

TECHNICAL SPECIFICATION

**ISO/IEC TS
19568**

First edition
2015-10-01

Programming Languages — C++ Extensions for Library Fundamentals

*Langages de programmation — Extensions C++ pour les
fondamentaux de bibliothèque*

iTeh STANDARD PREVIEW
(standards.iteh.ai)

[ISO/IEC TS 19568:2015](#)

<https://standards.iteh.ai/catalog/standards/sist/cbc3a510-4101-4ca9-af58-82e9028b4b8b/iso-iec-ts-19568-2015>



Reference number
ISO/IEC TS 19568:2015(E)

© ISO/IEC 2015

iTeh STANDARD PREVIEW (standards.iteh.ai)

[ISO/IEC TS 19568:2015](#)

<https://standards.iteh.ai/catalog/standards/sist/cbc3a510-4101-4ca9-af58-82e9028b4b8b/iso-iec-ts-19568-2015>



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2015, Published in Switzerland

All rights reserved. Unless otherwise specified, no part of this publication may be reproduced or utilized otherwise in any form or by any means, electronic or mechanical, including photocopying, or posting on the internet or an intranet, without prior written permission. Permission can be requested from either ISO at the address below or ISO's member body in the country of the requester.

ISO copyright office
Ch. de Blandonnet 8 • CP 401
CH-1214 Vernier, Geneva, Switzerland
Tel. +41 22 749 01 11
Fax +41 22 749 09 47
copyright@iso.org
www.iso.org

Contents

Foreword	6
1 General	7
1.1 Scope	7
1.2 Normative references	7
1.3 Namespaces, headers, and modifications to standard classes	7
1.4 Terms and definitions	8
1.5 Future plans (Informative)	8
1.6 Feature-testing recommendations (Informative)	8
2 Modifications to the C++ Standard Library	10
2.1 Uses-allocator construction	10
3 General utilities library	11
3.1 Utility components	11
3.1.1 Header <experimental/utility> synopsis	11
3.1.2 Class erased_type	11
3.2 Tuples	11
3.2.1 Header <experimental/tuple> synopsis	11
3.2.2 Calling a function with a tuple of arguments	12
3.3 Metaprogramming and type traits	12
3.3.1 Header <experimental/type_traits> synopsis	12
3.3.2 Other type transformations	15
3.4 Compile-time rational arithmetic	16
3.4.1 Header <experimental/ratio> synopsis	16
3.5 Time utilities	17
3.5.1 Header <experimental/chrono> synopsis	17
3.6 System error support	17
3.6.1 Header <experimental/system_error> synopsis	17
4 Function objects	18
4.1 Header <experimental/functional> synopsis	18
4.2 Class template function	19
4.2.1 function construct/copy/destroy	21
4.2.2 function modifiers	21
4.3 Searchers	22
4.3.1 Class template default_searcher	22
4.3.1.1 default_searcher creation functions	23
4.3.2 Class template boyer_moore_searcher	23
4.3.2.1 boyer_moore_searcher creation functions	24
4.3.3 Class template boyer_moore_horspool_searcher	24
4.3.3.1 boyer_moore_horspool_searcher creation functions	25
5 Optional objects	26
5.1 In general	26
5.2 Header <experimental/optional> synopsis	26
5.3 optional for object types	27
5.3.1 Constructors	29
5.3.2 Destructor	30
5.3.3 Assignment	31
5.3.4 Swap	33
5.3.5 Observers	33
5.4 In-place construction	34

