ISO/IEC JTC1 SC22 WG21 N4937
Date: 2022-12-15
ISO/IEC PDTS 19568
ISO/IEC JTC1 SC22 WG21
Secretariat: ANSI

Programming Languages — C++ Extensions for Library Fundamentals, Version 3

Langages de programmation —
Extensions C++ pour la bibliothèque fondamentaux, version 3

Document type: Proposed Draft Technical Specification
Document stage: (30) Committee
Document language: E

Contents

1 Scope [general.scope]

1 This technical specification describes extensions to the C++ Standard Library (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.

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 14882:2020, *Programming Languages — C++*

2 ISO/IEC 14882:2020 is herein called the *C++ Standard*. References to clauses within the C++ Standard are written as "C++20 §3.2". The library described in ISO/IEC 14882:2020 clauses 16–32 is herein called the *C++ Standard Library*.

3 Unless otherwise specified, the whole of the C++ Standard's Library introduction (C++20 §16) is included into this Technical Specification by reference.

3 Terms and definitions [general.terms]

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

2 This document does not contain any additional terminological entries.

4 General principles [general]

4.1 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_v3` 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_v3::pmr` because the C++ Standard Library defines `std::pmr`. — *end example*]

2 Each header described in this technical specification shall import the contents of `std::experimental::fundamentals_v3` into `std::experimental` as if by

```
namespace std::experimental::inline fundamentals_v3 {}
```

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_v3::`, 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/memory>`	`<experimental/scope>`
`<experimental/functional>`	`<experimental/memory_resource>`	`<experimental/type_traits>`
`<experimental/future>`	`<experimental/propagate_const>`	`<experimental/utility>`
`<experimental/iterator>`	`<experimental/random>`	

7 [*Note:* This is the last in a series of revisions of this technical specification planned by the C++ committee; while there are no plans to resume the series, any future versions will define their contents in `std::experimental::fundamentals_v4`, `std::experimental::fundamentals_v5`, etc., with the most recent implemented version inlined into `std::experimental`. — *end note*]

4.2 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
N4388	A Proposal to Add a Const-Propagating Wrapper to the Standard Library	6.1	`propagate_const`	201505	`<experimental/propagate_const>`
P0052R10	Generic Scope Guard and RAII Wrapper for the Standard Library	6.2	`scope`	201902	`<experimental/scope>`
N3866	Invocation type traits	6.3.2	`invocation_type`	201406	`<experimental/type_traits>`
N4502	The C++ Detection Idiom	6.3.3	`detect`	201505	`<experimental/type_traits>`
P0987R1	Polymorphic allocator for `std::function`	7.2	`function_polymorphic_allocator`	202211	`<experimental/functional>`
N3916	Polymorphic Memory	8.3	`memory_resources`	201803	`<experimental/memory_resouce>`

Doc. No.	Title	Primary Section	Macro Name Suffix	Value	Header
	Resources				
N4282	The World's Dumbest Smart Pointer	8.2	`observer_ptr`	201411	`<experimental/memory>`
N4257	Delimited iterators	9.2	`ostream_joiner`	201411	`<experimental/iterator>`
N3925	A `sample` Proposal	10.2	`sample`	201402	`<experimental/algorithm>`
N4531	`std::rand` replacement	11.1.2	`randint`	201511	`<experimental/random>`

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

5.1 General [mods.general]

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

5.2 Exception Requirements [mods.exception.requirements]

1 The following changes to the library introduction allow the destructor of `scope_success` to throw exceptions.

16.5.4.8 Other functions [res.on.functions]

1 In certain cases [...]

2 In particular, the effects are undefined in the following cases:

- [...]
- if any replacement function or handler function or destructor operation exits via an exception, unless specifically allowed in the applicable *Required behavior:* or *Throws:* paragraph.
- if an incomplete type (6.9) is used as a template argument when instantiating a template component, unless specifically allowed for that component.

16.5.5.13 Restrictions on exception handling [res.on.exception.handling]

1 [...]

2 Functions from the C standard library shall not throw exceptions[181] except when such a function calls a program-supplied function that throws an exception.[182]

3 Unless otherwise specified, destructor~~Destructor~~ operations defined in the C++ standard library shall not throw exceptions. Every destructor without an exception specification in the C++ standard library shall behave as if it had a non-throwing exception specification.

