Doc No: N1518 = 03-0101
Date: September 20, 2003
Reply to: Matt Austern
austern@apple.com

(Draft) Technical Report on Standard Library Extensions

Contents

Chapter 1

General [tr.intro]

This technical report describes extensions to the *C++ standard library* that is described in the International Standard for the C++ programming language [6].
This technical report is non-normative. Some of the library components in this technical report may be considered for standardization in a future version of C++, but they are not currently part of any C++ standard. Some of the components in this technical report may never be standardized, and others may be standardized in a substantially changed form.
The goal of this technical report 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 Method of description [tr.description]

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

1.2 Categories of extensions [tr.intro.ext]

This technical report describes four general categories of library extensions:

1. New requirement tables, such as the regular expression traits requirements in clause 7.2. These are not directly expressed as software; they specify the circumstances under which user-written components will interoperate with the components described in this technical report.

2. New library components (types and functions) that are declared in entirely new headers, such as the class templates in the `<unordered_set>` header 6.2.4.1.

3. New library components declared as additions to existing standard headers, such as the mathematical special functions added to the header `<cmath>` 5.2.1.

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

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

New headers are distinguished from extensions to existing headers by the title of the *synopsis* clause. In the first case the title is of the form "Header `<foo>` synopsis", and the synopsis includes all namespace scope declarations contained in the header. In the second case the title is of the form "Additions to header `<foo>` synopsis" and the synopsis includes only the extensions, *i.e.* those namespace scope declarations that are not present in the C++ standard [6].

1.3 Namespaces and headers [tr.intro.namespaces]

Since the extensions described in this technical report are not part of the C++ standard library, they should not be declared within namespace `std`. Unless otherwise specifed, all components described in this technical report are declared in namespace `tr1`. [*Note:* Some components are declared in subnamespaces of namespace `tr1`. *—end note*]

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

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

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

1.4 Caveat [tr.intro.caveat]

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

Chapter 2

General Utilities [tr.util]

2.1 Reference wrappers [tr.util.refwrap]

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

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

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

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

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

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

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

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

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

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

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

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

Effects: Constructs a reference_wrapper object that stores a reference to t.
Throws: Does not throw.

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

```
operator () const;
```

Returns: The stored reference.
Throws: Does not throw.

```
get() const;
```

Returns: The stored reference.
Throws: Does not throw.

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

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

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

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

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

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

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

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

2.1.2.5 implementation quantities [tr.util.refwrp.lim]

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

2.2 Smart pointers [tr.util.smartptr]

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

```
namespace tr1 {
  class bad_weak_ptr;

  template<class T> class shared_ptr;

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

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

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

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

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

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

  template<class E, class T, class Y>
    basic_ostream<E, T> & operator<< (basic_ostream<E, T> & os, shared_ptr<Y> const

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

  template<class T> class weak_ptr;

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

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

  template<class T> class enable_shared_from_this;
}
```

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

```
namespace tr1 {
  class bad_weak_ptr: public exception
  {
  public:
```

```
    bad_weak_ptr();
  };
}
```

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

```
bad_weak_ptr();
```

Postconditions: `what()` returns `"tr1::bad_weak_ptr"`.
Throws: nothing.

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

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

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

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

    // destructor
    ~shared_ptr();

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

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

    // observers
    T * get() const;
    T & operator*() const;
```

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

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

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

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

  // other operators
  template<class E, class T, class Y>
    basic_ostream<E, T> & operator<< (basic_ostream<E, T> & os, shared_ptr<Y> const

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

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

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

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

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

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

—end example.]

2.2.3.1 `shared_ptr` constructors [tr.util.smartptr.shared.const]

```
shared_ptr();
```

Effects: Constructs an *empty* `shared_ptr`.
Postconditions: `use_count() == 0 && get() == 0`.
Throws: nothing.

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

Requires: `p` must be convertible to `T *`. `Y` must be a complete type. The expression `delete p` must be well-formed, must not invoke undefined behavior, and must not throw exceptions.
Effects: Constructs a `shared_ptr` that *owns* the pointer `p`.
Postconditions: `use_count() == 1 && get() == p`.
Throws: `bad_alloc` or an implementation-defined exception when a resource other than memory could not be obtained.
Exception safety: If an exception is thrown, `delete p` is called.

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

Requires: `p` must be convertible to `T *`. `D` must be `CopyConstructible`. The copy constructor and destructor of `D` must not throw. The expression `d(p)` must be well-formed, must not invoke undefined behavior, and must not throw exceptions.
Effects: Constructs a `shared_ptr` that *owns* the pointer `p` and the deleter `d`.
Postconditions: `use_count() == 1 && get() == p`.
Throws: `bad_alloc` or an implementation-defined exception when a resource other than memory could not be obtained.
Exception safety: If an exception is thrown, `d(p)` is called.

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

Effects: If `r` is *empty*, constructs an *empty* `shared_ptr`; otherwise, constructs a `shared_ptr` that *shares ownership* with `r`.
Postconditions: `get() == r.get() && use_count() == r.use_count()`.
Throws: nothing.

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

Effects: Constructs a `shared_ptr` that *shares ownership* with `r` and stores a copy of the pointer stored in `r`.
Postconditions: `use_count() == r.use_count()`.
Throws: `bad_weak_ptr` when `r.expired()`.
Exception safety: If an exception is thrown, the constructor has no effect.

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

Requires: `r.release()` must be convertible to `T *`. `Y` must be a complete type. The expression `delete r.release()` must be well-formed, must not invoke undefined behavior, and must not throw exceptions.
Effects: Constructs a `shared_ptr` that stores and *owns* `r.release()`.
Postconditions: `use_count() == 1`.
Throws: `bad_alloc` or an implementation-defined exception when a resource other than memory could not be obtained.

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

2.2.3.2 `shared_ptr` destructor [tr.util.smartptr.shared.dest]

```
~shared_ptr();
```

Effects:

- If `*this` is *empty*, or *shares ownership* with another `shared_ptr` instance (`use_count() > 1`), there are no side effects.
- Otherwise, if `*this` *owns* a pointer `p` and a deleter `d`, `d(p)` is called.
- Otherwise, `*this` *owns* a pointer `p`, and `delete p` is called.

Throws: nothing.

2.2.3.3 `shared_ptr` assignment [tr.util.smartptr.shared.assign]

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

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

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

both assignments may be no-ops.

2.2.3.4 `shared_ptr` modifiers [tr.util.smartptr.shared.mod]

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

Effects: Exchanges the contents of `*this` and `r`.
Throws: nothing.

```
void reset();
```

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

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

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

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

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

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

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

Returns: the stored pointer.
Throws: nothing.

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

Requires: `get() != 0`.
Returns: `*get()`.
Throws: nothing.
Notes: When `T` is `void`, the return type of this member function is unspecified, and an attempt to instantiate it renders the program ill-formed.

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

Requires: `get() != 0`.
Returns: `get()`.
Throws: nothing.

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

Returns: the number of `shared_ptr` objects, `*this` included, that *share ownership* with `*this`, or `0` when `*this` is *empty*.
Throws: nothing.
Notes: `use_count()` is not necessarily efficient. Use only for debugging and testing purposes, not for production code.

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

Returns: `use_count() == 1`.
Throws: nothing.
Notes: `unique()` may be faster than `use_count()`. If you are using `unique()` to implement copy on write, do not rely on a specific value when `get() == 0`.

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

Returns: an unspecified value that, when used in boolean contexts, is equivalent to `get() != 0`.
Throws: nothing.
Notes: This conversion operator allows *shared_ptr* objects to be used in boolean contexts, like `if (p && p->valid())` . The actual target type is typically a pointer to a member function, avoiding many of the implicit conversion pitfalls.

2.2.3.6 `shared_ptr` comparison [tr.util.smartptr.shared.cmp]

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

Returns: `a.get() == b.get()`.
Throws: nothing.

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

Returns: `a.get() != b.get()`.
Throws: nothing.

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

Returns: an unspecified value such that

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

Throws: nothing.
Notes: Allows `shared_ptr` objects to be used as keys in associative containers.

2.2.3.7 `shared_ptr` operators [tr.util.smartptr.shared.op]

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

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

2.2.3.8 `shared_ptr` specialized algorithms [tr.util.smartptr.shared.spec]

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

Effects: Equivalent to `a.swap(b)`.
Throws: nothing.

2.2.3.9 `shared_ptr` casts [tr.util.smartptr.shared.cast]

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

Requires: The expression `static_cast<T*>(r.get())` must be well-formed.
Returns: If `r` is *empty*, an *empty* `shared_ptr<T>`; otherwise, a `shared_ptr<T>` object that stores `static_cast<T*>(r.get())` and *shares ownership* with `r`.
Throws: nothing.
Notes: the seemingly equivalent expression `shared_ptr<T>(static_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice.

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

Requires: The expression `dynamic_cast<T*>(r.get())` must be well-formed and its behavior defined.
Returns:

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

- Otherwise, an *empty* `shared_ptr<T>` object.

Throws: nothing.
Notes: the seemingly equivalent expression `shared_ptr<T>(dynamic_cast<T*>(r.get()))` will eventually result in undefined behavior, attempting to delete the same object twice.

2.2.3.10 get_deleter [tr.util.smartptr.getdeleter]

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

Returns: If `*this` *owns* a deleter `d` of type cv-unqualified `D`, returns `&d`; otherwise returns `0`.
Throws: nothing.

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

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

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

  public:
    typedef T element_type;

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

    // destructor
    ~weak_ptr();

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

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

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

```
  };

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

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

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

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

```
weak_ptr();
```

Effects: Constructs an *empty* `weak_ptr`.
Postconditions: `use_count() == 0`.
Throws: nothing.

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

Effects: If `r` is *empty*, constructs an *empty* `weak_ptr`; otherwise, constructs a `weak_ptr` that *shares ownership* with `r` and stores a copy of the pointer stored in `r`.
Postconditions: `use_count() == r.use_count()`.
Throws: nothing.

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

```
~weak_ptr();
```

Effects: Destroys this `weak_ptr` but has no effect on the object its stored pointer points to.
Throws: nothing.

2.2.4.3 `weak_ptr` assignment [tr.util.smartptr.weak.assign]

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

Effects: Equivalent to `weak_ptr(r).swap(*this)`.
Throws: nothing.
Notes: The implementation is free to meet the effects (and the implied guarantees) via different means, without creating a temporary.

2.2.4.4 `weak_ptr` modifiers [tr.util.smartptr.weak.mod]

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

Effects: Exchanges the contents of `*this` and `r`.
Throws: nothing.

```
void reset();
```

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

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

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

Returns: `0` if `*this` is *empty*; otherwise, the number of `shared_ptr` instances that *share ownership* with `*this`.
Throws: nothing.
Notes: `use_count()` is not necessarily efficient. Use only for debugging and testing purposes, not for production code.

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

Returns: `use_count() == 0`.
Throws: nothing.
Notes: `expired()` may be faster than `use_count()`.

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

Returns: `expired()? shared_ptr<T>(): shared_ptr<T>(*this)`.
Throws: nothing.

2.2.4.6 `weak_ptr` comparison [tr.util.smartptr.weak.cmp]

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

Returns: an unspecified value such that

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

Throws: nothing.
Notes: Allows `weak_ptr` objects to be used as keys in associative containers.

2.2.4.7 `weak_ptr` specialized algorithms [tr.util.smartptr.weak.spec]

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

Effects: Equivalent to `a.swap(b)`.
Throws: nothing.

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

A class can derive from the `enable_shared_from_this` class template, passing itself as a template parameter, to inherit the `shared_from_this` member functions that obtain a *shared_ptr* instance pointing to `this`.
[*Example:*

```
struct X: public enable_shared_from_this<X>
{
};

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

—*end example.*]

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

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

Effects: Constructs an `enable_shared_from_this<T>` instance.
Throws: nothing.

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

Returns: `*this`.
Throws: nothing.

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

Effects: Destroys `*this`.
Throws: nothing.

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

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

Requires: `enable_shared_from_this<T>` must be an accessible base class of `T`. `*this` must be a subobject of an instance `t` of type `T`. There must exist at least one `shared_ptr` instance `p` that *owns* `&t`.
Returns: A `shared_ptr<T>` instance `r` that *shares ownership with* `p`.
Postconditions: `r.get() == this`.
[*Note:* a typical implementation is shown below:

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

    weak_ptr<T> __weak_this;

protected:

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

public:

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

The three *shared_ptr* constructors that create unique pointers should detect the presence of an *enable_shared_from_this* base and assign the newly created *shared_ptr* to its *__weak_this* member. —*end note.*]

Chapter 3

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

3.1 Function return types [tr.func.ret]

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

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

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

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

Given an lvalue `f` of type `F` and lvalues `t1`, `t2`, ..., `tN` of types `T1`, `T2`, ..., `TN`, respectively, the `type` member type defines the result type of the expression `f(t1, t2, ...,tN)`.
The implementation may determine the member type `type` via any means that produces the exact type of the expression `f(t1, t2, ..., tN)` for the given types. [*Note:* The intent is that implementations are permitted to use special compiler hooks —*end note*]
If the implementation cannot determine the type of the expression `f(t1, t2, ..., tN)`, or if the expression is ill-formed, the implementation shall use the following process to determine the member type `type`:

1. If `F` is a function type, `type` is the return type of the function type `F`.
2. If `F` is a member function type, `type` is the return type of the member function type `F`.

3. If `F` is a function object defined by the standard library, the method of determining `type` is unspecified.

4. If `F` is a class type with a member type `result_type`, `type` is `F::result_type`.

5. If `F` is a class type with no member named `result_type` or if `F::result_type` is not a type:

   (a) If `N=0` (no arguments), `type` is `void`.

   (b) If `N>0`, `type` is `F::result<F(T1, T2,..., TN)>::type`.

6. Otherwise, the program is ill-formed.

3.2 Member pointer adaptors [tr.func.memfn]

3.2.1 Additions to header `<functional` synopsis [tr.func.memfn.synopsis]

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

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

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

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

Returns:

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

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

Throws: nothing.
Notes: Implementations are allowed to implement mem_fn as a set of overloaded function templates.

3.2.3 implementation quantities [tr.func.memfn.lim]

The maximum value for n is implementation defined.

3.3 Function object binders [tr.func.bind]

3.3.1 Additions to header ¡functional¿ synopsis [tr.func.bind.synopsis]

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

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

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

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

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

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

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

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

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

```
static const bool value;
```

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

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

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

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

