

**Doc No:** N1596=04-0036  
**Date:** February 17, 2004  
**Reply to:** Matt Austern  
austern@apple.com

# **(Draft) Technical Report on Standard Library Extensions**

# Contents

<b>1</b>	<b>General</b>	<b>11</b>
1.1	Method of description . . . . .	11
1.2	Categories of extensions . . . . .	11
1.3	Namespaces and headers . . . . .	12
1.4	Caveat . . . . .	12
<b>2</b>	<b>General Utilities</b>	<b>13</b>
2.1	Reference wrappers . . . . .	13
2.1.1	Additions to header <utility> synopsis . . . . .	13
2.1.2	Class template <code>reference_wrapper</code> . . . . .	13
2.1.2.1	<code>reference_wrapper</code> construct/copy/destroy . . . . .	14
2.1.2.2	<code>reference_wrapper</code> access . . . . .	14
2.1.2.3	<code>reference_wrapper</code> invocation . . . . .	14
2.1.2.4	<code>reference_wrapper</code> helper functions . . . . .	14
2.1.2.5	implementation quantities . . . . .	15
2.2	Smart pointers . . . . .	15
2.2.1	Additions to header <memory> synopsis . . . . .	15
2.2.2	Class <code>bad_weak_ptr</code> . . . . .	16
2.2.3	Class template <code>shared_ptr</code> . . . . .	16
2.2.3.1	<code>shared_ptr</code> constructors . . . . .	18
2.2.3.2	<code>shared_ptr</code> destructor . . . . .	19
2.2.3.3	<code>shared_ptr</code> assignment . . . . .	20
2.2.3.4	<code>shared_ptr</code> modifiers . . . . .	20
2.2.3.5	<code>shared_ptr</code> observers . . . . .	20
2.2.3.6	<code>shared_ptr</code> comparison . . . . .	21
2.2.3.7	<code>shared_ptr</code> operators . . . . .	22
2.2.3.8	<code>shared_ptr</code> specialized algorithms . . . . .	22

2.2.3.9	shared_ptr casts . . . . .	22
2.2.3.10	get_deleter . . . . .	23
2.2.4	Class template weak_ptr . . . . .	23
2.2.4.1	weak_ptr constructors . . . . .	24
2.2.4.2	weak_ptr destructor . . . . .	25
2.2.4.3	weak_ptr assignment . . . . .	25
2.2.4.4	weak_ptr modifiers . . . . .	25
2.2.4.5	weak_ptr observers . . . . .	25
2.2.4.6	weak_ptr comparison . . . . .	26
2.2.4.7	weak_ptr specialized algorithms . . . . .	26
2.2.5	Class template enable_shared_from_this . . . . .	26
<b>3</b>	<b>Function objects and higher-order programming</b>	<b>29</b>
3.1	Function return types . . . . .	29
3.1.1	Additions to <functional> synopsis . . . . .	29
3.1.2	Class template result_of . . . . .	29
3.2	Member pointer adaptors . . . . .	30
3.2.1	Additions to header <functional> synopsis . . . . .	30
3.2.2	Function template mem_fn . . . . .	30
3.2.3	implementation quantities . . . . .	31
3.3	Function object binders . . . . .	31
3.3.1	Additions to header <functional> synopsis . . . . .	31
3.3.2	Class template is_bind_expression . . . . .	32
3.3.3	Class template is_placeholder . . . . .	32
3.3.4	Function template bind . . . . .	32
3.3.5	Placeholders . . . . .	35
3.3.6	Implementation quantities . . . . .	36
3.4	Polymorphic function wrappers . . . . .	36
3.4.1	Additions to <functional> synopsis . . . . .	36
3.4.2	Class bad_function_call . . . . .	36
3.4.2.1	bad_function_call constructor . . . . .	37
3.4.3	Class template function . . . . .	37
3.4.3.1	function construct/copy/destroy . . . . .	38
3.4.3.2	function modifiers . . . . .	40
3.4.3.3	function capacity . . . . .	40
3.4.3.4	function invocation . . . . .	40
3.4.3.5	specialized algorithms . . . . .	41

3.4.3.6	undefined operators . . . . .	41
3.4.4	Implementation quantities . . . . .	41
<b>4</b>	<b>Metaprogramming and type traits</b>	<b>42</b>
4.1	Requirements . . . . .	42
4.1.1	Unary type traits . . . . .	42
4.1.2	Binary type traits . . . . .	43
4.1.3	Transformation type traits . . . . .	43
4.2	Header <code>&lt;type_traits&gt;</code> synopsis . . . . .	43
4.3	Unary Type Traits . . . . .	45
4.3.1	Helper classes . . . . .	46
4.3.2	Primary Type Categories . . . . .	46
4.3.3	Composite type traits . . . . .	48
4.3.4	Type properties . . . . .	50
4.4	Relationships between types . . . . .	54
4.4.1	Type relationships . . . . .	54
4.5	Transformations between types . . . . .	55
4.5.1	Const-volatile modifications . . . . .	55
4.5.2	Reference modifications . . . . .	56
4.5.3	Array modifications . . . . .	57
4.5.4	Pointer modifications . . . . .	58
4.5.5	Other transformations . . . . .	58
4.6	Implementation requirements . . . . .	59
<b>5</b>	<b>Numerical facilities</b>	<b>60</b>
5.1	Random number generation . . . . .	60
5.1.1	Requirements . . . . .	60
5.1.2	Header <code>&lt;random&gt;</code> synopsis . . . . .	63
5.1.3	Class template <code>variate_generator</code> . . . . .	64
5.1.4	Random number engine class templates . . . . .	66
5.1.4.1	Class template <code>linear_congruential</code> . . . . .	67
5.1.4.2	Class template <code>mersenne_twister</code> . . . . .	68
5.1.4.3	Class template <code>subtract_with_carry</code> . . . . .	71
5.1.4.4	Class template <code>subtract_with_carry_01</code> . . . . .	73
5.1.4.5	Class template <code>discard_block</code> . . . . .	75
5.1.4.6	Class template <code>xor_combine</code> . . . . .	77
5.1.5	Engines with predefined parameters . . . . .	80

5.1.6	Class <code>random_device</code> . . . . .	80
5.1.7	Random distribution class templates . . . . .	82
5.1.7.1	Class template <code>uniform_int</code> . . . . .	82
5.1.7.2	Class <code>bernoulli_distribution</code> . . . . .	83
5.1.7.3	Class template <code>geometric_distribution</code> . . . . .	83
5.1.7.4	Class template <code>poisson_distribution</code> . . . . .	84
5.1.7.5	Class template <code>binomial_distribution</code> . . . . .	85
5.1.7.6	Class template <code>uniform_real</code> . . . . .	85
5.1.7.7	Class template <code>exponential_distribution</code> . . . . .	86
5.1.7.8	Class template <code>normal_distribution</code> . . . . .	87
5.1.7.9	Class template <code>gamma_distribution</code> . . . . .	88
5.2	Mathematical special functions . . . . .	88
5.2.1	Additions to header <code>&lt;cmath&gt;</code> synopsis . . . . .	88
5.2.1.1	associated Laguerre polynomials . . . . .	92
5.2.1.2	associated Legendre functions . . . . .	92
5.2.1.3	beta function . . . . .	92
5.2.1.4	(complete) elliptic integral of the first kind . . . . .	93
5.2.1.5	(complete) elliptic integral of the second kind . . . . .	93
5.2.1.6	(complete) elliptic integral of the third kind . . . . .	93
5.2.1.7	confluent hypergeometric functions . . . . .	94
5.2.1.8	regular modified cylindrical Bessel functions . . . . .	94
5.2.1.9	cylindrical Bessel functions (of the first kind) . . . . .	94
5.2.1.10	irregular modified cylindrical Bessel functions . . . . .	94
5.2.1.11	cylindrical Neumann functions . . . . .	95
5.2.1.12	(incomplete) elliptic integral of the first kind . . . . .	95
5.2.1.13	(incomplete) elliptic integral of the second kind . . . . .	95
5.2.1.14	(incomplete) elliptic integral of the third kind . . . . .	96
5.2.1.15	exponential integral . . . . .	96
5.2.1.16	Hermite polynomials . . . . .	96
5.2.1.17	hypergeometric functions . . . . .	96
5.2.1.18	Laguerre polynomials . . . . .	97
5.2.1.19	Legendre polynomials . . . . .	97
5.2.1.20	Riemann zeta function . . . . .	97
5.2.1.21	spherical Bessel functions (of the first kind) . . . . .	98
5.2.1.22	spherical associated Legendre functions . . . . .	98
5.2.1.23	spherical Neumann functions . . . . .	98
5.2.2	Additions to header <code>&lt;math.h&gt;</code> synopsis . . . . .	99

<b>6</b>	<b>Containers</b>	<b>100</b>
6.1	Tuple types	100
6.1.1	Header <code>&lt;tuple&gt;</code> synopsis	100
6.1.2	Class template <code>tuple</code>	102
6.1.2.1	Construction	102
6.1.2.2	Tuple creation functions	104
6.1.2.3	Valid expressions for tuple types	104
6.1.2.4	Element access	105
6.1.2.5	Equality and inequality comparisons	105
6.1.2.6	<code>&lt;</code> , <code>&gt;</code> comparisons	106
6.1.2.7	<code>&lt;=</code> and <code>&gt;=</code> comparisons	106
6.1.2.8	Input and output	107
6.1.2.9	Tuple formatting manipulators	107
6.1.3	Pairs	108
6.1.4	Implementation quantities	109
6.2	Fixed size array	109
6.2.1	Header <code>&lt;array&gt;</code> synopsis	109
6.2.2	Class template <code>array</code>	109
6.2.2.1	<code>array</code> constructors, copy, and assignment	111
6.2.2.2	<code>array</code> specialized algorithms	111
6.2.2.3	<code>array</code> size	112
6.2.2.4	Zero sized arrays	112
6.3	Unordered associative containers	112
6.3.1	Unordered associative container requirements	112
6.3.1.1	Exception safety guarantees	118
6.3.2	Additions to header <code>&lt;functional&gt;</code> synopsis	118
6.3.3	Class template <code>hash</code>	119
6.3.4	Unordered associative container classes	119
6.3.4.1	Header <code>&lt;unordered_set&gt;</code> synopsis	119
6.3.4.2	Header <code>&lt;unordered_map&gt;</code> synopsis	119
6.3.4.3	Class template <code>unordered_set</code>	120
6.3.4.3.1	<code>unordered_set</code> constructors	122
6.3.4.3.2	<code>unordered_set</code> swap	123
6.3.4.4	Class template <code>unordered_map</code>	123
6.3.4.4.1	<code>unordered_map</code> constructors	126
6.3.4.4.2	<code>unordered_map</code> element access	126
6.3.4.4.3	<code>unordered_map</code> swap	126

6.3.4.5	Class template <code>unordered_multiset</code> . . . . .	127
6.3.4.5.1	<code>unordered_multiset</code> constructors . . . . .	129
6.3.4.5.2	<code>unordered_multiset</code> swap . . . . .	130
6.3.4.6	Class template <code>unordered_multimap</code> . . . . .	130
6.3.4.6.1	<code>unordered_multimap</code> constructors . . . . .	132
6.3.4.6.2	<code>unordered_multimap</code> swap . . . . .	133
<b>7</b>	<b>Regular expressions</b> . . . . .	<b>134</b>
7.1	Definitions . . . . .	134
7.2	Requirements . . . . .	134
7.3	Regular expressions summary . . . . .	138
7.4	Header <code>&lt;regex&gt;</code> synopsis . . . . .	138
7.5	Namespace <code>std::regex_constants</code> . . . . .	146
7.5.1	Bitmask Type <code>syntax_option_type</code> . . . . .	146
7.5.2	Bitmask Type <code>regex_constants::match_flag_type</code> . . . . .	148
7.5.3	Implementation defined <code>syntax_type</code> . . . . .	149
7.5.4	Implementation defined <code>escape_syntax_type</code> . . . . .	151
7.5.5	Implementation defined <code>error_type</code> . . . . .	152
7.6	Class <code>bad_expression</code> . . . . .	153
7.7	Class template <code>regex_traits</code> . . . . .	153
7.8	Class template <code>basic_regex</code> . . . . .	156
7.8.1	<code>basic_regex</code> constants . . . . .	159
7.8.2	<code>basic_regex</code> constructors . . . . .	159
7.8.3	<code>basic_regex</code> capacity . . . . .	162
7.8.4	<code>basic_regex</code> assign . . . . .	162
7.8.5	<code>basic_regex</code> constant operations . . . . .	163
7.8.6	<code>basic_regex</code> locale . . . . .	163
7.8.7	<code>basic_regex</code> swap . . . . .	164
7.8.8	<code>basic_regex</code> non-member functions . . . . .	164
7.8.8.1	<code>basic_regex</code> non-member swap . . . . .	164
7.9	Class template <code>sub_match</code> . . . . .	164
7.9.1	<code>sub_match</code> members . . . . .	165
7.9.2	<code>sub_match</code> non-member operators . . . . .	165
7.10	Class template <code>match_results</code> . . . . .	170
7.10.1	<code>match_results</code> constructors . . . . .	172
7.10.2	<code>match_results</code> size . . . . .	172
7.10.3	<code>match_results</code> element access . . . . .	173

7.10.4	match_results reformatting	173
7.10.5	match_results allocator	174
7.10.6	match_results swap	174
7.11	Regular expression algorithms	174
7.11.1	regex_match	174
7.11.2	regex_search	176
7.11.3	regex_replace	178
7.12	Regular expression Iterators	179
7.12.1	Class template regex_iterator	179
7.12.1.1	regex_iterator constructors	180
7.12.1.2	regex_iterator comparisons	181
7.12.1.3	regex_iterator dereference	181
7.12.1.4	regex_iterator increment	181
7.12.2	Class template regex_token_iterator	182
7.12.2.1	regex_token_iterator constructors	184
7.12.2.2	regex_token_iterator comparisons	185
7.12.2.3	regex_token_iterator dereference	185
7.12.2.4	regex_token_iterator increment	185
7.13	Modified ECMAScript regular expression grammar	186
<b>8</b>	<b>Iterators</b>	<b>190</b>
8.1	New iterator requirements	190
8.1.1	Iterator access requirements	190
8.1.1.1	Readable iterator requirements	190
8.1.1.2	Writable iterator requirements	191
8.1.1.3	Swappable iterator requirements	191
8.1.1.4	Lvalue iterator requirements	191
8.1.2	Iterator traversal concepts	192
8.1.2.1	Incrementable iterator requirements	192
8.1.2.2	Single Pass Iterators	192
8.1.2.3	Forward Traversal Iterators	193
8.1.2.4	Bidirectional Traversal Iterators	193
8.1.2.5	Random Access Traversal Iterators	194
8.2	New iterator primitives	195
8.2.1	Additions to header <iterator> synopsis	195
8.2.2	New iterator traits and tags	195
8.3	Iterator adaptor and facade	198



8.3.1	Header <code>&lt;iterator_helper&gt;</code> synopsis	198
8.3.2	Iterator facade	199
8.3.2.1	Class template <code>iterator_facade</code>	199
8.3.2.2	<code>iterator_facade</code> requirements	201
8.3.2.3	<code>iterator_facade</code> operations	202
8.3.3	Iterator adaptor	204
8.3.3.1	Class template <code>iterator_adaptor</code>	204
8.3.3.2	<code>iterator_adaptor</code> requirements	205
8.3.3.3	<code>iterator_adaptor</code> public operations	206
8.3.3.4	<code>iterator_adaptor</code> protected member functions	206
8.3.3.5	<code>iterator_adaptor</code> private member functions	207
8.4	Specialized adaptors	207
8.4.1	Indirect iterator	208
8.4.1.1	Class template <code>indirect_iterator</code>	208
8.4.1.2	<code>indirect_iterator</code> requirements	208
8.4.1.3	<code>indirect_iterator</code> operations	209
8.4.2	Reverse iterator	210
8.4.2.1	Class template <code>reverse_iterator</code>	210
8.4.2.2	<code>reverse_iterator</code> requirements	211
8.4.2.3	<code>reverse_iterator</code> operations	211
8.4.3	Transform iterator	211
8.4.3.1	Class template <code>transform_iterator</code>	211
8.4.3.2	<code>transform_iterator</code> requirements	212
8.4.3.3	<code>transform_iterator</code> public operations	212
8.4.3.4	<code>transform_iterator</code> private operations	213
8.4.4	Filter iterator	213
8.4.4.1	Class template <code>filter_iterator</code>	213
8.4.4.2	<code>filter_iterator</code> requirements	214
8.4.4.3	<code>filter_iterator</code> operations	214
8.4.5	Counting iterator	215
8.4.5.1	Class template <code>counting_iterator</code>	215
8.4.5.2	<code>counting_iterator</code> requirements	216
8.4.5.3	<code>counting_iterator</code> operations	216
8.4.6	Function output iterator	217
8.4.6.1	Class template <code>function_output_iterator</code>	217
8.4.6.2	<code>function_output_iterator</code> requirements	217
8.4.6.3	<code>function_output_iterator</code> operations	218

8.4.6.4 `function_output_iterator::output_proxy` operations . . . 218

# Chapter 1

## General

**[tr.intro]**

This technical report describes extensions to the *C++ standard library* that is described in the International Standard for the C++ programming language [16].

This technical report is non-normative. Some of the library components in this technical report 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 report may never be standardized, and others may be standardized in a substantially changed form.

The goal of this technical report 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.

### 1.1 Method of description

**[tr.description]**

The structure of clauses in this technical report, the elements that make up the subclauses, and the editorial conventions used to describe library components, are the same as described in clause 17.3 of the C++ standard.

### 1.2 Categories of extensions

**[tr.intro.ext]**

This technical report describes four general categories of library extensions:

1. New requirement tables, such as the regular expression traits requirements in clause 7.2. These are not directly expressed as software; they specify the circumstances under which user-written components will interoperate with the components described in this technical report.
2. New library components (types and functions) that are declared in entirely new headers, such as the class templates in the `<unordered_set>` header 6.3.4.1.
3. New library components declared as additions to existing standard headers, such as the mathematical special functions added to the headers `<cmath>` and `<math.h>` in clauses 5.2.1 and 5.2.2

4. Additions to standard library components, such as the extensions to class `std::pair` in section 6.1.3.

The first three categories are collectively called *pure* extensions, and the last is called an *impure* extension. All extensions are assumed to be pure unless otherwise specified.

New headers are distinguished from extensions to existing headers by the title of the *synopsis* clause. In the first case the title is of the form “Header <foo> synopsis”, and the synopsis includes all namespace scope declarations contained in the header. In the second case the title is of the form “Additions to header <foo> synopsis” and the synopsis includes only the extensions, *i.e.* those namespace scope declarations that are not present in the C++ standard [16].

### 1.3 Namespaces and headers [tr.intro.namespaces]

Since the extensions described in this technical report are 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 report are declared in namespace `std::tr1`. [*Note:* Some components are declared in subnamespaces of namespace `std::tr1`. —*end note*]

Unless otherwise specified, reference to other entities described in this technical report are assumed to be qualified with `std::tr1::`, and references to entities described in the standard are assumed to be qualified with `std::`.

Even when an extension is specified as additions to standard headers (the third category in section 1.2), vendors should not simply add declarations to standard headers in a way that would be visible to users by default. [*Note:* That would fail to be standard conforming, because the new names, even within a namespace, could conflict with user macros. —*end note*] Users should be required to take explicit action to have access to library extensions.

It is recommended either that additional declarations in standard headers be protected with a macro that is not defined by default, or else that all extended headers, including both new headers and parallel versions of standard headers with nonstandard declarations, be placed in a separate directory that is not part of the default search path.

### 1.4 Caveat [tr.intro.caveat]

**This is an early draft. It's known to be incomplet and incorrekt, and it has lots of bad formatting.**

## Chapter 2

# General Utilities

[tr.util]

### 2.1 Reference wrappers

[tr.util.refwrap]

#### 2.1.1 Additions to header `<utility>` synopsis

[tr.util.refwrp.synopsis]

```
namespace tr1 {
    template <typename T> class reference_wrapper;
    template <typename T> ref(T&);
    template <typename T> cref(const T&);
}
```

#### 2.1.2 Class template `reference_wrapper`

[tr.util.refwrp.refwrp]

```
template <typename T> class reference_wrapper {
public :
    // types
    typedef T type;

    // construct/copy/destroy
    explicit reference_wrapper(T&) ;

    // access
    operator () const ;
    get() const ;

    // invocation
    template <typename T1, typename T2, ..., typename TN>
```

```

    typename result_of<T(T1, T2, ..., TN)>::type
    operator () (T1, T2, ..., TN) const ;
};

```

`reference_wrapper<T>` is a CopyConstructible and Assignable wrapper around a reference to an object of type `T`.

`reference_wrapper` defines the member type `result_type` in the following cases:

1. `T` is a function pointer, then `result_type` is the return type of `T`.
2. `T` is a pointer to member function, then `result_type` is the return type of `T`.
3. `T` is a class type with a member type `result_type`, then `result_type` is `T::result_type`.

### 2.1.2.1 `reference_wrapper` construct/copy/destroy [tr.util.refwrp.const]

```
explicit reference_wrapper(T& t);
```

**Effects:** Constructs a `reference_wrapper` object that stores a reference to `t`.

**Throws:** Does not throw.

### 2.1.2.2 `reference_wrapper` access [tr.util.refwrp.access]

```
operator () const;
```

**Returns:** The stored reference.

**Throws:** Does not throw.

```
get() const;
```

**Returns:** The stored reference.

**Throws:** Does not throw.

### 2.1.2.3 `reference_wrapper` invocation [tr.util.refwrp.invoke]

```

template <typename T1, typename T2, ..., typename TN>
    typename result_of<T(T1, T2, ..., TN)>::type
    operator ()(T1 a1, T2 a1, ..., TN aN) const ;

```

**Effects:** `f.get()(a1, a2, ..., aN)` **Returns:** The result of the expression `f.get()(a1, a2, ..., aN)`

### 2.1.2.4 `reference_wrapper` helper functions [tr.util.refwrp.helpers]

```
template <typename T> ref(T& t);
```

**Returns:** `reference_wrapper <T>(t)` **Throws:** Does not throw.

```
template <typename T> cref( const T& t);
```

**Returns:** `reference_wrapper <const T>(t)` **Throws:** Does not throw.

### 2.1.2.5 implementation quantities

[tr.util.refwrp.lim]

The maximum number of arguments that can be forwarded by `reference_wrapper` is implementation defined. This limit should be at least 10.

## 2.2 Smart pointers

[tr.util.smartptr]

### 2.2.1 Additions to header `<memory>` synopsis

[tr.util.smartptr.synopsis]

```
namespace tr1 {
    class bad_weak_ptr;

    template<class T> class shared_ptr;

    template<class T, class U>
        bool
        operator==(shared_ptr<T> const& a, shared_ptr<U> const& b);

    template<class T, class U>
        bool
        operator!=(shared_ptr<T> const& a, shared_ptr<U> const& b);

    template<class T, class U>
        bool
        operator<(shared_ptr<T> const& a, shared_ptr<U> const& b);

    template<class T>
        void swap(shared_ptr<T>& a, shared_ptr<T>& b);

    template<class T, class U>
        shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r);

    template<class T, class U>
        shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r);

    template<class E, class T, class Y>
        basic_ostream<E, T>&
        operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);
```

```

template<class D, class T>
    D * get_deleter(shared_ptr<T> const& p);

template<class T> class weak_ptr;

template<class T, class U>
    bool operator<(weak_ptr<T> const& a, weak_ptr<U> const& b);

template<class T>
    void swap(weak_ptr<T>& a, weak_ptr<T>& b);

template<class T> class enable_shared_from_this;
}

```

## 2.2.2 Class `bad_weak_ptr`

[tr.util.smartptr.weakptr]

```

namespace tr1 {
    class bad_weak_ptr: public exception
    {
    public:
        bad_weak_ptr();
    };
}

```

An exception of type `bad_weak_ptr` is thrown by the `shared_ptr` constructor taking a `weak_ptr`.

```
bad_weak_ptr();
```

**Postconditions:** `what()` returns "`tr1::bad_weak_ptr`".

**Throws:** nothing.

## 2.2.3 Class template `shared_ptr`

[tr.util.smartptr.shared]

The `shared_ptr` class template stores a pointer, usually obtained via `new`. `shared_ptr` implements semantics of shared ownership; the last remaining owner of the pointer is responsible for destroying the object, or otherwise releasing the resources associated with the stored pointer.

```

namespace tr1 {
    template<class T> class shared_ptr {
    public:
        typedef T element_type;

        // constructors
        shared_ptr();

```



```

template<class Y> explicit shared_ptr(Y * p);
template<class Y, class D> shared_ptr(Y * p, D d);
shared_ptr(shared_ptr const& r);
template<class Y> shared_ptr(shared_ptr<Y> const& r);
template<class Y> explicit shared_ptr(weak_ptr<Y> const& r);
template<class Y> explicit shared_ptr(auto_ptr<Y>& r);

// destructor
~shared_ptr();

// assignment
shared_ptr& operator=(shared_ptr const& r);
template<class Y> shared_ptr& operator=(shared_ptr<Y> const& r);
template<class Y> shared_ptr& operator=(auto_ptr<Y>& r);

// modifiers
void swap(shared_ptr& r);
void reset();
template<class Y> void reset(Y * p);
template<class Y, class D> void reset(Y * p, D d);

// observers
T* get() const;
T& operator*() const;
T* operator->() const;
long use_count() const;
bool unique() const;
operator @/unspecified-bool-type/() const;
};

// comparison
template<class T, class U>
bool
operator==(shared_ptr<T> const& a, shared_ptr<U> const& b);

template<class T, class U>
bool
operator!=(shared_ptr<T> const& a, shared_ptr<U> const& b);

template<class T, class U>
bool
operator<(shared_ptr<T> const& a, shared_ptr<U> const& b);

// other operators
template<class E, class T, class Y>
basic_ostream<E, T>&

```

```

    operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);

    // specialized algorithms
    template<class T>
        void swap(shared_ptr<T>& a, shared_ptr<T>& b);

    // casts
    template<class T, class U>
        shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r);

    template<class T, class U>
        shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r);

    // get_deleter
    template<class D, class T>
        D * get_deleter(shared_ptr<T> const& p);
}

```

`shared_ptr` is CopyConstructible, Assignable, and LessThanComparable, allowing its use in standard containers. `shared_ptr` is convertible to `bool`, allowing its use in boolean expressions and declarations in conditions.

[Example:

```

    if(shared_ptr<X> px = dynamic_pointer_cast<X>(py))
    {
        // do something with px
    }

```

—end example.]

### 2.2.3.1 `shared_ptr` constructors

[tr.util.smartptr.shared.const]

```

    shared_ptr();

```

**Effects:** Constructs an *empty* `shared_ptr`.

**Postconditions:** `use_count() == 0` && `get() == 0`.

**Throws:** nothing.

```

    template<class Y> explicit shared_ptr(Y * p);

```

**Requires:** `p` is convertible to `T *`. `Y` is a complete type. The expression `delete p` is well-formed, does not invoke undefined behavior, and does not throw exceptions.

**Effects:** Constructs a `shared_ptr` that *owns* the pointer `p`.

**Postconditions:** `use_count() == 1` && `get() == p`.

**Throws:** `bad_alloc` or an implementation-defined exception when a resource other than memory could not be obtained.

**Exception safety:** If an exception is thrown, `delete p` is called.

```
template<class Y, class D> shared_ptr(Y * p, D d);
```

**Requires:** `p` is convertible to `T *`. `D` is `CopyConstructible`. The copy constructor and destructor of `D` do not throw. The expression `d(p)` is well-formed, does not invoke undefined behavior, and does not throw exceptions.

**Effects:** Constructs a `shared_ptr` that *owns* the pointer `p` and the deleter `d`.

**Postconditions:** `use_count() == 1` && `get() == p`.

**Throws:** `bad_alloc` or an implementation-defined exception when a resource other than memory could not be obtained.

**Exception safety:** If an exception is thrown, `d(p)` is called.

```
shared_ptr(shared_ptr const& r);  
template<class Y> shared_ptr(shared_ptr<Y> const& r);
```

**Effects:** If `r` is *empty*, constructs an *empty* `shared_ptr`; otherwise, constructs a `shared_ptr` that *shares ownership* with `r`.

**Postconditions:** `get() == r.get()` && `use_count() == r.use_count()`.

**Throws:** nothing.

```
template<class Y> explicit shared_ptr(weak_ptr<Y> const& r);
```

**Effects:** Constructs a `shared_ptr` that *shares ownership* with `r` and stores a copy of the pointer stored in `r`.

**Postconditions:** `use_count() == r.use_count()`.

**Throws:** `bad_weak_ptr` when `r.expired()`.

**Exception safety:** If an exception is thrown, the constructor has no effect.

```
template<class Y> shared_ptr(auto_ptr<Y>& r);
```

**Requires:** `r.release()` is convertible to `T *`. `Y` is a complete type. The expression `delete r.release()` is well-formed, does not invoke undefined behavior, and does not throw exceptions.

**Effects:** Constructs a `shared_ptr` that stores and *owns* `r.release()`.

**Postconditions:** `use_count() == 1` && `r.get() == 0`.

**Throws:** `bad_alloc` or an implementation-defined exception when a resource other than memory could not be obtained.

**Exception safety:** If an exception is thrown, the constructor has no effect.

### 2.2.3.2 `shared_ptr` destructor

[tr.util.smartptr.shared.dest]

```
~shared_ptr();
```

**Effects:**

- If `*this` is *empty*, or *shares ownership* with another `shared_ptr` instance (`use_count() > 1`), there are no side effects.

- Otherwise, if *\*this owns* a pointer *p* and a deleter *d*, *d(p)* is called.
- Otherwise, *\*this owns* a pointer *p*, and *delete p* is called.

**Throws:** nothing.

### 2.2.3.3 `shared_ptr` assignment

[tr.util.smartptr.shared.assign]

```
shared_ptr& operator=(shared_ptr const& r);
template<class Y> shared_ptr& operator=(shared_ptr<Y> const& r);
template<class Y> shared_ptr& operator=(auto_ptr<Y>& r);
```

**Effects:** Equivalent to `shared_ptr(r).swap(*this)`.

**Returns:** *\*this*.

**Notes:** The use count updates caused by the temporary object construction and destruction are not considered observable side effects, and the implementation is free to meet the effects (and the implied guarantees) via different means, without creating a temporary. In particular, in the example:

```
shared_ptr<int> p(new int);
shared_ptr<void> q(p);
p = p;
q = p;
```

both assignments may be no-ops.

### 2.2.3.4 `shared_ptr` modifiers

[tr.util.smartptr.shared.mod]

```
void swap(shared_ptr& r);
```

**Effects:** Exchanges the contents of *\*this* and *r*.

**Throws:** nothing.

```
void reset();
```

**Effects:** Equivalent to `shared_ptr().swap(*this)`.

```
template<class Y> void reset(Y * p);
```

**Effects:** Equivalent to `shared_ptr(p).swap(*this)`.

```
template<class Y, class D> void reset(Y * p, D d);
```

**Effects:** Equivalent to `shared_ptr(p, d).swap(*this)`.

### 2.2.3.5 `shared_ptr` observers

[tr.util.smartptr.shared.obs]

```
T * get() const;
```

**Returns:** the stored pointer.

**Throws:** nothing.

```
T& operator*() const;
```

**Requires:** `get() != 0`.

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

**Throws:** nothing.

**Notes:** When `T` is `void`, the return type of this member function is unspecified, and an attempt to instantiate it renders the program ill-formed.

```
T * operator->() const;
```

**Requires:** `get() != 0`.

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

**Throws:** nothing.

```
long use_count() const;
```

**Returns:** the number of `shared_ptr` objects, `*this` included, that *share ownership* with `*this`, or 0 when `*this` is *empty*.

**Throws:** nothing.

**Notes:** `use_count()` is not necessarily efficient. Use only for debugging and testing purposes, not for production code.

```
bool unique() const;
```

**Returns:** `use_count() == 1`.

**Throws:** nothing.

**Notes:** `unique()` may be faster than `use_count()`. If you are using `unique()` to implement copy on write, do not rely on a specific value when `get() == 0`.

```
operator @/unspecified-bool-type/ () const;
```

**Returns:** an unspecified value that, when used in boolean contexts, is equivalent to `get() != 0`.

**Throws:** nothing.

**Notes:** This conversion operator allows *shared\_ptr* objects to be used in boolean contexts, like `if (p && p->valid())`. The actual target type is typically a pointer to a member function, avoiding many of the implicit conversion pitfalls.

### 2.2.3.6 `shared_ptr` comparison

[tr.util.smartptr.shared.cmp]

```
template<class T, class U>
bool operator==(shared_ptr<T> const& a,
                shared_ptr<U> const& b);
```

**Returns:** `a.get() == b.get()`.

**Throws:** nothing.

```
template<class T, class U>
bool operator!=(shared_ptr<T> const& a, shared_ptr<U> const& b);
```

**Returns:** `a.get() != b.get()`.

**Throws:** nothing.

```
template<class T, class U>
bool operator<(shared_ptr<T> const& a, shared_ptr<U> const& b);
```

**Returns:** an unspecified value such that

- `operator<` is a strict weak ordering as described in section 25.3 [`lib.alg.sorting`];
- under the equivalence relation defined by `operator<, !(a < b) && !(b < a)`, two `shared_ptr` instances are equivalent if and only if they *share ownership*.

**Throws:** nothing.

**Notes:** Allows `shared_ptr` objects to be used as keys in associative containers.

### 2.2.3.7 `shared_ptr` operators

[`tr.util.smartptr.shared.op`]

```
template<class E, class T, class Y>
basic_ostream<E, T>&
operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);
```

**Effects:** `os << p.get()`;

**Returns:** `os`.

### 2.2.3.8 `shared_ptr` specialized algorithms

[`tr.util.smartptr.shared.spec`]

```
template<class T>
void swap(shared_ptr<T>& a, shared_ptr<T>& b);
```

**Effects:** Equivalent to `a.swap(b)`.

**Throws:** nothing.

### 2.2.3.9 `shared_ptr` casts

[`tr.util.smartptr.shared.cast`]

```
template<class T, class U>
shared_ptr<T> static_pointer_cast(shared_ptr<U> const& r);
```

**Requires:** The expression `static_cast<T*>(r.get())` is well-formed.

**Returns:** If `r` is *empty*, an *empty* `shared_ptr<T>`; otherwise, a `shared_ptr<T>` object that stores `static_cast<T*>(r.get())` and *shares ownership* with `r`.

**Throws:** nothing.

**Notes:** the seemingly equivalent expression `shared_ptr<T>(static_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice.

```
template<class T, class U>
shared_ptr<T> dynamic_pointer_cast(shared_ptr<U> const& r);
```

**Requires:** The expression `dynamic_cast<T*>(r.get())` is well-formed and does not invoke undefined behavior.

**Returns:**

- When `dynamic_cast<T*>(r.get())` returns a nonzero value, a `shared_ptr<T>` object that stores a copy of it and *shares ownership* with `r`;
- Otherwise, an *empty* `shared_ptr<T>` object.

**Throws:** nothing.

**Notes:** the seemingly equivalent expression `shared_ptr<T>(dynamic_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice.

### 2.2.3.10 `get_deleter`

[[tr.util.smartptr.getdeleter](#)]

```
template<class D, class T>
D * get_deleter(shared_ptr<T> const& p);
```

**Returns:** If `*this` owns a deleter `d` of type cv-unqualified `D`, returns `&d`; otherwise returns `0`.

**Throws:** nothing.

## 2.2.4 Class template `weak_ptr`

[[tr.util.smartptr.weak](#)]

The `weak_ptr` class template stores a “weak reference” to an object that’s already managed by a `shared_ptr`. To access the object, a `weak_ptr` can be converted to a `shared_ptr` using the member function `lock`.

```
namespace tr1 {
    template<class T> class weak_ptr {

    public:
        typedef T element_type;

        // constructors
        weak_ptr();
        template<class Y> weak_ptr(shared_ptr<Y> const& r);
```

```

weak_ptr(weak_ptr const& r);
template<class Y> weak_ptr(weak_ptr<Y> const& r);

// destructor
~weak_ptr();

// assignment
weak_ptr& operator=(weak_ptr const& r);
template<class Y>
    weak_ptr& operator=(weak_ptr<Y> const& r);
template<class Y>
    weak_ptr& operator=(shared_ptr<Y> const& r);

// modifiers
void swap(weak_ptr& r);
void reset();

// observers
long use_count() const;
bool expired() const;
shared_ptr<T> lock() const;
};

// comparison
template<class T, class U>
    bool operator<(weak_ptr<T> const& a, weak_ptr<U> const& b);

// specialized algorithms
template<class T>
    void swap(weak_ptr<T>& a, weak_ptr<T>& b);
}

```

`weak_ptr` is `CopyConstructible`, `Assignable`, and `LessThanComparable`, allowing its use in standard containers.

#### 2.2.4.1 `weak_ptr` constructors

[tr.util.smartptr.weak.const]

```
weak_ptr();
```

**Effects:** Constructs an *empty* `weak_ptr`.

**Postconditions:** `use_count() == 0`.

**Throws:** nothing.

```

template<class Y> weak_ptr(shared_ptr<Y> const& r);
weak_ptr(weak_ptr const& r);
template<class Y> weak_ptr(weak_ptr<Y> const& r);

```



**Effects:** If `r` is *empty*, constructs an *empty* `weak_ptr`; otherwise, constructs a `weak_ptr` that *shares ownership* with `r` and stores a copy of the pointer stored in `r`.

**Postconditions:** `use_count() == r.use_count()`.

**Throws:** nothing.

#### 2.2.4.2 `weak_ptr` destructor

[tr.util.smartptr.weak.dest]

```
~weak_ptr();
```

**Effects:** Destroys this `weak_ptr` but has no effect on the object its stored pointer points to.

**Throws:** nothing.

#### 2.2.4.3 `weak_ptr` assignment

[tr.util.smartptr.weak.assign]

```
weak_ptr& operator=(weak_ptr const& r);  
template<class Y> weak_ptr& operator=(weak_ptr<Y> const& r);  
template<class Y> weak_ptr& operator=(shared_ptr<Y> const& r);
```

**Effects:** Equivalent to `weak_ptr(r).swap(*this)`.

**Throws:** nothing.

**Notes:** The implementation is free to meet the effects (and the implied guarantees) via different means, without creating a temporary.

#### 2.2.4.4 `weak_ptr` modifiers

[tr.util.smartptr.weak.mod]

```
void swap(weak_ptr& r);
```

**Effects:** Exchanges the contents of `*this` and `r`.

**Throws:** nothing.

```
void reset();
```

**Effects:** Equivalent to `weak_ptr().swap(*this)`.

#### 2.2.4.5 `weak_ptr` observers

[tr.util.smartptr.weak.obs]

```
long use_count() const;
```

**Returns:** 0 if `*this` is *empty*; otherwise, the number of `shared_ptr` instances that *share ownership* with `*this`.

**Throws:** nothing.

**Notes:** `use_count()` is not necessarily efficient. Use only for debugging and testing purposes, not for production code.

```
bool expired() const;
```

**Returns:** `use_count() == 0`.

**Throws:** nothing.

**Notes:** `expired()` may be faster than `use_count()`.

```
shared_ptr<T> lock() const;
```

**Returns:** `expired() ? shared_ptr<T>(): shared_ptr<T>(*this)`.

**Throws:** nothing.

#### 2.2.4.6 weak\_ptr comparison

[tr.util.smartptr.weak.cmp]

```
template<class T, class U>  
bool operator<(weak_ptr<T> const& a, weak_ptr<U> const& b);
```

**Returns:** an unspecified value such that

- `operator<` is a strict weak ordering as described in section 25.3 [lib.alg.sorting];
- under the equivalence relation defined by `operator<, !(a < b) && !(b < a)`, two `weak_ptr` instances are equivalent if and only if they *share ownership*.

**Throws:** nothing.

**Notes:** Allows `weak_ptr` objects to be used as keys in associative containers.

#### 2.2.4.7 weak\_ptr specialized algorithms

[tr.util.smartptr.weak.spec]

```
template<class T>  
void swap(weak_ptr<T>& a, weak_ptr<T>& b)
```

**Effects:** Equivalent to `a.swap(b)`.

**Throws:** nothing.

### 2.2.5 Class template enable\_shared\_from\_this

[tr.util.smartptr.enab]

A class can derive from the `enable_shared_from_this` class template, passing itself as a template parameter, to inherit the `shared_from_this` member functions that obtain a *shared\_ptr* instance pointing to this.

[Example:

```
struct X: public enable_shared_from_this<X>  
{  
};  
  
int main()
```

```

{
  shared_ptr<X> p(new X);
  shared_ptr<X> q = p->shared_from_this();
  assert(p == q);
  assert(!(p < q ) \&\& !(q < p)); // p and q share ownership
}

```

—end example.]

```

namespace tr1 {
  template<class T> class enable_shared_from_this {
  protected:
    enable_shared_from_this();
    enable_shared_from_this(enable_shared_from_this const&);
    enable_shared_from_this&
    operator=(enable_shared_from_this const&);
    ~enable_shared_from_this();
  public:
    shared_ptr<T> shared_from_this();
    shared_ptr<T const> shared_from_this() const;
  };
}

```

```

enable_shared_from_this<T>::enable_shared_from_this();
enable_shared_from_this<T>::enable_shared_from_this(
  enable_shared_from_this<T> const&);

```

**Effects:** Constructs an `enable_shared_from_this<T>` instance.

**Throws:** nothing.

```

enable_shared_from_this<T>&
enable_shared_from_this<T>
  ::operator=(enable_shared_from_this<T> const&);

```

**Returns:** `*this`.

**Throws:** nothing.

```

enable_shared_from_this<T>::~~enable_shared_from_this();

```

**Effects:** Destroys `*this`.

**Throws:** nothing.

```

template<class T> shared_ptr<T>
enable_shared_from_this<T>::shared_from_this();

template<class T> shared_ptr<T const>
enable_shared_from_this<T>::shared_from_this() const;

```

**Requires:** `enable_shared_from_this<T>` is an accessible base class of `T`. `*this` is a subobject of an instance `t` of type `T`. There is at least one `shared_ptr` instance `p` that *owns* `&t`.

**Returns:** A `shared_ptr<T>` instance `r` that *shares ownership with* `p`.

**Postconditions:** `r.get() == this`.

[*Note:* a typical implementation is shown below:

```
template<class T> class enable_shared_from_this
{
private:

    weak_ptr<T> __weak_this;

protected:

    enable_shared_from_this() {}
    enable_shared_from_this(enable_shared_from_this const &) {}
    enable_shared_from_this &
    operator=(enable_shared_from_this const &) { return *this; }
    ~enable_shared_from_this() {}

public:

    shared_ptr<T> shared_from_this()
        { return shared_ptr<T>(__weak_this); }
    shared_ptr<T const> shared_from_this() const
        { return shared_ptr<T const>(__weak_this); }
};
```

The three *shared\_ptr* constructors that create unique pointers should detect the presence of an *enable-shared\_from\_this* base and assign the newly created *shared\_ptr* to its *\_\_weak\_this* member. —end note.]

## Chapter 3

# Function objects and higher-order programming [tr.func]

### 3.1 Function return types [tr.func.ret]

#### 3.1.1 Additions to <functional> synopsis [tr.func.ret.synopsis]

```
namespace tr1 {  
    template <typename FunctionCallTypes> class result_of;  
}
```

#### 3.1.2 Class template `result_of` [tr.func.ret.ret]

```
template <typename FunctionCallTypes>  
class result_of {  
public :  
    // types  
    typedef unspecified type;  
};
```

Given an lvalue `f` of type `F` and lvalues `t1`, `t2`, ..., `tN` of types `T1`, `T2`, ..., `TN`, respectively, the `type` member type defines the result type of the expression `f(t1, t2, ..., tN)`.

