T>>>;

  template <class Key, class T,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>>
  using unordered_multimap =
    std::unordered_multimap<Key, T, Hash, Pred,
                            polymorphic_allocator<pair<const Key,T>>>;

} // namespace pmr
} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```


8.11.9 Header <experimental/unordered_set> synopsis [header.unordered_set.synop]

```
#include <unordered_set>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
namespace pmr {

  template <class Key,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>>
  using unordered_set = std::unordered_set<Key, Hash, Pred,
                                           polymorphic_allocator<Key>>;

  template <class Key,
            class Hash = hash<Key>,
            class Pred = equal_to<Key>>
  using unordered_multiset =
    std::unordered_multiset<Key, Hash, Pred,
                            polymorphic_allocator<Key>>;

} // namespace pmr
} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

8.11.10 Header <experimental/regex> synopsis [header.regex.synop]

```
#include <regex>
#include <experimental/string>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
namespace pmr {

  template <class BidirectionalIterator>
  using match_results =
    std::match_results<BidirectionalIterator,
                       polymorphic_allocator<sub_match<BidirectionalIterator>>>;

  using cmatch = match_results<const char*>;
  using wcmatch = match_results<const wchar_t*>;
  using smatch = match_results<string::const_iterator>;
  using wsmatch = match_results<wstring::const_iterator>;

} // namespace pmr
} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```


8.12 Non-owning pointers [memory.observer.ptr]

1 A non-owning pointer, known as an *observer*, is an object `o` that stores a pointer to a second object, `w`. In this context, `w` is known as a *watched* object. [*Note:* There is no watched object when the stored pointer is `nullptr`. — *end note*] An observer takes no responsibility or ownership of any kind for its watched object, if any; in particular, there is no inherent relationship between the lifetimes of `o` and `w`.

2 Specializations of `observer_ptr` shall meet the requirements of a `CopyConstructible` and `CopyAssignable` type. The template parameter `W` of an `observer_ptr` shall not be a reference type, but may be an incomplete type.

3 [*Note:* The uses of `observer_ptr` include clarity of interface specification in new code, and interoperability with pointer-based legacy code. — *end note*]

8.12.1 Class template `observer_ptr` overview [memory.observer.ptr.overview]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  template <class W> class observer_ptr {
    using pointer = add_pointer_t<W>;            // exposition-only
    using reference = add_lvalue_reference_t<W>; // exposition-only
  public:
    // publish our template parameter and variations thereof
    using element_type = W;

    // 8.12.2, observer_ptr constructors
    // default c'tor
    constexpr observer_ptr() noexcept;

    // pointer-accepting c'tors
    constexpr observer_ptr(nullptr_t) noexcept;
    constexpr explicit observer_ptr(pointer) noexcept;

    // copying c'tors (in addition to compiler-generated copy c'tor)
    template <class W2> constexpr observer_ptr(observer_ptr<W2>) noexcept;

    // 8.12.3, observer_ptr observers
    constexpr pointer get() const noexcept;
    constexpr reference operator*() const;
    constexpr pointer operator->() const noexcept;
    constexpr explicit operator bool() const noexcept;

    // 8.12.4, observer_ptr conversions
    constexpr explicit operator pointer() const noexcept;

    // 8.12.5, observer_ptr modifiers
    constexpr pointer release() noexcept;
    constexpr void reset(pointer = nullptr) noexcept;
    constexpr void swap(observer_ptr&) noexcept;
  }; // observer_ptr<>