4 Functions defined in the C++ standard library [...]

6 General utilities library [utilities]

6.1 Constness propagation [propagate_const]

6.1.1 Header <experimental/propagate_const> synopsis [propagate_const.syn]

```
namespace std {
  namespace experimental::inline fundamentals_v3 {

    // 6.1.2.1, Overview
    template <class T> class propagate_const;

    // 6.1.2.9, 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);

  // 6.1.2.10, Specialized algorithms
  template <class T>
    constexpr void swap(propagate_const<T>& pt, propagate_const<T>& pt2) noexcept(see below);

  // 6.1.2.11, 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;

} // namespace experimental::inline fundamentals_v3

// 6.1.2.12, Hash support
template <class T> struct hash;
template <class T>
  struct hash<experimental::fundamentals_v3::propagate_const<T>>;

// 6.1.2.13, Comparison function objects
template <class T> struct equal_to;
template <class T>
  struct equal_to<experimental::fundamentals_v3::propagate_const<T>>;
```

```
  template <class T> struct not_equal_to;
  template <class T>
    struct not_equal_to<experimental::fundamentals_v3::propagate_const<T>>;
  template <class T> struct less;
  template <class T>
    struct less<experimental::fundamentals_v3::propagate_const<T>>;
  template <class T> struct greater;
  template <class T>
    struct greater<experimental::fundamentals_v3::propagate_const<T>>;
  template <class T> struct less_equal;
  template <class T>
    struct less_equal<experimental::fundamentals_v3::propagate_const<T>>;
  template <class T> struct greater_equal;
  template <class T>
    struct greater_equal<experimental::fundamentals_v3::propagate_const<T>>;

} // namespace std
```

6.1.2 Class template `propagate_const` [propagate_const.tmpl]

6.1.2.1 Overview [propagate_const.overview]

```
namespace std::experimental::inline fundamentals_v3 {

  template <class T> class propagate_const {
  public:
    using element_type = remove_reference_t<decltype(*declval<T&>())>;

    // 6.1.2.4, Constructors
    constexpr propagate_const() = default;
    propagate_const(const propagate_const& p) = delete;
    constexpr propagate_const(propagate_const&& p) = default;
    template <class U>
      explicit(!is_convertible_v<U, T>) constexpr propagate_const(propagate_const<U>&& pu);
    template <class U>
      explicit(!is_convertible_v<U, T>) constexpr propagate_const(U&& u);

    // 6.1.2.5, 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);

    // 6.1.2.6, 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;

    // 6.1.2.7, Non-const observers
    constexpr element_type* operator->();
    constexpr operator element_type*(); // Not always defined
    constexpr element_type& operator*();
    constexpr element_type* get();

    // 6.1.2.8, Modifiers
    constexpr void swap(propagate_const& pt) noexcept(is_nothrow_swappable<T>);

  private:
    T t_; //exposition only
  };

} // namespace std::experimental::inline fundamentals_v3
```

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.

6.1.2.2 General requirements on `T` [propagate_const.requirements]

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

2 [*Note:* `propagate_const<const int*>` is well-formed but `propagate_const<int* const>` is not. — *end note*]

6.1.2.3 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 subclause `t` denotes an lvalue of type `T`, `ct` denotes `as_const(t)`.

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 3 — 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`

6.1.2.4 Constructors [propagate_const.ctor]

1
```
template <class U>
    explicit(!is_convertible_v<U, T>) constexpr propagate_const(propagate_const<U>&& pu);
```

2 *Constraints:* `is_constructible_v<T, U>` is true.

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

4
```
template <class U>
    explicit(!is_convertible_v<U, T>) constexpr propagate_const(U&& u);
```

5 *Constraints:* `is_constructible_v<T, U>` is true and `decay_t<U>` is not a specialization of `propagate_const`.

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

6.1.2.5 Assignment [propagate_const.assignment]

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

2 *Constraints:* `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 *Constraints:* `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`.

6.1.2.6 Const observers [propagate_const.const_observers]

1
```
explicit constexpr operator bool() const;
```

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

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

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

5 *Returns:* `get()`.

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

7 *Constraints:* `T` is an object pointer type or has an implicit conversion to `const element_type*`.

8 *Returns:* `get()`.

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

10 *Preconditions:* `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()`.

6.1.2.7 Non-const observers [propagate_const.non_const_observers]

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

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

3 *Returns:* `get()`.

4 `constexpr operator element_type*();`

5 *Constraints:* `T` is an object pointer type or has an implicit conversion to `element_type*`.

6 *Returns:* `get()`.

7 `constexpr element_type& operator*();`

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

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

10 `constexpr element_type* get();`

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

6.1.2.8 Modifiers [propagate_const.modifiers]

1 `constexpr void swap(propagate_const& pt) noexcept(is_nothrow_swappable<T>);`

2 *Preconditions:* Lvalues of type `T` are swappable (C++20 §16.5.3.2).

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

6.1.2.9 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_`.

6.1.2.10 Specialized algorithms [propagate_const.algorithms]

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

2 *Constraints:* `is_swappable_v<T>` is `true`.

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

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

```
noexcept(pt1.swap(pt2))
```

6.1.2.11 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.

6.1.2.12 Hash support [propagate_const.hash]

1
```
template <class T>
  struct hash<experimental::fundamentals_v3::propagate_const<T>>;
```

2 The specialization `hash<experimental::fundamentals_v3::propagate_const<T>>` is enabled (C++20 §20.14.18) if and only if `hash<T>` is enabled. When enabled, for an object `p` of type `propagate_const<T>`, `hash<experimental::fundamentals_v3::propagate_const<T>>()(p)` evaluates to the same value as `hash<T>()(p.t_)`.

6.1.2.13 Comparison function objects [propagate_const.comparison_function_objects]

1
```
template <class T>
  struct equal_to<experimental::fundamentals_v3::propagate_const<T>>;
```

2 For objects `p, q` of type `propagate_const<T>`, `equal_to<experimental::fundamentals_v3::propagate_const<T>>()(p, q)` shall evaluate to the same value as `equal_to<T>()(p.t_, q.t_)`.