The implementation may determine the member type `type` via any means that produces the exact type of the expression `f(t1, t2, ..., tN)` for the given types. [*Note*: The intent is that implementations are permitted to use special compiler hooks —*end note*]

If the implementation cannot determine the type of the expression  $f(t_1, t_2, \dots, t_N)$ , or if the expression is ill-formed, the implementation shall use the following process to determine the member type `type`:

1. If  $F$  is a function type, `type` is the return type of the function type  $F$ .
2. If  $F$  is a member function type, `type` is the return type of the member function type  $F$ .
3. If  $F$  is a function object defined by the standard library, the method of determining `type` is unspecified.
4. If  $F$  is a class type with a member type `result_type`, `type` is  $F::result\_type$ .
5. If  $F$  is a class type with no member named `result_type` or if  $F::result\_type$  is not a type:
  - (a) If  $N=0$  (no arguments), `type` is `void`.
  - (b) If  $N>0$ , `type` is  $F::result<F(T_1, T_2, \dots, T_N)>::type$ .
6. Otherwise, the program is ill-formed.

## 3.2 Member pointer adaptors

[tr.func.memfn]

### 3.2.1 Additions to header `<functional>` synopsis

[tr.func.memfn.synopsis]

```
namespace tr1 {
    template<class R, class T>
        unspecified mem_fn(R T::* pm);
}
```

### 3.2.2 Function template `mem_fn`

[tr.func.memfn.memfn]

`mem_fn(&X::f)`, where `f` is a member function of `X`, returns an object through which `&X::f` can be called given a pointer, a smart pointer, an iterator, or a reference to `X` followed by the argument list required for `X::f`, if any. The returned object is `CopyConstructible` and `Assignable`, its copy constructor and assignment operator do not throw exceptions, and it has a nested typedef `result_type` defined as the return type of `f`.

`mem_fn(&X::m)`, where `m` is a data member of `X`, returns an object through which a reference to `&X::m` can be obtained given a pointer, a smart pointer, an iterator, or a reference to `X`. The returned object is `CopyConstructible` and `Assignable`, its copy constructor and assignment operator do not throw exceptions, and it has a nested typedef `result_type` defined as either `M` or `M const &`, where `M` is the type of `m`.

```
template<class R, class T>
    unspecified mem_fn(R T::* pm);
```

**Returns:**

- When `pm` is a pointer to a member function taking `n` arguments, a function object `f` such that the expression `f(t, a1, ..., an)` is equivalent to `(t.*pm)(a1, ..., an)` when `t` is an lvalue of type `T` or derived from `T`, `((*t).*pm)(a1, ..., an)` otherwise.
- When `pm` is a pointer to a data member, a function object `f` such that the expression `f(t)` is equivalent to `t.*pm` when `t` is an lvalue of type `T` or derived from `T`, `(*t).*pm` otherwise.

**Throws:** nothing.**Notes:** Implementations are allowed to implement `mem_fn` as a set of overloaded function templates.**3.2.3 implementation quantities****[tr.func.memfn.lim]**The maximum value for `n` is implementation defined.**3.3 Function object binders****[tr.func.bind]****3.3.1 Additions to header `<functional>` synopsis****[tr.func.bind.synopsis]**

```

template<class T> struct is_bind_expression;
template<class T> struct is_placeholder;

template<class F>
    unspecified bind(F f);
template<class R, class F>
    unspecified bind(F f);

template<class F, class A1>
    unspecified bind(F f, A1 a1);
template<class R, class T, class A1>
    unspecified bind(R T::* pm, A1 a1);
template<class R, class F, class A1>
    unspecified bind(F f, A1 a1);

// for all positive integers n in [2, N)

template<class F, class A1, ..., class An>
    unspecified bind(F f, A1 a1, ..., An an);
template<class R, class T, class A1, ..., class An>
    unspecified bind(R T::* pmf, A1 a1, ..., An an);
template<class R, class F, class A1, ..., class An>
    unspecified bind(F f, A1 a1, ..., An an);

```

```

namespace placeholders {
    // M is the implementation-defined number of placeholders
    extern unspecified _1;
    extern unspecified _2;
        .
        .
        .
    extern unspecified _M;
}

```

### 3.3.2 Class template `is_bind_expression`

[tr.func.bind.isbind]

```

namespace tr1 {
    template<class T> struct is_bind_expression {
        static const bool value = unspecified;
    };
}

```

`is_bind_expression` can be used to detect function objects generated by `bind`. `bind` uses `is_bind_expression` to detect subexpressions. The template can be specialized by users to indicate that a type should be treated as a subexpression in a `bind` call.

```

static const bool value;

```

true if T is a type returned from `bind`, false otherwise.

### 3.3.3 Class template `is_placeholder`

[tr.func.bind.isplace]

```

namespace tr1 {
    template<class T> struct is_placeholder {
        static const int value = unspecified;
    };
}

```

`is_placeholder` can be used to detect the standard placeholders `_1`, `_2`, and so on. `bind` uses `is_placeholder` to detect placeholders. The template can be specialized by users to indicate a placeholder type.

```

static const int value;

```

value is *N* if T is the type of `tr1::placeholders::_N`, 0 otherwise.

### 3.3.4 Function template `bind`

[tr.func.bind.bind]

The function  $\lambda(x)$  is defined as `x.get()` when `x` is of type `reference_wrapper<F>` for some `F`, `x` otherwise.



The function  $\mu(x, v_1, \dots, v_m)$   $x$  is of type  $X$ , and  $k$  is `is_placeholder<X>::value`, is defined as:

- `x.get()`, when  $X$  is a `reference_wrapper<T>` for some  $T$ ;
- $v_k$ , when  $k \neq 0$ ;
- $x(v_1, \dots, v_m)$ , when `is_bind_expression<X>::value` is `true`;
- $x$  otherwise.

A function object  $f$  of type  $F$  is called *simple*, if  $f$  is a pointer to a function with C++ linkage or `F::result_type` is a type.

The maximum number of supported arguments ( $N$  in the synopsis) is implementation defined. Implementations are allowed to define additional, more specialized, `bind` overloads, or to fold the pointer to member overload into the general function template, as long as the behavior of the `bind` calls is unchanged.

Given a list of arguments  $a_1, a_2, \dots, a_n$ , a function object  $h$  as returned from a call to `bind`, and the definition:

```
struct forward {
    template<typename T1, typename T2, ..., typename Tn>
        void operator()(T1&, T2&, ..., Tn&);
};
```

If the expression `forward()(a1, a2, ..., an)` is well-formed, then the expression `h(a1, a2, ..., an)` must be well-formed and must have the same argument passing semantics as `forward()(a1, a2, ..., an)`. [Note - Implementations are encouraged to support argument forwarding for non-const temporaries - end note]

```
template<class F> unspecified bind(F f);
```

**Requires:**  $F$  must be `CopyConstructible`.  $\lambda(f)()$  must be a valid expression. If  $f$  is not a *simple* function object, the behavior is implementation defined.

**Returns:** A function object  $g$  of an unspecified `CopyConstructible` type  $G$  such that the expression  $g(v_1, \dots, v_m)$  is equivalent to  $\lambda(f)()$ . The type of the expression  $g(v_1, \dots, v_m)$  is `result_of<R()>::type` where  $R$  is the type of  $\lambda(f)$ . If the function application is made via a cv-qualified reference to, or copy of,  $g$ , the same cv-qualifiers are applied to  $f$  before the evaluation. When  $f$  is a pointer to a function with C++ linkage, `G::result_type` is defined as the return type of the  $f$ . When `F::result_type` is defined, `G::result_type` is defined as the same type.

**Throws:** nothing unless the copy constructor of  $f$  throws an exception.

**Notes:** Implementations are allowed to impose an upper limit on  $m$  (typically one more than the number of supported placeholders). It is implementation defined whether  $G$  is `Assignable` or `DefaultConstructible`.

```
template<class R, class F> unspecified bind(F f);
```

**Requires:**  $F$  must be `CopyConstructible`.  $\lambda(f)()$  must be a valid expression convertible to  $R$ .

**Returns:** A function object  $g$  of an unspecified `CopyConstructible` type  $G$  such that the expression  $g(v_1, \dots, v_m)$  is equivalent to  $\lambda(f)()$ , implicitly converted to  $R$ . If the function application is

made via a cv-qualified reference to, or copy of,  $g$ , the same cv-qualifiers are applied to  $f$  before the evaluation.  $G::\text{result\_type}$  is defined as  $R$ .

**Throws:** nothing unless the copy constructor of  $f$  throws an exception.

**Notes:** Implementations are allowed to impose an upper limit on  $m$ . It is implementation defined whether  $G$  is Assignable or DefaultConstructible.

```
template<class F, class A1> unspecified bind(F f, A1 a1);
```

**Requires:**  $F$  and  $A1$  must be CopyConstructible.  $\lambda(f)(w1)$  must be a valid expression for some value  $w1$ . If  $f$  is not a *simple* function object, the behavior is implementation defined.

**Returns:** A function object  $g$  of an unspecified CopyConstructible type  $G$  such that the expression  $g(v_1, \dots, v_m)$  is equivalent to  $\lambda(f)(\mu(a_1, v_1, \dots, v_m))$ . The type of the expression  $g(v_1, \dots, v_m)$  is  $\text{result\_of}\langle R(T)\rangle::\text{type}$  where  $R$  is the type of  $\lambda(f)$  and  $T$  is the type of  $\mu(a_1, v_1, \dots, v_m)$ . If the function application is made via a cv-qualified reference to, or copy of,  $g$ , the same cv-qualifiers are applied to  $f$  and  $a_1$  before the evaluation. When  $f$  is a pointer to a function with C++ linkage,  $G::\text{result\_type}$  is defined as the return type of the  $f$ . When  $F::\text{result\_type}$  is defined,  $G::\text{result\_type}$  is defined as the same type.

**Throws:** nothing unless the copy constructors of  $f$  or  $a_1$  throw an exception.

**Notes:** Implementations are allowed to impose an upper limit on  $m$ . It is implementation defined whether  $G$  is Assignable or DefaultConstructible.

```
template<class R, class T, class A1>
unspecified bind(R T::* pm, A1 a1);
```

**Requires:**  $pm$  must be a pointer to data member or pointer to a member function taking no arguments.

**Returns:**  $\text{bind}(\text{mem\_fn}(pm), a_1)$ .

```
template<class R, class F, class A1>
unspecified bind(F f, A1 a1);
```

**Requires:**  $F$  and  $A1$  must be CopyConstructible.  $\lambda(f)(w1)$ , for some value  $w1$ , must be a valid expression convertible to  $R$ .

**Returns:** A function object  $g$  of an unspecified CopyConstructible type  $G$  such that the expression  $g(v_1, \dots, v_m)$  is equivalent to  $\lambda(f)(\mu(a_1, v_1, \dots, v_m))$ , implicitly converted to  $R$ . If the function application is made via a cv-qualified reference to, or copy of,  $g$ , the same cv-qualifiers are applied to  $f$  and  $a_1$  before the evaluation.  $G::\text{result\_type}$  is defined as  $R$ .

**Throws:** nothing unless the copy constructors of  $f$  or  $a_1$  throw an exception.

**Notes:** Implementations are allowed to impose an upper limit on  $m$ . It is implementation defined whether  $G$  is Assignable or DefaultConstructible.

```
template<class F, class A1, ..., class An>
unspecified bind(F f, A1 a1, ..., An an);
```

**Requires:**  $F$  and  $A_i$  must be CopyConstructible.  $\lambda(f)(w_1, \dots, w_n)$  must be a valid expression for some values  $w_i$ . If  $f$  is not a *simple* function object, the behavior is implementation defined.

**Returns:** A function object  $g$  of an unspecified CopyConstructible type  $G$  such that the expression  $g(v_1, \dots, v_m)$  is equivalent to  $\lambda(f)(\mu(a_1, v_1, \dots, v_m), \dots, \mu(a_n, v_1, \dots, v_m))$ .

The type of the expression  $g(v_1, \dots, v_m)$  is `result_of<R(T1, T2, ..., Tn)>::type` where  $R$  is the type of  $\lambda(f)$  and each  $T_i$  is the type of  $\mu(a_i, v_1, \dots, v_m)$ .

If the function application is made via a cv-qualified reference to, or copy of,  $g$ , the same cv-qualifiers are applied to  $f$  and  $a_i$  before the evaluation. When  $f$  is a pointer to a function with C++ linkage, `G::result_type` is defined as the return type of the  $f$ . When `F::result_type` is defined, `G::result_type` is defined as the same type.

**Throws:** nothing unless the copy constructors of  $f$  or  $a_i$  throw an exception.

**Notes:** Implementations are allowed to impose an upper limit on  $m$ . It is implementation defined whether  $G$  is Assignable or DefaultConstructible.

```
template<class R, class T, class A1, ..., class An>
    unspecified bind(R T::* pmf, A1 a1, ..., An an);
```

**Requires:**  $pmf$  must be a pointer to a member function taking  $n-1$  arguments.

**Returns:** `bind(mem_fn(pmf), a1, ..., an)`.

```
template<class R, class F, class A1, ..., class An>
    unspecified bind(F f, A1 a1, ..., An an);
```

**Requires:**  $F$  and  $A_i$  must be CopyConstructible.  $\lambda(f)(w_1, \dots, w_n)$ , for some values  $w_i$ , must be a valid expression convertible to  $R$ .

**Returns:** A function object  $g$  of an unspecified CopyConstructible type  $G$  such that the expression  $g(v_1, \dots, v_m)$  is equivalent to  $\lambda(f)(\mu(a_1, v_1, \dots, v_m), \dots, \mu(a_n, v_1, \dots, v_m))$ , implicitly converted to  $R$ . If the function application is made via a cv-qualified reference to, or copy of,  $g$ , the same cv-qualifiers are applied to  $f$  and  $a_i$  before the evaluation. `G::result_type` is defined as  $R$ .

**Throws:** nothing unless the copy constructors of  $f$  or  $a_i$  throw an exception.

**Notes:** Implementations are allowed to impose an upper limit on  $m$ . It is implementation defined whether  $G$  is Assignable or DefaultConstructible.

### 3.3.5 Placeholders

[tr.func.bind.place]

```
namespace tr1 {
    namespace placeholders {
        extern unspecified _1;
        extern unspecified _2;
        extern unspecified _3;
        // implementation defined number of additional placeholders
    }
}
```

All placeholder types are DefaultConstructible and CopyConstructible, and their default constructors and copy constructors do not throw. It is implementation defined whether placeholder types are Assignable. Assignable placeholders' copy assignment operators do not throw exceptions.

### 3.3.6 Implementation quantities

[tr.func.bind.limits]

The number of placeholder types in namespace `tr1::placeholders`, and the number of arguments that can be passed to a `bind` function object, are implementation defined. Recommended minimum values are:

- Number of placeholder types in namespace `tr1::placeholders` — 9.
- Number of arguments that can be passed to a `bind` function object — 10.

## 3.4 Polymorphic function wrappers

[tr.func.wrap]

### 3.4.1 Additions to `<functional>` synopsis

[tr.func.wrap.synopsis]

```
class bad_function_call;

template<typename Function,
         typename Allocator = std::allocator<void> >
class function;

template<typename Function, typename Allocator>
void swap(function<Function, Allocator>&,
          function<Function, Allocator>&);

template<typename Function1, typename Allocator1,
         typename Function2, typename Allocator2>
void operator==(const function<Function1, Allocator1>&,
               const function<Function2, Allocator2>&);

template<typename Function1, typename Allocator1,
         typename Function2, typename Allocator2>
void operator!=(const function<Function1, Allocator1>&,
               const function<Function2, Allocator2>&);
```

### 3.4.2 Class `bad_function_call`

[tr.func.wrap.badcall]

The `bad_function_call` exception class is thrown primarily when a polymorphic adaptor is invoked but is empty (see 20.3.10).

```
namespace tr1 {
    class bad_function_call : public std::runtime_error
    {
    public:
        // [tr.func.wrap.badcall.const] constructor
    };
};
```

```

        bad_function_call();
    };
}

```

### 3.4.2.1 `bad_function_call` constructor

[tr.func.wrap.badcall.const]

```
bad_function_call();
```

**Effects:** constructs a `bad_function_call` object.

### 3.4.3 Class template function

[tr.func.wrap.func]

The library provides polymorphic wrappers that generalize the notion of a function pointer. Wrappers can store, copy, and call arbitrary function objects given a function signature (denoted by a set of argument types and a return type), allowing functions to be first-class objects.

A function object `f` of type `F` is *Callable* given a set of argument types `T1`, `T2`, ..., `TN` and a return type `R`, if the appropriate following function definition is well-formed:

```

// If F is not a pointer to member function
R callable(F& f, T1 t1, T2 t2, ..., TN tN)
{
    return static_cast<R>(f(t1, t2, ..., tN));
}

// If F is a pointer to member function
R callable(F f, T1 t1, T2 t2, ..., TN tN)
{
    return static_cast<R>((*t1).*f)(t2, t3, ..., tN);
}

```

The function class template is a function object type whose call signature is defined by its first template argument (a function type).

```

namespace tr1 {
    // Function type R (T1, T2, ..., TN), 0 ≤ N ≤ Nmax
    template<typename Function,
            typename Allocator = std::allocator<void> >
    class function
    : public unary_function<R, T1> // iff N == 1
    : public binary_function<R, T1, T2> // iff N == 2
    {
    public:
        typedef R          result_type;
        typedef Allocator  allocator_type;

        // [tr.func.wrap.func.con] construct/copy/destroy

```

```

explicit function(const Allocator& = Allocator());
function(const function&);
template<typename F>
    function(F, const Allocator& = Allocator());
template<typename F>
    function(reference_wrapper<F>,
             const Allocator& = Allocator());
function& operator=(const function&);
template<typename F>
    function& operator=(F);
template<typename F>
    function& operator=(reference_wrapper<F>);
~function();

// [tr.func.wrap.func.mod] function modifiers
void swap(function&);
void clear();

// [tr.func.wrap.func.cap] function capacity
bool empty() const;
operator implementation-defined() const;

// [tr.func.wrap.func.inv] function invocation
R operator()(T1, T2, ..., TN) const;
};

// [tr.func.wrap.func.alg] specialized algorithms
template<typename Function1, typename Allocator1,
         typename Function2, typename Allocator2>
void swap(function<Function1, Allocator1>&,
          function<Function2, Allocator2>&);

// [tr.func.wrap.func.undef] undefined operators
template<typename Function1, typename Allocator1,
         typename Function2, typename Allocator2>
void operator==(const function<Function1, Allocator1>&,
                const function<Function2, Allocator2>&);
template<typename Function1, typename Allocator1,
         typename Function2, typename Allocator2>
void operator!=(const function<Function1, Allocator1>&,
                const function<Function2, Allocator2>&);
}

```

### 3.4.3.1 function construct/copy/destroy

[tr.func.wrap.func.con]

```
explicit function(const Allocator& = Allocator());
```

**Postconditions:** `this->empty()`.

**Throws:** will not throw.

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

**Postconditions:** `this->empty()` if `f.empty()`; otherwise, `*this` targets a copy of `f`.

**Throws:** will not throw if the target of `f` is a function pointer or a function object passed via `reference_wrapper`. Otherwise, may throw `bad_alloc` or any exception thrown by the copy constructor of the stored function object.

```
template<typename F>
function(F f, const Allocator& = Allocator());
```

**Requires:** `f` is a callable function object for argument types `T1, T2, ..., TN` and return type `R`.

**Postconditions:** `this->empty()` if any of the following hold:

- `f` is a NULL function pointer.
- `f` is a NULL member function pointer.
- `f` is an instance of the function class template and `f.empty()`

Otherwise, `*this` targets a copy of `f` if `f` is not a pointer to member function, and targets a copy of `mem_fun(f)` if `f` is a pointer to member function.

**Throws:** will not throw when `f` is a function pointer. Otherwise, may throw `bad_alloc` or any exception thrown by `F`'s copy constructor.

```
template<typename F> function(reference_wrapper<F> f,
                             const Allocator& = Allocator());
```

**Requires:** `f.get()` is a callable function object for argument types `T1, T2, ..., TN` and return type `R`.

**Postconditions:** `!this->empty()` and `*this` targets `f.get()`.

**Throws:** will not throw.

**Rationale:** a potential alternative would be to replace the `reference_wrapper` argument with an argument taking a function object pointer. This route was not taken because `reference_wrapper` is a general solution stating the user's intention to pass a reference to an object instead of a copy.

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

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

**Returns:** `*this`

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

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

**Returns:** `*this`

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

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

**Returns:** `*this`

**Throws:** will not throw.

```
~function();
```

**Effects:** if `!this->empty()`, destroys the target of `this`.

### 3.4.3.2 function modifiers

[tr.func.wrap.func.mod]

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

**Effects:** interchanges the targets of `*this` and `other` and the allocators of `*this` and `other`.

**Throws:** will not throw.

```
void clear();
```

**Effects:** If `!this->empty()`, deallocates current target.

**Postconditions:** `this->empty()`.

### 3.4.3.3 function capacity

[tr.func.wrap.func.cap]

```
bool empty() const
```

**Returns:** `true` if the function object has a target, `false` otherwise.

**Throws:** will not throw.

```
operator @\textit{implementation-defined}@() const
```

**Returns:** if `!this->empty()`, returns a value that will evaluate `true` in a boolean context; otherwise, returns a value that will evaluate `false` in a boolean context. The value type returned shall not be convertible to `int`.

**Throws:** will not throw.

**Notes:** This conversion can be used in contexts where a `bool` is expected (e.g., an `if` condition); however, implicit conversions (e.g., to `int`) that can occur with `bool` are not allowed, eliminating some sources of user error. The suggested implementation technique is to use a member function pointer whose class type is private to the `function` instantiation.

### 3.4.3.4 function invocation

[tr.func.wrap.func.inv]

```
R operator()(T1 t1, T2 t2, ..., TN tN) const
```



**Effects:**  $f(t_1, t_2, \dots, t_N)$ , where  $f$  is the target of `*this`.

**Returns:** nothing, if  $R$  is `void`; otherwise, the return value of the call to  $f$ .

**Throws:** `bad_function_call` if `this->empty()`; otherwise, any exception thrown by the wrapped function object.

### 3.4.3.5 specialized algorithms

[tr.func.wrap.func.alg]

```
template<typename Function, typename Allocator>
    void swap(function<Function, Allocator>& f1,
              function<Function, Allocator>& f2);
```

**Effects:** `f1.swap(f2)`;

### 3.4.3.6 undefined operators

[tr.func.wrap.func.undef]

```
template<typename Function1, typename Allocator1,
         typename Function2, typename Allocator2>
    void operator==(const function<Function1, Allocator1>&,
                   const function<Function2, Allocator2>&);
```

This function shall be left undefined.

**Rationale:** the boolean-like conversion opens a loophole whereby two `function` instances can be compared via `==`. This undefined `void` operator `==` closes the loophole and ensures a compile-time or link-time error.

```
template<typename Function1, typename Allocator1,
         typename Function2, typename Allocator2>
    void operator!=(const function<Function1, Allocator1>&,
                   const function<Function2, Allocator2>&);
```

This function shall be left undefined.

**Rationale:** the boolean-like conversion opens a loophole whereby two `function` instances can be compared via `!=`. This undefined `void` operator `!=` closes the loophole and ensures a compile-time or link-time error.

## 3.4.4 Implementation quantities

[tr.func.wrap.limits]

$N_{\max}$ , the maximum number of function call arguments supported by class `function` is implementation defined. Implementations are encouraged to support at least 10 arguments.

## Chapter 4

# Metaprogramming and type traits

## [tr.meta]

This clause describes components used by C++ programs, particularly in templates, to: support the widest possible range of types, optimise template code usage, detect type related user errors, and perform type inference and transformation at compile time.

### 4.1 Requirements

[tr.meta.rqmts]

#### 4.1.1 Unary type traits

[tr.meta.rqmts.unary]

In table 4.1, X is a class template that is a unary type trait and T is any arbitrary type.

Table 4.1: UnaryTypeTrait requirements

Expression	Return Type	Requirement
<code>X&lt;T&gt;::value_type</code>	An integral type	The type of <code>X&lt;T&gt;::value</code> .
<code>X&lt;T&gt;::value</code>	<code>value_type</code>	An integral constant expression that is true if T has the specified trait, and false otherwise.
<code>X&lt;T&gt;::type</code>	<code>integral_</code> <code>constant&lt;value_type,</code> <code>value&gt;</code>	A type for use in situations where a typename is more appropriate than a value. The class template <code>integral_constant</code> is declared in <code>&lt;type_traits&gt;</code> .

*continued from previous page*

<code>X&lt;T&gt;::type t = X&lt;T&gt;()</code>		Both lvalues of type <code>X&lt;T&gt;</code> <code>const&amp;</code> and rvalues of type <code>X&lt;T&gt;</code> are implicitly convertible to <code>X&lt;T&gt;::type</code> .
--	--	--

### 4.1.2 Binary type traits

[tr.meta.rqmts.binary]

In table 4.2, `X` is a class template that is a binary type trait and `T` and `U` are any arbitrary types.

Table 4.2: BinaryTypeTrait requirements

Expression	Return Type	Requirement
<code>X&lt;T,U&gt;::value</code>	<code>bool</code>	An integral constant expression that is true if <code>T</code> is related to <code>U</code> by the relation specified, and false otherwise.
<code>X&lt;T,U&gt;::value_type</code>	<code>bool</code>	A type that is the type of <code>X&lt;T,U&gt;::value</code> , this is always <code>bool</code> for <code>BinaryTypeTraits</code> .
<code>X&lt;T,U&gt;::type</code>	<code>integral_constant&lt;bool, value&gt;</code>	A type for use in situations where a typename is more appropriate than a value. The class template <code>integral_constant</code> is declared in <code>&lt;type_traits&gt;</code> .
<code>X&lt;T,U&gt;::type t = X&lt;T,U&gt;()</code>		Both lvalues of type <code>X&lt;T,U&gt;</code> <code>const&amp;</code> and rvalues of type <code>X&lt;T,U&gt;</code> are implicitly convertible to <code>X&lt;T,U&gt;::type</code> .

### 4.1.3 Transformation type traits

[tr.meta.rqmts.trans]

In table 4.3, `X` is a class template that is a transformation trait and `T` is any arbitrary type.

Table 4.3: TransformationTrait requirements

Expression	Requirement
<code>X&lt;T&gt;::type</code>	The result is a type that is the result of applying transformation <code>X</code> to type <code>T</code> .

## 4.2 Header `<type_traits>` synopsis

[tr.meta.type.synop]

```
namespace tr1{
```

```

// helper class:
template <class T, T v> struct integral_constant;
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;

// Primary type categories:
template <class T> struct is_void;
template <class T> struct is_integral;
template <class T> struct is_floating_point;
template <class T> struct is_array;
template <class T> struct is_pointer;
template <class T> struct is_reference;
template <class T> struct is_member_object_pointer;
template <class T> struct is_member_function_pointer;
template <class T> struct is_enum;
template <class T> struct is_union;
template <class T> struct is_class;
template <class T> struct is_function;

// composite type categories:
template <class T> struct is_arithmetic;
template <class T> struct is_fundamental;
template <class T> struct is_object;
template <class T> struct is_scalar;
template <class T> struct is_compound;
template <class T> struct is_member_pointer;

// type properties:
template <class T> struct is_const;
template <class T> struct is_volatile;
template <class T> struct is_pod;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct has_trivial_constructor;
template <class T> struct has_trivial_copy;
template <class T> struct has_trivial_assign;
template <class T> struct has_trivial_destructor;
template <class T> struct has_nothrow_constructor;
template <class T> struct has_nothrow_copy;
template <class T> struct has_nothrow_assign;
template <class T> struct is_signed;
template <class T> struct is_unsigned;
template <class T> struct alignment_of;
template <class T> struct has_virtual_destructor;

```

```

// type relations:
template <class T, class U> struct is_same;
template <class From, class To> struct is_convertible;
template <class Base, class Derived> struct is_base_of;

// const-volatile modifications:
template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;
template <class T> struct add_const;
template <class T> struct add_volatile;
template <class T> struct add_cv;

// reference modifications:
template <class T> struct remove_reference;
template <class T> struct add_reference;

// array modifications:
template <class T> struct remove_dimension;
template <class T> struct remove_all_dimensions;

// pointer modifications:
template <class T> struct remove_pointer;
template <class T> struct add_pointer;

// Other transformations
template <class T> struct aligned_storage;

} // namespace tr1

```

### 4.3 Unary Type Traits

[tr.meta.unary]

This sub-clause contains templates that may be used to query the properties of a type at compile time.

All of the class templates defined in this header satisfy the `UnaryTypeTrait` requirements.

For all of the class templates declared in this clause, all members declared `static const` shall be defined in such a way that they are usable as integral constant expressions.

For all of the class templates `X` declared in this clause, both rvalues of type `X const` and lvalues of type `X const&` shall be implicitly convertible to `X::type`. Whether this is accomplished by inheritance or by a member operator is unspecified. For exposition only the class templates in this clause are shown with a member conversion operator: `operator type() const`, which shall return `type()` in all cases.

For all of the class templates `X` declared in this clause, instantiating that template with a template-argument that is a class template specialization, may result in the implicit instantiation of the template argument if and only if the semantics of `X` require that the argument must be a complete type.

### 4.3.1 Helper classes

[tr.meta.unary.help]

```
template <class T, T v>
struct integral_constant
{
    static const T          value = v;
    typedef T              value_type;
    typedef integral_constant<T,v> type;
};
typedef integral_constant<bool, true> true_type;
typedef integral_constant<bool, false> false_type;
```

The class template `integral_constant` and its associated typedefs `true_type` and `false_type` are for use in situations where a type rather than a value is required.

### 4.3.2 Primary Type Categories

[tr.meta.unary.cat]

The primary type categories correspond to the descriptions given in section 3.9 of the C++ standard.

For any given type `T`, exactly one of the primary type categories shall have its member `value` evaluate to `true`.

For any given type `T`, the result of applying one of these templates to `T`, and to *cv-qualified* `T` shall yield the same result.

Undefined behaviour results if any C++ program adds specializations for any of the class templates defined in this clause.

```
template <class T> struct is_void {
    static const bool value = implementation_defined;
    typedef bool      value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};
```

`value`: defined to be `true` if `T` is `void` or a *cv-qualified* `void`. Otherwise defined to be `false`.

```
template <class T> struct is_integral {
    static const bool value = implementation_defined;
    typedef bool      value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};
```

`value`: defined to be `true` if `T` is an integral type (3.9.1). Otherwise defined to be `false`.

```
template <class T> struct is_floating_point {
    static const bool value = implementation_defined;
    typedef bool      value_type;
    typedef integral_constant<value_type,value> type;
};
```

```

    operator type() const;
};

```

value : defined to be true if T is a floating point type (3.9.1). Otherwise defined to be false.

```

template <class T> struct is_array {
    static const bool value = implementation_defined;
    typedef bool value_type;
| typedef| integral_constant<value_type,value> type;operator type() const;;

template <class T> struct is_pointer {
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is a pointer type (3.9.2), this includes all function pointer types, but not pointers to members or member functions. Otherwise defined to be false.

```

template <class T> struct is_reference {
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is a reference type (3.9.2), this includes all reference to function types. Otherwise defined to be false.

```

template <class T> struct is_member_object_pointer {
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is a pointer to a data member. Otherwise defined to be false.

```

template <class T> struct is_member_function_pointer {
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is a pointer to a member function. Otherwise defined to be false .

```

template <class T> struct is_enum {
    static const bool value = implementation_defined;

```

```

    typedef bool                                     value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is an enumeration type (3.9.2). Otherwise defined to be false.

```

template <class T> struct is_union {
    static const bool value = implementation_defined;
    typedef bool                                     value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is a union type (3.9.2). Otherwise defined to be false.

```

template <class T> struct is_class {
    static const bool value = implementation_defined;
    typedef bool                                     value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is a class type (3.9.2), in this context unions are not considered to be class types. Otherwise defined to be false.

```

template <class T> struct is_function {
    static const bool value = implementation_defined;
    typedef bool                                     value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is a function type (3.9.2). Otherwise defined to be false.

### 4.3.3 Composite type traits

[tr.meta.unary.comp]

These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in section 3.9.

For any given type T, the result of applying one of these templates to T, and to *cv-qualified* T shall yield the same result.

Undefined behaviour results if any C++ program adds specializations for any of the class templates defined in this clause.

```

template <class T> struct is_arithmetic {
    static const bool value =
        is_integral<T>::value || is_floating_point<T>::value;
    typedef bool                                     value_type;
};

```



```

    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is an arithmetic type (3.9.1). Otherwise defined to be false.

```

template <class T> struct is_fundamental{
    static const bool value =
        is_integral<T>::value
        || is_floating_point<T>::value
        || is_void<T>::value;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is a fundamental type (3.9.1). Otherwise defined to be false.

```

template <class T> struct is_object{
    static const bool value =
        !(is_function<T>::value
        || is_reference<T>::value
        || is_void<T>::value);
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is an object type (3.9). Otherwise defined to be false.

```

template <class T> struct is_scalar{
    static const bool value =
        is_arithmetic<T>::value
        || is_enum<T>::value
        || is_pointer<T>::value
        || is_member_pointer<T>::value;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is a scalar type (3.9). Otherwise defined to be false.

```

template <class T> struct is_compound{
    static const bool value = !is_fundamental<T>::value;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is a compound type (3.9.2). Otherwise defined to be false.

```
template <class T> struct is_member_pointer {
    static const bool value =
        is_member_object_pointer<T>::value
        || is_member_function_pointer<T>::value;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};
```

value : defined to be true if T is a pointer to a member or member function. Otherwise defined to be false.

#### 4.3.4 Type properties

[tr.meta.unary.prop]

These templates provide access to some of the more important properties of types; they reveal information which is available to the compiler, but which would not otherwise be detectable in C++ code.

Except where specified, it is undefined whether any of these templates have any full or partial specialisations defined. It is permitted for the user to specialise any of these templates on a user-defined type, provided the semantics of the specialisation match those given for the template in its description.

```
template <class T> struct is_const{
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};
```

value: defined to be true if T is const-qualified (3.9.3). Otherwise defined to be false.

```
template <class T> struct is_volatile{
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};
```

value : defined to be true if T is volatile-qualified (3.9.3). Otherwise defined to be false.

```
template <class T> struct is_pod{
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};
```

**Preconditions:** template argument T shall be a complete type.

value : defined to be true if T is a POD type (3.9). Otherwise defined to be false.

```

template <class T> struct is_empty{
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

**Preconditions:** template argument T shall be a complete type.

value : defined to be true if T is an empty class (10). Otherwise defined to be false.

```

template <class T> struct is_polymorphic{
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

**Preconditions:** template argument T shall be a complete type.

value : defined to be true if T is a polymorphic class (10.3). Otherwise defined to be false.

```

template <class T> struct is_abstract{
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

**Preconditions:** template argument T shall be a complete type.

value : defined to be true if T is a abstract class (10.4). Otherwise defined to be false.

```

template <class T> struct has_trivial_constructor{
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

**Preconditions:** template argument T shall be a complete type.

value : defined to be true if the default constructor for T is trivial(12.1). Otherwise defined to be false.

```

template <class T> struct has_trivial_copy{
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

**Preconditions:** template argument T shall be a complete type.

value : defined to be true if the copy constructor for T is trivial (12.8). Otherwise defined to be false.

```
template <class T> struct has_trivial_assign{
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};
```

**Preconditions:** template argument T shall be a complete type.

value : defined to be true if the assignment operator for T is trivial (12.8). Otherwise defined to be false.

```
template <class T> struct has_trivial_destructor{
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};
```

**Preconditions:** template argument T shall be a complete type.

value : defined to be true if the destructor for T is trivial (12.4). Otherwise defined to be false.

```
template <class T> struct has_nothrow_constructor{
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};
```

**Preconditions:** template argument T shall be a complete type.

value : defined to be true if the default constructor for T has an empty exception specification, or can otherwise be deduced never to throw an exception. Otherwise defined to be false.

```
template <class T> struct has_nothrow_copy{
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};
```

**Preconditions:** template argument T shall be a complete type.

value : defined to be true if the copy constructor for T has an empty exception specification, or can otherwise be deduced never to throw an exception. Otherwise defined to be false .

```
template <class T> struct has_nothrow_assign{
    static const bool value = implementation_defined;
```

```

    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

**Preconditions:** template argument T shall be a complete type.

value : defined to be true if the assignment operator for T has an empty exception specification, or can otherwise be deduced never to throw an exception. Otherwise defined to be false.

```

template <class T> struct is_signed{
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is a signed integral type (3.9.1). Otherwise defined to be false .

```

template <class T> struct is_unsigned{
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};

```

value : defined to be true if T is an unsigned integral type (3.9.1). Otherwise defined to be false.

```

template <class T> struct alignment_of {
    static const std::size_t value = implementation_defined;
    typedef std::size_t value_type;
    typedef integral_constant<value_type,value> type;
    operator type()const;
};

```

value: An implementation-defined integer value representing the number of bytes of the alignment of objects of type T; an object of type T may be allocated at an address that is a multiple of its alignment (3.9).

```

template <class T> struct has_virtual_destructor {
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<bool,value> type;
    operator type()const;
};

```

value: true if type T has a virtual destructor (12.4) otherwise false. [Note: An implementation that cannot determine whether a type has a virtual destructor, e.g. a pure library implementation with no compiler support, should return false. —end note]

## 4.4 Relationships between types

[tr.meta.rel]

All of the templates in this header satisfy the BinaryTypeTrait requirements.

For all of the class templates declared in this clause, all members declared static const shall be defined in such a way that they are usable as integral constant expressions.

For all of the class templates X declared in this clause, both rvalues of type X const and lvalues of type X const& shall be implicitly convertible to X::type. Whether this is accomplished by inheritance or by a member operator is unspecified. For exposition only the class templates in this clause are shown with a member conversion operator, operator type() const, which shall return —type()— in all cases.

### 4.4.1 Type relationships

[tr.meta.rel.rel]

```
template <class T, class U> struct is_same{
    static const bool value = false;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type()const;
};

template <class T> struct is_same<T,T>{
    static const bool value = true;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type()const;
};
```

value: defined to be true if T and U are the same type. Otherwise defined to be false.

```
template <class From, class To> struct is_convertible {
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type()const;
};
```

value: defined to be true only if an imaginary lvalue of type From is implicitly-convertible to type To (4.0). Otherwise defined to be false. Special conversions involving string-literals and null-pointer constants are not considered (4.2, 4.10 and 4.11). No function-parameter adjustments (8.3.5) are made to type To when determining whether From is convertible to To: this implies that if type To is a function type or an array type, then value must necessarily evaluate to false.

The expression is\_convertible<From,To>::value is ill-formed if:

- Type From, is a void or incomplete type (3.9).
- Type To, is an incomplete, void or abstract type (3.9).
- The conversion is ambiguous, for example if type From has multiple base classes of type To (10.2).

- Type `To` is of class type and the conversion would invoke a non-public constructor of `To` (11.0 and 12.3.1).
- Type `From` is of class type and the conversion would invoke a non-public conversion operator of `From` (11.0 and 12.3.2).

```
template < class Base, class Derived> struct is_base_of {
    static const bool value = implementation_defined;
    typedef bool value_type;
    typedef integral_constant<value_type,value> type;
    operator type()const;
};
```

**Preconditions:** template arguments `Base` and `Derived` shall both be complete types.

`value`: defined to be true only if type `Base` is a base class of type `Derived` (10). Otherwise defined to be false.

## 4.5 Transformations between types [tr.meta.trans]

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

All of the templates in this header satisfy the `TransformationTrait` requirements.

### 4.5.1 Const-volatile modifications [tr.meta.trans.cv]

```
template <class T> struct remove_const{
    typedef T type;
};
template <class T> struct remove_const<T const>{
    typedef T type;
};
```

`type` : defined to be a type that is the same as `T`, except that any top level `const`-qualifier has been removed. For example: `remove_const<const volatile int>::type` evaluates to `volatile int`, whereas `remove_const<const int*>` is `const int*`.

```
template <class T> struct remove_volatile{
    typedef T type;
};
template <class T> struct remove_volatile<T volatile>{
    typedef T type;
};
```

`type` : defined to be a type that is the same as `T`, except that any top level `volatile`-qualifier has been removed. For example: `remove_const<const volatile int>::type` evaluates to `const int`, whereas `remove_const<volatile int*>` is `volatile int*`.

```

template <class T> struct remove_cv{
    typedef typename remove_const<
        typename remove_volatile<T>::type
    >::type
    type;
};

```

type : defined to be a type that is the same as T, except that any top level cv-qualifiers have been removed. For example: `remove_cv<const volatile int>::type` evaluates to `int`, where as `remove_cv<const volatile int*>` is `const volatile int*`.

```

template <class T> struct add_const{
    typedef T const type;
};

```

type : if T is a reference, function, or top level const-qualified type, then the same type as T, otherwise T const.

```

template <class T> struct add_volatile{
    typedef T volatile type;
};

```

type: if T is a reference, function, or top level const-qualified type, then the same type as T, otherwise T volatile.

```

template <class T> struct add_cv{
    typedef typename add_const<
        typename add_volatile<T>::type
    >::type type;
};

```

type: the same type as `add_const< add_volatile<T>::type >::type`.

## 4.5.2 Reference modifications

[tr.meta.trans.ref]

```

template <class T> struct remove_reference{
    typedef T type;
};
template <class T> struct remove_reference<T&>{
    typedef T type;
};

```

type : defined to be a type that is the same as T, except any reference qualifier has been removed.

```

template <class T> struct add_reference{
    typedef T& type;
};
template <class T> struct add_reference<T&>{

```



```

    typedef T& type;
};

```

type: if T is a reference type, then T, otherwise T& .

### 4.5.3 Array modifications

[tr.meta.trans.arr]

```

template <class T> struct remove_dimension{
    typedef T type;
};
template <class T, std::size_t N> struct remove_dimension<T[N]>{
    typedef T type;
};
template <class T> struct remove_dimension<T[]>{
    typedefs T type;
};

```

type: for a type ‘array of U’, the resulting type is U; for any other type V, the resulting type is V.

*Note:* for multidimensional arrays, only the first array dimension is removed. For a type ‘array of const U’, the resulting type is const U. —end note]

[example

```

// the following assertions should all hold:
assert((is_same<remove_dimension<int>::type, int>::value));
assert((is_same<remove_dimension<int[2]>::type, int>::value));
assert((is_same<remove_dimension<int[2][3]>::type, int[3]>::value));
assert((is_same<remove_dimension<int[][3]>::type, int[3]>::value));

```

—end example]

```

template <class T> struct remove_all_dimensions{
    typedef T type;
};
template <class T, std::size_t N> struct remove_all_dimensions<T[N]>{
    typedef typename remove_all_dimensions<T>::type type;
};
template <class T> struct remove_all_dimensions<T[]>{
    typedef typename remove_all_dimensions<T>::type type;
};

```

type: for a type ‘multi-dimensional array of U’, the resulting type is U; for any other type V, the resulting type is V.

[example

```

// the following assertions should all hold:
assert((is_same<remove_all_dimensions<int>::type, int>::value));
assert((is_same<remove_all_dimensions<int[2]>::type, int>::value));

```

```

assert((is_same<remove_all_dimensions<int[2][3]>::type, int>::value));
assert((is_same<remove_all_dimensions<int[][3]>::type, int>::value));

```

—end example]

#### 4.5.4 Pointer modifications

[tr.meta.trans.ptr]

```

template <class T> struct remove_pointer{
    typedef T type;
};
template <class T> struct remove_pointer<T*>{
    typedef T type;
};
template <class T> struct remove_pointer<T* const>{
    typedef T type;
};
template <class T> struct remove_pointer<T* volatile>{
    typedef T type;
};
template <class T> struct remove_pointer<T* const volatile>{
    typedef T type;
};

```

`type`: defined to be a type that is the same as `T`, except any top level indirection has been removed. Note: pointers to members are left unchanged by `remove_pointer`.

```

template <class T> struct add_pointer{
    typedef typename remove_dimension<
        typename remove_reference<T>::type
    >::type*
    type;
};

```

`type`: defined to be a type that is the same as `remove_reference<T>::type*` if `T` is a reference type, otherwise `T*`.

#### 4.5.5 Other transformations

[tr.meta.trans.other]

```

template <std::size_t Len, std::size_t Align> struct aligned_storage{
    typedef unspecified type;
};

```

`type`: an implementation defined POD type with size `Len` and alignment `Align`, and suitable for use as uninitialized storage for any object of a type whose size is `Len` and whose alignment is `Align`.

## 4.6 Implementation requirements

[tr.meta.req]

The behaviour of all the class templates defined in `<type_traits>` shall conform to the specifications given, except where noted below.