} // inline namespace fundamentals_v2
```

```
} // namespace experimental
} // namespace std
```

8.12.2 `observer_ptr` constructors [memory.observer.ptr.ctor]

1 `constexpr observer_ptr() noexcept;`
`constexpr observer_ptr(nullptr_t) noexcept;`

2 *Effects:* Constructs an observer_ptr object that has no corresponding watched object.

3 *Postconditions:* `get() == nullptr`.

4 `constexpr explicit observer_ptr(pointer other) noexcept;`

5 *Postconditions:* `get() == other`.

6 `template <class W2> constexpr observer_ptr(observer_ptr<W2> other) noexcept;`

7 *Postconditions:* `get() == other.get()`.

8 *Remarks:* This constructor shall not participate in overload resolution unless `W2*` is convertible to `W*`.

8.12.3 `observer_ptr` observers [memory.observer.ptr.obs]

1 `constexpr pointer get() const noexcept;`

2 *Returns:* The stored pointer.

3 `constexpr reference operator*() const;`

4 *Requires:* `get() != nullptr`.

5 *Returns:* `*get()`.

6 *Throws:* Nothing.

7 `constexpr pointer operator->() const noexcept;`

8 *Returns:* `get()`.

9 `constexpr explicit operator bool() const noexcept;`

10 *Returns:* `get() != nullptr`.

8.12.4 `observer_ptr` conversions [memory.observer.ptr.conv]

1 `constexpr explicit operator pointer() const noexcept;`

2 *Returns:* `get()`.

8.12.5 `observer_ptr` modifiers [memory.observer.ptr.mod]

1 `constexpr pointer release() noexcept;`

2 *Postconditions:* `get() == nullptr`.

3 *Returns:* The value `get()` had at the start of the call to `release`.

4
```
constexpr void reset(pointer p = nullptr) noexcept;
```

5 *Postconditions:* `get() == p`.

6
```
constexpr void swap(observer_ptr& other) noexcept;
```

7 *Effects:* Invokes `swap` on the stored pointers of `*this` and `other`.

8.12.6 `observer_ptr` specialized algorithms [memory.observer.ptr.special]

1
```
template <class W>
  void swap(observer_ptr<W>& p1, observer_ptr<W>& p2) noexcept;
```

2 *Effects:* `p1.swap(p2)`.

3
```
template <class W> observer_ptr<W> make_observer(W* p) noexcept;
```

4 *Returns:* `observer_ptr<W>{p}`.

5
```
template <class W1, class W2>
  bool operator==(observer_ptr<W1> p1, observer_ptr<W2> p2);
```

6 *Returns:* `p1.get() == p2.get()`.

7
```
template <class W1, class W2>
  bool operator!=(observer_ptr<W1> p1, observer_ptr<W2> p2);
```

8 *Returns:* `not (p1 == p2)`.

9
```
template <class W>
  bool operator==(observer_ptr<W> p, nullptr_t) noexcept;
template <class W>
  bool operator==(nullptr_t, observer_ptr<W> p) noexcept;
```

10 *Returns:* `not p`.

11
```
template <class W>
  bool operator!=(observer_ptr<W> p, nullptr_t) noexcept;
template <class W>
  bool operator!=(nullptr_t, observer_ptr<W> p) noexcept;
```

12 *Returns:* `(bool)p`.

13
```
template <class W1, class W2>
  bool operator<(observer_ptr<W1> p1, observer_ptr<W2> p2);
```

14 *Returns:* `less<W3>()(p1.get(), p2.get())`, where `W3` is the composite pointer type (C++14 §5) of `W1*` and `W2*`.

15
```
template <class W1, class W2>
  bool operator>(observer_ptr<W1> p1, observer_ptr<W2> p2);
```

16 *Returns:* `p2 < p1`.

17
```
template <class W1, class W2>
  bool operator<=(observer_ptr<W1> p1, observer_ptr<W2> p2);
```

18 *Returns:* `not (p2 < p1)`.

19
```
template <class W1, class W2>
  bool operator>=(observer_ptr<W1> p1, observer_ptr<W2> p2);
```

20 *Returns:* `not (p1 < p2)`.

8.12.7 `observer_ptr` hash support [memory.observer.ptr.hash]

1
```
template <class T> struct hash<experimental::observer_ptr<T>>;
```

2 The template specialization shall meet the requirements of class template `hash` (C++14 §20.9.12). For an object `p` of type `observer_ptr<T>`, `hash<observer_ptr<T>>()(p)` shall evaluate to the same value as `hash<T*>()(p.get())`.

9 Containers [container]

9.1 Uniform container erasure [container.erasure]

9.1.1 Header synopsis [container.erasure.syn]

1 For brevity, this section specifies the contents of 9 headers, each of which behaves as described by 1.3.

