
**Programming languages — C++
extensions for library fundamentals**

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

iTeh Standards
(<https://standards.iteh.ai>)
Document Preview

[ISO/IEC TS 19568:2017](https://standards.iteh.ai/catalog/standards/iso/3ec9ecf6-f1ca-4ea0-8feb-4186403ea99e/iso-iec-ts-19568-2017)

<https://standards.iteh.ai/catalog/standards/iso/3ec9ecf6-f1ca-4ea0-8feb-4186403ea99e/iso-iec-ts-19568-2017>



iTeh Standards
(<https://standards.iteh.ai>)
Document Preview

[ISO/IEC TS 19568:2017](https://standards.iteh.ai/catalog/standards/iso/3ec9ecf6-f1ca-4ea0-8feb-4186403ea99e/iso-iec-ts-19568-2017)

<https://standards.iteh.ai/catalog/standards/iso/3ec9ecf6-f1ca-4ea0-8feb-4186403ea99e/iso-iec-ts-19568-2017>



COPYRIGHT PROTECTED DOCUMENT

© ISO/IEC 2017, 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

Forewordvii
1 General	1
1.1 Scope	1
1.2 Normative references	1
1.3 Namespaces, headers, and modifications to standard classes	1
1.4 Terms and definitions	2
1.5 Future plans (Informative)	2
1.6 Feature-testing recommendations (Informative)	3
2 Modifications to the C++ Standard Library	5
2.1 Uses-allocator construction	5
3 General utilities library	6
3.1 Utility components	6
3.1.1 Header <experimental/utility> synopsis	6
3.1.2 Class erased_type	6
3.2 Tuples	6
3.2.1 Header <experimental/tuple> synopsis	6
3.2.2 Calling a function with a tuple of arguments	7
3.3 Metaprogramming and type traits	7
3.3.1 Header <experimental/type_traits> synopsis	7
3.3.2 Other type transformations	11
3.3.3 Logical operator traits	12
3.3.4 Detection idiom	13
3.4 Compile-time rational arithmetic	14
3.4.1 Header <experimental/ratio> synopsis	14
3.5 Time utilities	15
3.5.1 Header <experimental/chrono> synopsis	15
3.6 System error support	15
3.6.1 Header <experimental/system_error> synopsis	15
3.7 Class template propagate_const	15
3.7.1 Class template propagate_const general	15
3.7.2 Header <experimental/propagate_const> synopsis	16
3.7.3 propagate_const requirements on T	18
3.7.3.1 propagate_const requirements on class type T	18
3.7.4 propagate_const constructors	19
3.7.5 propagate_const assignment	19
3.7.6 propagate_const const observers	20
3.7.7 propagate_const non-const observers	20
3.7.8 propagate_const modifiers	21
3.7.9 propagate_const relational operators	21
3.7.10 propagate_const specialized algorithms	23
3.7.11 propagate_const underlying pointer access	23
3.7.12 propagate_const hash support	23
3.7.13 propagate_const comparison function objects	23
4 Function objects	25
4.1 Header <experimental/functional> synopsis	25
4.2 Class template function	26
4.2.1 function construct/copy/destroy	28
4.2.2 function modifiers	29

4.3	Searchers	29
4.3.1	Class template default_searcher	29
4.3.1.1	default_searcher creation functions	30
4.3.2	Class template boyer_moore_searcher	30
4.3.2.1	boyer_moore_searcher creation functions	31
4.3.3	Class template boyer_moore_horspool_searcher	31
4.3.3.1	boyer_moore_horspool_searcher creation functions	32
4.4	Function template not_fn	33
5	Optional objects	34
5.1	In general	34
5.2	Header <experimental/optional> synopsis	34
5.3	optional for object types	35
5.3.1	Constructors	37
5.3.2	Destructor	39
5.3.3	Assignment	40
5.3.4	Swap	43
5.3.5	Observers	43
5.4	In-place construction	44
5.5	No-value state indicator	44
5.6	Class bad_optional_access	45
5.7	Relational operators	45
5.8	Comparison with nullopt	45
5.9	Comparison with T	46
5.10	Specialized algorithms	47
5.11	Hash support	47
6	Class any	48
6.1	Header <experimental/any> synopsis	48
6.2	Class bad_any_cast	49
6.3	Class any	49
6.3.1	any construct/destroy	49
6.3.2	any assignments	50
6.3.3	any modifiers	51
6.3.4	any observers	51
6.4	Non-member functions	51
7	string_view	54
7.1	Header <experimental/string_view> synopsis	54
7.2	Class template basic_string_view	55
7.3	basic_string_view constructors and assignment operators	57
7.4	basic_string_view iterator support	58
7.5	basic_string_view capacity	59
7.6	basic_string_view element access	59
7.7	basic_string_view modifiers	60
7.8	basic_string_view string operations	60
7.8.1	Searching basic_string_view	62
7.9	basic_string_view non-member comparison functions	63
7.10	Inserters and extractors	64
7.11	Hash support	65
8	Memory	66
8.1	Header <experimental/memory> synopsis	66
8.2	Shared-ownership pointers	69
8.2.1	Class template shared_ptr	69
8.2.1.1	shared_ptr constructors	72