```
static const int value;
```

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

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

The function λ`(x)` is defined as `x.get()` when `x` is of type `reference_wrapper<F>` for some `F`, `x` otherwise.
The function μ`(x, v`$_1$`, ..., v`$_m$ `x` is of type `X`, and `k` is `is_placeholder<X>::value`, is defined as:

- `x.get()`, when `X` is a `reference_wrapper<T>` for some `T`;
- `v`$_k$, when `k != 0`;
- `x(v`$_1$`, ..., v`$_m$`)`, when `is_bind_expression<X>::value` is `true`;
- `x` otherwise.

A function object `f` of type `F` is called *simple*, if `f` is a pointer to a function with C++ linkage or `F::result_type` is a type.
The maximum number of supported arguments (*N* in the synopsis) is implementation defined. Implementations are allowed to define additional, more specialized, `bind` overloads, or to fold the pointer to member overload into the general function template, as long as the behavior of the `bind` calls is unchanged.
Given a list of arguments `a1`, `a2`, ..., `a`*n*, a function object `h` as returned from a call to `bind`, and the definition:

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

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

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

Requires: `F` must be `CopyConstructible`. λ`(f)()` must be a valid expression. If `f` is not a *simple* function object, the behavior is implementation defined.
Returns: A function object `g` of an unspecified `CopyConstructible` type `G` such that the expression `g(`v_1`, ..., `v_m`)` is equivalent to λ`(f)()`. The type of the expression `g(`v_1`, ..., `v_m`)` is `result_of<R()>::type` where `R` is the type of λ`(f)`. If the function application is made via a cv-qualified reference to, or copy of, `g`, the same cv-qualifiers are applied to `f` before the evaluation. When `f` is a pointer to a function with C++ linkage, `G::result_type` is defined as the return type of the `f`. When `F::result_type` is defined, `G::result_type` is defined as the same type.
Throws: nothing unless the copy constructor of `f` throws an exception.
Notes: Implementations are allowed to impose an upper limit on *m* (typically one more than the number of supported placeholders). It is implementation defined whether `G` is `Assignable` or `DefaultConstructible`.

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

Requires: `F` must be `CopyConstructible`. λ`(f)()` must be a valid expression convertible to `R`.
Returns: A function object `g` of an unspecified `CopyConstructible` type `G` such that the expression `g(`v_1`, ..., `v_m`)` is equivalent to λ`(f)()`, implicitly converted to `R`. If the function application is made via a cv-qualified reference to, or copy of, `g`, the same cv-qualifiers are applied to `f` before the evaluation. `G::result_type` is defined as `R`.
Throws: nothing unless the copy constructor of `f` throws an exception.
Notes: Implementations are allowed to impose an upper limit on *m*. It is implementation defined whether `G` is `Assignable` or `DefaultConstructible`.

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

Requires: `F` and `A1` must be `CopyConstructible`. λ`(f)(w1)` must be a valid expression for some value `w1`. If `f` is not a *simple* function object, the behavior is implementation defined.
Returns: A function object `g` of an unspecified `CopyConstructible` type `G` such that the expression `g(`v_1`, ..., `v_m`)` is equivalent to λ`(f)(`μ`(a1, `v_1`, ..., `v_m`))`. The type of the expression `g(`v_1`, ..., `v_m`)` is `result_of<R(T)>::type` where `R` is the type of λ`(f)` and `T` is the type of μ`(a1, `v_1`, ..., `v_m`)`. If the function application is made via a cv-qualified reference to, or copy of, `g`, the same cv-qualifiers are applied to `f` and `a1` before the evaluation. When `f` is a pointer to a function with C++ linkage, `G::result_type` is defined as the return type of the `f`. When `F::result_type` is defined, `G::result_type` is defined as the same type.
Throws: nothing unless the copy constructors of `f` or `a1` throw an exception.
Notes: Implementations are allowed to impose an upper limit on *m*. It is implementation defined whether `G` is `Assignable` or `DefaultConstructible`.

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

Requires: `pm` must be a pointer to data member or pointer to a member function taking no arguments.
Returns: `bind(mem_fn(pm), a1)`.

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

Requires: F and A1 must be CopyConstructible. λ(f)(w1), for some value w1, must be a valid expression convertible to R.
Returns: A function object g of an unspecified CopyConstructible type G such that the expression g(v_1, ..., v_m) is equivalent to λ(f)(μ(a1, v_1, ..., v_m)), implicitly converted to R. If the function application is made via a cv-qualified reference to, or copy of, g, the same cv-qualifiers are applied to f and a1 before the evaluation. G::result_type is defined as R.
Throws: nothing unless the copy constructors of f or a1 throw an exception.
Notes: Implementations are allowed to impose an upper limit on *m*. It is implementation defined whether G is Assignable or DefaultConstructible.

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

Requires: F and A*i* must be CopyConstructible. λ(f)(w1, ..., w*n*) must be a valid expression for some values w*i*. If f is not a *simple* function object, the behavior is implementation defined.
Returns: A function object g of an unspecified CopyConstructible type G such that the expression g(v_1, ..., v_m) is equivalent to λ(f)(μ(a1, v_1, ..., v_m), ..., μ(a*n*, v_1, ..., v_m)).
The type of the expression g(v_1, ..., v_m) is result_of<R(T1, T2, ..., T*n*)>::type where R is the type of λ(f) and each T*i* is the type of μ(a*i*, v_1, ..., v_m).
If the function application is made via a cv-qualified reference to, or copy of, g, the same cv-qualifiers are applied to f and a*i* before the evaluation. When f is a pointer to a function with C++ linkage, G::result_type is defined as the return type of the f. When F::result_type is defined, G::result_type is defined as the same type.
Throws: nothing unless the copy constructors of f or a*i* throw an exception.
Notes: Implementations are allowed to impose an upper limit on *m*. It is implementation defined whether G is Assignable or DefaultConstructible.

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

Requires: pmf must be a pointer to a member function taking *n*-1 arguments.
Returns: bind(mem_fn(pmf), a1, ..., a*n*).

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

Requires: F and A*i* must be CopyConstructible. λ(f)(w1, ..., w*n*), for some values w*i*, must be a valid expression convertible to R.
Returns: A function object g of an unspecified CopyConstructible type G such that the expression g(v_1, ..., v_m) is equivalent to λ(f)(μ(a1, v_1, ..., v_m), ..., μ(a*n*, v_1, ..., v_m)), implicitly converted to R. If the function application is made via a cv-qualified reference to, or copy of, g, the same cv-qualifiers are applied to f and a*i* before the evaluation. G::result_type is defined as R.
Throws: nothing unless the copy constructors of f or a*i* throw an exception.
Notes: Implementations are allowed to impose an upper limit on *m*. It is implementation defined whether G is Assignable or DefaultConstructible.

3.3.5 Placeholders [tr.func.bind.place]

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

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

3.3.6 Implementation quantities [tr.func.bind.limits]

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

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

3.4 Polymorphic function wrappers [tr.func.wrap]

3.4.1 Additions to `<functional>` synopsis [tr.func.wrap.synopsis]

```
class bad_function_call;

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

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

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

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

3.4.2 Class `bad_function_call` [tr.func.wrap.badcall]

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

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

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

```
bad_function_call();
```

Effects: constructs a `bad_function_call` object.

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

The library provides polymorphic wrappers that generalize the notion of a function pointer. Wrappers can store, copy, and call arbitrary function objects given a function signature (denoted by a set of argument types and a return type), allowing functions to be first-class objects.
A function object `f` of type `F` is *Callable* given a set of argument types `T1`, `T2`, ..., `TN` and a return type `R`, if the appropriate following function definition is well-formed:

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

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

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

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

```
public:
  typedef R         result_type;
  typedef Allocator allocator_type;

  // [tr.func.wrap.func.con] construct/copy/destroy
  explicit function(const Allocator& = Allocator());
  function(const function&);
  template<typename F>
    function(F, const Allocator& = Allocator());
  template<typename F>
    function(reference_wrapper<F>,
             const Allocator& = Allocator());
  function& operator=(const function&);
  template<typename F>
    function& operator=(F);
  template<typename F>
    function& operator=(reference_wrapper<F>);
  ~function();

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

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

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

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

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

```
}
```

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

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

Postconditions: `this->empty()`.
Throws: will not throw.

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

Postconditions: `this->empty()`if `f.empty()`; otherwise, `*this` targets a copy of `f`.
Throws: will not throw if the target of `f` is a function pointer or a function object passed via `reference_wrapper`. Otherwise, may throw `bad_alloc` or any exception thrown by the copy constructor of the stored function object.

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

Requires: `f` is a callable function object for argument types `T1`, `T2`, ..., `TN` and return type `R`.
Postconditions: `this->empty()` if any of the following hold:

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

Otherwise, `*this` targets a copy of `f` if `f` is not a pointer to member function, and targets a copy of `mem_fun(f)` if `f` is a pointer to member function.
Throws: will not throw when `f` is a function pointer. Otherwise, may throw `bad_alloc` or any exception thrown by `F`'s copy constructor.

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

Requires: `f.get()` is a callable function object for argument types `T1`, `T2`, ..., `TN` and return type `R`.
Postconditions: `!this->empty()` and `*this` targets `f.get()`.
Throws: will not throw.
Rationale: a potential alternative would be to replace the `reference_wrapper` argument with an argument taking a function object pointer. This route was not taken because `reference_wrapper` is a general solution stating the user's intention to pass a reference to an object instead of a copy.

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

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

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

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

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

Effects: `function(f).swap(*this);`
Returns: `*this`
Throws: will not throw.

```
~function();
```

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

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

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

Effects: interchanges the targets of `*this` and `other` and the allocators of `*this` and `other`.
Throws: will not throw.

```
void clear();
```

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

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

```
bool empty() const
```

Returns: `true` if the function object has a target, `false` otherwise.
Throws: will not throw.

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

Returns: if `!this->empty()`, returns a value that will evaluate true in a boolean context; otherwise, returns a value that will evaluate false in a boolean context. The value type returned shall not be convertible to `int`.
Throws: will not throw.
Notes: This conversion can be used in contexts where a `bool` is expected (e.g., an `if` condition); however, implicit conversions (e.g., to `int`) that can occur with `bool` are not allowed, eliminating some sources of user error. The suggested implementation technique is to use a member function pointer whose class type is private to the `function` instantiation.

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

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

Effects: `f(t1, t2, ..., tN)`, where `f` is the target of `*this`.
Returns: nothing, if `R` is `void`; otherwise, the return value of the call to `f`.
Throws: `bad_function_call` if `this->empty()`; otherwise, any exception thrown by the wrapped function object.

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

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

Effects: `f1.swap(f2);`

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

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

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

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

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

3.4.4 Implementation quantities [tr.func.wrap.limits]

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

Chapter 4

Metaprogramming and type traits [tr.meta]

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

4.1 Requirements [tr.meta.rqmts]

4.1.1 Unary type traits [tr.meta.rqmts.unary]

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

Table 4.1: UnaryTypeTrait requirements

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

4.1.2 Binary type traits [tr.meta.rqmts.binary]

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

Table 4.2: BinaryTypeTrait requirements

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

4.1.3 Transformation type traits [tr.meta.rqmts.trans]

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

Table 4.3: TransformationTrait requirements

Expression	Requirement
`X<T>::type`	The result is a type that is the result of applying transformation X to type T.

4.2 Unary Type Traits [tr.meta.unary]

This sub-clause contains templates that may be used to query the properties of a type at compile time.
All of the class templates defined in this header satisfy the UnaryTypeTrait requirements.
For all of the class templates declared in this clause, all members declared `static const` shall be defined in such a way that they are usable as integral constant expressions.
For all of the class templates X declared in this clause, both rvalues of type `X const` and lvalues of type `X const&` shall be implicitly convertible to `X::type`. Whether this is accomplished by inheritance or by a member operator is unspecified. For exposition only the class templates in this clause are shown with a member conversion operator: `operator type() const`, which shall return `type()` in all cases.
For all of the class templates X declared in this clause, instantiating that template with a template-argument that is a class template specialization, may result in the implicit instantiation of the template

argument if and only if the semantics of X require that the argument must be a complete type.

4.2.1 Header `<type_traits>` synopsis [tr.meta.unary.synop]

```
namespace tr1{

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

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

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

// type properties:
template <class T> struct is_const;
template <class T> struct is_volatile;
template <class T> struct is_pod;
template <class T> struct is_empty;
template <class T> struct is_polymorphic;
template <class T> struct is_abstract;
template <class T> struct has_trivial_constructor;
template <class T> struct has_trivial_copy;
template <class T> struct has_trivial_assign;
template <class T> struct has_trivial_destructor;
template <class T> struct has_nothrow_constructor;
template <class T> struct has_nothrow_copy;
```

```
template <class T> struct has_nothrow_assign;
template <class T> struct is_signed;
template <class T> struct is_unsigned;

} // namespace tr1
```

4.2.2 Helper classes [tr.meta.unary.help]

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

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

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

The primary type categories correspond to the descriptions given in section 3.9 of the C++ standard.
For any given type `T`, exactly one of the primary type categories shall have its member `value` evaluate to `true`.
For any given type `T`, the result of applying one of these templates to `T`, and to *cv-qualified* `T` shall yield the same result.
Undefined behaviour results if any C++ program adds specializations for any of the class templates defined in this clause.

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

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

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

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

```
template <class T> struct is_floating_point {
```

```
   static  const bool value                    = implementation_defined;
   typedef bool                                value_type;
   typedef integral_constant<value_type,value> type;
   operator type() const;
};
```

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

4.2.4 Composite type traits [tr.meta.unary.comp]

These templates provide convenient compositions of the primary type categories, corresponding to the descriptions given in section 3.9.
For any given type `T`, the result of applying one of these templates to `T`, and to *cv-qualified* `T` shall yield the same result.
Undefined behaviour results if any C++ program adds specializations for any of the class templates defined in this clause.

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

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

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

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

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

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

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

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

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

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

```
template <class T> struct is_member_pointer {
   static const bool value =
```

```
        is_member_object_pointer<T>::value
        || is_member_function_pointer<T>::value;
    typedef bool                                value_type;
    typedef integral_constant<value_type,value> type;
    operator type() const;
};
```

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

4.2.5 Type properties [tr.meta.unary.prop]

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

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

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

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

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

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

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

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

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

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

```
   operator type() const;
};
```

Preconditions: template argument `T` shall be a complete type.
`value` : defined to be `true` if `T` is an empty class (10). Otherwise defined to be `false`.

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

Preconditions: template argument `T` shall be a complete type.
`value` : defined to be `true` if `T` is a polymorphic class (10.3). Otherwise defined to be `false`.

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

Preconditions: template argument `T` shall be a complete type.
`value` : defined to be `true` if `T` is a abstract class (10.4). Otherwise defined to be `false`.

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

Preconditions: template argument `T` shall be a complete type.
`value` : defined to be `true` if the default constructor for `T` is trivial(12.1). Otherwise defined to be `false`.

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

Preconditions: template argument `T` shall be a complete type.
`value` : defined to be `true` if the copy constructor for `T` is trivial (12.8). Otherwise defined to be `false`.

```
template <class T> struct has_trivial_assign{
   static  const bool value                    = implementation_defined;
   typedef bool                                value_type;
```

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

Preconditions: template argument `T` shall be a complete type.
`value` : defined to be `true` if the assignment operator for `T` is trivial (12.8). Otherwise defined to be `false`.

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

Preconditions: template argument `T` shall be a complete type.
`value` : defined to be `true` if the destructor for `T` is trivial (12.4). Otherwise defined to be `false`.

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

Preconditions: template argument `T` shall be a complete type.
`value` : defined to be `true` if the default constructor for `T` has an empty exception specification, or can otherwise be deduced never to throw an exception. Otherwise defined to be `false`.

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

Preconditions: template argument `T` shall be a complete type.
`value` : defined to be `true` if the copy constructor for `T` has an empty exception specification, or can otherwise be deduced never to throw an exception. Otherwise defined to be `false` .

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

Preconditions: template argument `T` shall be a complete type.
`value` : defined to be `true` if the assignment operator for `T` has an empty exception specification, or can otherwise be deduced never to throw an exception. Otherwise defined to be `false`.

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

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

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

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

4.3 Relationships between types [tr.meta.rel]

All of the templates in this header satisfy the BinaryTypeTrait requirements.
For all of the class templates declared in this clause, all members declared static const shall be defined in such a way that they are usable as integral constant expressions.
For all of the class templates `X` declared in this clause, both rvalues of type `X const` and lvalues of type `X const&` shall be implicitly convertible to `X::type`. Whether this is accomplished by inheritance or by a member operator is unspecified. For exposition only the class templates in this clause are shown with a member conversion operator, `operator type() const`, which shall return —type()— in all cases.

4.3.1 Header `<type_compare>` synopsis [tr.meta.rel.synopsis]

```
namespace tr1 {

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

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

} // namespace tr1
```

Inclusion of the header `<type_compare>` shall make the type `integral_constant`, and its associated typedefs `true_type` and `false_type` defined in `<type_traits>`, visible. Whether any other types declared in `<type_traits>` are made visible by the inclusion of `<type_compare>` is implementation defined.

4.3.2 Type relationships [tr.meta.rel.rel]

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

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

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

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

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

- Type `From`, is a void or incomplete type (3.9).
- Type `To`, is an incomplete, void or abstract type (3.9).
- The conversion is ambiguous, for example if type `From` has multiple base classes of type `To` (10.2).
- Type `To` is of class type and the conversion would invoke a non-public constructor of `To` (11.0 and 12.3.1).
- Type `From` is of class type and the conversion would invoke a non-public conversion operator of `From` (11.0 and 12.3.2).

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

Preconditions: template arguments `Base` and `Derived` shall both be complete types.
`value`: defined to be `true` only if type `Base` is a base class of type `Derived` (10). Otherwise defined to be `false`.

4.4 Transformations between types [tr.meta.trans]

This sub-clause contains templates that may be used to transform one type to another following some predefined rule.
All of the templates in this header satisfy the TransformationTrait requirements.

4.4.1 Header `<type_transform>` synopsis [tr.meta.trans.synopsis]

```
namespace tr1 {

// const-volatile modifications:

template <class T> struct remove_const;
template <class T> struct remove_volatile;
template <class T> struct remove_cv;
template <class T> struct add_const;
template <class T> struct add_volatile;
template <class T> struct add_cv;

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

// array modifications:
template <class T> struct remove_bounds;