[*Note*: The latitude granted to implementers in this clause is temporary, and is expected to be removed in future revisions of this document. —*end note*]

If there is no means by which the implementation can differentiate between class and union types, then the class templates `is_class` and `is_union` need not be provided.

If there is no means by which the implementation can detect polymorphic types, then the class template `is_polymorphic` need not be provided.

If there is no means by which the implementation can detect abstract types, then the class template `is_abstract` need not be provided.

It is unspecified under what circumstances, if any, `is_empty<T>::value` evaluates to true.

It is unspecified under what circumstances, if any, `is_pod<T>::value` evaluates to true, except that, for all types `T`:

```
is_pod<T>::value == is_pod<remove_dimension<T>::type>::value
is_pod<T>::value == is_pod<T const volatile>::value
is_pod<T>::value >= (is_scalar<T>::value || is_void<T>::value)
```

It is unspecified under what circumstances, if any, `has_trivial_*<T>::value` evaluates to true, except that:

```
has_trivial_*<T>::value ==
    has_trivial_*<remove_dimension<T>::type>::value
has_trivial_*<T>::value >=
    is_pod<T>::value
```

It is unspecified under what circumstances, if any, `has_nothrow_*<T>::value` evaluates to true.

There are trait templates whose semantics do not require their argument(s) to be completely defined, nor does such completeness in any way affect the exact definition of the traits class template specializations. However, in the absence of compiler support these traits cannot be implemented without causing implicit instantiation of their arguments; in particular: `is_class`, `is_enum`, and `is_scalar`. For these templates, it is unspecified whether their template argument(s) are implicitly instantiated when the traits class is itself instantiated.

## Chapter 5

# Numerical facilities

[tr.num]

## 5.1 Random number generation

[tr.rand]

This subclause defines a facility for generating random numbers.

### 5.1.1 Requirements

[tr.rand.req]

In table 5.1,  $X$  denotes a uniform random number generator class returning objects of type  $T$ ,  $u$  is a value of  $X$ , and  $v$  is a (possibly `const`) value of  $X$ .

Table 5.1: Uniform random number generator requirements

expression	Return Type	pre/post-condition	complexity
<code>X::result_type</code>	$T$	$T$ is an arithmetic type [basic.fundamental]	compile-time
<code>u()</code>	$T$	—	amortized constant
<code>v.min()</code>	$T$	Returns a value that is less than or equal to all values potentially returned by <code>operator()</code> . The return value of this function shall not change during the lifetime of $v$ .	constant
<code>v.max()</code>	$T$	If <code>std::numeric_limits&lt;T&gt;::is_integer</code> , returns a value that is greater than or equal to all values potentially returned by <code>operator()</code> , otherwise, returns a value that is strictly greater than all values potentially returned by <code>operator()</code> . In any case, the return value of this function shall not change during the lifetime of $v$ .	constant

In table 5.2,  $X$  denotes a pseudo-random number engine class returning objects of type  $T$ ,  $t$  is a value of  $T$ ,  $u$  is a value of  $X$ ,  $v$  is an lvalue of  $X$ ,  $it1$  is an lvalue and  $it2$  is a (possibly const) value of an input iterator type  $It$  having an unsigned integral value type,  $x, y$  are (possibly const) values of  $X$ ,  $os$  is convertible to an lvalue of type `std::ostream`, and  $is$  is convertible to an lvalue of type `std::istream`.

A pseudo-random number engine  $x$  has a state  $x(i)$  at any given time. The specification of each pseudo-random number engines defines the size of its state in multiples of the size of its `result_type`, given as an integral constant expression.

Table 5.2: Pseudo-random number engine requirements (in addition to uniform random number generator, CopyConstructible, and Assignable)

expression	Return Type	pre/post-condition	complexity
<code>X()</code>	—	creates an engine with the same initial state as all other default-constructed engines of type $X$ in the program.	$\mathcal{O}(\text{size of state})$
<code>X(it1, it2)</code>	—	creates an engine with the initial state given by the range $[it1, it2)$ . $it1$ is advanced by the size of state. If the size of the range $[it1, it2)$ is insufficient, leaves $it1 == it2$ and throws <code>std::invalid_argument</code> .	$\mathcal{O}(\text{size of state})$
<code>u.seed()</code>	<code>void</code>	post: $u == X()$	$\mathcal{O}(\text{size of state})$
<code>u.seed(it1, it2)</code>	<code>void</code>	post: If there are sufficiently many values in $[it1, it2)$ to initialize the state of $u$ , then $u == X(it1, it2)$ . Otherwise, $it1 == it2$ , throws <code>std::invalid_argument</code> , and further use of $u$ (except destruction) is undefined until a <code>seed</code> member function has been executed without throwing an exception.	$\mathcal{O}(\text{size of state})$
<code>u()</code>	$T$	given the state $u(i)$ of the engine, computes $u(i+1)$ , sets the state to $u(i+1)$ , and returns some output dependent on $u(i+1)$	amortized constant
<code>x == y</code>	<code>bool</code>	<code>==</code> is an equivalence relation. The current state $x(i)$ of $x$ is equal to the current state $y(j)$ of $y$ .	$\mathcal{O}(\text{size of state})$
<code>x != y</code>	<code>bool</code>	<code>!(x == y)</code>	$\mathcal{O}(\text{size of state})$

*continued from previous page*

<code>os &lt;&lt; x</code>	<code>std::ostream&amp;</code>	writes the textual representation of the state $x(i)$ of $x$ to <code>os</code> , with <code>os.fmtflags</code> set to <code>ios_base::dec ios_base::fixed ios_base::left</code> and the fill character set to the space character. In the output, adjacent numbers are separated by one or more space characters. post: The <code>os.fmtflags</code> and fill character are unchanged.	$\mathcal{O}(\text{size of state})$
<code>is &gt;&gt; v</code>	<code>std::istream&amp;</code>	sets the state $v(i)$ of $v$ as determined by reading its textual representation from <code>is</code> . post: The <code>is.fmtflags</code> are unchanged.	$\mathcal{O}(\text{size of state})$

Additional requirements: The complexity of both copy construction and assignment is  $\mathcal{O}(\text{size of state})$ .

In table 5.3,  $X$  denotes a random distribution class returning objects of type  $T$ ,  $u$  is a value of  $X$ ,  $x$  is a (possibly const) value of  $X$ , and  $e$  is an lvalue of an arbitrary type that meets the requirements of a uniform random number generator, returning values of type  $U$ .

Table 5.3: Random distribution requirements (in addition to CopyConstructible, and Assignable)

<b>expression</b>	<b>Return Type</b>	<b>pre/post-condition</b>	<b>complexity</b>
<code>X::input_type</code>	$U$	—	compile-time
<code>u.reset()</code>	<code>void</code>	subsequent uses of $u$ do not depend on values produced by $e$ prior to invoking <code>reset</code> .	constant
<code>u(e)</code>	$T$	the sequence of numbers returned by successive invocations with the same object $e$ is randomly distributed with some probability density function $p(x)$	amortized constant number of invocations of $e$
<code>os &lt;&lt; x</code>	<code>std::ostream&amp;</code>	writes a textual representation for the parameters and additional internal data of the distribution $x$ to <code>os</code> . post: The <code>os.fmtflags</code> and fill character are unchanged.	$\mathcal{O}(\text{size of state})$

<i>continued from previous page</i>			
<code>is &gt;&gt; u</code>	<code>std::istream&amp;</code>	restores the parameters and additional internal data of the distribution <code>u</code> . pre: <code>is</code> provides a textual representation that was previously written by <code>operator&lt;&lt;</code> . post: The <code>is.fmtflags</code> are unchanged.	$\mathcal{O}(\text{size of state})$

Additional requirements: The sequence of numbers produced by repeated invocations of `x(e)` does not change whether or not `os << x` is invoked between any of the invocations `x(e)`. If a textual representation is written using `os << x` and that representation is restored into the same or a different object `y` of the same type using `is >> y`, repeated invocations of `y(e)` produce the same sequence of random numbers as would repeated invocations of `x(e)`.

In the following subclauses, a template parameter named `UniformRandomNumberGenerator` shall denote a type that satisfies all the requirements of a uniform random number generator. Moreover, a template parameter named `Distribution` shall denote a type that satisfies all the requirements of a random distribution.

The effect of instantiating a template that has a template type parameter named `RealType` is undefined unless that type is one of `float`, `double`, or `long double`.

The effect of instantiating a template that has a template type parameter named `IntType` is undefined unless that type is one of `short`, `int`, `long`, or their unsigned variants.

The effect of instantiating a template that has a template type parameter named `UIntType` is undefined unless that type is one of `unsigned short`, `unsigned int`, or `unsigned long`.

## 5.1.2 Header `<random>` synopsis

[tr.rand.synopsis]

```
namespace tr1 {
    template<class UniformRandomNumberGenerator,
             class Distribution>
    class variate_generator;

    template<class IntType, IntType a, IntType c, IntType m>
    class linear_congruential;

    template<class UIntType, int w, int n, int m, int r,
             UIntType a, int u, int s,
             UIntType b, int t, UIntType c, int l>
    class mersenne_twister;

    template<class IntType, IntType m, int s, int r>
    class subtract_with_carry;

    template<class RealType, int w, int s, int r>
    class subtract_with_carry_01;
}
```

```

template<class UniformRandomNumberGenerator, int p, int r>
class discard_block;

template<class UniformRandomNumberGenerator1, int s1,
        class UniformRandomNumberGenerator2, int s2>
class xor_combine;

class random_device;

template<class IntType = int>
class uniform_int;

class bernoulli_distribution;

template<class IntType = int, class RealType = double>
class geometric_distribution;

template<class IntType = int, class RealType = double>
class poisson_distribution;

template<class IntType = int, class RealType = double>
class binomial_distribution;

template<class RealType = double>
class uniform_real;

template<class RealType = double>
class exponential_distribution;

template<class RealType = double>
class normal_distribution;

template<class RealType = double>
class gamma_distribution;

} // namespace tr1

```

### 5.1.3 Class template `variate_generator`

[tr.rand.var]

A `variate_generator` produces random numbers, drawing randomness from an underlying uniform random number generator and shaping the distribution of the numbers corresponding to a distribution function.

```

template<class Engine, class Distribution>

```



```

class variate_generator
{
public:
    typedef Engine engine_type;
    typedef implementation defined engine_value_type;
    typedef Distribution distribution_type;
    typedef typename Distribution::result_type result_type;

    variate_generator(engine_type eng, distribution_type d);

    result_type operator()();
    template<class T> result_type operator()(T value);

    engine_value_type& engine();
    const engine_value_type& engine() const;

    distribution_type& distribution();
    const distribution_type& distribution() const;

    result_type min() const;
    result_type max() const;
};

```

The template argument for the parameter `Engine` shall be of the form  $U$ ,  $U\&$ , or  $U^*$ , where  $U$  denotes a class that satisfies all the requirements of a uniform random number generator. The member `engine_value_type` shall name  $U$ .

Specializations of `variate_generator` satisfy the requirements of `CopyConstructible` and `Assignable`.

Except where specified otherwise, the complexity of all functions specified in this section is constant. No function described in this section except the constructor throws an exception.

```

    variate_generator(engine_type eng, distribution_type d)

```

**Effects:** Constructs a `variate_generator` object with the associated uniform random number generator `eng` and the associated random distribution `d`.

**Complexity:** Sum of the complexities of the copy constructors of `engine_type` and `distribution_type`.

**Throws:** If and what the copy constructor of `Engine` or `Distribution` throws.

```

    result_type operator()()

```

**Returns:** `distribution()(e)`

**Complexity:** Amortized constant.

**Notes:** The sequence of numbers produced by the uniform random number generator  $e$ ,  $s_e$ , is obtained from the sequence of numbers produced by the associated uniform random number generator `eng`,  $s_{eng}$ , as follows: Consider the values of `numeric_limits<T>::is_integer` for  $T$  both `Distribution::input_type` and `engine_value_type::result_type`. If the values for both types are true, then  $s_e$  is

identical to  $s_{eng}$ . Otherwise, if the values for both types are `false`, then the numbers in  $s_{eng}$  are divided by `engine().max()-engine().min()` to obtain the numbers in  $s_e$ . Otherwise, if the value for `engine_value_type::result_type` is `true` and the value for `Distribution::input_type` is `false`, then the numbers in  $s_{eng}$  are divided by `engine().max()-engine().min()+1` to obtain the numbers in  $s_e$ . Otherwise, the mapping from  $s_{eng}$  to  $s_e$  is implementation-defined. In all cases, an implicit conversion from `engine_value_type::result_type` to `Distribution::input_type` is performed. If such a conversion does not exist, the program is ill-formed.

```
template<class T> result_type operator()(T value)
```

**Returns:** `distribution()(e, value)`. For the semantics of `e`, see the description of `operator()()`.

```
engine_value_type& engine()
```

**Returns:** A reference to the associated uniform random number generator.

```
const engine_value_type& engine() const
```

**Returns:** A reference to the associated uniform random number generator.

```
distribution_type& distribution()
```

**Returns:** A reference to the associated random distribution.

```
const distribution_type& distribution() const
```

**Returns:** A reference to the associated random distribution.

```
result_type min() const
```

**Precondition:** `distribution().min()` is well-formed.

**Returns:** `distribution().min()`

```
result_type max() const
```

**Precondition:** `distribution().max()` is well-formed

**Returns:** `distribution().max()`

## 5.1.4 Random number engine class templates

[tr.rand.eng]

Except where specified otherwise, the complexity of all functions specified in the following sections is constant. No function described in this section except the constructor and seed functions taking an iterator range `[it1,it2)` throws an exception.

The class templates specified in this section satisfy all the requirements of a pseudo-random number engine (given in table 5.2), except where specified otherwise. Descriptions are provided here only for operations on the engines that are not described in one of these tables or for operations where there is additional semantic information.

All members declared `static const` in any of the following class templates shall be defined in such a way that they are usable as integral constant expressions.

### 5.1.4.1 Class template `linear_congruential`

[tr.rand.eng.lcong]

A `linear_congruential` engine produces random numbers using a linear function  $x(i+1) := (a * x(i) + c) \bmod m$ .

```
namespace tr1 {
    template<class UIntType, UIntType a, UIntType c, UIntType m>
    class linear_congruential
    {
    public:
        // types
        typedef UIntType result_type;

        // parameter values
        static const UIntType multiplier = a;
        static const UIntType increment = c;
        static const UIntType modulus = m;

        // constructors and member function
        explicit linear_congruential(UIntType x0 = 1);
        template<class In> linear_congruential(In& first, In last);
        void seed(UIntType x0 = 1);
        template<class In> void seed(In& first, In last);
        result_type min() const;
        result_type max() const;
        result_type operator()();
    };

    template<class UIntType, UIntType a, UIntType c, UIntType m>
    bool operator==(const linear_congruential<UIntType, a, c, m>& x,
                   const linear_congruential<UIntType, a, c, m>& y);
    template<class UIntType, UIntType a, UIntType c, UIntType m>
    bool operator!=(const linear_congruential<UIntType, a, c, m>& x,
                   const linear_congruential<UIntType, a, c, m>& y);

    template<class CharT, class traits,
              class UIntType, UIntType a, UIntType c, UIntType m>
    basic_ostream<CharT, traits>&
    operator<<(basic_ostream<CharT, traits>& os,
              const linear_congruential<UIntType, a, c, m>& x);
    template<class CharT, class traits,
              class UIntType, UIntType a, UIntType c, UIntType m>
    basic_istream<CharT, traits>&
    operator>>(basic_istream<CharT, traits>& is,
              linear_congruential<UIntType, a, c, m>& x);
}
```

The template parameter `UIntType` shall denote an unsigned integral type large enough to store values up to  $(m-1)$ . If the template parameter `m` is 0, the modulus `m` used throughout this section is `std::numeric_limits<UIntType>::max()` plus 1. [Note: The result is not representable as a value of type `UIntType`. —end note] Otherwise, the template parameters `a` and `c` shall be less than `m`.

The size of the state `x(i)` is 1.

```
explicit linear_congruential(UIntType x0 = 1)
```

**Effects:** Constructs a `linear_congruential` engine and invokes `seed(x0)`.

```
void seed(UIntType x0 = 1)
```

**Effects:** If  $c \bmod m = 0$  and  $x0 \bmod m = 0$ , sets the state `x(i)` of the engine to  $1 \bmod m$ , else sets the state of the engine to  $x0 \bmod m$ .

```
template<class In> linear_congruential(In& first, In last)
```

**Effects:** If  $c \bmod m = 0$  and  $*first \bmod m = 0$ , sets the state `x(i)` of the engine to  $1 \bmod m$ , else sets the state of the engine to  $*first \bmod m$ .

**Complexity:** Exactly one dereference of `*first`.

```
template<class CharT, class traits,
         class UIntType, UIntType a, UIntType c, UIntType m>
basic_ostream<CharT, traits>&
operator<<(basic_ostream<CharT, traits>& os,
          const linear_congruential<UIntType, a, c, m>& x);
```

**Effects:** Writes `x(i)` to `os`.

#### 5.1.4.2 Class template `mersenne_twister`

[tr.rand.eng.mers]

A `mersenne_twister` engine produces random numbers  $o(x(i))$  using the following computation, performed modulo  $2^w$ . `um` is a value with only the upper  $w-r$  bits set in its binary representation. `lm` is a value with only its lower  $r$  bits set in its binary representation. `rshift` is a bitwise right shift with zero-valued bits appearing in the high bits of the result. `lshift` is a bitwise left shift with zero-valued bits appearing in the low bits of the result.

- $y(i) = (x(i-n) \text{ bitand } um) \text{ --- } (x(i-(n-1)) \text{ bitand } lm)$
- If the lowest bit of the binary representation of  $y(i)$  is set,  $x(i) = x(i-(n-m)) \text{ xor } (y(i) \text{ rshift } 1) \text{ xor } a$ ; otherwise  $x(i) = x(i-(n-m)) \text{ xor } (y(i) \text{ rshift } 1)$ .
- $z1(i) = x(i) \text{ xor } (x(i) \text{ rshift } u)$
- $z2(i) = z1(i) \text{ xor } ((z1(i) \text{ lshift } s) \text{ bitand } b)$
- $z3(i) = z2(i) \text{ xor } ((z2(i) \text{ lshift } t) \text{ bitand } c)$
- $o(x(i)) = z3(i) \text{ xor } (z3(i) \text{ rshift } 1)$

```

template<class UIntType, int w, int n, int m, int r,
        UIntType a, int u, int s,
        UIntType b, int t, UIntType c, int l>
class mersenne_twister
{
public:
    // types
    typedef UIntType result_type;

    // parameter values
    static const int word_size = w;
    static const int state_size = n;
    static const int shift_size = m;
    static const int mask_bits = r;
    static const UIntType parameter_a = a;
    static const int output_u = u;
    static const int output_s = s;
    static const UIntType output_b = b;
    static const int output_t = t;
    static const UIntType output_c = c;
    static const int output_l = l;

    // constructors and member function
    mersenne_twister();
    explicit mersenne_twister(unsigned long value);
    template<class In> mersenne_twister(In& first, In last);
    void seed();
    void seed(unsigned long value);
    template<class In> void seed(In& first, In last);
    result_type min() const;
    result_type max() const;
    result_type operator()();
};

template<class UIntType, int w, int n, int m, int r,
        UIntType a, int u, int s,
        UIntType b, int t, UIntType c, int l>
bool
operator==(const mersenne_twister<UIntType,
                                w,n,m,r,a,u,s,b,t,c,l>& y,
           const mersenne_twister<UIntType,
                                w,n,m,r,a,u,s,b,t,c,l>& x);

template<class UIntType, int w, int n, int m, int r,
        UIntType a, int u, int s,
        UIntType b, int t, UIntType c, int l>
bool

```

```

operator!=(const mersenne_twister<UIntType,
            w,n,m,r,a,u,s,b,t,c,l>& y,
            const mersenne_twister<UIntType,
            w,n,m,r,a,u,s,b,t,c,l>& x);

template<class CharT, class traits,
        class UIntType, int w, int n, int m, int r,
        UIntType a, int u, int s,
        UIntType b, int t, UIntType c, int l>
basic_ostream<CharT, traits>&
operator<<(basic_ostream<CharT, traits>& os,
          const mersenne_twister<UIntType,
          w,n,m,r,a,u,s,b,t,c,l>& x);

template<class CharT, class traits,
        class UIntType, int w, int n, int m, int r,
        UIntType a, int u, int s,
        UIntType b, int t, UIntType c, int l>
basic_istream<CharT, traits>&
operator>>(basic_istream<CharT, traits>& is,
          mersenne_twister<UIntType,
          w,n,m,r,a,u,s,b,t,c,l>& x);

```

The template parameter `UIntType` shall denote an unsigned integral type large enough to store values up to  $2^W - 1$ . Also, the following relations shall hold:  $1 \leq m \leq n$ .  $0 \leq r, u, s, t, l \leq w$ .  $0 \leq a, b, c \leq 2^w - 1$ .

The size of the state  $x(i)$  is  $n$ .

```
mersenne_twister()
```

**Effects:** Constructs a `mersenne_twister` engine and invokes `seed()`.

```
explicit mersenne_twister(unsigned long value)
```

**Effects:** Constructs a `mersenne_twister` engine and invokes `seed(value)`.

```
template<class In> mersenne_twister(In& first, In last)
```

**Effects:** Constructs a `mersenne_twister` engine and invokes `seed(first, last)`.

```
void seed()
```

**Effects:** Invokes `seed(0)`.

```
void seed(unsigned long value)
```

**Effects:** If `value == 0`, sets `value` to 4357. In any case, with a linear congruential generator `lcg(i)` having parameters  $m_{lcg} = 2^{32}$ ,  $a_{lcg} = 69069$ ,  $c_{lcg} = 0$ , and  $lcg(0) = value$ , sets  $x(-n) \dots x(-1)$  to `lcg(1) ... lcg(n)`, respectively.

**Complexity:**  $\mathcal{O}(n)$

```
template<class In> void seed(In& first, In last)
```

**Effects:** Given the values  $z_0 \dots z_{n-1}$  obtained by dereferencing  $[first, first+n)$ , sets  $x(-n) \dots x(-1)$  to  $z_0 \bmod 2^w \dots z_{n-1} \bmod 2^w$ .

**Complexity:** Exactly  $n$  dereferences of `first`.

```
template<class UIntType, int w, int n, int m, int r,
        UIntType a, int u, int s,
        UIntType b, int t, UIntType c, int l>
bool
operator==(const mersenne_twister<UIntType,
                                w,n,m,r,a,u,s,b,t,c,l>& y,
           const mersenne_twister<UIntType,
                                w,n,m,r,a,u,s,b,t,c,l>& x)
```

**Returns:**  $x(i-n) == y(j-n)$  and...and  $x(i-1) == y(j-1)$

**Notes:** Assumes the next output of `x` is  $o(x(i))$  and the next output of `y` is  $o(y(j))$ .

**Complexity:**  $\mathcal{O}(n)$

```
template<class CharT, class traits,
        class UIntType, int w, int n, int m, int r,
        UIntType a, int u, int s,
        UIntType b, int t, UIntType c, int l>
basic_ostream<CharT, traits>&
operator<<(basic_ostream<CharT, traits>& os,
          const mersenne_twister<UIntType,
                                w,n,m,r,a,u,s,b,t,c,l>& x)
```

**Effects:** Writes  $x(i-n), \dots x(i-1)$  to `os`, in that order.

**Complexity:**  $\mathcal{O}(n)$

### 5.1.4.3 Class template `subtract_with_carry`

[`tr.rand.eng.sub`]

A `subtract_with_carry` engine produces integer random numbers using  $x(i) = (x(i-s) - x(i-r) - \text{carry}(i-1)) \bmod m$ ; and setting  $\text{carry}(i) = 1$  if  $x(i-s) - x(i-r) - \text{carry}(i-1) < 0$ , else  $\text{carry}(i) = 0$ .

```
template<class IntType, IntType m, int s, int r>
class subtract_with_carry
{
public:
    // types
    typedef IntType result_type;

    // parameter values
    static const IntType modulus = m;
    static const int long_lag = r;
```

```

static const int short_lag = s;

// constructors and member function
subtract_with_carry();
explicit subtract_with_carry(unsigned long value);
template<class In> subtract_with_carry(In& first, In last);
void seed(unsigned long value = 19780503ul);
template<class In> void seed(In& first, In last);
result_type min() const;
result_type max() const;
result_type operator()();
};

template<class IntType, IntType m, int s, int r>
bool operator==(const subtract_with_carry<IntType, m, s, r> & x,
               const subtract_with_carry<IntType, m, s, r> & y);

template<class IntType, IntType m, int s, int r>
bool operator!=(const subtract_with_carry<IntType, m, s, r> & x,
               const subtract_with_carry<IntType, m, s, r> & y);

template<class CharT, class Traits,
         class IntType, IntType m, int s, int r>
std::basic_ostream<CharT, Traits>&
operator<<(std::basic_ostream<CharT, Traits>& os,
          const subtract_with_carry<IntType, m, s, r>& f);

template<class CharT, class Traits,
         class IntType, IntType m, int s, int r>
std::basic_istream<CharT, Traits>&
operator>>(std::basic_istream<CharT, Traits>& is,
          subtract_with_carry<IntType, m, s, r>& f);

```

The template parameter `IntType` shall denote a signed integral type large enough to store values up to `m`. The following relation shall hold:  $0 < s < r$ . Let `w` the number of bits in the binary representation of `m`.

The size of the state is `r`.

```
subtract_with_carry()
```

**Effects:** Constructs a `subtract_with_carry` engine and invokes `seed()`.

```
explicit subtract_with_carry(unsigned long value)
```

**Effects:** Constructs a `subtract_with_carry` engine and invokes `seed(value)`.

```
template<class In> subtract_with_carry(In& first, In last)
```

**Effects:** Constructs a `subtract_with_carry` engine and invokes `seed(first, last)`.



```
void seed(unsigned long value = 19780503)
```

**Effects:** If `value == 0`, sets `value` to 19780503. In any case, with a linear congruential generator `lcg(i)` having parameters  $m_{lcg} = 2147483563$ ,  $a_{lcg} = 40014$ ,  $c_{lcg} = 0$ , and  $lcg(0) = \text{value}$ , sets `x(-r) ... x(-1)` to  $lcg(1) \bmod m \dots lcg(r) \bmod m$ , respectively. If `x(-1) == 0`, sets `carry(-1) = 1`, else sets `carry(-1) = 0`.

**Complexity:**  $\mathcal{O}(r)$

```
template<class In> void seed(In& first, In last)
```

**Effects:** With  $n = (w + 31)/32$  (rounded downward) and given the values  $z_0 \dots z_{n*r-1}$  obtained by dereferencing `[first, first+n*r)`, sets `x(-r) ... x(-1)` to  $(z_0 \cdot 2^{32} + \dots + z_{n-1} \cdot 2^{32(n-1)}) \bmod m \dots (z_{(r-1)n} \cdot 2^{32} + \dots + z_{r-1} \cdot 2^{32(n-1)}) \bmod m$ . If `x(-1) == 0`, sets `carry(-1) = 1`, else sets `carry(-1) = 0`.

**Complexity:** Exactly  $r*n$  dereferences of `first`.

```
template<class IntType, IntType m, int s, int r>
bool operator==(const subtract_with_carry<IntType, m, s, r> & x,
               const subtract_with_carry<IntType, m, s, r> & y)
```

**Returns:** `x(i-r) == y(j-r)` and `...and x(i-1) == y(j-1)`.

**Notes:** Assumes the next output of `x` is `x(i)` and the next output of `y` is `y(j)`.

**Complexity:**  $\mathcal{O}(r)$

```
template<class CharT, class Traits,
         class IntType, IntType m, int s, int r>
std::basic_ostream<CharT, Traits>&
operator<<(std::basic_ostream<CharT, Traits>& os,
          const subtract_with_carry<IntType, m, s, r>& f)
```

**Effects:** Writes `x(i-r) ... x(i-1)`, `carry(i-1)` to `os`, in that order.

**Complexity:**  $\mathcal{O}(r)$

#### 5.1.4.4 Class template `subtract_with_carry_01`

[tr.rand.eng.sub1]

A `subtract_with_carry_01` engine produces floating-point random numbers using  $x(i) = (x(i-s) - x(i-r) - \text{carry}(i-1)) \bmod 1$ ; and setting  $\text{carry}(i) = 2^{-w}$  if  $x(i-s) - x(i-r) - \text{carry}(i-1) < 0$ , else  $\text{carry}(i) = 0$ .

```
template<class RealType, int w, int s, int r>
class subtract_with_carry_01
{
public:
    // types
    typedef RealType result_type;

    // parameter values
    static const int word_size = w;
    static const int long_lag = r;
```

```

static const int short_lag = s;

// constructors and member function
subtract_with_carry_01();
explicit subtract_with_carry_01(unsigned long value);
template<class In> subtract_with_carry_01(In& first, In last);
void seed(unsigned long value = 19780503);
template<class In> void seed(In& first, In last);
result_type min() const;
result_type max() const;
result_type operator()();
};
template<class RealType, int w, int s, int r>
bool operator==(const subtract_with_carry_01<RealType, w, s, r> x,
                const subtract_with_carry_01<RealType, w, s, r> y);

template<class RealType, int w, int s, int r>
bool operator!=(const subtract_with_carry_01<RealType, w, s, r> x,
                const subtract_with_carry_01<RealType, w, s, r> y);

template<class CharT, class Traits,
          class RealType, int w, int s, int r>
std::basic_ostream<CharT,Traits>&
operator<<(std::basic_ostream<CharT,Traits>& os,
          const subtract_with_carry_01<RealType, w, s, r>& f);

template<class CharT, class Traits,
          class RealType, int w, int s, int r>
std::basic_istream<CharT,Traits>&
operator>>(std::basic_istream<CharT,Traits>& is,
          subtract_with_carry_01<RealType, w, s, r>& f);

```

The following relation shall hold:  $0 < s < r$ .

The size of the state is  $r$ .

```
subtract_with_carry_01()
```

**Effects:** Constructs a `subtract_with_carry_01` engine and invokes `seed()`.

```
explicit subtract_with_carry_01(unsigned long value)
```

**Effects:** Constructs a `subtract_with_carry_01` engine and invokes `seed(value)`.

```
template<class In> subtract_with_carry_01(In& first, In last)
```

**Effects:** Constructs a `subtract_with_carry_01` engine and invokes `seed(first, last)`.

```
void seed(unsigned long value = 19780503)
```

**Effects:** If `value == 0`, sets `value` to 19780503. In any case, with a linear congruential generator `lcg(i)` having parameters  $m_{lcg} = 2147483563$ ,  $a_{lcg} = 40014$ ,  $c_{lcg} = 0$ , and  $lcg(0) = \text{value}$ , sets `x(-r) ... x(-1)` to  $(lcg(1) \cdot 2^{-w}) \bmod 1 \dots (lcg(r) \cdot 2^{-w} \bmod 1)$ , respectively. If `x(-1) == 0`, sets `carry(-1) = 2^{-w}`, else sets `carry(-1) = 0`.

**Complexity:**  $\mathcal{O}(n \cdot r)$ .

```
template<class In> void seed(In& first, In last)
```

**Effects:** With  $n = (w + 31)/32$  (rounded downward) and given the values  $z_0 \dots z_{n \cdot r - 1}$  obtained by dereferencing `[first, first+n*r)`, sets `x(-r) ... x(-1)` to  $(z_0 \cdot 2^{32} + \dots + z_{n-1} \cdot 2^{32(n-1)}) \cdot 2^{-w} \bmod 1 \dots (z_{(r-1)n} \cdot 2^{32} + \dots + z_{r-1} \cdot 2^{32(n-1)}) \cdot 2^{-w} \bmod 1$ . If `x(-1) == 0`, sets `carry(-1) = 2^{-w}`, else sets `carry(-1) = 0`.

**Complexity:**  $\mathcal{O}(r \cdot n)$

```
template<class RealType, int w, int s, int r>
bool operator==(const subtract_with_carry<RealType, w, s, r> x,
                const subtract_with_carry<RealType, w, s, r> y);
```

**Returns:** true, if and only if `x(i-r) == y(j-r)` and ... and `x(i-1) == y(j-1)`.

**Complexity:**  $\mathcal{O}(r)$

```
template<class CharT, class Traits,
         class RealType, int w, int s, int r>
std::basic_ostream<CharT, Traits>&
operator<<(std::basic_ostream<CharT, Traits>& os,
          const subtract_with_carry<RealType, w, s, r>& f);
```

**Effects:** Write `x(i-r)*2w ... x(i-1)*2w, carry(i-1)*2w` to `os`, in that order.

**Complexity:**  $\mathcal{O}(r)$

#### 5.1.4.5 Class template `discard_block`

[tr.rand.eng.disc]

A `discard_block` engine produces random numbers from some base engine by discarding blocks of data.

```
template<class UniformRandomNumberGenerator, int p, int r>
class discard_block
{
public:
    // types
    typedef UniformRandomNumberGenerator base_type;
    typedef typename base_type::result_type result_type;

    // parameter values
    static const int block_size = p;
    static const int used_block = r;

    // constructors and member function
```

```

discard_block();
explicit discard_block(const base_type & rng);
template<class In> discard_block(In& first, In last);
void seed();
template<class In> void seed(In& first, In last);
const base_type& base() const;
result_type min() const;
result_type max() const;
result_type operator()();
private:
    base_type b;                // exposition only
    int n;                      // exposition only
};

template<class UniformRandomNumberGenerator, int p, int r>
bool
operator==(const discard_block<UniformRandomNumberGenerator,p,r> & x,
           const discard_block<UniformRandomNumberGenerator,p,r> & y);

template<class UniformRandomNumberGenerator, int p, int r>
bool
operator!=(const discard_block<UniformRandomNumberGenerator,p,r> & x,
           const discard_block<UniformRandomNumberGenerator,p,r> & y);

template<class CharT, class traits,
         class UniformRandomNumberGenerator, int p, int r>
basic_ostream<CharT, traits>&
operator<<(basic_ostream<CharT, traits>& os,
          const discard_block<UniformRandomNumberGenerator,p,r> & x);

template<class CharT, class traits,
         class UniformRandomNumberGenerator, int p, int r>
basic_istream<CharT, traits>&
operator>>(basic_istream<CharT, traits>& is,
          discard_block<UniformRandomNumberGenerator,p,r> & x);

```

The template parameter `UniformRandomNumberGenerator` shall denote a class that satisfies all the requirements of a uniform random number generator, given in table 5.1 in clause 5.1.1.  $0 \leq r \leq p$ . The size of the state is the size of `b` plus 1.

```
discard_block()
```

**Effects:** Constructs a `discard_block` engine. To construct the subobject `b`, invokes its default constructor. Sets `n = 0`.

```
explicit discard_block(const base_type & rng)
```

**Effects:** Constructs a `discard_block` engine. Initializes `b` with a copy of `rng`. Sets `n = 0`.

```
template<class In> discard_block(In& first, In last)
```

**Effects:** Constructs a `discard_block` engine. To construct the subobject *b*, invokes the `b(first, last)` constructor. Sets `n = 0`.

```
void seed()
```

**Effects:** Invokes `b.seed()` and sets `n = 0`.

```
template<class In> void seed(In& first, In last)
```

**Effects:** Invokes `b.seed(first, last)` and sets `n = 0`.

```
const base_type& base() const
```

**Returns:** *b*

```
result_type operator()()
```

**Effects:** If `n >= r`, invokes *b* (`p-r`) times, discards the values returned, and sets `n = 0`. In any case, then increments `n` and returns `b()`.

```
template<class CharT, class traits,
         class UniformRandomNumberGenerator, int p, int r>
basic_ostream<CharT, traits>&
operator<<(basic_ostream<CharT, traits>& os,
          const discard_block<UniformRandomNumberGenerator,p,r> & x);
```

**Effects:** Writes *b*, then *n* to *os*.

#### 5.1.4.6 Class template `xor_combine`

[tr.rand.eng.xor]

A `xor_combine` engine produces random numbers from two integer base engines by merging their random values with bitwise exclusive-or.

```
template<class UniformRandomNumberGenerator1, int s1,
         class UniformRandomNumberGenerator2, int s2>
class xor_combine
{
public:
    // types
    typedef UniformRandomNumberGenerator1 base1_type;
    typedef UniformRandomNumberGenerator2 base2_type;
    typedef /* see below */ result_type;

    // parameter values
    static const int shift1 = s1;
    static const int shift2 = s2;

    // constructors and member function
```

```

xor_combine();
xor_combine(const base1_type & rng1, const base2_type & rng2);
template<class In> xor_combine(In& first, In last);
void seed();
template<class In> void seed(In& first, In last);
const base1_type& base1() const;
const base2_type& base2() const;
result_type min() const;
result_type max() const;
result_type operator()();
private:
    base1_type b1;                // exposition only
    base2_type b2;                // exposition only
};
template<class UniformRandomNumberGenerator1, int s1,
         class UniformRandomNumberGenerator2, int s2>
bool
operator==(const xor_combine<UniformRandomNumberGenerator1, s1,
                          UniformRandomNumberGenerator2, s2> & x,
           const xor_combine<UniformRandomNumberGenerator1, s1,
                          UniformRandomNumberGenerator2, s2> & y);

template<class UniformRandomNumberGenerator1, int s1,
         class UniformRandomNumberGenerator2, int s2>
operator!=(const xor_combine<UniformRandomNumberGenerator1, s1,
                          UniformRandomNumberGenerator2, s2> & x,
           const xor_combine<UniformRandomNumberGenerator1, s1,
                          UniformRandomNumberGenerator2, s2> & y);

template<class CharT, class traits,
         class UniformRandomNumberGenerator1, int s1,
         class UniformRandomNumberGenerator2, int s2>
basic_ostream<CharT, traits>&
operator<<(basic_ostream<CharT, traits>& os,
          const xor_combine<UniformRandomNumberGenerator1, s1,
                          UniformRandomNumberGenerator2, s2> & x);

template<class CharT, class traits,
         class UniformRandomNumberGenerator1, int s1,
         class UniformRandomNumberGenerator2, int s2>
basic_istream<CharT, traits>&
operator>>(basic_istream<CharT, traits>& is,
          xor_combine<UniformRandomNumberGenerator1, s1,
                    UniformRandomNumberGenerator2, s2> & x);

```

The template parameters `UniformRandomNumberGenerator1` and `UniformRandomNumberGenerator2`

shall denote classes that satisfy all the requirements of a uniform random number generator, given in table 5.1 in clause 5.1.1. Both `UniformRandomNumberGenerator1::result_type` and `UniformRandomNumberGenerator2::result_type` shall denote (possibly different) unsigned integral types. The following relation shall hold:  $0 \leq s1$  and  $0 < s2$ . The size of the state is the size of the state of *b1* plus the size of the state of *b2*.

The member `result_type` is defined to that of `UniformRandomNumberGenerator1::result_type` and `UniformRandomNumberGenerator2::result_type` that provides the most storage [basic.fundamental].

```
xor_combine()
```

**Effects:** Constructs a `xor_combine` engine. To construct each of the subobjects *b1* and *b2*, invokes their respective default constructors.

```
xor_combine(const base1_type & rng1, const base2_type & rng2)
```

**Effects:** Constructs a `xor_combine` engine. Initializes *b1* with a copy of `rng1` and *b2* with a copy of `rng2`.

```
template<class In> xor_combine(In& first, In last)
```

**Effects:** Constructs a `xor_combine` engine. To construct the subobject *b1*, invokes the `b1(first, last)` constructor. Then, to construct the subobject *b2*, invokes the `b2(first, last)` constructor.

```
void seed()
```

**Effects:** Invokes `b1.seed()` and `b2.seed()`.

```
template<class In> void seed(In& first, In last)
```

**Effects:** Invokes `b1.seed(first, last)`, then invokes `b2.seed(first, last)`.

```
const base1_type& base1() const
```

**Returns:** *b1*

```
const base2_type& base2() const
```

**Returns:** *b2*

```
result_type operator()()
```

**Returns:**  $(b1() \ll s1) \wedge (b2() \ll s2)$ .

```
template<class CharT, class traits,
         class UniformRandomNumberGenerator1, int s1,
         class UniformRandomNumberGenerator2, int s2>
basic_ostream<CharT, traits>&
operator<<(basic_ostream<CharT, traits>& os,
          const xor_combine<UniformRandomNumberGenerator1, s1,
                          UniformRandomNumberGenerator2, s2> & x);
```

**Effects:** Writes *b1*, then *b2* to *os*.

### 5.1.5 Engines with predefined parameters

[tr.rand.predef]

```
typedef linear_congruential<implementation-defined,  
                             16807, 0, 2147483647>  
    minstd_rand0;  
typedef linear_congruential<implementation-defined,  
                             48271, 0, 2147483647>  
    minstd_rand;  
  
typedef mersenne_twister<implementation-defined,  
                          32,624,397,31,0x9908b0df,11,7,0x9d2c5680,15,0xefc60000,18>  
    mt19937;  
  
typedef subtract_with_carry_01<float, 24, 10, 24> ranlux_base_01;  
typedef subtract_with_carry_01<double, 48, 10, 24> ranlux64_base_01;  
  
typedef discard_block<subtract_with_carry<implementation-defined,  
                      (1<<24), 10, 24>, 223, 24>  
    ranlux3;  
typedef discard_block<subtract_with_carry<implementation-defined,  
                      (1<<24), 10, 24>, 389, 24>  
    ranlux4;  
  
typedef discard_block<subtract_with_carry_01<float, 24, 10, 24>, 223, 24>  
    ranlux3_01;  
typedef discard_block<subtract_with_carry_01<float, 24, 10, 24>, 389, 24>  
    ranlux4_01;
```

For a default-constructed `minstd_rand0` object,  $x(10000) = 1043618065$ . For a default-constructed `minstd_rand` object,  $x(10000) = 399268537$ .

For a default-constructed `mt19937` object,  $x(10000) = 3346425566$ .

For a default-constructed `ranlux3` object,  $x(10000) = 5957620$ . For a default-constructed `ranlux4` object,  $x(10000) = 8587295$ . For a default-constructed `ranlux3_01` object,  $x(10000) = 5957620 \cdot 2^{-24}$ . For a default-constructed `ranlux4_01` object,  $x(10000) = 8587295 \cdot 2^{-24}$ .

### 5.1.6 Class `random_device`

[tr.rand.device]

A `random_device` produces non-deterministic random numbers. It satisfies all the requirements of a uniform random number generator (given in table 5.1 in clause 5.1.1). Descriptions are provided here only for operations on the engines that are not described in one of these tables or for operations where there is additional semantic information.



If implementation limitations prevent generating non-deterministic random numbers, the implementation can employ a pseudo-random number engine.

```
class random_device
{
public:
    // types
    typedef unsigned int result_type;

    // constructors, destructors and member functions
    explicit random_device(const std::string& token = implementation-defined);
    result_type min() const;
    result_type max() const;
    double entropy() const;
    result_type operator()();

private:
    random_device(const random_device& );
    void operator=(const random_device& );
};

explicit random_device(const std::string& token = implementation-defined);
```

**Effects:** Constructs a `random_device` non-deterministic random number engine. The semantics and default value of the `token` parameter are implementation-defined.<sup>1</sup>

**Throws:** A value of some type derived from `exception` if the `random_device` could not be initialized.

```
result_type min() const
```

**Returns:** `numeric_limits<result_type>::min()`

```
result_type max() const
```

**Returns:** `numeric_limits<result_type>::max()`

```
double entropy() const
```

**Returns:** An entropy estimate for the random numbers returned by `operator()`, in the range `min()` to  $\log_2(\max() + 1)$ . A deterministic random number generator (e.g. a pseudo-random number engine) has entropy 0.

**Throws:** Nothing.

```
result_type operator()()
```

**Returns:** A non-deterministic random value, uniformly distributed between `min()` and `max()`, inclusive. It is implementation-defined how these values are generated.

---

<sup>1</sup> The parameter is intended to allow an implementation to differentiate between different sources of randomness.

**Throws:** A value of some type derived from `exception` if a random number could not be obtained.

### 5.1.7 Random distribution class templates [tr.rand.dist]

The class templates specified in this section satisfy all the requirements of a random distribution (given in tables in clause 5.1.1). Descriptions are provided here only for operations on the distributions that are not described in one of these tables or for operations where there is additional semantic information.