	8.2.1.2	shared_ptr observers	74
	8.2.1.3	shared_ptr casts	75
	8.2.1.4	shared_ptr hash support	75
	8.2.2	Class template weak_ptr	75
	8.2.2.1	weak_ptr constructors	76
8.3		Type-erased allocator	77
8.4		Header <experimental/memory_resource> synopsis	77
8.5		Class memory_resource	78
	8.5.1	Class memory_resource overview	78
	8.5.2	memory_resource public member functions	79
	8.5.3	memory_resource protected virtual member functions	79
	8.5.4	memory_resource equality	80
8.6		Class template polymorphic_allocator	80
	8.6.1	Class template polymorphic_allocator overview	80
	8.6.2	polymorphic_allocator constructors	81
	8.6.3	polymorphic_allocator member functions	81
	8.6.4	polymorphic_allocator equality	83
8.7		template alias resource_adaptor	83
	8.7.1	resource_adaptor	83
	8.7.2	resource_adaptor_imp constructors	84
	8.7.3	resource_adaptor_imp member functions	84
8.8		Access to program-wide memory_resource objects	85
8.9		Pool resource classes	85
	8.9.1	Classes synchronized_pool_resource and unsynchronized_pool_resource	85
	8.9.2	pool_options data members	87
	8.9.3	pool resource constructors and destructors	88
	8.9.4	pool resource members	88
8.10		Class monotonic_buffer_resource	89
	8.10.1	Class monotonic_buffer_resource overview	89
	8.10.2	monotonic_buffer_resource constructor and destructor	90
	8.10.3	monotonic_buffer_resource members	91
8.11		Alias templates using polymorphic memory resources	91
	8.11.1	Header <experimental/string> synopsis	91
	8.11.2	Header <experimental/deque> synopsis	92
	8.11.3	Header <experimental/forward_list> synopsis	92
	8.11.4	Header <experimental/list> synopsis	92
	8.11.5	Header <experimental/vector> synopsis	93
	8.11.6	Header <experimental/map> synopsis	93
	8.11.7	Header <experimental/set> synopsis	94
	8.11.8	Header <experimental/unordered_map> synopsis	94
	8.11.9	Header <experimental/unordered_set> synopsis	95
	8.11.10	Header <experimental/regex> synopsis	95
8.12		Non-owning pointers	96
	8.12.1	Class template observer_ptr overview	96
	8.12.2	observer_ptr constructors	97
	8.12.3	observer_ptr observers	97
	8.12.4	observer_ptr conversions	97
	8.12.5	observer_ptr modifiers	97
	8.12.6	observer_ptr specialized algorithms	98
	8.12.7	observer_ptr hash support	99
9		Containers	100
	9.1	Uniform container erasure	100

