ISO/IEC PDTR 19768

Doc No: N1745=05-0005
Date: 2005-01-17
Reply to: Matt Austern
austern@apple.com

Proposed Draft Technical Report on C++ Library Extensions

Contents

List of Tables

1 General [tr.intro]

1 This technical report describes extensions to the *C++ standard library* that is described in the International Standard for the C++ programming language [14].

2 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.

3 The goal of this technical report it 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 Relation to C++ Standard Library Introduction [tr.description]

1 Unless otherwise specified, the whole of the ISO C++ Standard Library introduction [lib.library] is included into this Technical Report by reference.

1.2 Categories of extensions [tr.intro.ext]

1 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 6.1.4.

2 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 [14].

1.3 Namespaces and headers [tr.intro.namespaces]

1 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 specifed, 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*]

2 Unless otherwise specified, references 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::`.

3 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.

4 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.

2 General Utilities [tr.util]

1 This clause describes basic components used to implement other library facilities. They may also be used by C++ programs.

2 The following subclauses describe reference wrappers and smart pointers, as summarized in Table 1.

Table 1: Utilities library summary

Subclause	Header(s)
2.1 Reference wrapper	`<functional>`
2.2 Smart pointers	`<memory>`

2.1 Reference wrappers [tr.util.refwrap]

2.1.1 Additions to header `<functional>` synopsis [tr.util.refwrp.synopsis]

```
namespace std {
namespace tr1 {
  template <class T> class reference_wrapper;

  template <class T> reference_wrapper<T> ref(T&);
  template <class T> reference_wrapper<const T> cref(const T&);

  template <class T> reference_wrapper<T> ref(reference_wrapper<T>);
  template <class T> reference_wrapper<const T> cref(reference_wrapper<T>);
} // namespace tr1
} // namespace std
```

2.1.2 Class template `reference_wrapper` [tr.util.refwrp.refwrp]

```
template <class T> class reference_wrapper
  : public unary_function<T1, R>          // see below
  : public binary_function<T1, T2, R>     // see below
{
public :
  // types
  typedef T type;
  typedef see below result_type; // Not always defined
```

```
    // construct/copy/destroy
    explicit reference_wrapper(T&);
    reference_wrapper(const reference_wrapper<T>& x);

    // assignment
    reference_wrapper& operator=(const reference_wrapper<T>& x);

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

    // invocation
    template <class T1, class T2, ..., class TN>
    typename result_of<T(T1, T2, ..., TN)>::type
    operator() (T1&, T2&, ..., TN&) const;
  };
```

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

2 `reference_wrapper` has a weak result type ([3.3]).

3 The template instantiation `reference_wrapper<T>` shall be derived from `std::unary_function<T1, R>` only if the type `T` is any of the following:

- a function type or a pointer to function type taking one argument of type `T1` and returning `R`
- a pointer to member function type with cv-qualifier *cv* and no arguments; the type `T1` is *cv* `T*` and `R` is the return type of the pointer to member function
- a class type that is derived from `std::unary_function<T1, R>`

4 The template instantiation `reference_wrapper<T>` shall be derived from `std::binary_function<T1, T2, R>` only if the type `T` is any of the following:

- a function type or a pointer to function type taking two arguments of types `T1` and `T2` and returning `R`
- a pointer to member function with cv-qualifier *cv* and taking one argument of type `T2`; the type `T1` is *cv* `T*` and `R` is the return type of the pointer to member function
- a class type that is derived from `std::binary_function<T1, T2, R>`

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

`explicit reference_wrapper(T& t);`

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

2 *Throws:* nothing.

`reference_wrapper(const reference_wrapper<T>& x);`

3 *Postconditions:* `*this` references the object that `x` references.

4 *Throws:* nothing.

2.1.2.2 `reference_wrapper` assignment [tr.util.refwrp.assign]

```
reference_wrapper& operator=(const reference_wrapper<T>& x);
```

1 *Postconditions:* `*this` references the object that `x` references.

2 *Throws:* Nothing.

2.1.2.3 `reference_wrapper` access [tr.util.refwrp.access]

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

1 *Returns:* The stored reference.

2 *Throws:* nothing.

```
T& get() const;
```

3 *Returns:* The stored reference.

4 *Throws:* nothing.

2.1.2.4 reference_wrapper invocation [tr.util.refwrp.invoke]

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

1 *Returns:* *`INVOKE`*`(get(), a1, a2, ..., aN)`. ([3.3])

2 *Note:* `operator()` is described for exposition only. Implementations are not required to provide an actual `reference_wrapper::operator()`. Implementations are permitted to support `reference_wrapper` function invocation through multiple overloaded operators or through other means.

2.1.2.5 reference_wrapper helper functions [tr.util.refwrp.helpers]

```
template <class T> reference_wrapper<T> ref(T& t);
```

1 *Returns:* `reference_wrapper<T>(t)`

2 *Throws:* nothing.

```
template <class T> reference_wrapper<T> ref(reference_wrapper<T>t);
```

3 *Returns:* `ref(t.get())`

4 *Throws:* nothing.

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

5 *Returns:* `reference_wrapper <const T>(t)`

6 *Throws:* nothing.

```
template <class T> reference_wrapper<const T> cref(reference_wrapper<T> t);
```

7 *Returns:* `cref(t.get());`

8 *Throws:* nothing.

2.2 Smart pointers [tr.util.smartptr]

2.2.1 Additions to header `<memory>` synopsis [tr.util.smartptr.synopsis]

```
namespace std {
namespace tr1 {
  // [2.2.2] Class bad_weak_ptr
  class bad_weak_ptr;

  // [2.2.3] Class template shared_ptr
  template<class T> class shared_ptr;

  // [2.2.3.6] shared_ptr comparisons
  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);

  // [2.2.3.8] shared_ptr specialized algorithms
  template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);

  // [2.2.3.9] shared_ptr 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);
  template<class T, class U> shared_ptr<T> const_pointer_cast(shared_ptr<U> const& r);

  // [2.2.3.7] shared_ptr I/O
  template<class E, class T, class Y>
    basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);

  // [2.2.3.10] shared_ptr get_deleter
  template<class D, class T> D * get_deleter(shared_ptr<T> const& p);

  // [2.2.4] Class template weak_ptr
  template<class T> class weak_ptr;

  // [2.2.4.6] weak_ptr comparison
  template<class T, class U> bool operator<(weak_ptr<T> const& a, weak_ptr<U> const& b);

  // [2.2.4.7] weak_ptr specialized algorithms
  template<class T> void swap(weak_ptr<T>& a, weak_ptr<T>& b);
```

```
    // [2.2.5] Class enable_shared_from_this
    template<class T> class enable_shared_from_this;
  } // namespace tr1
  } // namespace std
```

2.2.2 Class `bad_weak_ptr` [tr.util.smartptr.weakptr]

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

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

`bad_weak_ptr();`

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

3 *Throws:* nothing.

2.2.3 Class template `shared_ptr` [tr.util.smartptr.shared]

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

    // [2.2.3.1] 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);

    // [2.2.3.2] destructor
    ~shared_ptr();

    // [2.2.3.3] 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);

    // [2.2.3.4] 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);

    // [2.2.3.5] observers
    T* get() const;
    T& operator*() const;
    T* operator->() const;
    long use_count() const;
    bool unique() const;
    operator unspecified-bool-type() const;
  };

  // [2.2.3.6] shared_ptr comparisons
  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);

  // [2.2.3.7] shared_ptr I/O
  template<class E, class T, class Y>
    basic_ostream<E, T>& operator<< (basic_ostream<E, T>& os, shared_ptr<Y> const& p);

  // [2.2.3.8] shared_ptr specialized algorithms
  template<class T> void swap(shared_ptr<T>& a, shared_ptr<T>& b);

  // [2.2.3.9] shared_ptr 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);
  template<class T, class U> shared_ptr<T> const_pointer_cast(shared_ptr<U> const& r);

  // [2.2.3.10] shared_ptr get_deleter
  template<class D, class T> D * get_deleter(shared_ptr<T> const& p);
} // namespace tr1
} // namespace std
```

2 Specializations of `shared_ptr` shall be CopyConstructible, Assignable, and LessThanComparable, allowing their use in standard containers. Specializations of `shared_ptr` shall be convertible to `bool`, allowing their use in boolean expressions and declarations in conditions. The template parameter `T` of `shared_ptr` may be an incomplete type.

3 [*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();
```

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

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

3 *Throws:* nothing.

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

4 *Requires:* `p` is convertible to `T *`. `Y` is a complete type. The expression `delete p` shall be well-formed, shall have well defined behavior, and shall not throw exceptions.

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

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

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

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

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

9 *Requires:* `p` is convertible to `T *`. `D` is `CopyConstructible`. The copy constructor and destructor of `D` shall not throw exceptions. The expression `d(p)` shall be well-formed, shall have well defined behavior, and shall not throw exceptions.

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

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

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

13 *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);
```

14 *Requires:* For the second constructor `Y*` is convertible to `T*`.

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

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

17 *Throws:* nothing.

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

18 *Requires:* `Y*` is convertible to `T*`.

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

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

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

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

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

23 *Requires:* `r.release()` shall be convertible to `T *`. `Y` shall be a complete type. The expression `delete r.release()` shall be well-formed, shall have well defined behavior, and shall not throw exceptions.

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

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

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

27 *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();
```

1 *Effects:*

- If `*this` is *empty*, or if it *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.

2 *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);
```

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

2 *Returns:* `*this`.

3 *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);
```

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

2 *Throws:* nothing.

```
void reset();
```

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

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

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

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

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

2.2.3.5 `shared_ptr` observers [tr.util.smartptr.shared.obs]

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

1 *Returns:* the stored pointer.

2 *Throws:* nothing.

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

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

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

5 *Throws:* nothing.

6 *Notes:* When `T` is `void`, attempting to instantiate this member function renders the program ill-formed. [*Note:* Instantiating `shared_ptr<void>` does not necessarily result in instantiating this member function. *–end note*]

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

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

8 *Returns:* `get()`.

9 *Throws:* nothing.

`long use_count() const;`

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

11 *Throws:* nothing.

12 [*Note:* `use_count()` is not necessarily efficient. Use only for debugging and testing purposes, not for production code. *—end note*]

`bool unique() const;`

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

14 *Throws:* nothing.

15 [*Note:* `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`. *—end note*]

`operator` *`unspecified-bool-type`* `() const;`

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

17 *Throws:* nothing.

18 [*Note:* This conversion operator allows *shared_ptr* objects to be used in boolean contexts. [*Example:* `if (p && p->valid())` *—end example.*] The actual target type is typically a pointer to a member function, avoiding many of the implicit conversion pitfalls. *—end note*]

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);`

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

2 *Throws:* nothing.

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

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

4 *Throws:* nothing.

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

5 *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* or are both empty.

6 *Throws:* nothing.

7 [*Note:* Allows `shared_ptr` objects to be used as keys in associative containers. *—end note*]

2.2.3.7 `shared_ptr` I/O [tr.util.smartptr.shared.io]

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

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

2 *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);
```

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

2 *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);
```

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

2 *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`.

3 *Throws:* nothing.

4 [*Note:* 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. *—end note*]

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

5 *Requires:* The expression `dynamic_cast<T*>(r.get())` shall be well-formed and shall have well defined behavior.

6 *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.

7 *Throws:* nothing.

8 [*Note:* 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. *—end note*]

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

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

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

11 *Throws:* nothing.

12 [*Note:* The seemingly equivalent expression `shared_ptr<T>(const_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice. —*end note*]

2.2.3.10 get_deleter [tr.util.smartptr.getdeleter]

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

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

2 *Throws:* nothing.

2.2.4 Class template `weak_ptr` [tr.util.smartptr.weak]

1 The `weak_ptr` class template stores a weak reference to an object that is 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 std {
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);
  } // namespace tr1
  } // namespace std
```

2 Specializations of `weak_ptr` shall be CopyConstructible, Assignable, and LessThanComparable, allowing their use in standard containers. The template parameter `T` of `weak_ptr` may be an incomplete type.

2.2.4.1 `weak_ptr` constructors [tr.util.smartptr.weak.const]

```
weak_ptr();
```

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

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

3 *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);
```

4 *Requires:* For the second and third constructors, `Y*` shall be convertible to `T*`.

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

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

7 *Throws:* nothing.

2.2.4.2 `weak_ptr` destructor [tr.util.smartptr.weak.dest]

```
~weak_ptr();
```

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

2 *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);
```

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

2 *Throws:* nothing.

3 *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);
```

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

2 *Throws:* nothing.

```
void reset();
```

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

2.2.4.5 `weak_ptr` observers [tr.util.smartptr.weak.obs]

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

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

2 *Throws:* nothing.

3 [*Note:* `use_count()` is not necessarily efficient. Use only for debugging and testing purposes, not for production code. *—end note*]

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

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

5 *Throws:* nothing.

6 [*Note:* `expired()` may be faster than `use_count()`. *—end note*]

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

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

8 *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);
```

1 *Returns:* an unspecified value such that

- `operator<` is a strict weak ordering as described in [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* or are both empty.

2 *Throws:* nothing.

3 [*Note:* Allows `weak_ptr` objects to be used as keys in associative containers. —*end note*]

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)
```

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

2 *Throws:* nothing.

2.2.5 Class template `enable_shared_from_this` [tr.util.smartptr.enab]

1 A class `T` can inherit from `enable_shared_from_this<T>`, to inherit the `shared_from_this` member functions that obtain a *shared_ptr* instance pointing to `*this`.

2 [*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 std {
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;
  };
} // namespace tr1
} // namespace std
```

3 The template parameter `T` of `enable_shared_from_this` may be an incomplete type.

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

4 *Effects:* Constructs an `enable_shared_from_this<T>` object.

5 *Throws:* nothing.

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

6 *Returns:* `*this`.

7 *Throws:* nothing.

```
~enable_shared_from_this();
```

8 *Effects:* Destroys `*this`.

9 *Throws:* nothing.

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

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

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

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

13 [*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); }
};
```

14 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*]

3 Function objects [tr.func]

1 This clause defines components for creating and manipulating function objects, and for higher-order programming.

2 The following subclauses describe class template result_of, function template mem_fn, function object binders, and the polymorphic function wrapper function, as summarized in Table 2.

Table 2: Function object library summary

Subclause	Header(s)
3.4 result_of	<functional>
3.5 mem_fn	<functional>
3.6 Function object binders	<functional>
3.7 Function object wrappers	<functional>

3.1 Definitions [tr.func.def]

1 The following definitions shall apply to this clause:

2 A *call signature* is the name of a return type followed by a parenthesized comma-separated list of zero or more argument types.

3 A *call wrapper* is an object of a call wrapper type.

4 A *call wrapper type* is a type that holds a callable object and supports a call operation that forwards to that object.

5 A *callable object* is an object of a callable type.

6 A *callable type* is a pointer to function, a pointer to member function, a pointer to member data, or a class type whose objects can appear immediately to the left of a function call operator.

7 A *target object* is the callable object held by a call wrapper object.

3.2 Additions to <functional> synopsis [tr.func.syn]

```
namespace std {
namespace tr1 {
  // [3.4] class template result_of
  template <class FunctionCallTypes> class result_of;

  // [3.5] function template mem_fn
  template<class R, class T> unspecified mem_fn(R T::* pm);
```

```
// [3.6] Function object binders
template<class T> struct is_bind_expression;
template<class T> struct is_placeholder;

template<class F, class T1, ..., class Tn>          unspecified bind(F f, T1 t1, ..., Tn tn);
template<class R, class F, class T1, ..., class Tn> unspecified bind(F f, T1 t1, ..., Tn tn);

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

// [3.7] polymorphic function wrappers
class bad_function_call;

template<class Function> class function;

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

template<class Function1, class Function2>
  void operator==(const function<Function1>&, const function<Function2>&);

template<class Function1, class Function2>
  void operator!=(const function<Function1>&, const function<Function2>&);

template <class Function>
  bool operator==(const function<Function>&, unspecified-null-pointer-type);

template <class Function>
  bool operator==(unspecified-null-pointer-type , const function<Function>&);

template <class Function>
  bool operator!=(const function<Function>&, unspecified-null-pointer-type);

template <class Function>
  bool operator!=(unspecified-null-pointer-type , const function<Function>&);
} // namespace tr1
} // namespace std
```

3.3 Requirements [tr.func.require]

1 Define *INVOKE*`(f, t1, t2, ..., tN)` as follows:

— `(t1.*f)(t2, ..., tN)` when `f` is a pointer to a member function of a class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`;

— `((*t1).*f)(t2, ..., tN)` when `f` is a pointer to a member function of a class `T` and `t1` is not one of the types described in the previous item;

— `t1.*f` when `f` is a pointer to member data of a class `T` and `t1` is an object of type `T` or a reference to an object of type `T` or a reference to an object of a type derived from `T`;

— `(*t1).*f` when `f` is a pointer to member data of a class `T` and `t1` is not one of the types described in the previous item;

— `f(t1, t2, ..., tN)` in all other cases.

2 Define *INVOKE*`(f, t1, t2, ..., tN, R)` as *INVOKE*`(f, t1, t2, ..., tN)` implicitly converted to `R`.

3 If a call wrapper ([3.1]) has a *weak result type* the type of its member type `result_type` is based on the type `T` of the wrapper's target object ([3.1]):

— if `T` is a function, reference to function, or pointer to function type, `result_type` shall be a synonym for the return type of `T`;

— if `T` is a pointer to member function, `result_type` shall be a synonym for the return type of `T`;

— if `T` is a class type with a member type `result_type`, then `result_type` shall be a synonym for `T::result_type`;

— otherwise `result_type` shall not be defined.

4 Every call wrapper [3.1] shall be CopyConstructible. A *simple call wrapper* is a call wrapper that is Assignable and whose copy constructor and assignment operator do not throw exceptions. A *forwarding call wrapper* is a call wrapper that can be called with an argument list `t1, t2, ..., tN` where each `ti` is an lvalue. The effect of calling a forwarding call wrapper with one or more arguments that are rvalues is implementation defined. [*Note:* in a typical implementation forwarding call wrappers have overloaded function call operators of the form

```
template<class T1, class T2, ..., class TN>
R  operator()(T1& t1, T2& t2, ..., TN& tN) cv-qual;
```

—end note]

3.4 Function return types [tr.func.ret]

```
namespace std {
namespace tr1 {
  template <class FunctionCallTypes> // F(T1, T2, ..., TN)
  class result_of {
  public :
    // types
    typedef unspecified type;
  };
} // namespace tr1
} // namespace std
```

1 Given an rvalue `f` of type `F` and values `t1, t2, ..., tN` of types `T1, T2, ..., TN`, respectively, the `type` member is the result type of the expression `f(t1, t2, ...,tN)`. The values `ti` are lvalues when the corresponding type `Ti` is a reference type, and rvalues otherwise.

2 The implementation may determine the `type` member 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*]

3 If `F` is not a function object defined by the standard library, and if either the implementation cannot determine the type of the expression `f(t1, t2, ..., tN)` or if the expression is ill-formed, the implementation shall use the following process to determine the `type` member:

1. If `F` is a function pointer or function reference type, `type` is the return type of the function type.
2. If `F` is a member function pointer type, `type` is the return type of the member function type.
3. If `F` is a possibly *cv*-qualified class type with a member type `result_type`, `type` is `typename F::result_type`.
4. If `F` is a possibly *cv*-qualified class type with no member named `result_type` or if `typename F::result_type` is not a type:
   (a) If `N=0` (no arguments), `type` is `void`.
   (b) If `N>0`, `type` is `typename F::template result<F(T1, T2,..., TN)>::type`.
5. Otherwise, the program is ill-formed.

3.5 Function template `mem_fn` [tr.func.memfn]

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

1 *Returns:* A simple call wrapper ([3.1]) `f` such that the expression `f(t, a2, ..., aN)` is equivalent to *INVOKE*`(pm, t, a2, ..., aN)` ([3.3]). `f` shall have a nested type `result_type` that is a synonym for the return type of `pm` when `pm` is a pointer to member function.

2 The simple call wrapper shall be derived from `std::unary_function<`*cv* `T*,` *Ret*`>` when `pm` is a pointer to member function with cv-qualifier *cv* and taking no arguments, where *Ret* is `pm`'s return type.

3 The simple call wrapper shall be derived from `std::binary_function<`*cv* `T*, T1,` *Ret*`>` when `pm` is a pointer to member function with cv-qualifier *cv* and taking one argument of type `T1`, where *Ret* is `pm`'s return type.

4 *Throws:* Nothing.

5 *Notes:* Implementations may implement mem_fn as a set of overloaded function templates.

3.6 Function object binders [tr.func.bind]

1 3.6 describes a uniform mechanism for binding arguments of function objects.

3.6.1 Class template `is_bind_expression` [tr.func.bind.isbind]

```
namespace std {
namespace tr1 {
  template<class T> struct is_bind_expression {
    static const bool value = see below;
  };
} // namespace tr1
} // namespace std
```

1 `is_bind_expression` can be used to detect function objects generated by `bind`. `bind` uses `is_bind_expression` to detect subexpressions. Users may specialize this template to indicate that a type should be treated as a subexpression in a `bind` call.

`static const bool value;`

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

3.6.2 Class template `is_placeholder` [tr.func.bind.isplace]

```
namespace std {
namespace tr1 {
  template<class T> struct is_placeholder {
    static const int value = see below;
  };
} // namespace tr1
} // namespace std
```

1 `is_placeholder` can be used to detect the standard placeholders `_1`, `_2`, and so on. `bind` uses `is_placeholder` to detect placeholders. Users may specialize this template to indicate a placeholder type.

`static const int value;`

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

3.6.3 Function template `bind` [tr.func.bind.bind]

```
template<class F, class T1, class T2, ...., class TN>
  unspecified bind(F f, T1 t1, T2 t2, ..., TN tN);
```

1 *Requires:* `F` and `Ti` shall be CopyConstructible. `INVOKE (f, w1, w2, ..., wN)` ([3.3]) shall be a valid expression for some values *w1, w2, ..., wN*.

2 *Returns:* A forwarding call wrapper `g` with a weak result type ([3.3]). The effect of `g(u1, u2, ..., uM)` shall be `INVOKE(f, v1, v2, ..., vN, result_of<F cv (V1, V2, ..., VN)>::type)`, where *cv* represents the *cv*-qualifiers of `g` and the values and types of the bound arguments `v1, v2, ..., vN` are determined as specified below.

```
template<class R, class F, class T1, class T2, ...., class TN>
  unspecified bind(F f, T1 t1, T2 t2, ..., TN tN);
```

3 *Requires:* F and Ti shall be CopyConstructible. *INVOKE*(f, w1, w2, ..., wN) shall be a valid expression for some values w1, w2, ..., wN.

4 *Returns:* A forwarding call wrapper g with a nested type result_type defined as a synonym for R. The effect of g(u1, u2, ..., uM) shall be *INVOKE*(f, v1, v2, ..., vN, R), where the values and types of the bound arguments v1, v2, ..., vN are determined as specified below.