Given an object whose type is specified in this subclause, if the lifetime of the uniform random number generator referred to in the constructor invocation for that object has ended, any use of that object is undefined.

The algorithms for producing each of the specified distributions are implementation-defined.

#### 5.1.7.1 Class template `uniform_int` [tr.rand.dist.iunif]

A `uniform_int` random distribution produces integer random numbers  $x$  in the range  $\min \leq x \leq \max$ , with equal probability. `min` and `max` are the parameters of the distribution.

A `uniform_int` random distribution satisfies all the requirements of a uniform random number generator (given in table 5.1 in clause 5.1.1).

```
template<class IntType = int>
class uniform_int
{
public:
    // types
    typedef IntType input_type;
    typedef IntType result_type;

    // constructors and member function
    explicit uniform_int(IntType min = 0, IntType max = 9);
    result_type min() const;
    result_type max() const;
    void reset();

    template<class UniformRandomNumberGenerator>
        result_type
        operator()(UniformRandomNumberGenerator& urng);
    template<class UniformRandomNumberGenerator>
        result_type
        operator()(UniformRandomNumberGenerator& urng, result_type n);
};

uniform_int(IntType min = 0, IntType max = 9)
```

**Requires:**  $\min \leq \max$

**Effects:** Constructs a `uniform_int` object. `min` and `max` are the parameters of the distribution.

```
result_type min() const
```

**Returns:** The “min” parameter of the distribution.

```
result_type max() const
```

**Returns:** The “max” parameter of the distribution.

```
result_type  
operator()(UniformRandomNumberGenerator& urng, result_type n)
```

**Returns:** A uniform random number  $x$  in the range  $0 \leq x < n$ . [Note: This allows a variate\_generator object with a uniform\_int distribution to be used with std::random\_shuffle, see [lib.alg.random.shuffle]. —end note]

### 5.1.7.2 Class bernoulli\_distribution

[tr.rand.dist.bern]

A bernoulli\_distribution random distribution produces bool values distributed with probabilities  $p(\text{true}) = p$  and  $p(\text{false}) = 1 - p$ .  $p$  is the parameter of the distribution.

```
class bernoulli_distribution  
{  
public:  
    // types  
    typedef int input_type;  
    typedef bool result_type;  
  
    // constructors and member function  
    explicit bernoulli_distribution(double p = 0.5);  
    RealType p() const;  
    void reset();  
    template<class UniformRandomNumberGenerator>  
    result_type operator()(UniformRandomNumberGenerator& urng);  
};
```

```
bernoulli_distribution(double p = 0.5)
```

**Requires:**  $0 \leq p \leq 1$

**Effects:** Constructs a bernoulli\_distribution object.  $p$  is the parameter of the distribution.

```
RealType p() const
```

**Returns:** The “p” parameter of the distribution.

### 5.1.7.3 Class template geometric\_distribution

[tr.rand.dist.geom]

A geometric\_distribution random distribution produces integer values  $i > 1$  with  $p(i) = (1 - p) \cdot p^{i-1}$ .  $p$  is the parameter of the distribution.

```

template<class IntType = int, class RealType = double>
class geometric_distribution
{
public:
    // types
    typedef RealType input_type;
    typedef IntType result_type;

    // constructors and member function
    explicit geometric_distribution(const RealType& p = RealType(0.5));
    RealType p() const;
    void reset();
    template<class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
};

geometric_distribution(const RealType& p = RealType(0.5))

```

**Requires:**  $0 < p < 1$

**Effects:** Constructs a `geometric_distribution` object; `p` is the parameter of the distribution.

```
RealType p() const
```

**Returns:** The “*p*” parameter of the distribution.

#### 5.1.7.4 Class template `poisson_distribution`

[tr.rand.dist.pois]

A `poisson_distribution` random distribution produces integer values  $i > 0$  with probability distribution  $p(i) = e^{-mean} \cdot mean^i / i!$ , where *mean* is the parameter of the distribution.

```

template<class IntType = int, class RealType = double>
class poisson_distribution
{
public:
    // types
    typedef RealType input_type;
    typedef IntType result_type;

    // constructors and member function
    explicit poisson_distribution(const RealType& mean = RealType(1));
    RealType mean() const;
    void reset();
    template<class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
};

poisson_distribution(const RealType& mean = RealType(1))

```

**Requires:**  $mean > 0$

**Effects:** Constructs a `poisson_distribution` object; `mean` is the parameter of the distribution.

```
RealType mean() const
```

**Returns:** The *mean* parameter of the distribution.

### 5.1.7.5 Class template `binomial_distribution`

[tr.rand.dist.bin]

A `binomial_distribution` random distribution produces integer values  $i > 0$  with  $p(i) = \binom{n}{i} \cdot p^i \cdot (1-p)^{n-i}$ .  $t$  and  $p$  are the parameters of the distribution.

```
template<class IntType = int, class RealType = double>
class binomial_distribution
{
public:
    // types
    typedef implementation-defined input_type;
    typedef IntType result_type;

    // constructors and member function
    explicit binomial_distribution(IntType t = 1,
                                   const RealType& p = RealType(0.5));

    IntType t() const;
    RealType p() const;
    void reset();
    template<class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
};

binomial_distribution(IntType t = 1,
                      const RealType& p = RealType(0.5))
```

**Requires:**  $0 \leq p \leq 1$  and  $t \geq 0$ .

**Effects:** Constructs a `binomial_distribution` object; `t` and `p` are the parameters of the distribution.

```
IntType t() const
```

**Returns:** The “*t*” parameter of the distribution.

```
RealType p() const
```

**Returns:** The “*p*” parameter of the distribution.

### 5.1.7.6 Class template `uniform_real`

[tr.rand.dist.runif]

A `uniform_real` random distribution produces floating-point random numbers  $x$  in the range  $min \leq x < max$ , with equal probability. `min` and `max` are the parameters of the distribution.

A `uniform_real` random distribution satisfies all the requirements of a uniform random number generator (given in table 5.1 in clause 5.1.1).

```
template<class RealType = double>
class uniform_real
{
public:
    // types
    typedef RealType input_type;
    typedef RealType result_type;

    // constructors and member function
    explicit uniform_real(RealType min = RealType(0),
                          RealType max = RealType(1));
    result_type min() const;
    result_type max() const;
    void reset();
    template<class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
};

uniform_real(RealType min = RealType(0), RealType max = RealType(1))
```

**Requires:**  $\min \leq |max|$ .

**Effects:** Constructs a `uniform_real` object; `min` and `max` are the parameters of the distribution.

```
result_type min() const
```

**Returns:** The “min” parameter of the distribution.

```
result_type max() const
```

**Returns:** The “max” parameter of the distribution.

### 5.1.7.7 Class template `exponential_distribution`

[tr.rand.dist.exp]

An `exponential_distribution` random distribution produces random numbers  $x > 0$  distributed with probability density function  $p(x) = \lambda e^{-\lambda x}$ , where  $\lambda$  is the parameter of the distribution.

```
template<class RealType = double>
class exponential_distribution
{
public:
    // types
    typedef RealType input_type;
    typedef RealType result_type;

    // constructors and member function
```

```

    explicit exponential_distribution(
        const result_type& lambda = result_type(1));
    RealType lambda() const;
    void reset();
    template<class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
};

exponential_distribution(
    const result_type& lambda = result_type(1))

```

**Requires:**  $\lambda > 0$ .

**Effects:** Constructs an `exponential_distribution` object with `rng` as the reference to the underlying source of random numbers. `lambda` is the parameter for the distribution.

```
RealType lambda() const
```

**Returns:** The “ $\lambda$ ” parameter of the distribution.

#### 5.1.7.8 Class template `normal_distribution`

[tr.rand.dist.norm]

A `normal_distribution` random distribution produces random numbers  $x$  distributed with probability density function  $(1/\sqrt{2\pi})\sigma e^{-(x-mean)^2/(2\sigma^2)}$ , where *mean* and  $\sigma$  are the parameters of the distribution.

```

template<class RealType = double>
class normal_distribution
{
public:
    // types
    typedef RealType input_type;
    typedef RealType result_type;

    // constructors and member function
    explicit normal_distribution(const result_type& mean = 0,
                               const result_type& sigma = 1);

    RealType mean() const;
    RealType sigma() const;
    void reset();
    template<class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
};

explicit normal_distribution(const result_type& mean = 0,
                            const result_type& sigma = 1);

```

**Requires:**  $\sigma > 0$ .

**Effects:** Constructs a `normal_distribution` object; `mean` and `sigma` are the parameters for the distribution.

```
RealType mean() const
```

**Returns:** The “*mean*” parameter of the distribution.

```
RealType sigma() const
```

**Returns:** The “*σ*” parameter of the distribution.

### 5.1.7.9 Class template `gamma_distribution`

[tr.rand.dist.gamma]

A `gamma_distribution` random distribution produces random numbers  $x$  distributed with probability density function  $p(x) = 1/\Gamma(\alpha)x^{\alpha-1}e^{-x}$ , where  $\alpha$  is the parameter of the distribution.

```
template<class RealType = double>
class gamma_distribution
{
public:
    // types
    typedef RealType input_type;
    typedef RealType result_type;

    // constructors and member function
    explicit gamma_distribution(
        const result_type& alpha = result_type(1));
    RealType alpha() const;
    void reset();
    template<class UniformRandomNumberGenerator>
    result_type operator()(UniformRandomNumberGenerator& urng);
};

explicit gamma_distribution(
    const result_type& alpha = result_type(1));
```

**Requires:**  $\alpha > 0$ .

**Effects:** Constructs a `gamma_distribution` object;  $\alpha$  is the parameter for the distribution.

```
RealType alpha() const
```

**Returns:** The “ $\alpha$ ” parameter of the distribution.

## 5.2 Mathematical special functions

[tr.num.sf]

### 5.2.1 Additions to header `<cmath>` synopsis

[tr.math.sf.cmath]

The Table 5.4 summarizes the functions that are added to header `<cmath>`. The detailed signatures are given in the synopsis.



Table 5.4: Summary of additions to header <cmath>

Type	Name(s)		
<b>Functions:</b>			
assoc_laguerre	conf_hyperg	ellint_2	legendre
assoc_legendre	cyl_bessel_i	ellint_3	riemann_zeta
beta	cyl_bessel_j	expint	sph_bessel
comp_ellint_1	cyl_bessel_k	hermite	sph_legendre
comp_ellint_2	cyl_neumann	hyperg	sph_neumann
comp_ellint_3	ellint_1	laguerre	

Each of these functions is provided for arguments of type float, double, and long double. The signatures added to header <cmath> are:

```

namespace std {
  namespace tr1 {
    // associated Laguerre polynomials:
    double      assoc_laguerre(unsigned n, unsigned m, double x);
    float       assoc_laguerref(unsigned n, unsigned m, float x);
    long double assoc_laguerrel(unsigned n, unsigned m, long double x);

    // associated Legendre functions:
    double      assoc_legendre(unsigned l, unsigned m, double x)
    float       assoc_legendref(unsigned l, unsigned m, float x)
    long double assoc_legendrel(unsigned l, unsigned m, long double x)

    // beta function:
    double      beta(double x, double y);
    float       betaf(float x, float y);
    long double betal(long double x, long double y);

    // (complete) elliptic integral of the first kind:
    double      comp_ellint_1(double k);
    float       comp_ellint_1f(float k);
    long double comp_ellint_1l(long double k);

    // (complete) elliptic integral of the second kind:
    double      comp_ellint_2(double k);
    float       comp_ellint_2f(float k);
    long double comp_ellint_2l(long double k);

    // (complete) elliptic integral of the third kind:
    double      comp_ellint_3(double k, double nu);
    float       comp_ellint_3f(float k, float nu);
    long double comp_ellint_3l(long double k, long double nu);
  }
}

```

```

// confluent hypergeometric functions:
double      conf_hyperg(double a, double c, double x);
float       conf_hypergf(float a, float c, float x);
long double conf_hypergl(long double a, long double c, long double x);

// regular modified cylindrical Bessel functions:
double      cyl_bessel_i(double nu, double x);
float       cyl_bessel_if(float nu, float x);
long double cyl_bessel_il(long double nu, long double x);

// cylindrical Bessel functions (of the first kind):
double      cyl_bessel_j(double nu, double x);
float       cyl_bessel_jf(float nu, float x);
long double cyl_bessel_jl(long double nu, long double x);

// irregular modified cylindrical Bessel functions:
double      cyl_bessel_k(double nu, double x);
float       cyl_bessel_kf(float nu, float x);
long double cyl_bessel_kl(long double nu, long double x);

// cylindrical Neumann functions;
// cylindrical Bessel functions (of the second kind):
double      cyl_neumann(double nu, double x);
float       cyl_neumannf(float nu, float x);
long double cyl_neumannl(long double nu, long double x);

// (incomplete) elliptic integral of the first kind:
double      ellint_1(double k, double phi);
float       ellint_1f(float k, float phi);
long double ellint_1l(long double k, long double phi);

// (incomplete) elliptic integral of the second kind:
double      ellint_2(double k, double phi);
float       ellint_2f(float k, float phi);
long double ellint_2l(long double k, long double phi);

// (incomplete) elliptic integral of the third kind:
double      ellint_3(double k, double nu, double phi);
float       ellint_3f(float k, float nu, float phi);
long double ellint_3l(long double k, long double nu, long double phi);

// exponential integral:
double      expint(double x);
float       expintf(float x);
long double expintl(long double x);

```

```

// Hermite polynomials:
double      hermite(unsigned n, double x);
float       hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);

// hypergeometric functions:
double      hyperg(double a, double b, double c, double x);
float       hypergf(float a, float b, float c, float x);
long double hypergl(long double a, long double b, long double c,
                    long double x);

// Laguerre polynomials:
double      laguerre(unsigned n, double x);
float       laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);

// Legendre polynomials:
double      legendre(unsigned l, double x)
float       legendref(unsigned l, float x)
long double legendrel(unsigned l, long double x)

// Riemann zeta function:
double      riemann_zeta(double);
float       riemann_zetaf(float);
long double riemann_zetal(long double);

// spherical Bessel functions (of the first kind):
double      sph_bessel(unsigned n, double x);
float       sph_besself(unsigned n, float x);
long double sph_bessell(unsigned n, long double x);

// spherical associated Legendre functions:
double      sph_legendre(unsigned l, unsigned m, double theta)
float       sph_legendref(unsigned l, unsigned m, float theta)
long double sph_legendrel(unsigned l, unsigned m, long double theta)

// spherical Neumann functions;
// spherical Bessel functions (of the second kind):
double      sph_neumann(unsigned n, double x);
float       sph_neumannf(unsigned n, float x);
long double sph_neumannl(unsigned n, long double x);
} // namespace tr1
} // namespace std

```

Each of the functions declared above that has one or more double parameters (the double version) shall have two additional overloads:

1. a version with each `double` parameter replaced with a `float` parameter (the `float` version), and
2. a version with each `double` parameter replaced with a `long double` parameter (the `long double` version).

The return type of each such `float` version shall be `float`, and the return type of each such `long double` version shall be `long double`.

Moreover, each `double` version shall have sufficient additional overloads to determine which of the above three versions to actually call, by the following ordered set of rules:

1. First, if any argument corresponding to a `double` parameter in the `double` version has type `long double`, the `long double` version is called.
2. Otherwise, if any argument corresponding to a `double` parameter in the `double` version has type `double` or has an integer type, the `double` version is called.
3. Otherwise, the `float` version is called.

### 5.2.1.1 associated Laguerre polynomials [tr.math.sf.Lnm]

```
double      assoc_laguerre(unsigned n, unsigned m, double x);
float       assoc_laguerref(unsigned n, unsigned m, float x);
long double assoc_laguerrel(unsigned n, unsigned m, long double x);
```

**Effects:** These functions compute the associated Laguerre polynomials of their respective arguments `n`, `m`, and `x`.

**Returns:** The `assoc_laguerre` functions return

$$L_n^m(x) = e^x \frac{d^m}{dx^m} L_n(x).$$

### 5.2.1.2 associated Legendre functions [tr.math.sf.Plm]

```
double      assoc_legendre(unsigned l, unsigned m, double x)
float       assoc_legendref(unsigned l, unsigned m, float x)
long double assoc_legendrel(unsigned l, unsigned m, long double x)
```

**Effects:** These functions compute the associated Legendre functions of their respective arguments `l`, `m`, and `x`. A domain error occurs if `m` is greater than `l`. A domain error may occur if the magnitude of `x` is greater than one.

**Returns:** The `assoc_legendre` functions return

$$P_l^m(x) = (1 - x^2)^{m/2} \frac{d^m}{dx^m} P_l(x).$$

### 5.2.1.3 beta function [tr.math.sf.beta]

```
double      beta(double x, double y);
float       betaf(float x, float y);
long double betal(long double x, long double y);
```

**Effects:** These functions compute the beta function of their respective arguments  $x$  and  $y$ . A domain error may occur (a) if either  $x$  or  $y$  is a negative integer, or (b) if either  $x$  or  $y$  is zero.

**Returns:** The beta functions return

$$B(x, y) = \frac{\Gamma(x)\Gamma(y)}{\Gamma(x+y)}.$$

#### 5.2.1.4 (complete) elliptic integral of the first kind

[tr.math.sf.ellK]

```
double      comp_ellint_1(double k);
float       comp_ellint_1f(float k);
long double comp_ellint_1l(long double k);
```

**Effects:** These functions compute the complete elliptic integral of the first kind of their respective arguments  $k$ . A domain error occurs if the magnitude of  $k$  is greater than one.

**Returns:** The `comp_ellint_1` functions return

$$K(k) = F(k, \pi/2) = \int_0^{\pi/2} \frac{d\theta}{\sqrt{1 - k^2 \sin^2 \theta}}.$$

#### 5.2.1.5 (complete) elliptic integral of the second kind

[tr.math.sf.ellEx]

```
double      comp_ellint_2(double k);
float       comp_ellint_2f(float k);
long double comp_ellint_2l(long double k);
```

**Effects:** These functions compute the complete elliptic integral of the second kind of their respective arguments  $k$ . A domain error occurs if the magnitude of  $k$  is greater than one.

**Returns:** The `comp_ellint_2` functions return

$$E(k, \pi/2) = \int_0^{\pi/2} \sqrt{1 - k^2 \sin^2 \theta} d\theta.$$

#### 5.2.1.6 (complete) elliptic integral of the third kind

[tr.math.sf.ellPx]

```
double      comp_ellint_3(double k, double nu);
float       comp_ellint_3f(float k, float nu);
long double comp_ellint_3l(long double k, long double nu);
```

**Effects:** These functions compute the complete elliptic integral of the third kind of their respective arguments  $k$  and  $nu$ . A domain error occurs if the magnitude of  $k$  is greater than one.

**Returns:** The `comp_ellint_3` functions return

$$\Pi(\nu, k, \pi/2) = \int_0^{\pi/2} \frac{d\theta}{(1 - \nu \sin^2 \theta) \sqrt{1 - k^2 \sin^2 \theta}}.$$

### 5.2.1.7 confluent hypergeometric functions

[tr.math.sf.conhyp]

```
double      conf_hyperg(double a, double c, double x) ;
float       conf_hypergf(float a, float c, float x) ;
long double conf_hypergl(long double a, long double c, long double x) ;
```

**Effects:** These functions compute the confluent hypergeometric functions of their respective arguments  $a$ ,  $c$ , and  $x$ . A domain error occurs (a) if  $c$  is a negative integer, or (b) if  $c$  is zero.

**Returns:** The `conf_hyperg` functions return

$$F(a; c; x) = \frac{\Gamma(c)}{\Gamma(a)} \sum_{n=0}^{\infty} \frac{\Gamma(a+n)}{\Gamma(c+n)} \frac{x^n}{n!}.$$

### 5.2.1.8 regular modified cylindrical Bessel functions

[tr.math.sf.I]

```
double      cyl_bessel_i(double nu, double x);
float       cyl_bessel_if(float nu, float x);
long double cyl_bessel_il(long double nu, long double x);
```

**Effects:** These functions compute the regular modified cylindrical Bessel functions of their respective arguments  $\nu$  and  $x$ . A domain error may occur if  $x$  is less than zero.

**Returns:** The `cyl_bessel_i` functions return

$$I_{\nu}(x) = i^{-\nu} J_{\nu}(ix) = \sum_{k=0}^{\infty} \frac{(x/2)^{\nu+2k}}{k! \Gamma(\nu+k+1)}.$$

### 5.2.1.9 cylindrical Bessel functions (of the first kind)

[tr.math.sf.J]

```
double      cyl_bessel_j(double nu, double x);
float       cyl_bessel_jf(float nu, float x);
long double cyl_bessel_jl(long double nu, long double x);
```

**Effects:** These functions compute the cylindrical Bessel functions of the first kind of their respective arguments  $\nu$  and  $x$ . A domain error may occur if  $x$  is less than zero.

**Returns:** The `cyl_bessel_j` functions return

$$J_{\nu}(x) = \sum_{k=0}^{\infty} \frac{(-1)^k (x/2)^{\nu+2k}}{k! \Gamma(\nu+k+1)}.$$

### 5.2.1.10 irregular modified cylindrical Bessel functions

[tr.math.sf.K]

```
double      cyl_bessel_k(double nu, double x);
float       cyl_bessel_kf(float nu, float x);
long double cyl_bessel_kl(long double nu, long double x);
```

**Effects:** These functions compute the irregular modified cylindrical Bessel functions of their respective arguments  $\nu$  and  $x$ . A domain error may occur if  $x$  is less zero.

**Returns:** The `cyl_bessel_k` functions return

$$K_\nu(x) = (\pi/2)i^{\nu+1}(J_\nu(ix) + iN_\nu(ix)) = \begin{cases} \frac{\pi}{2} \frac{I_{-\nu}(x) - I_\nu(x)}{\sin \nu\pi} & \text{for non-integral } \nu \\ \frac{\pi}{2} \lim_{\mu \rightarrow \nu} \frac{I_{-\mu}(x) - I_\mu(x)}{\sin \mu\pi} & \text{for integral } \nu \end{cases} .$$

### 5.2.1.11 cylindrical Neumann functions

[tr.math.sf.N]

```
double    cyl_neumann(double nu, double x);
float     cyl_neumannf(float nu, float x);
long double cyl_neumannl(long double nu, long double x);
```

**Effects:** These functions compute the cylindrical Neumann functions, also known as the cylindrical Bessel functions of the second kind, of their respective arguments  $\nu$  and  $x$ . A domain error may occur if  $x$  is less than zero.

**Returns:** The `cyl_neumann` functions return

$$N_\nu(x) = \begin{cases} \frac{J_\nu(x) \cos \nu\pi - J_{-\nu}(x)}{\sin \nu\pi} & \text{for non-integral } \nu \\ \lim_{\mu \rightarrow \nu} \frac{J_\mu(x) \cos \mu\pi - J_{-\mu}(x)}{\sin \mu\pi} & \text{for integral } \nu \end{cases} .$$

### 5.2.1.12 (incomplete) elliptic integral of the first kind

[tr.math.sf.ellF]

```
double    ellint_1(double k, double phi);
float     ellint_1f(float k, float phi);
long double ellint_1l(long double k, long double phi);
```

**Effects:** These functions compute the incomplete elliptic integral of the first kind of their respective arguments  $k$  and  $\phi$ . A domain error may occur if the magnitude of  $k$  is greater than one.

**Returns:** The `ellint_1` functions return

$$F(k, \phi) = \int_0^\phi \frac{d\theta}{\sqrt{1 - k^2 \sin^2 \theta}} .$$

### 5.2.1.13 (incomplete) elliptic integral of the second kind

[tr.math.sf.ellE]

```
double    ellint_2(double k, double phi);
float     ellint_2f(float k, float phi);
long double ellint_2l(long double k, long double phi);
```

**Effects:** These functions compute the incomplete elliptic integral of the second kind of their respective arguments  $k$  and  $\phi$ . A domain error may occur if the magnitude of  $k$  is greater than one.

**Returns:** The `ellint_2` functions return

$$E(k, \phi) = \int_0^\phi \sqrt{1 - k^2 \sin^2 \theta} \, d\theta .$$

#### 5.2.1.14 (incomplete) elliptic integral of the third kind

[tr.math.sf.ellp]

```
double      ellint_3(double k, double nu, double phi);
float       ellint_3f(float k, float nu, float phi);
long double ellint_3l(long double k, long double nu, long double phi);
```

**Effects:** These functions compute the incomplete elliptic integral of the third kind of their respective arguments  $k$ ,  $\nu$ , and  $\phi$ . A domain error may occur if the magnitude of  $k$  is greater than one.

**Returns:** The `ellint_3` functions return

$$\Pi(\nu, k, \phi) = \int_0^\phi \frac{d\theta}{(1 - \nu \sin^2 \theta) \sqrt{1 - k^2 \sin^2 \theta}} .$$

#### 5.2.1.15 exponential integral

[tr.math.sf.ei]

```
double      expint(double x);
float       expintf(float x);
long double expintl(long double x);
```

**Effects:** These functions compute the exponential integral of their respective arguments  $x$ .

**Returns:** The `expint` functions return

$$\text{Ei}(x) = - \int_{-x}^{\infty} \frac{e^{-t}}{t} \, dt .$$

#### 5.2.1.16 Hermite polynomials

[tr.math.sf.Hn]

```
double      hermite(unsigned n, double x);
float       hermitef(unsigned n, float x);
long double hermitel(unsigned n, long double x);
```

**Effects:** These functions compute the Hermite polynomials of their respective arguments  $n$  and  $x$ .

**Returns:** The `hermite` functions return

$$H_n(x) = (-1)^n e^{x^2} \frac{d^n}{dx^n} e^{-x^2} .$$

#### 5.2.1.17 hypergeometric functions

[tr.math.sf.hyper]



```

double      hyperg(double a, double b, double c, double x);
float       hypergf(float a, float b, float c, float x);
long double hypergl(long double a, long double b, long double c,
                    long double x);

```

**Effects:** These functions compute the hypergeometric functions of their respective arguments a, b, c, and x. A domain error may occur if the magnitude of x is greater than or equal to one.

**Returns:** The hyperg functions return

$$F(a, b; c; x) = \frac{\Gamma(c)}{\Gamma(a)\Gamma(b)} \sum_{n=0}^{\infty} \frac{\Gamma(a+n)\Gamma(b+n)}{\Gamma(c+n)} \frac{x^n}{n!}.$$

#### 5.2.1.18 Laguerre polynomials

[tr.math.sf.Ln]

```

double      laguerre(unsigned n, double x);
float       laguerref(unsigned n, float x);
long double laguerrel(unsigned n, long double x);

```

**Effects:** These functions compute the Laguerre polynomials of their respective arguments n and x.

**Returns:** The laguerre functions return

$$L_n(x) = e^x \frac{d^n}{dx^n} (x^n e^{-x}).$$

#### 5.2.1.19 Legendre polynomials

[tr.math.sf.Pl]

```

double      legendre(unsigned l, double x)
float       legendref(unsigned l, float x)
long double legendrel(unsigned l, long double x)

```

**Effects:** These functions compute the Legendre polynomials of their respective arguments l and x. A domain error may occur if the magnitude of x is greater than one.

**Returns:** The legendre functions return

$$P_l(x) = \frac{1}{2^l l!} \frac{d^l}{dx^l} (x^2 - 1)^l.$$

#### 5.2.1.20 Riemann zeta function

[tr.math.sf.riemannzeta]

```

double      riemann_zeta(double x);
float       riemann_zetaf(float x);
long double riemann_zetal(long double x);

```

**Effects:** These functions compute the Riemann zeta function of their respective arguments x. A domain error occurs if x is equal to one.

**Returns:** The `riemann_zeta` functions return

$$\zeta(x) = \begin{cases} \sum_{k=1}^{\infty} k^{-x} & \text{for } x > 1 \\ 2^x \pi^{x-1} \sin\left(\frac{\pi x}{2}\right) \Gamma(1-x) \zeta(1-x) & \text{for } x < 1 \end{cases} .$$

### 5.2.1.21 spherical Bessel functions (of the first kind)

[tr.math.sf.j]

```
double    sph_bessel(unsigned n, double x);
float     sph_besself(unsigned n, float  x);
long double sph_bessell(unsigned n, long double x);
```

**Effects:** These functions compute the spherical Bessel functions of the first kind of their respective arguments `n` and `x`. A domain error may occur if `x` is less than zero.

**Returns:** The `sph_bessel` functions return

$$j_n(x) = (\pi/2x)^{1/2} J_{n+1/2}(x) .$$

### 5.2.1.22 spherical associated Legendre functions

[tr.math.sf.Ylm]

```
double    sph_legendre(unsigned l, unsigned m, double theta)
float     sph_legendref(unsigned l, unsigned m, float  theta)
long double sph_legendrel(unsigned l, unsigned m, long double theta)
```

**Effects:** These functions compute the spherical associated Legendre functions of their respective arguments `l`, `m`, and `theta`. A domain error occurs if the magnitude of `m` is greater than `l`.

**Returns:** The `sph_legendre` functions return

$$Y_l^m(\theta, 0)$$

where

$$Y_l^m(\theta, \phi) = (-1)^m \left[ \frac{(2l+1)}{4\pi} \frac{(l-m)!}{(l+m)!} \right]^{1/2} P_l^m(\cos \theta) e^{im\phi} .$$

[Note: This formulation avoids any need to return non-real numbers. End note.]

### 5.2.1.23 spherical Neumann functions

[tr.math.sf.n]

```
double    sph_neumann(unsigned n, double x);
float     sph_neumannf(unsigned n, float  x);
long double sph_neumannl(unsigned n, long double x);
```

**Effects:** These functions compute the spherical Neumann functions, also known as the spherical Bessel functions of the second kind, of their respective arguments  $n$  and  $x$ . A domain error may occur if  $x$  is less than zero.

**Returns:** The `sph_neumann` functions return

$$n_n(x) = (\pi/2x)^{1/2} N_{n+1/2}(x) .$$

### 5.2.2 Additions to header `<math.h>` synopsis [tr.math.sf.mathh]

The header `<math.h>` shall have sufficient additional using declarations to import into the global name space all of the function names declared in the previous section.

# Chapter 6

## Containers

[tr.cont]

### 6.1 Tuple types

[tr.tuple]

This clause describes the tuple library that provides a tuple type as the class template `tuple` that can be instantiated with any number of arguments. An implementation can set an upper limit for the number of arguments. The minimum value for this implementation quantity is defined in Annex B. Each template argument specifies the type of an element in the `tuple`. Consequently, tuples are heterogeneous, fixed-size collections of values.

#### 6.1.1 Header `<tuple>` synopsis

[tr.tuple.synopsis]

```
template <class T1 = implementation-defined,
          class T2 = implementation-defined,
          ...,
          class TM = implementation-defined> class tuple;
```

```
template<class T1, class T2, ..., class TM,
         class U1, class U2, ..., class UM>
bool operator==(const tuple<T1, T2, ..., TM>&,
               const tuple<U1, U2, ..., UM>&);
```

```
template<class T1, class T2, ..., class TM,
         class U1, class U2, ..., class UM>
bool operator!=(const tuple<T1, T2, ..., TM>&,
               const tuple<U1, U2, ..., UM>&);
```

```
template<class T1, class T2, ..., class TM,
         class U1, class U2, ..., class UM>
bool operator<(const tuple<T1, T2, ..., TM>&,
```

```

        const tuple<U1, U2, ..., UM>&);

template<class T1, class T2, ..., class TM,
        class U1, class U2, ..., class UM>
bool operator<=(const tuple<T1, T2, ..., TM>&,
               const tuple<U1, U2, ..., UM>&);

template<class T1, class T2, ..., class TM,
        class U1, class U2, ..., class UM>
bool operator>(const tuple<T1, T2, ..., TM>&,
               const tuple<U1, U2, ..., UM>&);

template<class T1, class T2, ..., class TM,
        class U1, class U2, ..., class UM>
bool operator>=(const tuple<T1, T2, ..., TM>&,
                const tuple<U1, U2, ..., UM>&);

template <class T> class tuple_size;

template <int I, class T> class tuple_element;

template <int I, class T1, class T2, ..., class TN>
RI get(tuple<T1, T2, ..., TN>&);

template <int I, class T1, class T2, ..., class TN>
PI get(const tuple<T1, T2, ..., TN>&);

template<class T1, class T2, ..., class TN>
tuple<VI, V2, ..., VN>
make_tuple(const T1&, const T2& , ..., const TN&);

template<class T1, class T2, ..., class TN>
tuple<T1&, T2&, ..., TN&> tie(T1&, T2& , ..., TN&);

template<class CharType, class CharTrait,
        class T1, class T2, ..., class TN>
basic_ostream<CharType, CharTrait>&
operator<<(basic_ostream<CharType, CharTrait>&,
          const tuple<T1, T2, ..., TN>&);

template<class CharType, class CharTrait,
        class T1, class T2, ..., class TN>
basic_istream<CharType, CharTrait>&
operator>>(basic_istream<CharType, CharTrait>&,
          tuple<T1, T2, ..., TN>&);

```

```

tuple_manip1 tuple_open(char_type c);
tuple_manip2 tuple_close(char_type c);
tuple_manip3 tuple_delimiter(char_type c);

```

## 6.1.2 Class template tuple

[tr.tuple.tuple]

M is used to denote the implementation-defined number of template type parameters to the tuple class template, and N is used to denote the number of template arguments specified in an instantiation.

[Example: Given the instantiation `tuple<int, float, char>`, N is 3. —end example]

```

template <class T1 = implementation-defined,
         class T2 = implementation-defined,
         ...,
         class TM = implementation-defined>
class tuple
{
public:
    tuple();
    explicit tuple(P1, P2, ..., PN); // iff N > 0

    tuple(const tuple&);

    template <class U1, class U2, ..., class UN>
    tuple(const tuple<U1, U2, ..., UN>&);

    template <class U1, class U2>
    tuple(const pair<U1, U2>&);

    tuple& operator=(const tuple&);

    template <class U1, class U2, ..., class UN>
    tuple& operator=(const tuple<U1, U2, ..., UN>&);

    template <class U1, class U2>
    tuple& operator=(const pair<U1, U2>&);
};

```

### 6.1.2.1 Construction

[tr.tuple.cnstr]

```
tuple();
```

**Requires:** Each tuple element type  $T_i$  can be default constructed.

**Effects:** Default initializes each element.

```
tuple(P1, P2, ..., PN);
```

Where, if  $T_i$  is a reference type then  $P_i$  is  $T_i$ , otherwise  $P_i$  is `const  $T_i$ &`.

**Requires:** Each tuple element type  $T_i$  is copy constructible.

**Effects:** Copy initializes each element with the value of the corresponding parameter.

```
tuple(const tuple& u);
```

**Requires:** all types  $T_i$  shall be copy constructible.

**Effects:** Copy constructs each element of `*this` with the corresponding element of `u`.

```
template <class U1, class U2, ..., class UN>
tuple(const tuple<U1, U2, ..., UN>& u);
```

**Requires:** Each type  $T_i$  shall be constructible from the corresponding type  $U_i$ .

**Effects:** Constructs each element of `*this` with the corresponding element of `u`.

[*Note:* In an implementation where one template definition serves for many different values for  $N$ , `enable_if` can be used to make the converting constructor and assignment operator exist only in the cases where the source and target have the same number of elements. Another way of achieving this is adding an extra integral template parameter which defaults to  $N$  (more precisely, a metafunction that computes  $N$ ), and then defining the converting copy constructor and assignment only for tuples where the extra parameter in the source is  $N$ . —*end note*]

```
template <class U1, class U2> tuple(const pair<U1, U2>& u);
```

**Requires:**  $T_1$  shall be constructible from  $U_1$ ,  $T_2$  shall be constructible from  $U_2$ .  $N == 2$ .

**Effects:** Constructs the first element with `u.first` and the second element with `u.second`.

```
tuple& operator=(const tuple& u);
```

**Requires:** All types  $T_i$  are assignable.

**Effects:** Assigns each element of `u` to the corresponding element of `*this`.

**Returns:** `*this`

```
template <class U1, class U2, ..., class UN>
tuple& operator=(const tuple<U1, U2, ..., UN>& u);
```

**Requires:** Each type  $T_i$  shall be assignable from the corresponding type  $U_i$ .

**Effects:** Assigns each element of `u` to the corresponding element of `*this`.

**Returns:** `*this`

```
template <class U1, class U2>
tuple& operator=(const pair<U1, U2>& u);
```

**Requires:**  $T_1$  shall be assignable from  $U_1$ ,  $T_2$  shall be assignable from  $U_2$ .  $N == 2$ .

**Effects:** Assigns `u.first` to the first element of `*this` and `u.second` to the second element of `*this`.

**Returns:** `*this`

[*Note:* There seem to exist (rare) conditions where the converting copy constructor is a better match than the element-wise construction, even though the user might intend differently. An example of this is if one is constructing a one-element tuple where the element type is another tuple type  $T$  and if the parameter passed to the constructor is not of type  $T$ , but rather a tuple type that is convertible to  $T$ . The effect of the

converting copy construction is most likely the same as the effect of the element-wise construction would have been. However, it is possible to compare the 'nesting depths' of the source and target tuples and decide to select the element-wise constructor if the source nesting depth is smaller than the target nesting-depth. This can be accomplished using an `enable_if` template or other tools for constrained templates. —*end note*]

### 6.1.2.2 Tuple creation functions

[tr.tuple.helper]

```
template<class T1, class T2, ..., class TN>
tuple<V1, V2, ..., VN>
make_tuple(const T1& t1, const T2& t2, ..., const TN& tn);
```

where  $V_i$  is  $X\&$ , if the cv-qualified type  $T_i$  is `reference_wrapper<X>`, otherwise  $V_i$  is  $T_i$ .

The `make_tuple` function template shall be implemented for each different number of arguments from 0 to the maximum number of allowed tuple elements.

**Returns:** `tuple<V1, V2, ..., VN>(t1, t2, ..., tn)`.

[*Example:*

```
int i; float j;
make_tuple(1, ref(i), cref(j))
```

creates a tuple of type

```
tuple<int, int&, const float&>
```

—*end example*]

```
template<class T1, class T2, ..., class TN>
tuple<T1&, T2&, ..., TN> tie(T1& t1, T2& t2, ..., TN& tn);
```

The `tie` function template shall be implemented for each different number of arguments from 0 to the maximum number of allowed tuple elements.

**Returns:** `tuple<T1&, T2&, ..., TN&>(t1, t2, ..., tn)`

[*Example:*

`tie` functions allow one to create tuples that unpack tuples into variables. `ignore` can be used for elements that are not needed:

```
int i; std::string s;
tie(i, ignore, s) = make_tuple(42, 3.14, "C++");
// i == 42, s == "C++"
```

—*end example*]

### 6.1.2.3 Valid expressions for tuple types

[tr.tuple.expr]

```
tuple_size<T>::value
```



**Requires:** T is an instantiation of class template tuple.

**Type:** integral constant expression.

**Value:** Number of elements in T.

```
tuple\_element<I, T>::type
```

**Requires:**  $0 \leq I < \text{tuple\_size}<T>::\text{value}$ . The program is ill-formed if I is out of bounds.

**Value:** The type of the Ith element of T, where indexing is zero-based.

#### 6.1.2.4 Element access

[tr.tuple.elem]

```
template <int I, class T1, class T2, ..., class TN>  
RI get(tuple<T1, T2, ..., TN>& t);
```

**Requires:**  $0 \leq I < N$ . The program is ill-formed if I is out of bounds.

**Return type:** RI. If TI is a reference type, then RI is TI, otherwise RI is TI&.

**Returns:** A reference to the Ith element of t, where indexing is zero-based.

```
template <int I, class T1, class T2, ..., class TN>  
PI get(const tuple<T1, T2, ..., TN>& t);
```

**Requires:**  $0 \leq I < N$ . The program is ill-formed if I is out of bounds.

**Return type:** PI. If TI is a reference type, then PI is TI, otherwise PI is const TI&.

**Returns:** A const reference to the Ith element of t, where indexing is zero-based.

[*Note:* Constness is shallow. If TI is some reference type X&, the return type is X&, not const X&. However, if the element type is non-reference type T, the return type is const T&. This is consistent with how constness is defined to work for member variables of reference type. —end note.]

[*Note:* Implementing get as a member function of tuple, would require using the template keyword in invocations where the type of the tuple object is dependent on a template parameter. For example: t.template get<1>(); —end note]

#### 6.1.2.5 Equality and inequality comparisons

[tr.tuple.eq]

```
template<class T1, class T2, ..., class TM,  
         class U1, class U2, ..., class UM>  
bool operator==(const tuple<T1, T2, ..., TM>& t,  
               const tuple<U1, U2, ..., UM>& u);
```

**Requires:**  $\text{tuple\_size}<\text{tuple}<T1, T2, \dots, TM> >::\text{value} == \text{tuple\_size}<\text{tuple}<U1, U2, \dots, UM> >::\text{value} == N$ . For all i, where  $0 \leq i < N$ ,  $\text{get}<i>(t) == \text{get}<i>(u)$  is a valid expression returning a type that is convertible to bool.

**Return type:** bool

**Returns:** true iff  $\text{get}<i>(t) == \text{get}<i>(u)$  for all i. For any two zero-length tuples e and f,  $e == f$  returns true.

**Effects:** The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first equality comparison that evaluates to false.

```

template<class T1, class T2, ..., class TM,
        class U1, class U2, ..., class UM>
bool operator!=(const tuple<T1, T2, ..., TM>& t,
               const tuple<U1, U2, ..., UM>& u);

```

**Requires:** `tuple_size<tuple<T1, T2, ..., TM> >::value == tuple_size<tuple<U1, U2, ..., UM> >::value == N`. For all  $i$ , where  $0 \leq i < N$ , `get<i>(t) != get<i>(u)` is a valid expression returning a type that is convertible to `bool`.

**Return type:** `bool`

**Returns:** true iff `get<i>(t) != get<i>(u)` for any  $i$ . For any two zero-length tuples  $e$  and  $f$ , `e != f` returns false.

**Effects:** The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first inequality comparison that evaluates to true.

### 6.1.2.6 <, > comparisons

[tr.tuple.lt]

```

template<class T1, class T2, ..., class TN,
        class U1, class U2, ..., class UN>
bool operator<(const tuple<T1, T2, ..., TN>&,
              const tuple<U1, U2, ..., UN>&);

```

```

template<class T1, class T2, ..., class TN,
        class U1, class U2, ..., class UN>
bool operator>(const tuple<T1, T2, ..., TN>&,
              const tuple<U1, U2, ..., UN>&);

```

**Requires:** `tuple_size<tuple<T1, T2, ..., TM> >::value == tuple_size<tuple<U1, U2, ..., UM> >::value == N`. For all  $i$ , where  $0 \leq i < N$ , `get<i>(t)  $\odot$  get<i>(u)` is a valid expression returning a type that is convertible to `bool`, where  $\odot$  is either `<` or `>`.

**Return type:** `bool`

**Returns:** The result of a lexicographical comparison with  $\odot$  between  $t$  and  $u$ , defined equivalently to:  
`(bool)(get<0>(t)  $\odot$  get<0>(u)) || !((bool)(get<0>(u)  $\odot$  get<0>(t)) && ttail  $\odot$  utail,`

where  $r_{\text{tail}}$  for some tuple  $r$  is a tuple containing all but the first element of  $r$ . For any two zero-length tuples  $e$  and  $f$ , `e  $\odot$  f` returns false.

### 6.1.2.7 <= and >= comparisons

[tr.tuple.le]

```

template<class T1, class T2, ..., class TN,
        class U1, class U2, ..., class UN>
bool operator<=(const tuple<T1, T2, ..., TN>&,
               const tuple<U1, U2, ..., UN>&);

```

```

template<class T1, class T2, ..., class TN,
        class U1, class U2, ..., class UN>

```

```
bool operator>=(const tuple<T1, T2, ..., TN>&,
               const tuple<U1, U2, ..., UN>&);
```

**Requires:** `tuple_size<tuple<T1, T2, ..., TM> >::value == tuple_size<tuple<U1, U2, ..., UM> >::value == N`. For all  $i$ , where  $0 \leq i < N$ , `get<i>(t)  $\odot$  get<i>(u)` is a valid expression returning a type that is convertible to `bool`, where  $\odot$  is either `<=` or `>=`.