9.1.1	Header synopsis	100
9.1.2	Function template <code>erase_if</code>	101
9.1.3	Function template <code>erase</code>	102
9.2	Class template <code>array</code>	102
9.2.1	Header <code><experimental/array></code> synopsis	102
9.2.2	Array creation functions	103
10	Iterators library	104
10.1	Header <code><experimental/iterator></code> synopsis	104
10.2	Class template <code>ostream_joiner</code>	104
10.2.1	<code>ostream_joiner</code> constructor	105
10.2.2	<code>ostream_joiner</code> operations	105
10.2.3	<code>ostream_joiner</code> creation function	105
11	Futures	106
11.1	Header <code><experimental/future></code> synopsis	106
11.2	Class template <code>promise</code>	106
11.3	Class template <code>packaged_task</code>	107
12	Algorithms library	109
12.1	Header <code><experimental/algorithm></code> synopsis	109
12.2	<code>Search</code>	109
12.3	<code>Sampling</code>	110
12.4	<code>Shuffle</code>	110
13	Numerics library	111
13.1	Generalized numeric operations	111
13.1.1	Header <code><experimental/numeric></code> synopsis	111
13.1.2	Greatest common divisor	111
13.1.3	Least common multiple	111
13.2	Random number generation	112
13.2.1	Header <code><experimental/random></code> synopsis	112
13.2.2	Utilities	112
13.2.2.1	Function template <code>randint</code>	112
14	Reflection library	113
14.1	Class <code>source_location</code>	113
14.1.1	Header <code><experimental/source_location></code> synopsis	113
14.1.2	<code>source_location</code> creation	114
14.1.3	<code>source_location</code> field access	114

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

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 ISO documents 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 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 information given for the convenience of users and does not constitute an endorsement. For an explanation on 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 the following URL: www.iso.org/iso/foreword.html

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 edition of ISO/IEC 19568:2017 cancels and replaces the edition ISO/IEC 19568:2015, which has been technically revised and includes the following changes:

- Addition of the `sample` algorithm.
- Addition of new random-number generation facilities, and algorithms which use them.
- Addition of algorithms for uniform container erasure.
- Addition of function template `not_fn`.
- Addition of logical operator type traits `conjunction`, `disjunction`, and `negation`.
- Addition of templates to support the "detection idiom".
- Addition of the `propagate_const` class template.
- Addition of the `observer_ptr` class template.
- Addition of the `make_array` and `to_array` function templates.
- Addition of the `ostream_joiner` class template.
- Addition of the `gcd` and `lcm` algorithms.
- Addition of the `source_location` struct.
- Changes to the return types of search algorithms.
- Moving all libraries to the inline namespace `fundamentals_v2`.
- Miscellaneous defect resolutions.

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 Language — C++*
- ² ISO/IEC 14882:2014 is herein called the *C++ Standard*. References to clauses within the C++ Standard are written as "C++14 §3.2". The library described in ISO/IEC 14882:2014 clauses 17–30 is herein called the *C++ Standard Library*.
- ³ Unless otherwise specified, the whole of the C++ Standard's Library introduction (C++14 §17) is included into this Technical Specification by reference.

1.3 Namespaces, headers, and modifications to standard classes

[general.namespaces]

- ¹ Since the extensions described in this technical specification are experimental and not part of the C++ standard library, they should not be declared directly within namespace `std`. Unless otherwise specified, all components described in this technical specification either:
 - modify an existing interface in the C++ Standard Library in-place,
 - are declared in a namespace whose name appends `::experimental::fundamentals_v2` to a namespace defined in the C++ Standard Library, such as `std` or `std::chrono`, or
 - are declared in a subnamespace of a namespace described in the previous bullet, whose name is not the same as an existing subnamespace of namespace `std`.

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

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

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

```

}
}

```

- 3 This technical specification also describes some experimental modifications to existing interfaces in the C++ Standard Library. These modifications are described by quoting the affected parts of the standard and using underlining to represent added text and ~~strike-through~~ to represent deleted text.
- 4 Unless otherwise specified, references to other entities described in this technical specification are assumed to be qualified with `std::experimental::fundamentals_v2::`, and references to entities described in the standard are assumed to be qualified with `std::`.
- 5 Extensions that are expected to eventually be added to an existing header `<meow>` are provided inside the `<experimental/meow>` header, which shall include the standard contents of `<meow>` as if by

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

Table 1 — C++ library headers

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

1.4 Terms and definitions

[\[general.defns\]](#)

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

1.4.1

direct-non-list-initialization

[\[general.defns.direct-non-list-init\]](#)

A direct-initialization that is not list-initialization.