5 The values of the *bound arguments* v1, v2, ..., vN and their corresponding types V1, V2, ..., VN depend on the type of the corresponding argument ti of type Ti in the call to bind and the *cv*-qualifiers *cv* of the call wrapper g as follows:

— if ti is of type reference_wrapper<T> the argument is ti.get() and its type Vi is T&;

— if the value of std::tr1::is_bind_expression<Ti>::value is true the argument is ti(u1, u2, ..., uM) and its type Vi is result_of<Ti *cv* (U1&, U2&, ..., UM&)>::type;

— if the value j of std::tr1::is_placeholder<Ti>::value is not zero the argument is uj and its type Vi is Uj&;

— otherwise the value is ti and its type Vi is Ti *cv* &.

3.6.4 Placeholders [tr.func.bind.place]

```
namespace std {
namespace tr1 {
  namespace placeholders {
    // M is the implementation-defined number of placeholders
    extern unspecified _1;
    extern unspecified _2;
                .
                .
                .
    extern unspecified _M;
  }
} // namespace tr1
} // namespace std
```

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

3.7 Polymorphic function wrappers [tr.func.wrap]

1 3.7 describes a polymorphic wrapper class that encapsulates arbitrary function objects.

3.7.1 Class bad_function_call [tr.func.wrap.badcall]

1 An exception of type bad_function_call is thrown by function::operator() ([3.7.2.4]) when the function wrapper object has no target.

```
namespace std {
namespace tr1 {
```

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

3.7.1.1 `bad_function_call` constructor [tr.func.wrap.badcall.const]

`bad_function_call();`

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

3.7.2 Class template `function` [tr.func.wrap.func]

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

      // [3.7.2.1] construct/copy/destroy
      explicit function();
      function(unspecified-null-pointer-type);
      function(const function&);
      template<class F> function(F);

      function& operator=(const function&);
      function& operator=(unspecified-null-pointer-type);
      template<class F> function& operator=(F);
      template<class F> function& operator=(reference_wrapper<F>);

      ~function();

      // [3.7.2.2] function modifiers
      void swap(function&);

      // [3.7.2.3] function capacity
      operator unspecified-bool-type() const;

      // [3.7.2.4] function invocation
      R operator()(T1, T2, ..., TN) const;
```

```
    // [3.7.2.5] function target access
    const std::type_info& target_type() const;
    template <typename T>       T* target();
    template <typename T> const T* target() const;

  private:
    // [3.7.2.6] undefined operators
    template<class Function2> void operator==(const function<Function2>&);
    template<class Function2> void operator!=(const function<Function2>&);
  };

  // [3.7.2.7] Null pointer comparisons
  template <class Function>
    bool operator==(const function<Function>&, unspecified-null-pointer-type);

  template <class Function>
    bool operator==(unspecified-null-pointer-type , const function<Function>&);

  template <class Function>
    bool operator!=(const function<Function>&, unspecified-null-pointer-type);

  template <class Function>
    bool operator!=(unspecified-null-pointer-type , const function<Function>&);

  // [3.7.2.8] specialized algorithms
  template<class Function> void swap(function<Function>&, function<Function>&);
} // namespace tr1
} // namespace std
```

1 The `function` class template provides polymorphic wrappers that generalize the notion of a function pointer. Wrappers can store, copy, and call arbitrary callable objects ([3.1]), given a call signature ([3.1]), allowing functions to be first-class objects.

2 A function object `f` of type `F` is Callable for argument types `T1, T2, ..., TN` and a return type `R`, if, given lvalues `t1, t2, ..., tN` of types `T1, T2, ..., TN`, respectively, *INVOKE*`(f, t1, t2, ..., tN)` is well-formed ([3.3]) and, if `R` is not `void`, convertible to `R`.

3 The `function` class template is a call wrapper ([3.1]) whose call signature ([3.1]) is `R(T1, T2, ..., TN)`.

3.7.2.1 `function` construct/copy/destroy [tr.func.wrap.func.con]

`explicit function();`

1 *Postconditions:* `!*this`.

2 *Throws:* nothing.

`function(unspecified-null-pointer-type);`

3 *Postconditions:* `!*this`.

4 *Throws:* nothing.

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

5 *Postconditions:* `!*this` if `!f`; otherwise, `*this` targets a copy of `f`.

6 *Throws:* shall not throw exceptions if `f`'s target 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<class F> function(F f);
```

7 *Requires:* `f` shall be callable for argument types `T1`, `T2`, ..., `TN` and return type `R`.

8 *Postconditions:* `!*this` 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`

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

10 *Throws:* shall not throw exceptions when `f` is a function pointer or a `reference_wrapper<T>` for some `T`. Otherwise, may throw `bad_alloc` or any exception thrown by `F`'s copy constructor.

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

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

12 *Returns:* `*this`

```
function& operator=(unspecified-null-pointer-type);
```

13 *Effects:* If `(bool) (*this)`, deallocates current target.

14 *Postconditions:* `!(*this)`.

15 *Returns:* `*this`

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

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

17 *Returns:* `*this`

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

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

19 *Returns:* `*this`

20 *Throws:* nothing.

```
~function();
```

21 *Effects:* if `*this != NULL`, destroys the target of `this`.

3.7.2.2 `function` modifiers [tr.func.wrap.func.mod]

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

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

2 *Throws:* nothing.

3.7.2.3 `function` capacity [tr.func.wrap.func.cap]

```
operator unspecified-bool-type() const
```

1 *Returns:* if `*this` has a target, 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`.

2 *Throws:* nothing.

3 [*Note:* 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. One possible implementation choice for this type is pointer-to-member. *—end note*]

3.7.2.4 `function` invocation [tr.func.wrap.func.inv]

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

1 *Effects:* `INVOKE(f, t1, t2, ..., tN, R)` ([3.3]), where `f` is the target object ([3.1]) of `*this`.

2 *Returns:* nothing, if `R` is `void`, otherwise the return value of `INVOKE(f, t1, t2, ..., tN, R)`.

3 *Throws:* `bad_function_call` if `!*this`; otherwise, any exception thrown by the wrapped function object.

3.7.2.5 function target access [tr.func.wrap.func.targ]

```
const std::type_info& target_type() const;
```

1 *Returns:* If `*this` has a target of type `T`, `typeid(T)`; otherwise, `typeid(void)`.

2 *Throws:* nothing.

```
template<typename T>       T* target();
template<typename T> const T* target() const;
```

3 *Requires:* `T` is a function object type that is Callable ([3.7.2]) for parameter types `T1, T2, ..., TN` and return type `R`.

4 *Returns:* If `type() == typeid(T)`, a pointer to the stored function target; otherwise a null pointer.

5 *Throws:* nothing.

3.7.2.6 undefined operators [tr.func.wrap.func.undef]

```
template<class Function2> void operator==(const function<Function2>&);
template<class Function2> void operator!=(const function<Function2>&);
```

1 These member functions shall be left undefined.

2 [*Note:* the boolean-like conversion opens a loophole whereby two `function` instances can be compared via `==` or `!=`. These undefined `void` operators close the loophole and ensure a compile-time error. *—end note*]

3.7.2.7 null pointer comparison operators [tr.func.wrap.func.nullptr]

```
template <class Function>
  bool operator==(const function<Function>& f, unspecified-null-pointer-type);

template <class Function>
  bool operator==(unspecified-null-pointer-type, const function<Function>& f);
```

1 *Returns:* `!f`.

2 *Throws:* nothing.

```
template <class Function>
  bool operator!=(const function<Function>& f, unspecified-null-pointer-type);

template <class Function>
  bool operator!=(unspecified-null-pointer-type, const function<Function>& f);
```

3 *Returns:* `(bool) f`.

4 *Throws:* nothing.

3.7.2.8 specialized algorithms [tr.func.wrap.func.alg]

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

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

4 Metaprogramming and type traits [tr.meta]

1 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.

2 The following subclauses describe type traits requirements, unary type traits, traits that describe relationships between types, and traits that perform transformations on types, as summarized in Table 3.

Table 3: Type traits library summary

Subclause	Header(s)
4.1 Requirements	
4.5 Unary type traits	<type_traits>
4.6 Relationships between types	<type_traits>
4.7 Transformations between types	<type_traits>

4.1 Requirements [tr.meta.rqmts]

A *UnaryTypeTrait* is a template that describes a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the property being described. It shall be *DefaultConstructible* and derived, directly or indirectly, from an instance of the template `integral_constant` (4.3), with the arguments to the template `integral_constant` determined by the requirements for the particular property being described.

A *BinaryTypeTrait* is a template that describes a relationship between two types. It shall be a class template that takes two template type arguments and, optionally, additional arguments that help define the relationship being described. It shall be *DefaultConstructible* and derived, directly or indirectly, from an instance of the template `integral_constant` (4.3), with the arguments to the template `integral_constant` determined by the requirements for the particular relationship being described.

A *TransformationTypeTrait* is a template that modifies a property of a type. It shall be a class template that takes one template type argument and, optionally, additional arguments that help define the modification. It shall define a nested type named `type` which shall be a synonym for the modified type.

4.2 Header <type_traits> synopsis [tr.meta.type.synop]

```
namespace std {
namespace tr1 {
  // [4.3] helper class:
```

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

// [4.5.1] 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;

// [4.5.2] 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;

// [4.5.3] 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 has_virtual_destructor;
template <class T> struct is_signed;
template <class T> struct is_unsigned;
template <class T> struct alignment_of;
template <class T> struct rank;
template <class T, unsigned I = 0> struct extent;

// [4.6] type relations:
template <class T, class U> struct is_same;
template <class Base, class Derived> struct is_base_of;
```

```
  template <class From, class To> struct is_convertible;

  // [4.7.1] 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;

  // [4.7.2] reference modifications:
  template <class T> struct remove_reference;
  template <class T> struct add_reference;

  // [4.7.3] array modifications:
  template <class T> struct remove_extent;
  template <class T> struct remove_all_extents;

  // [4.7.4] pointer modifications:
  template <class T> struct remove_pointer;
  template <class T> struct add_pointer;

  // [4.8] other transformations:
  template <std::size_t Len, std::size_t Align> struct aligned_storage;
} // namespace tr1
} // namespace std
```

4.3 Helper classes [tr.meta.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;
```

1 The class template `integral_constant` and its associated typedefs `true_type` and `false_type` are used as base classes to define the interface for various type traits.

4.4 General Requirements [tr.meta.requirements]

1 Tables 4, 5, 6, and 8 define type predicates. Each type predicate `pred<T>` shall be a *UnaryTypeTrait* (4.1), derived directly or indirectly from `true_type` if the corresponding condition is true, otherwise from `false_type`. Each type predicate `pred<T, U>` shall be a *BinaryTypeTrait* (4.1), derived directly or indirectly from `true_type` if the corresponding condition is true, otherwise from `false_type`.

2 Table 7 defines various type queries. Each type query shall be a *UnaryTypeTrait* (4.1), derived directly or indirectly from `integral_constant<std::size_t, value>`, where `value` is the value of the property being queried.

3 Tables 9, 10, 11, and 12 define type transformations. Each transformation shall be a *TransformationTrait* (4.1).

4 Table 13 defines a template that can be instantiated to define a type with a specific alignment and size.

4.5 Unary Type Traits [tr.meta.unary]

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

2 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.

4.5.1 Primary Type Categories [tr.meta.unary.cat]

1 The primary type categories correspond to the descriptions given in section [basic.types] of the C++ standard.

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

3 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.

4 The behavior of a program that adds specializations for any of the class templates defined in this clause is undefined.

Table 4: Primary Type Category Predicates

Template	Condition	Comments
`template <class T> struct is_void;`	`T` is `void` or a *cv-qualified* `void`	
`template <class T> struct is_integral;`	`T` is an integral type ([basic.fundamental])	
`template <class T> struct is_floating_point;`	`T` is a floating point type ([basic.fundamental])	
`template <class T> struct is_array;`	`T` is an array type ([basic.compound])	Class template `array` ([6.2]) is *not* an array type.
`template <class T> struct is_pointer;`	`T` is a pointer type ([basic.compound])	Includes function pointers, but not pointers to members.
`template <class T> struct is_reference;`	`T` is a reference type ([basic.fundamental])	Includes references to functions.
`template <class T> struct is_member_object_pointer;`	`T` is a pointer to data member	
`template <class T> struct is_member_function_pointer;`	`T` is a pointer to member function	
`template <class T> struct is_enum;`	`T` is an enumeration type ([basic.compound])	
`template <class T> struct is_union;`	`T` is a union type ([basic.compound])	
`template <class T> struct is_class;`	`T` is a class type but not a union type ([basic.compound])	
`template <class T> struct is_function;`	`T` is a function type ([basic.compound])	

4.5.2 Composite type traits **[tr.meta.unary.comp]**

1 These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in section [basic.types].

2 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.

3 The behavior of a program that adds specializations for any of the class templates defined in this clause is undefined.

Table 5: Composite Type Category Predicates

Template	Condition	Comments
`template <class T> struct is_arithmetic;`	`T` is an arithmetic type ([basic.fundamental])	
`template <class T> struct is_fundamental;`	`T` is a fundamental type ([basic.fundamental])	
`template <class T> struct is_object;`	`T` is an object type ([basic.types])	
`template <class T> struct is_scalar;`	`T` is a scalar type ([basic.types])	
`template <class T> struct is_compound;`	`T` is a compound type ([basic.compound])	
`template <class T> struct is_member_pointer;`	`T` is a pointer to a member or member function	

4.5.3 Type properties **[tr.meta.unary.prop]**

1 These templates provide access to some of the more important properties of types.

2 It is unspecified whether the library defines any full or partial specialisations of any of these templates. A program may 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.

Table 6: Type Property Predicates

Template	Condition	Preconditions
`template <class T> struct is_const;`	`T` is const-qualified ([basic.qualifier])	
`template <class T> struct is_volatile;`	`T` is volatile-qualified ([basic.qualifier])	
`template <class T> struct is_pod;`	`T` is a POD type ([basic.types])	`T` shall be a complete type.
`template <class T> struct is_empty;`	`T` is an empty class ([class])	`T` shall be a complete type.
`template <class T> struct is_polymorphic;`	`T` is a polymorphic class ([class.virtual])	`T` shall be a complete type.

`template <class T>` `struct is_abstract;`	`T` is an abstract class ([class.abstract])	`T` shall be a complete type.
`template <class T>` `struct has_trivial_constructor;`	The default constructor for `T` is trivial ([class.ctor])	`T` shall be a complete type.
`template <class T>` `struct has_trivial_copy;`	The copy constructor for `T` is trivial ([class.copy])	`T` shall be a complete type.
`template <class T>` `struct has_trivial_assign;`	The assignment operator for `T` is trivial ([class.copy])	`T` shall be a complete type.
`template <class T>` `struct has_trivial_destructor;`	The destructor for `T` is trivial ([class.dtor])	`T` shall be a complete type.
`template <class T>` `struct has_nothrow_constructor;`	The default constructor for `T` has an empty exception specification or can otherwise be deduced never to throw an exception	`T` shall be a complete type.
`template <class T>` `struct has_nothrow_copy;`	The copy constructor for `T` has an empty exception specification or can otherwise be deduced never to throw an exception	`T` shall be a complete type.
`template <class T>` `struct has_nothrow_assign;`	The assignment operator for `T` has an empty exception specification or can otherwise be deduced never to throw an exception	`T` shall be a complete type.
`template <class T>` `struct has_virtual_destructor;`	`T` has a virtual destructor ([class.dtor])	`T` shall be a complete type.
`template <class T>` `struct is_signed;`	`T` is a signed integral type ([basic.fundamental])	
`template <class T>` `struct is_unsigned;`	`T` is an unsigned integral type ([basic.fundamental])	

Table 7: Type Property Queries

Template	**value**
`template <class T>` `struct alignment_of;`	An 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 ([basic.types]). *Precondition:* `T` shall be a complete type.
`template <class T>` `struct rank;`	An integer value representing the rank of objects of type `T` ([dcl.array]). [*Note*: The term “rank” here is used to describe the number of dimensions of an array type. *—end note*]

template <class T, unsigned I = 0> struct extent;	An integer value representing the extent (dimension) of the I'th bound of objects of type T (8.3.4). If the type T is not an array type, has rank of less than I, or if I == 0 and T is of type "array of unknown bound of U," then value shall evaluate to zero; otherwise value shall evaluate to the number of elements in the I'th array bound of T. [*Note*: The term "extent" here is used to describe the number of elements in an array type —*end note*]

3 [*Example*:

```
// the following assertions hold:
assert(rank<int>::value == 0);
assert(rank<int[2]>::value == 1);
assert(rank<int[][4]>::value == 2);
```

—*end example*]

4 [*Example*:

```
// the following assertions hold:
assert(extent<int>::value == 0);
assert(extent<int[2]>::value == 2);
assert(extent<int[2][4]>::value == 2);
assert(extent<int[][4]>::value == 0);
assert((extent<int, 1>::value) == 0);
assert((extent<int[2], 1>::value) == 0);
assert((extent<int[2][4], 1>::value) == 4);
assert((extent<int[][4], 1>::value) == 4);
```

—*end example*]

4.6 Relationships between types [tr.meta.rel]

Table 8: Type Relationship Predicates

Template	Condition	Comments
template <class T, class U> struct is_same;	T and U name the same type	
template <class Base, class Derived> struct is_base_of;	Base is a base class of Derived ([class.derived]) or Base and Derived name the same type	*Preconditions:* Base and Derived shall be complete types.

`template <class From, class To>` `struct is_convertible;`	An imaginary lvalue of type `From` is implicitly convertible to type `To` ([conv])	Special conversions involving string-literals and null-pointer constants are not considered ([conv.array], [conv.ptr], and [conv.mem]). No function-parameter adjustments ([dcl.fct]) 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 the condition is false. See below.

1 The expression `is_convertible<From,To>::value` is ill-formed if:

- Type `From` is a void or incomplete type ([basic.types]).
- Type `To` is an incomplete, void or abstract type ([basic.types]).
- The conversion is ambiguous. An example of an ambiguous conversion is a type `From` that has multiple base classes of type `To` ([class.member.lookup]).
- Type `To` is of class type and the conversion would invoke a non-public constructor of `To` ([class.access] and [class.conv.ctor]).
- Type `From` is of class type and the conversion would invoke a non-public conversion operator of `From` ([class.access] and [class.conv.fct]).

4.7 Transformations between types [tr.meta.trans]

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

2 Each of the templates in this header shall be a *TransformationTrait* (4.1).

4.7.1 Const-volatile modifications [tr.meta.trans.cv]

Table 9: Const-volatile modifications

Template	Comments
`template <class T>` `struct remove_const;`	The member typedef `type` shall be the same as `T` except that any top level const-qualifier has been removed. [*Example:* `remove_const<const volatile int>::type` evaluates to `volatile int`, whereas `remove_const<const int*>` is `const int*`. *—end example*]

`template <class T> struct remove_volatile;`	The member typedef `type` shall be the same as `T` except that any top level volatile-qualifier has been removed. [*Example:* `remove_volatile<const volatile int>::type` evaluates to `const int`, whereas `remove_volatile<volatile int*>` is `volatile int*`. *—end example*]
`template <class T> struct remove_cv;`	The member typedef `type` shall be the same as `T` except that any top level cv-qualifier has been removed. [*Example:* `remove_cv<const volatile int>::type` evaluates to `int`, whereas `remove_cv<const volatile int*>` is `const volatile int*`. *—end example*]
`template <class T> struct add_const;`	If `T` is a reference, function, or top level const-qualified type, then `type` shall be the same type as `T`, otherwise `T const`.
`template <class T> struct add_volatile;`	If `T` is a reference, function, or top level volatile-qualified type, then `type` shall be the same type as `T`, otherwise `T volatile`.
`template <class T> struct add_cv;`	The member typedef `type` shall be the same type as `add_const<typename add_volatile<T>::type>::type`.

4.7.2 Reference modifications [tr.meta.trans.ref]

Table 10: Reference modifications

Template	Comments
`template <class T> struct remove_reference;`	The member typedef `type` shall be the same as `T`, except any reference qualifier has been removed.
`template <class T> struct add_reference;`	If `T` is a reference type, then the member typedef `type` shall be `T`, otherwise `T&`.

4.7.3 Array modifications [tr.meta.trans.arr]

Table 11: Array modifications

Template	Comments
`template <class T> struct remove_extent;`	If `T` is "array of `U`", the member typedef `type` shall be `U`, otherwise `T`. For multidimensional arrays, only the first array dimension is removed. For a type "array of `const U`", the resulting type is `const U`.
`template <class T> struct remove_all_extents;`	If `T` is "multi-dimensional array of `U`", the resulting member typedef `type` is `U`, otherwise `T`.

1 [*Example*

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

—*end example*]

2 [*Example*

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

—*end example*]

4.7.4 Pointer modifications [tr.meta.trans.ptr]

Table 12: Pointer modifications

Template	Comments
`template <class T> struct remove_pointer;`	The member typedef `type` shall be 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;`	The member typedef `type` shall be the same as `remove_reference<T>::type*` if `T` is a reference type, otherwise `T*`.

4.8 Other transformations [tr.meta.trans.other]

Table 13: Other transformations

Template	Condition	Comments
`template <std::size_t Len, std::size_t Align> struct aligned_storage;`	`Len` is nonzero. `Align` is equal to `alignment_of<T>::value` for some type `T`.	The member typedef `type` shall be a POD type suitable for use as uninitialized storage for any object whose size is at most *Len* and whose alignment is a divisor of *Align*.

1 [*Note:* a typical implementation would define `type` as:

```
union type
{
  unsigned char __data[Len];
  Aligner __align;
};
```

where *Aligner* is the smallest POD type for which `alignment_of<Aligner>::value` is `Align`. —*end note*]

4.9 Implementation requirements [tr.meta.req]

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

2 [*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*]

3 If the implementation cannot differentiate between class and union types, then the class templates `is_class` and `is_union` shall be defined as follows:

```
template <class T> struct is_class {};
template <class T> struct is_union {};
```

4 If the implementation cannot detect polymorphic types, then the class template `is_polymorphic` shall be defined as follows:

```
template <class T> struct is_polymorphic {};
```

5 If the implementation cannot detect abstract types, then the class template `is_abstract` shall be defined as follows:

```
template <class T> struct is_abstract {};
```

6 If the implementation cannot determine whether a type `T` has a virtual destructor, *e.g.* a pure library implementation with no compiler support, then `has_virtual_destructor<T>` shall be derived, directly or indirectly, from `false_type` (4.1).

