
Programming Languages — C++ Extensions for Library Fundamentals

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



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

AVA Documents
Document Sharing Hub
© ISO/IEC 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/destroy	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	Inserters 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	86
10.1	Header <experimental/algorithm> 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 information given for the convenience of users and does not constitute an endorsement.

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

Table 1 — C++ library headers

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

## 1.4 Terms and definitions

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

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

### 1.4.1

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

#### direct-non-list-initialization

A direct-initialization that is not list-initialization.

## 1.5 Future plans (Informative)

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

- <sup>1</sup> This section describes tentative plans for future versions of this technical specification and plans for moving content into future versions of the C++ Standard.
- <sup>2</sup> 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`.
- <sup>3</sup> 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)

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

- <sup>1</sup> 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* ]

- <sup>2</sup> 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.
- <sup>3</sup> 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               | optional                       | 201411 | <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]

- <sup>1</sup> 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]

- <sup>1</sup> 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_v1 {

 3.1.2, erased-type placeholder
 struct erased_type { };

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

#### 3.1.2 Class `erased_type`

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

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

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

## 3.2 Tuples

[\[tuple\]](#)

#### 3.2.1 Header <experimental/tuple> synopsis

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

```
#include <tuple>

namespace std {
namespace experimental {
inline namespace fundamentals_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 {

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

```

```

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

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

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

```

```

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

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

```

```

 = extent<T, I>::value;

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

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

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

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

```

### 3.3.2 Other type transformations

[\[meta.trans.other\]](#)

- <sup>1</sup> This sub-clause contains templates that may be used to transform one type to another following some predefined rule.
- <sup>2</sup> Each of the templates in this subclause shall be a *TransformationTrait* (C++14 §20.10.1).
- <sup>3</sup> Within this section, define the *invocation parameters* of *INVOKE*(*f*, *t*<sub>1</sub>, *t*<sub>2</sub>, ..., *t*<sub>N</sub>) as follows, in which *T*<sub>1</sub> is the possibly cv-qualified type of *t*<sub>1</sub> and *U*<sub>1</sub> denotes *T*<sub>1</sub>& if *t*<sub>1</sub> is an lvalue or *T*<sub>1</sub>&& if *t*<sub>1</sub> is an rvalue:
  - When *f* is a pointer to a member function of a class *T* the *invocation parameters* are *U*<sub>1</sub> followed by the parameters of *f* matched by *t*<sub>2</sub>, ..., *t*<sub>N</sub>.
  - When *N* == 1 and *f* is a pointer to member data of a class *T* the *invocation parameter* is *U*<sub>1</sub>.
  - If *f* is a class object, the *invocation parameters* are the parameters matching *t*<sub>1</sub>, ..., *t*<sub>N</sub> of the best viable function (C++14 §13.3.3) for the arguments *t*<sub>1</sub>, ..., *t*<sub>N</sub> among the function call operators of *f*.
  - In all other cases, the *invocation parameters* are the parameters of *f* matching *t*<sub>1</sub>, ... *t*<sub>N</sub>.
- <sup>4</sup> In all of the above cases, if an argument *t*<sub>I</sub> 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*<sub>I</sub>.

[ *Example*: Assume *s* is defined as

```

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

```

- The invocation parameters of *INVOKE*(&*S*::*f*, *S*(), 3.5) are (*S* &&, double const &).
- The invocation parameters of *INVOKE*(*S*(), 1, 2) are (int, int).

- The invocation parameters of `INVOKE(S(), "abc", 5)` are `(const char *, int)`. The defaulted parameter `j` does not correspond to an argument.
  - The invocation parameters of `INVOKE(S(), locale(), 5)` are `(locale, int)`. Arguments corresponding to ellipsis maintain their types.
- *end example* ]