1.5 Future plans (Informative)

[\[general.plans\]](#)

- 1 This section describes tentative plans for future versions of this technical specification and plans for moving content into future versions of the C++ Standard.
- 2 The C++ committee intends to release a new version of this technical specification approximately every year, containing the library extensions we hope to add to a near-future version of the C++ Standard. Future versions will define their contents in `std::experimental::fundamentals_v3`, `std::experimental::fundamentals_v4`, etc., with the most recent implemented version inlined into `std::experimental`.
- 3 When an extension defined in this or a future version of this technical specification represents enough existing practice, it will be moved into the next version of the C++ Standard by removing the `experimental::fundamentals_vN` segment of its namespace and by removing the `experimental/` prefix from its header's path.

1.6 Feature-testing recommendations (Informative)

[\[general.feature.test\]](#)

- ¹ For the sake of improved portability between partial implementations of various C++ standards, WG21 (the ISO technical committee for the C++ programming language) recommends that implementers and programmers follow the guidelines in this section concerning feature-test macros. [*Note:* WG21's SD-6 makes similar recommendations for the C++ Standard itself. — *end note*]
- ² Implementers who provide a new standard feature should define a macro with the recommended name, in the same circumstances under which the feature is available (for example, taking into account relevant command-line options), to indicate the presence of support for that feature. Implementers should define that macro with the value specified in the most recent version of this technical specification that they have implemented. The recommended macro name is "`__cpp_lib_experimental_`" followed by the string in the "Macro Name Suffix" column.
- ³ Programmers who wish to determine whether a feature is available in an implementation should base that determination on the presence of the header (determined with `__has_include(<header/name>)`) and the state of the macro with the recommended name. (The absence of a tested feature may result in a program with decreased functionality, or the relevant functionality may be provided in a different way. A program that strictly depends on support for a feature can just try to use the feature unconditionally; presumably, on an implementation lacking necessary support, translation will fail.)

Table 2 — Significant features in this technical specification

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

Doc. No.	Title	Primary Section	Macro Name Suffix	Value	Header
N3921	string_view: a non-owning reference to a string	7	string_view	201411	<experimental/string_view>
N3920	Extending shared_ptr to Support Arrays	8.2	shared_ptr_arrays	201406	<experimental/memory>
N3916	Polymorphic Memory Resources	8.4	memory_resources	201402	<experimental/memory_resource>
N4282	The World's Dumbest Smart Pointer	8.12	observer_ptr	201411	<experimental/memory>
N4273	Uniform Container Erasure	9.1	erase_if	201411	<experimental/vector>
N4391	make_array	9.2.2	make_array	201505	<experimental/array>
N4257	Delimited iterators	10.2	ostream_joiner	201411	<experimental/iterator>
N3916	Type-erased allocator for std::promise	11.2	promise_erased_allocator	201406	<experimental/future>
N3916	Type-erased allocator for std::packaged_task	11.3	packaged_task_erased_allocator	201406	<experimental/future>
N3925	A sample Proposal	12.3	sample	201402	<experimental/algorithm>
N4061	Greatest Common Divisor and Least Common Multiple	13.1.2, 13.1.3	gcd_lcm	201411	<experimental/numeric>
N4531	std::rand replacement	13.2.2.1	randint	201511	<experimental/random>
N4519	Source-Code Information Capture	14.1	source_location-4ea0-8feb-41	201505	<experimental/source_location>

2 Modifications to the C++ Standard Library

[mods]

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

2.1 Uses-allocator construction

[mods.allocator.uses]

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

20.7.7 `uses_allocator` [allocator.uses]

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

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

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

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

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

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

3 General utilities library

[\[utilities\]](#)

3.1 Utility components

[\[utility\]](#)

3.1.1 Header <experimental/utility> synopsis

[\[utility.synop\]](#)

```
#include <utility>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

    // 3.1.2, Class erased_type
    struct erased_type { };

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

3.1.2 Class `erased_type`

[\[utility.erased.type\]](#)

```
1 struct erased_type { };
```

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

<https://standards.iteh.ai/catalog/standards/iso/3ec9ecf6-fl-ca-4ea0-8feb-4186403ea99e/iso-iec-ts-19568-2017>

<https://standards.iteh.ai/catalog/standards/iso/3ec9ecf6-fl-ca-4ea0-8feb-4186403ea99e/iso-iec-ts-19568-2017> [\[tuple\]](#)

3.2 Tuples

[\[tuple\]](#)

3.2.1 Header <experimental/tuple> synopsis

[\[header.tuple.synop\]](#)

```
#include <tuple>

namespace std {
namespace experimental {
inline namespace fundamentals_v2 {

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

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

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