Technical Specification

ISO/IEC TS 19568

Programming Languages — C++ Extensions for Library Fundamentals

Langages de programmation — Extensions C++ pour la bibliothèque fondamentaux

Third edition
2024-08

Reference number
ISO/IEC TS 19568:2024(en)

Contents

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work.

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of document should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives or www.iec.ch/members_experts/refdocs).

ISO and IEC draw attention to the possibility that the implementation of this document may involve the use of (a) patent(s). ISO and IEC take no position concerning the evidence, validity or applicability of any claimed patent rights in respect thereof. As of the date of publication of this document, ISO and IEC had not received notice of (a) patent(s) which may be required to implement this document. However, implementers are cautioned that this may not represent the latest information, which may be obtained from the patent database available at www.iso.org/patents and https://patents.iec.ch. ISO and IEC shall not be held responsible for identifying any or all such patent rights.

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT) see www.iso.org/iso/foreword.html. In the IEC, see www.iec.ch/understanding-standards.

This document was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

This third edition cancels and replaces the second edition (ISO/IEC TS 19568:2017), which has been technically revised.

The main changes are as follows:

- The document now refers to the C++ language as defined in ISO/IEC 14882:2020; the previous edition referred to ISO/IEC 14882:2017.
- Removal of features that have been added to ISO/IEC 14882: tuple utilities, logical

operator traits, rational arithmetic, time utilities, error support, searchers, `not_fn`, `optional`, `any`, `string_view`, shared-ownership pointers, `memory_resource`, search algorithm, numeric operations (gcd/lcm), `source_location`.

- New feature: scope guard class templates for guard types that perform automatic actions on scope exit.
- Feature modification: type-erasing classes now use `polymorphic_allocator<>`.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html and www.iec.ch/national-committees.

Introduction [introduction]

In this document, the phrase *C++ Standard Library* refers to the library described in ISO/IEC 14882:2020, clauses 16–32.

Clauses and subclauses in this document are annotated with a so-called stable name, presented in square brackets next to the (sub)clause heading (such as "[introduction]" for this clause). Stable names aid in the discussion and evolution of this document by serving as stable references to subclauses across editions that are unaffected by changes of subclause numbering.

In addition to the main font for the document body, this document uses `upright monospace font` to display C++ source code, some of which forms part of the normative specification verbatim, *`italic monospace font`* for placeholders within source code that necessary for the specification, but whose spelling is not significant, and *ItalicSerifCamelCase* for certain concepts (comprising syntactic and semantic constraints) from the C++ Standard Library.

Programming languages — C++ Extensions for Library Fundamentals

1 Scope [general.scope]

This document 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.

It is intended that some of the library components be considered for standardization in a future version of C++. At present, they are not part of any C++ standard.

The goal of this document 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]

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements 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++*

3 Terms and definitions [general.terms]

For the purposes of this document, the terms and definitions given in ISO/IEC 14882:2020 apply.

ISO and IEC maintain terminology databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at https://www.electropedia.org/

4 General principles [general]

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

Since the extensions described in this document 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 document 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 document does not define `std::experimental::fundamentals_v3::pmr` because the C++ Standard Library defines `std::pmr`.

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

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

This document also describes some experimental modifications to existing interfaces in the C++ Standard Library.

Unless otherwise specified, references to other entities described in this document 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::`.

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>
```

New headers are also provided in the `<experimental/>` directory, but without such an `#include`. Table 1 lists the headers that are specified by this document.

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>`	

NOTE This is the last in a series of revisions of this document 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`.

4.2 Feature-testing recommendations [general.feature.test]

For the sake of improved portability between partial implementations of various C++ standards, implementers and programmers are recommended to follow the guidelines in this subclause concerning feature-test macros.

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 document that they have implemented. The recommended macro name is "`__cpp_lib_experimental_`" followed by the string in the "Macro Name Suffix" column. Table 2 lists the headers and recommended feature-test macros for the features specified in this document.

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 document

Feature	Primary Subclause	Macro Name Suffix	Value	Header
Const-propagating wrapper	6.1	`propagate_const`	201505	`<experimental/propagate_const>`
Generic scope guard and RAII wrapper	6.2	`scope`	201902	`<experimental/scope>`
Invocation type traits	6.3.2	`invocation_type`	201406	`<experimental/type_traits>`
Detection metaprograms	6.3.3	`detect`	201505	`<experimental/type_traits>`

Feature	Primary Subclause	Macro Name Suffix	Value	Header
Polymorphic allocator for `std::function`	7.2	`function_polymor-phic_allocator`	202211	`<experimental/functional>`
Polymorphic memory resources	8.3	`memory_resources`	201803	`<experimental/memory_resouce>`
Non-owning pointer wrapper	8.2	`observer_ptr`	201411	`<experimental/memory>`
Delimited iterators	9.2	`ostream_joiner`	201411	`<experimental/iterator>`
Random sampling	10.2	`sample`	201402	`<experimental/algorithm>`
Replacement for `std::rand`	11.1.2	`randint`	201511	`<experimental/random>`

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

5.1 General [mods.general]

Implementations that conform to this document shall behave as if the modifications contained in this clause are made to ISO/IEC 14882:2020.

References to clauses within ISO/IEC 14882:2020 are written as "C++20, §3.2".

Unless otherwise specified, the whole of the Library introduction of ISO/IEC 14882:2020 (C++20, §16) is included into this document by reference.

5.2 Exception requirements [mods.exception.requirements]

The following modifications to the Library introduction (C++20, §16) allow the destructor of `scope_success` (6.2) to throw exceptions.

The requirements of C++20, §16.4.5.8 shall apply with the following modification: An applicable *Throws:* paragraph (in addition to any applicable *Required behavior:* paragraph) can allow a replacement function or handler function or destructor operation to exit via an exception.

The requirements of C++20, §16.4.6.13 shall apply with the following modifications: Destructor operations defined in the C++ standard library are permitted to throw exceptions if this is explicitly specified. Only those destructors in the C++ standard library that do not have an exception specification shall behave as if they had a non-throwing exception specification.

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
```