```
namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  // 9.1.2, Function template erase_if
  // 9.1.3, Function template erase

  // <experimental/string>
  template <class charT, class traits, class A, class Predicate>
    void erase_if(basic_string<charT, traits, A>& c, Predicate pred);
  template <class charT, class traits, class A, class U>
    void erase(basic_string<charT, traits, A>& c, const U& value);

  // <experimental/deque>
  template <class T, class A, class Predicate>
    void erase_if(deque<T, A>& c, Predicate pred);
  template <class T, class A, class U>
    void erase(deque<T, A>& c, const U& value);

  // <experimental/vector>
  template <class T, class A, class Predicate>
    void erase_if(vector<T, A>& c, Predicate pred);
  template <class T, class A, class U>
    void erase(vector<T, A>& c, const U& value);

  // <experimental/forward_list>
  template <class T, class A, class Predicate>
    void erase_if(forward_list<T, A>& c, Predicate pred);
  template <class T, class A, class U>
    void erase(forward_list<T, A>& c, const U& value);

  // <experimental/list>
  template <class T, class A, class Predicate>
    void erase_if(list<T, A>& c, Predicate pred);
  template <class T, class A, class U>
    void erase(list<T, A>& c, const U& value);

  // <experimental/map>
  template <class K, class T, class C, class A, class Predicate>
    void erase_if(map<K, T, C, A>& c, Predicate pred);
  template <class K, class T, class C, class A, class Predicate>
    void erase_if(multimap<K, T, C, A>& c, Predicate pred);
```



```
    // <experimental/set>
    template <class K, class C, class A, class Predicate>
      void erase_if(set<K, C, A>& c, Predicate pred);
    template <class K, class C, class A, class Predicate>
      void erase_if(multiset<K, C, A>& c, Predicate pred);

    // <experimental/unordered_map>
    template <class K, class T, class H, class P, class A, class Predicate>
      void erase_if(unordered_map<K, T, H, P, A>& c, Predicate pred);
    template <class K, class T, class H, class P, class A, class Predicate>
      void erase_if(unordered_multimap<K, T, H, P, A>& c, Predicate pred);

    // <experimental/unordered_set>
    template <class K, class H, class P, class A, class Predicate>
      void erase_if(unordered_set<K, H, P, A>& c, Predicate pred);
    template <class K, class H, class P, class A, class Predicate>
      void erase_if(unordered_multiset<K, H, P, A>& c, Predicate pred);

  } // inline namespace fundamentals_v2
  } // namespace experimental
  } // namespace std
```

9.1.2 Function template `erase_if` [container.erasure.erase_if]

1
```
template <class charT, class traits, class A, class Predicate>
    void erase_if(basic_string<charT, traits, A>& c, Predicate pred);
template <class T, class A, class Predicate>
    void erase_if(deque<T, A>& c, Predicate pred);
template <class T, class A, class Predicate>
    void erase_if(vector<T, A>& c, Predicate pred);
```

2 *Effects:* Equivalent to: `c.erase(remove_if(c.begin(), c.end(), pred), c.end());`

3
```
template <class T, class A, class Predicate>
    void erase_if(forward_list<T, A>& c, Predicate pred);
template <class T, class A, class Predicate>
    void erase_if(list<T, A>& c, Predicate pred);
```

4 *Effects:* Equivalent to: `c.remove_if(pred);`

5

```
template <class K, class T, class C, class A, class Predicate>
   void erase_if(map<K, T, C, A>& c, Predicate pred);
template <class K, class T, class C, class A, class Predicate>
   void erase_if(multimap<K, T, C, A>& c, Predicate pred);
template <class K, class C, class A, class Predicate>
   void erase_if(set<K, C, A>& c, Predicate pred);
template <class K, class C, class A, class Predicate>
   void erase_if(multiset<K, C, A>& c, Predicate pred);
template <class K, class T, class H, class P, class A, class Predicate>
   void erase_if(unordered_map<K, T, H, P, A>& c, Predicate pred);
template <class K, class T, class H, class P, class A, class Predicate>
   void erase_if(unordered_multimap<K, T, H, P, A>& c, Predicate pred);
template <class K, class H, class P, class A, class Predicate>
   void erase_if(unordered_set<K, H, P, A>& c, Predicate pred);
template <class K, class H, class P, class A, class Predicate>
   void erase_if(unordered_multiset<K, H, P, A>& c, Predicate pred);
```

6 *Effects:* Equivalent to:

```
for (auto i = c.begin(), last = c.end(); i != last; ) {
  if (pred(*i)) {
    i = c.erase(i);
  } else {
    ++i;
  }
}
```

9.1.3 Function template erase [container.erasure.erase]

1

```
template <class charT, class traits, class A, class U>
   void erase(basic_string<charT, traits, A>& c, const U& value);
template <class T, class A, class U>
   void erase(deque<T, A>& c, const U& value);
template <class T, class A, class U>
   void erase(vector<T, A>& c, const U& value);
```

2 *Effects:* Equivalent to: `c.erase(remove(c.begin(), c.end(), value), c.end());`

3

```
template <class T, class A, class U>
   void erase(forward_list<T, A>& c, const U& value);
template <class T, class A, class U>
   void erase(list<T, A>& c, const U& value);
```

4 *Effects:* Equivalent to: `erase_if(c, [&](auto& elem) { return elem == value; });`

[*Note:* Overloads of `erase()` for associative containers and unordered associative containers are intentionally not provided. — *end note*]

9.2 Class template `array` [container.array]

9.2.1 Header <experimental/array> synopsis [header.array.synop]

```
#include <array>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {
```



```
  // 9.2.2, Array creation functions
  template <class D = void, class... Types>
    constexpr array<VT, sizeof...(Types)> make_array(Types&&... t);
  template <class T, size_t N>
    constexpr array<remove_cv_t<T>, N> to_array(T (&a)[N]);

} // inline namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

9.2.2 Array creation functions [container.array.creation]

1
```
template <class D = void, class... Types>
      constexpr array<VT, sizeof...(Types)> make_array(Types&&... t);
```

2 Let *Ui* be `decay_t<Ti>` for each *Ti* in `Types`.

3 *Remarks:* The program is ill-formed if `D` is `void` and at least one *Ui* is a specialization of `reference_wrapper`.

4 *Returns:* `array<VT, sizeof...(Types)>{ std::forward<Types>(t)... }`, where *`VT`* is `common_type_t<Types...>` if `D` is `void`, otherwise *`VT`* is `D`.

[*Example:*

```
int i = 1; int& ri = i;
auto a1 = make_array(i, ri);          // a1 is of type array<int, 2>
auto a2 = make_array(i, ri, 42L);     // a2 is of type array<long, 3>
auto a3 = make_array<long>(i, ri);    // a3 is of type array<long, 2>
auto a4 = make_array<long>();         // a4 is of type array<long, 0>
auto a5 = make_array();               // ill-formed
```

— *end example*]

5
```
template <class T, size_t N>
    constexpr array<remove_cv_t<T>, N> to_array(T (&a)[N]);
```

6 *Returns:* An `array<remove_cv_t<T>, N>` such that each element is copy-initialized with the corresponding element of `a`.

10 Iterators library [iterator]

10.1 Header `<experimental/iterator>` synopsis [iterator.synopsis]

```
#include <iterator>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  // 10.2, Class template ostream_joiner
  template <class DelimT, class charT = char, class traits = char_traits<charT> >
      class ostream_joiner;
  template <class charT, class traits, class DelimT>
    ostream_joiner<decay_t<DelimT>, charT, traits>
    make_ostream_joiner(basic_ostream<charT, traits>& os, DelimT&& delimiter);

} // inline namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

10.2 Class template `ostream_joiner` [iterator.ostream.joiner]

1 `ostream_joiner` writes (using `operator<<`) successive elements onto the output stream from which it was constructed. The delimiter that it was constructed with is written to the stream between every two `T`s that are written. It is not possible to get a value out of the output iterator. Its only use is as an output iterator in situations like

```
while (first != last)
  *result++ = *first++;
```

2 `ostream_joiner` is defined as