**Returns:** The result of a lexicographical comparison with  $\odot$  between  $t$  and  $u$ , defined equivalently to: `(bool)(get<0>(t)  $\odot$  get<0>(u)) && (!(bool)(get<0>(u)  $\odot$  get<0>(t)) || ttail  $\odot$  utail)`, where  $r_{\text{tail}}$  for some tuple  $r$  is a tuple containing all but the first element of  $r$ . For any two zero-length tuples  $e$  and  $f$ , `e  $\odot$  f` returns `true`.

**Notes:** The above definitions for comparison operators do not impose the requirement that  $t_{\text{tail}}$  (or  $u_{\text{tail}}$ ) shall be constructed. It may be even impossible, as  $t$  (or  $u$ ) is not required to be copy constructible. Also, all comparison operators are short circuited to not perform element accesses beyond what is required to determine the result of the comparison.

### 6.1.2.8 Input and output

[tr.tuple.io]

```
template<class CharType, class CharTrait,
         class T1, class T2, ..., class TN>
basic_ostream<CharType, CharTrait>&
operator<<(basic_ostream<CharType, CharTrait>& os,
          const tuple<T1, T2, ..., TN>& t);
```

**Requires:** For all  $i = 0, 1, \dots, N-1$  in `os << get<i>(t)` is a valid expression.

**Effects:** Inserts  $t$  into `os` as  $Lt_0dt_1d \dots dt_nR$ , where  $L$  is the opening,  $d$  the delimiter and  $R$  the closing character set by tuple formatting manipulators. Each element  $t_i$  is output by invoking `os << get<i>(t)`. A zero-element tuple is output as  $LR$  and a one-element tuple is output as  $Lt_0R$ .

**Returns:** `os`

```
template<class CharType, class CharTrait,
         class T1, class T2, ..., class TN>
basic_istream<CharType, CharTrait>&
operator>>(basic_istream<CharType, CharTrait>& is,
          tuple<T1, T2, ..., TN>& t);
```

**Requires:** For all  $i = 0, 1, \dots, N-1$  in `is >> get<i>(t)` is a valid expression.

**Effects:** Extracts a tuple of the form  $Lt_0dt_1d \dots dt_nR$ , where  $L$  is the opening,  $d$  the delimiter and  $R$  the closing character set by tuple formatting manipulators. Each element  $t_i$  is extracted by invoking `is >> get<i>(t)`. A zero-element tuple expects to extract  $LR$  from the stream and one-element tuple expects to extract  $Lt_0R$ . If bad input is encountered, calls `is.set_state(ios::failbit)` (which may throw `ios::failure` (27.4.4.3)).

**Returns:** `is`

**Notes:** It is not guaranteed that a tuple written to a stream can be extracted back to a tuple of the same type.

### 6.1.2.9 Tuple formatting manipulators

[tr.tuple.form]

The library defines the following three stream manipulator functions. The types designated *tuple\_manip1*,

*tuple\_manip2* and *tuple\_manip3* are implementation-specified.

```
tuple_manip1 tuple_open(char_type c);  
tuple_manip2 tuple_close(char_type c);  
tuple_manip3 tuple_delimiter(char_type c);
```

**Returns:** Each of these functions returns an object *s* of unspecified type such that if *out* is an instance of `basic_ostream<charT,traits>`, *in* is an instance of `basic_istream<charT,traits>` and *char\_type* equals `charT`, then the expression `out << s` (respectively `in >> s`) sets *c* to be the opening, closing, or delimiter character (depending on the manipulator function called) to be used when writing tuples into *out* (respectively extracting tuples from *in*).

**Notes:** Implementations are not required to support these manipulators for streams with `sizeof(charT) > sizeof(long)`; `out << s` and `in >> s` are required to fail at compile time if *out* and *in* are such streams and the implementation does not support tuple formatting manipulators for them.

[*Note:* The constraint stated in the above **Notes** section allows an implementation where the delimiter characters are stored in space allocated by `xalloc`, which allocates an array of longs. A more general alternative is to store pointers to the delimiter characters in the `xalloc`-allocated array, and register a callback function (with `ios_base::register_callback`) for the stream to take care of deallocating the memory. If this approach is taken, the delimiters could be chosen to be strings instead of single characters. This might be worthwhile, such as to allow delimiters like `" , "`. —*end note*]

### 6.1.3 Pairs

[tr.tuple.pairs]

This is an *impure* extension (as defined in section 1.2) to the standard library class template `std::pair`.

```
template<class T1, class T2>  
struct tuple_size<pair<T1, T2> > {  
    static const int value = 2;  
};  
  
template<class T1, class T2>  
struct tuple_element<0, pair<T1, T2> > {  
    typedef T1 type;  
};  
  
template<class T1, class T2>  
struct tuple_element<1, pair<T1, T2> > {  
    typedef T2 type;  
};  
  
template<int I, class T1, class T2>  
P& get(pair<T1, T2>&);  
  
template<int I, class T1, class T2>  
const P& get(const pair<T1, T2>&);
```

**Return type:** If I is 0 then P is T1, if I is 1 then P is T2, otherwise the program is ill-formed.

**Returns:** If I == 0 returns p.first, otherwise returns p.second.

### 6.1.4 Implementation quantities

[tr.tuple.lim]

The maximum number of elements in one tuple type is implementation defined. This limits should be at least 10.

## 6.2 Fixed size array

[tr.array]

### 6.2.1 Header <array> synopsis

[tr.array.syn]

```
namespace tr1 {
    template <class T, size_t N > struct array;
    template <class T, size_t N >
        bool operator==
            (const array<T,N>& x, const array<T,N>& y);
    template <class T, size_t N >
        bool operator<
            (const array<T,N>& x, const array<T,N>& y);
    template <class T, size_t N >
        bool operator!=
            (const array<T,N>& x, const array<T,N>& y);
    template <class T, size_t N >
        bool operator>
            (const array<T,N>& x, const array<T,N>& y);
    template <class T, size_t N >
        bool operator>=
            (const array<T,N>& x, const array<T,N>& y);
    template <class T, size_t N >
        bool operator<=
            (const array<T,N>& x, const array<T,N>& y);
    template <class T, size_t N >
        void swap(array<T,N>& x, array<T,N>& y);
}
```

### 6.2.2 Class template array

[tr.array.array]

The header <array> defines a class template for storing fixed-size sequences of objects. An array supports random access iterators. An instance of array<T, N> stores N elements of type T, so that size() == N is an invariant. The elements of an array are stored contiguously, meaning that if a is an array<T, N> then it obeys the identity &a[n] == &a[0] + n for all 0 <= n < N.

An array is an aggregate (8.5.1) that can be initialized with the syntax

```
array a = { initializer-list };
```

where *initializer-list* is a comma separated list of up to N elements of type convertible-to-T.

Unless specified others, all array operations are as described in 23.1 [lib.container.requirements]. Descriptions are provided here only for operations on array that are not described in this clause or for operations where there is additional semantic information.

```
namespace tr1 {
    template <class T, size_t N >
    struct array {
        // types:
        typedef T & reference;
        typedef const T & const_reference;
        typedef implementation defined iterator;
        typedef implementation defined const_iterator;
        typedef size_t size_type;
        typedef ptrdiff_t difference_type;
        typedef T value_type;
        typedef std::reverse_iterator<iterator> reverse_iterator;
        typedef std::reverse_iterator<const_iterator> const_reverse_iterator;

        T elems[N];

        // No explicit construct/copy/destroy for aggregate type

        void assign(const T& u);
        void swap( array<T, N> &);

        // iterators:
        iterator begin();
        const_iterator begin() const;
        iterator end();
        const_iterator end() const;
        reverse_iterator rbegin();
        const_reverse_iterator rbegin() const;
        reverse_iterator rend();
        const_reverse_iterator rend() const;

        // capacity:
        size_type size() const;
        size_type max_size() const;
        bool empty() const;

        // element access:
```

```

reference      operator[](size_type n);
const_reference operator[](size_type n) const;
const_reference at(size_type n) const;
reference      at(size_type n);
reference      front();
const_reference front() const;
reference      back();
const_reference back() const;

T *           data();
const T *     data() const;
};

template <class T, size_t N>
    bool operator==(const array<T,N>& x,
                    const array<T,N>& y);
template <class T, size_t N>
    bool operator< (const array<T,N>& x,
                    const array<T,N>& y);
template <class T, size_t N>
    bool operator!=(const array<T,N>& x,
                    const array<T,N>& y);
template <class T, size_t N>
    bool operator> (const array<T,N>& x,
                    const array<T,N>& y);
template <class T, size_t N>
    bool operator>=(const array<T,N>& x,
                    const array<T,N>& y);
template <class T, size_t N>
    bool operator<=(const array<T,N>& x,
                    const array<T,N>& y);

// specialized algorithms:
template <class T, size_t N>
    void swap(array<T,N>& x, array<T,N>& y);
}

```

### 6.2.2.1 array constructors, copy, and assignment

[tr.array.cons]

**Initialization:** The conditions for an aggregate (8.5.1) must be met. Class `array` relies on the implicitly-declared special member functions (12.1, 12.4, and 12.8) to satisfy the requirements of table 65.

### 6.2.2.2 array specialized algorithms

[tr.array.special]

```
template <class T, size_t N>
void swap(array<T,N>& x, array<T,N>& y);
```

**Effects:**

```
swap_ranges(x.begin(), x.end(), y.begin() );
```

### 6.2.2.3 array size

[tr.array.size]

```
template <class T, size_t N>
size_type array<T,N>::size();
```

**Returns:** N

### 6.2.2.4 Zero sized arrays

[tr.array.zero]

array shall provide support for the special case  $N == 0$ .

In the case that  $N == 0$ :

- `elems` is not required.
- `begin() == end() == unique value`.

## 6.3 Unordered associative containers

[tr.hash]

### 6.3.1 Unordered associative container requirements

[tr.unord.req]

Unordered associative containers provide an ability for fast retrieval of data based on keys. The worst-case complexity for most operations is linear, but the average case is much faster. The library provides four basic kinds of hashed associative containers: `unordered_set`, `unordered_map`, `unordered_multiset`, and `unordered_multimap`.

Each hashed associative container is parameterized by `Key`, by a function object `Hash` that acts as a hash function for values of type `Key`, and on a binary predicate `Pred` that induces an equivalence relation on values of type `Key`. Additionally, `unordered_map` and `unordered_multimap` associate an arbitrary *mapped type* `T` with the `Key`.

A hash function is a function object that takes a single argument of type `Key` and returns a value of type `std::size_t` in the range  $[0, \text{std::numeric\_limits}<\text{std::size\_t}>::\text{max}())$ .

Two values `k1` and `k2` of type `Key` are considered equal if the container's equality function object returns `true` when passed those values. If `k1` and `k2` are equal, the hash function shall return the same value for both.

A hashed associative container supports *unique keys* if it may contain at most one element for each key. Otherwise, it supports *equivalent keys*. `unordered_set` and `unordered_map` support unique keys.



`unordered_multiset` and `unordered_multimap` support equivalent keys. In containers that support equivalent keys, elements with equivalent keys are adjacent to each other.

For `unordered_set` and `unordered_multiset` the value type is the same as the key type. For `unordered_map` and `unordered_multimap` it is equal to `std::pair<const Key, T>`.

The elements of a hashed associative container are organized into *buckets*. Keys with the same hash code appear in the same bucket. The number of buckets is automatically increased as elements are added to a hashed associative container, so that the average number of elements per bucket is kept below a bound. Rehashing invalidates iterators, changes ordering between elements, and changes which buckets elements appear in, but does not invalidate pointers or references to elements.

In table 6.1:

X is a hashed associative container class, a is an object of type X, b is a possibly const object of type X, `a_unique` is an object of type X when X supports unique keys, `a_eq` is an object of type X when X supports equivalent keys, i and j are input iterators that refer to `value_type`, [i, j) is a valid range, p and q2 are valid iterators to a, q and q1 are valid dereferenceable iterators to a, [q1, q2) is a valid range in a, r and r1 are valid dereferenceable const iterators to a, r2 is a valid const iterator to a, [r1, r2) is a valid range in a, t is a value of type `X::value_type`, k is a value of type `key_type`, hf is a possibly const value of type `hasher`, eq is a possibly const value of type `key_equal`, n is a value of type `size_type`, and z is a value of type `float`.

Table 6.1: Unordered associative container requirements (in addition to container)

expression	Return Type	assertion/note/pre/post-condition	complexity
<code>X::key_type</code>	Key	Key is Assignable and CopyConstructible	compile time
<code>X::hasher</code>	Hash	Hash is a unary function object that take an argument of type Key and returns a value of type <code>std::size_t</code> .	compile time
<code>X::key_equal</code>	Pred	Pred is a binary predicate that takes two arguments of type Key. Pred is an equivalence relation.	compile time
<code>X::local_iterator</code>	An iterator type whose category, value type, difference type, and pointer and reference types are the same as <code>X::iterator</code> 's.	A <code>local_iterator</code> object may be used to iterate through a single bucket, but may not be used to iterate across buckets.	compile time
<code>X::const_local_iterator</code>	An iterator type whose category, value type, difference type, and pointer and reference types are the same as <code>X::const_iterator</code> 's.	A <code>const_local_iterator</code> object may be used to iterate through a single bucket, but may not be used to iterated across buckets.	compile time

<i>continued from previous page</i>			
X(n, hf, eq) X a(n, hf, eq)	X	Constructs an empty container with at least n buckets, using hf as the hash function and eq as the key equality predicate.	$\mathcal{O}(n)$
X(n, hf) X a(n, hf)	X	Constructs an empty container with at least n buckets, using hf as the hash function and key_equal() as the key equality predicate.	$\mathcal{O}(n)$
X(n) X a(n)	X	Constructs an empty container with at least n buckets, using hasher() as the hash function and key_equal() as the key equality predicate.	$\mathcal{O}(n)$
X() X a	X	Constructs an empty container with an unspecified number of buckets, using hasher() as the hash function and key_equal as the key equality predicate.	constant
X(i, j, n, hf, eq) X a(i, j, n, hf, eq)	X	Constructs an empty container with at least n buckets, using hf as the hash function and eq as the key equality predicate, and inserts elements from [i, j) into it.	Average case $\mathcal{O}(N)$ (N is distance(i, j)), worst case $\mathcal{O}(N^2)$
X(i, j, n, hf) X a(i, j, n, hf)	X	Constructs an empty container with at least n buckets, using hf as the hash function and key_equal() as the key equality predicate, and inserts elements from [i, j) into it.	Average case $\mathcal{O}(N)$ (N is distance(i, j)), worst case $\mathcal{O}(N^2)$
X(i, j, n) X a(i, j, n)	X	Constructs an empty container with at least n buckets, using hasher() as the hash function and key_equal() as the key equality predicate, and inserts elements from [i, j) into it.	Average case $\mathcal{O}(N)$ (N is distance(i, j)), worst case $\mathcal{O}(N^2)$

continued from previous page

<code>X(i, j)</code> <code>X a(i, j)</code>	X	Constructs an empty container with an unspecified number of buckets, using <code>hasher()</code> as the hash function and <code>key_equal</code> as the key equality predicate, and inserts elements from <code>[i, j)</code> into it.	Average case $\mathcal{O}(N)$ ( $N$ is <code>distance(i, j)</code> ), worst case $\mathcal{O}(N^2)$
<code>X(b)</code> <code>X a(b)</code>	X	Copy constructor. In addition to the contained elements, the hash function, predicate, and maximum load factor are copied.	Average case linear in <code>b.size()</code> , worst case quadratic.
<code>a = b</code>	X	Copy assignment operator. In addition to the contained elements, the hash function, predicate, and maximum load factor are copied.	Average case linear in <code>b.size()</code> , worst case quadratic.
<code>b.hash_function()</code>	<code>hasher</code>	Returns the hash function out of which <code>b</code> was constructed.	constant
<code>b.key_eq()</code>	<code>key_equal</code>	Returns the key equality function out of which <code>b</code> was constructed.	constant
<code>a_uniq.insert(t)</code>	<code>pair&lt;iterator, bool&gt;</code>	Inserts <code>t</code> if and only if there is no element in the container with key equivalent to the key of <code>t</code> . The <code>bool</code> component of the returned pair indicates whether the insertion takes place, and the <code>iterator</code> component points to the element with key equivalent to the key of <code>t</code> .	Average case $\mathcal{O}(1)$ , worst case $\mathcal{O}(a\_uniq.size())$ .
<code>a_eq.insert(t)</code>	<code>iterator</code>	Inserts <code>t</code> , and returns an iterator pointing to the newly inserted element.	Average case $\mathcal{O}(1)$ , worst case $\mathcal{O}(a\_uniq.size())$ .
<code>a.insert(r, t)</code>	<code>iterator</code>	Equivalent to <code>a.insert(t)</code> . Return value is an iterator pointing to the element with the key equivalent to that of <code>t</code> . The <code>const iterator r</code> is a hint pointing to where the search should start. Implementations are permitted to ignore the hint.	Average case $\mathcal{O}(1)$ , worst case $\mathcal{O}(a\_uniq.size())$ .

continued from previous page

<code>a.insert(i, j)</code>	<code>void</code>	Pre: <code>i</code> and <code>j</code> are not iterators in <code>a</code> . Equivalent to <code>a.insert(t)</code> for each element in <code>[i, j)</code> .	Average case $\mathcal{O}(N)$ , where $N$ is <code>distance(i, j)</code> . Worst case $\mathcal{O}(N * a.size())$ .
<code>a.erase(k)</code>	<code>size_type</code>	Erases all elements with key equivalent to <code>k</code> . Returns the number of elements erased.	Average case $\mathcal{O}(a.count(k))$ . Worst case $\mathcal{O}(a.size())$ .
<code>a.erase(r)</code>	<code>void</code>	Erases the element pointed to by <code>r</code> .	Average case $\mathcal{O}(1)$ , worst case $\mathcal{O}(a.size())$ .
<code>a.erase(r1, r2)</code>	<code>void</code>	Erases all elements in the range <code>[r1, r2)</code> .	Average case $\mathcal{O}(N)$ , where $N$ is <code>distance(r1, r2)</code> , worst case $\mathcal{O}(a.size())$ .
<code>a.clear()</code>	<code>void</code>	Erases all elements in the container. Post: <code>a.size() == 0</code>	Linear.
<code>b.find(k)</code>	<code>iterator;</code> <code>const_iterator</code> for <code>const b</code> .	Returns an iterator pointing to an element with key equivalent to <code>k</code> , or <code>b.end()</code> if no such element exists.	Average case $\mathcal{O}(1)$ , worst case $\mathcal{O}(b.size())$ .
<code>b.count(k)</code>	<code>size_type</code>	Returns the number of elements with key equivalent to <code>k</code> .	Average case $\mathcal{O}(1)$ , worst case $\mathcal{O}(b.size())$ .
<code>b.equal_range(k)</code>	<code>pair&lt;iterator,</code> <code>iterator&gt;;</code> <code>pair&lt;const_</code> <code>iterator, const_</code> <code>iterator&gt;</code> for <code>const</code> <code>b</code> .	Returns a range containing all elements with keys equivalent to <code>k</code> . Returns <code>make_pair(b.end(), b.end())</code> if no such elements exist.	Average case $\mathcal{O}(b.count(k))$ . Worst case $\mathcal{O}(b.size())$ .
<code>b.bucket_count()</code>	<code>size_type</code>	Returns the number of buckets that <code>b</code> contains.	Constant
<code>b.max_bucket_count()</code>	<code>size_type</code>	Returns an upper bound on the number of buckets that <code>b</code> might ever contain.	Constant

<i>continued from previous page</i>			
<code>b.bucket(k)</code>	<code>size_type</code>	Returns the index of the bucket in which elements with keys equivalent to <code>k</code> would be found, if any such element existed. Post: the return value is in the range $[0, b.bucket\_count())$ .	Constant
<code>b.bucket_size(n)</code>	<code>size_type</code>	Pre: <code>n</code> is in the range $[0, b.bucket\_count())$ . Returns the number of elements in the $n^{\text{th}}$ bucket.	$\mathcal{O}(b.bucket\_size(n))$
<code>b.begin(n)</code>	<code>local_iterator;</code> <code>const_local_</code> <code>iterator for const</code> <code>b.</code>	Pre: <code>n</code> is in the range $[0, b.bucket\_count())$ . Note: $[b.begin(n), b.end(n))$ is a valid range containing all of the elements in the $n^{\text{th}}$ bucket.	Constant
<code>b.end(n)</code>	<code>local_iterator;</code> <code>const_local_</code> <code>iterator for const</code> <code>b.</code>	Pre: <code>n</code> is in the range $[0, b.bucket\_count())$ .	Constant
<code>b.load_factor()</code>	<code>float</code>	Returns the average number of elements per bucket. $\zeta$	Constant
<code>b.max_load_factor()</code>	<code>float</code>	Returns a number that the container attempts to keep the load factor less than or equal to. The container automatically increases the number of buckets as necessary to keep the load factor below this number. Post: return value is positive.	Constant
<code>a.max_load_factor(z)</code>	<code>void</code>	Pre: <code>z</code> is positive. Changes the container's maximum load factor, using <code>z</code> as a hint.	Constant
<code>a.rehash(n)</code>	<code>void</code>	Pre: $n > a.size() / a.max\_load\_factor()$ . Changes the number of buckets so that it is at least <code>n</code> .	Average case linear in <code>a.size()</code> , worst case quadratic.

Unordered associative containers are not required to support the expressions `a == b` or `a != b`. [Note: This is because the container requirements define operator equality in terms of equality of ranges. Since the elements of an unordered associative container appear in an arbitrary order, range equality is not a useful operation. —end note]

The iterator types `iterator` and `const_iterator` of an unordered associative container are of at

least the forward iterator category. For unordered associative containers where the key type and value type are the same, both `iterator` and `const_iterator` are const iterators.

The insert members shall not affect the validity of references to container elements, but may invalidate all iterators to the container. The erase members shall invalidate only iterators and references to the erased elements.

### 6.3.1.1 Exception safety guarantees

[tr.unord.req.except]

- For unordered associative containers, no `clear()` function throws an exception. No `erase()` function throws an exception unless that exception is thrown by the container's Hash or Pred object (if any).
- For unordered associative containers, if an exception is thrown by an `insert()` function while inserting a single element other than by the container's hash function, the `insert()` function has no effects.
- For unordered associative containers, no `swap` function throws an exception unless that exception is thrown by the copy constructor or copy assignment operator of the container's Hash or Pred object (if any).

### 6.3.2 Additions to header `<functional>` synopsis

[tr.unord.fun.syn]

```
namespace tr1 {
    // Hash function base template
    template <class T> struct hash;

    // Hash function specializations

    template <> struct hash<bool>;
    template <> struct hash<char>;
    template <> struct hash<signed char>;
    template <> struct hash<unsigned char>;
    template <> struct hash<wchar_t>;
    template <> struct hash<short>;
    template <> struct hash<int>;
    template <> struct hash<long>;
    template <> struct hash<unsigned short>;
    template <> struct hash<unsigned int>;
    template <> struct hash<unsigned long>;

    template <> struct hash<float>;
    template <> struct hash<double>;
    template <> struct hash<long double>;

    template<class T>
    struct hash<T*>
```

```

template <class charT, class traits, class Allocator>
struct hash<std::basic_string<charT, traits, Allocator> >;
}

```

### 6.3.3 Class template hash

[tr.unord.hash]

The function object hash is used as the default hash function by the *unordered associative containers*. This class template is only required to be instantiable for integer types (3.9.1), floating point types (3.9.1), pointer types (8.3.1), and (for any valid set of charT, traits, and Alloc) std::basic\_string<charT, traits, Alloc>.

```

template <class T>
struct hash : public std::unary_function<T, std::size_t>
{
    std::size_t operator()(T val) const;
};

```

The return value of operator() is unspecified, except that equal arguments yield the same result. operator() shall not throw exceptions.

### 6.3.4 Unordered associative container classes

[tr.unord.unord]

#### 6.3.4.1 Header <unordered\_set> synopsis

[tr.unord.syn.set]

```

namespace tr1 {
    template <class Value,
              class Hash = hash<Value>,
              class Pred = std::equal_to<Value>,
              class Alloc = std::allocator<Value> >
    class unordered_set;

    template <class Value,
              class Hash = hash<Value>,
              class Pred = std::equal_to<Value>,
              class Alloc = std::allocator<Value> >
    class unordered_multiset;
}

```

#### 6.3.4.2 Header <unordered\_map> synopsis

[tr.unord.syn.map]

```

namespace tr1 {
    template <class Key,
              class T,
              class Hash = hash<Key>,
              class Pred = std::equal_to<Key>,
              class Alloc = std::allocator<std::pair<const Key, T> > >
    class unordered_map;

    template <class Key,
              class T,
              class Hash = hash<Key>,
              class Pred = std::equal_to<Key>,
              class Alloc = std::allocator<std::pair<const Key, T> > >
    class unordered_multimap;
}

```

### 6.3.4.3 Class template `unordered_set`

[tr.unord.set]

An `unordered_set` is a kind of unordered associative container that supports unique keys (an `unordered_set` contains at most one of each key value) and in which the elements' keys are the elements themselves.

An `unordered_set` satisfies all of the requirements of a container and of a unordered associative container. It provides the operations described in the preceding requirements table for unique keys; that is, an `unordered_set` supports the `a_uniq` operations in that table, not the `a_eq` operations. For a `unordered_set<Value>` the key type and the value type are both `Value`. The iterator and `const_iterator` types are both `const` iterator types. It is unspecified whether or not they are the same type.

This section only describes operations on `unordered_set` that are not described in one of the requirement tables, or for which there is additional semantic information.

```

template <class Value,
          class Hash = hash<Value>,
          class Pred = std::equal_to<Value>,
          class Alloc = std::allocator<Value> >
class unordered_set
{
public:
    // types
    typedef Value          key_type;
    typedef Value          value_type;
    typedef Hash           hasher;
    typedef Pred           key_equal;
    typedef Alloc          allocator_type;
    typedef typename allocator_type::pointer      pointer;
    typedef typename allocator_type::const_pointer const_pointer;
    typedef typename allocator_type::reference    reference;

```



```

typedef typename allocator_type::const_reference const_reference;
typedef implementation-defined size_type;
typedef implementation-defined difference_type;

typedef implementation-defined iterator;
typedef implementation-defined const_iterator;
typedef implementation-defined local_iterator;
typedef implementation-defined const_local_iterator;

// construct/destroy/copy
explicit unordered_set(size_type n = implementation-defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
template <class InputIterator>
    unordered_set(InputIterator f, InputIterator l,
                  size_type n = implementation-defined,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
unordered_set(const unordered_set&);
~unordered_set();
unordered_set& operator=(const unordered_set&);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator      begin();
const_iterator begin() const;
iterator      end();
const_iterator end() const;

// modifiers
std::pair<iterator, bool> insert(const value_type& obj);
iterator insert(const_iterator hint, const value_type& obj);
template <class InputIterator>
    void insert(InputIterator first, InputIterator last);

void erase(const_iterator position);
size_type erase(const key_type& k);
void erase(const_iterator first, const_iterator last);

```

```

void clear();

void swap(unordered_set&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
std::pair<iterator, iterator>
    equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>
    equal_range(const key_type& k) const;

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n) const;
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n);
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float z);
void rehash(size_type n);
};

template <class Value, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
              unordered_set<Value, Hash, Pred, Alloc>& y);

```

### 6.3.4.3.1 unordered\_set constructors

[tr.unord.set.cnstr]

```

explicit unordered_set(size_type n = implementation-defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());

```

**Effects:** Constructs an empty `unordered_set` using the specified hash function, key equality function, and allocator, and using at least  $n$  buckets. If  $n$  is not provided, the number of buckets is implementation defined. `max_load_factor()` is 1.0.

**Complexity:** Constant.

```
template <class InputIterator>
    unordered_set(InputIterator f, InputIterator l,
                  size_type n = implementation-defined,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
```

**Effects:** Constructs an empty `unordered_set` using the specified hash function, key equality function, and allocator, and using at least  $n$  buckets. (If  $n$  is not provided, the number of buckets is implementation defined.) Then inserts elements from the range `[first, last)`. `max_load_factor()` is 1.0.

**Complexity:** Average case linear, worst case quadratic.

#### 6.3.4.3.2 `unordered_set` swap

[tr.unord.set.swap]

```
template <class Value, class Hash, class Pred, class Alloc>
    void swap(unordered_set<Value, Hash, Pred, Alloc>& x,
              unordered_set<Value, Hash, Pred, Alloc>& y);
```

**Effects:** `x.swap(y)`.

#### 6.3.4.4 Class template `unordered_map`

[tr.unord.map]

An `unordered_map` is a kind of unordered associative container that supports unique keys (an `unordered_map` contains at most one of each key value) and that associates values of another type mapped\_type with the keys.

An `unordered_map` satisfies all of the requirements of a container and of a unordered associative container. It provides the operations described in the preceding requirements table for unique keys; that is, an `unordered_map` supports the `a_uniq` operations in that table, not the `a_eq` operations. For a `unordered_map<Key, T>` the key type is `Key`, the mapped type is `T`, and the value type is `std::pair<const Key, T>`.

This section only describes operations on `unordered_map` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
template <class Key,
          class T,
          class Hash = hash<Key>,
          class Pred = std::equal_to<Key>,
          class Alloc = std::allocator<std::pair<const Key, T> > >
    class unordered_map
    {
```

```

public:
    // types
    typedef Key                key_type;
    typedef std::pair<const Key, T> value_type;
    typedef T                  mapped_type;
    typedef Hash               hasher;
    typedef Pred               key_equal;
    typedef Alloc              allocator_type;
    typedef typename allocator_type::pointer pointer;
    typedef typename allocator_type::const_pointer const_pointer;
    typedef typename allocator_type::reference reference;
    typedef typename allocator_type::const_reference const_reference;
    typedef implementation-defined size_type;
    typedef implementation-defined difference_type;

    typedef implementation-defined iterator;
    typedef implementation-defined const_iterator;
    typedef implementation-defined local_iterator;
    typedef implementation-defined const_local_iterator;

    // construct/destroy/copy
    explicit unordered_map(size_type n = implementation-defined,
                          const hasher& hf = hasher(),
                          const key_equal& eql = key_equal(),
                          const allocator_type& a = allocator_type());
    template <class InputIterator>
        unordered_map(InputIterator f, InputIterator l,
                      size_type n = implementation-defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
    unordered_map(const unordered_map&);
    ~unordered_map();
    unordered_map& operator=(const unordered_map&);
    allocator_type get_allocator() const;

    // size and capacity
    bool empty() const;
    size_type size() const;
    size_type max_size() const;

    // iterators
    iterator      begin();
    const_iterator begin() const;
    iterator      end();
    const_iterator end() const;

```

```

// modifiers
std::pair<iterator, bool> insert(const value_type& obj);
iterator insert(const_iterator hint, const value_type& obj);
template <class InputIterator>
    void insert(InputIterator first, InputIterator last);

void erase(const_iterator position);
size_type erase(const key_type& k);
void erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_map&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator          find(const key_type& k);
const_iterator    find(const key_type& k) const;
size_type count(const key_type& k) const;
std::pair<iterator, iterator>
    equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>
    equal_range(const key_type& k) const;

mapped_type& operator[](const key_type& k);

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n);
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n) const;
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float z);
void rehash(size_type n);
};

```

```

template <class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_map<Key, T, Hash, Pred, Alloc>& x,
          unordered_map<Key, T, Hash, Pred, Alloc>& y);

```

#### 6.3.4.4.1 unordered\_map constructors

[tr.unord.map.cnstr]

```

explicit unordered_map(size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());

```

**Effects:** Constructs an empty `unordered_map` using the specified hash function, key equality function, and allocator, and using at least  $n$  buckets. If  $n$  is not provided, the number of buckets is implementation defined. `max_load_factor()` is 1.0.

**Complexity:** Constant.

```

template <class InputIterator>
unordered_map(InputIterator f, InputIterator l,
              size_type n = implementation-defined,
              const hasher& hf = hasher(),
              const key_equal& eql = key_equal(),
              const allocator_type& a = allocator_type());

```

**Effects:** Constructs an empty `unordered_map` using the specified hash function, key equality function, and allocator, and using at least  $n$  buckets. (If  $n$  is not provided, the number of buckets is implementation defined.) Then inserts elements from the range  $[first, last)$ . `max_load_factor()` is 1.0.

**Complexity:** Average case linear, worst case quadratic.

#### 6.3.4.4.2 unordered\_map element access

[tr.unord.map.elem]

```

mapped_type& operator[](const key_type& k);

```

**Effects:** If the `unordered_map` does not already contain an element whose key is equivalent to  $k$ , inserts `std::pair<const key_type, mapped_type>(k, mapped_type())`.

**Returns:** A reference to `x.second`, where  $x$  is the (unique) element whose key is equivalent to  $k$ .

#### 6.3.4.4.3 unordered\_map swap

[tr.unord.map.swap]

```

template <class Value, class Hash, class Pred, class Alloc>
void swap(unordered_map<Value, Hash, Pred, Alloc>& x,
          unordered_map<Value, Hash, Pred, Alloc>& y);

```

**Effects:** `x.swap(y)`.

### 6.3.4.5 Class template `unordered_multiset`

[`tr.unord.multiset`]

An `unordered_multiset` is a kind of unordered associative container that supports equivalent keys (an `unordered_multiset` may contain multiple copies of the same key value) and in which the elements' keys are the elements themselves.

An `unordered_multiset` satisfies all of the requirements of a container and of a unordered associative container. It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multiset` supports the `a_eq` operations in that table, not the `a_uniq` operations. For a `unordered_multiset<Value>` the `key_type` and the `value_type` are both `Value`. The `iterator` and `const_iterator` types are both `const` iterator types. It is unspecified whether or not they are the same type.

This section only describes operations on `unordered_multiset` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
template <class Value,
          class Hash = hash<Value>,
          class Pred = std::equal_to<Value>,
          class Alloc = std::allocator<Value> >
class unordered_multiset
{
public:
    // types
    typedef Value          key_type;
    typedef Value          value_type;
    typedef Hash           hasher;
    typedef Pred           key_equal;
    typedef Alloc          allocator_type;
    typedef typename allocator_type::pointer      pointer;
    typedef typename allocator_type::const_pointer const_pointer;
    typedef typename allocator_type::reference    reference;
    typedef typename allocator_type::const_reference const_reference;
    typedef implementation-defined              size_type;
    typedef implementation-defined              difference_type;

    typedef implementation-defined              iterator;
    typedef implementation-defined              const_iterator;
    typedef implementation-defined              local_iterator;
    typedef implementation-defined              const_local_iterator;

    // construct/destroy/copy
    explicit unordered_multiset(size_type n = implementation-defined,
                               const hasher& hf = hasher(),
                               const key_equal& eql = key_equal(),
                               const allocator_type& a = allocator_type());
    template <class InputIterator>
```

```

    unordered_multiset(InputIterator f, InputIterator l,
                       size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
unordered_multiset(const unordered_multiset&);
~unordered_multiset();
unordered_multiset& operator=(const unordered_multiset&);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator      begin();
const_iterator begin() const;
iterator      end();
const_iterator end() const;

// modifiers
iterator insert(const value_type& obj);
iterator insert(const_iterator hint, const value_type& obj);
template <class InputIterator>
    void insert(InputIterator first, InputIterator last);

void erase(const_iterator position);
size_type erase(const key_type& k);
void erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_multiset&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
std::pair<iterator, iterator>
    equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>
    equal_range(const key_type& k) const;

```



```

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n);
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n) const;
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float z);
void rehash(size_type n);
};

template <class Value, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
         unordered_multiset<Value, Hash, Pred, Alloc>& y);
}

```

#### 6.3.4.5.1 unordered\_multiset constructors

[tr.unord.multiset.cnstr]

```

explicit unordered_multiset(size_type n = implementation-defined,
                          const hasher& hf = hasher(),
                          const key_equal& eql = key_equal(),
                          const allocator_type& a = allocator_type());

```

**Effects:** Constructs an empty `unordered_multiset` using the specified hash function, key equality function, and allocator, and using at least  $n$  buckets. If  $n$  is not provided, the number of buckets is implementation defined. `max_load_factor()` is 1.0.

**Complexity:** Constant.

```

template <class InputIterator>
unordered_multiset(InputIterator f, InputIterator l,
                  size_type n = implementation-defined,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());

```

**Effects:** Constructs an empty `unordered_multiset` using the specified hash function, key equality function, and allocator, and using at least  $n$  buckets. (If  $n$  is not provided, the number of buckets is implementation defined.) Then inserts elements from the range  $[first, last)$ . `max_load_factor()` is 1.0.

**Complexity:** Average case linear, worst case quadratic.

#### 6.3.4.5.2 unordered\_multiset swap

[tr.unord.multiset.swap]

```
template <class Value, class Hash, class Pred, class Alloc>
void swap(unordered_multiset<Value, Hash, Pred, Alloc>& x,
          unordered_multiset<Value, Hash, Pred, Alloc>& y);
```

**Effects:** `x.swap(y);`

#### 6.3.4.6 Class template unordered\_multimap

[tr.unord.multimap]

An `unordered_multimap` is a kind of unordered associative container that supports equivalent keys (an `unordered_multimap` may contain multiple copies of each key value) and that associates values of another type `mapped_type` with the keys.

An `unordered_multimap` satisfies all of the requirements of a container and of a unordered associative container. It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multimap` supports the `a_eq` operations in that table, not the `a_uniq` operations. For an `unordered_multimap<Key, T>` the key type is `Key`, the mapped type is `T`, and the value type is `std::pair<const Key, T>`.

This section only describes operations on `unordered_multimap` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
template <class Key,
          class T,
          class Hash = hash<Key>,
          class Pred = std::equal_to<Key>,
          class Alloc = std::allocator<std::pair<const Key, T> > >
class unordered_multimap
{
public:
    // types
    typedef Key                key_type;
    typedef std::pair<const Key, T> value_type;
    typedef T                  mapped_type;
    typedef Hash               hasher;
    typedef Pred               key_equal;
    typedef Alloc              allocator_type;
    typedef typename allocator_type::pointer        pointer;
    typedef typename allocator_type::const_pointer  const_pointer;
    typedef typename allocator_type::reference      reference;
    typedef typename allocator_type::const_reference const_reference;
    typedef implementation-defined                size_type;
    typedef implementation-defined                difference_type;
```

```

typedef implementation-defined iterator;
typedef implementation-defined const_iterator;
typedef implementation-defined local_iterator;
typedef implementation-defined const_local_iterator;

// construct/destroy/copy
explicit unordered_multimap(size_type n = implementation-defined,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a = allocator_type());

template <class InputIterator>
    unordered_multimap(InputIterator f, InputIterator l,
                      size_type n = implementation-defined,
                      const hasher& hf = hasher(),
                      const key_equal& eql = key_equal(),
                      const allocator_type& a = allocator_type());
unordered_multimap(const unordered_multimap&);
~unordered_multimap();
unordered_multimap& operator=(const unordered_multimap&);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator      begin();
const_iterator begin() const;
iterator      end();
const_iterator end() const;

// modifiers
iterator insert(const value_type& obj);
iterator insert(const_iterator hint, const value_type& obj);
template <class InputIterator>
    void insert(InputIterator first, InputIterator last);

void erase(const_iterator position);
size_type erase(const key_type& k);
void erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_multimap&);

// observers

```

```

hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator      find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
std::pair<iterator, iterator>
    equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>
    equal_range(const key_type& k) const;

mapped_type& operator[](const key_type& k);

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n);
size_type bucket(const key_type& k) const;
local_iterator begin(size_type n) const;
const_local_iterator begin(size_type n) const;
local_iterator end(size_type n);
const_local_iterator end(size_type n) const;

// hash policy
float load_factor() const;
float max_load_factor() const;
void max_load_factor(float z);
void rehash(size_type n);
};

template <class Key, class T, class Hash, class Pred, class Alloc>
void swap(unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
          unordered_multimap<Key, T, Hash, Pred, Alloc>& y);

```

#### 6.3.4.6.1 unordered\_multimap constructors

[tr.unord.multimap.cnstr]

```

explicit unordered_multimap(size_type n = implementation-defined,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a = allocator_type());

```

**Effects:** Constructs an empty `unordered_multimap` using the specified hash function, key equality function, and allocator, and using at least `n` buckets. If `n` is not provided, the number of buckets is implementation defined. `max_load_factor()` is 1.0.

**Complexity:** Constant.