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

8 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_extent<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)
```

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

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

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

11 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.

5 Numerical facilities [tr.num]

1 This clause descibes components that C++ programs may use to perform numerical and seminumerical operations.

2 The following subclauses describe random number generators and mathematical special functions, as summarized in Table 14.

Table 14: Numerical library summary

Subclause	Header(s)
5.1 Random number generation	<random>
5.2 Mathematical special functions	<cmath> <math.h>

5.1 Random number generation [tr.rand]

1 This subclause defines a facility for generating random numbers.

5.1.1 Requirements [tr.rand.req]

1 In table 15, `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 15: Uniform random number generator requirements

expression	return type	pre/post-condition	complexity
`X::result_type`	`T`	`T` is an arithmetic type [basic.fundamental]	compile-time
`u()`	`T`	—	amortized constant
`v.min()`	`T`	Returns a value that is less than or equal to all values potentially returned by `operator()`. The return value of this function shall not change during the lifetime of `v`.	constant

expression	return type	pre/post-condition	complexity
`v.max()`	`T`	If `std::numeric_limits<T>::is_integer`, returns a value that is greater than or equal to all values potentially returned by `operator()`, otherwise, returns a value that is strictly greater than all values potentially returned by `operator()`. In any case, the return value of this function shall not change during the lifetime of `v`.	constant

2 In table 16, `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`, `s` is a value of integral type, `g` is an lvalue of a zero-argument function object returning values of unsigned integral type, `x` and `y` are (possibly const) values of `X`, `os` is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`, and `is` is an lvalue of the type of some class template specialization `basic_istream<charT, traits>`, where `charT` and `traits` are constrained according to [lib.strings] and [lib.input.output].

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

Table 16: Pseudo-random number engine requirements (in addition to uniform random number generator, `CopyConstructible`, and `Assignable`)

expression	return type	pre/post-condition	complexity
`X()`	—	creates an engine with the same initial state as all other default-constructed engines of type `X` in the program.	$\mathcal{O}$(size of state)
`X(s)`	—	creates an engine with the initial internal state determined by `s`	$\mathcal{O}$(size of state)
`X(g)`	—	creates an engine with the initial internal state given by the results of successive invocations of `g`. Throws what and when `g` throws.	$\mathcal{O}$(size of state)
`u.seed()`	`void`	post: `u == X()`	$\mathcal{O}$(size of state)
`u.seed(s)`	`void`	sets the internal state of `u` so that `u == X(s)`.	same as `X(s)`

expression	return type	pre/post-condition	complexity
`u.seed(g)`	`void`	post: The internal state of `u` is the same as if `u` had been newly constructed by `X u(g)`. If an invocation of `g` throws, that exception is rethrown, and further use of `u` (except destruction) is undefined until a `seed` member function has been executed without throwing an exception.	same as `X(g)`
`u()`	`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
`x == y`	`bool`	Given the current state `x(i)` of `x` and the current state `y(j)` of `y`, returns true if `x(i+k)` is equal to `y(j+k)` for all integer `k >= 0`, false otherwise.	$\mathcal{O}$(size of state)
`x != y`	`bool`	`!(x == y)`	$\mathcal{O}$(size of state)
`os << x`	reference to the type of `os`	writes the textual representation of the state `x(i)` of `x` to `os`, with `os.`*fmtflags* set to `ios_base::dec|ios_-base::fixed|ios_base::left` and the fill character set to the space character. In the output, adjacent numbers are separated by one or more space characters. post: The `os.`*fmtflags* and fill character are unchanged.	$\mathcal{O}$(size of state)
`is >> v`	reference to the type of `is`	sets the state `v(i)` of `v` as determined by reading its textual representation from `is`. pre: The textual representation was previously written using an `os` whose imbued locale and whose type's template specialization arguments `charT` and `traits` were the same as those of `is`. post: The `is`.*fmtflags* are unchanged.	$\mathcal{O}$(size of state)

4 Additional requirements:

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

— the constructor `template<class Gen> X(Gen& g)` shall have the same effect as `X(static_cast<Gen>(g))` if `Gen` is a fundamental type.

— The member function of the form `template<class Gen> void seed(Gen& g)` shall have the same effect as `X(static_cast<Gen>(g))` if `Gen` is a fundamental type.

[*Note*: The casts make `g` an rvalue, unsuitable for binding to a reference. *—end note*]

5 If a textual representation was written by `os << x` and that representation was read by `is >> v`, then `x == v`, provided that no intervening invocations of `x or v` have occurred.

6 In table 17, `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`, `e` is an lvalue of an arbitrary type that meets the requirements of a uniform random number generator, returning values of type `U`, `os` is an lvalue of the type of some class template specialization `basic_ostream<charT, traits>`, and is an lvalue of the type of some class template specialization `basic_istream<charT, traits>`, where `charT` and `traitsw` are constrained according to [lib.strings] and [lib.input.output].

Table 17: Random distribution requirements (in addition to `CopyConstructible`, and `Assignable`)

expression	return type	pre/post-condition	complexity
`X::input_type`	`U`	—	compile-time
`u.reset()`	`void`	subsequent uses of `u` do not depend on values produced by `e` prior to invoking `reset`.	constant
`u(e)`	`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`
`os << x`	reference to the type of `os`	writes a textual representation for the parameters and additional internal data of the distribution `x` to `os`. post: The `os.`*fmtflags* and fill character are unchanged.	$\mathcal{O}$(size of state)
`is >> u`	reference to the type os `is`	restores the parameters and additional internal data of the distribution `u`. pre: `is` provides a textual representation that was previously written using an `os` whose imbued locale and whose type's template specialization arguments `charT` and `traits` were the same than those of `is`. post: The `is.`*fmtflags* are unchanged.	$\mathcal{O}$(size of state)

7 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)`.

8 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.

9 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`.

10 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.

11 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 std {
namespace tr1 {
  // [5.1.3] Class template variate_generator
  template<class UniformRandomNumberGenerator, class Distribution>
    class variate_generator;

  // [5.1.4.1] Class template linear_congruential
  template<class IntType, IntType a, IntType c, IntType m>
    class linear_congruential;

  // [5.1.4.2] Class template mersenne_twister
  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;

  // [5.1.4.3] Class template substract_with_carry
  template<class IntType, IntType m, int s, int r>
    class subtract_with_carry;

  // [5.1.4.4] Class template substract_with_carry_01
  template<class RealType, int w, int s, int r>
    class subtract_with_carry_01;

  // [5.1.4.5] Class template discard_block
  template<class UniformRandomNumberGenerator, int p, int r>
    class discard_block;

  // [5.1.4.6] Class template xor_combine
  template<class UniformRandomNumberGenerator1, int s1,
           class UniformRandomNumberGenerator2, int s2>
    class xor_combine;
```

```
    // [5.1.6] Class random_device
    class random_device;

    // [5.1.7.1] Class template uniform_int
    template<class IntType = int>
      class uniform_int;

    // [5.1.7.2] Class bernoulli_distribution
    class bernoulli_distribution;

    // [5.1.7.3] Class template geometric_distribution
    template<class IntType = int, class RealType = double>
      class geometric_distribution;

    // [5.1.7.4] Class template poisson_distribution
    template<class IntType = int, class RealType = double>
      class poisson_distribution;

    // [5.1.7.5] Class template binomial_distribution
    template<class IntType = int, class RealType = double>
      class binomial_distribution;

    // [5.1.7.6] Class template uniform_real
    template<class RealType = double>
      class uniform_real;

    // [5.1.7.7] Class template exponential_distribution
    template<class RealType = double>
      class exponential_distribution;

    // [5.1.7.8] Class template normal_distribution
    template<class RealType = double>
      class normal_distribution;

    // [5.1.7.9] Class template gamma_distribution
    template<class RealType = double>
      class gamma_distribution;
  } // namespace tr1
  } // namespace std
```

5.1.3 Class template `variate_generator` [tr.rand.var]

1 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 see below 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;
  };
```

2 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`*.

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

4 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)`

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

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

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

`result_type operator()()`

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

9 *Complexity:* Amortized constant.

10 *Notes:* The sequence of numbers produced by the uniform random number generator `e`, $s_{\mathtt{e}}$, is obtained from the sequence of numbers produced by the associated uniform random number generator `eng`, $s_{\mathtt{eng}}$, 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 $s_{\mathtt{e}}$ is identical to $s_{\mathtt{eng}}$. Otherwise, if the values for both types are `false`, then the numbers in $s_{\mathtt{eng}}$ are divided by `engine().max() - engine().min()` to obtain the numbers in $s_{\mathtt{e}}$. 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 $s_{\mathtt{eng}}$ are divided by `engine().max() - engine().min() + 1` to obtain the numbers in $s_{\mathtt{e}}$. Otherwise, the mapping from $s_{\mathtt{eng}}$ to

$s_{\mathtt{e}}$ 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)`

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

`engine_value_type& engine()`

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

`const engine_value_type& engine() const`

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

`distribution_type& distribution()`

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

`const distribution_type& distribution() const`

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

`result_type min() const`

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

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

`result_type max() const`

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

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

5.1.4 Random number engine class templates [tr.rand.eng]

1 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 a zero-argument function object, throws an exception.

2 The class templates specified in this section satisfy all the requirements of a pseudo-random number engine (given in table 16), 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.

3 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]

1 A `linear_congruential` engine produces random numbers using a linear function x(i+1) := (a * x(i) + c) mod 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(unsigned long x0 = 1);
  template<class Gen> linear_congruential(Gen& g);
  void seed(unsigned long x0 = 1);
  template<class Gen> void seed(Gen& g);
  result_type min() const;
  result_type max() const;
  result_type operator()();
};
}
```

2 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.

3 The size of the state x(i) is 1. The textual representation is the value of x(i).

`explicit linear_congruential(unsigned long x0 = 1)`

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

`void seed(unsigned long x0 = 1)`

5 *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 Gen> linear_congruential(Gen& g)`

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

7 *Complexity:* Exactly one invocation of `g`.

5.1.4.2 Class template `mersenne_twister` [tr.rand.eng.mers]

1 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) *bitand* um) | (x(i-(n-1)) *bitand* lm)

— If the lowest bit of the binary representation of y(i) is set, x(i) = x(i-(n-m)) *xor* (y(i) *rshift* 1) *xor* a; otherwise x(i) = x(i-(n-m)) *xor* (y(i) *rshift* 1).

— z1(i) = x(i) *xor* (x(i) *rshift* u)

— z2(i) = z1(i) *xor* ((z1(i) *lshift* s) *bitand* b)

— z3(i) = z2(i) *xor* ((z2(i) *lshift* t) *bitand* c)

— o(x(i)) = z3(i) *xor* (z3(i) *rshift* l)

```
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 Gen> mersenne_twister(Gen& g);
  void seed();
  void seed(unsigned long value);
  template<class Gen> void seed(Gen& g);
  result_type min() const;
  result_type max() const;
  result_type operator()();
};
```

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

3 The size of the state x(i) is `n`. The textual representation is the values of x(i-n), . . . , x(i-1), in that order.

```
mersenne_twister()
```

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

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

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

```
template<class Gen> mersenne_twister(Gen& g)
```

6 *Effects:* Given the values $z_0 \ldots z_{n-1}$ obtained by successive invocations of g, sets x(-n) ... x(-1) to z_0 mod $2^w \ldots z_{n-1}$ mod 2^w.

7 *Complexity:* Exactly `n` invocations of g.

```
void seed()
```

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

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

9 *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) ... x(-1) to lcg(1) ... lcg(n), respectively.

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

```
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)
```

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

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

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

5.1.4.3 Class template `subtract_with_carry` [tr.rand.eng.sub]

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

```
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 Gen> subtract_with_carry(Gen& g);
    void seed(unsigned long value = 19780503ul);
    template<class Gen> void seed(Gen& g);
    result_type min() const;
    result_type max() const;
    result_type operator()();
  };
```

2 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.

3 The size of the state is `r`. The textual representation is the values of x(i-r), ..., x(i-1), carry(i-1), in that order.

`subtract_with_carry()`

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

`explicit subtract_with_carry(unsigned long value)`

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

`template<class Gen> subtract_with_carry(Gen& g)`

6 *Effects:* With $n = (w+31)/32$ (rounded downward) and given the values $z_0 \ldots z_{n*r-1}$ obtained by successive invocations of g, sets x(-r) ... x(-1) to $(z_0 \cdot 2^(32) + \cdots + z_{n-1} \cdot 2^{32(n-1)}) \mathrm{mod} m \ldots (z_{(r-1)n} \cdot 2^{32} + \cdots + z_(r-1) \cdot 2^{32(n-1)}) \mathrm{mod} m$. If x(-1) == 0, sets carry(-1) = 1, else sets carry(-1) = 0.

7 *Complexity:* Exactly `r*n` invocations of g.

`void seed(unsigned long value = 19780503)`

8 *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 $lcg(0) =$ `value`, sets x(-r) ... x(-1) to lcg(1) mod m ... lcg(r) mod m, respectively. If x(-1) == 0, sets carry(-1) = 1, else sets carry(-1) = 0.

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

```
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)
```

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

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

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

5.1.4.4 Class template `subtract_with_carry_01` [tr.rand.eng.sub1]

1 A `subtract_with_carry_01` engine produces floating-point random numbers using `x(i) = (x(i-s) - x(i-r) - carry(i-1)) mod 1;` and setting `carry(i) =` 2^{-w} if `x(i-s) - x(i-r) - carry(i-1) < 0`, else `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 Gen> subtract_with_carry_01(Gen& g);
  void seed(unsigned long value = 19780503);
  template<class Gen> void seed(Gen& g);
  result_type min() const;
  result_type max() const;
  result_type operator()();
};
```

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

3 The size of the state is `r`. With $n = (w+31)/32$ (rounded downward) and integer numbers $z[k, j]$ such that $x(i-k) * 2^w = z[k,0] + z[k,1] * 2^{32} + z[k,n-1] * 2^{32}(n-1)$, the textual representation is the values of $z[r,0], \dots z[r,n-1], \dots z[1,0], \dots z[1,n-1], \text{carry}(i-1) * 2^w$, in that order. [*Note*: The algorithm ensures that only integer numbers representable in 32 bits are written. *—end note*]

`subtract_with_carry_01()`

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

`explicit subtract_with_carry_01(unsigned long value)`

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

`template<class Gen> subtract_with_carry_01(Gen& g)`

6 *Effects:* With $n = (w+31)/32$ (rounded downward) and given the values $z_0 \dots z_{n \cdot r-1}$ obtained by successive invocations of g, sets x(-r) ... x(-1) to $\left(z_0 \cdot 2^{32} + \cdots + z_{n-1} \cdot 2^{32(n-1)}\right) \cdot 2^{-w} \text{mod} 1 \dots (z_{(r-1)n} \cdot 2^{32} + \cdots + z_{r-1} \cdot 2^{32(n-1)}) \cdot 2^{-w} \text{mod} 1$. If x(-1) == 0, sets carry(-1) = 2^{-w}, else sets carry(-1) = 0.

7 *Complexity:* Exactly `r*n` invocations of g.

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

8 *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 $lcg(0) =$ `value`, sets x(-r) ... x(-1) to $(lcg(1) \cdot 2^{-w}) \bmod 1 \ldots (lcg(r) \cdot 2^{-w}) \bmod 1$, respectively. If $x(-1) == 0$, sets $\text{carry}(-1) = 2^{-w}$, else sets $\text{carry}(-1) = 0$.

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

```
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);
```

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

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

5.1.4.5 Class template `discard_block` [tr.rand.eng.disc]

1 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);
  explicit discard_block(unsigned long s);
  template<class Gen> discard_block(Gen& g);
  void seed();
  template<class Gen> void seed(Gen& g);
  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
};
```

2 The template parameter `UniformRandomNumberGenerator` shall denote a class that satisfies all the requirements of a uniform random number generator, given in table 15 in clause 5.1.1. $0 \le$ `r` $\le$ `p`. The size of the state is the size of *b* plus 1. The textual representation is the textual representation of *b* followed by the value of `n`.

`discard_block()`

3 *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)`

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

`explicit discard_block(unsigned long s)`

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

`template<class Gen> discard_block(Gen)`

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

`void seed()`

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

`const base_type& base() const`

8 *Returns: b*

`result_type operator()()`

9 *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()`*.

5.1.4.6 Class template `xor_combine` [tr.rand.eng.xor]

1 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);
    xor_combine(unsigned long s);
    template<class Gen> xor_combine(Gen& g);
    void seed();
    template<class Gen> void seed(Gen& g);
    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
  };
```

2 The template parameters `UniformRandomNumberGenerator1` and `UniformRandomNumberGenerator2` shall denote classes that satisfy all the requirements of a uniform random number generator, given in table 15 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 $0 \leq$ `s2`. The size of the state is the size of the state of *b1* plus the size of the state of *b2*. The textual representation is the textual representation of *b1* followed by the textual representation of *b2*.

3 The member `result_type` is defined to be either the type `UniformRandomNumberGenerator1 ::result_type` or the type `UniformRandomNumberGenerator2 ::result_type`, whichever one provides the most storage. (As defined in clause [basic.fundamental].)

`xor_combine()`

4 *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)`

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

`xor_combine(unsigned long s)`

6 *Effects:* Constructs a `xor_combine` engine. To construct the subobject *b1*, invokes the `b1(s)` constructor. Then, to construct the subobject *b2*, invokes the `b2(s+1)` constructor [*Note:* If both *b1* and *b2* are of the same type, both engines should not be initialized with the same seed. *—end note*]

`template<class Gen> xor_combine(Gen& g)`

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

`void seed()`

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

`const base1_type& base1() const`

9 *Returns: b1*

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

10 *Returns: b2*

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

11 *Returns:* `(b1() << s1) ^ (b2() << s2)`.

12 [*Note:* Two shift values are provided for simplicity of interface. When using this class template, however, it is advisable for at most one of these values to be nonzero. (If both `s1` and `s2` are nonzero then the low bits will always be zero.) It is also advisable for the unshifted engine to have the property that `max()` is $2^n - 1 -$ `min()` for some n, and for the shift applied to the other engine to be no greater than n.

13 An `xor_combine` that does not meet these conditions may have significantly worse uniformity properties than either of the component engines it is based on. *—end note*]

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;
```

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

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

3 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]

1 A `random_device` produces non-deterministic random numbers. It satisfies all the requirements of a uniform random number generator (given in table 15 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.

2 If implementation limitations prevent generating non-deterministic random numbers, the implementation may 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);`

3 *Effects:* Constructs a `random_device` non-deterministic random number engine. The semantics and default value of the `token` parameter are implementation-defined.[1]

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

`result_type min() const`

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

`result_type max() const`

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

`double entropy() const`

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

8 *Throws:* Nothing.

`result_type operator()()`

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

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

[1] The parameter is intended to allow an implementation to differentiate between different sources of randomness.

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

1 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.

2 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.

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

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

1 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.

2 A `uniform_int` random distribution satisfies all the requirements of a uniform random number generator (given in table 15 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)`

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

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

`result_type min() const`

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

`result_type max() const`

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

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

7 *Returns:* A uniform random number x in the range 0 <= 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]

1 A `bernoulli_distribution` random distribution produces `bool` values distributed with probabilities p(`true`) = p and p(`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)
```

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

3 *Effects:* Constructs a `bernoulli_distribution` object. `p` is the parameter of the distribution.

```
RealType p() const
```

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

5.1.7.3 Class template `geometric_distribution` [tr.rand.dist.geom]

1 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))`

2 *Requires:* $0 < \mathtt{p} < 1$

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

`RealType p() const`

4 *Returns:* The "*p*" parameter of the distribution.

5.1.7.4 Class template `poisson_distribution` [tr.rand.dist.pois]

1 A `poisson_distribution` random distribution produces integer values $i > 0$ with probability distribution

$$p(i) = \frac{mean^i}{i!} e^{-mean},$$

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))`

2 *Requires:* $mean > 0$

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

`RealType mean() const`

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

5.1.7.5 Class template `binomial_distribution` [tr.rand.dist.bin]

1 A `binomial_distribution` random distribution produces integer values $i > 0$ with

$$p(i) = \binom{n}{i} \cdot p^i \cdot (1-p)^{t-i},$$

where 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))`

2 *Requires:* $0 \le$ `p` ≤ 1 and `t` ≥ 0.

3 *Effects:* Constructs a `binomial_distribution` object; `t` and `p` are the parameters of the distribution.

`IntType t() const`

4 *Returns:* The “t” parameter of the distribution.

`RealType p() const`

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

5.1.7.6 Class template `uniform_real` [tr.rand.dist.runif]

1 A `uniform_real` random distribution produces floating-point random numbers `x` in the range `min` $\le x <$ `max`, with equal probability. `min` and `max` are the parameters of the distribution.

2 A `uniform_real` random distribution satisfies all the requirements of a uniform random number generator (given in table 15 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))`

3 *Requires:* `min` $\leq$ `max`.

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

`result_type min() const`

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

`result_type max() const`

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

5.1.7.7 Class template `exponential_distribution` [tr.rand.dist.exp]

1 An `exponential_distribution` random distribution produces random numbers $x > 0$ distributed with probability density function

$$p(x) = \lambda e^{-\lambda x},$$

where λ 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))`

2 *Requires:* lambda > 0.

3 *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`

4 *Returns:* The "λ" parameter of the distribution.

5.1.7.8 Class template `normal_distribution` [tr.rand.dist.norm]

1 A `normal_distribution` random distribution produces random numbers x distributed with probability density function

$$p(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(x-mean)^2/(2\sigma^2)},$$

where *mean* and σ 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);`

2 *Requires:* sigma > 0.

3 *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`

4 *Returns:* The "σ" parameter of the distribution.

5.1.7.9 Class template `gamma_distribution` [tr.rand.dist.gamma]

1 A `gamma_distribution` random distribution produces random numbers x distributed with probability density function

$$p(x) = \frac{1}{\Gamma(\alpha)} x^{\alpha-1} e^{-x},$$

where α 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));`

2 *Requires:* alpha > 0.

3 *Effects:* Constructs a `gamma_distribution` object; `alpha` is the parameter for the distribution.

`RealType alpha() const`

4 *Returns:* The "α" parameter of the distribution.

5.2 Mathematical special functions [tr.num.sf]

5.2.1 Additions to header `<cmath>` synopsis [tr.num.sf.cmath]

1 Table 18 summarizes the functions that are added to header `<cmath>`. The detailed signatures are given in the synopsis.

2 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 {
  // [5.2.1.1] 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);

  // [5.2.1.2] associated Legendre functions:
```

Table 18: Additions to header <cmath> synopsis

Type	Name(s)		
Functions:			
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	

```
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);

// [5.2.1.3] beta function:
double       beta(double x, double y);
float        betaf(float x, float y);
long double  betal(long double x, long double y);

// [5.2.1.4] (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);

// [5.2.1.5] (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);

// [5.2.1.6] (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);

// [5.2.1.7] 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);

// [5.2.1.8] 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);

// [5.2.1.9] 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);

// [5.2.1.10] 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);

// [5.2.1.11] 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);

// [5.2.1.12] (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);

// [5.2.1.13] (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);

// [5.2.1.14] (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);

// [5.2.1.15] exponential integral:
double       expint(double x);
float        expintf(float x);
long double  expintl(long double x);

// [5.2.1.16] Hermite polynomials:
double       hermite(unsigned n, double x);
float        hermitef(unsigned n, float x);
long double  hermitel(unsigned n, long double x);

// [5.2.1.17] 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);

// [5.2.1.18] Laguerre polynomials:
double       laguerre(unsigned n, double x);
float        laguerref(unsigned n, float x);
long double  laguerrel(unsigned n, long double x);

// [5.2.1.19] Legendre polynomials:
```

```
  double       legendre(unsigned l, double x);
  float        legendref(unsigned l, float x);
  long double  legendrel(unsigned l, long double x);

  // [5.2.1.20] Riemann zeta function:
  double       riemann_zeta(double);
  float        riemann_zetaf(float);
  long double  riemann_zetal(long double);

  // [5.2.1.21] 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);

  // [5.2.1.22] 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);

  // [5.2.1.23] 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
```

3 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).

4 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`.

5 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.