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

} // namespace tr
```

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

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

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

`volatile int`, whereas `remove_const<const int*> is const int*`.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`type` : defined to be a type that is the same as `T`, except any top level array bounds have been removed.

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

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

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

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

`type` : defined to be a type that is the result of the expression `&t` , where `t` is an imaginary lvalue of type `T`.

4.5 Implementation requirements [tr.meta.req]

The behaviour of all the class templates defined in `<type_traits>`, `<type_compare>` and

`<type_transform>` shall conform to the specifications given, except where noted below.
[*Note:* The latitude granted to implementers in this clause is temporary, and is expected to be removed in future revisions of this document. *—end note*]
If there is no means by which the implementation can differentiate between class and union types, then the class templates `is_class` and `is_union` need not be provided.
If there is no means by which the implementation can detect polymorphic types, then the class template `is_polymorphic` need not be provided.
If there is no means by which the implementation can detect abstract types, then the class template `is_abstract` need not be provided.
It is unspecified under what circumstances, if any, `is_empty<T>::value` evaluates to `true`.
It is unspecified under what circumstances, if any, `is_pod<T>::value` evaluates to `true`, except that, for all types `T`:

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

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

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

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

Chapter 5

Numerical facilities [tr.num]

5.1 Random number generation [tr.rand]

This subclause defines a facility for generating random numbers.

5.1.1 Requirements [tr.rand.req]

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

Table 5.1: Number generator requirements (in addition to function object)

expression	Return Type	pre/post-condition	complexity
`X::result_type`	`T`	`std::numeric_limits<T>::is_specialized` is `true`	compile-time

In table 5.2, X denotes a uniform random number generator class returning objects of type T, u is a value of X, and v is a (possibly const) value of X.

Table 5.2: Uniform random number generator requirements (in addition to number generator)

expression	Return Type	pre/post-condition	complexity
`u()`	`T`	—	amortized constant
`v.min()`	`T`	Returns some `l` where `l` 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

continued from previous page			
`v.max()`	`T`	If `std::numeric_limits<T>::is_integer`, returns `l` where `l` is less than or equal to all values potentially returned by `operator()`, otherwise, returns `l` where `l` is strictly less 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

In table 5.3, `X` denotes a pseudo-random number engine class returning objects of type `T`, `t` is a value of `T`, `u` is a value of `X`, `v` is an lvalue of `X`, `it1` is an lvalue and `it2` is a (possibly const) value of an input iterator type `It` having an unsigned integral value type, `x`, `y` are (possibly const) values of `X`, `os` is convertible to an lvalue of type `std::ostream`, and `is` is convertible to an lvalue of type `std::istream`.
A pseudo-random number engine `x` has a state `x(i)` at any given time. The specification of each pseudo-random number engines defines the size of its state in multiples of the size of its `result_type`, given as an integral constant expression.

Table 5.3: 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(it1, it2)`	—	creates an engine with the initial state given by the range `[it1,it2)`. `it1` is advanced by the size of state. If the size of the range `[it1,it2)` is insufficient, leaves `it1 == it2` and throws `std::invalid_argument`.	$\mathcal{O}$(size of state)
`u.seed()`	`void`	post: `u == X()`	$\mathcal{O}$(size of state)
`u.seed(it1, it2)`	void	post: If there are sufficiently many values in `[it1, it2)` to initialize the state of `u`, then `u == X(it1,it2)`. Otherwise, `it1 == it2`, throws `std::invalid_argument`, and further use of `u` (except destruction) is undefined until a `seed` member function has been executed without throwing an exception.	$\mathcal{O}$(size of state)

continued from previous page			
`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`	`==` is an equivalence relation. The current state `x(i)` of `x` is equal to the current state `y(j)` of `y`.	$\mathcal{O}$(size of state)
`x != y`	`bool`	`!(x == y)`	$\mathcal{O}$(size of state)
`os << x`	`std::ostream&`	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`	`std::istream&`	sets the state `v(i)` of `v` as determined by reading its textual representation from `is`. post: The `is.`*fmtflags* are unchanged.	$\mathcal{O}$(size of state)

In table 5.4, `X` denotes a random distribution class returning objects of type `T`, `u` is a value of `X`, `x` is a (possibly const) value of `X`, and `e` is an lvalue of an arbitrary type that meets the requirements of a uniform random number generator, returning values of type `U`.

Table 5.4: Random distribution requirements (in addition to number generator, `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`	`std::ostream&`	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)

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

Additional requirements: The sequence of numbers produced by repeated invocations of `x(e)` does not change whether or not `os << x` is invoked between any of the invocations `x(e)`. If a textual representation is written using `os << x` and that representation is restored into the same or a different object `y` of the same type using `is >> y`, repeated invocations of `y(e)` produce the same sequence of random numbers as would repeated invocations of `x(e)`.
In the following subclauses, a template parameter named `UniformRandomNumberGenerator` shall denote a class 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. Furthermore, a template parameter named `RealType` shall denote a type that holds an approximation to a real number. This type shall meet the requirements for a numeric type ([lib.numeric.requirements]), the binary operators `+`, `-`, `*`, `/` shall be applicable to it, a conversion from `double` shall exist, and function signatures corresponding to those for type `double` in subclause [lib.c.math] shall be available by argument-dependent lookup ([basic.lookup.koenig]). [*Note:* The built-in floating-point types `float` and `double` meet these requirements. *—end note*]

5.1.2 Header `<random>` synopsis [tr.rand.synopsis]

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

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

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

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

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

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

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

  class random_device;

  template<class IntType = int>
  class uniform_int;

  template<class RealType = double>
  class bernoulli_distribution;

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

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

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

  template<class RealType = double>
  class uniform_real;

  template<class RealType = double>
  class exponential_distribution;

  template<class RealType = double>
  class normal_distribution;

  template<class RealType = double>
  class gamma_distribution;

} // namespace tr1
```

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

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

```
template<class Engine, class Distribution>
class variate_generator
{
public:
  typedef Engine engine_type;
```

```
  typedef implementation defined engine_value_type;
  typedef Distribution distribution_type;
  typedef typename Distribution::result_type result_type;

  variate_generator(engine_type eng, distribution_type d);

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

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

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

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

The template argument for the parameter `Engine` shall be of the form *U*, *U*&, or *U**, where *U* denotes a class that satisfies all the requirements of a uniform random number generator. The member `engine_value_type` shall name *U*.
Specializations of `variate_generator` satisfy the requirements of CopyConstructible. They also satisfy the requirements of Assignable unless the template parameter `Engine` is of the form *U*&.
The complexity of all functions specified in this section is constant. No function described in this section except the constructor throws an exception.

```
variate_generator(engine_type eng, distribution_type d)
```

Effects: Constructs a `variate_generator` object with the associated uniform random number generator `eng` and the associated random distribution `d`.
Throws: If and what the copy constructor of Engine or Distribution throws.

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

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

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

```
engine_value_type& engine()
```

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

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

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

```
distribution_type& distribution()
```

Returns: A reference to the associated random distribution.

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

Returns: A reference to the associated random distribution.

```
result_type min() const
```

Precondition: `distribution().min()` is
Returns: `distribution().min()`

```
result_type max() const
```

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

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

Except where specified otherwise, the complexity of all functions specified in the following sections is constant. No function described in this section except the constructor and seed functions taking an iterator range [it1,it2) throws an exception.
The class templates specified in this section satisfy all the requirements of a pseudo-random number engine (given in tables in section x.x), except where specified otherwise. Descriptions are provided here only for operations on the engines that are not described in one of these tables or for operations where there is additional semantic information.
All members declared `static const` in any of the following class templates shall be defined in such a way that they are usable as integral constant expressions.

5.1.4.1 Class template `linear_congruential` [tr.rand.eng.lcong]

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

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

    // parameter values
```

```
    static const IntType multiplier = a;
    static const IntType increment = c;
    static const IntType modulus = m;

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

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

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

The template parameter `IntType` shall denote an integral type large enough to store values up to (m-1). If the template parameter `m` is 0, the behaviour is implementation-defined. Otherwise, the template parameters `a` and `c` shall be less than m.
The size of the state x(i) is 1.

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

Requires: `c > 0 || (x0 % m) > 0`
Effects: Constructs a `linear_congruential` engine with state x(0) := `x0` mod m.

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

Requires: `c > 0 || (x0 % m) > 0`
Effects: Sets the state x(i) of the engine to `x0` mod m.

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

Requires: `c > 0` — *first ¿ 0—
Effects: Sets the state x(i) of the engine to `*first` mod m.
Complexity: Exactly one dereference of `*first`.

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

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

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

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

- y(i) = (x(i-n) *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(UIntType value);
  template<class In> mersenne_twister(In& first, In last);
  void seed();
  void seed(UIntType value);
  template<class In> void seed(In& first, In last);
  result_type min() const;
  result_type max() const;
  result_type operator()();
};

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

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

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

The template parameter `UIntType` shall denote an unsigned integral type large enough to store

values up to $2^{\mathtt{w}} - 1$. Also, the following relations shall hold: $1 \leq m \leq n$. $0 \leq r, u, s, t, l \leq w$. $0 \leq a, b, c \leq 2^w - 1$.
The size of the state x(i) is n.

```
mersenne_twister()
```

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

```
explicit mersenne_twister(result_type value)
```

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

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

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

```
void seed()
```

Effects: Invokes `seed(4357)`.

```
void seed(result_type value)
```

Requires: `value > 0`
Effects: With a linear congruential generator l(i) having parameters $m_l = 2^32$, $a_l = 69069$, $c_l = 0$, and $l(0) =$ `value`, sets x(-n) ... x(-1) to l(1) ... l(n), respectively.
Complexity: $\mathcal{O}$(n)

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

Effects: Given the values $z_0 \ldots z_{n-1}$ obtained by dereferencing [first, first+n), sets x(-n) ... x(-1) to $z_0 \mathrm{mod} 2^w \ldots z_{n-1} \mathrm{mod} 2^w$.
Complexity: Exactly n dereferences of `first`.

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

Returns: `x(i-n) == y(j-n)` and ... and `x(i-1) == y(j-1)`
Notes: Assumes the next output of x is o(x(i)) and the next output of y is o(y(j)).
Complexity: $\mathcal{O}$(n)

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

Effects: Writes x(i-n), ... x(i-1) to os, in that order.
Complexity: $\mathcal{O}$(n)

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

A `subtract_with_carry` engine produces integer random numbers using x(i) = (x(i-s) - x(i-r) - carry(i-1)) mod m; carry(i) = 1 if x(i-s) - x(i-r) - carry(i-1) ¡ 0, else 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(IntType value);
  template<class In> subtract_with_carry(In& first, In last);
  void seed(IntType value = 19780503);
  template<class In> void seed(In& first, In last);
  result_type min() const;
  result_type max() const;
  result_type operator()();
};

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

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

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

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

The template parameter `IntType` shall denote a signed integral type large enough to store values up to m-1. The following relation shall hold: $0 < \texttt{s} < \texttt{r}$. Let w the number of bits in the binary representation of m.
The size of the state is `r`.

```
subtract_with_carry()
```

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

```
explicit subtract_with_carry(IntType value)
```

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

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

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

```
void seed(IntType value = 19780503)
```

Requires: `value > 0`
Effects: With a linear congruential generator l(i) having parameters $m_l = 2147483563$, $a_l = 40014$, $c_l = 0$, and $l(0) =$ `value`, sets x(-r) ... x(-1) to l(1) mod m ... l(r) mod m, respectively. If x(-1) == 0, sets carry(-1) = 1, else sets carry(-1) = 0.
Complexity: $\mathcal{O}$(r)

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

Effects: With $n = w/32 + 1$ (rounded downward) and given the values $z_0 \ldots z_{n*r-1}$ obtained by dereferencing `[first, first+n*r)`, 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.
Complexity: Exactly `r*n` dereferences of `first`.

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

Returns: `x(i-r) == y(j-r)` and ... and `x(i-1) == y(j-1)`.
Notes: Assumes the next output of `x` is x(i) and the next output of `y` is y(j).
Complexity: $\mathcal{O}$(r)

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

Effects: Writes x(i-r) ... x(i-1), carry(i-1) to `os`, in that order.
Complexity: $\mathcal{O}$(r)

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

A `subtract_with_carry_01` engine produces floating-point random numbers using x(i) = (x(i-s) - x(i-r) - carry(i-1)) mod 1; 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 int value);
  template<class In> subtract_with_carry_01(In& first, In last);
  void seed(unsigned int value = 19780503);
  template<class In> void seed(In& first, In last);
  result_type min() const;
  result_type max() const;
  result_type operator()();
};
template<class RealType, int w, int s, int r>
bool operator==(const subtract_with_carry_01<RealType, w, s, r> x,
                const subtract_with_carry_01<RealType, w, s, r> y);

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

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

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

The following relation shall hold: $0 < \texttt{s} < \texttt{r}$.
The size of the state is `r`.

```
subtract_with_carry_01()
```

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

```
explicit subtract_with_carry_01(unsigned int value)
```

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

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

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

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

Effects: With a linear congruential generator l(i) having parameters m = 2147483563, a = 40014, c = 0, and l(0) = `value`, sets x(-r) ... x(-1) to $(l(1) \cdot 2^{-w}) \bmod 1 \ldots (l(r) \cdot 2^{-w}) \bmod 1$, respectively. If x(-1) == 0, sets carry(-1) = 2^{-w}, else sets carry(-1) = 0.
Complexity: $\mathcal{O}$(r)

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

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

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

Returns: true, if and only if x(i-r) == y(j-r) and ... and x(i-1) == y(j-1).
Complexity: $\mathcal{O}$(r)

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

Effects: Write x(i-r)*2^w ... x(i-1)*2^w, carry(i-1)*2^w to `os`, in that order.
Complexity: $\mathcal{O}$(r)

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

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

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

  // parameter values
```

```
  static const int block_size = p;
  static const int used_block = r;

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

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

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

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

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

The template parameter `UniformRandomNumberGenerator` shall denote a class that satisfies all the requirements of a uniform random number generator, given in table 5.2 in clause 5.1.1. `r` $\leq |p|$. The size of the state is the size of *b* plus 1.

`discard_block()`

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

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

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

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

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

```
void seed()
```

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

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

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

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

Returns: *b*

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

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

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

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

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

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

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

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

  // constructors and member function
```

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

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

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

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

The template parameters `UniformRandomNumberGenerator1` and `UniformRandomNumberGenerator1`

shall denote classes that satisfy all the requirements of a uniform random number generator, given in table 5.2 in clause 5.1.1. The size of the state is the size of *b1* plus the size of *b2*.

`xor_combine()`

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

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

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

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

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

`void seed()`

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

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

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

`const base1_type& base1() const`

Returns: *b1*

`const base2_type& base2() const`

Returns: *b2*

`result_type operator()()`

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

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

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

5.1.5 Engines with predefined parameters [tr.rand.predef]

```
typedef linear_congruential<implementation-defined,
                            16807, 0, 2147483647>
        minstd_rand0;
typedef linear_congruential<implementation-defined,
                            48271, 0, 2147483647>
```

```
        minstd_rand;

typedef mersenne_twister<implementation-defined,
                         32,624,397,31,0x9908b0df,11,7,0x9d2c5680,15,0xefc60000,18>
        mt19937;

typedef subtract_with_carry_01<float, 24, 10, 24> ranlux_base_01;
typedef subtract_with_carry_01<double, 48, 10, 24> ranlux64_base_01;

typedef discard_block<subtract_with_carry<implementation-defined,
                                          (1<<24), 10, 24>, 223, 24>
        ranlux3;
typedef discard_block<subtract_with_carry<implementation-defined,
                                          (1<<24), 10, 24>, 389, 24>
        ranlux4;

typedef discard_block<subtract_with_carry_01<float, 24, 10, 24>, 223, 24>
        ranlux3_01;
typedef discard_block<subtract_with_carry_01<float, 24, 10, 24>, 389, 24>
        ranlux4_01;
```

For a default-constructed `minstd_rand0` object, x(10000) = 1043618065. For a default-constructed `minstd_rand` object, x(10000) = 399268537.
For a default-constructed `mt19937` object, x(10000) = 3346425566.
For a default-constructed `ranlux3` object, x(10000) = 5957620. For a default-constructed `ranlux4` object, x(10000) = 8587295. For a default-constructed `ranlux3_01` object, x(10000) = $5957620 \cdot 2^{-24}$. For a default-constructed `ranlux4_01` object, x(10000) = $8587295 \cdot 2^{-24}$.