```
template <class InputIterator>
    unordered_multimap(InputIterator f, InputIterator l,
                       size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
```

**Effects:** Constructs an empty `unordered_multimap` using the specified hash function, key equality function, and allocator, and using at least  $n$  buckets. (If  $n$  is not provided, the number of buckets is implementation defined.) Then inserts elements from the range  $[first, last)$ . `max_load_factor()` is 1.0.

**Complexity:** Average case linear, worst case quadratic.

#### 6.3.4.6.2 `unordered_multimap` swap

[tr.unord.multimap.swap]

```
template <class Value, class Hash, class Pred, class Alloc>
    void swap(unordered_multimap<Value, Hash, Pred, Alloc>& x,
              unordered_multimap<Value, Hash, Pred, Alloc>& y);
```

**Effects:** `x.swap(y)`.

## Chapter 7

# Regular expressions

[tr.re]

### 7.1 Definitions

[tr.re.def]

The following definitions shall apply to this clause:

**Collating element:** A sequence of one or more characters within the current locale that collate as if they were a single character.

**Finite state machine:** An unspecified data structure that is used to represent a regular expression, and which permits efficient matches against the regular expression to be obtained.

**Format specifier:** A sequence of one or more characters that is to be replaced with some part of a regular expression match.

**Matched:** A sequence of zero or more characters shall be said to be matched by a regular expression when the characters in the sequence correspond to a sequence of characters defined by the pattern.

**Partial match:** A match that is obtained by matching one or more characters at the end of a character-container sequence, but only a prefix of the regular expression.

**Primary equivalence class:** A set of one or more characters which share the same primary sort key: that is the sort key weighting that depends only upon character shape, and not accentation, case, or locale specific tailorings.

**Regular expression:** A pattern that selects specific strings from a set of character strings.

**Sub-expression:** A subset of a regular expression that has been marked by parenthesis.

### 7.2 Requirements

[tr.re.req]

This subclause defines requirements on classes representing regular expression traits. [*Note:* The class template `regex_traits<charT>`, defined in clause 7.7, satisfies these requirements. —*end note*]

The class template `basic_regex`, defined in clause 7.8, needs a set of related types and functions to complete the definition of its semantics. These types and functions are provided as a set of member type-

defs and functions in the template parameter `traits` used by the `basic_regex` class template. This subclause defines the semantics guaranteed by these members.

To specialize the `basic_regex` template to generate a regular expression class to handle a particular character container type `CharT`, that and its related regular expression traits class `Traits` is passed as a pair of parameters to the `basic_regex` template as formal parameters `charT` and `Traits`.

In Table 7.1 `X` denotes a traits class defining types and functions for the character container type `charT`; `u` is an object of type `X`; `v` is an object of type `const X`; `p` is a value of type `const charT*`; `I1` and `I2` are Input Iterators; `c` is a value of type `const charT`; `s` is an object of type `X::string_type`; `cs` is an object of type `const X::string_type`; `b` is a value of type `bool`; `I` is a value of type `int`; and `loc` is an object of type `X::locale_type`.

Table 7.1: regular expression traits class requirements

Expression	Return Type	Assertion / Note / Pre / Post condition
<code>X::char_type</code>	<code>charT</code>	The character container type used in the implementation of class template <code>basic_regex</code> .
<code>X::size_type</code>		An unsigned integer type, capable of holding the length of a null-terminated string of <code>charTs</code> .
<code>X::string_type</code>	<code>std::basic_string&lt;charT&gt;</code>	
<code>X::locale_type</code>	Implementation defined	A copy constructible type that represents the locale used by the traits class.
<code>X::char_class_type</code>	Implementation defined	A bitmask type representing a particular character classification. Multiple values of this type can be bitwise-or'ed together to obtain a new valid value.
<code>X::version_tag</code>	Either <code>regex_traits_version_1_tag</code> or a class that publicly inherits from <code>regex_traits_version_1_tag</code> .	A type that signifies the interface to which this traits class conforms.
<code>X::sentry</code>	Implementation defined	A type used to force initialization of an instance of this traits class. An instance of this type must be copy constructed from a traits class instance, and verified that it does not compare equal to a null pointer constant, before any member function of that traits class instance may be called other than <code>length</code> , <code>imbue</code> , and <code>getloc</code> .

<i>continued from previous page</i>		
<code>X::sentry s(u);</code>	<code>void</code>	An object of type <code>X::sentry</code> shall be copy constructible from an instance of <code>X</code> .
<code>X::sentry(u)</code>	Convertible to <code>bool</code> .	Conversion of the <code>sentry</code> class to <code>bool</code> , is used to verify that the traits class has been correctly initialized.
<code>X::length(p)</code>	<code>X::size_type</code>	Yields the smallest <code>i</code> such that <code>p[i] == 0</code> . Complexity is linear in <code>i</code> .
<code>v.syntax_type(c)</code>	<code>regex_constants::syntax_type</code>	Returns a symbolic value of type <code>regex_constants::syntax_type</code> that signifies the meaning of character <code>c</code> within the regular expression grammar.
<code>v.escape_syntax_type(c)</code>	<code>regex_constants::escape_syntax_type</code>	Returns a symbolic value of type <code>regex_constants::escape_syntax_type</code> , that signifies the meaning of character <code>c</code> within the regular expression grammar, when <code>c</code> has been preceded by an escape character. Precondition: if <code>b</code> is the character preceding <code>c</code> in the expression being parsed then: <code>v.syntax_type(b) == syntax_escape</code> .
<code>v.translate(c, b)</code>	<code>X::char_type</code>	Returns a character <code>d</code> such that: for any character <code>d</code> that is to be considered equivalent to <code>c</code> then <code>v.translate(c, false) == v.translate(d, false)</code> . Likewise for all characters <code>C</code> that are to be considered equivalent to <code>c</code> when comparisons are to be performed without regard to case, then <code>v.translate(c, true) == v.translate(C, true)</code> .
<code>v.transform(cs)</code>	<code>X::string_type</code>	Returns a sort key for string <code>cs</code> such that: if string <code>cs</code> sorts before string <code>ct</code> then: <code>v.transform(cs) &lt; v.transform(ct)</code> .
<code>v.transform_primary(cs)</code>	<code>X::string_type</code>	Returns a primary sort key for string <code>cs</code> , such that if <code>cs</code> sorts before string <code>ct</code> , when character case is not considered, then <code>v.transform(cs) &lt; v.transform(ct)</code> . Returns an empty string on error.



<i>continued from previous page</i>		
<code>v.lookup_classname(cs)</code>	<code>X::char_class_type</code>	Converts the string <code>cs</code> into a bit-mask type that can subsequently be passed to <code>is_class</code> . Values returned from <code>lookup_classname</code> can be safely bitwise or'ed together. Returns an empty string if <code>cs</code> is not the name of a character class recognized by <code>X</code> . At least the names "d", "w", "s", "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper" and "xdigit", shall be recognized. The value returned shall be independent of the case of the characters in <code>cs</code> .
<code>v.lookup_collatename(cs)</code>	<code>X::string_type</code>	Returns a string containing the collating element named by <code>cs</code> , or an empty string if <code>cs</code> is not a recognized name.
<code>v.is_class(c, v.lookup_classname(cs))</code>	<code>bool</code>	Returns <code>true</code> if character <code>c</code> is a member of the character class <code>cs</code> , <code>false</code> otherwise.
<code>v.toi(I1, I2, i)</code>	<code>int</code>	Behaves as follows: if <code>p==q</code> or <code>v.is_class(*p, v.lookup_classname("d")) == false</code> then returns <code>-1</code> . Otherwise performs formatted numeric input on the sequence <code>[p,q)</code> and returns the result as an <code>int</code> . Postcondition: either <code>p == q</code> or <code>v.is_class(*p, v.lookup_classname("d")) == false</code> .
<code>u.imbue(loc)</code>	<code>X::locale_type</code>	imbues <code>u</code> with the locale <code>loc</code> , returns the previous locale used by <code>u</code> if any.
<code>v.getloc()</code>	<code>X::locale_type</code>	Returns the current locale used by <code>v</code> , if any.
<code>v.error_string(i)</code>	<code>std::string</code>	Returns a human readable error string for the error condition <code>i</code> , where <code>i</code> is one of the values enumerated by type <code>regex_constants::error_type</code> . If the value <code>i</code> is not recognized then returns the string "Unknown error" or a localized equivalent.

The header `<regex>` defines the class template `regex_traits` which shall be capable of being special-

ized for character container types `char` and `wchar_t`, and which satisfies the requirements for a regular expression traits class. Class template `regex_traits` is described in clause 7.7.

### 7.3 Regular expressions summary [tr.re.sym]

The header `<regex>` defines a basic regular expression class template and its traits that can handle all char-like (`lib.strings`) template arguments.

The header `<regex>` defines a class template that holds the result of a regular expression match.

The header `<regex>` defines a series of algorithms that allow an iterator sequence to be operated upon by a regular expression.

The header `<regex>` defines two specific template classes, `regex` and `wregex` and their special traits.

The header `<regex>` also defines two iterator types for enumerating regular expression matches.

### 7.4 Header `<regex>` synopsis [tr.re.syn]

```
namespace tr1 {

namespace regex_constants {
    typedef bitmask_type syntax_option_type;
    typedef bitmask_type match_flag_type;
    typedef implementation-defined syntax_type;
    typedef implementation-defined escape_syntax_type;
    typedef implementation-defined error_type;
} // namespace regex_constants

class bad_expression;

template <class charT>
struct regex_traits;

template <class charT,
         class traits = regex_traits<charT> >
class basic_regex;

template <class charT, class traits>
void swap(basic_regex<charT, traits>& e1,
         basic_regex<charT, traits>& e2);

typedef basic_regex<char> regex;
typedef basic_regex<wchar_t> wregex;

template <class BidirectionalIterator>
```

```

class sub_match;

template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);

template <class BidirectionalIterator, class traits, class Allocator>
bool operator == (const std::basic_string<
                 iterator_traits<BidirectionalIterator>::value_type,
                 traits,
                 Allocator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator != (const std::basic_string<
                 iterator_traits<BidirectionalIterator>::value_type,
                 traits,
                 Allocator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator < (const std::basic_string<
                 iterator_traits<BidirectionalIterator>::value_type,
                 traits,
                 Allocator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator > (const std::basic_string<
                 iterator_traits<BidirectionalIterator>::value_type,
                 traits,
                 Allocator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);

```

```

template <class BidirectionalIterator, class traits, class Allocator>
bool operator >= (const std::basic_string<
    iterator_traits<BidirectionalIterator>::value_type,
    traits,
    Allocator>& lhs,
    const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator <= (const std::basic_string<
    iterator_traits<BidirectionalIterator>::value_type,
    traits,
    Allocator>& lhs,
    const sub_match<BidirectionalIterator>& rhs);

template <class BidirectionalIterator, class traits, class Allocator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
    const std::basic_string<
    iterator_traits<BidirectionalIterator>::value_type,
    traits,
    Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
    const std::basic_string<
    iterator_traits<BidirectionalIterator>::value_type,
    traits,
    Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
    const std::basic_string<
    iterator_traits<BidirectionalIterator>::value_type,
    traits,
    Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
    const std::basic_string<
    iterator_traits<BidirectionalIterator>::value_type,
    traits,
    Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
    const std::basic_string<
    iterator_traits<BidirectionalIterator>::value_type,
    traits,
    Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
    const std::basic_string<

```

```

        iterator_traits<BidirectionalIterator>::value_type,
        traits,
        Allocator>& rhs);

template <class BidirectionalIterator>
bool operator == (typename iterator_traits<BidirectionalIterator>
                 ::value_type const* lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator != (typename iterator_traits<BidirectionalIterator>
                 ::value_type const* lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator < (typename iterator_traits<BidirectionalIterator>
                ::value_type const* lhs,
                const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator > (typename iterator_traits<BidirectionalIterator>
                ::value_type const* lhs,
                const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator >= (typename iterator_traits<BidirectionalIterator>
                 ::value_type const* lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator <= (typename iterator_traits<BidirectionalIterator>
                 ::value_type const* lhs,
                 const sub_match<BidirectionalIterator>& rhs);

template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                 ::value_type const* rhs);
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                 ::value_type const* rhs);
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                 ::value_type const* rhs);
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                 ::value_type const* rhs);
template <class BidirectionalIterator>

```

```

bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                 ::value_type const* rhs);
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                 ::value_type const* rhs);

template <class BidirectionalIterator>
bool operator == (typename iterator_traits<BidirectionalIterator>
                 ::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator != (typename iterator_traits<BidirectionalIterator>
                 ::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator < (typename iterator_traits<BidirectionalIterator>
                 ::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator > (typename iterator_traits<BidirectionalIterator>
                 ::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator >= (typename iterator_traits<BidirectionalIterator>
                 ::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator <= (typename iterator_traits<BidirectionalIterator>
                 ::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);

template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                 ::value_type const& rhs);
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                 ::value_type const& rhs);
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                 ::value_type const& rhs);
template <class BidirectionalIterator>

```

```

bool operator > (const sub_match<BidirectionalIterator>& lhs,
                typename iterator_traits<BidirectionalIterator>
                    ::value_type const& rhs);
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                     ::value_type const& rhs);
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                     ::value_type const& rhs);

template <class charT, class traits, class BidirectionalIterator>
basic_ostream<charT, traits>&
    operator << (basic_ostream<charT, traits>& os,
                const sub_match<BidirectionalIterator>& m);

template <class BidirectionalIterator,
          class Allocator = allocator<
              typename iterator_traits<BidirectionalIterator>::value_type > >
class match_results;

template <class BidirectionalIterator, class Allocator>
bool operator == (const match_results<BidirectionalIterator, Allocator>& m1,
                 const match_results<BidirectionalIterator, Allocator>& m2);
template <class BidirectionalIterator, class Allocator>
bool operator != (const match_results<BidirectionalIterator, Allocator>& m1,
                 const match_results<BidirectionalIterator, Allocator>& m2);

template <class charT, class traits,
          class BidirectionalIterator, class Allocator>
basic_ostream<charT, traits>&
    operator << (basic_ostream<charT, traits>& os,
                const match_results<BidirectionalIterator, Allocator>& m);

template <class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
          match_results<BidirectionalIterator, Allocator>& m2);

typedef match_results<const char*> cmatch;
typedef match_results<const wchar_t*> wcmatch;
typedef match_results<string::const_iterator> smatch;
typedef match_results<wstring::const_iterator> wsmatch;

template <class BidirectionalIterator, class Allocator,
          class charT, class traits>

```

```

bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 match_results<BidirectionalIterator, Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);
template <class BidirectionalIterator,
          class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);
template <class charT, class Allocator, class traits>
bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);
template <class ST, class SA, class Allocator,
          class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 match_results<typename basic_string<charT, ST, SA>
                               ::const_iterator,
                               Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);
template <class charT, class traits>
bool regex_match(const charT* str,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);
template <class ST, class SA, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);

template <class BidirectionalIterator, class Allocator,
          class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                  match_results<BidirectionalIterator, Allocator>& m,
                  const basic_regex<charT, traits>& e,
                  regex_constants::match_flag_type flags
                  = regex_constants::match_default);
template <class charT, class Allocator, class traits>
bool regex_search(const charT* str,
                  match_results<const charT*, Allocator>& m,

```



```

        const basic_regex<charT, traits>& e,
        regex_constants::match_flag_type flags
        = regex_constants::match_default);
template <class BidirectionalIterator,
        class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
        const basic_regex<charT, traits>& e,
        regex_constants::match_flag_type flags
        = regex_constants::match_default);
template <class charT, class traits>
bool regex_search(const charT* str
        const basic_regex<charT, traits>& e,
        regex_constants::match_flag_type flags
        = regex_constants::match_default);
template <class ST, class SA,
        class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
        const basic_regex<charT, traits>& e,
        regex_constants::match_flag_type flags
        = regex_constants::match_default);
template <class ST, class SA, class Allocator,
        class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
        match_results<typename basic_string<charT, ST, SA>
                ::const_iterator,
                Allocator>& m,
        const basic_regex<charT, traits>& e,
        regex_constants::match_flag_type flags
        = regex_constants::match_default);

template <class OutputIterator, class BidirectionalIterator,
        class traits, class charT>
OutputIterator regex_replace(OutputIterator out,
        BidirectionalIterator first,
        BidirectionalIterator last,
        const basic_regex<charT, traits>& e,
        const basic_string<charT>& fmt,
        regex_constants::match_flag_type flags
        = regex_constants::match_default);
template <class traits, class charT>
basic_string<charT> regex_replace(const basic_string<charT>& s,
        const basic_regex<charT, traits>& e,
        const basic_string<charT>& fmt,
        regex_constants::match_flag_type flags
        = regex_constants::match_default);

```

```

// regular expression iterators:
template <class BidirectionalIterator,
         class charT = iterator_traits<BidirectionalIterator>::value_type,
         class traits = regex_traits<charT> >
class regex_iterator;
template <class BidirectionalIterator,
         class charT = iterator_traits<BidirectionalIterator>::value_type,
         class traits = regex_traits<charT> >
class regex_token_iterator;

} // namespace tr1

```

## 7.5 Namespace `std::regex_constants` [tr.re.const]

The namespace `std::regex_constants` acts as a repository for the symbolic constants used by the regular expression library.

The namespace `std::regex_constants` defines five types: `syntax_option_type`, `regex_constants::match_flag_type`, `syntax_type`, `escape_syntax_type` and `error_type`, along with a series of constants of these types.

### 7.5.1 Bitmask Type `syntax_option_type` [tr.re.synopt]

```

namespace tr1 { namespace regex_constants {

typedef bitmask_type syntax_option_type;
// these flags are required:
static const syntax_option_type normal;
static const syntax_option_type icode;
static const syntax_option_type nosubs;
static const syntax_option_type optimize;
static const syntax_option_type collate;
static const syntax_option_type ECMAScript = normal;
static const syntax_option_type JavaScript = normal;
static const syntax_option_type JScript = normal;
// these flags are optional, if the functionality is supported
// then the flags shall take these names.
static const syntax_option_type basic;
static const syntax_option_type extended;
static const syntax_option_type awk;
static const syntax_option_type grep;
static const syntax_option_type egrep;
static const syntax_option_type sed = basic;
static const syntax_option_type perl;

```

```

} // namespace regex_constants
} // namespace tr1

```

The type `syntax_option_type` is an implementation defined bitmask type (§17.3.2.1.2). Setting its elements has the effects listed in table 7.2, a valid value of type `syntax_option_type` will always have exactly one of the elements `normal`, `basic`, `extended`, `awk`, `grep`, `egrep`, `sed` or `perl` set.

Table 7.2: `syntax_option_type` effects

Element	Effect if set
<code>normal</code>	Specifies that the grammar recognized by the regular expression engine uses its normal semantics: that is the same as that given in the ECMA-262 [9].
<code>icase</code>	Specifies that matching of regular expressions against a character container sequence shall be performed without regard to case.
<code>nosubs</code>	Specifies that when a regular expression is matched against a character container sequence, then no sub-expression matches are to be stored in the supplied <code>match_results</code> structure.
<code>optimize</code>	Specifies that the regular expression engine should pay more attention to the speed with which regular expressions are matched, and less to the speed with which regular expression objects are constructed. Otherwise it has no detectable effect on the program output.
<code>collate</code>	Specifies that character ranges of the form “[a-b]” should be locale sensitive.
<code>ECMAScript</code>	The same as <code>normal</code> .
<code>JavaScript</code>	The same as <code>normal</code> .
<code>JScript</code>	The same as <code>normal</code> .
<code>basic</code>	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX basic regular expressions [13] in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX ), Base Definitions and Headers, Section 9, Regular Expressions .
<code>extended</code>	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX extended regular expressions [13] in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX ), Base Definitions and Headers, Section 9, Regular Expressions .
<code>awk</code>	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX utility <code>awk</code> in IEEE Std 1003.1-2001 [13].
<code>grep</code>	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX utility <code>grep</code> in IEEE Std 1003.1-2001 [13].

<i>continued from previous page</i>	
egrep	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX utility grep when given the -E option in IEEE Std 1003.1-2001 [13].
sed	The same as basic.
perl	Specifies that the grammar recognized by the regular expression is an implementation defined extension of the normal syntax.

### 7.5.2 Bitmask Type `regex_constants::match_flag_type` [tr.re.matchflag]

```
namespace tr1 { namespace regex_constants{

typedef bitmask_type regex_constants::match_flag_type;

static const regex_constants::match_flag_type match_default = 0;
static const regex_constants::match_flag_type match_not_bol;
static const regex_constants::match_flag_type match_not_eol;
static const regex_constants::match_flag_type match_not_bow;
static const regex_constants::match_flag_type match_not_eow;
static const regex_constants::match_flag_type match_any;
static const regex_constants::match_flag_type match_not_null;
static const regex_constants::match_flag_type match_continuous;
static const regex_constants::match_flag_type match_partial;
static const regex_constants::match_flag_type match_prev_avail;
static const regex_constants::match_flag_type format_default = 0;
static const regex_constants::match_flag_type format_sed;
static const regex_constants::match_flag_type format_perl;
static const regex_constants::match_flag_type format_no_copy;
static const regex_constants::match_flag_type format_first_only;

} // namespace regex_constants
} // namespace tr1
```

The type `regex_constants::match_flag_type` is an implementation defined bitmask type (§17.3.2.1.2). Matching a regular expression against a sequence of characters [`first`, `last`) proceeds according to the normal rules of ECMA-262 [9], modified according to the effects listed in table 7.3 for any bitmask elements set.

Table 7.3: `regex_constants::match_flag_type` effects when obtaining a match against a character container sequence [`first`,`last`).

Element	Effect if set
<code>match_default</code>	No effect.
<code>match_not_bol</code>	The expression " <code>^</code> " is not matched against the sub-sequence [ <code>first</code> , <code>first</code> ).

<i>continued from previous page</i>	
<code>match_not_eol</code>	The expression "\$" is not matched against the sub-sequence [last,last).
<code>match_not_bow</code>	The expression "\b" is not matched against the sub-sequence [first,first).
<code>match_not_eow</code>	The expression "\b" should not be matched against the sub-sequence [last,last).
<code>match_any</code>	If more than one match is possible then any match is an acceptable result.
<code>match_not_null</code>	The expression does not match an empty sequence.
<code>match_continuous</code>	The expression only matches a sub-sequence that begins at first .
<code>match_partial</code>	If no match is found, then it is acceptable to return a match [from, last) where from!=last, if there exists some sequence of characters [from,to) of which [from,last) is a prefix, and which would result in a full match.
<code>match_prev_avail</code>	–first is a valid iterator position. When this flag is set then the flags <code>match_not_bol</code> and <code>match_not_bow</code> are ignored by the regular expression algorithms 7.11 and iterators 7.12.
<code>format_default</code>	When a regular expression match is to be replaced by a new string, the new string is constructed using the rules used by the ECMAScript replace function in ECMA-262 [9], part 5.4.11 <code>String.prototype.replace</code> . In addition, during search and replace operations all non-overlapping occurrences of the regular expression are located and replaced, and sections of the input that did not match the expression, are copied unchanged to the output string.
<code>format_sed</code>	When a regular expression match is to be replaced by a new string, the new string is constructed using the rules used by the POSIX sed utility in IEEE Std 1003.1-2001 [13].
<code>format_perl</code>	When a regular expression match is to be replaced by a new string, the new string is constructed using an implementation defined superset of the rules used by the ECMAScript replace function in ECMA-262 [9], part 5.4.11 <code>String.prototype.replace</code> .
<code>format_no_copy</code>	During a search and replace operation, sections of the character container sequence being searched that do match the regular expression are not copied to the output string.
<code>format_first_only</code>	When specified during a search and replace operation, only the first occurrence of the regular expression is replaced.

### 7.5.3 Implementation defined `syntax_type`

[tr.re.syntype]

```
namespace tr1 { namespace regex_constants{
typedef implementation-defined syntax_type;
```

```

static const syntax_type syntax_char;
static const syntax_type syntax_open_mark;
static const syntax_type syntax_close_mark;
static const syntax_type syntax_dollar;
static const syntax_type syntax_caret;
static const syntax_type syntax_dot;
static const syntax_type syntax_star;
static const syntax_type syntax_plus;
static const syntax_type syntax_question;
static const syntax_type syntax_open_set;
static const syntax_type syntax_close_set;
static const syntax_type syntax_or;
static const syntax_type syntax_escape;
static const syntax_type syntax_dash;
static const syntax_type syntax_open_brace;
static const syntax_type syntax_close_brace;
static const syntax_type syntax_digit;
static const syntax_type syntax_comma;
static const syntax_type syntax_equal;
static const syntax_type syntax_colon;
static const syntax_type syntax_not;

} // namespace regex_constants
} // namespace tr1

```

The type `syntax_type` is an implementation defined enumeration type (§17.3.2.1.2). Values of type `syntax_type` represent how individual characters should be interpreted within a localized regular expression grammar, table 7.4 shows which special characters defined in ECMA-262 [9], are equivalent to which `syntax_type` values in the C locale:

Table 7.4: `syntax_type` values in the C locale

Value	Equivalent character(s) in the syntax specified by ECMA-262 [9].
<code>syntax_char</code>	Any character not listed below.
<code>syntax_open_mark</code>	(
<code>syntax_close_mark</code>	)
<code>syntax_dollar</code>	\$
<code>syntax_caret</code>	^
<code>syntax_dot</code>	.
<code>syntax_star</code>	*
<code>syntax_plus</code>	+
<code>syntax_question</code>	?
<code>syntax_open_set</code>	[
<code>syntax_close_set</code>	]

<i>continued from previous page</i>	
<code>syntax_or</code>	
<code>syntax_escape</code>	\
<code>syntax_dash</code>	-
<code>syntax_open_brace</code>	{
<code>syntax_close_brace</code>	}
<code>syntax_digit</code>	0123456789
<code>syntax_comma</code>	,
<code>syntax_equal</code>	=
<code>syntax_colon</code>	:
<code>syntax_not</code>	!

#### 7.5.4 Implementation defined `escape_syntax_type`

[tr.re.escsyn]

```

namespace tr1 { namespace regex_constants {

typedef implementation-defined escape_syntax_type;

static const escape_syntax_type escape_type_word_assert;
static const escape_syntax_type escape_type_not_word_assert;
static const escape_syntax_type escape_type_f;
static const escape_syntax_type escape_type_n;
static const escape_syntax_type escape_type_r;
static const escape_syntax_type escape_type_t;
static const escape_syntax_type escape_type_v;
static const escape_syntax_type escape_type_control;
static const escape_syntax_type escape_type_hex;
static const escape_syntax_type escape_type_unicode;
static const escape_syntax_type escape_type_identity;
static const escape_syntax_type escape_type_backref;
static const escape_syntax_type escape_type_decimal;
static const escape_syntax_type escape_type_class;
static const escape_syntax_type escape_type_not_class;

} // namespace regex_constants
} // namespace tr1

```

The type `escape_syntax_type` is an implementation defined enumeration type (§17.3.2.1.2). Values of type `escape_syntax_type` represent how individual escaped characters should be interpreted within a localized regular expression grammar, table 7.5 shows which special characters defined in ECMA-262 [9], are equivalent to which `escape_syntax_type` values in the C locale:

Table 7.5: `escape_syntax_type` values in the C locale

Value	Equivalent character(s) in syntax specified in ECMA-262 [9].
<code>escape_type_word_assert</code>	b
<code>escape_type_not_word_assert</code>	B
<code>escape_type_f</code>	f
<code>escape_type_n</code>	n
<code>escape_type_r</code>	r
<code>escape_type_t</code>	t
<code>escape_type_v</code>	v
<code>escape_type_control</code>	c
<code>escape_type_hex</code>	x
<code>escape_type_unicode</code>	u
<code>escape_type_identity</code>	<code>\   ^ \$ * + ? { } . ( ) [ ]</code>
<code>escape_type_backref</code>	123456789
<code>escape_type_decimal</code>	0
<code>escape_type_not_class</code>	Any upper case character not listed above.
<code>escape_type_class</code>	Any non-upper case character not listed above.

### 7.5.5 Implementation defined `error_type`

[tr.re.err]

```
namespace tr1 { namespace regex_constants {

typedef implementation defined error_type;

static const error_type error_collate;
static const error_type error_ctype;
static const error_type error_escape;
static const error_type error_subreg;
static const error_type error_brack;
static const error_type error_paren;
static const error_type error_brace;
static const error_type error_badbrace;
static const error_type error_range;
static const error_type error_space;
static const error_type error_badrepeat;
static const error_type error_complexity;
static const error_type error_stack;

} // namespace regex_constants
} // namespace tr1
```

The type `error_type` is an implementation defined enumeration type (§17.3.2.1.2). Values of type `error_type` represent the error conditions as described in table 7.6:



Table 7.6: `error_type` values in the C locale

Value	Error condition
<code>error_collate</code>	The expression contained an invalid collating element name.
<code>error_ctype</code>	The expression contained an invalid character class name.
<code>error_escape</code>	The expression contained an invalid escaped character, or a trailing escape.
<code>error_subreg</code>	The expression contained an invalid backreference.
<code>error_brack</code>	The expression contained mismatched [ and ].
<code>error_paren</code>	The expression contained mismatched ( and ).
<code>error_brace</code>	The expression contained mismatched { and }.
<code>error_badbrace</code>	The expression contained an invalid range in a { } expression.
<code>error_range</code>	The expression contained an invalid character range, for example [b-a].
<code>error_space</code>	There was insufficient memory to convert the expression into a finite state machine.
<code>error_badrepeat</code>	One of *?+{ was not preceded by a valid regular expression.
<code>error_complexity</code>	The complexity of an attempted match against a regular expression exceeded a pre-set level.
<code>error_stack</code>	There was insufficient memory to determine whether the regular expression could match the specified character sequence.

## 7.6 Class `bad_expression`

[tr.re.badexp]

```
class bad_expression : public std::runtime_error
{
public:
    explicit bad_expression(const std::string& what_arg);
};
```

The class `bad_expression` defines the type of objects thrown as exceptions to report errors during the conversion from a string representing a regular expression to a finite state machine.

```
bad_expression(const string& what_arg );
```

**Effects:** Constructs an object of class `bad_expression`.

**Postcondition:** `strcmp(what(), what_arg.c_str()) == 0`.

## 7.7 Class template `regex_traits`

[tr.re.traits]

```
template <class charT>
struct regex_traits
{
```

```

public:
    typedef charT                char_type;
    typedef std::size_t         size_type;
    typedef std::basic_string<char_type> string_type;
    typedef std::locale         locale_type;
    typedef bitmask_type       char_class_type;

    struct sentry
    {
        sentry(regex_traits<charT>&);
        operator void*();
    };

    regex_traits();
    static size_type length(const char_type* p);
    regex_constants::syntax_type syntax_type(charT c) const;
    regex_constants::escape_syntax_type
        escape_syntax_type(charT c) const;
    charT translate(charT c, bool icase) const;
    string_type transform(const string_type& in) const;
    string_type transform_primary(const string_type& in) const;
    char_class_type lookup_classname(const string_type& name) const;
    string_type lookup_collatename(const string_type& name) const;
    bool is_class(charT c, char_class_type f) const;
    template<class InputIterator>
    int toi(InputIterator& first,
            InputIterator last, int radix) const;
    locale_type imbue(locale_type l);
    locale_type getloc() const;
    std::string error_string(regex_constants::error_type) const;
};

```

The class template `regex_traits` is capable of being specialized for the types `char` and `wchar_t` and satisfies the requirements for a regular expression traits class (7.2).

```

typedef bitmask_type         char_class_type;

```

The type `char_class_type` is used to represent a character classification and is capable of holding an implementation specific set returned by `lookup_classname`.

```

struct sentry
{
    sentry(regex_traits<charT>&);
    operator void*();
};

```

An object of type `regex_traits<charT>::sentry` shall be constructed from a `regex_traits` object, and tested to be not equal to null, before any of the member functions of that object other than

length, getloc, and imbue shall be called. Type sentry performs implementation specific initialization of the traits class object, and represents an opportunity for the traits class to cache data obtained from the locale object.

```
static size_type length(const char_type* p);
```

**Effects:** Returns `char_traits<charT>::length(p)`;

```
regex_constants::syntax_type syntax_type(charT c) const;
```

**Effects:** For a character `c` in the right hand column of table 7.4, returns the corresponding `regex_constants::syntax_type` value.

```
regex_constants::escape_syntax_type  
escape_syntax_type(charT c) const;
```

**Effects:** For a character `c` in the right hand column of table 7.5, returns the corresponding `regex_constants::escape_syntax_type` value.

```
charT translate(charT c, bool icalse) const;
```

**Returns:** (`icalse ?`

```
use_facet<ctype<charT> >(getloc()).tolower(c) : c)
```

```
string_type transform(const string_type& in) const;
```

**Returns:** `use_facet<collate<charT> >(getloc()).transform(in.begin(), in.end())`

```
string_type transform_primary(const string_type& in) const;
```

**Effects:** if `typeid(use_facet<collate<charT> >) == typeid(collate_byname<charT>)` and the form of the sort key returned by `collate_byname<charT>::transform` is known and can be converted into a primary sort key, then returns that key, otherwise returns a empty string.

```
char_class_type lookup_classname(const string_type& name) const;
```

**Effects:** returns an unspecified value that represents the character classification name. If `name` is not recognized then returns a value that compares equal to 0. At least the names "d", "w", "s", "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper" and "xdigit", shall be recognized. The value returned shall be independent of the case of the characters in `name`.

```
string_type lookup_collatename(const string_type& name) const;
```

**Effects:** returns the sequence of one or more characters that represents the collating element name. Returns an empty string if `name` is not recognized. At least the names specified in IEEE Std 1003.1-2001 [13], Base Definitions and Headers, Section 6.1, Portable Character Set shall be recognized.

```
bool is_class(charT c, char_class_type f) const;
```

**Effects:** Determines if the character `c` is a member of the character classification represented by `f`.

**Returns:** Converts `f` into a value `m` of type `std::ctype_base::mask` in an unspecified manner, and returns `true` if `use_facet<ctype<charT>>(getloc()).is(c, m)` is `true`. Otherwise returns `true` if `f & lookup_classname("w") == lookup_classname("w")` and `c == '_'`, otherwise returns `false`.

```
template <class InputIterator>
int toi(InputIterator& first, InputIterator last, int radix) const;
```

**Precondition:** Argument `radix` shall take one of the values 8, 10 or 16.

**Effects:** If `first == last` or if `is_class(*first, lookup_classname("d")) == false` returns 0. Otherwise constructs an object `result` of type `int` and a `basic_istream<charT>` object which uses an unspecified stream buffer type to represent the character sequence `[first, last)`, and sets the format flags on that stream object as appropriate for argument `radix`. Performs unsigned integer formatted input on the `basic_istream<charT>` object setting the value `result` to the value obtained, and updates `first` to point to the first non-digit character in the sequence `[first, last)`. Returns `result`.

**Postcondition:** `is_class(*first, lookup_classname("d")) == false`.

```
locale_type imbue(locale_type loc);
```

**Effects:** Imbues `this` with a copy of the locale `loc`. The effect of calling `imbue` is to invalidate all cached data held by `this`. No member functions other than `length`, `getloc`, and `imbue` may be called until an object of type `sentry` has been copy-constructed from `*this`.

**Returns:** if no locale has been previously imbued then a copy of the global locale in effect at the time of construction of `this`, otherwise a copy of the last argument passed to `imbue`.

**Postcondition:** `getloc() == loc`.

```
locale_type getloc() const;
```

**Returns:** if no locale has been imbued then a copy of the global locale in effect at the time of construction of `this`, otherwise a copy of the last argument passed to `imbue`.

```
std::string error_string(regex_constants::error_type e) const;
```

**Returns:** A human readable error string for error condition `e`.

## 7.8 Class template `basic_regex` [tr.re.regex]

For a char-like type `charT`, the template class `basic_regex` describes objects that represent a regular expression constructed from a sequence of `charT`s. In the rest of this clause, `charT` denotes such a given char-like type. Storage for the regular expression is allocated and freed as necessary by the member functions of class `basic_regex`.

Objects of type specialization of `basic_regex` are responsible for converting the sequence of `charT` objects to an internal representation. It is not specified what form this representation takes, nor how it is accessed by algorithms that operate on regular expressions. [*Note:* implementations will typically declare some function template as friends of `basic_regex` to achieve this —*end note*]

The regular expression grammar recognized by class template `basic_regex` is described in clause 7.13. The functions described in this clause report errors by throwing exceptions of type `bad_expression`.

```
template <class charT,
          class traits = regex_traits<charT> >
class basic_regex
{
public:
    // types:
    typedef          charT                               value_type;
    typedef          regex_constants::syntax_option_type flag_type;
    typedef typename traits::locale_type                 locale_type;

    // constants:
    static const regex_constants::syntax_option_type normal
        = regex_constants::normal;
    static const regex_constants::syntax_option_type  icase
        = regex_constants::icase;
    static const regex_constants::syntax_option_type nosubs
        = regex_constants::nosubs;
    static const regex_constants::syntax_option_type optimize
        = regex_constants::optimize;
    static const regex_constants::syntax_option_type collate
        = regex_constants::collate;
    static const regex_constants::syntax_option_type ECMAScript
        = normal;
    static const regex_constants::syntax_option_type JavaScript
        = normal;
    static const regex_constants::syntax_option_type JScript
        = normal;
    // these flags are optional, if the functionality is supported
    // then the flags shall take these names.
    static const regex_constants::syntax_option_type basic
        = regex_constants::basic;
    static const regex_constants::syntax_option_type extended
        = regex_constants::extended;
    static const regex_constants::syntax_option_type awk
        = regex_constants::awk;
    static const regex_constants::syntax_option_type grep
        = regex_constants::grep;
    static const regex_constants::syntax_option_type egrep
        = regex_constants::egrep;
    static const regex_constants::syntax_option_type sed
        = basic = regex_constants::sed;
    static const regex_constants::syntax_option_type perl
        = regex_constants::perl;
```

```

// construct/copy/destroy:
basic_regex();
explicit basic_regex(const charT* p,
    flag_type f = regex_constants::normal);
basic_regex(const charT* p, size_type len, flag_type f);
basic_regex(const basic_regex&);
template <class ST, class SA>
explicit basic_regex(const basic_string<charT, ST, SA>& p,
    flag_type f = regex_constants::normal);
template <class InputIterator>
basic_regex(InputIterator first, InputIterator last,
    flag_type f = regex_constants::normal);

~basic_regex();
basic_regex& operator=(const basic_regex&);
basic_regex& operator=(const charT* ptr);
template <class ST, class SA>
basic_regex& operator=(const basic_string<charT, ST, SA>& p);

// capacity:
bool empty() const;
unsigned mark_count() const;

//
// modifiers:
basic_regex& assign(const basic_regex& that);
basic_regex& assign(const charT* ptr,
    flag_type f = regex_constants::normal);
basic_regex& assign(const charT* p, size_type len, flag_type f);
template <class string_traits, class A>
basic_regex& assign(
    const basic_string<charT, string_traits, A>& s,
    flag_type f = regex_constants::normal);
template <class InputIterator>
basic_regex& assign(InputIterator first, InputIterator last,
    flag_type f = regex_constants::normal);

// const operations:
flag_type flags() const;
// locale:
locale_type imbue(locale_type loc);
locale_type getloc() const;
// swap
void swap(basic_regex&) throw();
};

```

### 7.8.1 basic\_regex constants

[tr.re.regex.const]

```
static const regex_constants::syntax_option_type normal
    = regex_constants::normal;
static const regex_constants::syntax_option_type icase
    = regex_constants::icase;
static const regex_constants::syntax_option_type nosubs
    = regex_constants::nosubs;
static const regex_constants::syntax_option_type optimize
    = regex_constants::optimize;
static const regex_constants::syntax_option_type collate
    = regex_constants::collate;
static const regex_constants::syntax_option_type ECMAScript
    = normal;
static const regex_constants::syntax_option_type JavaScript
    = normal;
static const regex_constants::syntax_option_type JScript
    = normal;
// these flags are optional, if the functionality is supported
// then the flags shall take these names.
static const regex_constants::syntax_option_type basic
    = regex_constants::basic;
static const regex_constants::syntax_option_type extended
    = regex_constants::extended;
static const regex_constants::syntax_option_type awk
    = regex_constants::awk;
static const regex_constants::syntax_option_type grep
    = regex_constants::grep;
static const regex_constants::syntax_option_type egrep
    = regex_constants::egrep;
static const regex_constants::syntax_option_type sed
    = basic = regex_constants::sed;
static const regex_constants::syntax_option_type perl
    = regex_constants::perl;
```

The static constant members are provided as synonyms for the constants declared in namespace `regex_constants`; for each constant of type `syntax_option_type` declared in namespace `regex_constants` then a constant with the same name, type and value shall be declared within the scope of `basic_regex`.

### 7.8.2 basic\_regex constructors

[tr.re.regex.construct]

```
basic_regex();
```

**Effects:** Constructs an object of class `basic_regex`. The postconditions of this function are indicated in Table 7.7

Table 7.7: `basic_regex()` effects

Element	Value
<code>empty()</code>	<code>true</code>
<code>size()</code>	<code>0</code>
<code>str()</code>	<code>basic_string&lt;charT&gt;()</code>

```
basic_regex(const charT* p,
            flag_type f = regex_constants::normal);
```

**Requires:**  $p$  shall not be a null pointer.

**Throws:** `bad_expression` if  $p$  is not a valid regular expression.

**Effects:** Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the null-terminated string  $p$ , and interpreted according to the flags specified in  $f$ . The postconditions of this function are indicated in Table 7.8.

Table 7.8: `basic_regex(const charT* p, flag_type f)` effects

Element	Value
<code>empty()</code>	<code>false</code>
<code>size()</code>	<code>char_traits&lt;charT&gt;::length(p)</code>
<code>str()</code>	<code>basic_string&lt;charT&gt;(p)</code>
<code>flags()</code>	<code>f</code>
<code>mark_count()</code>	The number of marked sub-expressions within the expression.

```
basic_regex(const charT* p, size_type len,
            flag_type f);
```

**Requires:**  $p$  shall not be a null pointer.

**Throws:** `bad_expression` if  $p$  is not a valid regular expression.

**Effects:** Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the sequence of characters  $[p, p+len)$ , and interpreted according the flags specified in  $f$ . The postconditions of this function are indicated in Table 7.9

Table 7.9: `basic_regex(const charT* p1, size_type len, flag_type f)` effects

Element	Value
<code>empty()</code>	<code>false</code>
<code>size()</code>	<code>len</code>
<code>str()</code>	<code>basic_string&lt;charT&gt;(p, len)</code>
<code>flags()</code>	<code>f</code>



<i>continued from previous page</i>	
<code>mark_count()</code>	The number of marked sub-expressions within the expression.

```
basic_regex(const basic_regex& e);
```

**Effects:** Constructs an object of class `basic_regex` as a copy of the object `e`. The postconditions of this function are indicated in Table 7.10

Table 7.10: `basic_regex(const basic_regex& e)` effects

Element	Value
<code>empty()</code>	<code>e.empty</code>
<code>size()</code>	<code>e.size()</code>
<code>str()</code>	<code>e.str()</code>
<code>flags()</code>	<code>e.flags()</code>
<code>mark_count()</code>	<code>e.mark_count()</code>

```
template <class ST, class SA>
basic_regex(const basic_string<charT, ST, SA>& s,
            flag_type f = regex_constants::normal);
```

**Throws:** `bad_expression` if `s` is not a valid regular expression.

**Effects:** Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the string `s`, and interpreted according to the flags specified in `f`. The postconditions of this function are indicated in Table 7.11

Table 7.11: `basic_regex(const basic_string&)` effects

Element	Value
<code>empty()</code>	<code>false</code>
<code>size()</code>	<code>s.size()</code>
<code>str()</code>	<code>s</code>
<code>flags()</code>	<code>f</code>
<code>mark_count()</code>	The number of marked sub-expressions within the expression.

```
template <class ForwardIterator>
basic_regex(ForwardIterator first, ForwardIterator last,
            flag_type f = regex_constants::normal);
```

**Throws:** `bad_expression` if the sequence `[first, last)` is not a valid regular expression.

**Effects:** Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the sequence of characters `[first, last)`, and inter-

puted according to the flags specified in *f*. The postconditions of this function are indicated in Table 7.12.

Table 7.12: `basic_regex(ForwardIterator first, ForwardIterator last, flag_type f, const Allocator&)` effects

Element	Value
<code>empty()</code>	<code>false</code>
<code>size()</code>	<code>std::distance(first, last)</code>
<code>str()</code>	<code>std::basic_string&lt;charT&gt;(first, last)</code>
<code>flags()</code>	<code>f</code>
<code>mark_count()</code>	The number of marked sub-expressions within the expression.

```
basic_regex& operator=(const basic_regex& e);
```

**Effects:** Returns the result of `assign(e.str(), e.flags())`.

```
basic_regex& operator=(const charT* ptr);
```

**Requires:** `ptr` shall not be a null pointer.

**Effects:** Returns the result of `assign(ptr)`.

```
template <class ST, class SA>
basic_regex& operator=(const basic_string<charT, ST, SA>& p);
```

**Effects:** Returns the result of `assign(p)`.

### 7.8.3 `basic_regex` capacity

[tr.re.regex.cap]

```
bool empty() const;
```

**Effects:** Returns true if the object does not contain a valid regular expression, otherwise false.

```
unsigned mark_count() const;
```

**Effects:** Returns the number of marked sub-expressions within the regular expression.

### 7.8.4 `basic_regex` assign

[tr.re.regex.assign]

```
basic_regex& assign(const basic_regex& that);
```

**Effects:** Returns `assign(that.str(), that.flags())`.

```
basic_regex& assign(const charT* ptr,
                  flag_type f = regex_constants::normal);
```

**Effects:** Returns `assign(string_type(ptr), f)`.

```
basic_regex& assign(const charT* ptr, size_t len,
                  flag_type f = regex_constants::normal);
```

**Effects:** Returns `assign(string_type(ptr, len), f)`.

```
template <class string_traits, class A>
basic_regex& assign(const basic_string<charT, string_traits, A>& s,
                  flag_type f = regex_constants::normal);
```

**Throws:** `bad_expression` if `s` is not a valid regular expression.

**Returns:** `*this`.

**Effects:** Assigns the regular expression contained in the string `s`, interpreted according the flags specified in `f`. The postconditions of this function are indicated in Table 7.13.

Table 7.13: `basic_regex& assign(const basic_string<charT, string_traits, A>& s, flag_type f)` effects

Element	Value
<code>empty()</code>	<code>false</code>
<code>size()</code>	<code>s.size()</code>
<code>str()</code>	<code>s</code>
<code>flags()</code>	<code>f</code>
<code>mark_count()</code>	The number of marked sub-expressions within the expression.

```
template <class InputIterator>
basic_regex& assign(InputIterator first, InputIterator last,
                  flag_type f = regex_constants::normal);
```

**Requires:** The type `InputIterator` corresponds to the Input Iterator requirements (§24.1.1).

**Effects:** Returns `assign(string_type(first, last), f)`.

### 7.8.5 `basic_regex` constant operations

[tr.re.regex.operations]

```
flag_type flags() const;
```

**Effects:** Returns a copy of the regular expression syntax flags that were passed to the object's constructor, or the last call to `assign`.

### 7.8.6 `basic_regex` locale

[tr.re.regex.locale]

```
locale_type imbue(locale_type l);
```

**Effects:** Returns the result of `traits_inst.imbue(l)` where `traits_inst` is a (default initialized) instance of the template parameter `traits` stored within the object. Calls to `imbue` invalidate any currently contained regular expression.

**Postcondition:** `empty() == true`.

```
locale_type getloc() const;
```

**Effects:** Returns the result of `traits_inst.getloc()` where `traits_inst` is a (default initialized) instance of the template parameter `traits` stored within the object.

### 7.8.7 `basic_regex` swap

[tr.re.regex.swap]

```
void swap(basic_regex& e) throw();
```

**Effects:** Swaps the contents of the two regular expressions.

**Postcondition:** `*this` contains the characters that were in `e`, `e` contains the regular expression that was in `*this`.

**Complexity:** constant time.

### 7.8.8 `basic_regex` non-member functions

[tr.re.regex.nonmemb]

#### 7.8.8.1 `basic_regex` non-member swap

[tr.re.regex.nmswap]

```
template <class charT, class traits>
void swap(basic_regex<charT, traits>& lhs,
         basic_regex<charT, traits>& rhs);
```

**Effects:** Calls `lhs.swap(rhs)`.

## 7.9 Class template `sub_match`

[tr.re.submatch]

Class template `sub_match` denotes the sequence of characters matched by a particular marked sub-expression.