3 *Mandates:* The specialization `equal_to<T>` is well-formed.

4 *Preconditions:* The specialization `equal_to<T>` is well-defined.

5 `template <class T>`
`struct not_equal_to<experimental::fundamentals_v3::propagate_const<T>>;`

6 For objects `p, q` of type `propagate_const<T>`, `not_equal_to<experimental::fundamentals_v3::propagate_const<T>>()(p, q)` shall evaluate to the same value as `not_equal_to<T>()(p.t_, q.t_)`.

7 *Mandates:* The specialization `not_equal_to<T>` is well-formed.

8 *Preconditions:* The specialization `not_equal_to<T>` is well-defined.

9 `template <class T>`
`struct less<experimental::fundamentals_v3::propagate_const<T>>;`

10 For objects `p, q` of type `propagate_const<T>`, `less<experimental::fundamentals_v3::propagate_const<T>>()(p, q)` shall evaluate to the same value as `less<T>()(p.t_, q.t_)`.

11 *Mandates:* The specialization `less<T>` is well-formed.

12 *Preconditions:* The specialization `less<T>` is well-defined.

13 `template <class T>`
`struct greater<experimental::fundamentals_v3::propagate_const<T>>;`

14 For objects `p, q` of type `propagate_const<T>`, `greater<experimental::fundamentals_v3::propagate_const<T>>()(p, q)` shall evaluate to the same value as `greater<T>()(p.t_, q.t_)`.

15 *Mandates:* The specialization `greater<T>` is well-formed.

16 *Preconditions:* The specialization `greater<T>` is well-defined.

17 `template <class T>`
`struct less_equal<experimental::fundamentals_v3::propagate_const<T>>;`

18 For objects `p, q` of type `propagate_const<T>`, `less_equal<experimental::fundamentals_v3::propagate_const<T>>()(p, q)` shall evaluate to the same value as `less_equal<T>()(p.t_, q.t_)`.

19 *Mandates:* The specialization `less_equal<T>` is well-formed.

20 *Preconditions:* The specialization `less_equal<T>` is well-defined.

21 `template <class T>`
`struct greater_equal<experimental::fundamentals_v3::propagate_const<T>>;`

22 For objects `p, q` of type `propagate_const<T>`, `greater_equal<experimental::fundamentals_v3::propagate_const<T>>()(p, q)` shall evaluate to the same value as `greater_equal<T>()(p.t_, q.t_)`.

23 *Mandates:* The specialization `greater_equal<T>` is well-formed.

24 *Preconditions:* The specialization `greater_equal<T>` is well-defined.

6.2 Scope guard support [scopeguard]

6.2.1 Header `<experimental/scope>` synopsis [scope.syn]

```
namespace std::experimental::inline fundamentals_v3 {

  // 6.2.2, Class templates scope_exit, scope_fail, and scope_success
  template <class EF>
    class scope_exit;
  template <class EF>
    class scope_fail;
  template <class EF>
    class scope_success;

  // 6.2.3, Class template unique_resource
  template <class R, class D>
    class unique_resource;

  // 6.2.3.6, unique_resource creation
  template <class R, class D, class S=decay_t<R>>
    unique_resource<decay_t<R>, decay_t<D>>
      make_unique_resource_checked(R&& r, const S& invalid, D&& d) noexcept(see below);

} // namespace std::experimental::inline fundamentals_v3
```

6.2.2 Class templates `scope_exit`, `scope_fail`, and `scope_success` [scopeguard.exit]

1 The class templates `scope_exit`, `scope_fail`, and `scope_success` define scope guards that wrap a function object to be called on their destruction.

2 In this subclause, the placeholder *`scope-guard`* denotes each of these class templates. In descriptions of the class members, *`scope-guard`* refers to the enclosing class.

```
namespace std::experimental::inline fundamentals_v3 {
```

```
  template <class EF> class scope-guard {
  public:
    template <class EFP>
      explicit scope-guard(EFP&& f) noexcept(see below);
    scope-guard(scope-guard&& rhs) noexcept(see below);

    scope-guard(const scope-guard&) = delete;
    scope-guard& operator=(const scope-guard&) = delete;
    scope-guard& operator=(scope-guard&&) = delete;

    ~scope-guard () noexcept(see below);

    void release() noexcept;

  private:
    EF exit_function;                                    // exposition only
    bool execute_on_destruction{true};                   // exposition only
    int uncaught_on_creation{uncaught_exceptions()};     // exposition only
  };

  template <class EF>
    scope-guard(EF) -> scope-guard<EF>;

} // namespace std::experimental::inline fundamentals_v3
```

3 The class template `scope_exit` is a general-purpose scope guard that calls its exit function when a scope is exited. The class templates `scope_fail` and `scope_success` share the `scope_exit` interface, only the situation when the exit function is called differs.