5.1.6 Class `random_device` [tr.rand.device]

A `random_device` produces non-deterministic random numbers. It satisfies all the requirements of a uniform random number generator (given in table 5.2 in clause 5.1.1). Descriptions are provided here only for operations on the engines that are not described in one of these tables or for operations where there is additional semantic information.
If implementation limitations prevent generating non-deterministic random numbers, the implementation can employ a pseudo-random number engine.

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

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

```
  result_type operator()();

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

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

Effects: Constructs a `random_device` non-deterministic random number engine. The semantics and default value of the `token` parameter are implementation-defined.[1]
Throws: A value of some type derived from `exception` if the `random_device` could not be initialized.

```
result_type min() const
```

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

```
result_type max() const
```

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

```
double entropy() const
```

Returns: An entropy estimate for the random numbers returned by operator(), in the range `min()` to $\log_2(\texttt{max}() + 1)$. A deterministic random number generator (e.g. a pseudo-random number engine) has entropy 0.
Throws: Nothing.

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

Returns: A non-deterministic random value, uniformly distributed between `min()` and `max()`, inclusive. It is implementation-defined how these values are generated.
Throws: A value of some type derived from `exception` if a random number could not be obtained.

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

The class templates specified in this section satisfy all the requirements of a random distribution (given in tables in clause 5.1.1). Descriptions are provided here only for operations on the distributions that are not described in one of these tables or for operations where there is additional semantic information.
A template parameter named `IntType` shall denote a type that represents an integer number. This type shall meet the requirements for a numeric type ([lib.numeric.requirements]), the binary operators `+`, `-`, `*`, `/`, `%` shall be applicable to it, and a conversion from `int` shall exist.[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.
No function described in this section throws an exception, unless an operation on values of `IntType` or `RealType` throws an exception. [*Note:* Then, the effects are undefined, see [lib.numeric.requirements —*end note*].]

[1] The parameter is intended to allow an implementation to differentiate between different sources of randomness.
[2] The built-in types `int` and `long` meet these requirements.

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

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

A `uniform_int` random distribution produces integer random numbers x in the range `min` $\leq x \leq$ `max`, with equal probability. `min` and `max` are the parameters of the distribution.
A `uniform_int` random distribution satisfies all the requirements of a uniform random number generator (given in table 5.2 in clause 5.1.1).

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

  // constructors and member function
  explicit uniform_int(IntType min = 0, IntType max = 9);
  result_type min() const;
  result_type max() const;
  void reset();
  template<class UniformRandomNumberGenerator>
  result_type operator()(UniformRandomNumberGenerator& urng);
  template<class UniformRandomNumberGenerator>
  result_type operator()(UniformRandomNumberGenerator& urng, result_type n);
};

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

Requires: `min` $\leq$ `max`
Effects: Constructs a `uniform_int` object. `min` and `max` are the parameters of the distribution.

```
result_type min() const
```

Returns: The "min" parameter of the distribution.

```
result_type max() const
```

Returns: The "max" parameter of the distribution.

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

Returns: A uniform random number x in the range 0 ¡= x ¡ n. [*Note:* This allows a `variate_generator` object with a `uniform_int` distribution to be used with std::random_shuffe, see [lib.alg.random.shuffle]. *—end note*]

5.1.7.2 Class template `bernoulli_distribution` [tr.rand.dist.bern]

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.

```
template<class RealType = double>
class bernoulli_distribution
{
public:
  // types
  typedef int input_type;
  typedef bool result_type;

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

bernoulli_distribution(const RealType& p = RealType(0.5))
```

Requires: $0 \leq$ p ≤ 1
Effects: Constructs a `bernoulli_distribution` object. p is the parameter of the distribution.

```
RealType p() const
```

Returns: The "p" parameter of the distribution.

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

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

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

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

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

Requires: $0 <$ p < 1
Effects: Constructs a `geometric_distribution` object; p is the parameter of the distribution.

```
RealType p() const
```

Returns: The "p" parameter of the distribution.

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

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

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

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

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

Requires: *mean* > 0

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

```
RealType mean() const
```

Returns: The *mean* parameter of the distribution.

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

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

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

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

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

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

Requires: $0 \leq$ p ≤ 1 and t ≥ 0.
Effects: Constructs a binomial_distribution object; t and p are the parameters of the distribution.

```
IntType t() const
```

Returns: The "t" parameter of the distribution.

```
RealType p() const
```

Returns: The "p" parameter of the distribution.

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

A uniform_real random distribution produces floating-point random numbers x in the range min $\leq x \leq$ max, with equal probability. min and max are the parameters of the distribution.
A uniform_real random distribution satisfies all the requirements of a uniform random number generator (given in table 5.2 in clause 5.1.1).

```
template<class RealType = double>
class uniform_real
{
public:
  // types
  typedef RealType input_type;
  typedef RealType result_type;

  // constructors and member function
  explicit uniform_real(RealType min = RealType(0), RealType max = RealType(1));
  result_type min() const;
  result_type max() const;
  void reset();
  template<class UniformRandomNumberGenerator>
  result_type operator()(UniformRandomNumberGenerator& urng);
};

uniform_real(RealType min = RealType(0), RealType max = RealType(1))
```

Requires: min $\leq |max|$.
Effects: Constructs a uniform_real object; min and max are the parameters of the distribution.

```
result_type min() const
```

Returns: The "min" parameter of the distribution.

```
result_type max() const
```

Returns: The "max" parameter of the distribution.

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

An `exponential_distribution` random distribution produces random numbers $x > 0$ distributed with probability density function $p(x) = \lambda e^{-\lambda x}$, where 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))
```

Requires: lambda > 0.
Effects: Constructs an `exponential_distribution` object with `rng` as the reference to the underlying source of random numbers. `lambda` is the parameter for the distribution.

```
RealType lambda() const
```

Returns: The "λ" parameter of the distribution.

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

A `normal_distribution` random distribution produces random numbers x distributed with probability density function $(1/\sqrt{2\pi\sigma})e^{-(x-mean)^2/(2\sigma^2)}$, where *mean* and 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);
```

Requires: sigma > 0.
Effects: Constructs a `normal_distribution` object; `mean` and `sigma` are the parameters for the distribution.

```
RealType mean() const
```

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

```
RealType sigma() const
```

Returns: The "σ" parameter of the distribution.

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

A `gamma_distribution` random distribution produces random numbers x distributed with probability density function $p(x) = 1/\Gamma(\alpha)x^{\alpha-1}e^{-x}$, where 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));
```

Requires: alpha > 0.
Effects: Constructs a `gamma_distribution` object; `alpha` is the parameter for the distribution.

```
RealType alpha() const
```

Returns: The "α" parameter of the distribution.

5.2 Mathematical special functions [tr.num.sf]

5.2.1 Additions to header <cmath> synopsis [tr.math.sf.syn]

The following table summarizes the additional signatures in header <cmath>.

Table 5.5: Summary of additions to header <cmath>

Type	Name(s)		
Functions:			
bessel_I	ellint_E	hermite	neumann_N
bessel_J	ellint_E2	hyperg_1F1	neumann_n
bessel_K	ellint_F	hyperg_2F1	sph_Y
bessel_j	ellint_K	laguerre_0	zeta
beta	ellint_P	laguerre_m	
ei	ellint_P2	legendre_Pl	
		legendre_Plm	

Each of these functions is provided for arguments of type float, double, and long double. The signatures added to header <cmath> are:

```
// cylindrical Bessel functions (of the first kind):
double       bessel_J( double l, double x );
float        bessel_J( float l, float  x );
long double  bessel_J( long double l, long double x );

// cylindrical Neumann functions;
// cylindrical Bessel functions (of the second kind):
double       neumann_N( double l, double x );
float        neumann_N( float l, float  x );
long double  neumann_N( long double l, long double x );

// regular modified cylindrical Bessel functions:
double       bessel_I( double l, double x );
float        bessel_I( float l, float  x );
long double  bessel_I( long double l, long double x );

// irregular modified cylindrical Bessel functions:
double       bessel_K( double l, double x );
float        bessel_K( float l, float  x );
long double  bessel_K( long double l, long double x );

// spherical Bessel functions (of the first kind):
double       bessel_j( double l, double x );
float        bessel_j( float l, float  x );
long double  bessel_j( long double l, long double x );
```

```
// spherical Neumann functions;
// spherical Bessel functions (of the second kind):
double       neumann_n( double l, double x );
float        neumann_n( float l, float  x );
long double  neumann_n( long double l, long double x );

// Legendre polynomials:
double       legendre_Pl( unsigned l, double x )
float        legendre_Pl( unsigned l, float x )
long double  legendre_Pl( unsigned l, long double x )

// associated Legendre functions:
double       legendre_Plm( unsigned l, unsigned m, double x )
float        legendre_Plm( unsigned l, unsigned m, float x )
long double  legendre_Plm( unsigned l, unsigned m, long double x )

// spherical harmonics:
double       sph_Y( unsigned l, int m, double theta, double phi )
float        sph_Y( unsigned l, int m, float theta, float phi )
long double  sph_Y( unsigned l, int m, long double theta, long double phi )

// Hermite polynomials:
double       hermite( unsigned n, double x );
float        hermite( unsigned n, float x );
long double  hermite( unsigned n, long double x );

// Laguerre polynomials:
double       laguerre_0(unsigned n, double x);
float        laguerre_0(unsigned n, float x);
long double  laguerre_0(unsigned n, long double x);

// associated Laguerre polynomials:
double       laguerre_m(unsigned n, unsigned m, double x);
float        laguerre_m(unsigned n, unsigned m, float x);
long double  laguerre_m(unsigned n, unsigned m, long double x);

// hypergeometric functions:
double       hyperg_2F1(double a, double b, double c, double x) ;
float        hyperg_2F1(float a, float b, float c, float x) ;
long double  hyperg_2F1(long double a, long double b, long double c, long double x)

// confluent hypergeometric functions:
double       hyperg_1F1(double a, double c, double x) ;
float        hyperg_1F1(float a, float c, float x) ;
long double  hyperg_1F1(long double a, long double c, long double x) ;
```

```
// (incomplete) elliptic integral of the first kind:
double       ellint_F( double k, double phi );
float        ellint_F( float k, float phi );
long double  ellint_F( long double k, long double phi );

// (complete) elliptic integral of the first kind:
double       ellint_K( double k );
float        ellint_K( float k );
long double  ellint_K( long double k );

// (incomplete) elliptic integral of the second kind:
double       ellint_E( double k, double phi );
float        ellint_E( float k, float phi );
long double  ellint_E( long double k, long double phi );

// (complete) elliptic integral of the second kind:
double       ellint_E2( double k );
float        ellint_E2( float k );
long double  ellint_E2( long double k );

// (incomplete) elliptic integral of the third kind:
double       ellint_P( double k, double n, double phi );
float        ellint_P( float k, float n, float phi );
long double  ellint_P( long double k, long double n, long double phi );

// (complete) elliptic integral of the third kind:
double       ellint_P2( double k, double n, double phi );
float        ellint_P2( float k, float n, float phi );
long double  ellint_P2( long double k, long double n, long double phi );

// beta function:
double       beta( double x, double y );
float        beta( float x, float y );
long double  beta( long double x, long double y );

// exponential integral:
double       ei( double );
float        ei( float );
long double  ei( long double );

// Riemann zeta function:
double       zeta( double );
float        zeta( float );
long double  zeta( long double );
```

5.2.2 cylindrical Bessel functions (of the first kind) [tr.math.sf.J]

```
double       bessel_J( double l, double x );
float        bessel_J( float l, float  x );
long double  bessel_J( long double l, long double x );
```

Effects: These functions compute the cylindrical Bessel functions of the first kind (as defined by Abramowitz and Stegun [1], §9.1.10, *etc.*) of their respective arguments `l` and `x`. A domain error occurs if `l` is less than zero. A range error (due to underflow) occurs if `x` is too close to any of the roots of the `bessel_J` function.
Returns: The `bessel_J` functions return $J_l(x)$ as denoted in ISO:31 [4], Item No. 11-14.1.

5.2.3 cylindrical Neumann functions [tr.math.sf.N]

```
double       neumann_N( double l, double x );
float        neumann_N( float l, float  x );
long double  neumann_N( long double l, long double x );
```

Effects: These functions compute the cylindrical Neumann functions (as defined by Abramowitz and Stegun [1], §9.1.2, *etc.*), also known as the cylindrical Bessel functions of the second kind, of their respective arguments `l` and `x`. A domain error occurs if `x` is less than or equal to zero. A range error (due to overflow) occurs (a) if the magnitude of `l` is too large, or (b) if `x` is too small. A range error (due to underflow) occurs if `x` is too close to any of the roots of the `neumann_N` function.
Returns: The `neumann_N` functions return $N_l(x)$ as denoted in ISO:31 [4], Item No. 11-14.2.

5.2.4 regular modified cylindrical Bessel functions [tr.math.sf.I]

```
double       bessel_I( double l, double x );
float        bessel_I( float l, float  x );
long double  bessel_I( long double l, long double x );
```

Effects: These functions compute the regular modified cylindrical Bessel functions (as defined by Abramowitz and Stegun [1], §9.6.3, *etc.*) of their respective arguments `l` and `x`. A domain error occurs if `l` is not an integer and either (a) `l` is less than or equal to zero or (b) `x` is less than or equal to zero. A range error occurs if `x` is too large.
Returns: The `bessell_I` functions return $I_l(x)$ as denoted in ISO:31 [4], Item No. 11-14.4.

5.2.5 irregular modified cylindrical Bessel functions [tr.math.sf.K]

```
double       bessel_K( double l, double x );
float        bessel_K( float l, float  x );
long double  bessel_K( long double l, long double x );
```

Effects: These functions compute the irregular modified cylindrical Bessel functions (as defined by Abramowitz and Stegun [1], §9.6.4, *etc.*) of their respective arguments `l` and `x`. A domain error occurs if `l` is not an integer and either (a) `l` is less than or equal to zero or (b) `x` is less than or equal to zero. A range error occurs (a) if the magnitude of `l` is too large, or (b) if `x` is too small.
Returns: The `bessell_K` functions return $K_l(x)$ as denoted in ISO:31 [4], Item No. 11-14.4.

5.2.6 spherical Bessel functions (of the first kind) [tr.math.sf.j]

```
double       bessel_j( double l, double x );
float        bessel_j( float l, float  x );
long double  bessel_j( long double l, long double x );
```

Effects: These functions compute the spherical Bessel functions of the first kind (as defined by Abramowitz and Stegun [1], §10.1.1, *etc.*) of their respective arguments `l` and `x`. A domain error occurs if `l` is less than zero. A range error (due to overflow) occurs if `l` equals zero and the magnitude of `x` is too small. A range error (due to underflow) occurs if `x` is too close to any of the roots of the `bessel_j` function.
Returns: The `bessell_j` functions return $j_l(x)$ as denoted in ISO:31 [4], Item No. 11-14.5.

5.2.7 spherical Neumann functions [tr.math.sf.n]

```
double       neumann_n( double l, double x );
float        neumann_n( float l, float  x );
long double  neumann_n( long double l, long double x );
```

Effects: These functions compute the spherical Neumann functions (as defined by Abramowitz and Stegun [1], §10.1.1, *etc.*), also known as the spherical Bessel functions of the second kind, of their respective arguments `x`. A domain error occurs if `x` is less than or equal to zero. A range error (due to overflow) occurs (a) if the magnitude of `l` is too large, or (b) if the magnitude of `x` is too small. A range error (due to underflow) occurs if `x` is too close to any of the roots of the `neumann_n` function.
Returns: The `neumann_n` functions return $n_l(x)$ as denoted in ISO:31 [4], Item No. 11-14.6.

5.2.8 Legendre polynomials [tr.math.sf.Pl]

```
double       legendre_Pl( unsigned l, double x )
float        legendre_Pl( unsigned l, float x )
long double  legendre_Pl( unsigned l, long double x )
```

Effects: These functions compute the Legendre polynomials (as defined by Abramowitz and Stegun [1] §8.1.1, *etc.*) of their respective arguments `l` and `x`. A domain error occurs if `x` is greater than one. A range error (due to underflow) occurs if `x` is too close to any of the roots of the `legendre_Pl` function.
Returns: The `legendre_Pl` functions return $P_l(x)$ as denoted in ISO:31 [4], Item No. 11-14.8.