```
template <class BidirectionalIterator>
class sub_match
    : public std::pair<BidirectionalIterator, BidirectionalIterator>
{
public:
    typedef typename iterator_traits<BidirectionalIterator>::value_type
        value_type;
    typedef typename iterator_traits<BidirectionalIterator>::difference_type
        difference_type;
```

```

typedef BidirectionalIterator
    iterator;

bool matched;

difference_type length()const;
operator basic_string<value_type>()const;
basic_string<value_type> str()const;

int compare(const sub_match& s)const;
int compare(const basic_string<value_type>& s)const;
int compare(const value_type* s)const;
};

```

### 7.9.1 sub\_match members

[tr.re.submatch.members]

```

difference_type length();

```

**Returns:** (matched ? distance(first, second) : 0).

```

operator basic_string<value_type>()const;

```

**Returns:** (matched ? basic\_string<value\_type>(first, second) : basic\_string<value\_type>()).

```

basic_string<value_type> str()const;

```

**Returns:** (matched ? basic\_string<value\_type>(first, second) : basic\_string<value\_type>()).

```

int compare(const sub_match& s)const;

```

**Returns:** str().compare(s.str()).

```

int compare(const basic_string<value_type>& s)const;

```

**Returns:** str().compare(s).

```

int compare(const value_type* s)const;

```

**Returns:** str().compare(s).

### 7.9.2 sub\_match non-member operators

[tr.re.submatch.op]

```

template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
    const sub_match<BidirectionalIterator>& rhs);

```

**Returns:** lhs.compare(rhs) == 0.

```

template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);

```

**Returns:** lhs.compare(rhs) != 0.

```

template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                const sub_match<BidirectionalIterator>& rhs);

```

**Returns:** lhs.compare(rhs) < 0.

```

template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);

```

**Returns:** lhs.compare(rhs) <= 0.

```

template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);

```

**Returns:** lhs.compare(rhs) >= 0.

```

template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);

```

**Returns:** lhs.compare(rhs) > 0.

```

template <class BidirectionalIterator>
bool operator == (
    typename iterator_traits<BidirectionalIterator>::value_type
    const* lhs,
    const sub_match<BidirectionalIterator>& rhs);

```

**Returns:** lhs == rhs.str().

```

template <class BidirectionalIterator>
bool operator != (
    typename iterator_traits<BidirectionalIterator>::value_type
    const* lhs,
    const sub_match<BidirectionalIterator>& rhs);

```

**Returns:** lhs != rhs.str().

```

template <class BidirectionalIterator>
bool operator < (
    typename iterator_traits<BidirectionalIterator>::value_type
    const* lhs,
    const sub_match<BidirectionalIterator>& rhs);

```

**Returns:** lhs < rhs.str().

```
template <class BidirectionalIterator>
bool operator < (
    typename iterator_traits<BidirectionalIterator>::value_type
    const* lhs,
    const sub_match<BidirectionalIterator>& rhs);
```

**Returns:** lhs > rhs.str().

```
template <class BidirectionalIterator>
bool operator >= (
    typename iterator_traits<BidirectionalIterator>::value_type
    const* lhs,
    const sub_match<BidirectionalIterator>& rhs);
```

**Returns:** lhs >= rhs.str().

```
template <class BidirectionalIterator>
bool operator <= (
    typename iterator_traits<BidirectionalIterator>::value_type
    const* lhs,
    const sub_match<BidirectionalIterator>& rhs);
```

**Returns:** lhs <= rhs.str().

```
template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
    typename iterator_traits<BidirectionalIterator>::value_type
    const* rhs);
```

**Returns:** lhs.str() == rhs.

```
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
    typename iterator_traits<BidirectionalIterator>::value_type
    const* rhs);
```

**Returns:** lhs.str() != rhs.

```
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
    typename iterator_traits<BidirectionalIterator>::value_type
    const* rhs);
```

**Returns:** lhs.str() < rhs.

```
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
    typename iterator_traits<BidirectionalIterator>::value_type
    const* rhs);
```

**Returns:** lhs.str() > rhs.

```
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type
                 const* rhs);
```

**Returns:** lhs.str() >= rhs.

```
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type
                 const* rhs);
```

**Returns:** lhs.str() <= rhs.

```
template <class BidirectionalIterator>
bool operator == (
    typename iterator_traits<BidirectionalIterator>::value_type
    const& lhs,
    const sub_match<BidirectionalIterator>& rhs);
```

**Returns:** lhs == rhs.str().

```
template <class BidirectionalIterator>
bool operator != (
    typename iterator_traits<BidirectionalIterator>::value_type
    const& lhs,
    const sub_match<BidirectionalIterator>& rhs);
```

**Returns:** lhs != rhs.str().

```
template <class BidirectionalIterator>
bool operator < (
    typename iterator_traits<BidirectionalIterator>::value_type
    const& lhs,
    const sub_match<BidirectionalIterator>& rhs);
```

**Returns:** lhs < rhs.str().

```
template <class BidirectionalIterator>
bool operator > (
    typename iterator_traits<BidirectionalIterator>::value_type
    const& lhs,
    const sub_match<BidirectionalIterator>& rhs);
```

**Returns:** lhs > rhs.str().

```
template <class BidirectionalIterator>
bool operator >= (
    typename iterator_traits<BidirectionalIterator>::value_type
```



```
    const& lhs,  
    const sub_match<BidirectionalIterator>& rhs);
```

**Returns:** lhs >= rhs.str().

```
template <class BidirectionalIterator>  
bool operator <= (  
    typename iterator_traits<BidirectionalIterator>::value_type  
    const& lhs,  
    const sub_match<BidirectionalIterator>& rhs);
```

**Returns:** lhs <= rhs.str().

```
template <class BidirectionalIterator>  
bool operator == (const sub_match<BidirectionalIterator>& lhs,  
    typename iterator_traits<BidirectionalIterator>::value_type  
    const& rhs);
```

**Returns:** lhs.str() == rhs.

```
template <class BidirectionalIterator>  
bool operator != (const sub_match<BidirectionalIterator>& lhs,  
    typename iterator_traits<BidirectionalIterator>::value_type  
    const& rhs);
```

**Returns:** lhs.str() != rhs.

```
template <class BidirectionalIterator>  
bool operator < (const sub_match<BidirectionalIterator>& lhs,  
    typename iterator_traits<BidirectionalIterator>::value_type  
    const& rhs);
```

**Returns:** lhs.str() < rhs.

```
template <class BidirectionalIterator>  
bool operator > (const sub_match<BidirectionalIterator>& lhs,  
    typename iterator_traits<BidirectionalIterator>::value_type  
    const& rhs);
```

**Returns:** lhs.str() > rhs.

```
template <class BidirectionalIterator>  
bool operator >= (const sub_match<BidirectionalIterator>& lhs,  
    typename iterator_traits<BidirectionalIterator>::value_type  
    const& rhs);
```

**Returns:** lhs.str() >= rhs.

```
template <class BidirectionalIterator>  
bool operator <= (const sub_match<BidirectionalIterator>& lhs,  
    typename iterator_traits<BidirectionalIterator>::value_type  
    const& rhs);
```

**Returns:** lhs.str() <= rhs.

```
template <class charT, class traits, class BidirectionalIterator>
basic_ostream<charT, traits>&
    operator << (basic_ostream<charT, traits>& os
                const sub_match<BidirectionalIterator>& m);
```

**Returns:** (os << m.str()).

## 7.10 Class template match\_results

[tr.re.results]

Class template `match_results` denotes a collection of character sequences representing the result of a regular expression match. Storage for the collection is allocated and freed as necessary by the member functions of class `match_results`.

The class template `match_results` satisfies the requirements of a Sequence, as specified in [lib.sequence.reqmts], except that only operations defined for const-qualified Sequences are supported.

The `sub_match` object stored at index 0 represents sub-expression 0; that is to say the whole match. In this case the `sub_match` member `matched` is always true, unless a partial match was obtained as a result of the flag `regex_constants::partial_match` being passed to a regular expression algorithm, in which case member `matched` is false, and the members `first` and `second` represent the character range that formed the partial match. The `sub_match` object stored at index `n` denotes what matched the marked sub-expression `n` within the matched expression. If the sub-expression `n` participated in a regular expression match then the `sub_match` member `matched` evaluates to true, and members `first` and `second` denote the range of characters [`first`, `second`) which formed that match. Otherwise `matched` is false, and members `first` and `second` point to the end of the sequence that was searched. [Note: The `sub_match` objects representing different sub-expressions that did not participate in a regular expression match need not be distinct.—end note]

```
template <class BidirectionalIterator,
          class Allocator
          = allocator<sub_match<BidirectionalIterator> >
class match_results
{
public:
    typedef sub_match<BidirectionalIterator>
        value_type;
    typedef const typename allocator::const_reference
        const_reference;
    typedef const_reference
        reference;
    typedef implementation-defined
        const_iterator;
    typedef const_iterator
        iterator;
    typedef typename iterator_traits<BidirectionalIterator>
```

```

        ::difference_type difference_type;
typedef typename Allocator::size_type
    size_type;
typedef Allocator
    allocator_type;
typedef typename iterator_traits<BidirectionalIterator>
    ::value_type char_type;
typedef basic_string<char_type>
    string_type;

// construct/copy/destroy:
explicit match_results(const Allocator& a = Allocator());
match_results(const match_results& m);
match_results& operator=(const match_results& m);
~match_results();

// size:
size_type size() const;
size_type max_size() const;
bool empty() const;
// element access:
difference_type length(int sub = 0) const;
difference_type position(unsigned int sub = 0) const;
string_type str(int sub = 0) const;
const_reference operator[](int n) const;

const_reference prefix() const;

const_reference suffix() const;
const_iterator begin() const;
const_iterator end() const;
// format:
template <class OutputIterator>
OutputIterator format(OutputIterator out,
                    const string_type& fmt,
                    regex_constants::match_flag_type flags
                    = regex_constants::format_default) const;
string_type format(const string_type& fmt,
                 regex_constants::match_flag_type flags
                 = regex_constants::format_default) const;

allocator_type get_allocator() const;
void swap(match_results& that);
};

```

### 7.10.1 match\_results constructors

[tr.re.results.const]

In all match\_results constructors, a copy of the Allocator argument is used for any memory allocation performed by the constructor or member functions during the lifetime of the object.

```
match_results(const Allocator& a = Allocator());
```

**Effects:** Constructs an object of class match\_results. The postconditions of this function are indicated in Table 7.14

Table 7.14: match\_results(const Allocator&) effects

Element	Value
empty()	true
size()	0
str()	basic_string<charT>()

```
match_results(const match_results& m);
```

**Effects:** Constructs an object of class match\_results, as a copy of m.

```
match_results& operator=(const match_results& m);
```

**Effects:** Assigns m to \*this. The postconditions of this function are indicated in Table 7.15

Table 7.15: match\_results assignment operator effects

Element	Value
empty()	m.empty()
size()	m.size()
str(n)	m.str(n) for all integers n < m.size
prefix()	m.prefix()
suffix()	m.suffix()
(*this)[n]	m[n] for all integers n < m.size
length(n)	m.length(n) for all integers n < m.size
position(n)	m.position(n) for all integers n < m.size

### 7.10.2 match\_results size

[tr.re.results.size]

```
size_type size() const;
```

**Returns:** One plus the number of marked sub-expressions in the regular expression that was matched.

```
size_type max_size() const;
```

**Returns:** The maximum number of sub\_match elements that can be stored in \*this.

```
bool empty() const;
```

**Returns:** `size() == 0`.

### 7.10.3 `match_results` element access

[tr.re.results.acc]

```
difference_type length(int sub = 0) const;
```

**Returns:** `(*this)[sub].length()`.

```
difference_type position(unsigned int sub = 0) const;
```

**Returns:** `std::distance(prefix().first, (*this)[sub].first)`.

```
string_type str(int sub = 0) const;
```

**Returns:** `string_type((*this)[sub])`.

```
const_reference operator[](int n) const;
```

**Returns:** A reference to the `sub_match` object representing the character sequence that matched marked sub-expression `n`. If `n == 0` then returns a reference to a `sub_match` object representing the character sequence that matched the whole regular expression. If `n >= size()` then returns a `sub_match` object representing an unmatched sub-expression.

```
const_reference prefix() const;
```

**Returns:** A reference to the `sub_match` object representing the character sequence from the start of the string being matched/searched, to the start of the match found.

```
const_reference suffix() const;
```

**Returns:** A reference to the `sub_match` object representing the character sequence from the end of the match found to the end of the string being matched/searched.

```
const_iterator begin() const;
```

**Returns:** A starting iterator that enumerates over all the marked sub-expression matches stored in `*this`.

```
const_iterator end() const;
```

**Returns:** A terminating iterator that enumerates over all the marked sub-expression matches stored in `*this`.

### 7.10.4 `match_results` reformatting

[tr.re.results.reform]

```
OutputIterator format(OutputIterator out,  
    const string_type& fmt,  
    regex_constants::match_flag_type flags  
    = regex_constants::format_default);
```

**Requires:** The type `OutputIterator` conforms to the Output Iterator requirements [24.1.2].

**Effects:** Copies the character sequence `[fmt.begin(), fmt.end())` to `OutputIterator out`. For each format specifier or escape sequence in `fmt`, replace that sequence with either the character(s) it represents, or the sequence of characters within `*this` to which it refers. The bitmasks specified in `flags` determines what format specifiers or escape sequences are recognized, by default this is the format used by ECMA-262 [9], part 5.4.11 `String.prototype.replace`.

**Returns:** `out`.

```
string_type format(const string_type& fmt,
                  regex_constants::match_flag_type flags
                  = regex_constants::format_default);
```

**Effects:** Returns a copy of the string `fmt`. For each format specifier or escape sequence in `fmt`, replace that sequence with either the character(s) it represents, or the sequence of characters within `*this` to which it refers. The bitmasks specified in `flags` determines what format specifiers or escape sequences are recognized, by default this is the format used by ECMA-262 [9], part 5.4.11, `String.prototype.replace`.

### 7.10.5 `match_results` allocator [tr.re.results.all]

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

**Effects:** Returns a copy of the Allocator that was passed to the object's constructor.

### 7.10.6 `match_results` swap [tr.re.results.swap]

```
void swap(match_results& that);
```

**Effects:** Swaps the contents of the two sequences.

**Postcondition:** `*this` contains the sequence of matched sub-expressions that were in `that`, `that` contains the sequence of matched sub-expressions that were in `*this`.

**Complexity:** constant time.

## 7.11 Regular expression algorithms [tr.re.alg]

### 7.11.1 `regex_match` [tr.re.alg.match]

```
template <class BidirectionalIterator, class Allocator,
          class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 match_results<BidirectionalIterator, Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);
```

**Requires:** Type `BidirectionalIterator` satisfies the requirements of a Bidirectional Iterator (§24.1.4).

**Effects:** Determines whether there is an exact match between the regular expression `e`, and all of the character sequence `[first, last)`, parameter `flags` is used to control how the expression is matched against the character sequence. Returns `true` if such a match exists, `false` otherwise.

**Postconditions:** If the function returns `false`, then the effect on parameter `m` is unspecified, otherwise the effects on parameter `m` are given in table 7.16.

Table 7.16: Effects of `regex_match` algorithm

Element	Value
<code>m.size()</code>	<code>e.mark_count()</code>
<code>m.empty()</code>	<code>false</code>
<code>m.prefix().first</code>	<code>first</code>
<code>m.prefix().last</code>	<code>first</code>
<code>m.prefix().matched</code>	<code>false</code>
<code>m.suffix().first</code>	<code>last</code>
<code>m.suffix().last</code>	<code>last</code>
<code>m.suffix().matched</code>	<code>false</code>
<code>m[0].first</code>	<code>first</code>
<code>m[0].second</code>	<code>last</code>
<code>m[0].matched</code>	<code>true</code> if a full match was found, and <code>false</code> if it was a partial match (found as a result of the <code>regex_constants::match_partial</code> flag being set).
<code>m[n].first</code>	For all integers <code>n &lt; m.size()</code> , the start of the sequence that matched sub-expression <code>n</code> . Alternatively, if sub-expression <code>n</code> did not participate in the match, then <code>last</code> .
<code>m[n].second</code>	For all integers <code>n &lt; m.size()</code> , the end of the sequence that matched sub-expression <code>n</code> . Alternatively, if sub-expression <code>n</code> did not participate in the match, then <code>last</code> .
<code>m[n].matched</code>	For all integers <code>n &lt; m.size()</code> , <code>true</code> if sub-expression <code>n</code> participated in the match, <code>false</code> otherwise.

```
template <class BidirectionalIterator,
          class charT, class traits>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);
```

**Effects:** Behaves “as if” by constructing an instance of `match_results< BidirectionalIterator >` `what`, and then returning the result of `regex_match(first, last, what, e, flags)`.

```

template <class charT, class Allocator, class traits>
bool regex_match(const charT* str,
                 match_results<const charT*, Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);

```

**Returns:** `regex_match(str, str + char_traits<charT>::length(str), m, e, flags)`.

```

template <class ST, class SA, class Allocator,
          class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 match_results<typename basic_string<charT, ST, SA>::const_iterator,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);

```

**Returns:** `regex_match(s.begin(), s.end(), m, e, flags)`.

```

template <class charT, class traits>
bool regex_match(const charT* str,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);

```

**Returns:** `regex_match(str, str + char_traits<charT>::length(str), e, flags)`

```

template <class ST, class SA, class charT, class traits>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);

```

**Returns:** `regex_match(s.begin(), s.end(), e, flags)`.

### 7.11.2 `regex_search`

[[tr.re.alg.search](#)]

```

template <class BidirectionalIterator, class Allocator,
          class charT, class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                 match_results<BidirectionalIterator, Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);

```

**Requires:** Type `BidirectionalIterator` meets the requirements of a `Bidirectional Iterator` (24.1.4).



**Effects:** Determines whether there is some sub-sequence within  $[first, last)$  that matches the regular expression  $e$ , parameter  $flags$  is used to control how the expression is matched against the character sequence. Returns `true` if such a sequence exists, `false` otherwise.

**Postconditions:** If the function returns `false`, then the effect on parameter  $m$  is unspecified, otherwise the effects on parameter  $m$  are given in table 7.17.

Table 7.17: Effects of `regex_search` algorithm

Element	Value
<code>m.size()</code>	<code>e.mark_count()</code>
<code>m.empty()</code>	<code>false</code>
<code>m.prefix().first</code>	<code>first</code>
<code>m.prefix().last</code>	<code>m[0].first</code>
<code>m.prefix().matched</code>	<code>m.prefix().first != m.prefix().second</code>
<code>m.suffix().first</code>	<code>m[0].second</code>
<code>m.suffix().last</code>	<code>last</code>
<code>m.suffix().matched</code>	<code>m.suffix().first != m.suffix().second</code>
<code>m[0].first</code>	The start of the sequence of characters that matched the regular expression
<code>m[0].second</code>	The end of the sequence of characters that matched the regular expression
<code>m[0].matched</code>	<code>true</code> if a full match was found, and <code>false</code> if it was a partial match (found as a result of the <code>regex_constants::match_partial</code> flag being set).
<code>m[n].first</code>	For all integers $n < m.size()$ , the start of the sequence that matched sub-expression $n$ . Alternatively, if sub-expression $n$ did not participate in the match, then <code>last</code> .
<code>m[n].second</code>	For all integers $n < m.size()$ , the end of the sequence that matched sub-expression $n$ . Alternatively, if sub-expression $n$ did not participate in the match, then <code>last</code> .
<code>m[n].matched</code>	For all integers $n < m.size()$ , <code>true</code> if sub-expression $n$ participated in the match, <code>false</code> otherwise.

```
template <class charT, class Allocator, class traits>
bool regex_search(const charT* str, match_results<const charT*, Allocator>& m,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);
```

**Returns:** The result of `regex_search(str,`

str + char\_traits<charT>::length(str), m, e, flags).

```
template <class ST, class SA, class Allocator,
          class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                 match_results<typename basic_string<charT, ST, SA>::const_iterator,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);
```

**Returns:** The result of `regex_search(s.begin(), s.end(), m, e, flags)`.

```
template <class iterator, class charT, class traits>
bool regex_search(iterator first, iterator last,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);
```

**Effects:** Behaves “as if” by constructing an object what of type `match_results<BidirectionalIterator>` and then returning the result of `regex_search(first, last, what, e, flags)`.

```
template <class charT, class traits>
bool regex_search(const charT* str
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);
```

**Returns:** `regex_search(str, str + char_traits<charT>::length(str), e, flags)`

```
template <class ST, class SA,
          class charT, class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                 const basic_regex<charT, traits>& e,
                 regex_constants::match_flag_type flags
                 = regex_constants::match_default);
```

**Returns:** `regex_search(s.begin(), s.end(), e, flags)`.

### 7.11.3 `regex_replace`

[tr.re.alg.replace]

```
template <class OutputIterator, class BidirectionalIterator,
          class traits, class charT>
OutputIterator
regex_replace(OutputIterator out,
             BidirectionalIterator first,
             BidirectionalIterator last,
             const basic_regex<charT, traits>& e,
```

```

const basic_string<charT>& fmt,
regex_constants::match_flag_type flags
    = regex_constants::match_default);

```

**Effects:** Constructs a `regex_iterator` object `regex_iterator<BidirectionalIterator, charT, traits, Allocator> i(first, last, e, flags)`, and uses `i` to enumerate through all of the matches `m` of type `match_results<BidirectionalIterator>` that occur within the sequence `[first, last)`. If no such matches are found and `!(flags & regex_constants::format_no_copy)` then calls `std::copy(first, last, out)`. Otherwise, for each match found, if `!(flags & regex_constants::format_no_copy)` calls `std::copy(m.prefix().first, m.prefix().second, out)`, and then calls `m.format(out, fmt, flags)`. Finally, if `!(flags & regex_constants::format_no_copy)` calls `std::copy(last_m.suffix().first, last_m.suffix().second, out)` where `last_m` is a copy of the last match found. If `flags & regex_constants::format_first_only` is non-zero then only the first match found is replaced.

**Returns:** `out`.

```

template <class traits, class charT>
basic_string<charT>
regex_replace(const basic_string<charT>& s,
              const basic_regex<charT, traits>& e,
              const basic_string<charT>& fmt,
              regex_constants::match_flag_type flags
                = regex_constants::match_default);

```

**Effects:** Constructs an object `basic_string<charT> result`, calls `regex_replace(back_inserter(result), s.begin(), s.end(), e, fmt, flags)`, and then returns `result`.

## 7.12 Regular expression Iterators

[tr.re.iter]

### 7.12.1 Class template `regex_iterator`

[tr.re.regiter]

The class template `regex_iterator` is an iterator adapter; that is to say it represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence. `regex_iterator` finds (using `regex_search`) successive regular expression matches within the sequence from which it was constructed. After it is constructed, and every time `operator++` is used, the iterator finds and stores a value of `match_results<BidirectionalIterator>`. If the end of the sequence is reached (`regex_search` returns `false`), the iterator becomes equal to the end-of-sequence iterator value. The constructor with no arguments `regex_iterator()` always constructs an end of sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end of sequence is not defined. For any other iterator value a `const match_results<BidirectionalIterator>&` is returned. The result of `operator->` on an end of sequence is not defined. For any other iterator value a `const match_results<BidirectionalIterator>*` is returned. It is impossible to store things into `regex_iterators`. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

```

template <class BidirectionalIterator,
         class charT = iterator_traits<BidirectionalIterator>::value_type,
         class traits = regex_traits<charT> >
class regex_iterator
{
public:
    typedef basic_regex<charT, traits>
        regex_type;
    typedef match_results<BidirectionalIterator>
        value_type;
    typedef typename iterator_traits<BidirectionalIterator>::difference_type
        difference_type;
    typedef const value_type*
        pointer;
    typedef const value_type&
        reference;
    typedef std::forward_iterator_tag
        iterator_category;

    regex_iterator();
    regex_iterator(BidirectionalIterator a,
                  BidirectionalIterator b,
                  const regex_type& re,
                  regex_constants::match_flag_type m
                    = regex_constants::match_default);
    regex_iterator(const regex_iterator&);
    regex_iterator& operator=(const regex_iterator&);
    bool operator==(const regex_iterator&);
    bool operator!=(const regex_iterator&);
    const value_type& operator*();
    const value_type* operator->();
    regex_iterator& operator++();
    regex_iterator operator++(int);
private:
    match_results<BidirectionalIterator> what; // exposition only
    BidirectionalIterator end; // exposition only
    const regex_type* pre; // exposition only
    regex_constants::match_flag_type flags; // exposition only
};

```

### 7.12.1.1 regex\_iterator constructors

[tr.re.regiter.cnstr]

```
regex_iterator();
```

**Effects:** Default constructs the members `what`, `end` and `flags`, and sets the member `pre` equal to the

null-pointer constant.

```
regex_iterator(BidirectionalIterator a, BidirectionalIterator b,  
               const regex_type& re,  
               regex_constants::match_flag_type m  
               = regex_constants::match_default);
```

**Effects:** Default constructs all data members, and then calls `regex_search(a, b, what, re, m)`. If this returns `false` then sets `*this` equal to the end of sequence iterator, otherwise sets `end` equal to `b`, `pre` equal to `&re`, and `flags` equal to `m`.

```
regex_iterator(const regex_iterator& that);
```

**Effects:** Constructs all data members as a copy of `that`.

**Postconditions:** `*this == that`.

```
regex_iterator& operator=(const regex_iterator& that);
```

**Effects:** Sets all data members equal to those in `that`.

**Postconditions:** `*this == that`.

#### 7.12.1.2 `regex_iterator` comparisons

[tr.re.regiter.comp]

```
bool operator==(const regex_iterator& that);
```

**Effects:** If `(pre == 0) && (that.pre == 0)` then returns `true`, otherwise returns the result of `(pre == that.pre) && (end == that.end) && (flags == that.flags) && (what[0].first == that.what[0].first) && (what[0].second == that.what[0].second)`.

```
bool operator!=(const regex_iterator&);
```

**Effects:** Returns `!(*this == that)`.

#### 7.12.1.3 `regex_iterator` dereference

[tr.re.regiter.deref]

```
const value_type& operator*();
```

**Returns:** `what`.

```
const value_type* operator->();
```

**Returns:** `&what`.

#### 7.12.1.4 `regex_iterator` increment

[tr.re.regiter.incr]

```
regex_iterator& operator++();
```

**Effects:** If `what.prefix().first != what[0].second` and if the element `regex_constants::match_prev_avail` is not set in `flags` then sets it. Then calls `regex_search(what[0].second, end, what, *pre, ((what[0].first == what[0].second) ? flags | regex_constants::match_not_null : flags))`, and if this returns `false` then sets `*this` equal to the end of sequence iterator.

**Returns:** `*this`.

```
regex_iterator operator++(int);
```

**Effects:** Constructs a copy result of `*this`, then calls `++(*this)`.

**Returns:** result.

## 7.12.2 Class template `regex_token_iterator`

[tr.re.tokiter]

The class template `regex_token_iterator` is an iterator adapter; that is to say it represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence, and presenting one or more new strings for each match found. Each position enumerated by the iterator is a string that represents what matched a particular sub-expression within the regular expression. When class `regex_token_iterator` is used to enumerate a single sub-expression with index `-1`, then the iterator performs field splitting: that is to say it enumerates one string for each section of the character container sequence that does not match the regular expression specified.

After it is constructed, the iterator finds and stores a value of `match_results<BidirectionalIterator>` `what` (by calling `regex_search`) and sets the internal count `N` to zero. Every time `operator++` is used the count `N` is incremented; if `N` exceeds or equals `this->subs.size()`, then the iterator finds and stores the next value of `match_results<BidirectionalIterator>` and sets count `N` to zero.

If the end of sequence is reached (`regex_search` returns `false`), the iterator becomes equal to the *end-of-sequence* iterator value, unless the sub-expression being enumerated has index `-1`: In which case the iterator enumerates one last string that contains all the characters from the end of the last regular expression match to the end of the input sequence being enumerated, provided that this would not be an empty string.

The constructor with no arguments, `regex_iterator()`, always constructs an end of sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end of sequence is not defined. For any other iterator value a `const basic_string<charT>&` is returned (obtained by calling `this->what[subs[N]]`). The result of `operator->` on an end of sequence is not defined. For any other iterator value a `const basic_string<charT>*` is returned. It is impossible to store things into `regex_iterators`. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

```
template <class BidirectionalIterator,
          class charT = iterator_traits<BidirectionalIterator>::value_type,
          class traits = regex_traits<charT> >
class regex_token_iterator
{
public:
    typedef basic_regex<charT, traits>
```

```

        regex_type;
typedef basic_string<charT>
    value_type;
typedef typename iterator_traits<BidirectionalIterator>::difference_type
    difference_type;
typedef const value_type*
    pointer;
typedef const value_type&
    reference;
typedef std::forward_iterator_tag
    iterator_category;

regex_token_iterator();
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
    const regex_type& re,
    int submatch = 0, regex_constants::match_flag_type m
        = regex_constants::match_default);
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
    const regex_type& re,
    const std::vector<int>& submatches,
    regex_constants::match_flag_type m
        = regex_constants::match_default);

template <std::size_t N>
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
    const regex_type& re,
    const int (&submatches)[N],
    regex_constants::match_flag_type m
        = regex_constants::match_default);
regex_token_iterator(const regex_token_iterator&);
regex_token_iterator& operator=(const regex_token_iterator&);
bool operator==(const regex_token_iterator&);
bool operator!=(const regex_token_iterator&);
const value_type& operator*();
const value_type* operator->();
regex_token_iterator& operator++();
regex_token_iterator operator++(int);
private:
    match_results<BidirectionalIterator> what;    // exposition only
    BidirectionalIterator end;                  // exposition only
    const regex_type* pre;                      // exposition only
    regex_constants::match_flag_type flags;     // exposition only
    basic_string<charT> result;                 // exposition only
    std::size_t N;                             // exposition only
    std::vector<int> subs;                      // exposition only
};

```

### 7.12.2.1 `regex_token_iterator` constructors

[tr.re.tokiter.cnstr]

```
regex_token_iterator();
```

**Effects:** Constructs an end of sequence iterator, by default constructing all data members.

**Postconditions:** `pre == 0`.

```
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                    const regex_type& re,
                    int submatch = 0, regex_constants::match_flag_type m
                    = regex_constants::match_default);
```

**Preconditions:** `!re.empty()`.

**Effects:** Default constructs all data members and pushes the value `submatch` onto `subs`. Then if `regex_search(a, b, what, re, m) == true` sets `N` equal to zero, `flags` equal to `m`, `pre` equal to `&re`, `end` equal to `b`, and sets `result` equal to `((submatch == -1) ? value_type(what.prefix().str() : value_type(what[submatch].str()))`. Otherwise if the call to `regex_search` returns false, then if `(submatch == -1) && (a != b)` sets `result` equal to `value_type(a, b)` and `N` equal to `-1`. Otherwise sets `*this` equal to the end of sequence iterator.

```
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                    const regex_type& re,
                    const std::vector<int>& submatches,
                    regex_constants::match_flag_type m
                    = regex_constants::match_default);
```

**Preconditions:** `submatches.size() && !re.empty()`.

**Effects:** Copy constructs member `subs` from `submatches`, and default constructs the other data members. Then if `regex_search(a, b, what, re, m) == true` sets `N` equal to zero, `flags` equal to `m`, `pre` equal to `&re`, `end` equal to `b`, and sets `result` equal to `((submatch[N] == -1) ? value_type(what.prefix().str() : value_type(what[submatch[N]].str()))`. Otherwise if the call to `regex_search` returns false, then if `(submatch[0] == -1) && (a != b)` sets `result` equal to `value_type(a, b)` and `N` equal to `-1`. Otherwise sets `*this` equal to the end of sequence iterator.

```
template <std::size_t N>
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                    const regex_type& re,
                    const int (&submatches)[R],
                    regex_constants::match_flag_type m
                    = regex_constants::match_default);
```

**Preconditions:** `!re.empty()`.

**Effects:** Copy constructs member `subs` from the iterator sequence `[submatches, submatches+R)`, and default constructs the other data members. Then if `regex_search(a, b, what, re, m) == true` sets `N` equal to zero, `flags` equal to `m`, `pre` equal to `&re`, `end` equal to `b`, and sets `result` equal to `((subs[N] == -1) ? value_type(what.prefix().str() : value_type(what[subs[N]].str()))`



Otherwise if the call to `regex_search` returns false, then if `(subs[0] == -1) && (a != b)` sets `result` equal to `value_type(a, b)` and `N` equal to `-1`. Otherwise sets `*this` equal to the end of sequence iterator.

```
regex_token_iterator(const regex_token_iterator& that);
```

**Effects:** Constructs all data members as a copy of `that`.

**Postconditions:** `*this == that`.

```
regex_token_iterator& operator=(const regex_token_iterator& that);
```

**Effects:** Sets all data members equal to those in `that`.

**Postconditions:** `*this == that`.

#### 7.12.2.2 `regex_token_iterator` comparisons

[tr.re.tokiter.comp]

```
bool operator==(const regex_token_iterator&);
```

**Effects:** If `(pre == 0) && (that.pre == 0)` then returns true, otherwise returns the result of `(pre == that.pre) && (end == that.end) && (flags == that.flags) && (N == that.N) && (what[0].first == that.what[0].first) && (what[0].second == that.what[0].second)`.

```
bool operator!=(const regex_token_iterator&);
```

**Returns:** `!(*this == that)`.

#### 7.12.2.3 `regex_token_iterator` dereference

[tr.re.tokiter.deref]

```
const value_type& operator*();
```

**Effects:** Returns `result`.

```
const value_type* operator->();
```

**Effects:** Returns `&result`.

#### 7.12.2.4 `regex_token_iterator` increment

[tr.re.tokiter.incr]

```
regex_token_iterator& operator++();
```

**Effects:** if `N == -1` then sets `*this` equal to the end of sequence iterator. Otherwise if `N+1 < subs.size()`, then increments `N` and sets `result` equal to `((subs[N] == -1) ? value_type(what.prefix().str()) : value_type(what[subs[N]].str()))`.

Otherwise if `what.prefix().first != what[0].second` and if the element `regex_constants::match_prev_avail` is not set in `flags` then sets it. Then if `regex_search(what[0].second, end,`

```
what, *pre, ((what[0].first == what[0].second) ? flags regex_constants::match_
not_null : flags)) == true— sets N equal to zero, and sets result equal to ((subs[N] == -1) ?
value_type(what.prefix().str()) : value_type(what[subs[N]].str())).
```

Otherwise if the call to `regex_search` returns `false`, then let `last_end` be the value of `what[0].second` prior to the call to `regex_search`. Then if `last_end != end` and `subs[0] == -1` sets N equal to -1 and sets `result` equal to `value_type(last_end, end)`. Otherwise sets `*this` equal to the end of sequence iterator.

**Returns:** `*this`.

```
regex_token_iterator& operator++(int);
```

**Effects:** Constructs a copy result of `*this`, then calls `++(*this)`.

**Returns:** `result`.

## 7.13 Modified ECMAScript regular expression grammar[tr.re.grammar]

The regular expression grammar recognized by class template `basic_regex` is that specified by ECMA-262 [9], except as specified below.

Objects of type specialization of `basic_regex` store within themselves a default-constructed instance of their `traits` template parameter, henceforth referred to as `traits_inst`. This `traits_inst` object is used to support localization of the regular expression; `basic_regex` object member functions shall not call any locale dependent C or C++ API, including the formatted string input functions. Instead they shall call the appropriate `traits` member function to achieve the required effect.

The transformation from a sequence of characters to a finite state machine is accomplished by first transforming the sequence of characters to the sequence of tokens obtained by calling `traits_inst.syntax_type(c)` for each input character `c`. The regular expression grammar is then applied to the sequence of tokens in order to construct the finite state machine (this is to allow the regular expression syntax to be localized to a specific character set; for example to use Far Eastern ideographs, rather than Latin characters). Where `traits_inst.syntax_type(c)` returns `syntax_escape`, then the implementation shall call `traits_inst.escape_syntax_type(c)` for the character following the escape, in order to determine the grammatical meaning of the escape sequence. When `traits_inst.escape_syntax_type(c)` returns `escape_type_class` or `escape_type_not_class`, then the single character following the escape is converted to a string `n`, and passed to `traits_inst.lookup_classname(n)`, to determine the character classification to be matched against. If `traits_inst.lookup_classname(n)` returns `null`, then the escape is treated as an identity escape. The mapping of values returned by `syntax_type` and `escape_syntax_type` to characters within the ECMAScript grammar is specified in table 7.4 and table 7.5.

The following productions within the ECMAScript grammar are modified as follows:

```
CharacterClass ::
  [ [lookahead ∉ {^}] ClassRanges ]
  [ ^ ClassRanges ]
```

```
ClassAtom ::
```

```

-
ClassAtomNoDash
ClassAtomExClass
ClassAtomCollatingElement
ClassAtomEquivalence

```

The following new productions are then added:

```

ClassAtomExClass ::
    [: ClassName :]

ClassAtomCollatingElement ::
    [. ClassName .]

ClassAtomEquivalence ::
    [= ClassName =]

ClassName ::
    ClassNameCharacter
    ClassNameCharacter ClassName

ClassNameCharacter ::
    SourceCharacter but not one of "." "=" ":"

```

The productions `ClassAtomExClass`, `ClassAtomCollatingElement` and `ClassAtomEquivalence` provide the equivalent functionality to the same features in POSIX regular expressions [13].

The regular expression grammar may be modified by any `regex_constants::syntax_option_`-type flags specified when constructing an object of type specialization of `basic_regex` according to the rules in table 7.2.

A `ClassName` production when used in `ClassAtomExClass` is not valid if the value returned by `traits_inst.lookup_classname` for that name is zero. The names recognized as valid `ClassNames` are determined by the type of the traits class, but at least the following names shall be recognized: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`, `d`, `s`, `w`. In addition the following expressions shall be equivalent:

```

\d and [[:digit:]]
\D and [^[:digit:]]
\s and [[:space:]]
\S and [^[:space:]]
\w and [[:alnum:]]
\W and [^[:alnum:]]

```

The results from multiple calls to `traits_inst.lookup_classname` can be bitwise OR'ed together and subsequently passed to `traist_inst.is_class`.

A `ClassName` production when used in a `ClassAtomCollatingElement` production is not valid if the value returned by `traits_inst.lookup_collatename` for that name is an empty string.

A `ClassName` production when used in a `ClassAtomEquivalence` production is not valid if the value returned by `traits_inst.lookup_collatename` for that name is an empty string or if the value returned by `traits_inst.transform_primary` for the result of the call to `traits_inst.lookup_collatename` is an empty string.

When the sequence of characters being transformed to a finite state machine contains an invalid class name the translator shall throw an exception object of type `bad_expression`.

Where the regular expression grammar requires the conversion of a sequence of characters to an integral value, this is accomplished by calling `traits_inst.toi`.

The behavior of the internal finite state machine representation when used to match a sequence of characters is as described in ECMAScript [9]. The behavior is modified according to any `match_flag_type` flags 7.5.2 specified when using the regular expression object in one of the regular expression algorithms 7.11. The behavior is also localized by interaction with the traits class template parameter as follows:

- During matching of a regular expression finite state machine against a sequence of characters, two characters `c` and `d` are compared using `traits_inst.translate(c, flags() & icaase) == traits_inst.translate(d, flags() & icaase)`.
- During matching of a regular expression finite state machine against a sequence of characters, comparison of a collating element range `c1-c2` against a character `c` is conducted as follows: if `flags() & regex_constants::collate` is false then the character `c` is matched if `c1 <= c && c <= c2`, otherwise `c` is matched in accordance with the following algorithm:

```
string_type str1 =
    string_type(1, traits_inst.translate(c1,
        flags() & icaase);
string_type str2 =
    string_type(1, traits_inst.translate(c2,
        flags() & icaase);
string_type str =
    string_type(1, traits_inst.translate(c,
        flags() & icaase);
return traits_inst.transform(str1)
    <= traits_inst.transform(str)
    && traits_inst.transform(str)
    <= traits_inst.transform(str2);
```

- During matching of a regular expression finite state machine against a sequence of characters, testing whether a collating element is a member of a primary equivalence class is conducted by first converting the collating element and the equivalence class to sort keys using `traits::transform_primary`, and then comparing the sort keys for equality.

- During matching of a regular expression finite state machine against a sequence of characters, a character `c` is a member of character class `some_name`, if `traits_inst.is_class(c, traits_inst.lookup_classname(some_name))`.

# Chapter 8

## Iterators

[tr.iter]

This clause describes extensions to the iterator facilities of the C++ standard. It describes a new set of iterator requirements, and an iterator adapter class that uses the new requirements.

### 8.1 New iterator requirements

[tr.iter.req]

#### 8.1.1 Iterator access requirements

[tr.iter.acc]

In the tables below,  $X$  is an iterator type,  $a$  is a constant object of type  $X$ ,  $T$  is `std::iterator_traits<X>::value_type`, and  $v$  is a constant object of type  $T$ .

##### 8.1.1.1 Readable iterator requirements

[tr.iter.readable]

A class or built-in type  $X$  models the *Readable Iterator* concept for the value type  $T$  if the expressions in table 8.1 are valid and respect the stated semantics.  $U$  is the type of any specified member of type  $T$ .

Table 8.1: Readable Iterator Requirements (in addition to CopyConstructible)

Expression	Return Type	Note/Precondition
<code>iterator_traits&lt;X&gt;::value_type</code>	$T$	Any non-reference, non-cv-qualified type
<code>iterator_traits&lt;X&gt;::reference</code>	Convertible to <code>iterator_traits&lt;X&gt;::value_type</code>	
<code>is_readable&lt;X&gt;::type</code>	<code>true_type</code>	
<code>*a</code>	<code>iterator_traits&lt;X&gt;::reference</code>	pre: $a$ is dereferenceable. If $a == b$ then $*a$ is equivalent to $*b$ .

*continued from previous page*

a->m	U&	pre: (*a).m is well-defined. Equivalent to (*a).m
------	----	--

### 8.1.1.2 Writeable iterator requirements

[tr.iter.writeable]

A class or built-in type X models the *Writeable Iterator* concept if the expressions in table 8.2 are valid and respect the stated semantics. In addition, a model of *Writeable Iterator* must include in its documentation the set of *value types* that it allows for output.

Table 8.2: Writeable Iterator Requirements (in addition to CopyConstructible)

Expression	Return Type	Note/Precondition
is_writeable<X>::type	true_type	
*a = o		pre: The type of o is in the set of value types of X.

### 8.1.1.3 Swappable iterator requirements

[tr.iter.swappable]

A class or built-in type X models the *Swappable Iterator* concept if the expressions in table 8.3 are valid and respect the stated semantics.

Table 8.3: Swappable Iterator Requirements (in addition to CopyConstructible)

Expression	Return Type	Note/Precondition
is_swappable<X>::type	true_type	
iter_swap(a,b)	void	the pointed to values are exchanged

[Note: !n iterator that is a model of the *Readable* and *Writable Iterator* concepts is also a model of *Swappable Iterator*. -end note]

### 8.1.1.4 Lvalue iterator requirements

[tr.iter.lvalue]

The *Lvalue Iterator* concept adds the requirement that the `reference` type be a reference to the value type of the iterator.

Table 8.4: Lvalue Iterator Requirements

Expression	Return Type	Note/Precondition
<code>iterator_traits&lt;X&gt;::reference</code>	<code>T&amp;</code>	<code>T</code> is <i>cv</i> <code>iterator_traits&lt;X&gt;::value_type</code> where <i>cv</i> is an optional <i>cv</i> -qualification.

## 8.1.2 Iterator traversal concepts

[tr.iter.traversal]

In the tables below, `X` is an iterator type, `a` and `b` are constant objects of type `X`, `r` and `s` are mutable objects of type `X`, `T` is `std::iterator_traits<X>::value_type`, and `v` is a constant object of type `T`.

### 8.1.2.1 Incrementable iterator requirements

[tr.iter.incrementable]

A class or built-in type `X` models the *Incrementable Iterator* concept if the expressions in table 8.5 are valid and respect the stated semantics.

Table 8.5: Incrementable Iterator Requirements (in addition to Assignable, CopyConstructible)

Expression	Return Type	Assertion/Semantics
<code>++r</code>	<code>X&amp;</code>	<code>&amp;r == &amp;++r</code>
<code>r++</code>	<code>X</code>	<code>X tmp = r;</code> <code>++r</code> <code>return tmp;</code>
<code>traversal_category&lt;X&gt;::type</code>	Convertible to <code>incrementable_iterator_tag</code>	

### 8.1.2.2 Single Pass Iterators

[tr.iter.singlepass]

A class or built-in type `X` models the *Single Pass Iterator* concept if the expressions in table 8.6 are valid and respect the stated semantics.

Table 8.6: Single Pass Iterator Requirements (in addition to Incrementable Iterator and Equality Comparable)

Expression	Return Type	Assertion/Semantics/Pre-/Post-condition
<code>++r</code>	<code>X&amp;</code>	pre: <code>r</code> is dereferenceable; post: <code>r</code> is dereferenceable or <code>r</code> is past-the-end
<code>a == b</code>	convertible to <code>bool</code>	<code>==</code> is an equivalence relation over its domain



<i>continued from previous page</i>		
<code>a != b</code>	convertible to <code>bool</code>	<code>!(a == b)</code>
<code>traversal_category&lt;X&gt;::type</code>	Convertible to <code>single_pass_iterator_tag</code>	

### 8.1.2.3 Forward Traversal Iterators

[tr.iter.forward]

A class or built-in type `X` models the *Forward Traversal Iterator* concept if the expressions in table 8.7 are valid and respect the stated semantics.

Table 8.7: Forward Traversal Iterator Requirements (in addition to Single Pass Iterator)

Expression	Return Type	Assertion/Semantics/Pre-/Post-condition
<code>X u;</code>	<code>X&amp;</code>	note: <code>u</code> may have a singular value.
<code>++r</code>	<code>X&amp;</code>	<code>r == s</code> and <code>r</code> is dereferenceable implies <code>++r == ++s</code> .
<code>iterator_traits&lt;X&gt;::difference_type</code>	A signed integral type representing the distance between iterators	
<code>traversal_category&lt;X&gt;::type</code>	Convertible to <code>forward_traversal_iterator_tag</code>	

### 8.1.2.4 Bidirectional Traversal Iterators

[tr.iter.bidirectional]

A class or built-in type `X` models the *Bidirectional Traversal Iterator* concept if the expressions in table 8.8 are valid and respect the stated semantics.

Table 8.8: Bidirectional Traversal Iterator Requirements (in addition to Forward Traversal Iterator)

Expression	Return Type	Assertion/Semantics/Pre-/Post-condition
<code>--r</code>	<code>X&amp;</code>	pre: there exists <code>s</code> such that <code>r == ++s</code> . post: <code>s</code> is dereferenceable. <code>--(++r) == r</code> . <code>--r == --s</code> implies <code>r == s</code> . <code>&amp;r == &amp;--r</code> .
<code>r--</code>	convertible to <code>const X&amp;</code>	<code>X tmp = r;</code> <code>--r;</code> <code>return tmp;</code>
<code>traversal_category&lt;X&gt;::type</code>	Convertible to <code>bidirectional_traversal_iterator_tag</code>	

### 8.1.2.5 Random Access Traversal Iterators

[tr.iter.random]

A class or built-in type  $X$  models the *Random Access Traversal Iterator* concept if the expressions in table 8.9 are valid and respect the stated semantics. In this table,  $\text{Distance}$  is `iterator_traits<X>::difference_type` and  $n$  represents a constant object of type  $\text{Distance}$ .

Table 8.9: Random Access Traversal Iterator Requirements (in addition to Bidirectional Traversal Iterator)

Expression	Return Type	Assertion/Semantics/Pre-/Post-condition
<code>r += n</code>	$X\&$	<pre>Distance m = n; if (m &gt;= 0) while (m--) ++r; else while (m++) --r; return r;</pre>
<code>a + n, n + a</code>	$X$	<code>X tmp = a; return tmp += n;</code>
<code>r -= n</code>	$X\&$	<code>return r += -n</code>
<code>a - n</code>	$X$	<code>X tmp = a; return tmp -= n;</code>
<code>b - a</code>	$\text{Distance}$	<pre>a &lt; b ? distance(a,b) : -distance(b,a).    pre: there exists a value n of Distance such that a + n == b.    b == a + (b - a).</pre>
<code>a[n]</code>	convertible to $T$	<code>*(a + n)</code> . pre: $a$ is a readable iterator
<code>a[n] = v</code>	convertible to $T$	<code>*(a + n) = v</code> . pre: $a$ is a writable iterator
<code>a &lt; b</code>	convertible to <code>bool</code>	<code>b - a &gt; 0</code> . <code>&lt;</code> is a total ordering relation
<code>a &gt; b</code>	convertible to <code>bool</code>	<code>b &lt; a</code> . <code>&gt;</code> is a total ordering relation
<code>a &gt;= b</code>	convertible to <code>bool</code>	<code>!(a &lt; b)</code>
<code>a &lt;= b</code>	convertible to <code>bool</code>	<code>!(a &gt; b)</code>
<code>traversal_category&lt;X&gt;::type</code>	Convertible to <code>random_access_traversal_iterator_tag</code>	

## 8.2 New iterator primitives

[tr.iter.primitives]

### 8.2.1 Additions to header `<iterator>` synopsis

[tr.iter.synopsis]

```
// tr.iter.traits, traits and tags
template <class Iterator> struct is_readable;
template <class Iterator> struct is_writable;
template <class Iterator> struct is_swappable;
template <class Iterator> struct traversal_category;

enum iterator_access { readable_iterator = 1, writable_iterator = 2,
    swappable_iterator = 4, lvalue_iterator = 8 };

template <unsigned int access_bits, class TraversalTag>
struct iterator_tag : /* appropriate old category or categories */ {
    static const iterator_access access =
        (iterator_access)access_bits &
        (readable_iterator | writable_iterator | swappable_iterator);
    typedef TraversalTag traversal;
};

struct incrementable_iterator_tag { };
struct single_pass_iterator_tag : incrementable_iterator_tag { };
struct forward_traversal_tag : single_pass_iterator_tag { };
struct bidirectional_traversal_tag : forward_traversal_tag { };
struct random_access_traversal_tag : bidirectional_traversal_tag { };

struct null_category_tag { };
struct input_output_iterator_tag : input_iterator_tag, output_iterator_tag {};
```

### 8.2.2 New iterator traits and tags

[tr.iter.traits]

The `iterator_tag` class template is an iterator category tag that encodes the access enum and traversal tag in addition to being compatible with the original iterator tags. The `iterator_tag` class inherits from one of the original iterator tags according to the following pseudo-code.