Table 3 — Other type transformations

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

- <sup>5</sup> Access checking is performed as if in a context unrelated to `Fn` and `ArgTypes`. Only the validity of the immediate context of the expression is considered. [ *Note*: The compilation of the expression can result in side effects such as the instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. — *end note* ]
- <sup>6</sup> The nested typedef `raw_invocation_type<Fn(ArgTypes...)>::type` shall be defined as follows. If the expression `INVOKE(declval<Fn>(), declval<ArgTypes>()...)` is ill-formed when treated as an unevaluated operand (C++14 §5), there shall be no member type. Otherwise:
- Let `R` denote `result_of_t<Fn(ArgTypes...)>`.
  - Let the types `Ti` be the invocation parameters of `INVOKE(declval<Fn>(), declval<ArgTypes>()...)`.
  - Then the member typedef `type` shall name the function type `R(T1, T2, ...)`.
- <sup>7</sup> The nested typedef `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 typedef `type`. Otherwise:
- Let `A1, A2, ...` denote `ArgTypes...`
  - Let `R(T1, T2, ...)` denote `raw_invocation_type_t<Fn(ArgTypes...)>`
  - Then the member typedef `type` shall name the function type `R(U1, U2, ...)` where `Ui` is `decay_t<Ai>` if `declval<Ai>()` is an rvalue otherwise `Ti`.

### 3.4 Compile-time rational arithmetic

[ratio]

#### 3.4.1 Header <experimental/ratio> synopsis

[header.ratio.synop]

```
#include <ratio>

namespace std {
namespace experimental {
inline namespace fundamentals_v1 {

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

```

 = ratio_less_equal<R1, R2>::value;
template <class R1, class R2> constexpr bool ratio_greater_v
 = ratio_greater<R1, R2>::value;
template <class R1, class R2> constexpr bool ratio_greater_equal_v
 = ratio_greater_equal<R1, R2>::value;

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

```

## 3.5 Time utilities

[\[time\]](#)

### 3.5.1 Header <experimental/chrono> synopsis

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

```

#include <chrono>

namespace std {
namespace chrono {
namespace experimental {
inline namespace fundamentals_v1 {

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

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

```

## 3.6 System error support

[\[syserror\]](#)

### 3.6.1 Header <experimental/system\_error> synopsis

[\[header.system\\_error.synop\]](#)

```

#include <system_error>

namespace std {
namespace experimental {
inline namespace fundamentals_v1 {

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

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

```

## 4 Function objects

[func]

### 4.1 Header <experimental/functional> synopsis

[header.functional.synop]

```
#include <functional>

namespace std {
 namespace experimental {
 inline namespace fundamentals_v1 {

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

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

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

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

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

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

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

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

```

 BinaryPredicate pred = BinaryPredicate();

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

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

} // namespace fundamentals_v1
} // namespace experimental

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

} // namespace std

```

## 4.2 Class template function

[\[func.wrap.func\]](#)

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

```

namespace std {
 namespace experimental {
 inline namespace fundamentals_v1 {

 template<class> class function; // undefined

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

 typedef erased_type allocator_type;

 function() noexcept;
 function(nullptr_t) noexcept;
 function(const function&);

```

```

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

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

~function();

void swap(function&);
template<class F, class A> void assign(F&&, const A&);

explicit operator bool() const noexcept;

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

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

pmr::memory_resource* get_memory_resource();
};

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

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

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

} // namespace fundamentals_v1
} // namespace experimental

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

```

```
} // namespace std
```

#### 4.2.1 function construct/copy/destroy

[\[func.wrap.func.con\]](#)

- 1 When a `function` constructor that takes a first argument of type `allocator_arg_t` is invoked, the second argument is treated as a *type-erased allocator* (8.3). If the constructor moves or makes a copy of a function object (C++14 §20.9), including an instance of the `experimental::function` class template, then that move or copy is performed by *using-allocator construction* with allocator `get_memory_resource()`.
- 2 In the following descriptions, let `ALLOCATOR_OF(f)` be the allocator specified in the construction of function `f`, or `allocator<char>()` if no allocator was specified.
- 3 `function& operator=(const function& f);`
  - 4 **Effects:** `function(allocator_arg, ALLOCATOR_OF(*this), f).swap(*this);`
  - 5 **Returns:** `*this`
- 6 `function& operator=(function&& f);`
  - 7 **Effects:** `function(allocator_arg, ALLOCATOR_OF(*this), std::move(f)).swap(*this);`
  - 8 **Returns:** `*this`
- 9 `function& operator=(nullptr_t) noexcept;`
  - 10 **Effects:** If `*this != nullptr`, destroys the target of `this`.
  - 11 **Postconditions:** `!(*this)`. The memory resource returned by `get_memory_resource()` after the assignment is equivalent to the memory resource before the assignment. [ *Note:* the address returned by `get_memory_resource()` might change — *end note* ]
  - 12 **Returns:** `*this`
- 13 `template<class F> function& operator=(F&& f);`
  - 14 **Effects:** `function(allocator_arg, ALLOCATOR_OF(*this), std::forward<F>(f)).swap(*this);`
  - 15 **Returns:** `*this`
- 16 `template<class F> function& operator=(reference_wrapper<F> f);`
  - 17 **Effects:** `function(allocator_arg, ALLOCATOR_OF(*this), f).swap(*this);`
  - 18 **Returns:** `*this`