5.5	No-value state indicator	34
5.6	Class <code>bad_optional_access</code>	35
5.7	Relational operators	35
5.8	Comparison with <code>nullopt</code>	35
5.9	Comparison with <code>T</code>	36
5.10	Specialized algorithms	37
5.11	Hash support	37
6	Class <code>any</code>	38
6.1	Header <code><experimental/any></code> synopsis	38
6.2	Class <code>bad_any_cast</code>	39
6.3	Class <code>any</code>	39
6.3.1	<code>any</code> construct/destruct	39
6.3.2	<code>any</code> assignments	40
6.3.3	<code>any</code> modifiers	41
6.3.4	<code>any</code> observers	41
6.4	Non-member functions	41
7	string_view	43
7.1	Header <code><experimental/string_view></code> synopsis	43
7.2	Class template <code>basic_string_view</code>	44
7.3	<code>basic_string_view</code> constructors and assignment operators	46
7.4	<code>basic_string_view</code> iterator support	47
7.5	<code>basic_string_view</code> capacity	47
7.6	<code>basic_string_view</code> element access	48
7.7	<code>basic_string_view</code> modifiers	48
7.8	<code>basic_string_view</code> string operations	49
7.8.1	Searching <code>basic_string_view</code>	50
7.9	<code>basic_string_view</code> non-member comparison functions	52
7.10	Inverters and extractors	53
7.11	Hash support	53
8	Memory	54
8.1	Header <code><experimental/memory></code> synopsis	54
8.2	Shared-ownership pointers	56
8.2.1	Class template <code>shared_ptr</code>	56
8.2.1.1	<code>shared_ptr</code> constructors	60
8.2.1.2	<code>shared_ptr</code> observers	61
8.2.1.3	<code>shared_ptr</code> casts	62
8.2.2	Class template <code>weak_ptr</code>	63
8.2.2.1	<code>weak_ptr</code> constructors	64
8.3	Type-erased allocator	64
8.4	Header <code><experimental/memory_resource></code> synopsis	64
8.5	Class <code>memory_resource</code>	65
8.5.1	Class <code>memory_resource</code> overview	65
8.5.2	<code>memory_resource</code> public member functions	66
8.5.3	<code>memory_resource</code> protected virtual member functions	66
8.5.4	<code>memory_resource</code> equality	67
8.6	Class template <code>polymorphic_allocator</code>	67
8.6.1	Class template <code>polymorphic_allocator</code> overview	67
8.6.2	<code>polymorphic_allocator</code> constructors	68
8.6.3	<code>polymorphic_allocator</code> member functions	68
8.6.4	<code>polymorphic_allocator</code> equality	70
8.7	template alias <code>resource_adaptor</code>	70
8.7.1	<code>resource_adaptor</code>	70

8.7.2	resource_adaptor_imp constructors	71	
8.7.3	resource_adaptor_imp member functions	71	
8.8	Access to program-wide memory_resource objects	72	
8.9	Pool resource classes	72	
8.9.1	Classes synchronized_pool_resource and unsynchronized_pool_resource	72	
8.9.2	pool_options data members	74	
8.9.3	pool resource constructors and destructors	75	
8.9.4	pool resource members	75	
8.10	Class monotonic_buffer_resource	76	
8.10.1	Class monotonic_buffer_resource overview	76	
8.10.2	monotonic_buffer_resource constructor and destructor	77	
8.10.3	monotonic_buffer_resource members	78	
8.11	Alias templates using polymorphic memory resources	78	
8.11.1	Header <experimental/string> synopsis	78	
8.11.2	Header <experimental/deque> synopsis	79	
8.11.3	Header <experimental/forward_list> synopsis	79	
8.11.4	Header <experimental/list> synopsis	79	
8.11.5	Header <experimental/vector> synopsis	80	
8.11.6	Header <experimental/map> synopsis	80	
8.11.7	Header <experimental/set> synopsis	81	
8.11.8	Header <experimental/unordered_map> synopsis	81	
8.11.9	Header <experimental/unordered_set> synopsis	82	
8.11.10	Header <experimental/regex> synopsis	82	
9	Futures	83	
9.1	Header <experimental/future> synopsis	83	
9.2	Class template promise	83	
9.3	Class template packaged_task	84	
10	Algorithms library	ISO/IEC TS 19568:2015 https://public.dhe.ibm.com/atsys/standards/sist/cbc3a510-4101-4ca9-a58-82e9028b4b8b/iso-iec-ts-19568-2015	86
10.1	Header <experimental/algorithms> synopsis	86	
10.2	Search	86	
10.3	Shuffling and sampling	87	

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. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

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).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO and IEC shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

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

<http://Standards.iteh.ai/catalog/standards/sist/cbc3a510-4101-4ca9-a58-82e9028b4b8b/iso-iec-ts-19568-2015>

For an explanation on the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the WTO principles in the Technical Barriers to Trade (TBT), see the following URL: [Foreword — Supplementary information](#).

The committee responsible for this document is ISO/IEC JTC 1.

1 General

[general]

1.1 Scope

[general.scope]

- ¹ This technical specification describes extensions to the C++ Standard Library (1.2). These extensions are classes and functions that are likely to be used widely within a program and/or on the interface boundaries between libraries written by different organizations.
- ² This technical specification is non-normative. Some of the library components in this technical specification may be considered for standardization in a future version of C++, but they are not currently part of any C++ standard. Some of the components in this technical specification may never be standardized, and others may be standardized in a substantially changed form.
- ³ The goal of this technical specification is to build more widespread existing practice for an expanded C++ standard library. It gives advice on extensions to those vendors who wish to provide them.