```
namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  template <class DelimT, class charT = char, class traits = char_traits<charT> >
  class ostream_joiner {
  public:
    using char_type = charT;
    using traits_type = traits;
    using ostream_type = basic_ostream<charT, traits>;
    using iterator_category = output_iterator_tag;
    using value_type = void;
    using difference_type = void;
    using pointer = void;
    using reference = void;

    ostream_joiner(ostream_type& s, const DelimT& delimiter);
    ostream_joiner(ostream_type& s, DelimT&& delimiter);
    template<typename T>
```

```
    ostream_joiner& operator=(const T& value);
    ostream_joiner& operator*() noexcept;
    ostream_joiner& operator++() noexcept;
    ostream_joiner& operator++(int) noexcept;
  private:
    ostream_type* out_stream; // exposition only
    DelimT delim;             // exposition only
    bool first_element;       // exposition only
  };
} // inline namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

10.2.1 `ostream_joiner` constructor [iterator.ostream.joiner.cons]

1 `ostream_joiner(ostream_type& s, const DelimT& delimiter);`

2 *Effects:* Initializes `out_stream` with `std::addressof(s)`, `delim` with `delimiter`, and `first_element` with `true`.

3 `ostream_joiner(ostream_type& s, DelimT&& delimiter);`

4 *Effects:* Initializes `out_stream` with `std::addressof(s)`, `delim` with `move(delimiter)`, and `first_element` with `true`.

10.2.2 `ostream_joiner` operations [iterator.ostream.joiner.ops]

1
```
template<typename T>
  ostream_joiner& operator=(const T& value);
```

2 *Effects:*

```
if (!first_element)
  *out_stream << delim;
first_element = false;
*out_stream << value;
return *this;
```

3 `ostream_joiner& operator*() noexcept;`

4 *Returns:* `*this`.

5
```
ostream_joiner& operator++() noexcept;
ostream_joiner& operator++(int) noexcept;
```

6 *Returns:* `*this`.

10.2.3 `ostream_joiner` creation function [iterator.ostream.joiner.creation]

1
```
template <class charT, class traits, class DelimT>
  ostream_joiner<decay_t<DelimT>, charT, traits>
  make_ostream_joiner(basic_ostream<charT, traits>& os, DelimT&& delimiter);
```

2 *Returns:* `ostream_joiner<decay_t<DelimT>, charT, traits>(os, forward<DelimT>(delimiter));`

11 Futures [futures]

11.1 Header <experimental/future> synopsis [header.future.synop]

```
#include <future>

namespace std {
  namespace experimental {
  inline namespace fundamentals_v2 {

    template <class R> class promise;
    template <class R> class promise<R&>;
    template <> class promise<void>;

    template <class R>
    void swap(promise<R>& x, promise<R>& y) noexcept;

    template <class> class packaged_task; // undefined
    template <class R, class... ArgTypes>
    class packaged_task<R(ArgTypes...)>;

    template <class R, class... ArgTypes>
    void swap(packaged_task<R(ArgTypes...)>&, packaged_task<R(ArgTypes...)>&) noexcept;

  } // namespace fundamentals_v2
  } // namespace experimental

  template <class R, class Alloc>
  struct uses_allocator<experimental::promise<R>, Alloc>;

  template <class R, class Alloc>
  struct uses_allocator<experimental::packaged_task<R>, Alloc>;

} // namespace std
```

11.2 Class template `promise` [futures.promise]

1 The specification of all declarations within this sub-clause 11.2 and its sub-clauses are the same as the corresponding declarations, as specified in C++14 §30.6.5, unless explicitly specified otherwise.

```
namespace std {
  namespace experimental {
  inline namespace fundamentals_v2 {

    template <class R>
    class promise {
    public:
      using allocator_type = erased_type;

      promise();
```



```
      template <class Allocator>
      promise(allocator_arg_t, const Allocator& a);
      promise(promise&& rhs) noexcept;
      promise(const promise& rhs) = delete;
      ~promise();

      promise& operator=(promise&& rhs) noexcept;
      promise& operator=(const promise& rhs) = delete;
      void swap(promise& other) noexcept;

      future<R> get_future();

      void set_value(see below);
      void set_exception(exception_ptr p);

      void set_value_at_thread_exit(const R& r);
      void set_value_at_thread_exit(see below);
      void set_exception_at_thread_exit(exception_ptr p);

      pmr::memory_resource* get_memory_resource() const noexcept;
    };

    template <class R>
    void swap(promise<R>& x, promise<R>& y) noexcept;

  } // namespace fundamentals_v2
  } // namespace experimental

  template <class R, class Alloc>
  struct uses_allocator<experimental::promise<R>, Alloc>;

} // namespace std
```

2 When a `promise` constructor that takes a first argument of type `allocator_arg_t` is invoked, the second argument is treated as a type-erased allocator (8.3).