5.2.9 associated Legendre functions [tr.math.sf.Plm]

```
double       legendre_Plm( unsigned l, unsigned m, double x )
float        legendre_Plm( unsigned l, unsigned m, float x )
long double  legendre_Plm( unsigned l, unsigned m, long double x )
```

Effects: These functions compute the associated Legendre functions (as defined by Abramowitz and Stegun [1], §8.1.1, *etc.*) of their respective arguments `l`, `m`, and `x`. A domain error occurs (a) if `m` is greater than `l`, or (b) if `x` is greater than one. A range error (due to overflow) occurs if `l` is too large. A range error (due to underflow) occurs if `x` is too close to any of the roots of the `legendre_Plm` function.
Returns: The `legendre_Plm` functions return $P_l^m(x)$ as denoted in ISO:31 [4], Item No. 11-14.9.

5.2.10 spherical harmonics [tr.math.sf.Ylm]

```
double       sph_Y( unsigned l, int m, double theta, double phi )
float        sph_Y( unsigned l, int m, float theta, float phi )
long double  sph_Y( unsigned l, int m, long double theta, long double phi )
```

Effects: These functions compute spherical harmonic functions (as defined by Abramowitz and Stegun [1], §8.1.1 footnote 2, *etc.*) of their respective arguments `l`, `m`, `theta`, and `phi`. A domain error occurs if the magnitude of `m` is greater than `l`. A range error (due to overflow) occurs if `l` is too large. A range error (due to underflow) occurs if either `theta` or `phi` is too close to any of the roots of the `sph_Y` function.
Returns: The `sph_Y` functions return $Y_l^m(\phi, \theta)$ as denoted in ISO:31 [4], Item No. 11-14.10.

5.2.11 Hermite polynomials [tr.math.sf.Hn]

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

Effects: These functions compute the Hermite polynomials (as defined by Abramowitz and Stegun [1], §13.6.17 and -.18, *etc.*) of their respective arguments `n` and `x`. A range error (due to overflow) occurs if the magnitude of `x` is too large. A range error (due to underflow) occurs if `x` is too close to any of the roots of the `hermite` function.
Returns: The `hermite` functions return $H_n(x)$ as denoted in ISO:31 [4], Item No. 11-14.11.

5.2.12 Laguerre polynomials [tr.math.sf.Ln]

```
double       laguerre_0(unsigned n, double x);
float        laguerre_0(unsigned n, float x);
long double  laguerre_0(unsigned n, long double x);
```

Effects: These functions compute the Laguerre polynomials (as defined by Abramowitz and Stegun [1], §13.6.9, *etc.*) of their respective arguments `n` and `x`. A range error occurs (due to overflow) if the magnitude of `x` is too large. A range error (due to underflow) occurs if `x` is too close to any of the roots of the `laguerre_0` function.
Returns: The `laguerre_0` functions return $L_n(x)$ as denoted in ISO:31 [4], Item No. 11-14.12.

5.2.13 associated Laguerre polynomials [tr.math.sf.Lnm]

```
double       laguerre_m(unsigned n, unsigned m, double x);
float        laguerre_m(unsigned n, unsigned m, float x);
long double  laguerre_m(unsigned n, unsigned m, long double x);
```

Effects: These functions compute the associated Laguerre polynomials (as defined by Abramowitz and Stegun [1], §13.6.9, *etc.*) of their respective arguments `n`, `m`, and `x`. A domain error occurs if `m` is greater than `n`. A range error (due to overflow) occurs if the magnitude of `x` is too large. A range error (due to underflow) occurs if `x` is too close to any of the roots of the `laguerre_m` function.
Returns: The `laguerre_m` functions return $L_n^m(x)$ as denoted in ISO:31 [4], Item No. 11-14.13.

5.2.14 hypergeometric functions [tr.math.sf.hyper]

```
double       hyperg_2F1(double a, double b, double c, double x) ;
float        hyperg_2F1(float a, float b, float c, float x) ;
long double  hyperg_2F1(long double a, long double b, long double c, long double x)
```

Effects: These functions compute the hypergeometric functions (as defined by Abramowitz and Stegun [1], §15.1.1, *etc.*) of their respective arguments `a`, `b`, `c`, and `x`. A domain error occurs if the magnitude of `x` is greater than or equal to one. A range error (due to overflow) occurs if the magnitude of `x` is too close to one at the same time that `c-a-b` is an integer. A range error (due to underflow) occurs if `x` is too close to any of the roots of the `hyperg_2F1` function.
Returns: The `hyperg_2F1` functions return $F(a, b; c; x)$ as denoted in ISO:31 [4], Item No. 11-14.14.

5.2.15 confluent hypergeometric functions [tr.math.sf.conhyp]

```
double       hyperg_1F1(double a, double c, double x) ;
float        hyperg_1F1(float a, float c, float x) ;
long double  hyperg_1F1(long double a, long double c, long double x) ;
```

Effects: These functions compute the confluent hypergeometric functions (as defined by Abramowitz and Stegun [1], §13.1.2, *etc.*) of their respective arguments `a`, `c`, and `x`. A domain error occurs (a) if `c` is a negative integer, or (b) if `c` is zero. A range error (due to overflow) occurs if the magnitude of `x` is too large. A range error (due to underflow) occurs if `x` is too close to any of the roots of the `hyperg_1F1` function.
Returns: The `hyperg_1F1` functions return $F(a; c; x)$ as denoted in ISO:31 [4], Item No. 11-14.15.

5.2.16 (incomplete) elliptic integral of the first kind [tr.math.sf.ellF]

```
double       ellint_F( double k, double phi );
float        ellint_F( float k, float phi );
long double  ellint_F( long double k, long double phi );
```

Effects: These functions compute the incomplete elliptic integral of the first kind (as defined by Abramowitz and Stegun [1], §17.2.6, *etc.*) of their respective arguments `k` and `phi`. A domain error occurs (a) if `k` is less than or equal to zero, or (b) if `k` is greater than or equal to one, or (c) if `phi` is negative. A range error occurs if the magnitude of `k` is too close to one.
Returns: The `ellint_F` functions return $F(k, \phi)$ as denoted in ISO:31 [4], Item No. 11-14.16.

5.2.17 (complete) elliptic integral of the first kind [tr.math.sf.ellK]

```
double       ellint_K( double k );
float        ellint_K( float k);
long double  ellint_K( long double k);
```

Effects: These functions compute the complete elliptic integral of the first kind (as defined by Abramowitz and Stegun [1], §17.3.1, *etc.*) of their respective arguments `k`. A domain error occurs (a) if `k` is less than or equal to zero, or (b) if `k` is greater than or equal to one. A range error occurs if the magnitude of `k` is too close to one.
Returns: The `ellint_K` functions return $K(k)$ as denoted in ISO:31 [4], Item No. 11-14.16.

5.2.18 (incomplete) elliptic integral of the second kind [tr.math.sf.ellE]

```
double       ellint_E( double k, double phi );
float        ellint_E( float k, float phi );
long double  ellint_E( long double k, long double phi );
```

Effects: These functions compute the incomplete elliptic integral of the second kind (as defined by Abramowitz and Stegun [1], §17.2.9, *etc.*) of their respective arguments `k` and `phi`. A domain error occurs (a) if `k` is less than or equal to zero, or (b) if `k` is greater than or equal to one, or (c) if `phi` is negative. A range error occurs if the magnitude of `k` is too close to one.
Returns: The `ellint_E` functions return $E(k, \phi)$ as denoted in ISO:31 [4], Item No. 11-14.17.

5.2.19 (complete) elliptic integral of the second kind [tr.math.sf.ellEx]

```
double       ellint_E2( double k );
float        ellint_E2( float k );
long double  ellint_E2( long double k );
```

Effects: These functions compute the complete elliptic integral of the second kind (as defined by Abramowitz and Stegun [1], §17.2.9, *etc.*) of their respective arguments `k` and `phi`. A domain error occurs (a) if `k` is less than or equal to zero, or (b) if `k` is greater than or equal to one. A range error occurs if the magnitude of `k` is too close to one.
Returns: The `ellint_E2` functions return $E(k)$ as denoted in ISO:31 [4], Item No. 11-14.17.

5.2.20 (incomplete) elliptic integral of the third kind [tr.math.sf.ellP]

```
double       ellint_P( double k, double n, double phi );
float        ellint_P( float k, float n, float phi );
long double  ellint_P( long double k, long double n, long double phi );
```

Effects: These functions compute the incomplete elliptic integral of the third kind (as defined by Abramowitz and Stegun [1], §17.7.1, *etc.*) of their respective arguments `k`, `n`, and `phi`. A domain error occurs (a) if `k` is less than or equal to zero, or (b) if `k` is greater than or equal to one, or (c) if `phi` is negative. A range error occurs if the magnitude of `k` is too close to one.
Returns: The `ellint_P` functions return $\Pi(k, n, \phi)$ as denoted in ISO:31 [4], Item No. 11-14.18.

5.2.21 (complete) elliptic integral of the third kind [tr.math.sf.ellPx]

```
double       ellint_P2( double k, double n );
float        ellint_P2( float k, float n );
long double  ellint_P2( long double k, long double n );
```

Effects: These functions compute the complete elliptic integral of the third kind (as defined by Abramowitz and Stegun [1], §17.7.2, *etc.*) of their respective arguments `k`, `n`, and `phi`. A domain error occurs (a) if `k` is less than or equal to zero, or (b) if `k` is greater than or equal to one. A range error occurs if the magnitude of `k` is too close to one.
Returns: The `ellint_P2` functions return $\Pi(k, n, \pi/2)$ as denoted in ISO:31 [4], Item No. 11-14.18.

5.2.22 beta function [tr.math.sf.beta]

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

Effects: These functions compute the beta function (as defined by Abramowitz and Stegun [1], §6.2 and §6.1]) of their respective arguments `x` and `y`. A domain error occurs (a) if either `x` or `y` is a

negative integer, or (b) if either x or y is zero. A range error occurs if the magnitude of x or the magnitude of y is too large or too small.
Returns: The beta functions return $B(x, y)$ as denoted in ISO:31 [4], Item No. 11-14.20.

5.2.23 exponential integral [tr.math.sf.ei]

```
double       ei( double x );
float        ei( float x );
long double  ei( long double x );
```

Effects: These functions compute the exponential integral (as defined by Abramowitz and Stegun [1], §5.1.1, *etc.*) of their respective arguments x. A domain error occurs if x is less than or equal to zero. A range error (due to overflow) occurs (a) if x is to small, or (b) if x x is too large. A range error (due to underflow) occurs if x is too close to the single root of the ei function.
Returns: The ei functions return $Ei(x)$ as denoted in ISO:31 [4], Item No. 11-14.21.

5.2.24 Riemann zeta function [tr.math.sf.zeta]

```
double       zeta( double x );
float        zeta( float x );
long double  zeta( long double x );
```

Effects: These functions compute the Riemann zeta function (as defined by Abramowitz and Stegun [1], §23.2.1, *etc.*) of their respective arguments x. A range error (due to overflow) occurs if x is too close to one.
Returns: The zeta functions return $\zeta(x)$ as denoted in ISO:31 [4], Item No. 11-14.23.

Chapter 6

Containers [tr.cont]

6.1 Tuple types [tr.tuple]

This clause describes the tuple library that provides a tuple type as the class template `tuple` that can be instantiated with any number of arguments. An implementation can set an upper limit for the number of arguments. The minimum value for this implementation quantity is defined in Annex B. Each template argument specifies the type of an element in the `tuple`. Consequently, tuples are heterogeneous, fixed-size collections of values.

6.1.1 Header `<tuple>` synopsis [tr.tuple.synopsis]

```
template <class T1 = implementation-defined,
          class T2 = implementation-defined,
          ...,
          class TM = implementation-defined> class tuple;

template<class T1, class T2, ..., class TM,
         class U1, class U2, ..., class UM>
bool operator==(const tuple<T1, T2, ..., TM>&,
                const tuple<U1, U2, ..., UM>&);

template<class T1, class T2, ..., class TM,
         class U1, class U2, ..., class UM>
bool operator!=(const tuple<T1, T2, ..., TM>&,
                const tuple<U1, U2, ..., UM>&);

template<class T1, class T2, ..., class TM,
         class U1, class U2, ..., class UM>
bool operator<(const tuple<T1, T2, ..., TM>&,
               const tuple<U1, U2, ..., UM>&);

template<class T1, class T2, ..., class TM,
```

```
         class U1, class U2, ..., class UM>
bool operator<=(const tuple<T1, T2, ..., TM>&,
                const tuple<U1, U2, ..., UM>&);

template<class T1, class T2, ..., class TM,
         class U1, class U2, ..., class UM>
bool operator>(const tuple<T1, T2, ..., TM>&,
               const tuple<U1, U2, ..., UM>&);

template<class T1, class T2, ..., class TM,
         class U1, class U2, ..., class UM>
bool operator>=(const tuple<T1, T2, ..., TM>&,
                const tuple<U1, U2, ..., UM>&);

template <class T> class tuple_size;

template <int I, class T> class tuple_element;

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

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

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

template<class CharType, class CharTrait,
         class T1, class T2, ..., class TN>
basic_ostream<CharType, CharTrait>&
operator<<(basic_ostream<CharType, CharTrait>&,
           const tuple<T1, T2, ..., TN>&);

template<class CharType, class CharTrait,
         class T1, class T2, ..., class TN>
basic_istream<CharType, CharTrait>&
operator>>(basic_istream<CharType, CharTrait>&,
           tuple<T1, T2, ..., TN>&);

tuple_manip1 tuple_open(char_type c);
tuple_manip2 tuple_close(char_type c);
tuple_manip3 tuple_delimiter(char_type c);
```

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

`M` is used to denote the implementation-defined number of template type parameters to the tuple class template, and `N` is used to denote the number of template arguments specified in an instantiation. [*Example:* Given the instantiation `tuple<int, float, char>`, `N` is 3. *—end example*]

```
template <class T1 = implementation-defined,
          class T2 = implementation-defined,
          ...,
          class TM = implementation-defined>
class tuple
{
public:
  tuple();
  explicit tuple(P1, P2, ..., PN); // iff N > 0

  tuple(const tuple&);

  template <class U1, class U2, ..., class UN>
  tuple(const tuple<U1, U2, ..., UN>&);

  template <class U1, class U2>
  tuple(const pair<U1, U2>&);

  tuple& operator=(const tuple&);

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

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

6.1.2.1 Construction [tr.tuple.cnstr]

```
tuple();
```

Requires: Each tuple element type `Ti` can be default constructed.
Effects: Default initializes each element.

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

Where, if `Ti` is a reference type then `Pi` is `Ti`, otherwise `Pi` is `const Ti&`.
Requires: Each tuple element type `Ti` is copy constructible.
Effects: Copy initializes each element with the value of the corresponding parameter.

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

Requires: all types `Ti` shall be copy constructible.
Effects: Copy constructs each element of `*this` with the corresponding element of `u`.

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

Requires: Each type `Ti` shall be constructible from the corresponding type `Ui`.
Effects: Constructs each element of `*this` with the corresponding element of `u`.
[*Note:* In an implementation where one template definition serves for many different values for `N`, enable_if can be used to make the converting constructor and assignment operator exist only in the cases where the source and target have the same number of elements. Another way of achieving this is adding an extra integral template parameter which defaults to `N` (more precisely, a metafunction that computes `N`), and then defining the converting copy constructor and assignment only for tuples where the extra parameter in the source is `N`. —*end note*]

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

Requires: `T1` shall be constructible from `U1`, `T2` shall be constructible from `U2`. `N == 2`.
Effects: Constructs the first element with `u.first` and the second element with `u.second`.

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

Requires: All types `Ti` are assignable.
Effects: Assigns each element of `u` to the corresponding element of `*this`.
Returns: `*this`

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

Requires: Each type `Ti` shall be assignable from the corresponding type `Ui`.
Effects: Assigns each element of `u` to the corresponding element of `*this`.
Returns: `*this`

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

Requires: `T1` shall be assignable from `U1`, `T2` shall be assignable from `U2`. `N == 2`.
Effects: Assigns `u.first` to the first element of `*this` and `u.second` to the second element of `*this`.
Returns: `*this`
[*Note:* There seem to exist (rare) conditions where the converting copy constructor is a better match than the element-wise construction, even though the user might intend differently. An example of this is if one is constructing a one-element tuple where the element type is another tuple type `T` and if the parameter passed to the constructor is not of type `T`, but rather a tuple type that is convertible to `T`. The effect of the converting copy construction is most likely the same as the effect of the element-wise construction would have been. However, it 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*]

6.1.2.2 Tuple creation functions [tr.tuple.helper]

```
template<class T1, class T2, ..., class TN>
tuple<V1, V2, ..., VN>
make_tuple(const T1& t1, const T2& t2, ..., const TN& tn);
```

where `Vi` is `X&`, if the cv-unqualified type `Ti` is `reference_wrapper<X>`, otherwise `Vi` is `Ti`.
Returns: `tuple<V1, V2, ..., VN>(t1, t2, ..., tn)`.
Notes: The `make_tuple` function template must be implemented for each different number of arguments from 0 to the maximum number of allowed tuple elements.
[*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);
```

Returns: `tuple<T1&, T2&, ..., TN&>(t1, t2, ..., tn)`
Notes: The `tie` function template must be implemented for each different number of arguments from 0 to the maximum number of allowed tuple elements.
[*Example:*
`tie` functions allow one to create tuples that unpack tuples into variables. `ignore` can be used for elements that are not needed:

```
int i; std::string s;
tie(i, ignore, s) = make_tuple(42, 3.14, "C++");
// i == 42, s == "C++"
```

—*end example*]

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

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

Requires: `T` is an instantiation of class template tuple.
Type: integral constant expression.
Value: Number of elements in `T`.

```
tuple\_element<I, T>::type
```

Requires: $0 \leq$ `I` $<$ `tuple_size<T>::value`. The program is ill-formed if `I` is out of bounds.
Value: The type of the `I`th element of `T`, where indexing is zero-based.