1.2 Normative references

[general.references]

- ¹ The following referenced document is indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.
 - ISO/IEC 14882:2014, *Programming Languages — C++*

iTeh STANDARD PREVIEW (standards.iteh.ai)

- ² ISO/IEC 14882:— is herein called the *C++ Standard*. References to clauses within the C++ Standard are written as "C++14 §3.2". The library described in ISO/IEC 14882:— clauses 17–30 is herein called the *C++ Standard Library*.
- ³ Unless otherwise specified, the whole of the C++ Standard's ISO/IEC TS 19568:2015 introduction (C++14 §17) is included into this Technical Specification by reference. <https://standards.iteh.ai/catalog/standards/sist/cbc3a510-4101-4ca9-a58-82e9028b4b8b/iso-iec-ts-19568-2015>

1.3 Namespaces, headers, and modifications to standard classes

[general.namespaces]

- ¹ Since the extensions described in this technical specification are experimental and not part of the C++ standard library, they should not be declared directly within namespace `std`. Unless otherwise specified, all components described in this technical specification either:
 - modify an existing interface in the C++ Standard Library in-place,
 - are declared in a namespace whose name appends `::experimental::fundamentals_v1` 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_v1::chrono` because the C++ Standard Library defines `std::chrono`. This TS does not define `std::pmr::experimental::fundamentals_v1` because the C++ Standard Library does not define `std::pmr`. — *end example*]

- ² Each header described in this technical specification shall import the contents of `std::experimental::fundamentals_v1` into `std::experimental` as if by

```
namespace std {
    namespace experimental {
        inline namespace fundamentals_v1 {} // ...
    }
}
```

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_v1::`, 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

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

1.4 Terms and definitions iTeh STANDARD PREVIEW [general.defns]

(standards.iteh.ai)

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

1.4.1

direct-non-list-initialization

[ISO/IEC TS 19568:2015](#)

[general.defns.direct-non-list-init]

A direct-initialization that is not list-initialization.
<https://standards.iteh.ai/catalog/standards/sist/cbc3a510-4101-4ca9-aaf58-82e9028b4b8b/iso-iec-ts-19568-2015>

1.5 Future plans (Informative) [general.plans]

- ¹ This section describes tentative plans for future versions of this technical specification and plans for moving content into future versions of the C++ Standard.
- ² The C++ committee intends to release a new version of this technical specification approximately every year, containing the library extensions we hope to add to a near-future version of the C++ Standard. Future versions will define their contents in `std::experimental::fundamentals_v2`, `std::experimental::fundamentals_v3`, etc., with the most recent implemented version inlined into `std::experimental`.
- ³ When an extension defined in this or a future version of this technical specification represents enough existing practice, it will be moved into the next version of the C++ Standard by removing the `experimental::fundamentals_vN` segment of its namespace and by removing the `experimental/` prefix from its header's path.

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

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

- ² Implementers who provide a new standard feature should define a macro with the recommended name, in the same circumstances under which the feature is available (for example, taking into account relevant command-line options), to indicate the presence of support for that feature. Implementers should define that macro with the value specified in the most recent version of this technical specification that they have implemented. The recommended macro name is "`__cpp_lib_experimental_`" followed by the string in the "Macro Name Suffix" column.
- ³ Programmers who wish to determine whether a feature is available in an implementation should base that determination on the presence of the header (determined with `_has_include(<header/name>)`) and the state of the macro with the recommended name. (The absence of a tested feature may result in a program with decreased functionality, or the relevant functionality may be provided in a different way. A program that strictly depends on support for a feature can just try to use the feature unconditionally; presumably, on an implementation lacking necessary support, translation will fail.)