6 Each of the functions declared above shall return a NaN (Not a Number) if any argument value is a NaN, but it shall not report a domain error. Otherwise, each of the functions declared above shall report a domain error for just those argument values for which:

- — the function description's Returns clause explicitly specifies a domain, and those arguments fall outside the specified domain; or
- — the corresponding mathematical function value has a non-zero imaginary component; or
- — the corresponding mathematical function is not mathematically defined.[2)]

7 Unless otherwise specified, a function is defined for all finite values, for negative infinity, and for positive infinity.

5.2.1.1 associated Laguerre polynomials [tr.num.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);
```

1 *Effects:* These functions compute the associated Laguerre polynomials of their respective arguments `n`, `m`, and `x`.

2 *Returns:* The `assoc_laguerre` functions return

$$\mathsf{L}_n^m(x) = (-1)^m \frac{\mathsf{d}^m}{\mathsf{d}x^m} \mathsf{L}_{n+m}(x), \quad \text{for } x \geq 0.$$

3 *Note:* The effect of calling each of these functions is implementation-defined if `n >= 128`.

5.2.1.2 associated Legendre functions [tr.num.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);
```

1 *Effects:* These functions compute the associated Legendre functions of their respective arguments `l`, `m`, and `x`.

2 *Returns:* The `assoc_legendre` functions return

$$\mathsf{P}_\ell^m(x) = (1-x^2)^{m/2} \frac{\mathsf{d}^m}{\mathsf{d}x^m} \mathsf{P}_\ell(x), \quad \text{for } x \geq 0.$$

3 *Note:* The effect of calling each of these functions is implementation-defined if `l >= 128`.

5.2.1.3 beta function [tr.num.sf.beta]

```
double       beta(double x, double y);
float        betaf(float x, float y);
long double  betal(long double x, long double y);
```

[2)] A mathematical function is mathematically defined for a given set of argument values if if it explicitly defined for that set of argument values or if its limiting value exists and does not depend on the direction of approach.

1 *Effects:* These functions compute the beta function of their respective arguments `x` and `y`.

2 *Returns:* The `beta` functions return

$$\mathsf{B}(x,y) = \frac{\Gamma(x)\,\Gamma(y)}{\Gamma(x+y)} \; .$$

5.2.1.4 (complete) elliptic integral of the first kind [tr.num.sf.ellK]

```
double       comp_ellint_1(double k);
float        comp_ellint_1f(float k);
long double  comp_ellint_1l(long double k);
```

1 *Effects:* These functions compute the complete elliptic integral of the first kind of their respective arguments `k`.

2 *Returns:* The `comp_ellint_1` functions return

$$\mathsf{K}(k) = \mathsf{F}(k, \pi/2) = \int_0^{\pi/2} \frac{\mathsf{d}\theta}{\sqrt{1-k^2\sin^2\theta}} \; .$$

5.2.1.5 (complete) elliptic integral of the second kind [tr.num.sf.ellEx]

```
double       comp_ellint_2(double k);
float        comp_ellint_2f(float k);
long double  comp_ellint_2l(long double k);
```

1 *Effects:* These functions compute the complete elliptic integral of the second kind of their respective arguments `k`.

2 *Returns:* The `comp_ellint_2` functions return

$$\mathsf{E}(k, \pi/2) = \int_0^{\pi/2} \sqrt{1-k^2\sin^2\theta}\,\mathsf{d}\theta \; .$$

5.2.1.6 (complete) elliptic integral of the third kind [tr.num.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);
```

1 *Effects:* These functions compute the complete elliptic integral of the third kind of their respective arguments `k` and `n`.

2 *Returns:* The `comp_ellint_3` functions return

$$\mathsf{\Pi}(\nu, k, \pi/2) = \int_0^{\pi/2} \frac{\mathsf{d}\theta}{(1-\nu\,\sin^2\theta)\sqrt{1-k^2\sin^2\theta}} \; .$$

5.2.1.7 confluent hypergeometric functions [tr.num.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);
```

1 *Effects:* These functions compute the confluent hypergeometric functions of their respective arguments `a`, `c`, and `x`.

2 *Returns:* The `conf_hyperg` functions return

$$\mathsf{F}(a;c;x) = \frac{\Gamma(c)}{\Gamma(a)} \sum_{n=0}^{\infty} \frac{\Gamma(a+n)\,x^n}{\Gamma(c+n)\,n!} \;.$$

5.2.1.8 regular modified cylindrical Bessel functions [tr.num.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);
```

1 *Effects:* These functions compute the regular modified cylindrical Bessel functions of their respective arguments `nu` and `x`.

2 *Returns:* The `cyl_bessel_i` functions return

$$\mathsf{I}_\nu(x) = \mathrm{i}^{-\nu}\,\mathsf{J}_\nu(\mathrm{i}x) = \sum_{k=0}^{\infty} \frac{(x/2)^{\nu+2k}}{k!\,\Gamma(\nu+k+1)}, \quad \text{for } x \geq 0.$$

3 *Note:* The effect of calling each of these functions is implementation-defined if `nu >= 128`.

5.2.1.9 cylindrical Bessel functions (of the first kind) [tr.num.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);
```

1 *Effects:* These functions compute the cylindrical Bessel functions of the first kind of their respective arguments `nu` and `x`.

2 *Returns:* The `cyl_bessel_j` functions return

$$\mathsf{J}_\nu(x) = \sum_{k=0}^{\infty} \frac{(-1)^k (x/2)^{\nu+2k}}{k!\,\Gamma(\nu+k+1)}, \quad \text{for } x \geq 0.$$

3 *Note:* The effect of calling each of these functions is implementation-defined if `nu >= 128`.

5.2.1.10 irregular modified cylindrical Bessel functions [tr.num.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);
```

1 *Effects:* These functions compute the irregular modified cylindrical Bessel functions of their respective arguments `nu` and `x`.

2 *Returns:* The `cyl_bessel_k` functions return

$$\mathsf{K}_\nu(x) = (\pi/2)\mathrm{i}^{\nu+1}(\mathsf{J}_\nu(\mathrm{i}x) + \mathrm{i}\mathsf{N}_\nu(\mathrm{i}x)) = \begin{cases} \dfrac{\pi}{2}\dfrac{\mathsf{I}_{-\nu}(x) - \mathsf{I}_\nu(x)}{\sin\nu\pi}, & \text{for } x \ge 0 \text{ and non-integral } \nu \\ \dfrac{\pi}{2}\lim\limits_{\mu\to\nu}\dfrac{\mathsf{I}_{-\mu}(x) - \mathsf{I}_\mu(x)}{\sin\mu\pi}, & \text{for } x \ge 0 \text{ and integral } \nu \end{cases}.$$

3 *Note:* The effect of calling each of these functions is implementation-defined if `nu >= 128`.

5.2.1.11 cylindrical Neumann functions [tr.num.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);
```

1 *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`.

2 *Returns:* The `cyl_neumann` functions return

$$\mathsf{N}_\nu(x) = \begin{cases} \dfrac{\mathsf{J}_\nu(x)\cos\nu\pi - \mathsf{J}_{-\nu}(x)}{\sin\nu\pi}, & \text{for } x \ge 0 \text{ and non-integral } \nu \\ \lim\limits_{\mu\to\nu}\dfrac{\mathsf{J}_\mu(x)\cos\mu\pi - \mathsf{J}_{-\mu}(x)}{\sin\mu\pi}, & \text{for } x \ge 0 \text{ and integral } \nu \end{cases}.$$

3 *Note:* The effect of calling each of these functions is implementation-defined if `nu >= 128`.

5.2.1.12 (incomplete) elliptic integral of the first kind [tr.num.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);
```

1 *Effects:* These functions compute the incomplete elliptic integral of the first kind of their respective arguments `k` and `phi` (`phi` measured in radians).

2 *Returns:* The `ellint_1` functions return

$$\mathsf{F}(k,\phi) = \int_0^\phi \frac{\mathrm{d}\theta}{\sqrt{1-k^2\sin^2\theta}}, \quad \text{for } |k| \le 1.$$

5.2.1.13 (incomplete) elliptic integral of the second kind [tr.num.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);
```

1 *Effects:* These functions compute the incomplete elliptic integral of the second kind of their respective arguments `k` and `phi` (`phi` measured in radians).

2 *Returns:* The `ellint_2` functions return

$$\mathsf{E}(k,\phi) = \int_0^{\phi} \sqrt{1-k^2\sin^2\theta}\,\mathrm{d}\theta, \quad \text{for } |k| \le 1.$$

5.2.1.14 (incomplete) elliptic integral of the third kind [tr.num.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);
```

1 *Effects:* These functions compute the incomplete elliptic integral of the third kind of their respective arguments `k`, `nu`, and `phi` (`phi` measured in radians).

2 *Returns:* The `ellint_3` functions return

$$\Pi(\nu,k,\phi) = \int_0^{\phi} \frac{\mathrm{d}\theta}{(1-\nu\sin^2\theta)\sqrt{1-k^2\sin^2\theta}}, \quad \text{for } |k| \le 1.$$

5.2.1.15 exponential integral [tr.num.sf.ei]

```
double       expint(double x);
float        expintf(float x);
long double  expintl(long double x);
```

1 *Effects:* These functions compute the exponential integral of their respective arguments `x`.

2 *Returns:* The `expint` functions return

$$\mathsf{Ei}(x) = -\int_{-x}^{\infty} \frac{e^{-t}}{t}\,\mathrm{d}t\ .$$

5.2.1.16 Hermite polynomials [tr.num.sf.Hn]

```
double       hermite(unsigned n, double x);
float        hermitef(unsigned n, float x);
long double  hermitel(unsigned n, long double x);
```

1 *Effects:* These functions compute the Hermite polynomials of their respective arguments `n` and `x`.

2 *Returns:* The `hermite` functions return

$$\mathsf{H}_n(x) = (-1)^n e^{x^2} \frac{\mathsf{d}^n}{\mathsf{d}x^n} e^{-x^2} \ .$$

3 *Note:* The effect of calling each of these functions is implementation-defined if `n >= 128`.

5.2.1.17 hypergeometric functions [tr.num.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);
```

1 *Effects:* These functions compute the hypergeometric functions of their respective arguments `a`, `b`, `c`, and `x`.

2 *Returns:* The `hyperg` functions return

$$\mathsf{F}(a,b;c;x) = \frac{\Gamma(c)}{\Gamma(a)\Gamma(b)} \sum_{n=0}^{\infty} \frac{\Gamma(a+n)\,\Gamma(b+n)}{\Gamma(c+n)} \frac{x^n}{n!} \ .$$

5.2.1.18 Laguerre polynomials [tr.num.sf.Ln]

```
double       laguerre(unsigned n, double x);
float        laguerref(unsigned n, float x);
long double  laguerrel(unsigned n, long double x);
```

1 *Effects:* These functions compute the Laguerre polynomials of their respective arguments `n` and `x`.

2 *Returns:* The `laguerre` functions return

$$\mathsf{L}_n(x) = \frac{e^x}{n!} \frac{\mathsf{d}^n}{\mathsf{d}x^n} (x^n e^{-x}), \quad \text{for } x \geq 0.$$

3 *Note:* The effect of calling each of these functions is implementation-defined if `n >= 128`.

5.2.1.19 Legendre polynomials [tr.num.sf.Pl]

```
double       legendre(unsigned l, double x);
float        legendref(unsigned l, float x);
long double  legendrel(unsigned l, long double x);
```

1 *Effects:* These functions compute the Legendre polynomials of their respective arguments `l` and `x`.

2 *Returns:* The `legendre` functions return

$$\mathsf{P}_\ell(x) = \frac{1}{2^\ell\,\ell!} \frac{\mathsf{d}^\ell}{\mathsf{d}x^\ell} (x^2-1)^\ell, \quad \text{for } |x| \leq 1.$$

3 *Note:* The effect of calling each of these functions is implementation-defined if `l >= 128`.

5.2.1.20 Riemann zeta function [tr.num.sf.riemannzeta]

```
double       riemann_zeta(double x);
float        riemann_zetaf(float x);
long double  riemann_zetal(long double x);
```

1 *Effects:* These functions compute the Riemann zeta function of their respective arguments `x`.

2 *Returns:* The `riemann_zeta` functions return

$$\zeta(x) = \begin{cases} \displaystyle\sum_{k=1}^{\infty} k^{-x}, & \text{for } x > 1 \\ 2^x \pi^{x-1} \sin(\frac{\pi x}{2})\, \Gamma(1-x)\, \zeta(1-x), & \text{for } x < 1 \end{cases} .$$

5.2.1.21 spherical Bessel functions (of the first kind) [tr.num.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);
```

1 *Effects:* These functions compute the spherical Bessel functions of the first kind of their respective arguments `n` and `x`.

2 *Returns:* The `sph_bessel` functions return

$$\mathsf{j}_n(x) = (\pi/2x)^{1/2} \mathsf{J}_{n+1/2}(x), \quad \text{for } x \geq 0.$$

3 *Note:* The effect of calling each of these functions is implementation-defined if `n >= 128`.

5.2.1.22 spherical associated Legendre functions [tr.num.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);
```

1 *Effects:* These functions compute the spherical associated Legendre functions of their respective arguments `l`, `m`, and `theta` (`theta` measured in radians).

2 *Returns:* The `sph_legendre` functions return

$$\mathsf{Y}_\ell^m(\theta, 0)$$

where

$$\mathsf{Y}_\ell^m(\theta,\phi) = (-1)^m \left[\frac{(2\ell+1)}{4\pi} \frac{(\ell-m)!}{(\ell+m)!} \right]^{1/2} \mathsf{P}_\ell^m(\cos\theta) e^{im\phi}, \quad \text{for } |m| \leq \ell.$$

[*Note:* This formulation avoids any need to return non-real numbers. *—end note*]

3 *Note:* The effect of calling each of these functions is implementation-defined if `l >= 128`.

5.2.1.23 spherical Neumann functions [tr.num.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);
```

1 *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`.

2 *Returns:* The `sph_neumann` functions return

$$\mathsf{n}_n(x) = (\pi/2x)^{1/2}\mathsf{N}_{n+1/2}(x), \quad \text{for } x \geq 0.$$

3 *Note:* The effect of calling each of these functions is implementation-defined if `n >= 128`.

5.2.2 Additions to header `<math.h>` synopsis [tr.num.sf.mathh]

1 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.

6 Containers [tr.cont]

1 This clause describes components that C++ programs may use to organize collections of information.

2 The following subclauses describe tuples, fixed size arrays, and unordered associated containers, as summarized in Table 19.

Table 19: Container library summary

Subclause	Header(s)
6.1 Tuple types	`<tuple>` `<utility>`
6.2 Fixed size array	`<array>`

6.3 Unordered associative containers	`<functional>`
	`<unordered_set>`
	`<unordered_map>`

6.1 Tuple types [tr.tuple]

1 6.1 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 A. 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]

```
namespace std {
namespace tr1 {
  // [6.1.3] Class template tuple
  template <class T1 = unspecified ,
            class T2 = unspecified ,
            ...,
            class TM = unspecified> class tuple;

  // [6.1.3.2] Tuple creation functions
  const unspecified ignore;

  template<class T1, class T2, ..., class TN>
    tuple<V1, 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&);

  // [6.1.3.3] Tuple helper classes
  template <class T> class tuple_size;

  template <int I, class T> class tuple_element;

  // [6.1.3.4] Element access
  template <int I, class T1, class T2, ..., class TN>
    RJ get(tuple<T1, T2, ..., TN>&);

  template <int I, class T1, class T2, ..., class TN>
    PJ get(const tuple<T1, T2, ..., TN>&);

  // [6.1.3.5] relational operators
  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>&);
} // namespace tr1
} // namespace std
```

6.1.2 Additions to header `<utility>` synopsis [tr.tuple.synopsis.pair]

```
namespace std {
namespace tr1 {
  template <class T> class tuple_size; // forward declaration
  template <int I, class T> class tuple_element; // forward declaration

  template <class T1, class T2> struct tuple_size<std::pair<T1, T2> >;
  template <class T1, class T2> struct tuple_element<0, std::pair<T2, T2> >;
  template <class T1, class T2> struct tuple_element<1, std::pair<T2, T2> >;

  // see below for definition of "P".
  template<int I, class T1, class T2> P& get(std::pair<T1, T2>&);
  template<int I, class T1, class T2> const P& get(const std::pair<T1, T2>&);
```

```
} // namespace tr1
} // namespace std
```

6.1.3 Class template `tuple` [tr.tuple.tuple]

1 `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.

2 [*Example:* Given the instantiation `tuple<int, float, char>`, `N` is 3. *—end example*]

```
template <class T1 = unspecified,
          class T2 = unspecified,
          ...,
          class TM = unspecified>
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>&);           // iff N == 2

  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>&); // iff N == 2
};
```

6.1.3.1 Construction [tr.tuple.cnstr]

`tuple();`

1 *Requires:* Each type `Ti` shall be default constructible.

2 *Effects:* Default initializes each element.

`tuple(P1, P2, ..., PN);`

3 Where, if `Ti` is a reference type then `Pi` is `Ti`, otherwise `Pi` is `const Ti&`.

4 *Requires:* Each type `Ti` shall be copy constructible.

5 *Effects:* Copy initializes each element with the value of the corresponding parameter.

`tuple(const tuple& u);`

6 *Requires:* Each type `Ti` shall be copy constructible.

7 *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);`

8 *Requires:* Each type `Ti` shall be constructible from the corresponding type `Ui`.

9 *Effects:* Constructs each element of `*this` with the corresponding element of `u`.

10 [*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);
```

11 *Requires:* `T1` shall be constructible from `U1`, `T2` shall be constructible from `U2`. `N == 2`.

12 *Effects:* Constructs the first element with `u.first` and the second element with `u.second`.

```
tuple& operator=(const tuple& u);
```

13 *Requires:* Each type `Ti` shall be assignable.

14 *Effects:* Assigns each element of `u` to the corresponding element of `*this`.

15 *Returns:* `*this`

```
template <class U1, class U2, ..., class UN> tuple& operator=(const tuple<U1, U2, ..., UN>& u);
```

16 *Requires:* Each type `Ti` shall be assignable from the corresponding type `Ui`.

17 *Effects:* Assigns each element of `u` to the corresponding element of `*this`.

18 *Returns:* `*this`

```
template <class U1, class U2> tuple& operator=(const pair<U1, U2>& u);
```

19 *Requires:* `T1` shall be assignable from `U1`, `T2` shall be assignable from `U2`. `N == 2`.

20 *Effects:* Assigns `u.first` to the first element of `*this` and `u.second` to the second element of `*this`.

21 *Returns:* `*this`

22 [*Note:* There are 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 it 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*]

6.1.3.2 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);
```

1 where `Vi` is `X&`, if the cv-unqualified type `Ti` is `reference_wrapper<X>`, otherwise `Vi` is `Ti`.

2 The `make_tuple` function template shall be implemented for each different number of arguments from 0 to the maximum number of allowed tuple elements.

3 *Returns:* `tuple<V1, V2, ..., VN>(t1, t2, ..., tn)`.

4 [*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);
```

5 The `tie` function template shall be implemented for each different number of arguments from 0 to the maximum number of allowed tuple elements.

6 *Returns:* `tuple<T1&, T2&, ..., TN&>(t1, t2, ..., tn)`. When an argument `t`$_i$ is `ignore`, assigning any value to the corresponding tuple element has no effect.

7 [*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*]

6.1.3.3 Valid expressions for tuple types [tr.tuple.expr]

```
tuple_size<T>::value
```

1 *Requires:* `T` is an instantiation of class template tuple.

2 `Type`: integral constant expression.

3 `Value`: Number of elements in `T`.

```
tuple_element<I, T>::type
```

4 *Requires:* $0 \leq$ `I` $<$ `N`. The program is ill-formed if `I` is out of bounds.

5 `Value`: The type of the `I`th element of `T`, where indexing is zero-based.

6.1.3.4 Element access [tr.tuple.elem]

```
template <int I, class T1, class T2, ..., class TN>
  RJ get(tuple<T1, T2, ..., TN>& t);
```

1 *Requires:* $0 \leq I < N$. The program is ill-formed if I is out of bounds.

2 *Return type:* RJ, where J=I+1. If TJ is a reference type, then RJ is TJ, otherwise RJ is TJ&.

3 *Returns:* A reference to the Ith element of t, where indexing is zero-based.

```
template <int I, class T1, class T2, ..., class TN>
  PJ get(const tuple<T1, T2, ..., TN>& t);
```

4 *Requires:* $0 \leq I < N$. The program is ill-formed if I is out of bounds.

5 *Return type:* PJ, where J=I+1. If TJ is a reference type, then PJ is TJ, otherwise PJ is const TJ&.

6 *Returns:* A const reference to the Ith element of t, where indexing is zero-based.

7 [*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*.]

8 [*Note:* The reason get is defined to be a nonmember function is that, if this functionality had been provided as a member function, invocations where the type depended on a template parameter would have required using the template keyword. *—end note*]

6.1.3.5 Relational operators [tr.tuple.rel]

```
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);
```

1 *Requires:* tuple_size<tuple<T1, T2, ..., TM> >::value == tuple_size<tuple<U1, U2, ..., UM> >::value == N. For all i, where 0 <= i < N, get<i>(t) == get<i>(u) is a valid expression returning a type that is convertible to bool.

2 *Returns:* true iff get<i>(t) == get<i>(u) for all i. For any two zero-length tuples e and f, e == f returns true.

3 *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 TN, class U1, class U2, ..., class UN>
  bool operator<(const tuple<T1, T2, ..., TN>& t, const tuple<U1, U2, ..., UN>& u);
```

4 *Requires:* tuple_size<tuple<T1, T2, ..., TM> >::value == tuple_size<tuple<U1, U2, ..., UM> >::value == N. For all i, where 0 <= i < N, get<i>(t) < get<i>(u) is a valid expression returning a type that is convertible to bool.

5 *Returns:* The result of a lexicographical comparison between t and u. The result is defined as: (bool)(get<0>(t) < get<0>(u)) || (!(bool)(get<0>(u) < get<0>(t)) && t_{tail} < u_{tail}), where r_{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 < f returns 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);
```

6 *Returns:* `!(t == u)`.

```
template<class T1, class T2, ..., class TN, class U1, class U2, ..., class UN>
  bool operator>(const tuple<T1, T2, ..., TN>& t, const tuple<U1, U2, ..., UN>& u);
```

7 *Returns:* `u < t`.

```
template<class T1, class T2, ..., class TN, class U1, class U2, ..., class UN>
  bool operator<=(const tuple<T1, T2, ..., TN>& t, const tuple<U1, U2, ..., UN>& u);
```

8 *Returns:* `!(u < t)`

```
template<class T1, class T2, ..., class TN, class U1, class U2, ..., class UN>
  bool operator>=(const tuple<T1, T2, ..., TN>& t, const tuple<U1, U2, ..., UN>& u);
```

9 *Returns:* `!(t < u)`

10 [*Note:* The above definitions for comparison operators do not impose the requirement that $\mathtt{t}_{\text{tail}}$ (or $\mathtt{u}_{\text{tail}}$) shall be constructed. It may not even be possible, as `t` and `u` are 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. —*end note*]

6.1.4 Pairs [tr.tuple.pairs]

1 These templates are extensions 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>&);
```

2 *Return type:* If `I` is `0` then `P` is `T1`, if `I` is `1` then `P` is `T2`, otherwise the program is ill-formed.

3 *Returns:* If `I == 0` returns `p.first`, otherwise returns `p.second`.

6.2 Fixed size array [tr.array]

6.2.1 Header `<array>` synopsis [tr.array.syn]

```
namespace std {
namespace tr1 {
  // [6.2.2] Class template array
  template <class T, size_t N > struct array;

  // Array comparisons
  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);

  // [6.2.2.2] Specialized algorithms
  template <class T, size_t N > void swap(array<T,N>& x, array<T,N>& y);

  // [6.2.2.5] Tuple interface to class template array
  template <class T> class tuple_size; // forward declaration
  template <int I, class T> class tuple_element; // forward declaration

  template <class T, size_t N>        struct tuple_size<array<T, N> >;
  template <int I, class T, size_t N> struct tuple_element<I, array<T, N> >;

  template <int I, class T, size_t N>       T& get(      array<T, N>&);
  template <int I, class T, size_t N> const T& get(const array<T, N>&);
} // namespace tr1
} // namespace std
```

6.2.2 Class template `array` [tr.array.array]

1 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`.

2 An `array` is an aggregate ([dcl.init.aggr]) that can be initialized with the syntax

```
array a = { initializer-list };
```

where *initializer-list* is a comma separated list of up to `N` elements whose types are convertible to `T`.

3 Unless otherwise specified, all `array` operations are as described in [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.

4 The effect of calling `front()` or `back()` for a zero-sized array is implementation defined.