#### 4.2.2 function modifiers

[\[func.wrap.func.mod\]](#)

- 1 `void swap(function& other);`
  - 2 **Requires:** `this->get_memory_resource() == other->get_memory_resource()`.
  - 3 **Effects:** Interchanges the targets of `*this` and `other`.
  - 4 **Remarks:** The allocators of `*this` and `other` are not interchanged.

### 4.3 Searchers

[func.searchers]

- <sup>1</sup> This sub-clause provides function object types (C++14 §20.9) for operations that search for a sequence [pat\_first, pat\_last) in another sequence [first, last) that is provided to the object's function call operator. The first sequence (the pattern to be searched for) is provided to the object's constructor, and the second (the sequence to be searched) is provided to the function call operator.
- <sup>2</sup> Each specialization of a class template specified in this sub-clause 4.3 shall meet the CopyConstructible and CopyAssignable requirements. Template parameters named ForwardIterator, ForwardIterator1, ForwardIterator2, RandomAccessIterator, RandomAccessIterator1, RandomAccessIterator2, and BinaryPredicate of templates specified in this sub-clause 4.3 shall meet the same requirements and semantics as specified in C++14 §25.1. Template parameters named Hash shall meet the requirements as specified in C++14 §17.6.3.4.
- <sup>3</sup> The Boyer-Moore searcher implements the Boyer-Moore search algorithm. The Boyer-Moore-Horspool searcher implements the Boyer-Moore-Horspool search algorithm. In general, the Boyer-Moore searcher will use more memory and give better run-time performance than Boyer-Moore-Horspool

#### 4.3.1 Class template default\_searcher

[func.searchers.default]

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

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

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

- <sup>1</sup> default\_searcher(ForwardIterator pat\_first, ForwardIterator pat\_last, BinaryPredicate pred = BinaryPredicate());
- <sup>2</sup> **Effects:** Constructs a default\_searcher object, initializing pat\_first\_ with pat\_first, pat\_last\_ with pat\_last, and pred\_ with pred.
- <sup>3</sup> **Throws:** Any exception thrown by the copy constructor of BinaryPredicate or ForwardIterator1.
- <sup>4</sup> template<class ForwardIterator2> ForwardIterator2 operator()(ForwardIterator2 first, ForwardIterator2 last) const;
- <sup>5</sup> **Effects:** Equivalent to return std::search(first, last, pat\_first\_, pat\_last\_, pred\_);

## 4.3.1.1 default\_searcher creation functions

[\[func.searchers.default.creation\]](#)

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

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

## 4.3.2 Class template boyer\_moore\_searcher

[\[func.searchers.boyer\\_moore\]](#)

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

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

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

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

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

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

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

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

- 6 `template<class RandomAccessIterator2>`  
`RandomAccessIterator2 operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;`
- 7 **Requires:** `RandomAccessIterator1` and `RandomAccessIterator2` shall have the same value type.
- 8 **Effects:** Finds a subsequence of equal values in a sequence.
- 9 **Returns:** The first iterator `i` in the range `[first, last - (pat_last_ - pat_first_))` such that for every non-negative integer `n` less than `pat_last_ - pat_first_` the following condition holds:  
`pred(*(i + n), *(pat_first_ + n)) != false`. Returns `first` if `[pat_first_, pat_last_)` is empty, otherwise returns `last` if no such iterator is found.
- 10 **Complexity:** At most `(last - first) * (pat_last_ - pat_first_)` applications of the predicate.