[*Example:*

```
void grow(vector<int>& v) {
  scope_success guard([]{ cout << "Good!" << endl; });
  v.resize(1024);
}
```

— *end example*]

4 [*Note:* If the exit function object of a `scope_success` or `scope_exit` object refers to a local variable of the function where it is defined, e.g., as a lambda capturing the variable by reference, and that variable is used as a return operand in that function, it is possible for that variable to already have been returned when the *scope-guard*'s destructor executes, calling the exit function. This can lead to surprising behavior.
— *end note*]

5 Template argument `EF` shall be a function object type (C++20 §20.14), lvalue reference to function, or lvalue reference to function object type. If `EF` is an object type, it shall meet the *Cpp17Destructible* requirements (C++20 Table 30). Given an lvalue `g` of type `remove_reference_t<EF>`, the expression `g()`

shall be well-formed.

6 The constructor parameter `f` in the following constructors shall be a reference to a function or a reference to a function object (C++20 §20.14).

7
```
template <class EFP>
    explicit scope-guard(EFP&& f) noexcept(
        is_nothrow_constructible_v<EF, EFP> ||
        is_nothrow_constructible_v<EF, EFP&>);
```

8 *Constraints:* `is_same_v<remove_cvref_t<EFP>, scope-guard>` is `false` and `is_constructible_v<EF, EFP>` is `true`.

9 *Mandates:* The expression `f()` is well-formed.

10 *Preconditions:* Calling `f()` has well-defined behavior. For `scope_exit` and `scope_fail`, calling `f()` does not throw an exception.

11 *Effects:* If `EFP` is not an lvalue reference type and `is_nothrow_constructible_v<EF, EFP>` is `true`, initialize `exit_function` with `std::forward<EFP>(f)`; otherwise initialize `exit_function` with `f`. For `scope_exit` and `scope_fail`, if the initialization of `exit_function` throws an exception, calls `f()`. [*Note:* For `scope_success`, `f()` will not be called if the initialization fails. — *end note*]

12 *Throws:* Any exception thrown during the initialization of `exit_function`.

13 *scope-guard*(*scope-guard*&& `rhs`) `noexcept`(*see below*)

14 *Constraints:* `(is_nothrow_move_constructible_v<EF> || is_copy_constructible_v<EF>)` is `true`.

15 *Preconditions:* If `EF` is an object type:

- if `is_nothrow_move_constructible_v<EF>` is `true`, `EF` meets the *Cpp17MoveConstructible* requirements (C++20 Table 26),
- otherwise `EF` meets the *Cpp17CopyConstructible* requirements (C++20 Table 27).

16 *Effects:* If `is_nothrow_move_constructible_v<EF>` is `true`, initializes `exit_function` with `std::forward<EF>(rhs.exit_function)`, otherwise initializes `exit_function` with `rhs.exit_function`. Initializes `execute_on_destruction` from `rhs.execute_on_destruction` and `uncaught_on_creation` from `rhs.uncaught_on_creation`. If construction succeeds, call `rhs.release()`. [*Note:* Copying instead of moving provides the strong exception guarantee. — *end note*]

17 *Postconditions:* `execute_on_destruction` yields the value `rhs.execute_on_destruction` yielded before the construction. `uncaught_on_creation` yields the value `rhs.uncaught_on_creation` yielded before the construction.

18 *Throws:* Any exception thrown during the initialization of `exit_function`.

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

```
is_nothrow_move_constructible_v<EF> || is_nothrow_copy_constructible_v<EF>
```

20 `~scope_exit() noexcept(true);`

21 *Effects:* Equivalent to:

```
if (execute_on_destruction)
  exit_function();
```

22 `~scope_fail() noexcept(true);`

23 *Effects:* Equivalent to:

```
if (execute_on_destruction && uncaught_exceptions() > uncaught_on_creation)
  exit_function();
```

24 `~scope_success() noexcept(noexcept(exit_function()));`

25 *Effects:* Equivalent to:

```
if (execute_on_destruction && uncaught_exceptions() <= uncaught_on_creation)
  exit_function();
```

[*Note:* If `noexcept(exit_function())` is `false`, `exit_function()` may throw an exception, notwithstanding the restrictions of C++20 §16.5.5.13. — *end note*]

26 *Throws:* Any exception thrown by `exit_function()`.

27 `void release() noexcept;`

28 *Effects:* Equivalent to `execute_on_destruction = false`.

6.2.3 Class template `unique_resource` [scopeguard.uniqueres]

6.2.3.1 Overview [scopeguard.uniqueres.overview]

```
namespace std::experimental::inline fundamentals_v3 {

  template <class R, class D> class unique_resource {
  public:
    // 6.2.3.2, Constructors
    unique_resource();
    template <class RR, class DD>
      unique_resource(RR&& r, DD&& d) noexcept(see below);
    unique_resource(unique_resource&& rhs) noexcept(see below);

    // 6.2.3.3, Destructor
    ~unique_resource();

    // 6.2.3.4, Assignment
    unique_resource& operator=(unique_resource&& rhs) noexcept(see below);
```



```
    // 6.2.3.5, Other member functions
    void reset() noexcept;
    template <class RR>
      void reset(RR&& r);
    void release() noexcept;
    const R& get() const noexcept;
    see below operator*() const noexcept;
    R operator->() const noexcept;
    const D& get_deleter() const noexcept;

  private:
    using R1 = conditional_t<is_reference_v<R>, reference_wrapper<remove_reference_t<R>>, R>;  // ex
    R1 resource;                  // exposition only
    D deleter;                    // exposition only
    bool execute_on_reset{true};  // exposition only
  };

  template<class R, class D>
    unique_resource(R, D) -> unique_resource<R, D>;

} // namespace std::experimental::inline fundamentals_v3
```