```
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];            // Exposition only

    // 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);
}
```

5 [*Note*: The member variable `elems` is shown for exposition only, to empahasize that `array` is a class aggregate. The name `elems` is not part of `array`'s interface. *—end note*]

6.2.2.1 `array` constructors, copy, and assignment [tr.array.cons]

1 **Initialization:** The conditions for an aggregate ([dcl.init.aggr]) shall be met. Class `array` relies on the implicitly-declared special member functions ([class.ctor], [class.dtor], and [class.copy]) to conform to the container requirements table in [lib.container.requirements].

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);
```

1 *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();
```

1 *Returns:* `N`

6.2.2.4 Zero sized arrays [tr.array.zero]

1 `array` shall provide support for the special case `N == 0`.

2 In the case that `N == 0`, `begin() == end() ==` unique value.

6.2.2.5 Tuple interface to class template `array` [tr.array.tuple]

```
tuple_size<array<T, N> >::value
```

1 *Return type:* integral constant expression.

2 `Value`: `N`

```
tuple_element<I, array<T, N> >::type
```

3 *Requires:*`0 <= I < N`. The program is ill-formed if I is out of bounds.

4 `value`: The type T.

```
template <int I, class T, size_t N> T& get(array<T, N>& a);
```

5 *Requires:* `0 <= I < N`. The program is ill-formed if `I` is out of bounds.

6 *Returns:* A reference to the `I`th element of `a`, where indexing is zero-based.

```
template <int I, class T, size_t N> const T& get(const array<T, N>& a);
```

7 *Requires:* `0 <= I < N`. The program is ill-formed if `I` is out of bounds.

8 *Return type:* `const T&`.

9 *Returns:* A const reference to the `I`th element of `a`, where indexing is zero-based.

6.3 Unordered associative containers [tr.hash]

6.3.1 Unordered associative container requirements [tr.unord.req]

1 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 unordered associative containers: `unordered_set`, `unordered_map`, `unordered_multiset`, and `unordered_multimap`.

2 Unordered associative containers conform to the requirements for Containers ([lib.container.requirements]), except that the expressions in table 20 are not required to be valid, where a and b denote values of a type X, and X is an unordered associative container class:

Table 20: Container requirements that are not supported by unordered associative containers

unsupported expressions
`a == b`
`a != b`
`a < b`
`a > b`
`a <= b`
`a >= b`

3 Each unordered associative container is parameterized by `Key`, by a function object `Hash` that acts as a hash function for values of type `Key`, and by 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`.

4 A hash function is a function object that takes a single argument of type `Key` and returns a value of type `std::size_t`.

5 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.

6 An unordered 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.

7 For `unordered_set` and `unordered_multiset` the value type is the same as the key type. For `unordered_map` and `unordered_multimap` it is `std::pair<const Key, T>`.

8 The elements of an unordered 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 an unordered 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.

9 In table 21: `X` is an unordered associative container class, `a` is an object of type `X`, `b` is a possibly const object of type `X`, `a_uniq` 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 21: Unordered associative container requirements (in addition to container)

expression	return type	assertion/notepre/post-condition	complexity
`X::key_type`	`Key`	`Key` is `Assignable` and `CopyConstructible`	compile time
`X::hasher`	`Hash`	`Hash` is a unary function object type such that the expression `hf(k)` has type `std::size_t`.	compile time
`X::key_equal`	`Pred`	`Pred` is a binary predicate that takes two arguments of type `Key`. `Pred` is an equivalence relation.	compile time
`X::local_iterator`	An iterator type whose category, value type, difference type, and pointer and reference types are the same as `X::iterator`'s.	A `local_iterator` object may be used to iterate through a single bucket, but may not be used to iterate across buckets.	compile time
`X::const_local_-iterator`	An iterator type whose category, value type, difference type, and pointer and reference types are the same as `X::const_iterator`'s.	A `const_local_iterator` object may be used to iterate through a single bucket, but may not be used to iterated across buckets.	compile time

expression	return type	assertion/notepre/post-condition	complexity
`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 $\mathcal{O}(N^2)$
`X(i, j)` `X a(i, j)`	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, and inserts elements from `[i, j)` into it.	Average case $\mathcal{O}(N)$ (N is `distance(i, j)`), worst case $\mathcal{O}(N^2)$

expression	return type	assertion/notepre/post-condition	complexity
X(b) X a(b)	X	Copy constructor. In addition to the contained elements, the hash function, predicate, and maximum load factor are copied.	Average case linear in b.size(), worst case quadratic.
a = b	X	Copy assignment operator. In addition to the contained elements, the hash function, predicate, and maximum load factor are copied.	Average case linear in b.size(), worst case quadratic.
b.hash_function()	hasher	Returns the hash function out of which b was constructed.	constant
b.key_eq()	key_equal	Returns the key equality function out of which b was constructed.	constant
a_uniq.insert(t)	pair<iterator, bool>	Inserts t if and only if there is no element in the container with key equivalent to the key of t. The bool component of the returned pair indicates whether the insertion takes place, and the iterator component points to the element with key equivalent to the key of t.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(a_-uniq.size()).
a_eq.insert(t)	iterator	Inserts t, and returns an iterator pointing to the newly inserted element.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(a_-uniq.size()).
a.insert(r, t)	iterator	Equivalent to a.insert(t). Return value is an iterator pointing to the element with the key equivalent to that of t. The const iterator r 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()).
a.insert(i, j)	void	Pre: i and j are not iterators in a. Equivalent to a.insert(t) for each element in [i,j).	Average case $\mathcal{O}(N)$, where N is distance(i, j). Worst case $\mathcal{O}$(N * a.size()).
a.erase(k)	size_type	Erases all elements with key equivalent to k. Returns the number of elements erased.	Average case $\mathcal{O}$(a.count(k)). Worst case $\mathcal{O}$(a.size()).
a.erase(r)	void	Erases the element pointed to by r.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(a.size()).

expression	return type	assertion/notepre/post-condition	complexity
`a.erase(r1, r2)`	`void`	Erases all elements in the range `[r1, r2)`.	Average case $\mathcal{O}(\mathcal{N})$, where N is `distance(r1, r2)`), worst case $\mathcal{O}$(`a.size()`).
`a.clear()`	`void`	Erases all elements in the container. Post: `a.size() == 0`	Linear.
`b.find(k)`	`iterator`; `const_iterator` for const `b`.	Returns an iterator pointing to an element with key equivalent to `k`, or `b.end()` if no such element exists.	Average case $\mathcal{O}$(1), worst case $\mathcal{O}$(`b.size()`).
`b.count(k)`	`size_type`	Returns the number of elements with key equivalent to `k`.	Average case $\mathcal{O}$(1), worst case $\mathcal{O}$(`b.size()`).
`b.equal_range(k)`	`pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for const `b`.	Returns a range containing all elements with keys equivalent to `k`. Returns `make_pair(b.end(), b.end())` if no such elements exist.	Average case $\mathcal{O}$(`b.count(k)`). Worst case $\mathcal{O}$(`b.size()`).
`b.bucket_count()`	`size_type`	Returns the number of buckets that `b` contains.	Constant
`b.max_bucket_count()`	`size_type`	Returns an upper bound on the number of buckets that `b` might ever contain.	Constant
`b.bucket(k)`	`size_type`	Returns the index of the bucket in which elements with keys equivalent to `k` would be found, if any such element existed. Post: the return value is in the range `[0, b.bucket_count())`.	Constant
`b.bucket_size(n)`	`size_type`	Pre: `n` is in the range `[0, b.bucket_count())`. Returns the number of elements in the `n`th bucket.	$\mathcal{O}$(`b.bucket_-size(n)`)
`b.begin(n)`	`local_iterator`; `const_local_iterator` for const `b`.	Pre: `n` 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`th bucket.	Constant
`b.end(n)`	`local_iterator`; `const_local_iterator` for const `b`.	Pre: `n` is in the range `[0, b.bucket_count())`.	Constant
`b.load_factor()`	`float`	Returns the average number of elements per bucket.	Constant

expression	return type	assertion/notepre/post-condition	complexity
`b.max_load_factor()`	`float`	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
`a.max_load_factor(z)`	`void`	Pre: `z` is positive. Changes the container's maximum load load factor, using `z` as a hint.	Constant
`a.rehash(n)`	`void`	Post: `a.bucket_count() > a.size() / a.max_load_factor()` and `a.bucket_count() >= n`.	Average case linear in `a.size()`, worst case quadratic.

10 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*]

11 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.

12 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.

13 The insert members shall not affect the validity of iterators if `(N+n) < z * B`, where `N` is the number of elements in the container prior to the insert operation, `n` is the number of elements inserted, `B` is the container's bucket count, and `z` is the container's maximum load factor.

6.3.1.1 Exception safety guarantees [tr.unord.req.except]

1 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).

2 For unordered associative containers, if an exception is thrown by any operation other than the container's hash function from within an `insert()` function inserting a single element, the `insert()` function has no effect.

3 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).

4 For unordered associative containers, if an exception is thrown from within a rehash() function other than by the container's hash function or comparison function, the rehash() function has no effect.

6.3.2 Additions to header `<functional>` synopsis [tr.unord.fun.syn]

```
namespace std {
namespace tr1 {
  // [6.3.3] 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 <> struct hash<std::string>;
  template <> struct hash<std::wstring>;
} // namespace tr1
} // namespace std
```

6.3.3 Class template `hash` [tr.unord.hash]

1 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 ([basic.fundamental]), floating point types ([basic.fundamental]), pointer types ([dcl.ptr]), and `std::string` and `std::wstring`.

```
template <class T>
struct hash : public std::unary_function<T, std::size_t>
{
  std::size_t operator()(T val) const;
};
```

2 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 std {
namespace tr1 {
  // [6.3.4.3] Class template unordered_set
  template <class Value,
            class Hash = hash<Value>,
            class Pred = std::equal_to<Value>,
```

```
          class Alloc = std::allocator<Value> >
    class unordered_set;

  // [6.3.4.5] Class template unordered_multiset
  template <class Value,
            class Hash = hash<Value>,
            class Pred = std::equal_to<Value>,
            class Alloc = std::allocator<Value> >
    class unordered_multiset;

  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);

  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);
} // namespace tr1
} // namespace std
```

6.3.4.2 Header `<unordered_map>` synopsis [tr.unord.syn.map]

```
namespace std {
namespace tr1 {
  // [6.3.4.4] Class template 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_map;

  // [6.3.4.6] Class template unordered_multimap
  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;

  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);

  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);
} // namespace tr1
} // namespace std
```

6.3.4.3 Class template `unordered_set` [tr.unord.set]

1 An `unordered_set` is an 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.

2 An `unordered_set` satisfies all of the requirements of a container and of an 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 an `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 they are the same type.

3 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());
```

1 *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.

2 *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());
```

3 *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 $[f, l)$. `max_load_factor()` is 1.0.

4 *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);
```

1 *Effects:* `x.swap(y)`.

6.3.4.4 Class template `unordered_map` [tr.unord.map]

1 An `unordered_map` is an 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.

2 An `unordered_map` satisfies all of the requirements of a container and of an 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 an `unordered_map<Key, T>` the `key type` is `Key`, the mapped type is `T`, and the value type is `std::pair<const Key, T>`.

3 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());
```

1 *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.

2 *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());
```

3 *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 `[`*f*`, `*l*`)`. `max_load_factor()` is 1.0.

4 *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);
```

1 *Effects:* If the `unordered_map` does not already contain an element whose key is equivalent to *k*, inserts the value `std::pair<const key_type, mapped_type>(k, mapped_type())`.

2 *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 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);
```

1 *Effects:* `x.swap(y)`.

6.3.4.5 Class template `unordered_multiset` [tr.unord.multiset]

1 An `unordered_multiset` is an unordered associative container that supports equivalent keys (an `unordered_multiset` may contain multiple copies of the same key value) and in which each element's key is the element itself.

2 An `unordered_multiset` satisfies all of the requirements of a container and of an 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 an `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 they are the same type.

3 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());
```

1 *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.

2 *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());
```

3 *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 `[`f`, `l`)`. `max_load_factor()` is 1.0.

4 *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);
```

1 *Effects:* `x.swap(y);`

6.3.4.6 Class template `unordered_multimap` [tr.unord.multimap]

1 An `unordered_multimap` is an 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.

2 An `unordered_multimap` satisfies all of the requirements of a container and of an 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>`.

3 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;

// 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());
```

1 *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.

2 *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());
```

3 *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 `[f, l)`. `max_load_factor()` is 1.0.

4 *Complexity:* Average case linear, worst case quadratic.

6.3.4.6.2 `unordered_multimap` swap [tr.unord.multimap.swap]

```
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);
```

1 *Effects:* `x.swap(y)`.

7 Regular expressions [tr.re]

1 This clause describes components that C++ programs may use to perform operations involving regular expression matching and searching.

7.1 Definitions [tr.re.def]

1 The following definitions shall apply to this clause:

2 *Collating element*: A sequence of one or more characters within the current locale that collate as if they were a single character.

3 *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.

4 *Format specifier*: A sequence of one or more characters that is to be replaced with some part of a regular expression match.

5 *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.

6 *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.

7 *Regular expression*: A pattern that selects specific strings from a set of character strings.

8 *Sub-expression*: A subset of a regular expression that has been marked by parenthesis.

7.2 Requirements [tr.re.req]

1 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*]

2 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 typedefs and functions in the template parameter `traits` used by the `basic_regex` class template. This subclause defines the semantics guaranteed by these members.

3 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`.

4 In Table 22 `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; `F1`

and F2 are forward 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; cl is an object of type X::char_class_type, and loc is an object of type X::locale_type.

Table 22: regular expression traits class requirements

expression	Return Type	Assertion / Note / Pre / Post condition
X::char_type	charT	The character container type used in the implementation of class template basic_regex.
X::string_type	std::basic_string<charT>	
X::locale_type	A copy constructible type	A type that represents the locale used by the traits class.
X::char_class_type	A bitmask type [lib.bitmask.types].	A bitmask type representing a particular character classification.
X::length(p)	std::size_t	Yields the smallest i such that p[i] == 0. Complexity is linear in i .
v.translate(c)	X::char_type	Returns a character such that for any character d that is to be considered equivalent to c then v.translate(c) == v.translate(d).
v.translate_nocase(c)	X::char_type	For all characters C that are to be considered equivalent to c when comparisons are to be performed without regard to case, then v.translate_nocase(c) == v.translate_nocase(C).
v.transform(F1, F2)	X::string_type	Returns a sort key for the character sequence designated by the iterator range [F1, F2) such that if the character sequence [G1, G2) sorts before the character sequence [H1, H2) then v.transform(G1, G2) < v.transform(H1, H2).
v.transform_primary(F1, F2)	X::string_type	Returns a sort key for the character sequence designated by the iterator range [F1, F2) such that if the character sequence [G1, G2) sorts before the character sequence [H1, H2) when character case is not considered then v.transform_primary(G1, G2) < v.transform_primary(H1, H2).
v.lookup_collatename(F1, F2)	X::string_type	Returns a sequence of characters that represents the collating element consisting of the character sequence designated by the iterator range [F1, F2). Returns an empty string if the character sequence is not a valid collating element.

expression	Return Type	Assertion / Note / Pre / Post condition
`v.lookup_classname(F1, F2)`	`X::char_class_-type`	Converts the character sequence designated by the iterator range `[F1,F2)` into a value of a bitmask type that can subsequently be passed to `isctype`. Values returned from `lookup_classname` can be safely bitwise or'ed together. Returns 0 if the character sequence is not the name of a character class recognized by `X`. The value returned shall be independent of the case of the characters in the sequence.
`v.isctype(c, cl)`	`bool`	Returns `true` if character `c` is a member of one of the character classes designated by `cl`, `false` otherwise.
`v.value(c, I)`	`int`	Returns the value represented by the digit *c* in base *I* if the character *c* is a valid digit in base *I*; otherwise returns `-1`. [*Note:* the value of *I* will only be 8, 10, or 16. *—end note*]
`u.imbue(loc)`	`X::locale_type`	Imbues `u` with the locale `loc` and returns the previous locale used by `u` if any.
`v.getloc()`	`X::locale_type`	Returns the current locale used by `v`, if any.

5 The header `<regex>` defines the class template `regex_traits` which shall be capable of being specialized 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.sum]**

1 The header `<regex>` defines a basic regular expression class template and its traits that can handle all char-like template arguments ([lib.strings]).

2 The header `<regex>` defines a class template that holds the result of a regular expression match.

3 The header `<regex>` defines a series of algorithms that allow an iterator sequence to be operated upon by a regular expression.

4 The header `<regex>` defines two specific template classes, `regex` and `wregex`, and their special traits.

5 The header `<regex>` also defines two iterator types for enumerating regular expression matches.

7.4 Header `<regex>` synopsis **[tr.re.syn]**

```
namespace std {
namespace tr1 {

  // [7.5] Regex constants
  namespace regex_constants {
    typedef bitmask_type syntax_option_type;
    typedef bitmask_type match_flag_type;
    typedef implementation-defined error_type;
```

```
} // namespace regex_constants

// [7.6] Class regex_error
class regex_error;

// [7.7] Class template regex_traits
template <class charT> struct regex_traits;

// [7.8] Class template basic_regex
template <class charT, class traits = regex_traits<charT> > class basic_regex;

typedef basic_regex<char>    regex;
typedef basic_regex<wchar_t> wregex;

// [7.8.6] basic_regex swap
template <class charT, class traits>
  void swap(basic_regex<charT, traits>& e1, basic_regex<charT, traits>& e2);

// [7.9] Class template sub_match
template <class BidirectionalIterator>
  class sub_match;

typedef sub_match<const char*>             csub_match;
typedef sub_match<const wchar_t*>          wcsub_match;
typedef sub_match<string::const_iterator>  ssub_match;
typedef sub_match<wstring::const_iterator> wssub_match;

// [7.9.2] sub_match non-member operators
template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);


template <class BiIter, class ST, class SA>
  bool operator==(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator!=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator<(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
```

```
                 const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator>(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator>=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter, class ST, class SA>
  bool operator<=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
                  const sub_match<BiIter>& rhs);

template <class BiIter, class ST, class SA>
  bool operator==(const sub_match<BiIter>& lhs,
                  const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator!=(const sub_match<BiIter>& lhs,
                  const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator<(const sub_match<BiIter>& lhs,
                 const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator>(const sub_match<BiIter>& lhs,
                 const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator>=(const sub_match<BiIter>& lhs,
                  const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
template <class BiIter, class ST, class SA>
  bool operator<=(const sub_match<BiIter>& lhs,
                  const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);

template <class BiIter>
  bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator!=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<(typename iterator_traits<BiIter>::value_type const* lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>(typename iterator_traits<BiIter>::value_type const* lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);

template <class BiIter>
```

```
  bool operator==(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);

template <class BiIter>
  bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,
                 const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
template <class BiIter>
  bool operator<=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);

template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
```

```
  bool operator>=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);

template <class charT, class ST, class BiIter>
  basic_ostream<charT, ST>&
  operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);

// [7.10] Class template match_results
template <class BidirectionalIterator,
          class Allocator = allocator<sub_match<BidirectionalIterator> > >
  class match_results;

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;

// match_results comparisons
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);

// [7.10.6] match_results swap
template <class BidirectionalIterator, class Allocator>
  void swap(match_results<BidirectionalIterator, Allocator>& m1,
            match_results<BidirectionalIterator, Allocator>& m2);

// [7.11.2] Function template regex_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);
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);

// [7.11.3] Function template regex_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);
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 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 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);

// [7.11.4] Function template regex_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);
```

```
  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);

  // [7.12.1] Class template regex_iterator
  template <class BidirectionalIterator,
            class charT = typename iterator_traits<BidirectionalIterator>::value_type,
            class traits = regex_traits<charT> >
    class regex_iterator;

  typedef regex_iterator<const char*>             cregex_iterator;
  typedef regex_iterator<const wchar_t*>          wcregex_iterator;
  typedef regex_iterator<string::const_iterator>  sregex_iterator;
  typedef regex_iterator<wstring::const_iterator> wsregex_iterator;

  // [7.12.2] Class template regex_token_iterator
  template <class BidirectionalIterator,
            class charT = typename iterator_traits<BidirectionalIterator>::value_type,
            class traits = regex_traits<charT> >
    class regex_token_iterator;

  typedef regex_token_iterator<const char*>             cregex_token_iterator;
  typedef regex_token_iterator<const wchar_t*>          wcregex_token_iterator;
  typedef regex_token_iterator<string::const_iterator>  sregex_token_iterator;
  typedef regex_token_iterator<wstring::const_iterator> wsregex_token_iterator;
} // namespace tr1
} // namespace std
```

7.5 Namespace `tr1::regex_constants` [tr.re.const]

1 The namespace `tr1::regex_constants` acts as a repository for the symbolic constants used by the regular expression library. This namespace provides three types, `syntax_option_type`, `match_flag_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;
  static const syntax_option_type icase;
  static const syntax_option_type nosubs;
  static const syntax_option_type optimize;
  static const syntax_option_type collate;
  static const syntax_option_type ECMAScript;
  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;
  } // namespace regex_constants
  } // namespace tr1
```

1 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 23. A valid value of type syntax_option_type shall have exactly one of the elements ECMAScript, basic, extended, awk, grep, egrep, set.

Table 23: syntax_option_type effects

Element	Effect(s) if set
icase	Specifies that matching of regular expressions against a character container sequence shall be performed without regard to case.
nosubs	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 match_-results structure.
optimize	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.
collate	Specifies that character ranges of the form “[a-b]” should be locale sensitive.
ECMAScript	Specifies that the grammar recognized by the regular expression engine is the same as that used by ECMAScript in ECMA-262 [7].
basic	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX basic regular expressions [11] in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Base Definitions and Headers, Section 9, Regular Expressions .
extended	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX extended regular expressions [11] in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Base Definitions and Headers, Section 9, Regular Expressions .
awk	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX utility awk in IEEE Std 1003.1-2001 [11].
grep	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX utility grep in IEEE Std 1003.1-2001 [11].
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 [11].

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 match_flag_type match_default = 0;
```

```
    static const match_flag_type match_not_bol;
    static const match_flag_type match_not_eol;
    static const match_flag_type match_not_bow;
    static const match_flag_type match_not_eow;
    static const match_flag_type match_any;
    static const match_flag_type match_not_null;
    static const match_flag_type match_continuous;
    static const match_flag_type match_prev_avail;
    static const match_flag_type format_default = 0;
    static const match_flag_type format_sed;
    static const match_flag_type format_no_copy;
    static const match_flag_type format_first_only;
  } // namespace regex_constants
  } // namespace tr1
```

1 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 [7], modified according to the effects listed in table 24 for any bitmask elements set.