#### 4.3.2.1 `boyer_moore_searcher` creation functions

[\[func.searchers.boyer\\_moore.creation\]](#)

- 1 `template<class RandomAccessIterator,`  
`class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,`  
`class BinaryPredicate = equal_to<>>`  
`boyer_moore_searcher<RandomAccessIterator, Hash, BinaryPredicate>`  
`make_boyer_moore_searcher(RandomAccessIterator pat_first, RandomAccessIterator pat_last,`  
`Hash hf = Hash(), BinaryPredicate pred = BinaryPredicate());`
- 2 **Effects:** Equivalent to `return boyer_moore_searcher<RandomAccessIterator, Hash, BinaryPredicate>(`  
`pat_first, pat_last, hf, pred);`

#### 4.3.3 Class template `boyer_moore_horspool_searcher`

[\[func.searchers.boyer\\_moore\\_horspool\]](#)

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

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

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

- 1 `boyer_moore_horspool_searcher(`  
`RandomAccessIterator1 pat_first, RandomAccessIterator1 pat_last,`  
`Hash hf = Hash(), BinaryPredicate pred = BinaryPredicate());`
- 2 **Requires:** The value type of `RandomAccessIterator1` shall meet the `DefaultConstructible`, `CopyConstructible`, and `CopyAssignable` requirements.
- 3 **Requires:** For any two values `A` and `B` of the type `iterator_traits<RandomAccessIterator1>::value_type`, if `pred(A,B)==true`, then `hf(A)==hf(B)` shall be true.
- 4 **Effects:** Constructs a `boyer_moore_horspool_searcher` object, initializing `pat_first_` with `pat_first`, `pat_last_` with `pat_last`, `hash_` with `hf`, and `pred_` with `pred`.
- 5 **Throws:** Any exception thrown by the copy constructor of `RandomAccessIterator1`, or by the default constructor, copy constructor, or the copy assignment operator of the value type of `RandomAccessIterator1` or the copy constructor or operator() of `BinaryPredicate` or `Hash`. May throw `bad_alloc` if additional memory needed for internal data structures cannot be allocated..
- 6 `template<class RandomAccessIterator2>`  
`RandomAccessIterator2 operator()(RandomAccessIterator2 first, RandomAccessIterator2 last) const;`
- 7 **Requires:** `RandomAccessIterator1` and `RandomAccessIterator2` shall have the same value type.
- 8 **Effects:** Finds a subsequence of equal values in a sequence.
- 9 **Returns:** The first iterator `i` in the range `[first, last - (pat_last_ - pat_first_))` such that for every non-negative integer `n` less than `pat_last_ - pat_first_` the following condition holds:  
`pred(*(i + n), *(pat_first_ + n)) != false`. Returns `first` if `[pat_first_, pat_last_)` is empty, otherwise returns `last` if no such iterator is found.
- 10 **Complexity:** At most `(last - first) * (pat_last_ - pat_first_)` applications of the predicate.

#### 4.3.3.1 boyer\_moore\_horspool\_searcher creation functions

[\[func.searchers.boyer\\_moore\\_horspool.creation\]](#)

- 1 `template<class RandomAccessIterator,`  
`class Hash = hash<typename iterator_traits<RandomAccessIterator>::value_type>,`  
`class BinaryPredicate = equal_to<>>`  
`boyer_moore_searcher_horspool<RandomAccessIterator, Hash, BinaryPredicate>`  
`make_boyer_moore_horspool_searcher(`  
`RandomAccessIterator pat_first, RandomAccessIterator pat_last,`  
`Hash hf = Hash(), BinaryPredicate pred = BinaryPredicate());`
- 2 **Effects:** Equivalent to  
`return boyer_moore_horspool_searcher<RandomAccessIterator, Hash, BinaryPredicate>(`  
`pat_first, pat_last, hf, pred);`

## 5 Optional objects

[optional]

### 5.1 In general

[optional.general]

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

### 5.2 Header `<experimental/optional>` synopsis

[optional.synop]

```
namespace std {
 namespace experimental {
 inline namespace fundamentals_v1 {

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

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

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

 // 5.6, Class bad_optional_access
 class bad_optional_access;

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

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

```

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

```

// 5.9, Comparison with T

```

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

```

// 5.10, Specialized algorithms

```

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

```

} // namespace fundamentals\_v1

} // namespace experimental

// 5.11, Hash support

```

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

```

} // namespace std

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

## 5.3 optional for object types

[optional.object]

```

template <class T>
class optional
{
public:
 typedef T value_type;

```

// 5.3.1, Constructors

```

constexpr optional() noexcept;
constexpr optional(nullopt_t) noexcept;
optional(const optional&);
optional(optional&&) noexcept (see below);
constexpr optional(const T&);

```