11.3 Class template `packaged_task` [futures.task]

1 The specification of all declarations within this sub-clause 11.3 and its sub-clauses are the same as the corresponding declarations, as specified in C++14 §30.6.9, unless explicitly specified otherwise.

```
namespace std {
  namespace experimental {
  inline namespace fundamentals_v2 {

    template <class R, class... ArgTypes>
    class packaged_task<R(ArgTypes...)> {
    public:
      using allocator_type = erased_type;

      packaged_task() noexcept;
      template <class F>
      explicit packaged_task(F&& f);
```

```
      template <class F, class Allocator>
      explicit packaged_task(allocator_arg_t, const Allocator& a, F&& f);
      ~packaged_task();

      packaged_task(const packaged_task&) = delete;
      packaged_task& operator=(const packaged_task&) = delete;

      packaged_task(packaged_task&& rhs) noexcept;
      packaged_task& operator=(packaged_task&& rhs) noexcept;
      void swap(packaged_task& other) noexcept;

      bool valid() const noexcept;

      future<R> get_future();

      void operator()(ArgTypes... );
      void make_ready_at_thread_exit(ArgTypes...);

      void reset();

      pmr::memory_resource* get_memory_resource() const noexcept;
    };

    template <class R, class... ArgTypes>
    void swap(packaged_task<R(ArgTypes...)>&, packaged_task<R(ArgTypes...)>&) noexcept;

  } // namespace fundamentals_v2
  } // namespace experimental

  template <class R, class Alloc>
  struct uses_allocator<experimental::packaged_task<R>, Alloc>;

} // namespace std
```

2 When a `packaged_task` constructor that takes a first argument of type `allocator_arg_t` is invoked, the second argument is treated as a type-erased allocator (8.3).

12 Algorithms library [algorithms]

12.1 Header `<experimental/algorithm>` synopsis [header.algorithm.synop]

```
#include <algorithm>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  // 12.2, Search
  template<class ForwardIterator, class Searcher>
  ForwardIterator search(ForwardIterator first, ForwardIterator last,
                         const Searcher& searcher);

  // 12.3, Sampling
  template<class PopulationIterator, class SampleIterator, class Distance>
  SampleIterator sample(PopulationIterator first, PopulationIterator last,
                        SampleIterator out, Distance n);

  template<class PopulationIterator, class SampleIterator,
           class Distance, class UniformRandomNumberGenerator>
  SampleIterator sample(PopulationIterator first, PopulationIterator last,
                        SampleIterator out, Distance n,
                        UniformRandomNumberGenerator&& g);

  // 12.4, Shuffle
  template<class RandomAccessIterator>
  void shuffle(RandomAccessIterator first, RandomAccessIterator last);

} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

12.2 Search [alg.search]

1
```
template<class ForwardIterator, class Searcher>
  ForwardIterator search(ForwardIterator first, ForwardIterator last,
                         const Searcher& searcher);
```

2 *Effects:* Equivalent to `return searcher(first, last).first;`

3 *Remarks:* `Searcher` need not meet the `CopyConstructible` requirements.

12.3 Sampling [alg.random.sample]

1
```
template<class PopulationIterator, class SampleIterator, class Distance>
SampleIterator sample(PopulationIterator first, PopulationIterator last,
                      SampleIterator out, Distance n);