6.1.2.4 Element access [tr.tuple.elem]

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

Requires: $0 \leq$ `I` $<$ `N`. The program is ill-formed if `I` is out of bounds.
Return type: `RI`. If `TI` is a reference type, then `RI` is `TI`, otherwise `RI` is `TI&`.
Returns: A reference to the `I`th element of `t`, where indexing is zero-based.

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

Requires: $0 \leq$ `I` $<$ `N`. The program is ill-formed if `I` is out of bounds.
Return type: `PI`. If `TI` is a reference type, then `PI` is `TI`, otherwise `PI` is `const TI&`.
Returns: A const reference to the `I`th element of `t`, where indexing is zero-based.
[*Note:* Constness is shallow. If `TI` is some reference type `X&`, the return type is `X&`, not `const X&`. However, if the element type is non-reference type `T`, the return type is `const T&`. This is consistent with how constness is defined to work for member variables of reference type. *—end note*.]
[*Note:* Implementing `get` as a member function of tuple, would require using the `template` keyword in invocations where the type of the tuple object is dependent on a template parameter. For example: `t.template get<1>();` *—end note*]

6.1.2.5 Equality and inequality comparisons [tr.tuple.eq]

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

Requires: `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`.
Return type: `bool`
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`.
Effects: The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first equality comparison that evaluates to `false`.

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

Requires: `tuple_size<tuple<T1, T2, ..., TM> >::value == tuple_size<tuple<U1, U2, ..., UM> >::value == N`. For all `i`, where `0 <= i < N`, `get<i>(t) != get<i>(u)` is a valid expression returning a type that is convertible to `bool`.
Return type: `bool`
Returns: `true` iff `get<i>(t) != get<i>(u)` for any `i`. For any two zero-length tuples `e` and `f`, `e != f` returns `false`.
Effects: The elementary comparisons are performed in order from the zeroth index upwards. No comparisons or element accesses are performed after the first inequality comparison that evaluates to `true`.

6.1.2.6 <, > comparisons [tr.tuple.lt]

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

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

Requires: `tuple_size<tuple<T1, T2, ..., TM> >::value == tuple_size<tuple<U1, U2, ..., UM> >::value == N`. For all `i`, where `0 <= i < N`, `get<i>(t)` ⊙ `get<i>(u)` is a valid expression returning a type that is convertible to `bool`, where ⊙ is either `<` or `>`.
Return type: `bool`
Returns: The result of a lexicographical comparison with ⊙ between `t` and `u`, defined equivalently to: `(bool)(get<0>(t)` ⊙ `get<0>(u)) || !((bool)(get<0>(u)` ⊙ `get<0>(t)) && t`$_{\mathrm{tail}}$ ⊙ `u`$_{\mathrm{tail}}$,
where `r`$_{\mathrm{tail}}$ 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`.

6.1.2.7 <= and >= comparisons [tr.tuple.le]

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

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

Requires: `tuple_size<tuple<T1, T2, ..., TM> >::value == tuple_size<tuple<U1, U2, ..., UM> >::value == N`. For all `i`, where `0 <= i < N`, `get<i>(t)` ⊙ `get<i>(u)` is a valid expression returning a type that is convertible to `bool`, where ⊙ is either `<=` or `>=`.
Returns: The result of a lexicographical comparison with ⊙ between `t` and `u`, defined equivalently to: `(bool)(get<0>(t)` ⊙ `get<0>(u)) && (!((bool)(get<0>(u)` ⊙ `get<0>(t)) || t`$_{\mathrm{tail}}$ ⊙ `u`$_{\mathrm{tail}}$`)`, where `r`$_{\mathrm{tail}}$ 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 `true`.
Notes: The above definitions for comparison operators do not impose the requirement that `t`$_{\mathrm{tail}}$ (or `u`$_{\mathrm{tail}}$) must be constructed. It may be even impossible, as `t` (or `u`) is not required to be copy constructible. Also, all comparison operators are short circuited to not perform element accesses beyond what is required to determine the result of the comparison.

6.1.2.8 Input and output [tr.tuple.io]

```
template<class CharType, class CharTrait,
         class T1, class T2, ..., class TN>
basic_ostream<CharType, CharTrait>&
operator<<(basic_ostream<CharType, CharTrait>& os,
           const tuple<T1, T2, ..., TN>& t);
```

Requires: For all i = 0, 1,..., N-1 in `os << get<i>(t)` is a valid expression.
Effects: Inserts `t` into `os` as L`t`$_0$$d$`t`$_1$$d$`...`$d$`t`$_n$$R$, where L is the opening, d the delimiter and R

the closing character set by tuple formatting manipulators. Each element $\texttt{t}_i$ is output by invoking `os << get<i>(t)`. A zero-element tuple is output as LR and a one-element tuple is output as $L\texttt{t}_0R$.
Returns: `os`

```
template<class CharType, class CharTrait,
         class T1, class T2, ..., class TN>
basic_istream<CharType, CharTrait>&
operator>>(basic_istream<CharType, CharTrait>& is,
           tuple<T1, T2, ..., TN>& t);
```

Requires: For all i = 0, 1, ..., N-1 in `is >> get<i>(t)` is a valid expression.
Effects: Extracts a tuple of the form $L\texttt{t}_0d\texttt{t}_1d\ldots d\texttt{t}_nR$, where L is the opening, d the delimiter and R the closing character set by tuple formatting manipulators. Each element $\texttt{t}_i$ is extracted by invoking `is >> get<i>(t)`. A zero-element tuple expects to extract LR from the stream and one-element tuple expects to extract $L\texttt{t}_0R$. If bad input is encountered, calls `is.set_state(ios::failbit)` (which may throw `ios::failure` (27.4.4.3)).
Returns: `is`
Notes: It is not guaranteed that a tuple written to a stream can be extracted back to a tuple of the same type.

6.1.2.9 Tuple formatting manipulators [tr.tuple.form]

The library defines the following three stream manipulator functions. The types designated *tuple_manip1*, *tuple_manip2* and *tuple_manip3* are implementation-specified.

tuple_manip1 `tuple_open(char_type c);`
tuple_manip2 `tuple_close(char_type c);`
tuple_manip3 `tuple_delimiter(char_type c);`

Returns: Each of these functions returns an object `s` of unspecified type such that if `out` is an instance of `basic_ostream<charT,traits>`, `in` is an instance of `basic_istream<charT,traits>` and `char_type` equals `charT`, then the expression `out << s` (respectively `in >> s`) sets `c` to be the opening, closing, or delimiter character (depending on the manipulator function called) to be used when writing tuples into `out` (respectively extracting tuples from `in`).
Notes: Implementations are not required to support these manipulators for streams with `sizeof(charT) > sizeof(long)`; `out << s` and `in >> s` are required to fail at compile time if `out` and `in` are such streams and the implementation does not support tuple formatting manipulators for them.
[*Note:* The constraint stated in the above **Notes** section allows an implementation where the delimiter characters are stored in space allocated by `xalloc`, which allocates an array of `long`s. A more general alternative is to store pointers to the delimiter characters in the `xalloc`-allocated array, and register a callback function (with `ios_base::register_callback`) for the stream to take care of deallocating the memory. If this approach is taken, the delimiters could be chosen to be strings instead of single characters. This might be worthwhile, such as to allow delimiters like `", "`. *—end note*]

6.1.3 Pairs [tr.tuple.pairs]

[Additions to pair to work with tuples]
This is an *impure* extension 1.2 to the standard library class template `std::pair`.

```
template<class T1, class T2>
struct tuple_size<pair<T1, T2> > {
  static const int value = 2;
};

template<class T1, class T2>
struct tuple_element<0, pair<T1, T2> > {
  typedef T1 type;
};

template<class T1, class T2>
struct tuple_element<1, pair<T1, T2> > {
  typedef T2 type;
};

template<int I, class T1, class T2>
P& get(pair<T1, T2>&);

template<int I, class T1, class T2>
const P& get(const pair<T1, T2>&);
```

Return type: If `I` is `0` then `P` is `T1`, if `I` is `1` then `P` is `T2`, otherwise the program is ill-formed.
Returns: If `I == 0` returns `p.first`, otherwise returns `p.second`.

6.1.4 Implementation quantities [tr.tuple.lim]

The maximum number of elements in one tuple type is implementation defined. This limits should be at least 10.

6.2 Unordered associative containers [tr.hash]

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

Unordered associative containers provide an ability for fast retrieval of data based on keys. The worst-case complexity for most operations is linear, but the average case is much faster. The library provides four basic kinds of hashed associative containers: `unordered_set`, `unordered_map`, `unordered_multiset`, and `unordered_multimap`.
Each hashed associative container is parameterized by `Key`, by a function object `Hash` that acts as a hash function for values of type `Key`, and on a binary predicate `Pred` that induces an equivalence relation on values of type `Key`. Additionally, `unordered_map` and `unordered_multimap` associate an arbitrary *mapped type* `T` with the `Key`.
A hash function is a function object that takes a single argument of type `Key` and returns a value of type `std::size_t` in the range `[0, std::numeric_limits<std::size_t>::max())`.
Two values `k1` and `k2` of type `Key` are considered equal if the container's equality function object returns `true` when passed those values. If `k1` and `k2` are equal, the hash function must return the same value for both.
A hashed associative container supports *unique keys* if it may contain at most one element for each key. Otherwise, it supports *equivalent keys*. `unordered_set` and `unordered_map` support unique

keys. `unordered_multiset` and `unordered_multimap` support equivalent keys. In containers that support equivalent keys, elements with equivalent keys are adjacent to each other.
For `unordered_set` and `unordered_multiset` the value type is the same as the key type. For `unordered_map` and `unordered_multimap` it is equal to `std::pair<const Key, T>`.
The elements of a hashed associative container are organized into *buckets*. Keys with the same hash code appear in the same bucket. The number of buckets is automatically increased as elements are added to a hashed associative container, so that the average number of elements per bucket is kept below a bound. Rehashing invalidates iterators, changes ordering between elements, and changes which buckets elements appear in, but does not invalidate pointers or references to elements.
In table 6.1:
`X` is a hashed associative container class, `a` is an object of type `X`, `b` is a possibly const object of type `X`, `a_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 6.1: Unordered associative container requirements (in addition to container)

expression	**Return Type**	**assertion/note/pre/post-condition**	**complexity**
`X::key_type`	`Key`	`Key` is `Assignable` and `CopyConstructible`	compile time
`X::hasher`	`Hash`	`Hash` is a unary function object that take an argument of type `Key` and returns a value of 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

continued from previous page			
`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
`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 `std::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 `std::distance(i, j)`), worst case $\mathcal{O}(N^2)$

continued from previous page			
`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 `std::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 `std::distance(i, j)`), worst case $\mathcal{O}(N^2)$
`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 `a` was constructed.	constant
`b.key_eq()`	`key_equal`	Returns the key equality function out of which `a` 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()`).

continued from previous page			
`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 `std::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()`).
`a.erase(r1, r2)`	`void`	Erases all elements in the range `[r1, t2)`.	Average case $\mathcal{O}$(`std::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 `a`.	Returns an iterator pointing to an element with key equivalent to `k`, or `a.end()` if no such element exists.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).
`b.count(k)`	`size_type`	Returns the number of elements with key equivalent to `k`.	Average case $\mathcal{O}(1)$, worst case $\mathcal{O}$(`a.size()`).
`b.equal_range(k)`	`pair<iterator, iterator>`; `pair<const_iterator, const_iterator>` for const `a`.	Returns a range containing all elements with keys equivalent to `k`. Returns `std::make_pair(a.end(), a.end())` if no such elements exist.	Average case $\mathcal{O}$(`a.count(k)`). Worst case $\mathcal{O}$(`a.size()`).
`b.bucket_count()`	`size_type`	Returns the number of buckets that `b` contains.¿	Constant

continued from previous page			
`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.	$\mathcal{O}$(`a.bucket_size(n)`)
`b.begin(n)`	`local_iterator`; `const_local_iterator` for const `a`.	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 `a`.	Pre: `n` is in the range `[0, b.bucket_count())`.	Constant
`b.load_factor()`	`float`	Returns the average number of elements per bucket.¿	Constant
`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`	Pre: `n > a.size() / a.max_load_factor()`. Changes the number of buckets so that it is at least `n`.	Average case linear in `a.size()`, worst case quadratic.

Unordered associative containers are not required to support the expressions `a == b` or `a != b`. [*Note:* This is because the container requirements define operator equality in terms of equality of ranges. Since the elements of an unordered associative container appear in an arbitrary order, range

equality is not a useful operation. —*end note*]
The iterator types `iterator` and `const_iterator` of an unordered associative container are of at least the forward iterator category. For unordered associative containers where the key type and value type are the same, both `iterator` and `const_iterator` are const iterators.
The insert members shall not affect the validity of references to container elements, but may invalidate all iterators to the container. The erase members shall invalidate only iterators and references to the erased elements.

6.2.1.1 Exception safety guarantees [tr.unord.req.except]

- For unordered associative containers, no `clear()` function throws an exception. No `erase()` function throws an exception unless that exception is thrown by the container's Hash or Pred object (if any).
- For unordered associative containers, if an exception is thrown by an insert() function while inserting a single element other than by the container's hash function, the insert() function has no effects.
- For unordered associative containers, no `swap` function throws an exception unless that exception is thrown by the copy constructor or copy assignment operator of the container's Hash or Pred object (if any).

6.2.2 Additions to header `<functional>` synopsis [tr.unord.fun.syn]

```
namespace tr1 {
  // Hash function base template
  template <class T> struct hash;

  // Hash function specializations

  template <> struct hash<bool>;
  template <> struct hash<char>;
  template <> struct hash<signed char>;
  template <> struct hash<unsigned char>;
  template <> struct hash<wchar_t>;
  template <> struct hash<short>;
  template <> struct hash<int>;
  template <> struct hash<long>;
  template <> struct hash<unsigned short>;
  template <> struct hash<unsigned int>;
  template <> struct hash<unsigned long>;

  template <> struct hash<float>;
  template <> struct hash<double>;
  template <> struct hash<long double>;

  template<class T>
```

```
  struct hash<T*>

  template <class charT, class traits, class Allocator>
  struct hash<std::basic_string<charT, traits, Allocator> >;
}
```

6.2.3 Class template `hash` [tr.unord.hash]

The function object `hash` is used as the default hash function by the *unordered associative containers*. This class template is only required to be instantiable for integer types (3.9.1), floating point types (3.9.1), pointer types (8.3.1), and (for any valid set of `charT`, `traits`, and `Alloc`) `std::basic_string<charT, traits, Alloc>`.

```
template <class T>
struct hash : public std::unary_function<T, std::size_t>
{
  std::size_t operator()(T val) const;
};
```

The return value of `operator()` is unspecified, except that equal arguments yield the same result. `operator()` shall not throw exceptions.