Table 24: `regex_constants::match_flag_type` effects when obtaining a match against a character container sequence `[first,last)`.

Element	**Effect(s) if set**
match_not_bol	The first character in the sequence `[first, last)` is treated as though it is not at the beginning of a line, so the character `"^"` in the regular expression shall not match `[first, first)`.
match_not_eol	The last character in the sequence `[first, last)` is treated as though it is not at the end of a line, so the character `"$"` in the regular expression shall not match `[last, last)`.
match_not_bow	The expression `"\b"` is not matched against the sub-sequence [first,first).
match_not_eow	The expression `"\b"` should not be matched against the sub-sequence [last,last).
match_any	If more than one match is possible then any match is an acceptable result.
match_not_null	The expression does not match an empty sequence.
match_continuous	The expression only matchs a sub-sequence that begins at first .
match_prev_avail	`--first` is a valid iterator position. When this flag is set then the flags match_-not_bol and match_not_bow are ignored by the regular expression algorithms 7.11 and iterators 7.12.
format_default	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 [7], part 15.4.11 String.prototype.replace. 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.
format_sed	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 [11].

format_no_copy	During a search and replace operation, sections of the character container sequence being searched that do not match the regular expression are not copied to the output string.
format_first_only	When specified during a search and replace operation, only the first occurrence of the regular expression is replaced.

7.5.3 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_backref;
  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
```

1 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 25:

Table 25: `error_type` values in the C locale

Value	Error condition
`error_collate`	The expression contained an invalid collating element name.
`error_ctype`	The expression contained an invalid character class name.
`error_escape`	The expression contained an invalid escaped character, or a trailing escape.
`error_backref`	The expression contained an invalid backreference.
`error_brack`	The expression contained mismatched `[` and `]`.
`error_paren`	The expression contained mismatched `(` and `)`.
`error_brace`	The expression contained mismatched `{` and `}`
`error_badbrace`	The expression contained an invalid range in a `{}` expression.
`error_range`	The expression contained an invalid character range, for example `[b-a]`.
`error_space`	There was insufficient memory to convert the expression into a finite state machine.
`error_badrepeat`	One of `*?+{` was not preceded by a valid regular expression.

error_complexity	The complexity of an attempted match against a regular expression exceeded a pre-set level.
error_stack	There was insufficient memory to determine whether the regular expression could match the specified character sequence.

7.6 Class regex_error [tr.re.badexp]

```
class regex_error : public std::runtime_error
{
public:
   explicit regex_error(regex_constants::error_type ecode);
   regex_constants::error_type code() const;
};
```

1 The class regex_error defines the type of objects thrown as exceptions to report errors from the regular expression library.

```
regex_error(regex_constants::error_type ecode);
```

2 *Effects:* Constructs an object of class regex_error.

3 *Postcondition:*: ecode == code()

```
regex_constants::error_type code() const;
```

4 *Returns:* The error code that was passed to the constructor.

7.7 Class template regex_traits [tr.re.traits]

```
template <class charT>
struct regex_traits
{
public:
   typedef charT                         char_type;
   typedef std::basic_string<char_type> string_type;
   typedef std::locale                  locale_type;
   typedef bitmask_type                 char_class_type;

   regex_traits();
   static std::size_t length(const char_type* p);
   charT translate(charT c) const;
   charT translate_nocase(charT c) const;
   template <class ForwardIterator>
      string_type transform(ForwardIterator first, ForwardIterator last) const;
   template <class ForwardIterator>
      string_type transform_primary(ForwardIterator first, ForwardIterator last) const;
   template <class ForwardIterator>
      string_type lookup_collatename(ForwardIterator first, ForwardIterator last) const;
   template <class ForwardIterator>
```

```
      char_class_type lookup_classname(ForwardIterator first, ForwardIterator last) const;
   bool isctype(charT c, char_class_type f) const;
   int value(charT ch, int radix) const;
   locale_type imbue(locale_type l);
   locale_type getloc()const;
};
```

1 The class template `regex_traits` can be 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;
```

2 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`.

```
static std::size_t length(const char_type* p);
```

3 *Returns:* `char_traits<charT>::length(p);`

```
charT translate(charT c) const;
```

4 *Returns:* `(c)`.

```
charT translate_nocase(charT c) const;
```

5 *Returns:* `use_facet<ctype<charT> >(getloc()).tolower(c)`.

```
template <class ForwardIterator>
  string_type transform(ForwardIterator first, ForwardIterator last) const;
```

6 *Effects:*

```
string_type str(first, last);
return use_facet<collate<charT> >(getloc()).transform(&*str.begin(), &*str.end());
```

```
template <class ForwardIterator>
  string_type transform_primary(ForwardIterator first, ForwardIterator last) const;
```

7 *Effects:* if `typeid(use_facet<collate<charT> >) == typeid(collate_byname<charT>)` and the form of the sort key returned by `collate_byname<charT> ::transform(first, last)` is known and can be converted into a primary sort key then returns that key, otherwise returns an empty string.

```
template <class ForwardIterator>
  string_type lookup_collatename(ForwardIterator first, ForwardIterator last) const;
```

8 *Returns:* a sequence of one or more characters that represents the collating element consisting of the character sequence designated by the iterator range `[first, last)`. Returns an empty string if the character sequence is not a valid collating element.

```
template <class ForwardIterator>
  char_class_type lookup_classname(ForwardIterator first, ForwardIterator last) const;
```

9 *Returns:* an unspecified value that represents the character classification named by the character sequence designated by the iterator range [first, last). The value returned shall be independent of the case of the characters in the character sequence. If the name is not recognized then returns a value that compares equal to 0.

10 *Notes:* For regex_traits<char>, at least the names "d", "w", "s", "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper" and "xdigit" shall be recognized. For regex_traits<wchar_t>, at least the names L"d", L"w", L"s", L"alnum", L"alpha", L"blank", L"cntrl", L"digit", L"graph", L"lower", L"print", L"punct", L"space", L"upper" and L"xdigit" shall be recognized.

```
bool isctype(charT c, char_class_type f) const;
```

11 *Effects:* Determines if the character c is a member of the character classification represented by f.

12 *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 bitwise or'ed with the result of calling lookup_classname with an iterator pair that designates the character sequence "w" is not equal to 0 and c == '_', or if f bitwise or'ed with the result of calling lookup_classname with an iterator pair that designates the character sequence "blank" is not equal to 0 and c is one of an implementation-defined subset of the characters for which isspace(c, getloc()) returns true, otherwise returns false.

```
int value(charT ch, int radix) const;
```

13 *Precondition:* The value of *radix* shall be 8, 10, or 16.

14 *Returns:* the value represented by the digit *ch* in base *radix* if the character *ch* is a valid digit in base *radix*; otherwise returns -1.

```
locale_type imbue(locale_type loc);
```

15 *Effects:* Imbues this with a copy of the locale loc. [*Note:* calling imbue with a different locale than the one currently in use invalidates all cached data held by *this. — *end note*]

16 *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.

17 *Postcondition:* getloc() == loc.

```
locale_type getloc()const;
```

18 *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.

7.8 Class template basic_regex [tr.re.regex]

1 For a char-like type charT, specializations of class template basic_regex represent regular expressions constructed from character sequences of charT characters. In the rest of 7.8, charT denotes a given char-like type. Storage for a regular expression is allocated and freed as necessary by the member functions of class basic_regex.

2 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 templates as friends of `basic_regex` to achieve this *—end note*]

3 The regular expression grammar recognized by class template `basic_regex` is described in clause 7.13.

4 The functions described in this clause report errors by throwing exceptions of type `regex_error`.

```
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;

   // [7.8.1] constants
   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 = regex_constants::ECMAScript;
   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;

   // [7.8.2] construct/copy/destroy
   basic_regex();
   explicit basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript);
   basic_regex(const charT* p, size_t 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::ECMAScript);
   template <class InputIterator>
     basic_regex(InputIterator first, InputIterator last,
                 flag_type f = regex_constants::ECMAScript);

   ~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);

   // [7.8.3] assign
   basic_regex& assign(const basic_regex& that);
   basic_regex& assign(const charT* ptr, flag_type f = regex_constants::ECMAScript);
```

```
    basic_regex& assign(const charT* p, size_t 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::ECMAScript);
    template <class InputIterator>
      basic_regex& assign(InputIterator first, InputIterator last,
                          flag_type f = regex_constants::ECMAScript);

    // [7.8.4] const operations
    unsigned mark_count() const;
    flag_type flags() const;

    // [7.8.5] locale
    locale_type imbue(locale_type loc);
    locale_type getloc() const;

    // [7.8.6] swap
    void swap(basic_regex&) throw();
};
```

7.8.1 `basic_regex` constants **[tr.re.regex.const]**

```
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 = regex_constants::ECMAScript;
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;
```

1 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` 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();
```

1 *Effects:* Constructs an object of class `basic_regex` that does not match any character sequence.

```
basic_regex(const charT* p, flag_type f = regex_constants::ECMAScript);
```

2 *Requires:* *p* shall not be a null pointer.

3 *Throws:* `regex_error` if *p* is not a valid regular expression.

4 *Effects:* Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the array of `charT` of length `char_traits<charT>::length(p)` whose first element is designated by *p*, and interpreted according to the flags *f*.

5 *Postconditions:* `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```
basic_regex(const charT* p, size_t len, flag_type f);
```

6 *Requires: p* shall not be a null pointer.

7 *Throws:* `regex_error` if *p* is not a valid regular expression.

8 *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*.

9 *Postconditions:* `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```
basic_regex(const basic_regex& e);
```

10 *Effects:* Constructs an object of class `basic_regex` as a copy of the object `e`.

11 *Postconditions:* `flags()` and `mark_count()` return `e.flags()` and `e.mark_count()`, respectively.

```
template <class ST, class SA>
  basic_regex(const basic_string<charT, ST, SA>& s,
              flag_type f = regex_constants::ECMAScript);
```

12 *Throws:* `regex_error` if `s` is not a valid regular expression.

13 *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`.

14 *Postconditions:* `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```
template <class ForwardIterator>
  basic_regex(ForwardIterator first, ForwardIterator last,
              flag_type f = regex_constants::ECMAScript);
```

15 *Throws:* `regex_error` if the sequence *[first, last)* is not a valid regular expression.

16 *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 interpreted according to the flags specified in `f`.

17 *Postconditions:* `flags()` returns `f`. `mark_count()` returns the number of marked sub-expressions within the expression.

```
basic_regex& operator=(const basic_regex& e);
```

18 *Effects:* Returns the result of `assign(e)`.

```
basic_regex& operator=(const charT* ptr);
```

19 *Requires:* `ptr` shall not be a null pointer.

20 *Effects:* Returns the result of `assign(ptr)`.

```
template <class ST, class SA>
  basic_regex& operator=(const basic_string<charT, ST, SA>& p);
```

21 *Effects:* Returns the result of `assign(p)`.

7.8.3 `basic_regex` assign [tr.re.regex.assign]

```
basic_regex& assign(const basic_regex& that);
```

1 *Effects:* Copies `that` into `*this` and returns `*this`.

2 *Postconditions:* `flags()` and `mark_count()` return `that.flags()` and `that.mark_count()`, respectively.

```
basic_regex& assign(const charT* ptr, flag_type f = regex_constants::ECMAScript);
```

3 *Returns:* `assign(string_type(ptr), f)`.

```
basic_regex& assign(const charT* ptr, size_t len, flag_type f = regex_constants::ECMAScript);
```

4 *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::ECMAScript);
```

5 *Throws:* `regex_error` if `s` is not a valid regular expression.

6 *Returns:* `*this`.

7 *Effects:* Assigns the regular expression contained in the string `s`, interpreted according the flags specified in `f`. If an exception is thrown, `*this` is unchanged.

8 *Postconditions:* If an exception is thrown, `*this` is unchanged. Otherwise `flags()` returns `f`, and `mark_count()` returns the number of marked sub-expressions within the expression.

```
template <class InputIterator>
  basic_regex& assign(InputIterator first, InputIterator last,
                      flag_type f = regex_constants::ECMAScript);
```

9 *Requires:* The type `InputIterator` corresponds to the Input Iterator requirements (§24.1.1).

10 *Returns:* `assign(string_type(first, last), f)`.

7.8.4 `basic_regex` constant operations [tr.re.regex.operations]

```
unsigned mark_count() const;
```

1 *Effects:* Returns the number of marked sub-expressions within the regular expresion.

```
flag_type flags() const;
```

2 *Effects:* Returns a copy of the regular expression syntax flags that were passed to the object's constructor, or the last call to `assign`.

7.8.5 `basic_regex` locale [tr.re.regex.locale]

```
locale_type imbue(locale_type loc);
```

1 *Effects:* Returns the result of `traits_inst.imbue(loc)` where `traits_inst` is a (default initialized) instance of the template type argument `traits` stored within the object. After a call to `imbue` the `basic_regex` object does not match any character sequence.

```
locale_type getloc() const;
```

2 *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.

7.8.6 `basic_regex` swap [tr.re.regex.swap]

```
void swap(basic_regex& e) throw();
```

1 *Effects:* Swaps the contents of the two regular expressions.

2 *Postcondition:* `*this` contains the regular expression that was in `e`, `e` contains the regular expression that was in `*this`.

3 *Complexity:* constant time.

7.8.7 `basic_regex` non-member functions [tr.re.regex.nonmemb]

7.8.7.1 `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);
```

1 *Effects:* Calls `lhs.swap(rhs)`.

7.9 Class template `sub_match` [tr.re.submatch]

1 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();
```

1 *Returns:* `(matched ? distance(first, second) : 0)`.

```
operator basic_string<value_type>()const;
```

2 *Returns:* `matched ? basic_string<value_type>(first, second) : basic_string<value_type>()`.

```
basic_string<value_type> str()const;
```

3 *Returns:* `matched ? basic_string<value_type>(first, second) : basic_string<value_type>()`.

```
int compare(const sub_match& s)const;
```

4 *Returns:* `str().compare(s.str())`.

```
int compare(const basic_string<value_type>& s)const;
```

5 *Returns:* `str().compare(s)`.

```
int compare(const value_type* s)const;
```

6 *Returns:* `str().compare(s)`.

7.9.2 sub_match non-member operators [tr.re.submatch.op]

```
template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

1 *Returns:* `lhs.compare(rhs) == 0`.

```
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

2 *Returns:*`lhs.compare(rhs) != 0`.

```
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

3 *Returns:* `lhs.compare(rhs) < 0`.

```
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

4 *Returns:* `lhs.compare(rhs) <= 0`.

```
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

5 *Returns:* `lhs.compare(rhs) >= 0`.

```
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs, const sub_match<BiIter>& rhs);
```

6 *Returns:* `lhs.compare(rhs) > 0`.

```
template <class BiIter, class ST, class SA>
  bool operator==(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
                  const sub_match<BiIter>& rhs);
```

7 *Returns:* `lhs == rhs.str()`.

```
template <class BiIter, class ST, class SA>
  bool operator!=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
                  const sub_match<BiIter>& rhs);
```

8 *Returns:* `lhs != rhs.str()`.

```
template <class BiIter, class ST, class SA>
  bool operator<(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
                 const sub_match<BiIter>& rhs);
```

9 *Returns:* `lhs < rhs.str()`.

```
template <class BiIter, class ST, class SA>
  bool operator>(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
                 const sub_match<BiIter>& rhs);
```

10 *Returns:* `lhs > rhs.str()`.

```
template <class BiIter, class ST, class SA>
  bool operator>=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
                  const sub_match<BiIter>& rhs);
```

11 *Returns:* `lhs >= rhs.str()`.

```
template <class BiIter, class ST, class SA>
  bool operator<=(const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& lhs,
                  const sub_match<BiIter>& rhs);
```

12 *Returns:* `lhs <= rhs.str()`.

```
template <class BiIter, class ST, class SA>
  bool operator==(const sub_match<BiIter>& lhs,
                  const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

13 *Returns:* `lhs.str() == rhs`.

```
template <class BiIter, class ST, class SA>
  bool operator!=(const sub_match<BiIter>& lhs,
                  const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

14 *Returns:* `lhs.str() != rhs`.

```
template <class BiIter, class ST, class SA>
  bool operator<(const sub_match<BiIter>& lhs,
                 const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

15 *Returns:* `lhs.str() < rhs`.

```
template <class BiIter, class ST, class SA>
  bool operator>(const sub_match<BiIter>& lhs,
                 const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

16 *Returns:* `lhs.str() > rhs`.

```
template <class BiIter, class ST, class SA>
  bool operator>=(const sub_match<BiIter>& lhs,
                  const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

17 *Returns:* `lhs.str() >= rhs`.

```
template <class BiIter, class ST, class SA>
  bool operator<=(const sub_match<BiIter>& lhs,
                  const basic_string<typename iterator_traits<BiIter>::value_type, ST, SA>& rhs);
```

18 *Returns:* `lhs.str() <= rhs`.

```
template <class BiIter>
  bool operator==(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
```

19 *Returns:* `lhs == rhs.str()`.

```
template <class BiIter>
  bool operator!=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
```

20 *Returns:* `lhs != rhs.str()`.

```
template <class BiIter>
  bool operator<(typename iterator_traits<BiIter>::value_type const* lhs,
                 const sub_match<BiIter>& rhs);
```

21 *Returns:* `lhs < rhs.str()`.

```
template <class BiIter>
  bool operator>(typename iterator_traits<BiIter>::value_type const* lhs,
                 const sub_match<BiIter>& rhs);
```

22 *Returns:* `lhs > rhs.str()`.

```
template <class BiIter>
  bool operator>=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
```

23 *Returns:* `lhs >= rhs.str()`.

```
template <class BiIter>
  bool operator<=(typename iterator_traits<BiIter>::value_type const* lhs,
                  const sub_match<BiIter>& rhs);
```

24 *Returns:* `lhs <= rhs.str()`.

```
template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
```

25 *Returns:* `lhs.str() == rhs`.

```
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
```

26 *Returns:* `lhs.str() != rhs`.

```
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const* rhs);
```

27 *Returns:* `lhs.str() < rhs`.

```
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const* rhs);
```

28 *Returns:* `lhs.str() > rhs`.

```
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
```

29 *Returns:* `lhs.str() >= rhs`.

```
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const* rhs);
```

30 *Returns:* `lhs.str() <= rhs`.

```
template <class BiIter>
  bool operator==(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
```

31 *Returns:* `lhs == rhs.str()`.

```
template <class BiIter>
```

```
  bool operator!=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
```

32 *Returns:* `lhs != rhs.str()`.

```
template <class BiIter>
  bool operator<(typename iterator_traits<BiIter>::value_type const& lhs,
                 const sub_match<BiIter>& rhs);
```

33 *Returns:* `lhs < rhs.str()`.

```
template <class BiIter>
  bool operator>(typename iterator_traits<BiIter>::value_type const& lhs,
                 const sub_match<BiIter>& rhs);
```

34 *Returns:* `lhs > rhs.str()`.

```
template <class BiIter>
  bool operator>=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
```

35 *Returns:* `lhs >= rhs.str()`.

```
template <class BiIter>
  bool operator<=(typename iterator_traits<BiIter>::value_type const& lhs,
                  const sub_match<BiIter>& rhs);
```

36 *Returns:* `lhs <= rhs.str()`.

```
template <class BiIter>
  bool operator==(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
```

37 *Returns:* `lhs.str() == rhs`.

```
template <class BiIter>
  bool operator!=(const sub_match<BiIter>& lhs,
                  typename iterator_traits<BiIter>::value_type const& rhs);
```

38 *Returns:* `lhs.str() != rhs`.

```
template <class BiIter>
  bool operator<(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const& rhs);
```

39 *Returns:* `lhs.str() < rhs`.

```
template <class BiIter>
  bool operator>(const sub_match<BiIter>& lhs,
                 typename iterator_traits<BiIter>::value_type const& rhs);
```

40 *Returns:* `lhs.str() > rhs`.

```
template <class BiIter>
  bool operator>=(const sub_match<BiIter>& lhs,
```

```
                typename iterator_traits<BiIter>::value_type const& rhs);
```

41 *Returns:* `lhs.str() >= rhs`.

```
template <class BiIter>
  bool operator<=(const sub_match<BiIter>& lhs,
                typename iterator_traits<BiIter>::value_type const& rhs);
```

42 *Returns:* `lhs.str() <= rhs`.

```
template <class charT, class ST, class BiIter>
  basic_ostream<charT, ST>&
  operator<<(basic_ostream<charT, ST>& os, const sub_match<BiIter>& m);
```

43 *Returns:* `(os << m.str())`.

7.10 Class template `match_results` [tr.re.results]

1 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 template `match_results`.

2 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.

3 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. 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 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;

   // [7.10.1] 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();

    // [7.10.2] size
    size_type size() const;
    size_type max_size() const;
    bool empty() const;

    // [7.10.3] element access
    difference_type length(size_type sub = 0) const;
    difference_type position(size_type sub = 0) const;
    string_type str(size_type sub = 0) const;
    const_reference operator[](size_type n) const;

    const_reference prefix() const;
    const_reference suffix() const;
    const_iterator begin() const;
    const_iterator end() const;

    // [7.10.4] format
    template <class OutputIter>
      OutputIter
      format(OutputIter 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;

    // [7.10.5] allocator
    allocator_type get_allocator() const;

    // [7.10.6] swap
    void swap(match_results& that);
  };
```

7.10.1 `match_results` constructors [tr.re.results.const]

1 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());
```

2 *Effects:* Constructs an object of class `match_results`.

3 *Postconditions:* `size()` returns `0`. `str()` returns `basic_string<charT>()`.

```
match_results(const match_results& m);
```

4 *Effects:* Constructs an object of class `match_results`, as a copy of `m`.

```
match_results& operator=(const match_results& m);
```

5 *Effects:* Assigns `m` to `*this`. The postconditions of this function are indicated in Table 26

Table 26: `match_results` assignment operator effects

Element	Value
`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;
```

1 *Returns:* One plus the number of marked sub-expressions in the regular expression that was matched if `*this` represents the result of a successful match. Otherwise returns `0`. [*Note:* The state of a `match_results` object can be modified only by passing that object to `regex_match` or `regex_search`. Sections 7.11.2 and 7.11.3 specify the effects of those algorithms on their `match_results` arguments. *—end note*]

```
size_type max_size()const;
```

2 *Returns:* The maximum number of `sub_match` elements that can be stored in `*this`.

```
bool empty()const;
```

3 *Returns:* `size() == 0`.