1 [*Note:* `unique_resource` is a universal RAII wrapper for resource handles. Typically, such resource handles are of trivial type and come with a factory function and a clean-up or deleter function that do not throw exceptions. The clean-up function together with the result of the creation function is used to create a `unique_resource` variable, that on destruction will call the clean-up function. Access to the underlying resource handle is achieved through `get()` and in case of a pointer type resource through a set of convenience pointer operator functions. — *end note*]

2 The template argument `D` shall meet the requirements of a *Cpp17Destructible* (C++20 Table 30) function object type (C++20 §20.14), for which, given a lvalue `d` of type `D` and a lvalue `r` of type `R`, the expression `d(r)` shall be well-formed. `D` shall either meet the *Cpp17CopyConstructible* requirements (C++20 Table 27), or `D` shall meet the *Cpp17MoveConstructible* requirements (C++20 Table 26) and `is_nothrow_move_constructible_v<D>` shall be `true`.

3 For the purpose of this subclause, a resource type `T` is an object type that meets the requirements of *Cpp17CopyConstructible* (C++20 Table 27), or is an object type that meets the requirements of *Cpp17MoveConstructible* (C++20 Table 26) and `is_nothrow_move_constructible_v<T>` is `true`, or is an lvalue reference to a resource type. `R` shall be a resource type.

4 For the scope of the adjacent subclauses, let *RESOURCE* be defined as follows:

- `resource.get()` if `is_reference_v<R>` is `true`,
- `resource` otherwise.

5

6.2.3.2 Constructors [scopeguard.uniqueres.ctor]

1 `unique_resource()`

2 *Constraints:* `is_default_constructible_v<R> && is_default_constructible_v<D>` is `true`.

3 *Effects:* Value-initializes `resource` and `deleter`; `execute_on_reset` is initialized with `false`.

4 `template <class RR, class DD>`
`unique_resource(RR&& r, DD&& d) noexcept(`*see below*`)`

5 *Constraints:*
```
is_constructible_v<R1, RR> &&
is_constructible_v<D , DD> &&
(is_nothrow_constructible_v<R1, RR> || is_constructible_v<R1,RR&>) &&
(is_nothrow_constructible_v<D , DD> || is_constructible_v<D ,DD&>)
```
is `true`. [*Note:* The first two conditions prohibit initialization from an rvalue reference when either `R1` or `D` is a specialization of `reference_wrapper`. — *end note*]

6 *Mandates:* The expressions `d(r)`, `d(`*RESOURCE*`)` and `deleter(`*RESOURCE*`)` are well-formed.

7 *Preconditions:* Calling `d(r)`, `d(`*RESOURCE*`)` or `deleter(`*RESOURCE*`)` has well-defined behavior and does not throw an exception.

8 *Effects:* If `is_nothrow_constructible_v<R1, RR>` is `true`, initializes `resource` with `std::forward<RR>(r)`, otherwise initializes `resource` with `r`. Then, if `is_nothrow_constructible_v<D, DD>` is true, initializes `deleter` with `std::forward<DD>(d)`, otherwise initializes `deleter` with `d`. If initialization of `resource` throws an exception, calls `d(r)`. If initialization of `deleter` throws an exception, calls `d(`*RESOURCE*`)`. [*Note:* The explained mechanism ensures no leaking of resources. — *end note*]

9 *Throws:* Any exception thrown during initialization of `resource` or `deleter`.

10 *Remarks:* The expression inside `noexcept` is equivalent to:
```
(is_nothrow_constructible_v<R1, RR> || is_nothrow_constructible_v<R1, RR&>) &&
(is_nothrow_constructible_v<D , DD> || is_nothrow_constructible_v<D , DD&>)
```


11 `unique_resource(unique_resource&& rhs) noexcept(`*see below*`);`

12 *Effects:* First, initialize `resource` as follows:

- If `is_nothrow_move_constructible_v<R1>` is `true`, from `std::move(rhs.resource)`;
- otherwise, from `rhs.resource`.

[*Note:* If initialization of `resource` throws an exception, `rhs` is left owning the resource and will free it in due time. — *end note*] Then, initialize `deleter` as follows:

- If `is_nothrow_move_constructible_v<D>` is `true`, from `std::move(rhs.deleter)`;
- otherwise, from `rhs.deleter`.

If initialization of `deleter` throws an exception and `is_nothrow_move_constructible_v<R1>` is `true` and `rhs.execute_on_reset` is true:

```
rhs.deleter(RESOURCE);
rhs.release();
```

Finally, `execute_on_reset` is initialized with `exchange(rhs.execute_on_reset, false)`. [*Note:* The explained mechanism ensures no leaking and no double release of resources. — *end note*]

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

```
is_nothrow_move_constructible_v<R1> && is_nothrow_move_constructible_v<D>
```

6.2.3.3 Destructor [scopeguard.uniqueres.dtor]

1 `~unique_resource();`

2 *Effects:* Equivalent to `reset()`.