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]

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.

NOTE propagate_const<const int*> is well-formed but propagate_const<int* const> is not.

6.1.2.3 Requirements on class type T [propagate_const.class_type_requirements]

If T is class type then it shall satisfy the requirements described in this subclause and in Table 3. In this subclause t denotes an lvalue of type T, ct denotes as_const(t).

T and const T shall be contextually convertible to bool.

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

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]

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

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

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

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

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

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]

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

Constraints: `U` is implicitly convertible to `T`.

Effects: `t_ = std::move(pu.t_)`.

Returns: `*this`.

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

Constraints: `U` is implicitly convertible to `T` and `decay_t<U>` is not a specialization of `propagate_const`.

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

Returns: `*this`.

6.1.2.6 Const observers [propagate_const.const_observers]

`explicit constexpr operator bool() const;`

Returns: `(bool)t_`.

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

Preconditions: `get() != nullptr`.

Returns: `get()`.

`constexpr operator const element_type*() const;`

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

Returns: `get()`.

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

Preconditions: `get() != nullptr`.

Returns: `*get()`.

`constexpr const element_type* get() const;`

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

6.1.2.7 Non-const observers [propagate_const.non_const_observers]

`constexpr element_type* operator->();`

Preconditions: `get() != nullptr`.

Returns: `get()`.

`constexpr operator element_type*();`

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

Returns: `get()`.

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

Preconditions: `get() != nullptr`.

Returns: `*get()`.

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

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

6.1.2.8 Modifiers [propagate_const.modifiers]

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

Preconditions: Lvalues of type `T` are swappable (C++20, §16.4.4.3).

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

6.1.2.9 Relational operators [propagate_const.relational]

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

Returns: `pt.t_ == nullptr`.

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

Returns: `nullptr == pt.t_`.

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

Returns: `pt.t_ != nullptr`.

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

Returns: `nullptr != pt.t_`.

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

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

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

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

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

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

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

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

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

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

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

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

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

Returns: `pt.t_ == u`.

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

Returns: `pt.t_ != u`.

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

Returns: `pt.t_ < u`.

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

Returns: `pt.t_ > u`.

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

Returns: `pt.t_ <= u`.

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

Returns: `pt.t_ >= u`.

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

Returns: `t == pu.t_`.

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

Returns: `t != pu.t_`.

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

Returns: `t < pu.t_`.

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

Returns: `t > pu.t_`.

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

Returns: `t <= pu.t_`.

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

Returns: `t >= pu.t_`.

6.1.2.10 Specialized algorithms [propagate_const.algorithms]

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

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

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

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

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

6.1.2.11 Underlying pointer access [propagate_const.underlying]

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.

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

Returns: a reference to the underlying object pointer type.

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

Returns: a reference to the underlying object pointer type.

6.1.2.12 Hash support [propagate_const.hash]

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

The specialization `hash<experimental::fundamentals_v3::propagate_const<T>>` is enabled (C++20, §20.14.19) 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]

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

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_)`.

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

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

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

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_)`.

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

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

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

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_)`.

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

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

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

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_)`.

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

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

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

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_)`.

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

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

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

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_)`.

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

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]

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

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
```

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);
}
```

NOTE 1 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.

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.

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).

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

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

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

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

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 2 For `scope_success`, `f()` will not be called if the initialization fails.

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

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

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

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).

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 3 Copying instead of moving provides the strong exception guarantee.

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.

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

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

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

`~scope_exit() noexcept(true);`

Effects: Equivalent to:

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

`~scope_fail() noexcept(true);`

Effects: Equivalent to:

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

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

Effects: Equivalent to:

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

NOTE 4 If `noexcept(exit_function())` is `false`, `exit_function()` can throw an exception, notwithstanding the restrictions of C++20, §16.4.6.13.