7.10.3 `match_results` element access [tr.re.results.acc]

```
difference_type length(size_type sub = 0)const;
```

1 *Returns:* `(*this)[sub].length()`.

```
difference_type position(size_type sub = 0)const;
```

2 *Returns:* The distance from the start of the target sequence to `(*this)[sub].first`.

```
string_type str(size_type sub = 0)const;
```

3 *Returns:* `string_type((*this)[sub])`.

```
const_reference operator[](size_type n) const;
```

4 *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;
```

5 *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;
```

6 *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;
```

7 *Returns:* A starting iterator that enumerates over all the sub-expressions stored in `*this`.

```
const_iterator end()const;
```

8 *Returns:* A terminating iterator that enumerates over all the sub-expressions stored in `*this`.

7.10.4 `match_results` reformatting [tr.re.results.reform]

```
template <class OutputIter>
  OutputIter format(OutputIter out,
                    const string_type& fmt,
                    regex_constants::match_flag_type flags = regex_constants::format_default);
```

1 *Requires:* The type `OutputIter` conforms to the Output Iterator requirements [24.1.2].

2 *Effects:* Copies the character sequence `[fmt.begin(), fmt.end())` to OutputIter `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 [7], part 15.4.11 String.prototype.replace.

3 *Returns:* `out`.

```
string_type format(const string_type& fmt,
                   regex_constants::match_flag_type flags = regex_constants::format_default);
```

4 *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 [7], part 15.4.11, String.prototype.replace.

7.10.5 `match_results` allocator [tr.re.results.all]

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

1 *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);
```

1 *Effects:* Swaps the contents of the two sequences.

2 *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`.

3 *Complexity:* constant time.

```
template <class BidirectionalIterator, class Allocator>
  void swap(match_results<BidirectionalIterator, Allocator>& m1,
            match_results<BidirectionalIterator, Allocator>& m2);
```

4 *Effects:* `m1.swap(m2)`.

7.11 Regular expression algorithms [tr.re.alg]

7.11.1 exceptions [tr.re.except]

1 The algorithms described in this subclause may throw an exception of type `regex_error`. If such an exception `e` is thrown, `e.code()` shall return either `regex_constants::error_complexity` or `regex_constants::error_stack`.

7.11.2 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);
```

1 *Requires:* Type `BidirectionalIterator` satisfies the requirements of a Bidirectional Iterator (§24.1.4).

2 *Effects:* Determines whether there is a match between the regular expression `e`, and all of the character sequence `[first, last)`, where the parameter `flags` is used to control how the expression is matched against the character sequence. Returns `true` if such a match exists, `false` otherwise.

3 *Postconditions:* If the function returns `false`, then the effect on parameter `m` is unspecified except that `m.size()` returns `0` and `m.empty()` returns `true`. Otherwise the effects on parameter `m` are given in table 27.

Table 27: Effects of `regex_match` algorithm

Element	Value
`m.size()`	`1 + e.mark_count()`
`m.empty()`	false
`m.prefix().first`	first
`m.prefix().last`	first
`m.prefix().matched`	false
`m.suffix().first`	last

`m.suffix().last`	last
`m.suffix().matched`	false
`m[0].first`	first
`m[0].second`	last
`m[0].matched`	`true` if a full match was found, and `false` otherwise.
`m[n].first`	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 `last`.
`m[n].second`	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 `last`.
`m[n].matched`	For all integers `n < m.size()`, `true` if sub-expression `n` participated in the match, `false` 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);
```

4 *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);
```

5 *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,
                                 Allocator>& m,
                   const basic_regex<charT, traits>& e,
                   regex_constants::match_flag_type flags = regex_constants::match_default);
```

6 *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);
```

7 *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);
```

8 *Returns:* `regex_match(s.begin(), s.end(), e, flags)`.

7.11.3 `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);
```

1 *Requires:* Type `BidirectionalIterator` meets the requirements of a Bidirectional Iterator (24.1.4).

2 *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.

3 *Postconditions:* If the function returns `false`, then the effect on parameter `m` is unspecified except that `m.size()` returns `0` and `m.empty()` returns `true`. Otherwise the effects on parameter `m` are given in table 28.

Table 28: Effects of `regex_search` algorithm

Element	Value
`m.size()`	`1 + e.mark_count()`
`m.empty()`	`false`
`m.prefix().first`	`first`
`m.prefix().last`	`m[0].first`
`m.prefix().matched`	`m.prefix().first != m.prefix().second`
`m.suffix().first`	`m[0].second`
`m.suffix().last`	`last`
`m.suffix().matched`	`m.suffix().first != m.suffix().second`
`m[0].first`	The start of the sequence of characters that matched the regular expression
`m[0].second`	The end of the sequence of characters that matched the regular expression
`m[0].matched`	`true` if a match was found, and `false` otherwise.
`m[n].first`	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 `last`.
`m[n].second`	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 `last` .
`m[n].matched`	For all integers `n < m.size()`, `true` if sub-expression `n` participated in the match, `false` 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);
```

4 *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,
                                  Allocator>& m,
                    const basic_regex<charT, traits>& e,
                    regex_constants::match_flag_type flags = regex_constants::match_default);
```

5 *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);
```

6 *Effects:* Behaves “as if” by constructing an object `what` of type `match_results<iterator>` 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);
```

7 *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);
```

8 *Returns:* `regex_search(s.begin(), s.end(), e, flags)`.

7.11.4 `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);
```

1 *Effects:* Constructs a `regex_iterator` object `i` as if by `regex_iterator<BidirectionalIterator, charT, traits> 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.

2 *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);
```

3 *Effects:* Constructs an empty string `result` of type `basic_string<charT>`, 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]

1 The class template `regex_iterator` is an iterator adapter. It represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence. A `regex_iterator` uses `regex_search` to find successive regular expression matches within the sequence from which it was constructed. After the iterator 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 default constructor 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 iterator is not defined. For any other iterator value a const `match_results<BidirectionalIterator>&` is returned. The result of `operator->` on an end-of-sequence iterator 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 = typename 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 std::ptrdiff_t                       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:
    // these members are shown for exposition only:
    BidirectionalIterator                begin;
    BidirectionalIterator                end;
    const regex_type*                    pregex;
    regex_constants::match_flag_type     flags;
    match_results<BidirectionalIterator> match;
  };
```

2 A `regex_iterator` object that is not an end-of-sequence iterator holds a *zero-length match* if `match[0].matched == true` and `match[0].first == match[0].second`. [*Note:* this occurs when the part of the regular expression that matched consists only of an assertion (such as `'^'`, `'$'`, `'\b'`, `'\B'`). —*end note*]

7.12.1.1 `regex_iterator` constructors [tr.re.regiter.cnstr]

```
regex_iterator();
```

1 *Effects:* Constructs an end-of-sequence iterator.

```
regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
               const regex_type& re,
               regex_constants::match_flag_type m = regex_constants::match_default);
```

2 *Effects:* Initializes `begin` and `end` to point to the beginning and the end of the target sequence, sets `pregex` to `&re`, sets `flags` to `f`, then calls `regex_search(begin, end, match, *pregex, flags)`. If this call returns `false` the constructor sets `*this` to the end-of-sequence iterator.

7.12.1.2 `regex_iterator` comparisons [tr.re.regiter.comp]

```
bool operator==(const regex_iterator& right);
```

1 *Returns:* `true` if `*this` and `right` are both end-of-sequence iterators or if `begin == right.begin`, `end == right.end`, `pregex == right.pregex`, `flags == right.flags`, and `match[0] == right.match[0]`, otherwise `false`.

```
bool operator!=(const regex_iterator& right);
```

2 *Returns:* `!(*this == right)`.

7.12.1.3 `regex_iterator` dereference [tr.re.regiter.deref]

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

1 *Returns:* `match`.

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

2 *Returns:* `&match`.

7.12.1.4 `regex_iterator` increment [tr.re.regiter.incr]

```
regex_iterator& operator++();
```

1 *Effects:* Constructs a local variable `start` of type `BidirectionalIterator` and initializes it with the value of `match[0].second`.

2 If the iterator holds a zero-length match and `start == end` the operator sets `*this` to the end-of-sequence iterator and returns `*this`.

3 Otherwise, if the iterator holds a zero-length match the operator calls `regex_search(start, end, match, *pregex, flags | regex_constants::match_not_null | regex_constants::match_continuous)`. If the call returns `true` the operator returns `*this`. Otherwise the operator increments `start` and continues as if the most recent match was not a zero-length match.

4 If the most recent match was not a zero-length match, the operator sets `flags` to `flags | regex_constants ::match_prev_avail` and calls `regex_search(start, end, match, *pregex, flags)`. If the call returns `false` the iterator sets `*this` to the end-of-sequence iterator. The iterator then returns `*this`.

5 In all cases in which the call to `regex_search` returns `true`, `match.prefix().first` shall be equal to the previous value of `match[0].second`, and for each index `i` in the half-open range `[0, match.size())` for which `match[i].matched` is true, `match[i].position()` shall return `distance(begin, match[i].first])`.

6 [*Note:* this means that `match[i].position()` gives the offset from the beginning of the target sequence, which is often not the same as the offset from the sequence passed in the call to `regex_search`. *—end note*]

7 It is unspecified how the implementation makes these adjustments.

8 [*Note:* this means that a compiler may call an implementation-specific search function, in which case a user-defined specialization of `regex_search` will not be called. *—end note*]

```
regex_iterator operator++(int);
```

9 *Effects:*

```
regex_iterator tmp = *this;
++(*this);
return tmp;
```

7.12.2 Class template `regex_token_iterator` [tr.re.tokiter]

1 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 sub-expressions for each match found. Each position enumerated by the iterator is a `sub_match` class template instance that represents what matched a particular sub-expression within the regular expression.

2 When class `regex_token_iterator` is used to enumerate a single sub-expression with index -1 the iterator performs field splitting: that is to say it enumerates one sub-expression for each section of the character container sequence that does not match the regular expression specified.

3 After it is constructed, the iterator finds and stores a value `match_results<BidirectionalIterator> position` and sets the internal count `N` to zero. It also maintains a sequence `subs` which contains a list of the sub-expressions which will be enumerated. Every time `operator++` is used the count `N` is incremented; if `N` exceeds or equals `subs.size()`, then the iterator increments member `position` and sets count `N` to zero.

4 If the end of sequence is reached (`position` is equal to the end of sequence iterator), 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 sub-expression 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 sub-expression.

5 The default constructor 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 iterator is not defined. For any other iterator value a `const sub_match<BidirectionalIterator>&` is returned. The result of `operator->` on an end-of-sequence iterator is not defined. For any other iterator value a `const sub_match<BidirectionalIterator>*` is returned.

6 It is impossible to store things into `regex_iterator`s. 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 = typename iterator_traits<BidirectionalIterator>::value_type,
          class traits = regex_traits<charT> >
class regex_token_iterator
{
public:
   typedef basic_regex<charT, traits>       regex_type;
   typedef sub_match<BidirectionalIterator> value_type;
   typedef std::ptrdiff_t                   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:          // data members for exposition only:
    typedef regex_iterator<BidirectionalIterator, charT, traits> position_iterator;
    position_iterator position;
    const value_type *result;
    value_type suffix;
    std::size_t N;
    std::vector<int> subs;
  };
```

7 A *suffix iterator* is a `regex_token_iterator` object that points to a final sequence of characters at the end of the target sequence. In a suffix iterator the member `result` holds a pointer to the data member `suffix`, the value of the member `suffix.match` is `true`, `suffix.first` points to the beginning of the final sequence, and `suffix.second` points to the end of the final sequence.

8 [*Note:* for a suffix iterator, data member `suffix.first` is the same as the end of the last match found, and `suffix.second` is the same as the end of the target sequence — *end note*].

9 The *current match* is `(*position).prefix()` if `subs[N] == -1`, or `(*position)[subs[N]]` for any other value of `subs[N]`.

7.12.2.1 `regex_token_iterator` constructors [tr.re.tokiter.cnstr]

```
regex_token_iterator();
```

1 *Effects:* Constructs the end-of-sequence 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)[R],
                     regex_constants::match_flag_type m = regex_constants::match_default);
```

2 *Effects:* The first constructor initializes the member `subs` to hold the single value `submatch`. The second constructor initializes the member `subs` to hold a copy of the argument `submatches`. The third constructor initializes the member `subs` to hold a copy of the sequence of integer values pointed to by the iterator range `[&submatches, &submatches + R)`.

3 Each constructor then sets `N` to 0, and `position` to `position_iterator(a, b, re, f)`. If `position` is not an end-of-sequence iterator the constructor sets `result` to the address of the current match. Otherwise if any of the values stored in `subs` is equal to -1 the constructor sets `*this` to a suffix iterator that points to the range `[a, b)`, otherwise the constructor sets `*this` to an end-of-sequence iterator.

7.12.2.2 `regex_token_iterator` comparisons [tr.re.tokiter.comp]

```
bool operator==(const regex_token_iterator& right);
```

1 *Returns:* `true` if `*this` and `right` are both end-of-sequence iterators, or if `*this` and `right` are both suffix iterators and `suffix == right.suffix`; otherwise returns `false` if `*this` or `right` is an end-of-sequence iterator or a suffix iterator. Otherwise returns `true` if `position == right.position`, `N == right.N`, and `subs == right.subs`. Otherwise returns `false`.

```
bool operator!=(const regex_token_iterator& right);
```

2 *Returns:* `!(*this == right)`.

7.12.2.3 `regex_token_iterator` dereference [tr.re.tokiter.deref]

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

1 *Returns:* `*result`.

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

2 *Returns:* `result`.

7.12.2.4 `regex_token_iterator` increment [tr.re.tokiter.incr]

```
regex_token_iterator& operator++();
```

1 *Effects:* Constructs a local variable `prev` of type `position_iterator`, initialized with the value of `position`.

2 If `*this` is a suffix iterator, sets `*this` to an end-of-sequence iterator.

3 Otherwise, if `N + 1 < subs.size()`, increments `N` and sets `result` to the address of the current match.

4 Otherwise, sets `N` to 0 and increments `position`. If `position` is not an end-of-sequence iterator the operator sets `result` to the address of the current match.

5 Otherwise, if any of the values stored in `subs` is equal to -1 and `prev->suffix().length()` is not 0 the operator sets `*this` to a suffix iterator that points to the range `[prev->suffix().first, prev->suffix().second)`.

6 Otherwise, sets `*this` to an end-of-sequence iterator.

Returns: `*this`

```
regex_token_iterator& operator++(int);
```

7 *Effects:* Constructs a copy `tmp` of `*this`, then calls `++(*this)`.

8 *Returns:* `tmp`.

7.13 Modified ECMAScript regular expression grammar [tr.re.grammar]

1 The regular expression grammar recognized by class template `basic_regex` is that specified by ECMA-262 [7], except as specified below.

2 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.

3 The following productions within the ECMAScript grammar are modified as follows:

```
CharacterClass ::
  [ [lookahead ∉ {^}] ClassRanges ]
  [ ^ ClassRanges ]

ClassAtom ::
  -
  ClassAtomNoDash
  ClassAtomExClass
  ClassAtomCollatingElement
  ClassAtomEquivalence
```

4 The following new productions are then added:

```
ClassAtomExClass ::
  [: ClassName :]

ClassAtomCollatingElement ::
  [. ClassName .]

ClassAtomEquivalence ::
  [= ClassName =]

ClassName ::
  ClassNameCharacter
  ClassNameCharacter ClassName
```

```
ClassNameCharacter ::
  SourceCharacter but not one of "." "=" ":"
```

5 The productions `ClassAtomExClass`, `ClassAtomCollatingElement` and `ClassAtomEquivalence` provide functionality equivalent to that of the same features in POSIX regular expressions [11].

6 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 23.

7 A `ClassName` production, when used in `ClassAtomExClass`, is not valid if `traits_inst.lookup_classname` returns zero for that name. 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:]]`

8 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.

9 The results from multiple calls to `traits_inst.lookup_classname` can be bitwise OR'ed together and subsequently passed to `traist_inst.isctype`.

10 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.

11 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 `regex_error`.

12 If the *CV* of a *UnicodeEscapeSequence* is greater than the largest value that can be held in an object of type `charT` the translator shall throw an exception object of type `regex_error`. [*Note:* this means that values of the form `"uxxxx"` that do not fit in a character are invalid. *—end note*]

13 Where the regular expression grammar requires the conversion of a sequence of characters to an integral value, this is accomplished by calling `traits_inst.value`.

14 The behavior of the internal finite state machine representation when used to match a sequence of characters is as described in ECMAScript [7]. 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 the following rules:

  1. if `(flags() & regex_constants::icase)` the two characters are equal if `traits_inst.translate_nocase(c) == traits_inst.translate_nocase(d)`;

  2. otherwise, if `flags() & regex_constants::collate)` the two characters are equal if `traits_inst.translate(c) == traits_inst.translate(d)`;

  3. otherwise, the two characters are equal if `c == d`.

— 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,
    flags() & icase ? traits_inst.translate_nocase(c1) : traits_inst.translate(c1);
string_type str2 = string_type(1,
    flags() & icase ? traits_inst.translate_nocase(c2) : traits_inst.translate(c2);
string_type str = string_type(1,
    flags() & icase ? traits_inst.translate_nocase(c) : traits_inst.translate(c);
return traits_inst.transform(str1.begin(), str1.end())
        <= traits_inst.transform(str.begin(), str.end())
    && traits_inst.transform(str.begin(), str.end())
        <= traits_inst.transform(str2.begin(), str2.end());
```

— 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 a character class designated by an iterator range `[first, last)` if `traits_inst.isctype(c, traits_inst.lookup_classname(first, last))` is `true`.

8 C compatibility [tr.c99]

1 This clause describes additions designed to bring the Standard C++ library in closer agreement with the library described in ISO/IEC 9899:1999 Standard C, as corrected through 2003 (hereafter C99, for short).

2 To avoid the need for language changes:

1. Any use of the type `long long` in C99 is replaced in this clause by the type `_Longlong`, which behaves like a signed integer type that occupies at least 64 bits. Any header containing a declaration that uses `_Longlong` shall provide an idempotent definition of `_Longlong`.
2. Any use of the type `unsigned long long` in C99 is replaced in this clause by the type `_ULonglong`, which behaves like an unsigned integer type with the same number of bits as `_Longlong`. Any header containing a declaration that uses `_ULonglong` shall provide an idempotent definition of `_ULonglong`.
3. Any use of the type qualifier `restrict` in C99 shall be omitted in this clause.

8.1 Additions to header `<complex>` [tr.c99.cmplx]

8.1.1 Synopsis [tr.c99.cmplx.syn]

```
namespace std {
namespace tr1 {
  template<class T> complex<T> acos(complex<T>& x);
  template<class T> complex<T> asin(complex<T>& x);
  template<class T> complex<T> atan(complex<T>& x);

  template<class T> complex<T> acosh(complex<T>& x);
  template<class T> complex<T> asinh(complex<T>& x);
  template<class T> complex<T> atanh(complex<T>& x);
  template<class T> complex<T> fabs(complex<T>& x);
} // namespace tr1
} // namespace std
```

8.1.2 Function acos [tr.c99.cmplx.acos]

1 *Effects:* Behaves the same as C99 function `cacos`, defined in subclause 7.3.5.1.

8.1.3 Function asin [tr.c99.cmplx.asin]

1 *Effects:* Behaves the same as C99 function `casin`, defined in subclause 7.3.5.2.

8.1.4 Function atan [tr.c99.cmplx.atan]

1 *Effects:* Behaves the same as C99 function `catan`, defined in subclause 7.3.5.3.

8.1.5 Function acosh [tr.c99.cmplx.acosh]

1 *Effects:* Behaves the same as C99 function `cacosh`, defined in subclause 7.3.6.1.

8.1.6 Function asinh [tr.c99.cmplx.asinh]

1 *Effects:* Behaves the same as C99 function `casinh`, defined in subclause 7.3.6.2.

8.1.7 Function atanh [tr.c99.cmplx.atanh]

1 *Effects:* Behaves the same as C99 function `catanh`, defined in subclause 7.3.6.3.

8.1.8 Function fabs [tr.c99.cmplx.fabs]

1 *Effects:* Behaves the same as C99 function `cabs`, defined in subclause 7.3.8.1.

8.1.9 Additional Overloads [tr.c99.cmplx.over]

1 The following function templates shall have additional overloads:

```
arg         norm
conj        polar
imag        real
```

2 The additional overloads shall be sufficient to ensure:

1. If the argument has type `long double`, then it is effectively cast to `complex<long double>`.
2. Otherwise, if the argument has type `double` or an integer type, then it is effectively cast to `complex<double>`.
3. Otherwise, if the argument has type `float`, then it is effectively cast to `complex<float>`.

3 Function template `pow` shall have additional overloads sufficient to ensure, for a call with at least one argument of type `complex<T>`:

1. If either argument has type `complex<long double>` or type `long double`, then both arguments are effectively cast to `complex<long double>`.
2. Otherwise, if either argument has type `complex<double>`, `double`, or an integer type, then both arguments are effectively cast to `complex<double>`.
3. Otherwise, if either argument has type `complex<float>` or `float`, then both arguments are effectively cast to `complex<float>`.

8.2 Header <ccomplex> [tr.c99.ccmplx]

1 The header behaves as if it simply includes the header <complex>.

8.3 Header <complex.h> [tr.c99.cmplxh]

1 The header behaves as if it includes the header <ccomplex>, and provides sufficient *using* declarations to declare in the global namespace all function and type names declared or defined in the neader <complex>.

8.4 Additions to header <cctype> [tr.c99.cctype]

8.4.1 Synopsis [tr.c99.cctype.syn]

```
namespace std {
namespace tr1 {
  int isblank(int ch);
} // namespace tr1
} // namespace std
```

8.4.2 Function isblank [tr.c99.cctype.blank]

1 Function isblank behaves the same as C99 function isblank, defined in subclause 7.4.1.3.

8.5 Additions to header <ctype.h> [tr.c99.ctypeh]

1 The header behaves as if it includes the header <cctype>, and provides sufficient additional *using* declarations to declare in the global namespace the additional function name declared in the header <cctype>.

8.6 Header <cfenv> [tr.c99.cfenv]

8.6.1 Synopsis [tr.c99.cfenv.syn]

```
namespace std {
namespace tr1 {
  // types
  typedef object type  fenv_t;
  typedef integer type fexcept_t;

  // functions
  int feclearexcept(int except);
  int fegetexceptflag(fexcept_t *pflag, int except);
  int feraiseexcept(int except);
  int fesetexceptflag(const fexcept_t *pflag, int except);
  int fetestexcept(int except);

  int fegetround(void);
  int fesetround(int mode);

  int fegetenv(fenv_t *penv);
  int feholdexcept(fenv_t *penv);
  int fesetenv(const fenv_t *penv);
  int feupdateenv(const fenv_t *penv);
} // namespace tr1
```

```
} // namespace std
```

1 The header also defines the macros:

```
FE_ALL_EXCEPT
FE_DIVBYZERO
FE_INEXACT
FE_INVALID
FE_OVERFLOW
FE_UNDERFLOW

FE_DOWNWARD
FE_TONEAREST
FE_TOWARDZERO
FE_UPWARD

FE_DFL_ENV
```

8.6.2 Definitions [tr.c99.cfenv.def]

1 The header defines all functions, types, and macros the same as C99 subclause 7.6.

8.7 Header `<fenv.h>` [tr.c99.fenv]

1 The header behaves as if it includes the header `<cfenv>`, and provides sufficient *using* declarations to declare in the global namespace all function and type names declared or defined in the header `<cfenv>`.