6.2.3.4 Assignment [scopeguard.uniqueres.assign]

1 `unique_resource& operator=(unique_resource&& rhs) noexcept(`*see below*`);`

2 *Preconditions:* If `is_nothrow_move_assignable_v<R1>` is `true`, `R1` meets the *Cpp17MoveAssignable* (C++20 Table 28) requirements; otherwise `R1` meets the *Cpp17CopyAssignable* (C++20 Table 29) requirements. If `is_nothrow_move_assignable_v<D>` is `true`, `D` meets the *Cpp17MoveAssignable* (C++20 Table 28) requirements; otherwise `D` meets the *Cpp17CopyAssignable* (C++20 Table 29) requirements.

3 *Effects:* Equivalent to:

```
reset();
if constexpr (is_nothrow_move_assignable_v<R1>) {
  if constexpr (is_nothrow_move_assignable_v<D>) {
    resource = std::move(rhs.resource);
    deleter = std::move(rhs.deleter);
  } else {
    deleter = rhs.deleter;
    resource = std::move(rhs.resource);
  }
} else {
  if constexpr (is_nothrow_move_assignable_v<D>) {
    resource = rhs.resource;
    deleter = std::move(rhs.deleter);
  } else {
    resource = rhs.resource;
    deleter = rhs.deleter;
  }
}
execute_on_reset = exchange(rhs.execute_on_reset, false);
```

[*Note:* If a copy of a member throws an exception, this mechanism leaves `rhs` intact and `*this` in the released state. — *end note*]

4 *Returns:* `*this`.

5 *Throws:* Any exception thrown during a copy-assignment of a member that cannot be moved without an exception.

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

```
is_nothrow_move_assignable_v<R1> && is_nothrow_move_assignable_v<D>
```


6.2.3.5 Other member functions [scopeguard.uniqueres.members]

1 `void reset() noexcept;`

2 *Effects:* Equivalent to:

```
if (execute_on_reset) {
  execute_on_reset = false;
  deleter(RESOURCE);
}
```

3 `template <class RR> void reset(RR&& r);`

4 *Constraints:* the selected assignment expression statement assigning `resource` is well-formed.

5 *Mandates:* The expression `deleter(r)` is well-formed.

6 *Preconditions:* Calling `deleter(r)` has well-defined behavior and does not throw an exception.

7 *Effects:* Equivalent to:

```
reset();
if constexpr (is_nothrow_assignable_v<R1&, RR>) {
  resource = std::forward<RR>(r);
} else {
  resource = as_const(r);
}
execute_on_reset = true;
```

If copy-assignment of `resource` throws an exception, calls `deleter(r)`.

8 `void release() noexcept;`

9 *Effects:* Equivalent to `execute_on_reset = false`.

10 `const R& get() const noexcept;`

11 *Returns:* `resource`.

12 *see below* `operator*() const noexcept;`

13 *Constraints:* `is_pointer_v<R>` is `true` and `is_void_v<remove_pointer_t<R>>` is `false`.

14 *Effects:* Equivalent to: `return *get();`

15 *Remarks:* The return type is `add_lvalue_reference_t<remove_pointer_t<R>>`.

16 `R operator->() const noexcept;`

17 *Constraints:* `is_pointer_v<R>` is `true`.

18 *Returns:* `get()`.

19 `const D& get_deleter() const noexcept;`

20 *Returns:* `deleter`.

6.2.3.6 `unique_resource` creation [scopeguard.uniqueres.create]

1
```
template <class R, class D, class S=decay_t<R>>
    unique_resource<decay_t<R>, decay_t<D>>
      make_unique_resource_checked(R&& resource, const S& invalid, D&& d)
      noexcept(is_nothrow_constructible_v<decay_t<R>, R> &&
               is_nothrow_constructible_v<decay_t<D>, D>);
```

2 *Mandates:* The expression `(resource == invalid ? true : false)` is well-formed.

3 *Preconditions:* Evaluation of the expression `(resource == invalid ? true : false)` has well-defined behavior and does not throw an exception.

4 *Effects:* Returns an object constructed with members initialized from `std::forward<R>(resource)`, `std::forward<D>(d)`, and `!bool(resource == invalid)`. Any failure during construction of the return value will not call `d(resource)` if `bool(resource == invalid)` is `true`.

5 [*Note:* This creation function exists to avoid calling a deleter function with an invalid argument. — *end note*]

[*Example:* The following example shows its use to avoid calling `fclose` when `fopen` fails.

```
auto file = make_unique_resource_checked(
    ::fopen("potentially_nonexistent_file.txt", "r"),
    nullptr,
    [](auto fptr){ ::fclose(fptr); });
```

— *end example*]

6.3 Metaprogramming and type traits [meta]

6.3.1 Header <experimental/type_traits> synopsis [meta.type.syn]

```
#include <type_traits>

namespace std::experimental::inline fundamentals_v3 {
```



```
// 6.3.2, 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;

// 6.3.3, Detection idiom
struct nonesuch;

template <template<class...> class Op, class... Args>
  using is_detected = see below;
template <template<class...> class Op, class... Args>
  inline 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>
  inline 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>
  inline constexpr bool is_detected_convertible_v
    = is_detected_convertible<To, Op, Args...>::value;

} // namespace std::experimental::inline fundamentals_v3
```

6.3.2 Other type transformations [meta.trans.other]

1 This subclause 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++20 §20.15.1).

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++20 §12.4.3) 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++20 §7.6.1.2) 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 4 — 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++20 §7), 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`.