Table 2 — Significant features in this technical specification

Doc. No.	Title	Primary Section	Macro Name Suffix	Value	Header
N3915	apply() call a function with arguments from a tuple	3.2.2	apply	201402	<experimental/tuple>
N3932	Variable Templates For Type Traits	3.3.1	type_trait_variable_templates	201402	<experimental/type_traits>
N3866	Invocation type traits	3.3.2	invocation_type	201406	<experimental/type_traits>
N3916	Type-erased allocator for <code>std::function</code>	4.2	function_erased_allocator	201406	<experimental/functional>
N3905	Extending <code>std::search</code> to use Additional Searching Algorithms	4.3	(boyer_moore_searching)	201411	<experimental/functional>
N3672, N3793	A utility class to represent optional objects	5	ISO/IEC TS 19568:2015 optional	201411 101-4ca9-a58- 82e9028b4b8b/iso-iec-ts-19568-2015	<experimental/optional>
N3804	Any Library Proposal	6	any	201411	<experimental/any>
N3921	<code>string_view</code> : a non-owning reference to a string	7	string_view	201411	<experimental/string_view>
N3920	Extending <code>shared_ptr</code> to Support Arrays	8.2	shared_ptr_arrays	201406	<experimental/memory>
N3916	Polymorphic Memory Resources	8.4	memory_resources	201402	<experimental/memory_resource>
N3916	Type-erased allocator for <code>std::promise</code>	9.2	promise_erased_allocator	201406	<experimental/future>
N3916	Type-erased allocator for <code>std::packaged_task</code>	9.3	packaged_task_erased_allocator	201406	<experimental/future>
N3925	A sample Proposal	10.3	sample	201402	<experimental/algorithm>

2 Modifications to the C++ Standard Library

[**mods**]

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

2.1 Uses-allocator construction

[**mods.allocator.uses**]

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

20.7.7 `uses_allocator` [`allocator.uses`]

20.7.7.1 `uses_allocator` trait [`allocator.uses.trait`]

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

Remarks: Automatically detects whether `T` has a nested `allocator_type` that is convertible from `Alloc`.

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

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

<https://standards.iteh.ai/catalog/standards/sist/cbc3a510-4101-4ca9-aaf58-82002964181/section-20568-2015>

20.7.7.2 `uses_allocator` construction [`allocator.uses.construction`]

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

3 General utilities library

[utilities]

3.1 Utility components

[utility]

3.1.1 Header <experimental/utility> synopsis

[utility.synop]

```
#include <utility>

namespace std {
namespace experimental {
inline namespace fundamentals_v1 {

3.1.2, erased-type placeholder
struct erased_type { };

} // namespace fundamentals_v1
} // namespace experimental
} // namespace std
```

3.1.2 Class `erased_type`

iTeh STANDARD PREVIEW (standards.iteh.ai)

[utility.erased.type]

¹ struct erased_type { };

² The `erased_type` struct is an empty struct that serves as a placeholder for a type `T` in situations where the actual type `T` is determined at runtime. For example, the nested type, `allocator_type`, is an alias for `erased_type` in classes that use *type-erased allocators* (see [ISO/IEC TS 19568:2015](#)).

<https://standards.iteh.ai/catalog/standards/sist/cbc3a510-4101-4ca9-aaf58-82e9028b4b8b/iso-iec-ts-19568-2015>

3.2 Tuples

[tuple]

3.2.1 Header <experimental/tuple> synopsis

[header(tuple.synop)]

```
#include <tuple>

namespace std {
namespace experimental {
inline namespace fundamentals_v1 {

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

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

} // namespace fundamentals_v1
} // namespace experimental
} // namespace std
```

3.2.2 Calling a function with a tuple of arguments

[tuple.apply]

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

² Effects: Given the exposition only function

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

Equivalent to

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

3.3 Metaprogramming and type traits

[meta]

3.3.1 Header <experimental/type_traits> synopsis

[meta.type.synop]

```
#include <type_traits>

namespace std {
    namespace experimental {
        inline namespace fundamentals_v1
            iTeh STANDARD PREVIEW
            (standards.iteh.ai)

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

```

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

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

// See C++14 §20.10.4.3, type properties
template <class T> constexpr bool is_const_v
    = is_const<T>::value;
template <class T> constexpr bool is_volatile_v
    = is_volatile<T>::value;
template <class T> constexpr bool is_trivial_v
    = is_trivial<T>::value;
template <class T> constexpr bool is_trivially_copyable_v
    = is_trivially_copyable<T>::value;
https://standards.iteh.ai/catalog/standards/sist/cbc3a510-4101-4ca9-a5882e9028b4b8b/iso-iec-ts-19568-2015
template <class T> constexpr bool is_standard_layout_v
    = is_standard_layout<T>::value;
template <class T> constexpr bool is_pod_v
    = is_pod<T>::value;
template <class T> constexpr bool is_literal_type_v
    = is_literal_type<T>::value;
template <class T> constexpr bool is_empty_v
    = is_empty<T>::value;
template <class T> constexpr bool is_polymorphic_v
    = is_polymorphic<T>::value;
template <class T> constexpr bool is_abstract_v
    = is_abstract<T>::value;
template <class T> constexpr bool is_final_v
    = is_final<T>::value;
template <class T> constexpr bool is_signed_v
    = is_signed<T>::value;
template <class T> constexpr bool is_unsigned_v
    = is_unsigned<T>::value;
template <class T, class... Args> constexpr bool is_constructible_v
    = is_constructible<T, Args...>::value;
template <class T> constexpr bool is_default_constructible_v
    = is_default_constructible<T>::value;
template <class T> constexpr bool is_copy_constructible_v
    = is_copy_constructible<T>::value;

```

ITEh STANDARD PREVIEW (standards.iteh.ai)

ISO/IEC TS 19568:2015

<https://standards.iteh.ai/catalog/standards/sist/cbc3a510-4101-4ca9-a5882e9028b4b8b/iso-iec-ts-19568-2015>