6.2.4 Unordered associative container classes [tr.unord.unord]

6.2.4.1 Header `<unordered_set>` synopsis [tr.unord.syn.set]

```
namespace tr1 {
  template <class Value,
            class Hash = hash<Value>,
            class Pred = std::equal_to<Value>,
            class Alloc = std::allocator<Value> >
  class unordered_set;

  template <class Value,
            class Hash = hash<Value>,
            class Pred = std::equal_to<Value>,
            class Alloc = std::allocator<Value> >
  class unordered_multiset;
}
```

6.2.4.2 Header `<unordered_map>` synopsis [tr.unord.syn.map]

```
namespace tr1 {
  template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = std::equal_to<Key>,
            class Alloc = std::allocator<std::pair<const Key, T> > >
  class unordered_map;
```

```
  template <class Key,
            class T,
            class Hash = hash<Key>,
            class Pred = std::equal_to<Key>,
            class Alloc = std::allocator<std::pair<const Key, T> > >
  class unordered_multiset;
}
```

6.2.4.3 Class template `unordered_set` [tr.unord.set]

An `unordered_set` is a kind of unordered associative container that supports unique keys (an `unordered_set` contains at most one of each key value) and in which the elements' keys are the elements themselves.

An `unordered_set` satisfies all of the requirements of a container and of a unordered associative container. It provides the operations described in the preceding requirements table for unique keys; that is, an `unordered_set` supports the `a_uniq` operations in that table, not the `a_eq` operations. For a `unordered_set<Value>` the `key type` and the value type are both `Value`. The `iterator` and `const_iterator` types are both const iterator types. It is unspecified whether or not they are the same type.

This section only describes operations on `unordered_set` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
template <class Value,
          class Hash  = hash<Value>,
          class Pred  = std::equal_to<Value>,
          class Alloc = std::allocator<Value> >
class unordered_set
{
public:
  // types
  typedef Value                                    key_type;
  typedef Value                                    value_type;
  typedef Hash                                     hasher;
  typedef Pred                                     key_equal;
  typedef Alloc                                    allocator_type;
  typedef typename allocator_type::pointer         pointer;
  typedef typename allocator_type::const_pointer   const_pointer;
  typedef typename allocator_type::reference       reference;
  typedef typename allocator_type::const_reference const_reference;
  typedef implementation-defined                   size_type;
  typedef implementation-defined                   difference_type;

  typedef implementation-defined                   iterator;
  typedef implementation-defined                   const_iterator;
  typedef implementation-defined                   local_iterator;
  typedef implementation-defined                   const_local_iterator;
```

```
// construct/destroy/copy
explicit unordered_set(size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
template <class InputIterator>
  unordered_set(InputIterator f, InputIterator l,
                size_type n = implementation-defined,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
unordered_set(const unordered_set&);
~unordered_set();
unordered_set& operator=(const unordered_set&);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;

// modifiers
std::pair<iterator, bool> insert(const value_type& obj);
iterator insert(const_iterator hint, const value_type& obj);
template <class InputIterator>
  void insert(InputIterator first, InputIterator last);

void erase(const_iterator position);
size_type erase(const key_type& k);
void erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_set&);

// observers
hasher hash_function() const;
key_equal key_eq() const;
```

```
  // lookup
  iterator       find(const key_type& k);
  const_iterator find(const key_type& k) const;
  size_type count(const key_type& k) const;
  std::pair<iterator, iterator>
    equal_range(const key_type& k);
  std::pair<const_iterator, const_iterator>
    equal_range(const key_type& k) const;

  // bucket interface
  size_type bucket_count() const;
  size_type max_bucket_count() const;
  size_type bucket_size(size_type n);
  size_type bucket(const key_type& k);
  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(const unordered_set<Value, Hash, Pred, Alloc>& x,
            const unordered_set<Value, Hash, Pred, Alloc>& y);
```

6.2.4.3.1 `unordered_set` constructors [tr.unord.set.cnstr]

```
explicit unordered_set(size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
```

Effects: Constructs an empty `unordered_set` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation defined. `max_load_factor()` is 1.0.
Complexity: Constant.

```
template <class InputIterator>
  unordered_set(InputIterator f, InputIterator l,
           size_type n = implementation-defined,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
```

```
                const allocator_type& a = allocator_type());
```

Effects: Constructs an empty `unordered_set` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. (If *n* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range `[`*first*`, `*last*`)`. `max_load_factor()` is 1.0.

Complexity: Average case linear, worst case quadratic.

6.2.4.3.2 `unordered_set` swap [tr.unord.set.swap]

```
template <class Value, class Hash, class Pred, class Alloc>
  void swap(const unordered_set<Value, Hash, Pred, Alloc>& x,
            const unordered_set<Value, Hash, Pred, Alloc>& y);
```

Effects: `x.swap(y)`.

6.2.4.4 Class template `unordered_map` [tr.unord.map]

An `unordered_map` is a kind of unordered associative container that supports unique keys (an `unordered_map` contains at most one of each key value) and that associates values of another type `mapped_type` with the keys.

An `unordered_map` satisfies all of the requirements of a container and of a unordered associative container. It provides the operations described in the preceding requirements table for unique keys; that is, an `unordered_map` supports the `a_uniq` operations in that table, not the `a_eq` operations. For a `unordered_map<Key, T>` the `key type` is `Key`, the mapped type is `T`, and the value type is `std::pair<const Key, T>`.

This section only describes operations on `unordered_map` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
template <class Key,
          class T,
          class Hash  = hash<Key>,
          class Pred  = std::equal_to<Key>,
          class Alloc = std::allocator<std::pair<const Key, T> > >
class unordered_map
{
public:
  // types
  typedef Key                                        key_type;
  typedef std::pair<const Key, T>                    value_type;
  typedef T                                          mapped_type;
  typedef Hash                                       hasher;
  typedef Pred                                       key_equal;
  typedef Alloc                                      allocator_type;
  typedef typename allocator_type::pointer           pointer;
  typedef typename allocator_type::const_pointer     const_pointer;
  typedef typename allocator_type::reference         reference;
  typedef typename allocator_type::const_reference   const_reference;
  typedef implementation-defined                     size_type;
  typedef implementation-defined                     difference_type;
```

```
typedef implementation-defined                  iterator;
typedef implementation-defined                  const_iterator;
typedef implementation-defined                  local_iterator;
typedef implementation-defined                  const_local_iterator;

// construct/destroy/copy
explicit unordered_map(size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
template <class InputIterator>
  unordered_map(InputIterator f, InputIterator l,
                size_type n = implementation-defined,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
unordered_map(const unordered_map&);
~unordered_map();
unordered_map& operator=(const unordered_map&);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;

// modifiers
std::pair<iterator, bool> insert(const value_type& obj);
iterator insert(const_iterator hint, const value_type& obj);
template <class InputIterator>
  void insert(InputIterator first, InputIterator last);

void erase(const_iterator position);
size_type erase(const key_type& k);
void erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_map&);
```

```
  // observers
  hasher hash_function() const;
  key_equal key_eq() const;

  // lookup
  iterator       find(const key_type& k);
  const_iterator find(const key_type& k) const;
  size_type count(const key_type& k) const;
  std::pair<iterator, iterator>
    equal_range(const key_type& k);
  std::pair<const_iterator, const_iterator>
    equal_range(const key_type& k) const;

  mapped_type& operator[](const key_type& k);

  // bucket interface
  size_type bucket_count() const;
  size_type max_bucket_count() const;
  size_type bucket_size(size_type n);
  size_type bucket(const key_type& k);
  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 Key, class T, class Hash, class Pred, class Alloc>
  void swap(const unordered_map<Key, T, Hash, Pred, Alloc>& x,
            const unordered_map<Key, T, Hash, Pred, Alloc>& y);
```

6.2.4.4.1 `unordered_map` constructors [tr.unord.map.cnstr]

```
explicit unordered_map(size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
```

Effects: Constructs an empty `unordered_map` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation defined. `max_load_factor()` is 1.0.
Complexity: Constant.

```
template <class InputIterator>
  unordered_map(InputIterator f, InputIterator l,
                size_type n = implementation-defined,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
```

Effects: Constructs an empty `unordered_map` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. (If *n* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range [*first*, *last*). `max_load_factor()` is 1.0.
Complexity: Average case linear, worst case quadratic.

6.2.4.4.2 `unordered_map` element access [tr.unord.map.elem]

```
mapped_type& operator[](const key_type& k);
```

Effects: If the `unordered_map` does not already contain an element whose key is equivalent to *k*, inserts `std::pair<const key_type, mapped_type>(k, mapped_type())`.
Returns: A reference to `x.second`, where `x` is the (unique) element whose key is equivalent to *k*.

6.2.4.4.3 `unordered_map` swap [tr.unord.map.swap]

```
template <class Value, class Hash, class Pred, class Alloc>
  void swap(const unordered_map<Value, Hash, Pred, Alloc>& x,
            const unordered_map<Value, Hash, Pred, Alloc>& y);
```

Effects: `x.swap(y)`.

6.2.4.5 Class template `unordered_multiset` [tr.unord.multiset]

An `unordered_multiset` is a kind of unordered associative container that supports equivalent keys (an `unordered_multiset` may contain multiple copies of the same key value) and in which the elements' keys are the elements themselves.
An `unordered_multiset` satisfies all of the requirements of a container and of a unordered associative container. It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multiset` supports the `a_eq` operations in that table, not the `a_uniq` operations. For a `unordered_multiset<Value>` the `key type` and the value type are both `Value`. The `iterator` and `const_iterator` types are both const iterator types. It is unspecified whether or not they are the same type.
This section only describes operations on `unordered_multiset` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
template <class Value,
          class Hash  = hash<Value>,
          class Pred  = std::equal_to<Value>,
          class Alloc = std::allocator<Value> >
class unordered_multiset
{
public:
  // types
```

```
typedef Value                                       key_type;
typedef Value                                       value_type;
typedef Hash                                        hasher;
typedef Pred                                        key_equal;
typedef Alloc                                       allocator_type;
typedef typename allocator_type::pointer         pointer;
typedef typename allocator_type::const_pointer   const_pointer;
typedef typename allocator_type::reference       reference;
typedef typename allocator_type::const_reference const_reference;
typedef implementation-defined                      size_type;
typedef implementation-defined                      difference_type;

typedef implementation-defined                      iterator;
typedef implementation-defined                      const_iterator;
typedef implementation-defined                      local_iterator;
typedef implementation-defined                      const_local_iterator;


// construct/destroy/copy
explicit unordered_multiset(size_type n = implementation-defined,
                            const hasher& hf = hasher(),
                            const key_equal& eql = key_equal(),
                            const allocator_type& a = allocator_type());
template <class InputIterator>
  unordered_multiset(InputIterator f, InputIterator l,
                size_type n = implementation-defined,
                const hasher& hf = hasher(),
                const key_equal& eql = key_equal(),
                const allocator_type& a = allocator_type());
unordered_multiset(const unordered_multiset&);
~unordered_multiset();
unordered_multiset& operator=(const unordered_multiset&);
allocator_type get_allocator() const;

// size and capacity
bool empty() const;
size_type size() const;
size_type max_size() const;

// iterators
iterator       begin();
const_iterator begin() const;
iterator       end();
const_iterator end() const;

// modifiers
```

```
  iterator insert(const value_type& obj);
  iterator insert(const_iterator hint, const value_type& obj);
  template <class InputIterator>
    void insert(InputIterator first, InputIterator last);

  void erase(const_iterator position);
  size_type erase(const key_type& k);
  void erase(const_iterator first, const_iterator last);
  void clear();

  void swap(unordered_multiset&);

  // observers
  hasher hash_function() const;
  key_equal key_eq() const;

  // lookup
  iterator       find(const key_type& k);
  const_iterator find(const key_type& k) const;
  size_type count(const key_type& k) const;
  std::pair<iterator, iterator>
    equal_range(const key_type& k);
  std::pair<const_iterator, const_iterator>
    equal_range(const key_type& k) const;

  // bucket interface
  size_type bucket_count() const;
  size_type max_bucket_count() const;
  size_type bucket_size(size_type n);
  size_type bucket(const key_type& k);
  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(const unordered_multiset<Value, Hash, Pred, Alloc>& x,
            const unordered_multiset<Value, Hash, Pred, Alloc>& y);
  }
```

6.2.4.5.1 `unordered_multiset` constructors [tr.unord.multiset.cnstr]

```
explicit unordered_multiset(size_type n = implementation-defined,
                            const hasher& hf = hasher(),
                            const key_equal& eql = key_equal(),
                            const allocator_type& a = allocator_type());
```

Effects: Constructs an empty `unordered_multiset` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation defined. `max_load_factor()` is 1.0.
Complexity: Constant.

```
template <class InputIterator>
  unordered_multiset(InputIterator f, InputIterator l,
                     size_type n = implementation-defined,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
```

Effects: Constructs an empty `unordered_multiset` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. (If *n* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range [*first*, *last*). `max_load_factor()` is 1.0.
Complexity: Average case linear, worst case quadratic.

6.2.4.5.2 `unordered_multiset` swap [tr.unord.multiset.swap]

```
template <class Value, class Hash, class Pred, class Alloc>
  void swap(const unordered_multiset<Value, Hash, Pred, Alloc>& x,
            const unordered_multiset<Value, Hash, Pred, Alloc>& y);
```

Effects: `x.swap(y);`

6.2.4.6 Class template `unordered_multimap` [tr.unord.multimap]

An `unordered_multimap` is a kind of unordered associative container that supports equivalent keys (an `unordered_multimap` may contain multiple copies of each key value) and that associates values of another type `mapped_type` with the keys.
An `unordered_multimap` satisfies all of the requirements of a container and of a unordered associative container. It provides the operations described in the preceding requirements table for equivalent keys; that is, an `unordered_multimap` supports the `a_eq` operations in that table, not the `a_uniq` operations. For an `unordered_multimap<Key, T>` the `key type` is `Key`, the mapped type is `T`, and the value type is `std::pair<const Key, T>`.
This section only describes operations on `unordered_multimap` that are not described in one of the requirement tables, or for which there is additional semantic information.

```
template <class Key,
          class T,
          class Hash  = hash<Key>,
          class Pred  = std::equal_to<Key>,
          class Alloc = std::allocator<std::pair<const Key, T> > >
```

```
class unordered_multimap
{
public:
  // types
  typedef Key                                          key_type;
  typedef std::pair<const Key, T>                      value_type;
  typedef T                                            mapped_type;
  typedef Hash                                         hasher;
  typedef Pred                                         key_equal;
  typedef Alloc                                        allocator_type;
  typedef typename allocator_type::pointer             pointer;
  typedef typename allocator_type::const_pointer       const_pointer;
  typedef typename allocator_type::reference           reference;
  typedef typename allocator_type::const_reference     const_reference;
  typedef implementation-defined                       size_type;
  typedef implementation-defined                       difference_type;

  typedef implementation-defined                       iterator;
  typedef implementation-defined                       const_iterator;
  typedef implementation-defined                       local_iterator;
  typedef implementation-defined                       const_local_iterator;

  // construct/destroy/copy
  explicit unordered_multimap(size_type n = implementation-defined,
                              const hasher& hf = hasher(),
                              const key_equal& eql = key_equal(),
                              const allocator_type& a = allocator_type());
  template <class InputIterator>
    unordered_multimap(InputIterator f, InputIterator l,
                       size_type n = implementation-defined,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
  unordered_multimap(const unordered_multimap&);
  ~unordered_multimap();
  unordered_multimap& operator=(const unordered_multimap&);
  allocator_type get_allocator() const;

  // size and capacity
  bool empty() const;
  size_type size() const;
  size_type max_size() const;

  // iterators
  iterator       begin();
  const_iterator begin() const;
```

```
iterator       end();
const_iterator end() const;

// modifiers
iterator insert(const value_type& obj);
iterator insert(const_iterator hint, const value_type& obj);
template <class InputIterator>
  void insert(InputIterator first, InputIterator last);

void erase(const_iterator position);
size_type erase(const key_type& k);
void erase(const_iterator first, const_iterator last);
void clear();

void swap(unordered_multimap&);

// observers
hasher hash_function() const;
key_equal key_eq() const;

// lookup
iterator       find(const key_type& k);
const_iterator find(const key_type& k) const;
size_type count(const key_type& k) const;
std::pair<iterator, iterator>
  equal_range(const key_type& k);
std::pair<const_iterator, const_iterator>
  equal_range(const key_type& k) const;

mapped_type& operator[](const key_type& k);

// bucket interface
size_type bucket_count() const;
size_type max_bucket_count() const;
size_type bucket_size(size_type n);
size_type bucket(const key_type& k);
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 Key, class T, class Hash, class Pred, class Alloc>
  void swap(const unordered_multimap<Key, T, Hash, Pred, Alloc>& x,
            const unordered_multimap<Key, T, Hash, Pred, Alloc>& y);
```

6.2.4.6.1 `unordered_multimap` constructors [tr.unord.multimap.cnstr]

```
explicit unordered_multimap(size_type n = implementation-defined,
                            const hasher& hf = hasher(),
                            const key_equal& eql = key_equal(),
                            const allocator_type& a = allocator_type());
```

Effects: Constructs an empty `unordered_multimap` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. If *n* is not provided, the number of buckets is implementation defined. `max_load_factor()` is 1.0.
Complexity: Constant.