6.3.3 Detection idiom [meta.detect]

```
struct nonesuch {
  ~nonesuch() = delete;
  nonesuch(nonesuch const&) = delete;
  void operator=(nonesuch const&) = delete;
};
```

1 `nonesuch` has no default constructor (C++20 §11.4.4) or initializer-list constructor (C++20 §9.4.4), and is not an aggregate (C++20 §9.4.1).

```
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*]

7 Function objects [func]

7.1 Header <experimental/functional> synopsis [functional.syn]

```
#include <functional>

namespace std {
  namespace experimental::inline fundamentals_v3 {

    // 7.2, Class template function
    template<class> class function; // not defined
    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;

  } // namespace experimental::inline fundamentals_v3
} // namespace std
```

7.2 Class template function [func.wrap.func]

7.2.1 Overview [func.wrap.func.overview]

1 The specification of all declarations within subclause 7.2 are the same as the corresponding declarations, as specified in C++20 §20.14.16.2, 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 fundamentals_v3 {

    template<class> class function; // undefined

    template<class R, class... ArgTypes>
    class function<R(ArgTypes...)> {
    public:
      using result_type = R;

      using allocator_type = std::pmr::polymorphic_allocator<>;
```

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

    allocator_type get_allocator() const noexcept;
  };

} // namespace experimental::inline fundamentals_v3
} // namespace std
```

7.2.2 Construct/copy/destroy [func.wrap.func.con]

1 A function object stores an allocator object of type `std::pmr::polymorphic_allocator<>`, which it uses to allocate memory for its internal data structures. In the `function` constructors, the allocator is initialized (before the target object, if any) as follows:

— For the move constructor, the allocator is initialized from `f.get_allocator()`, where `f` is the

parameter of the constructor.
— For constructors having a first parameter of type `allocator_arg_t`, the allocator is initialized from the second parameter.
— For all other constructors, the allocator is value-initialized.

2 In all cases, the allocator of a parameter having type `function&&` is unchanged. If the constructor creates a target object, that target object is initialized by uses-allocator construction with the allocator and other target object constructor arguments. [*Note:* If a constructor parameter of type `experimental::function&&` has an allocator equal to that of the object being constructed, the implementation can often transfer ownership of the target rather than constructing a new one. — *end note*]

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

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

5 *Returns:* `*this`.

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

7 *Effects:* `function(allocator_arg, get_allocator(), 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)`. [*Note:* The stored allocator is unchanged. — *end note*]

12 *Returns:* `*this`.

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

14 *Constraints:* `declval<decay_t<F>&>()` is *Lvalue-Callable* (C++20 §20.14.16.2) for argument types `ArgTypes...` and return type `R`.

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

16 *Returns:* `*this`.

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

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

19 *Returns:* `*this`.

7.2.3 Modifiers [func.wrap.func.mod]

1 `void swap(function& other);`

2 *Preconditions:* `this->get_allocator() == other.get_allocator()`.

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

4 *Throws:* Nothing.

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

7.2.4 Observers [func.wrap.func.obs]

1 `allocator_type get_allocator() const noexcept;`

2 *Returns:* A copy of the allocator initialized during construction (7.2.2) of this object.

8 Memory [memory]

8.1 Header <experimental/memory> synopsis [memory.syn]

```
#include <memory>

namespace std {
  namespace experimental::inline fundamentals_v3 {

    // 8.2, Non-owning (observer) pointers
    template <class W> class observer_ptr;

    // 8.2.6, 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>);
```

```
  } // namespace experimental::inline fundamentals_v3

  // 8.2.7, observer_ptr hash support
  template <class T> struct hash;
  template <class T> struct hash<experimental::observer_ptr<T>>;

} // namespace std
```

8.2 Non-owning (observer) pointers [memory.observer.ptr]

8.2.1 Class template observer_ptr overview [memory.observer.ptr.overview]

```
namespace std::experimental::inline fundamentals_v3 {

  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;

    // 8.2.2, observer_ptr constructors
    // default constructor
    constexpr observer_ptr() noexcept;

    // pointer-accepting constructors
    constexpr observer_ptr(nullptr_t) noexcept;
    constexpr explicit observer_ptr(pointer) noexcept;

    // copying constructors (in addition to the implicit copy constructor)
    template <class W2> constexpr observer_ptr(observer_ptr<W2>) noexcept;

    // 8.2.3, observer_ptr observers
    constexpr pointer get() const noexcept;
    constexpr reference operator*() const;
    constexpr pointer operator->() const noexcept;
    constexpr explicit operator bool() const noexcept;

    // 8.2.4, observer_ptr conversions
    constexpr explicit operator pointer() const noexcept;

    // 8.2.5, observer_ptr modifiers
```

```
    constexpr pointer release() noexcept;
    constexpr void reset(pointer = nullptr) noexcept;
    constexpr void swap(observer_ptr&) noexcept;
  }; // observer_ptr<>