template<class PopulationIterator, class SampleIterator,
         class Distance, class UniformRandomNumberGenerator>
SampleIterator sample(PopulationIterator first, PopulationIterator last,
                      SampleIterator out, Distance n,
                      UniformRandomNumberGenerator&& g);
```

2 *Requires:*

— `PopulationIterator` shall meet the requirements of an `InputIterator` type.
— `SampleIterator` shall meet the requirements of an `OutputIterator` type.
— `SampleIterator` shall meet the additional requirements of a `RandomAccessIterator` type unless `PopulationIterator` meets the additional requirements of a `ForwardIterator` type.
— `PopulationIterator`'s value type shall be writable to `out`.
— `Distance` shall be an integer type.
— `UniformRandomNumberGenerator` shall meet the requirements of a uniform random number generator type (C++14 §26.5.1.3) whose return type is convertible to `Distance`.
— `out` shall not be in the range [`first`, `last`).

3 *Effects:* Copies `min(last-first, n)` elements (the *sample*) from [`first`, `last`) (the *population*) to `out` such that each possible sample has equal probability of appearance. [*Note:* Algorithms that obtain such effects include *selection sampling* and *reservoir sampling*. — *end note*]

4 *Returns:* The end of the resulting sample range.

5 *Complexity:* O(`last - first`).

6 *Remarks:*

— Stable if and only if `PopulationIterator` meets the requirements of a `ForwardIterator` type.
— If `g` is not given in the argument list, it denotes the per-thread engine (13.2.2.1). To the extent that the implementation of this function makes use of random numbers, the object `g` shall serve as the implementation's source of randomness.

12.4 Shuffle [alg.random.shuffle]

1
```
template<class RandomAccessIterator>
     void shuffle(RandomAccessIterator first, RandomAccessIterator last);
```

2 *Effects:* Permutes the elements in the range `[first,last)` such that each possible permutation of those elements has equal probability of appearance.

3 *Requires:* `RandomAccessIterator` shall satisfy the requirements of `ValueSwappable` (C++14 §17.6.3.2).

4 *Complexity:* Exactly `(last - first) - 1` swaps.

5 *Remarks:* To the extent that the implementation of this function makes use of random numbers, the per-thread engine (13.2.2.1) shall serve as the implementation's source of randomness.

13 Numerics library [numeric]

13.1 Generalized numeric operations [numeric.ops]

13.1.1 Header <experimental/numeric> synopsis [numeric.ops.overview]

```
#include <numeric>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  // 13.1.2, Greatest common divisor
  template<class M, class N>
  constexpr common_type_t<M,N> gcd(M m, N n);

  // 13.1.3, Least common multiple
  template<class M, class N>
  constexpr common_type_t<M,N> lcm(M m, N n);

} // inline namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

13.1.2 Greatest common divisor [numeric.ops.gcd]

1
```
template<class M, class N>
  constexpr common_type_t<M,N> gcd(M m, N n);
```

2 *Requires:* `|m|` and `|n|` shall be representable as values of `common_type_t<M, N>`. [*Note:* These requirements ensure, for example, that `gcd(m, m) = |m|` is representable as a value of type `M`. — *end note*]

3 *Remarks:* If either `M` or `N` is not an integer type, or if either is (possibly *cv*-qualified) `bool`, the program is ill-formed.

4 *Returns:* zero when `m` and `n` are both zero. Otherwise, returns the greatest common divisor of `|m|` and `|n|`.

5 *Throws:* Nothing.

13.1.3 Least common multiple [numeric.ops.lcm]

1
```
template<class M, class N>
  constexpr common_type_t<M,N> lcm(M m, N n);
```

2 *Requires:* `|m|` and `|n|` shall be representable as values of `common_type_t<M, N>`. The least common multiple of `|m|` and `|n|` shall be representable as a value of type `common_type_t<M,N>`.

3 *Remarks:* If either `M` or `N` is not an integer type, or if either is (possibly *cv*-qualified) `bool`, the program is ill-formed.

4 *Returns:* zero when either `m` or `n` is zero. Otherwise, returns the least common multiple of `|m|` and `|n|`.

5 *Throws:* Nothing.