```
inherit-category(access, traversal-tag) =
    if ((access & readable_iterator) && (access & lvalue_iterator)) {
        if (traversal-tag is convertible to random_access_traversal_tag)
            return random_access_iterator_tag;
        else if (traversal-tag is convertible to bidirectional_traversal_tag)
            return bidirectional_iterator_tag;
        else if (traversal-tag is convertible to forward_traversal_tag)
            return forward_iterator_tag;
```

```

        else if (traversal-tag is convertible to single_pass_traversal_tag)
            if (access-tag is convertible to writable_iterator_tag)
                return input_output_iterator_tag;
            else
                return input_iterator_tag;
        else
            return null_category_tag;
    } else if ((access & readable_iterator) and (access & writable_iterator)
        and traversal-tag is convertible to single_pass_iterator_tag)
        return input_output_iterator_tag;
    else if (access & readable_iterator
        and traversal-tag is convertible to single_pass_iterator_tag)
        return input_iterator_tag;
    else if (access & writable_iterator
        and traversal-tag is convertible to incrementable_iterator_tag)
        return output_iterator_tag;
    else
        return null_category_tag;

```

If the argument for `TraversalTag` is not convertible to `incrementable_iterator_tag` then the program is ill-formed.

The `is_readable`, `is_writable`, `is_swappable`, and `traversal_category` class templates are traits classes. For iterators whose `iterator_traits<Iter>::iterator_category` type is `iterator_tag`, these traits obtain the access enum and traversal member type from within `iterator_tag`. For iterators whose `iterator_traits<Iter>::iterator_category` type is not `iterator_tag` and instead is a tag convertible to one of the original tags, the appropriate traversal tag and access bits are deduced. The following pseudo-code describes the algorithm.

```

is-readable(Iterator) =
    cat = iterator_traits<Iterator>::iterator_category;
    if (cat == iterator_tag<Access,Traversal>)
        return Access & readable_iterator;
    else if (cat is convertible to input_iterator_tag)
        return true;
    else
        return false;

is-writable(Iterator) =
    cat = iterator_traits<Iterator>::iterator_category;
    if (cat == iterator_tag<Access,Traversal>)
        return Access & writable_iterator;
    else if (cat is convertible to output_iterator_tag)
        return true;
    else if (
        cat is convertible to forward_iterator_tag
        and iterator_traits<Iterator>::reference is a

```

```

        mutable reference)
    return true;
else
    return false;

is-swappable(Iterator) =
    cat = iterator_traits<Iterator>::iterator_category;
    if (cat == iterator_tag<Access,Traversal>)
        return Access & swappable_iterator;
    else if (cat is convertible to forward_iterator_tag) {
        if (iterator_traits<Iterator>::reference is a const reference)
            return false;
        else
            return true;
    } else
        return false;

traversal-category(Iterator) =
    cat = iterator_traits<Iterator>::iterator_category;
    if (cat == iterator_tag<Access,Traversal>)
        return Traversal;
    else if (cat is convertible to random_access_iterator_tag)
        return random_access_traversal_tag;
    else if (cat is convertible to bidirectional_iterator_tag)
        return bidirectional_traversal_tag;
    else if (cat is convertible to forward_iterator_tag)
        return forward_traversal_tag;
    else if (cat is convertible to input_iterator_tag)
        return single_pass_iterator_tag;
    else if (cat is convertible to output_iterator_tag)
        return incrementable_iterator_tag;
    else
        return null_category_tag;

```

The following specializations provide the access and traversal category tags for pointer types.

```

template <typename T>
struct is_readable<const T*> { typedef true_type type; };
template <typename T>
struct is_writable<const T*> { typedef false_type type; };
template <typename T>
struct is_swappable<const T*> { typedef false_type type; };

template <typename T>
struct is_readable<T*> { typedef true_type type; };
template <typename T>
struct is_writable<T*> { typedef true_type type; };

```

```

template <typename T>
struct is_swappable<T*> { typedef true_type type; };

template <typename T>
struct traversal_category<T*>
{
    typedef random_access_traversal_tag type;
};

```

## 8.3 Iterator adaptor and facade

[tr.iter.helpers]

### 8.3.1 Header <iterator\_helper> synopsis

[tr.iter.helper.syn]

```

struct use_default;
const unsigned use_default_access = -1;

struct iterator_core_access { /* implementation detail */ };

template <
    class Derived
    , class Value
    , unsigned AccessCategory
    , class TraversalCategory
    , class Reference = Value&
    , class Difference = ptrdiff_t
>
class iterator_facade;

template <
    class Derived
    , class Base
    , class Value = use_default
    , unsigned Access = use_default_access
    , class Traversal = use_default
    , class Reference = use_default
    , class Difference = use_default
>
class iterator_adaptor;

template <
    class Iterator
    , class Value = use_default
    , unsigned Access = use_default_access

```

```

    , class Traversal = use_default
    , class Reference = use_default
    , class Difference = use_default
>
class indirect_iterator;

template <class Iterator>
class reverse_iterator;

template <
    class UnaryFunction
    , class Iterator
    , class Reference = use_default
    , class Value = use_default
>
class transform_iterator;

template <class Predicate, class Iterator>
class filter_iterator;

template <
    class Incrementable
    , unsigned Access = use_default_access
    , class Traversal = use_default
    , class Difference = use_default
>
class counting_iterator

template <class UnaryFunction>
class function_output_iterator;

```

### 8.3.2 Iterator facade

[tr.iter.facade]

`iterator_facade` is a base class template that implements the interface of standard iterators in terms of a few core functions and associated types, to be supplied by a derived iterator class.

#### 8.3.2.1 Class template `iterator_facade`

[tr.iter.facade.facade]

```

template <
    class Derived
    , class Value
    , unsigned AccessCategory
    , class TraversalCategory
    , class Reference = /* see below */

```

```

    , class Difference = ptrdiff_t
>
class iterator_facade {
public:
    typedef remove_cv<Value>::type value_type;
    typedef Reference reference;
    typedef /* see description of operator-> */ pointer;
    typedef Difference difference_type;
    typedef iterator_tag<AccessCategory, TraversalCategory> iterator_category;

    reference operator*() const;
    /* see below */ operator->() const;
    /* see below */ operator[](difference_type n) const;
    Derived& operator++();
    Derived operator++(int);
    Derived& operator--();
    Derived operator--(int);
    Derived& operator+=(difference_type n);
    Derived& operator-=(difference_type n);
    Derived operator-(difference_type n) const;
};

// Comparison operators
template <class Dr1, class V1, class AC1, class TC1, class R1, class D1,
         class Dr2, class V2, class AC2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type // exposition
operator ==(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const& lhs,
           iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const& rhs);

template <class Dr1, class V1, class AC1, class TC1, class R1, class D1,
         class Dr2, class V2, class AC2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator !=(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const& lhs,
           iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const& rhs);

template <class Dr1, class V1, class AC1, class TC1, class R1, class D1,
         class Dr2, class V2, class AC2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator <(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const& lhs,
           iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const& rhs);

template <class Dr1, class V1, class AC1, class TC1, class R1, class D1,
         class Dr2, class V2, class AC2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator <=(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const& lhs,
           iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const& rhs);

```



```

template <class Dr1, class V1, class AC1, class TC1, class R1, class D1,
         class Dr2, class V2, class AC2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator >(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const& lhs,
          iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const& rhs);

template <class Dr1, class V1, class AC1, class TC1, class R1, class D1,
         class Dr2, class V2, class AC2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator >=(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const& lhs,
           iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const& rhs);

template <class Dr1, class V1, class AC1, class TC1, class R1, class D1,
         class Dr2, class V2, class AC2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator >=(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const& lhs,
           iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const& rhs);

// Iterator difference
template <class Dr1, class V1, class AC1, class TC1, class R1, class D1,
         class Dr2, class V2, class AC2, class TC2, class R2, class D2>
typename enable_if_interoperable<Dr1, Dr2, bool>::type
operator -(iterator_facade<Dr1, V1, AC1, TC1, R1, D1> const& lhs,
          iterator_facade<Dr2, V2, AC2, TC2, R2, D2> const& rhs);

// Iterator addition
template <class Derived, class V, class AC, class TC, class R, class D>
Derived operator+ (iterator_facade<Derived, V, AC, TC, R, D> const&,
                 typename Derived::difference_type n)

```

[*Note*: The `enable_if_interoperable` template used above is for exposition purposes. The member operators should be only be in an overload set provided the derived types `Dr1` and `Dr2` are interoperable, by which we mean they are convertible to each other. The `enable_if_interoperable` approach uses SFINAE to take the operators out of the overload set when the types are not interoperable. —*end note*]

### 8.3.2.2 `iterator_facade` requirements

[**tr.iter.facade.req**]

Some of the constraints on template parameters to `iterator_facade` are expressed in terms of resulting nested types and should be viewed in the context of their impact on `iterator_traits<Derived>`.

The `Derived` template parameter must be a class derived from `iterator_facade`.

The nested `::value_type` type will be the same as `remove_cv<Value>::type`, so the `Value` parameter must be an (optionally const-qualified) non-reference type.

`AccessCategory` must be an unsigned value which uses no more bits than the greatest value of `niterator_access`.

The nested `::reference` will be the same as the `Reference` parameter; it must be a suitable reference type for the resulting iterator. The default for the `Reference` parameter is `Value&`.

Table 8.10 describes the other requirements on the `Derived` parameter. Depending on the resulting iterator's `iterator_category`, a subset of the expressions listed in the table are required to be valid. The operations in the first column must be accessible to member functions of class `iterator_core_access`.

In table 8.10, `X` is the derived iterator type, `a` is an object of type `X`, `b` and `c` are objects of type `const X`, `n` is an object of `X::difference_type`, `y` is a constant object of a single pass iterator type interoperable with `X`, and `z` is a constant object of a random access traversal iterator type interoperable with `X`.

Table 8.10: Requirements on the `Derived` parameter

Expression	Return Type	Assertion/Note	Required to implement Iterator Concept(s)
<code>c.dereference()</code>	<code>X::reference</code>		Readable Iterator, Writable Iterator
<code>c.equal(b)</code>	convertible to <code>bool</code>	true iff <code>b</code> and <code>c</code> are equivalent.	Single Pass Iterator
<code>c.equal(y)</code>	convertible to <code>bool</code>	true iff <code>c</code> and <code>y</code> refer to the same position. Implements <code>c == y</code> and <code>c != y</code> .	Single Pass Iterator
<code>a.advance(n)</code>	unused		Random Access Traversal Iterator
<code>a.increment()</code>	unused		Incrementable Iterator
<code>a.decrement()</code>	unused		Bidirectional Traversal Iterator
<code>c.distance_to(b)</code>	convertible to <code>X::difference_type</code>	equivalent to <code>distance(c, b)</code>	Random Access Traversal Iterator
<code>c.distance_to(z)</code>	convertible to <code>X::difference_type</code>	equivalent to <code>distance(c, z)</code> . Implements <code>c - z</code> , <code>c &lt; z</code> , <code>c &lt;= z</code> , <code>c &gt; z</code> , and <code>c &gt;= z</code> .	Random Access Traversal Iterator

### 8.3.2.3 `iterator_facade` operations

[tr.iter.facade.ops]

The operations in this section are described in terms of operations on the core interface of `Derived` which may be inaccessible (i.e. `private`). The implementation should access these operations through member functions of class `iterator_core_access`.

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

**Returns:** `static_cast<Derived const*>(this)->dereference()`

```
operator->() const; // see below
```

**Returns:** If `X::reference` is a reference type, returns an object of type `X::pointer` equal to: `&static_cast<Derived const*>(this)->dereference()`

Otherwise returns an object of unspecified type such that, given an object `a` of type `X`, `a->m` is equivalent to `(w = *a, w.m)` for some temporary object `w` of type `X::value_type`.

The type `X::pointer` is `Value*` if `is_writable_iterator<X>::value` is true, and `Value const*` otherwise.

```
unspecified operator[](difference_type n) const;
```

**Returns:** an object convertible to `X::reference` and holding a copy `p` of `a+n` such that, for a constant object `v` of type `X::value_type`, `X::reference(a[n] = v)` is equivalent to `p = v`.

```
Derived& operator++();
```

**Effects:**

```
static_cast<Derived*>(this)->increment();  
return *this;
```

```
Derived operator++(int);
```

**Effects:**

```
Derived tmp(static_cast<Derived const*>(this));  
++*this;  
return tmp;
```

```
Derived& operator--();
```

**Effects:**

```
static_cast<Derived*>(this)->decrement();  
return *this;
```

```
Derived operator--(int);
```

**Effects:**

```
Derived tmp(static_cast<Derived const*>(this));  
--*this;  
return tmp;
```

```
Derived& operator+=(difference_type n);
```

**Effects:**

```
static_cast<Derived*>(this)->advance(n);  
return *this;
```

```
Derived& operator-=(difference_type n);
```

**Effects:**

```
static_cast<Derived*>(this)->advance(-n);
return *this;
```

```
Derived operator-(difference_type n) const;
```

**Effects:**

```
Derived tmp(static_cast<Derived const*>(this));
return tmp -= n;
```

**Returns:** `static_cast<Derived const*>(this)->advance(-n);`

**8.3.3 Iterator adaptor****[tr.iter.adapt]**

The `iterator_adaptor` is a base class template derived from an instantiation of `iterator_facade`. The core interface functions expected by `iterator_facade` are implemented in terms of the `iterator_adaptor`'s `Base` template parameter. A class derived from `iterator_adaptor` typically redefines some of the core interface functions to adapt the behavior of the `Base` type. Whether the derived class models any of the standard iterator concepts depends on the operations supported by the `Base` type and which core interface functions of `iterator_facade` are redefined in the `Derived` class.

**8.3.3.1 Class template `iterator_adaptor`****[tr.iter.adapt.adapt]**

```
bool
template <
    class Derived
    , class Base
    , class Value      = use_default
    , unsigned Access  = use_default_access
    , class Traversal  = use_default
    , class Reference  = use_default
    , class Difference = use_default
>
class iterator_adaptor
    : public iterator_facade<Derived, /* see details ...*/>
{
    friend class iterator_core_access;
public:
    iterator_adaptor();
    explicit iterator_adaptor(Base iter);
    Base base() const;
protected:
    Base const& base_reference() const;
    Base& base_reference();
```

```

private: // Core iterator interface for iterator_facade.
    typename iterator_adaptor::reference dereference() const;

    template <
        class OtherDerived, class OtherIterator, class V, class C, class R, class D
    >
    bool equal(iterator_adaptor<OtherDerived, OtherIterator, V, C, R, D> const& x) const;

    void advance(typename iterator_adaptor::difference_type n);
    void increment();
    void decrement();

    template <
        class OtherDerived, class OtherIterator, class V, class C, class R, class D
    >
    typename iterator_adaptor::difference_type distance_to(
        iterator_adaptor<OtherDerived, OtherIterator, V, C, R, D> const& y) const;

private:
    Base m_iterator;
};

```

### 8.3.3.2 iterator\_adaptor requirements

[tr.iter.adapt.req]

The Derived template parameter must be a derived class of iterator\_adaptor. The Base type must implement the expressions involving m\_iterator in the specifications of those private member functions of iterator\_adaptor that are not redefined by the Derived class and that are needed to model the concept corresponding to iterator\_traits<Derived>::iterator\_category according to the requirements of iterator\_facade. The rest of the template parameters specify the types for the member typedefs in iterator\_facade. The following pseudo-code specifies the traits types for iterator\_adaptor.

```

if (Value != use_default)
    value_type = remove_cv<Value>::type;
else
    value_type = iterator_traits<Base>::value_type;

if (Traversal != use_default)
    traversal_category = Traversal
else
    traversal_category = traversal_category< Base >::type

iterator_category = iterator_tag<
    access_category
    , traversal_category

```

```

        >

if (Access != use_default)
{
    access_category = Access
}
else
{
    access_category
        = access_category< Base >::value

    if (is_const<Value>)
        access_category &= ~writable_iterator;
}

iterator_category = iterator_tag<
    access_category
    , traversal_category
>

if (Reference != use_default)
    reference = Reference
else if (Value != use_default)
    reference = Value&
else
    reference = iterator_traits<Base>::reference

```

### 8.3.3.3 iterator\_adaptor public operations

[tr.iter.adapt.op]

```
iterator_adaptor();
```

**Requires:** The Base type must be Default Constructible.

**Returns:** An instance of iterator\_adaptor with m\_iterator default constructed.

```
explicit iterator_adaptor(Base iter);
```

**Returns:** An instance of iterator\_adaptor with m\_iterator copy constructed from iter.

```
Base base() const;
```

**Returns:** m\_iterator.

### 8.3.3.4 iterator\_adaptor protected member functions

[tr.iter.adapt.prot]

```
Base const& base_reference() const;
```

**Returns:** A const reference to `m_iterator`.

```
Base& base_reference();
```

**Returns:** A non-const reference to `m_iterator`.

### 8.3.3.5 `iterator_adaptor` private member functions

[tr.iter.adapt.priv]

```
typename iterator_adaptor::reference dereference() const;
```

**Returns:** `*m_iterator`

```
template <
class OtherDerived, class OtherIterator, class V, class C, class R, class D
>
bool equal(iterator_adaptor<OtherDerived, OtherIterator, V, C, R, D> const& x) const;
```

**Returns:** `m_iterator == x.base()`

```
void advance(typename iterator_adaptor::difference_type n);
```

**Effects:** `m_iterator += n`

```
void increment();
```

**Effects:** `++m_iterator`

```
void decrement();
```

**Effects:** `--m_iterator`

```
template <
class OtherDerived, class OtherIterator, class V, class C, class R, class D
>
typename iterator_adaptor::difference_type distance_to(
    iterator_adaptor<OtherDerived, OtherIterator, V, C, R, D> const& y) const;
```

**Returns:** `y.base() - m_iterator`.

## 8.4 Specialized adaptors

[tr.iter.spec]

[*Note:* The `enable_if_convertible<X,Y>::type` expression used in this section is for exposition purposes. The converting constructors for specialized adaptors should be only be in an overload set provided that an object of type `X` is implicitly convertible to an object of type `Y`. The `enable_if_convertible` approach uses SFINAE to take the constructor out of the overload set when the types are not implicitly convertible. —end note]

## 8.4.1 Indirect iterator

[tr.iter.ind]

The indirect iterator adapts an iterator by applying an extra dereference inside of `operator*`( ). For example, this iterator adaptor makes it possible to view a container of pointers (e.g. `list<foo*>`) as if it were a container of the pointed-to type (e.g. `list<foo>`).

### 8.4.1.1 Class template `indirect_iterator`

[tr.iter.ind.ind]

```
template <
    class Iterator
    , class Value = use_default
    , unsigned Access = use_default_access
    , class Traversal = use_default
    , class Reference = use_default
    , class Difference = use_default
>
class indirect_iterator
    : public iterator_adaptor< /* see discussion */>
{
    friend class iterator_core_access;
public:
    indirect_iterator();
    indirect_iterator(Iterator x);
    template <
        class Iterator2, class Value2, unsigned Access2, class Traversal2
        , class Reference2, class Difference2
    >
    indirect_iterator(
        indirect_iterator<
            Iterator2, Value2, Access2, Traversal2, Reference2, Difference2
        > const& y
        , typename enable_if_convertible<Iterator2, Iterator>::type* = 0
        // exposition
    );
private: // as-if specification
    typename indirect_iterator::reference dereference() const
    {
        return **this->base();
    }
};
```

### 8.4.1.2 `indirect_iterator` requirements

[tr.iter.ind.req]

The `value_type` of the `Iterator` template parameter should itself be dereferenceable. The return type of



the operator\* for the value\_type must be the same type as the Reference template parameter. The Value template parameter will be the value\_type for the indirect\_iterator, unless Value is const. If Value is const X, then value\_type will be non-const X. The default for Value is

```
iterator_traits< iterator_traits<Iterator>::value_type >::value_type
```

If the default is used for Value, then there must be a valid specialization of iterator\_traits for the value type of the base iterator.

The Reference parameter will be the reference type of the indirect\_iterator. The default is Value&.

The Access and Traversal parameters are passed unchanged to the corresponding parameters of the iterator\_adaptor base class, and the Iterator parameter is passed unchanged as the Base parameter to the iterator\_adaptor base class.

The indirect iterator will model the most refined traversal concept that is modeled by the Iterator type. The indirect iterator will model the most refined access concept that is modeled by the value type of Iterator.

### 8.4.1.3 indirect\_iterator operations

[tr.iter.ind.op]

```
indirect_iterator();
```

**Requires:** Iterator must be Default Constructible.

**Returns:** An instance of indirect\_iterator with a default constructed base object.

```
indirect_iterator(Iterator x);
```

**Returns:**

An instance of indirect\_iterator with the iterator\_adaptor subobject copy constructed from x.

```
template <
    class Iterator2, class Value2, unsigned Access, class Traversal
    , class Reference2, class Difference2
>
indirect_iterator(
    indirect_iterator<
        Iterator2, Value2, Access, Traversal, Reference2, Difference2
    > const& y
    , typename enable_if_convertible<Iterator2, Iterator>::type* = 0
    // exposition
);
```

**Requires:** Iterator2 is implicitly convertible to Iterator.

**Returns:** An instance of indirect\_iterator that is a copy of y.

## 8.4.2 Reverse iterator

[tr.iter.rev]

The reverse iterator adaptor flips the direction of a base iterator's motion. Invoking `operator++()` moves the base iterator backward and invoking `operator--()` moves the base iterator forward.

### 8.4.2.1 Class template `reverse_iterator`

[tr.iter.rev]

```
template <class Iterator>
class reverse_iterator :
    public iterator_adaptor< reverse_iterator<Iterator>, Iterator >
{
    friend class iterator_core_access;
public:
    reverse_iterator() {}
    explicit reverse_iterator(Iterator x) ;

    template<class OtherIterator>
    reverse_iterator(
        reverse_iterator<OtherIterator> const& r
        , typename enable_if_convertible<OtherIterator, Iterator>::type* = 0
        // exposition
    );

private: // as-if specification
    typename reverse_iterator::reference dereference() const
    {
        return *prior(this->base());
    }

    void increment() { --this->base_reference(); }
    void decrement() { ++this->base_reference(); }

    void advance(typename reverse_iterator::difference_type n)
    {
        this->base_reference() += -n;
    }

    template <class OtherIterator>
    typename reverse_iterator::difference_type
    distance_to(reverse_iterator<OtherIterator> const& y) const
    {
        return this->base_reference() - y.base();
    }
};
```

### 8.4.2.2 `reverse_iterator` requirements

[tr.iter.rev.req]

The base `Iterator` must be a model of Bidirectional Traversal Iterator. The resulting `reverse_iterator` will be a model of the most refined standard traversal and access concepts that are modeled by `Iterator`.

### 8.4.2.3 `reverse_iterator` operations

[tr.iter.rev.op]

```
reverse_iterator();
```

**Requires:** `Iterator` must be Default Constructible.

**Returns:** An instance of `reverse_iterator` with a default constructed base object.

```
explicit reverse_iterator(Iterator x);
```

**Returns:** An instance of `reverse_iterator` with a base object copy constructed from `x`.

```
template<class OtherIterator>
reverse_iterator(
    reverse_iterator<OtherIterator> const& r
    , typename enable_if_convertible<OtherIterator, Iterator>::type* = 0
    // exposition
);
```

**Requires:** `OtherIterator` is implicitly convertible to `Iterator`.

**Returns:** An instance of `reverse_iterator` that is a copy of `r`.

## 8.4.3 Transform iterator

[tr.iter.trans]

The transform iterator adapts an iterator by applying some function object to the result of dereferencing the iterator. In other words, the `operator*` of the transform iterator first dereferences the base iterator, passes the result of this to the function object, and then returns the result.

### 8.4.3.1 Class template `transform_iterator`

[tr.iter.trans.trans]

```
template <class UnaryFunction,
          class Iterator,
          class Reference = use_default,
          class Value = use_default>
class transform_iterator
: public iterator_adaptor</* see discussion */>
{
    friend class iterator_core_access;
public:
    transform_iterator();
    transform_iterator(Iterator const& x, UnaryFunction f);
```

```

template<class OtherIterator, class R2, class V2>
transform_iterator(
    transform_iterator<UnaryFunction, OtherIterator, R2, V2> const& t
    , typename enable_if_convertible<OtherIterator, Iterator>::type* = 0
    // exposition
);

UnaryFunction functor() const;
private:
    typename transform_iterator::value_type dereference() const;
    UnaryFunction m_f;
};

```

### 8.4.3.2 transform\_iterator requirements

[tr.iter.trans.req]

The type `UnaryFunction` must be `Assignable`, `Copy Constructible`, and the expression `f(*i)` must be valid where `f` is an object of type `UnaryFunction`, `i` is an object of type `Iterator`, and where the type of `f(*i)` must be `result_of<UnaryFunction(iterator_traits<Iterator>::reference)>::type`.

The type `Iterator` must at least model `Readable Iterator`. The resulting `transform_iterator` models the most refined of the following options that is also modeled by `Iterator`.

- `Writable Lvalue Iterator` if `result_of<UnaryFunction(iterator_traits<Iterator>::reference)>::type` is a non-const reference.
- `Readable Lvalue Iterator` if `result_of<UnaryFunction(iterator_traits<Iterator>::reference)>::type` is a const reference.
- `Readable Iterator` otherwise.

The `transform_iterator` models the most refined standard traversal concept that is modeled by `Iterator`.

The reference type of `transform_iterator` is `result_of<UnaryFunction(iterator_traits<Iterator>::reference)>::type`. The value\_type is `remove_cv<remove_reference<reference>>::type`.

### 8.4.3.3 transform\_iterator public operations

[tr.iter.trans.op]

```
transform_iterator();
```

**Returns:** An instance of `transform_iterator` with `m_f` and `m_iterator` default constructed.

```
transform_iterator(Iterator const& x, UnaryFunction f);
```

**Returns:** An instance of `transform_iterator` with `m_f` initialized to `f` and `m_iterator` initialized to `x`.

```

template<class OtherIterator, class R2, class V2>
transform_iterator(
    transform_iterator<UnaryFunction, OtherIterator, R2, V2> const& t
    , typename enable_if_convertible<OtherIterator, Iterator>::type* = 0
    // exposition
);

```

**Returns:** An instance of `transform_iterator` that is a copy of `t`.

**Requires:** `OtherIterator` is implicitly convertible to `Iterator`.

```
UnaryFunction functor() const;
```

**Returns:** `m_f`

#### 8.4.3.4 `transform_iterator` private operations

[tr.iter.trans.priv]

```
typename transform_iterator::value_type dereference() const;
```

**Returns:** `m_f(transform_iterator::dereference())`.

### 8.4.4 Filter iterator

[tr.iter.filt]

The filter iterator adaptor creates a view of an iterator range in which some elements of the range are skipped over. A predicate function object controls which elements are skipped. When the predicate is applied to an element, if it returns `true` then the element is retained and if it returns `false` then the element is skipped over. When skipping over elements, it is necessary for the filter adaptor to know when to stop so as to avoid going past the end of the underlying range. Therefore the constructor of the filter iterator takes two iterator parameters: the position for the filtered iterator and the end of the range.

#### 8.4.4.1 Class template `filter_iterator`

[tr.iter.filt.filt]

```

template <class Predicate, class Iterator>
class filter_iterator
    : public iterator_adaptor<
        filter_iterator<Predicate, Iterator>, Iterator
        , use_default
        , /* see details */
    >
{
public:
    filter_iterator();
    filter_iterator(Predicate f, Iterator x, Iterator end = Iterator());
    filter_iterator(Iterator x, Iterator end = Iterator());
    template<class OtherIterator>
    filter_iterator(

```

```

        filter_iterator<Predicate, OtherIterator> const& t
        , typename enable_if_convertible<OtherIterator, Iterator>::type* = 0
        // exposition
        );
    Predicate predicate() const;
    Iterator end() const;

private: // as-if specification
    void increment()
    {
        ++(this->base_reference());
        satisfy_predicate();
    }

    void satisfy_predicate()
    {
        while (this->base() != this->m_end && !this->m_predicate(*this->base()))
            ++(this->base_reference());
    }

    Predicate m_predicate;
    Iterator m_end;
};

```

#### 8.4.4.2 filter\_iterator requirements

[tr.iter.filt.req]

The base `Iterator` parameter must be a model of Readable Iterator and Single Pass Iterator. The resulting `filter_iterator` will be a model of Forward Traversal Iterator if `Iterator` is, otherwise the `filter_iterator` will be a model of Single Pass Iterator. The access category of the `filter_iterator` will be the same as the access category of `Iterator`.

The `Predicate` must be Assignable, Copy Constructible, and the expression `p(x)` must be valid where `p` is an object of type `Predicate`, `x` is an object of type `iterator_traits<Iterator>::value_type`, and where the type of `p(x)` must be convertible to `bool`.

#### 8.4.4.3 filter\_iterator operations

[tr.iter.filt.op]

```
filter_iterator();
```

**Requires:** `Predicate` and `Iterator` must be Default Constructible.

**Returns:** a `filter_iterator` whose predicate is a default constructed `Predicate` and whose `end` is a default constructed `Iterator`.

```
filter_iterator(Predicate f, Iterator x, Iterator end = Iterator());
```

**Returns:** A `filter_iterator` at position `x` that filters according to predicate `f` and that will not increment past `end`.

```
filter_iterator(Iterator x, Iterator end = Iterator());
```

**Requires:** Predicate must be Default Constructible.

**Returns:** A filter\_iterator at position x that filters according to a default constructed Predicate and that will not increment past end.

```
template <class OtherIterator>
filter_iterator(
    filter_iterator<Predicate, OtherIterator> const& t
    , typename enable_if_convertible<OtherIterator, Iterator>::type* = 0
    // exposition
    );
```

**Requires:** OtherIterator is implicitly convertible to Iterator.

**Returns:** A copy of iterator t.

```
Predicate predicate() const;
```

**Returns:** A copy of the predicate object used to construct \*this.

```
Iterator end() const;
```

**Returns:** The object end used to construct \*this.

## 8.4.5 Counting iterator

[tr.iter.count]

The counting iterator adaptor implements dereference by returning a reference to the base object. The other operations are implemented by the base m\_iterator, as per the inheritance from iterator\_adaptor.

### 8.4.5.1 Class template counting\_iterator

[tr.iter.count.count]

```
template <
    class Incrementable
    , unsigned Access = use_default_access
    , class Traversal = use_default
    , class Difference = use_default
>
class counting_iterator
: public iterator_adaptor<
    counting_iterator<Incrementable, Access, Traversal, Difference>
    , Incrementable
    , Incrementable
    , Access
    , /* see details for traversal category */
    , Incrementable const&
    , Incrementable const*
>
```

```

        , /* distance = Difference or a signed integral type */>
    {
        friend class iterator_core_access;
    public:
        counting_iterator();
        counting_iterator(counting_iterator const& rhs);
        counting_iterator(Incrementable x);
    private:
        typename counting_iterator::reference dereference() const
        {
            return this->base_reference();
        }
    };

```

[*Note*: implementers are encouraged to provide an implementation of `distance_to` and a `difference_type` that avoids overflows in the cases when the `Incrementable` type is a numeric type. —*end note*]

#### 8.4.5.2 `counting_iterator` requirements

[**tr.iter.count.req**]

The `Incrementable` type must be Default Constructible, Copy Constructible, and Assignable. The default `distance` is an implementation defined signed integral type.

The resulting `counting_iterator` models Readable Lvalue Iterator.

Furthermore, if you wish to create a counting iterator that is a Forward Traversal Iterator, then the following expressions must be valid:

```

Incrementable i, j;
++i           // pre-increment
i == j       // operator equal

```

If you wish to create a counting iterator that is a Bidirectional Traversal Iterator, then pre-decrement is also required:

```
--i
```

If you wish to create a counting iterator that is a Random Access Traversal Iterator, then these additional expressions are also required:

```

counting_iterator::difference_type n;
i += n
n = i - j
i < j

```

#### 8.4.5.3 `counting_iterator` operations

[**tr.iter.count.op**]

```
counting_iterator();
```

**Returns:** A default constructed instance of `counting_iterator`.



```
counting_iterator(counting_iterator const& rhs);
```

**Returns:** An instance of `counting_iterator` that is a copy of `rhs`.

```
counting_iterator(Incrementable x);
```

**Returns:** An instance of `counting_iterator` with its base object copy constructed from `x`.

## 8.4.6 Function output iterator

[tr.iter.fn]

The function output iterator adaptor makes it easier to create custom output iterators. The adaptor takes a unary function and creates a model of Output Iterator. Each item assigned to the output iterator is passed as an argument to the unary function. The motivation for this iterator is that creating a conforming output iterator is non-trivial, particularly because the proper implementation usually requires a proxy object.

### 8.4.6.1 Class template `function_output_iterator`

[tr.iter.fn.fn]

```
template <class UnaryFunction>
class function_output_iterator {
public:
    typedef iterator_tag<
        writable_iterator
        , incrementable_traversal_tag
    > iterator_category;
    typedef void          value_type;
    typedef void          difference_type;
    typedef void          pointer;
    typedef void          reference;

    explicit function_output_iterator(const UnaryFunction& f = UnaryFunction());

    struct output_proxy {
        output_proxy(UnaryFunction& f);
        template <class T> output_proxy& operator=(const T& value);
    };
    output_proxy operator*();
    function_output_iterator& operator++();
    function_output_iterator& operator++(int);
};
```

### 8.4.6.2 `function_output_iterator` requirements

[tr.iter.fn.req]

The `UnaryFunction` must be Assignable, Copy Constructible, and the expression  $f(x)$  must be valid, where `f` is an object of type `UnaryFunction` and `x` is an object of a type accepted by `f`. The resulting `function_output_iterator` is a model of the Writable and Incrementable Iterator concepts.

### 8.4.6.3 `function_output_iterator` operations

[tr.iter.fn.op]

```
explicit function_output_iterator(const UnaryFunction& f = UnaryFunction());
```

**Returns:** An instance of `function_output_iterator` with `f` stored as a data member.

```
output_proxy operator*();
```

**Returns:** An instance of `output_proxy` constructed with a copy of the unary function `f`.

```
function_output_iterator& operator++();
```

**Returns:** `*this`

```
function_output_iterator& operator++(int);
```

**Returns:** `*this`

### 8.4.6.4 `function_output_iterator::output_proxy` operations

[tr.iter.fn.proxy]

```
output_proxy(UnaryFunction& f);
```

**Returns:** An instance of `output_proxy` with `f` stored as a data member.

```
template <class T> output_proxy& operator=(const T& value);
```

**Effects:**

```
m_f(value);  
return *this;
```

# Bibliography

- [1] David Abrahams, Jeremy Siek, Thomas Witt, “Iterator Facade and Adaptor,” WG21 Document N1530=03-0113, 2003.
- [2] David Abrahams, Jeremy Siek, Thomas Witt, “New Iterator Concepts,” WG21 Document N1550=03-0133, 2003.
- [3] Milton Abramowitz and Irene A. Stegun (eds.): *Handbook of Mathematical Functions with Formulas, Graphs and Mathematical Tables*, volume 55 of National Bureau of Standards Applied Mathematics Series. U. S. Government Printing Office, Washington, DC: 1964. Reprinted with corrections, Dover Publications: 1972.
- [4] Matthew Austern, “A Proposal to Add Hash Tables to the Standard Library (revision 4),” WG21 Document N1456=03-0039, 2003.
- [5] Walter Brown, “A Proposal to Add Mathematical Special Functions to the C++ Standard Library,” WG21 Document N1422 = 03-0004, 2003.
- [6] Peter Dimov, “A Proposal to Add an Enhanced Member Pointer Adaptor to the Library Technical Report,” WG21 Document N1432=03-0014, 2003.
- [7] Peter Dimov, Beman Dawes, and Greg Colvin, “A Proposal to Add General Purpose Smart Pointers to the Library Technical Report,” WG21 Document N1450=03-0033, 2003.
- [8] Peter Dimov, Douglas Gregor, Jaakko Järvi, and Gary Powell, “A Proposal to Add an Enhanced Binder to the Library Technical Report (revision 1),” WG21 Document N1455=03-0038, 2003.
- [9] Ecma International, *ECMAScript Language Specification*, Standard Ecma-262, third edition, 1999.
- [10] Douglas Gregor, “A Proposal to add a Polymorphic Function Object Wrapper to the Standard Library,” WG21 Document N1402=02-0060, 2002.
- [11] Douglas Gregor, “A uniform method for computing function object return types (revision 1),” WG21 Document N1454=03-0037, 2003.
- [12] Douglas Gregor and Peter Dimov, “A proposal to add a reference wrapper to the standard library (revision 1),” WG21 Document N1453=03-0036, 2003.
- [13] IEEE, *Information Technology—Portable Operating System Interface (POSIX)*, IEEE Standard 1003.1-2001.

- [14] International Standards Organization: *Quantities and units, Third edition*. International Standard ISO 31-11:1992. ISBN 92-67-10185-4.
- [15] International Standards Organization: *Programming Languages – C, Second edition*. International Standard ISO/IEC 9899:1999.
- [16] International Standards Organization: *Programming Languages – C++*. International Standard ISO/IEC 14882:1998.
- [17] Jaakko Järvi, “Proposal for adding tuple types into the standard library Programming Language C++,” WG21 Document N1403=02-0061, 2002.
- [18] John Maddock, “A Proposal to add Type Traits to the Standard Library,” WG21 Document 03-0006 = N1424, 2003.
- [19] John Maddock, “A Proposal to add Regular Expressions to the Standard Library,” WG21 Document 03-0011= N1429, 2003.
- [20] Jens Maurer, “A Proposal to Add an Extensible Random Number Facility to the Standard Library (Revision 2),” WG21 Document N1452, 2003.