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

```
void release() noexcept;
```

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>;  // exposition only

    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
```

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.

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`.

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.

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

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

— `resource` otherwise.

6.2.3.2 Constructors [scopeguard.uniqueres.ctor]

`unique_resource()`

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

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

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

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 1 The first two conditions prohibit initialization from an rvalue reference when either `R1` or `D` is a specialization of `reference_wrapper`.

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

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

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 2 The explained mechanism ensures no leaking of resources.

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

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&>)
```

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

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 3 If initialization of `resource` throws an exception, `rhs` is left owning the resource and will free it in due time.

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 4 The explained mechanism ensures no leaking and no double release of resources.

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]

`~unique_resource();`

Effects: Equivalent to `reset()`.

6.2.3.4 Assignment [scopeguard.uniqueres.assign]

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

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.

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.

Returns: `*this`.

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

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]

```
void reset() noexcept;
```

Effects: Equivalent to:

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

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

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

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

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

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)`.

```
void release() noexcept;
```

Effects: Equivalent to `execute_on_reset = false`.

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

Returns: `resource`.

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

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

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

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

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

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

Returns: `get()`.

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

Returns: `deleter`.

6.2.3.6 unique_resource creation [scopeguard.uniqueres.create]

```
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>);
```

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

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

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`.

NOTE This creation function exists to avoid calling a deleter function with an invalid argument.

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); });
```

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]

This subclause contains templates that may be used to transform one type to another following some predefined rule.

Each of the templates in this subclause and in Table 4 shall be a *TransformationTrait* (C++20, §20.15.2).

Within this subclause, 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, §12.4.4) 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`.

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, §7.6.1.3) 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.

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*

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.

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, ...)`.

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;
};
```

`nonesuch` has no default constructor (C++20, §11.4.5) or initializer-list constructor (C++20, §9.4.5), and is not an aggregate (C++20, §9.4.2).

```
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 1

```
// 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>;
```

EXAMPLE 2

```
// 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>;
```

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]

The specification of all declarations within 7.2 are the same as the corresponding declarations, as specified in C++20, §20.14.17.3, unless explicitly specified otherwise.

NOTE `std::experimental::function` uses `std::bad_function_call`, there is no additional type `std::experimental::bad_function_call`

.

```
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;
};

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

```
      function(R(*)(ArgTypes...)) -> function<R(ArgTypes...)>;

    template<class F>
      function(F) -> function<see below>;

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

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

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.

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 1 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.

The deduction guide `template<class F> function(F) -> function<`*`see below`*`>;` is specified in C++20, §20.14.17.3.

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

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

Returns: `*this`.

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

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

Returns: `*this`.

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

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

Postconditions: `!(*this)`.

NOTE 2 The stored allocator is unchanged.

Returns: `*this`.

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

Constraints: `declval<decay_t<F>&>()` is *Lvalue-Callable* (C++20, §20.14.17.3) for argument types `ArgTypes...` and return type `R`.

Effects:

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

Returns: `*this`.

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

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

Returns: `*this`.

7.2.3 Modifiers [func.wrap.func.mod]

```
void swap(function& other);
```

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

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

Throws: Nothing.

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

7.2.4 Observers [func.wrap.func.obs]

```
allocator_type get_allocator() const noexcept;
```

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
```

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 1 There is no watched object when the stored pointer is `nullptr`.

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`.

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.

NOTE 2 The uses of `observer_ptr` include clarity of interface specification in new code, and interoperability with pointer-based legacy code.

8.2.2 observer_ptr constructors [memory.observer.ptr.ctor]

```
constexpr observer_ptr() noexcept;
constexpr observer_ptr(nullptr_t) noexcept;
```

Effects: Constructs an observer_ptr object that has no corresponding watched object.

Postconditions: `get() == nullptr`.

```
constexpr explicit observer_ptr(pointer other) noexcept;
```

Postconditions: `get() == other`.

```
template <class W2> constexpr observer_ptr(observer_ptr<W2> other) noexcept;
```

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

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

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

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

Returns: The stored pointer.

```
constexpr reference operator*() const;
```

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

Returns: `*get()`.

Throws: Nothing.

```
constexpr pointer operator->() const noexcept;
```

Returns: `get()`.

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

Returns: `get() != nullptr`.

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

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

Returns: `get()`.

8.2.5 `observer_ptr` modifiers [memory.observer.ptr.mod]

```
constexpr pointer release() noexcept;
```

Postconditions: `get() == nullptr`.

Returns: The value `get()` had at the start of the call to `release`.

```
constexpr void reset(pointer p = nullptr) noexcept;
```

Postconditions: `get() == p`.

```
constexpr void swap(observer_ptr& other) noexcept;
```

Effects: Invokes `swap` on the stored pointers of `*this` and `other`.