13.2 Random number generation [rand]

13.2.1 Header `<experimental/random>` synopsis [rand.synopsis]

```
#include <random>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  // 13.2.2.1, Function template randint
  template <class IntType>
  IntType randint(IntType a, IntType b);
  void reseed();
  void reseed(default_random_engine::result_type value);

} // inline namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

13.2.2 Utilities [rand.util]

13.2.2.1 Function template `randint` [rand.util.randint]

1 A separate *per-thread engine* of type `default_random_engine` (C++14 §26.5.5), initialized to an unpredictable state, shall be maintained for each thread.

2
```
template<class IntType>
  IntType randint(IntType a, IntType b);
```

3 *Requires:* `a` ≤ `b`.

4 *Remarks:* If the template argument does not meet the requirements for `IntType` (C++14 §26.5.1.1), the program is ill-formed.

5 *Returns:* A random integer *i*, `a` ≤ *i* ≤ `b`, produced from a thread-local instance of `uniform_int_distribution<IntType>` (C++14 §26.5.8.2.1) invoked with the per-thread engine.

6
```
void reseed();
void reseed(default_random_engine::result_type value);
```

7 *Effects:* Let `g` be the per-thread engine. The first form sets `g` to an unpredictable state. The second form invokes `g.seed(value)`.

8 *Postconditions:* Subsequent calls to `randint` do not depend on values produced by `g` before calling `reseed`. [*Note:* `reseed` also resets any instances of `uniform_int_distribution` used by `randint`. — *end note*]

14 Reflection library [reflection]

14.1 Class `source_location` [reflection.src_loc]

14.1.1 Header `<experimental/source_location>` synopsis [reflection.src_loc.synop]

```
namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

  struct source_location {
    // 14.1.2, source_location creation
    static constexpr source_location current() noexcept;

    constexpr source_location() noexcept;

    // 14.1.3, source_location field access
    constexpr uint_least32_t line() const noexcept;
    constexpr uint_least32_t column() const noexcept;
    constexpr const char* file_name() const noexcept;
    constexpr const char* function_name() const noexcept;
  };

} // namespace fundamentals_v2
} // namespace experimental
} // namespace std
```

1 [*Note:* The intent of `source_location` is to have a small size and efficient copying. — *end note*]

14.1.2 `source_location` creation [reflection.src_loc.creation]

1 `static constexpr source_location current() noexcept;`

2 *Returns:* When invoked by a function call (C++14 §5.2.2) whose *postfix-expression* is a (possibly parenthesized) *id-expression* naming `current`, returns a `source_location` with an implementation-defined value. The value should be affected by `#line` (C++14 §16.4) in the same manner as for `__LINE__` and `__FILE__`. If invoked in some other way, the value returned is unspecified.

3 *Remarks:* When a *brace-or-equal-initializer* is used to initialize a non-static data member, any calls to `current` should correspond to the location of the constructor or aggregate initialization that initializes the member.

4 [*Note:* When used as a default argument (C++14 §8.3.6), the value of the `source_location` will be the location of the call to `current` at the call site. — *end note*]

[*Example:*

```
struct s {
  source_location member = source_location::current();
  int other_member;
  s(source_location loc = source_location::current())
    : member(loc) // values of member will be from call-site
  {}
  s(int blather) : // values of member should be hereabouts
    other_member(blather) {}
  s(double) // values of member should be hereabouts
  {}
};

void f(source_location a = source_location::current()) {
  source_location b = source_location::current(); // values in b represent this line
}

void g() {
  f(); // f's first argument corresponds to this line of code

  source_location c = source_location::current();
  f(c); // f's first argument gets the same values as c, above
}
```

— *end example*]

5 `constexpr source_location() noexcept;`

6 *Effects:* Constructs an object of class `source_location`.

7 *Remarks:* The values are implementation-defined.

14.1.3 `source_location` field access [reflection.src_loc.fields]

1 `constexpr uint_least32_t line() const noexcept;`

2 *Returns:* The presumed line number (C++14 §16.8) represented by this object.

3 `constexpr uint_least32_t column() const noexcept;`

4 *Returns:* An implementation-defined value representing some offset from the start of the line represented by this object.

5 `constexpr const char* file_name() const noexcept;`

6 *Returns:* The presumed name of the current source file (C++14 §16.8) represented by this object as an NTBS.

7 `constexpr const char* function_name() const noexcept;`

8 *Returns:* If this object represents a position in the body of a function, returns an implementation-defined NTBS that should correspond to the function name. Otherwise, returns an empty string.