} // namespace std::experimental::inline fundamentals_v3
```

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 *Cpp17CopyConstructible* and *Cpp17CopyAssignable* 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*]

8.2.2 `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 *Constraints:* `W2*` is convertible to `W*`.

8 *Postconditions:* `get() == other.get()`.

8.2.3 `observer_ptr` observers [memory.observer.ptr.obs]

1
```
constexpr pointer get() const noexcept;
```

2 *Returns:* The stored pointer.

3 `constexpr reference operator*() const;`

4 *Preconditions:* `get() != nullptr` is `true`.

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`.

8.2.4 `observer_ptr` conversions [memory.observer.ptr.conv]

1 `constexpr explicit operator pointer() const noexcept;`

2 *Returns:* `get()`.

8.2.5 `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`.

8.2.6 `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++20 §7) 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)`.

8.2.7 `observer_ptr` hash support [memory.observer.ptr.hash]

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

2 The specialization is enabled (C++20 §20.14.18). For an object `p` of type `observer_ptr<T>`, `hash<observer_ptr<T>>()(p)` evaluates to the same value as `hash<T*>()(p.get())`.

8.3 Header `<experimental/memory_resource>` synopsis [memory.resource.syn]

```
namespace std::pmr::experimental::inline fundamentals_v3 {

  // 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>>;

} // namespace std::pmr::experimental::inline fundamentals_v3
```

8.4 Alias template `resource_adaptor` [memory.resource.adaptor]

8.4.1 `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 *Cpp17Allocator* requirements (C++20 §16.5.3.5), 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>>;
```

8.4.2 *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)`.

8.4.3 *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` (C++20 §20.12.2).

3 `void do_deallocate(void* p, size_t bytes, size_t alignment);`

4 *Preconditions:* `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`.

9 Iterators library [iterator]

9.1 Header <experimental/iterator> synopsis [iterator.syn]

```
#include <iterator>

namespace std::experimental::inline fundamentals_v3 {

  // 9.2, 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);

} // namespace std::experimental::inline fundamentals_v3
```

9.2 Class template ostream_joiner [iterator.ostream.joiner]

9.2.1 Overview [iterator.ostream.joiner.overview]

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 Ts 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::experimental::inline fundamentals_v3 {

  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
  };

} // namespace std::experimental::inline fundamentals_v3
```

9.2.2 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`.

9.2.3 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`.

9.2.4 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));
```

10 Algorithms library [algorithms]

10.1 Header <experimental/algorithm> synopsis [algorithm.syn]

```
#include <algorithm>

namespace std::experimental::inline fundamentals_v3 {

  // 10.2, Sampling
  template<class PopulationIterator, class SampleIterator, class Distance>
  SampleIterator sample(PopulationIterator first, PopulationIterator last,
                        SampleIterator out, Distance n);

  // 10.3, Shuffle
  template<class RandomAccessIterator>
  void shuffle(RandomAccessIterator first, RandomAccessIterator last);

} // namespace std::experimental::inline fundamentals_v3
```

10.2 Sampling [alg.random.sample]

1
```
template<class PopulationIterator, class SampleIterator, class Distance>
SampleIterator sample(PopulationIterator first, PopulationIterator last,
                      SampleIterator out, Distance n);
```

2 *Effects:* Equivalent to:

```
return ::std::sample(first, last, out, n, g);
```

where g denotes the per-thread engine (11.1.2). To the extent that the implementation of this function makes use of random numbers, the object g serves as the implementation's source of randomness.

10.3 Shuffle [alg.random.shuffle]

1
```
template<class RandomAccessIterator>
    void shuffle(RandomAccessIterator first, RandomAccessIterator last);
```

2 *Preconditions:* `RandomAccessIterator` meets the *Cpp17ValueSwappable* requirements (C++20 §16.5.3.2).

3 *Effects:* Permutes the elements in the range `[first,last)` such that each possible permutation of those elements has equal probability of appearance.

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 (11.1.2) serves as the implementation's source of randomness.

11 Numerics library [numeric]

11.1 Random number generation [rand]

11.1.1 Header <experimental/random> synopsis [rand.syn]

```
#include <random>

namespace std::experimental::inline fundamentals_v3 {

  // 11.1.2, Function template randint
  template <class IntType>
  IntType randint(IntType a, IntType b);
  void reseed();
  void reseed(default_random_engine::result_type value);

} // namespace std::experimental::inline fundamentals_v3
```

11.1.2 Function template randint [rand.randint]

1 A separate *per-thread engine* of type `default_random_engine` (C++20 §26.6.5), initialized to an unpredictable state, shall be maintained for each thread.

2
```
template<class IntType>
    IntType randint(IntType a, IntType b);
```

3 *Mandates:* The template argument meets the requirements for a template parameter named `IntType` in C++20 §26.6.2.1.

4 *Preconditions:* `a` $\leq$ `b`.

5 *Returns:* A random integer i, `a` $\leq i \leq$ `b`, produced from a thread-local instance of `uniform_int_distribution<IntType>` (C++20 §26.6.8.2.1) 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*]