```

    = is_copy_constructible<T>::value;
template <class T> constexpr bool is_move_constructible_v
    = is_move_constructible<T>::value;
template <class T, class U> constexpr bool is_assignable_v
    = is_assignable<T, U>::value;
template <class T> constexpr bool is_copyAssignable_v
    = is_copyAssignable<T>::value;
template <class T> constexpr bool is_moveAssignable_v
    = is_moveAssignable<T>::value;
template <class T> constexpr bool is_destructible_v
    = isDestructible<T>::value;
template <class T, class... Args> constexpr bool is_triviallyConstructible_v
    = isTriviallyConstructible<T, Args...>::value;
template <class T> constexpr bool is_triviallyDefaultConstructible_v
    = isTriviallyDefaultConstructible<T>::value;
template <class T> constexpr bool is_triviallyCopyConstructible_v
    = isTriviallyCopyConstructible<T>::value;
template <class T> constexpr bool is_triviallyMoveConstructible_v
    = isTriviallyMoveConstructible<T>::value;
template <class T, class U> constexpr bool is_triviallyAssignable_v
    = isTriviallyAssignable<T, U>::value;
template <class T> constexpr bool is_triviallyCopyAssignable_v
    = isTriviallyCopyAssignable<T>::value;
template <class T> constexpr bool is_triviallyMoveAssignable_v
    = isTriviallyMoveAssignable<T>::value;
template <class T> constexpr bool is_triviallyDestructible_v
    = isTriviallyDestructible<T>::value;
template <class T, class... Args> constexpr bool is_nothrowConstructible_v
    = isnothrowConstructible<T, Args...>::value;
https://standards.iteh.ai/catalog/standards/sist/cbc3a510-4101-4ca9-af58-82e9028ab8b/iso-iec-ts-19568-2015
template <class T> constexpr bool is_nothrowDefaultConstructible_v
    = isnothrowDefaultConstructible<T>::value;
template <class T> constexpr bool is_nothrowCopyConstructible_v
    = isnothrowCopyConstructible<T>::value;
template <class T> constexpr bool is_nothrowMoveConstructible_v
    = isnothrowMoveConstructible<T>::value;
template <class T, class U> constexpr bool is_nothrowAssignable_v
    = isnothrowAssignable<T, U>::value;
template <class T> constexpr bool is_nothrowCopyAssignable_v
    = isnothrowCopyAssignable<T>::value;
template <class T> constexpr bool is_nothrowMoveAssignable_v
    = isnothrowMoveAssignable<T>::value;
template <class T> constexpr bool is_nothrowDestructible_v
    = isnothrowDestructible<T>::value;
template <class T> constexpr bool hasVirtualDestructor_v
    = hasVirtualDestructor<T>::value;

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

```

```

= extent<T, I>::value;

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

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

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

} // namespace fundamentals_v1
} // namespace experimental
} // namespace std

```

The STANDARD PREVIEW (standards.iteh.ai)

3.3.2 Other type transformations

[meta.trans.other]

[ISO/IEC TS 19568:2015](#)

¹ This sub-clause contains templates that may be used to transform one type to another following some predefined rule. <https://standards.iec.org/catalog/standards/sis/iec62310-4101-4ca9-a156>

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

³ Within this section, define the *invocation parameters* of *INVOKE(f, t₁, t₂, ..., t_N)* 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, ..., t_N.
- 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, ..., t_N of the best viable function (C++14 §13.3.3) for the arguments t₁, ..., t_N among the function call operators of f.
- In all other cases, the *invocation parameters* are the parameters of f matching t₁, ... t_N.

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

[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).