8.8 Additions to header `<cfloat>` [tr.c99.cfloat]

1 The header defines the macros:

```
DECIMAL_DIG
FLT_EVAL_METHOD
```

the same as C99 subclause 5.2.4.2.2.

8.9 Additions to header `<float.h>` [tr.c99.floath]

1 The header behaves as if it defines the additional macros defined in `<cfloat>` by including the header `<cfloat>`.

8.10 Additions to header `<ios>` [tr.c99.ios]

8.10.1 Synopsis [tr.c99.ios.syn]

```
namespace std {
namespace tr1 {
  ios_base& hexfloat(ios_base& str);
} // namespace tr1
} // namespace std
```

8.10.2 Function `hexfloat` [tr.c99.ios.hex]

```
ios_base& hexfloat(ios_base& str);
```

1 *Effects:* Calls `str.setf(ios_base::fixed | ios_base::scientific, ios_base::floatfield)`.

2 *Returns:* `str`.

3 [*Note*: Adding the format flag hexfloat to class `ios_base` cannot be done without invading namespace std, so only the named manipulator is provided. Note also that the more obvious use of ios_base::hex to specify hexadecimal floating-point format would change the meaning of existing well defined programs. C++2003 gives no meaning to the combination of fixed and scientific. *—end note*]

8.11 Header `<cinttypes>` [tr.c99.cinttypes]

8.11.1 Synopsis [tr.c99.cinttypes.syn]

```
#include <cstdint>

namespace std {
namespace tr1 {
  // types
  typedef struct {
    intmax_t quot, rem;
  } imaxdiv_t;

  // functions
  intmax_t imaxabs(intmax_t i);
  intmax_t abs(intmax_t i);

  imaxdiv_t imaxdiv(intmax_t numer, intmax_t denom);
  imaxdiv_t div(intmax_t numer, intmax_t denom);

  intmax_t strtoimax(const char * s, char **endptr, int base);
  uintmax_t strtoumax(const char *s, char **endptr, int base);
  intmax_t wcstoimax(const wchar_t *s, wchar_t **endptr, int base);
  uintmax_t wcstoumax(const wchar_t *s, wchar_t **endptr, int base);
} // namespace tr1
} // namespace std
```

1 The header also defines numerous macros of the form:

```
  PRI{d i o u x X}[FAST LEAST]{8 16 32 64}
  PRI{d i o u x X}{MAX PTR}
  SCN{d i o u x}[FAST LEAST]{8 16 32 64}
  SCN{d i o u x}{MAX PTR}
```

8.11.2 Definitions [tr.c99.cinttypes.def]

1 The header defines all functions, types, and macros the same as C99 subclause 7.8.

8.12 Header `<inttypes.h>` [tr.c99.inttypesh]

1 The header behaves as if it includes the header `<cinttypes>`, and provides sufficient *using* declarations to declare in

the global namespace all function and type names declared or defined in the header `<cinttypes>`.

8.13 Additions to header `<climits>` [tr.c99.climits]

1 The header defines the macros:

```
LLONG_MIN
LLONG_MAX
ULLONG_MAX
```

the same as C99 subclause 5.2.4.2.1.

8.14 Additions to header `<limits.h>` [tr.c99.limitsh]

1 The header behaves as if it defines the additional macros defined in `<climits>` by including the header `<climits>`.

8.15 Additions to header `<locale>` [tr.c99.locale]

1 In subclause 22.2.2.2.2, Table 58 Floating-point conversions, after the line:

```
floatfield == ios_base::scientific     %E
```

add the two lines:

```
floatfield == ios_base::fixed | ios_base::scientific && !uppercase    %a
floatfield == ios_base::fixed | ios_base::scientific    %A
```

2 [*Note*: The additional requirements on print and scan functions, later in this clause, ensure that the print functions generate hexadecimal floating-point fields with a `%a` or `%A` conversion specifier, and that the scan functions match hexadecimal floating-point fields with a `%g` conversion specifier. *—end note*]

8.16 Additions to header `<cmath>` [tr.c99.cmath]

8.16.1 Synopsis [tr.c99.cmath.syn]

```
namespace std {
namespace tr1 {
  // types
  typedef floating-type double_t;
  typedef floating-type float_t;

  // functions
  double acosh(double x);
  float acoshf(float x);
  long double acoshl(long double x);

  double asinh(double x);
  float asinhf(float x);
  long double asinhl(long double x);

  double atanh(double x);
  float atanhf(float x);
  long double atanhl(long double x);
```

```
double cbrt(double x);
float cbrtf(float x);
long double cbrtl(long double x);

double copysign(double x, double y);
float copysignf(float x, float y);
long double copysignl(long double x, long double y);

double erf(double x);
float erff(float x);
long double erfl(long double x);

double erfc(double x);
float erfcf(float x);
long double erfcl(long double x);

double exp2(double x);
float exp2f(float x);
long double exp2l(long double x);

double expm1(double x);
float expm1f(float x);
long double expm1l(long double x);

double fdim(double x, double y);
float fdimf(float x, float y);
long double fdiml(long double x, long double y);

double fma(double x, double y, double z);
float fmaf(float x, float y, float z);
long double fmal(long double x, long double y, long double z);

double fmax(double x, double y);
float fmaxf(float x, float y);
long double fmaxl(long double x, long double y);

double fmin(double x, double y);
float fminf(float x, float y);
long double fminl(long double x, long double y);

double hypot(double x, double y);
float hypotf(float x, float y);
long double hypotl(long double x, long double y);

int ilogb(double x);
int ilogbf(float x);
int ilogbl(long double x);

double lgamma(double x);
```

```
float lgammaf(float x);
long double lgammal(long double x);

long long llrint(double x);
long long llrintf(float x);
long long llrintl(long double x);

long long llround(double x);
long long llroundf(float x);
long long llroundl(long double x);

double log1p(double x);
float log1pf(float x);
long double log1pl(long double x);

double log2(double x);
float log2f(float x);
long double log2l(long double x);

double logb(double x);
float logbf(float x);
long double logbl(long double x);

long lrint(double x);
long lrintf(float x);
long lrintl(long double x);

long lround(double x);
long lroundf(float x);
long lroundl(long double x);

double nan(const char *str);
float nanf(const char *str);
long double nanl(const char *str);

double nearbyint(double x);
float nearbyintf(float x);
long double nearbyintl(long double x);

double nextafter(double x, double y);
float nextafterf(float x, float y);
long double nextafterl(long double x, long double y);

double nexttoward(double x, long double y);
float nexttowardf(float x, long double y);
long double nexttowardl(long double x, long double y);

double remainder(double x, double y);
float remainderf(float x, float y);
long double remainderl(long double x, long double y);
```

```
    double remquo(double x, double y, int *pquo);
    float remquof(float x, float y, int *pquo);
    long double remquol(long double x, long double y, int *pquo);

    double rint(double x);
    float rintf(float x);
    long double rintl(long double x);

    double round(double x);
    float roundf(float x);
    long double roundl(long double x);

    double scalbln(double x, long ex);
    float scalblnf(float x, long ex);
    long double scalblnl(long double x, long ex);

    double scalbn(double x, int ex);
    float scalbnf(float x, int ex);
    long double scalbnl(long double x, int ex);

    double tgamma(double x);
    float tgammaf(float x);
    long double tgammal(long double x);

    double trunc(double x);
    float truncf(float x);
    long double truncl(long double x);

    // C99 macros defined as C++ templates
    template<class T> bool signbit(T x);

    template<class T> int fpclassify(T x);

    template<class T> bool isfinite(T x);
    template<class T> bool isinf(T x);
    template<class T> bool isnan(T x);
    template<class T> bool isnormal(T x);

    template<class T> bool isgreater(T x, T y);
    template<class T> bool isgreaterequal(T x, T y);
    template<class T> bool isless(T x, T y);
    template<class T> bool islessequal(T x, T y);
    template<class T> bool islessgreater(T x, T y);
    template<class T> bool isunordered(T x, T y);
  } // namespace tr1
  } // namespace std
```

1 The header also defines the macros:

```
FP_FAST_FMA
FP_FAST_FMAF
FP_FAST_FMAL

FP_ILOGB0
FP_ILOGBNAN

FP_INFINITE
FP_NAN
FP_NORMAL
FP_SUBNORMAL
FP_ZERO

HUGE_VALF
HUGE_VALL

INFINITY
NAN

MATH_ERRNO
MATH_ERREXCEPT
math_errhandling
```

8.16.2 Definitions [tr.c99.cmath.def]

1 The header defines all of the above (non-template) functions, types, and macros the same as C99 subclause 7.12.

8.16.3 Function template definitions [tr.c99.cmath.tmpl]

1 The function templates:

```
template<class T> bool signbit(T x);

template<class T> int fpclassify(T x);

template<class T> bool isfinite(T x);
template<class T> bool isinf(T x);
template<class T> bool isnan(T x);
template<class T> bool isnormal(T x);

template<class T> bool isgreater(T x, T y);
template<class T> bool isgreaterequal(T x, T y);
template<class T> bool isless(T x, T y);
template<class T> bool islessequal(T x, T y);
template<class T> bool islessgreater(T x, T y);
template<class T> bool isunordered(T x, T y);
```

behave the same as C99 macros with corresponding names defined in C99 subclause 7.12.3 Classification macros and

C99 subclause 7.12.14 Comparison macros.

8.16.4 Additional overloads [tr.c99.cmath.over]

1 The following functions shall have additional overloads:

```
acos        ilogb
acosh       ldexp
asin        lgamma
asinh       llrint
atan        llround
atan2       log10
atanh       log1p
cbrt        logb
ceil        lrint
copysign    lround
cos         nearbyint
cosh        nextafter
erf         nexttoward
erfc        pow
exp         remainder
exp2        remquo
expm1       rint
fabs        round
fdim        scalbln
floor       scalbn
fma         sin
fmax        sinh
fmin        sqrt
fmod        tan
frexp       tanh
hypot       tgamma
log         trunc
```

2 Each of the above functions shall have an overload with all parameters of type `double` replaced with `long double`. If the return type of the above function is type `double`, the return type of the overload shall be `long double`.

3 Each of the above functions shall also have an overload with all parameters of type `double` replaced with `float`. If the return type of the above function is type `double`, the return type of the overload shall be `float`.

4 Moreover, there shall be additional overloads sufficient to ensure:

1. If any argument corresponding to a `double` parameter has type `long double`, then all arguments corresponding to `double` parameters are effectively cast to `long double`.

2. Otherwise, if any argument corresponding to a `double` parameter has type `double` or an integer type, then all arguments corresponding to `double` parameters are effectively cast to `double`.

3. Otherwise, all arguments corresponding to `double` parameters are effectively cast to `float`.

8.17 Additions to header `<math.h>` [tr.c99.mathh]

1 The header behaves as if it includes the header `<cmath>`, and provides sufficient additional *using* declarations to declare in the global namespace all the additional function template, function, and type names declared or defined in the header `<cmath>`.

8.18 Additions to header `<cstdarg>` [tr.c99.cstdarg]

1 Add the function macro:

```
va_copy(va_list dest, va_list src)
```

as defined in C99 subclause 7.15.1.2.

8.19 Additions to header `<stdarg.h>` [tr.c99.stdargh]

1 The header behaves as if it defines the additional macro defined in `<cstdarg>` by including the header `<cstdarg>`.

8.20 The header `<cstdbool>` [tr.c99.cbool]

1 The header simply defines the macro:

```
__bool_true_false_are_defined
```

as defined in C99 subclause 7.16.

8.21 The header `<stdbool.h>` [tr.c99.boolh]

1 The header behaves as if it defines the additional macro defined in `<cstbool>` by including the header `<cstdbool>`.

8.22 The header `<cstdint>` [tr.c99.cstdint]

8.22.1 Synopsis [tr.c99.cstdint.syn]

```
namespace std {
namespace tr1 {
  typedef signed integer type int8_t;   // optional
  typedef signed integer type int16_t;  // optional
  typedef signed integer type int32_t;  // optional
  typedef signed integer type int64_t;  // optional

  typedef signed integer type int_fast8_t;
  typedef signed integer type int_fast16_t;
  typedef signed integer type int_fast32_t;
  typedef signed integer type int_fast64_t;

  typedef signed integer type int_least8_t;
  typedef signed integer type int_least16_t;
  typedef signed integer type int_least32_t;
  typedef signed integer type int_least64_t;

  typedef signed integer type intmax_t;
  typedef signed integer type intptr_t;
```

```
    typedef unsigned integer type uint8_t;      // optional
    typedef unsigned integer type uint16_t;     // optional
    typedef unsigned integer type uint32_t;     // optional
    typedef unsigned integer type uint64_t;     // optional

    typedef unsigned integer type uint_fast8_t;
    typedef unsigned integer type uint_fast16_t;
    typedef unsigned integer type uint_fast32_t;
    typedef unsigned integer type uint_fast64_t;

    typedef unsigned integer type uint_least8_t;
    typedef unsigned integer type uint_least16_t;
    typedef unsigned integer type uint_least32_t;
    typedef unsigned integer type uint_least64_t;

    typedef unsigned integer type uintmax_t;
    typedef unsigned integer type uintptr_t;
  } // namespace tr1
  } // namespace std
```

1 The header also defines numerous macros of the form:

```
INT[FAST LEAST]{8 16 32 64}_MIN
[U]INT[FAST LEAST]{8 16 32 64}_MAX
INT{MAX PTR}_MIN
[U]INT{MAX PTR}_MAX
{PTRDIFF SIG_ATOMIC WCHAR WINT}{_MAX _MIN}
SIZE_MAX
```

plus function macros of the form:

```
[U]INT{8 16 32 64 MAX}_C
```

8.22.2 Definitions [tr.c99.cstdint.def]

1 The header defines all functions, types, and macros the same as C99 subclause 7.18.

8.23 The header `<stdint.h>` [tr.c99.stdinth]

1 The header behaves as if it includes the header `<cstdint>`, and provides sufficient *using* declarations to declare in the global namespace all type names defined in the header `<cstdint>`.

8.24 Additions to header `<cstdio>` [tr.c99.cstdio]

8.24.1 Synopsis [tr.c99.cstdio.syn]

```
namespace std {
namespace tr1 {
  int snprintf(char *s, size_t n, const char *format, ...);
  int vsnprintf(char *s, size_t n, const char *format, va_list ap);

  int vfscanf(FILE *stream, const char *format, va_list ap);
```

```
    int vscanf(const char *format, va_list ap);
    int vsscanf(const char *s, const char *format, va_list ap);
  } // namespace tr1
  } // namespace std
```

8.24.2 Definitions [tr.c99.cstdio.def]

1 The header defines all added functions the same as C99 subclause 7.19.

8.24.3 Additional format specifiers [tr.c99.cstdio.spec]

1 The formatted output functions shall support the additional conversion specifications specified in C99 subclause 7.19.6.1.

2 The formatted input functions shall support the additional conversion specifications specified in C99 subclause 7.19.6.2.

3 [*Note*: These include the conversion specifiers `a` (for hexadecimal floating-point) and `F`, and the conversion qualifiers `hh`, `h`, `ll`, `t`, and `z` (for various integer types). They also include the ability to match and generate various text forms of infinity and NaN values. *—end note*]

8.24.4 Additions to header `<stdio.h>` [tr.c99.stdioh]

1 The header behaves as if it includes the header `<cstdio>`, and provides sufficient additional *using* declarations to declare in the global namespace all added function names defined in the header `<cstdio>`.

8.25 Additions to header `<cstdlib>` [tr.c99.cstdlib]

8.25.1 Synopsis [tr.c99.cstdlib.syn]

```
namespace std {
namespace tr1 {
  // types
  typedef struct {
    _Longlong quot, rem;
  } lldiv_t;

  // functions
  _Longlong llabs(long long i);
  lldiv_t lldiv(_Longlong numer, _Longlong denom);

  _Longlong atoll(const char *s);
  _Longlong strtoll(const char *s, char **endptr, int base);
  _ULonglong strtoull(const char *s, char **endptr, int base);

  float strtof(const char *s, char **endptr);
  long double strtold(const char *s, char **endptr);

  // overloads
  _Longlong abs(_Longlong i);
  lldiv_t div(_Longlong numer, _Longlong denom);
} // namespace tr1
} // namespace std
```

8.25.2 Definitions [tr.c99.cstdlib.def]

1 The header defines all added types and functions, other than the overloads of abs and div, the same as C99 subclause 7.20.

8.25.3 Function abs [tr.c99.cstdlib.abs]

```
_Longlong abs(_Longlong i);
```

1 *Effects:* Behaves the same as C99 function llabs, defined in subclause 7.20.6.1.

8.25.4 Function div [tr.c99.cstdlib.div]

```
lldiv_t div(_Longlong numer, _Longlong denom);
```

1 *Effects:* Behaves the same as C99 function lldiv, defined in subclause 7.20.6.2.

8.26 Additions to header <stdlib.h> [tr.c99.stdlibh]

1 The header behaves as if it includes the header <cstdlib>, and provides sufficient additional *using* declarations to declare in the global namespace all added type and function names defined in the header <cstdlib>.

8.27 Header <ctgmath> [tr.c99.ctgmath]

1 The header simply includes the headers <ccomplex> and <cmath>.

2 [*Note*: The overloads provided in C99 by magic macros are already provided in <ccomplex> and <cmath> by "sufficient" additional overloads. *—end note*]

8.28 Header <tgmath.h> [tr.c99.tgmathh]

1 The header effectively includes the headers <complex.h> and <math.h>.

8.29 Additions to header <ctime> [tr.c99.ctime]

1 The function strftime shall support the additional conversion specifiers and modifiers specified in C99 subclause 7.23.3.4.

2 [*Note*: These include the conversion specifiers C, D, e, F, g, G, h, r, R, t, T, u, V, and z, and the modifiers E and O. *—end note*]

8.30 Additions to header <cwchar> [tr.c99.cwchar]

8.30.1 Synopsis [tr.c99.cwchar.syn]

```
namespace std {
namespace tr1 {
  float wcstof(const wchar_t *nptr, wchar_t **endptr);
  long double wcstold(const wchar_t *nptr, wchar_t **endptr);
  _Longlong wcstoll(const wchar_t *nptr, wchar_t **endptr, int base);
  _ULonglong wcstoull(const wchar_t *nptr, wchar_t **endptr, int base);

  int vfwscanf(FILE *stream, const wchar_t *format, va_list arg);
```

```
    int vswscanf(const wchar_t *s, const wchar_t *format, va_list arg);
    int vwscanf(const wchar_t *format, va_list arg);
  } // namespace tr1
  } // namespace std
```

1 Moreover, the function `wcsftime` shall support the additional conversion specifiers and modifiers specified in C99 subclause 7.23.3.4.

8.30.2 Definitions [tr.c99.cwchar.def]

1 The header defines all added functions the same as C99 subclause 7.24.

8.30.3 Additional wide format specifiers [tr.c99.cwchar.spec]

1 The formatted wide output functions shall support the additional conversion specifications specified in C99 subclause 7.24.2.1.

2 The formatted wide input functions shall support the additional conversion specifications specified in C99 subclause 7.24.2.2.

3 [*Note*: These are essentially the same extensions as for the header `<cstdio>`. *—end note*]

8.31 Additions to header `<wchar.h>` [tr.c99.wcharh]

1 The header behaves as if it includes the header `<cwchar>`, and provides sufficient additional *using* declarations to declare in the global namespace all added function names defined in the header `<cwchar>`.

8.32 Additions to header `<cwctype>` [tr.c99.cwctype]

8.32.1 Synopsis [tr.c99.cwctype.syn]

```
namespace std {
namespace tr1 {
  int iswblank(wint_t ch);
} // namespace tr1
} // namespace std
```

8.32.2 Function `iswblank` [tr.c99.cwctype.iswblank]

1 Function `iswblank` behaves the same as C99 function `iswblank`, defined in subclause 7.25.2.1.3.

8.33 Additions to header `<wctype.h>` [tr.c99.wctypeh]

1 The header behaves as if it includes the header `<cwctype>`, and provides sufficient additional *using* declarations to declare in the global namespace the additional function name declared in the header `<cwctype>`.

Annex A (informative) Implementation quantities [tr.limits]

1 $\mathtt{N}_{\mathtt{max}}$, the maximum number of arguments that can be forwarded by the call wrappers ([3.1]) defined in [2.1], [3.5], [3.6], [3.7], and the maximum number of argument types that can be passed in the argument to `result_of` ([3.4]), is implementation defined. The value of $\mathtt{N}_{\mathtt{max}}$ should be at least 10.

2 The number of distinct placeholders ([3.2]) is implementation defined. The number should be at least 10.

3 The maximum number of elements in one tuple type (clause 6.1) is implementation defined. This limit should be at least 10.

Bibliography

[1] 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.

[2] Matthew Austern, “A Proposal to Add Hash Tables to the Standard Library (revision 4),” WG21 Document N1456=03-0039, 2003.

[3] Walter Brown, “A Proposal to Add Mathematical Special Functions to the C++ Standard Library,” WG21 Document N1422 = 03-0004, 2003.

[4] Peter Dimov, “A Proposal to Add an Enhanced Member Pointer Adaptor to the Library Technical Report,” WG21 Document N1432=03-0014, 2003.

[5] 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.

[6] 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.

[7] Ecma International, *ECMAScript Language Specification*, Standard Ecma-262, third edition, 1999.

[8] Douglas Gregor, “A Proposal to add a Polymorphic Function Object Wrapper to the Standard Library,” WG21 Document N1402=02-0060, 2002.

[9] Douglas Gregor, “A uniform method for computing function object return types (revision 1),” WG21 Document N1454=03-0037, 2003.

[10] Douglas Gregor and Peter Dimov, “A proposal to add a reference wrapper to the standard library (revision 1),” WG21 Document N1453=03-0036, 2003.

[11] IEEE, *Information Technology—Portable Operating System Interface (POSIX)*, IEEE Standard 1003.1-2001.

[12] International Standards Organization: *Quantities and units, Third edition*. International Standard ISO 31-11:1992. ISBN 92-67-10185-4.

[13] International Standards Organization: *Programming Languages – C, Second edition*. International Standard ISO/IEC 9899:1999.

[14] International Standards Organization: *Programming Languages – C++*. International Standard ISO/IEC 14882:1998.

[15] Jaakko Järvi, “Proposal for adding tuple types into the standard library Programming Language C++,”, WG21 Document N1403=02-0061, 2002.

[16] John Maddock, “A Proposal to add Type Traits to the Standard Library,” WG21 Document 03-0006 = N1424, 2003.

[17] John Maddock, “A Proposal to add Regular Expressions to the Standard Library,” WG21 Document 03-0011= N1429, 2003.

[18] Jens Maurer, “A Proposal to Add an Extensible Random Number Facility to the Standard Library (Revision 2),” WG21 Document N1452, 2003.

Index