```
template <class InputIterator>
  unordered_multimap(InputIterator f, InputIterator l,
                     size_type n = implementation-defined,
                     const hasher& hf = hasher(),
                     const key_equal& eql = key_equal(),
                     const allocator_type& a = allocator_type());
```

Effects: Constructs an empty `unordered_multimap` using the specified hash function, key equality function, and allocator, and using at least *n* buckets. (If *n* is not provided, the number of buckets is implementation defined.) Then inserts elements from the range [*first*, *last*). `max_load_factor()` is 1.0.
Complexity: Average case linear, worst case quadratic.

6.2.4.6.2 `unordered_multimap` swap [tr.unord.multimap.swap]

```
template <class Value, class Hash, class Pred, class Alloc>
  void swap(const unordered_multimap<Value, Hash, Pred, Alloc>& x,
            const unordered_multimap<Value, Hash, Pred, Alloc>& y);
```

Effects: `x.swap(y)`.

Chapter 7

Regular expressions [tr.re]

7.1 Definitions [tr.re.def]

The following definitions shall apply to this clause:
Collating element: A sequence of one or more characters within the current locale that collate as if they were a single character.
Finite state machine: An unspecified data structure that is used to represent a regular expression, and which permits efficient matches against the regular expression to be obtained.
Format specifier: A sequence of one or more characters that is to be replaced with some part of a regular expression match.
Matched: A sequence of zero or more characters shall be said to be matched by a regular expression when the characters in the sequence correspond to a sequence of characters defined by the pattern.
Partial match: A match that is obtained by matching one or more characters at the end of a character-container sequence, but only a prefix of the regular expression.
Primary equivalence class: A set of one or more characters which share the same primary sort key: that is the sort key weighting that depends only upon character shape, and not accentation, case, or locale specific tailorings.
Regular expression: A pattern that selects specific strings from a set of character strings.
Sub-expression: A subset of a regular expression that has been marked by parenthesis.

7.2 Requirements [tr.re.req]

This subclause defines requirements on classes representing regular expression traits. [*Note:* The class template `regex_traits<charT>`, defined in clause 7.7, satisfies these requirements. *—end note*]
The class template `basic_regex`, defined in clause 7.8, needs a set of related types and functions to complete the definition of it's 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.
To specialize the `basic_regex` template to generate a regular expression class to handle a particular character container type `CharT`, that and its related regular expression traits class `Traits` is passed as a pair of parameters to the `basic_regex` template as formal parameters `charT` and `Traits`.

In Table 7.1 X denotes a traits class defining types and functions for the character container type charT; u is an object of type X; v is an object of type const X; p is a value of type const charT*; I1 and I2 are Input Iterators; c is a value of type const charT; s is an object of type X::string_type; cs is an object of type const X::string_type; b is a value of type bool; I is a value of type int; and loc is an object of type X::locale_type.

Table 7.1: regular expression traits class requirements

Expression	Return Type	Assertion / Note / Pre / Post condition
X::char_type	charT	The character container type used in the implementation of class template basic_regex.
X::size_type		An unsigned integer type, capable of holding the length of a null-terminated string of charTs.
X::string_type	std::basic_string<charT>	
X::locale_type	Implementation defined	A copy constructible type that represents the locale used by the traits class.
X::char_class_type	Implementation defined	A bitmask type representing a particular character classification. Multiple values of this type can be bitwise-or'ed together to obtain a new valid value.
X::version_tag	Either regex_traits_version_1_tag or a class that publicly inherits from regex_traits_version_1_tag.	A type that signifies the interface to which this traits class conforms.
X::sentry	Implementation defined	A type used to force initialization of an instance of this traits class. An instance of this type must be copy constructed from a traits class instance, and verified that it does not compare equal to a null pointer constant, before any member function of that traits class instance may be called other than length, imbue, and getloc.
X::sentry s(u);	void	An object of type X::sentry shall be copy constructible from an instance of X.
X::sentry(u)	Convertible to bool.	Conversion of the sentry class to bool, is used to verify that the traits class has been correctly initialized.
X::length(p)	X::size_type	Yields the smallest i such that p[i] == 0. Complexity is linear in i .

continued from previous page		
`v.syntax_type(c)`	`regex_-constants::syntax_-type`	Returns a symbolic value of type `regex_constants::syntax_-type` that signifies the meaning of character `c` within the regular expression grammar.
`v.escape_syntax_-type(c)`	`regex_-constants::escape_-syntax_type`	Returns a symbolic value of type `regex_constants::escape_-syntax_type`, that signifies the meaning of character `c` within the regular expression grammar, when chas been preceded by an escape character. Precondition: if `b` is the character preceding `c` in the expression being parsed then: `v.syntax_type(b) == syntax_escape`.
`v.translate(c, b)`	`X::char_type`	Returns a character `d` such that: for any character `d` that is to be considered equivalent to `c` then `v.translate(c, false) == v.translate(d, false)`. Likewise 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(c, true) == v.translate(C, true)`.
`v.transform(cs)`	`X::string_type`	Returns a sort key for string `cs` such that: if string `cs` sorts before string `ct` then: `v.transform(cs) < v.transform(ct)`.
`v.transform_-primary(cs)`	`X::string_type`	Returns a primary sort key for string `cs`, such that if `cs` sorts before string `ct`, when character case is not considered, then `v.transform(cs) < v.transform(ct)`. Returns an empty string on error.

continued from previous page		
`v.lookup_-classname(cs)`	`X::char_class_type`	Converts the string `cs` into a bitmask type that can subsequently be passed to `is_class`. Values returned from `lookup_classname` can be safely bitwise or'ed together. Returns an empty string if `cs` is not the name of a character class recognized by `X`. At least the names `"d"`, `"w"`, `"s"`, `"alnum"`, `"alpha"`, `"blank"`, `"cntrl"`, `"digit"`, `"graph"`, `"lower"`, `"print"`, `"punct"`, `"space"`, `"upper"` and `"xdigit"`, shall be recognized. The value returned shall be independent of the case of the characters in `cs`.
`v.lookup_-collatename(cs)`	`X::string_type`	Returns a string containing the collating element named by `cs`, or an empty string if `cs` is not a recognized name.
`v.is_class(c, v.lookup_classname (cs))`	`bool`	Returns `true` if character `c` is a member of the character class `cs`, `false` otherwise.
`v.toi(I1, I2, i)`	`int`	Behaves as follows: if `p==q` or `v.is_class(*p, v.lookup_-classname("d")) == false` then returns -1. Otherwise performs formatted numeric input on the sequence `[p,q)` and returns the result as an `int`. Postcondition: either `p == q` or `v.is_class(*p, v.lookup_-classname("d")) == false`.
`u.imbue(loc)`	`X::locale_type`	Imbues `u` with the locale `loc`, returns the previous locale used by `u` if any.
`v.getloc()`	`X::locale_type`	Returns the current locale used by `v`, if any.
`v.error_string(i)`	`std::string`	Returns a human readable error string for the error condition `i`, where `i` is one of the values enumerated by type `regex_constants::error_-type`. If the value `i` is not recognized then returns the string `"Unknown error"` or a localized equivalent.

The header `<regex>` defines the class template `regex_traits` which shall be capable of being 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.sym]

The header `<regex>` defines a basic regular expression class template and its traits that can handle all char-like (lib.strings) template arguments.
The header `<regex>` defines a container-like class template that holds the result of a regular expression match.
The header `<regex>` defines a series of algorithms that allow an iterator sequence to be operated upon by a regular expression.
The header `<regex>` defines two specific template classes, regex and wregex and their special traits.
The header `<regex>` also defines two iterator types for enumerating regular expression matches.

7.4 Header `<regex>` synopsis [tr.re.syn]

```
namespace tr1 {

namespace regex_constants {
  typedef bitmask_type syntax_option_type;
  typedef bitmask_type match_flag_type;
  typedef implementation-defined syntax_type;
  typedef implementation-defined escape_syntax_type;
  typedef implementation-defined error_type;
} // namespace regex_constants

class bad_expression;

template <class charT>
struct regex_traits;

template <class charT,
          class traits = regex_traits<charT>,
          class Allocator = allocator<charT> >
class basic_regex;

template <class charT, class traits, class Allocator>
bool operator == (const basic_regex<charT, traits, Allocator>& lhs,
                  const basic_regex<charT, traits, Allocator>& rhs);
template <class charT, class traits, class Allocator>
bool operator != (const basic_regex<charT, traits, Allocator>& lhs,
                  const basic_regex<charT, traits, Allocator>& rhs);
template <class charT, class traits, class Allocator>
bool operator < (const basic_regex<charT, traits, Allocator>& lhs,
                 const basic_regex<charT, traits, Allocator>& rhs);
template <class charT, class traits, class Allocator>
bool operator <= (const basic_regex<charT, traits, Allocator>& lhs,
```

```
                  const basic_regex<charT, traits, Allocator>& rhs);
template <class charT, class traits, class Allocator>
bool operator >= (const basic_regex<charT, traits, Allocator>& lhs,
                  const basic_regex<charT, traits, Allocator>& rhs);
template <class charT, class traits, class Allocator>
bool operator > (const basic_regex<charT, traits, Allocator>& lhs,
                 const basic_regex<charT, traits, Allocator>& rhs);

template <class charT, class io_traits, class re_traits, class Allocator>
basic_ostream<charT, io_traits>&
   operator << (basic_ostream<charT, io_traits>& os,
                const basic_regex<charT, re_traits, Allocator>& e);

template <class charT, class traits, class Allocator>
void swap(basic_regex<charT, traits, Allocator>& e1,
          basic_regex<charT, traits, Allocator>& e2);

typedef basic_regex<char> regex;
typedef basic_regex<wchar_t> wregex;

template <class BidirectionalIterator>
class sub_match;

template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);


template <class BidirectionalIterator, class traits, class Allocator>
bool operator == (const std::basic_string<
                         iterator_traits<BidirectionalIterator> ::value_type,
```

```
                        traits,
                        Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator != (const std::basic_string<
                        iterator_traits<BidirectionalIterator> ::value_type,
                        traits,
                        Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator < (const std::basic_string<
                        iterator_traits<BidirectionalIterator> ::value_type,
                        traits,
                        Allocator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator > (const std::basic_string<
                        iterator_traits<BidirectionalIterator> ::value_type,
                        traits,
                        Allocator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator >= (const std::basic_string<
                        iterator_traits<BidirectionalIterator> ::value_type,
                        traits,
                        Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator <= (const std::basic_string<
                        iterator_traits<BidirectionalIterator> ::value_type,
                        traits,
                        Allocator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);

template <class BidirectionalIterator, class traits, class Allocator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<
                        iterator_traits<BidirectionalIterator> ::value_type,
                        traits,
                        Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<
                        iterator_traits<BidirectionalIterator> ::value_type,
                        traits,
                        Allocator>& rhs);
```

```
template <class BidirectionalIterator, class traits, class Allocator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 const std::basic_string<
                         iterator_traits<BidirectionalIterator> ::value_type,
                         traits,
                         Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 const std::basic_string<
                         iterator_traits<BidirectionalIterator> ::value_type,
                         traits,
                         Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<
                         iterator_traits<BidirectionalIterator> ::value_type,
                         traits,
                         Allocator>& rhs);
template <class BidirectionalIterator, class traits, class Allocator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  const std::basic_string<
                         iterator_traits<BidirectionalIterator> ::value_type,
                         traits,
                         Allocator>& rhs);

template <class BidirectionalIterator>
bool operator == (typename iterator_traits<BidirectionalIterator>
                             ::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator != (typename iterator_traits<BidirectionalIterator>
                             ::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator < (typename iterator_traits<BidirectionalIterator>
                             ::value_type const* lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator > (typename iterator_traits<BidirectionalIterator>
                             ::value_type const* lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator >= (typename iterator_traits<BidirectionalIterator>
                             ::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
```

```
bool operator <= (typename iterator_traits<BidirectionalIterator>
                                ::value_type const* lhs,
                  const sub_match<BidirectionalIterator>& rhs);

template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>
                                ::value_type const* rhs);
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>
                                ::value_type const* rhs);
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                                ::value_type const* rhs);
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                                ::value_type const* rhs);
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>
                                ::value_type const* rhs);
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>
                                ::value_type const* rhs);

template <class BidirectionalIterator>
bool operator == (typename iterator_traits<BidirectionalIterator>
                                ::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator != (typename iterator_traits<BidirectionalIterator>
                                ::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator < (typename iterator_traits<BidirectionalIterator>
                                ::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator > (typename iterator_traits<BidirectionalIterator>
                                ::value_type const& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
```

```
bool operator >= (typename iterator_traits<BidirectionalIterator>
                                ::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
template <class BidirectionalIterator>
bool operator <= (typename iterator_traits<BidirectionalIterator>
                                ::value_type const& lhs,
                  const sub_match<BidirectionalIterator>& rhs);

template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>
                                ::value_type const& rhs);
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>
                                ::value_type const& rhs);
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                                ::value_type const& rhs);
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>
                                ::value_type const& rhs);
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>
                                ::value_type const& rhs);
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>
                                ::value_type const& rhs);

template <class charT, class traits, class BidirectionalIterator>
basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& os,
                const sub_match<BidirectionalIterator>& m);

template <class BidirectionalIterator,
          class Allocator = allocator<
               typename iterator_traits<BidirectionalIterator>::value_type > >
class match_results;

template <class BidirectionalIterator, class Allocator>
bool operator == (const match_results<BidirectionalIterator, Allocator>& m1,
                  const match_results<BidirectionalIterator, Allocator>& m2);
```

```
template <class BidirectionalIterator, class Allocator>
bool operator != (const match_results<BidirectionalIterator, Allocator>& m1,
                  const match_results<BidirectionalIterator, Allocator>& m2);

template <class charT, class traits,
          class BidirectionalIterator, class Allocator>
basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& os,
                const match_results<BidirectionalIterator, Allocator>& m);

template <class BidirectionalIterator, class Allocator>
void swap(match_results<BidirectionalIterator, Allocator>& m1,
          match_results<BidirectionalIterator, Allocator>& m2);

typedef match_results<const char*> cmatch;
typedef match_results<const wchar_t*> wcmatch;
typedef match_results<string::const_iterator> smatch;
typedef match_results<wstring::const_iterator> wsmatch;

template <class BidirectionalIterator, class Allocator, class charT,
          class traits, class Allocator2>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 match_results<BidirectionalIterator, Allocator>& m,
                 const reg_expression<charT, traits, Allocator2>& e,
                 match_flag_type flags = match_default);
template <class BidirectionalIterator,
          class charT, class traits,
          class Allocator2>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 const reg_expression<charT, traits, Allocator2>& e,
                 match_flag_type flags = match_default);
template <class charT, class Allocator, class traits, class Allocator2>
bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
                 const reg_expression<charT, traits, Allocator2>& e,
                 match_flag_type flags = match_default);
template <class ST, class SA, class Allocator, class charT,
          class traits, class Allocator2>
bool regex_match(const basic_string<charT, ST, SA>& s,
                 match_results<typename basic_string<charT, ST, SA>
                                    ::const_iterator,
                               Allocator>& m,
                 const reg_expression<charT, traits, Allocator2>& e,
                 match_flag_type flags = match_default);
template <class charT, class traits, class Allocator2>
bool regex_match(const charT* str,
                 const reg_expression<charT, traits, Allocator2>& e,
```

```
                match_flag_type flags = match_default);
template <class ST, class SA, class charT, class traits, class Allocator2>
bool regex_match(const basic_string<charT, ST, SA>& s,
                const reg_expression<charT, traits, Allocator2>& e,
                match_flag_type flags = match_default);

template <class BidirectionalIterator, class Allocator, class charT,
          class traits, class Allocator2>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                 match_results<BidirectionalIterator, Allocator>& m,
                 const reg_expression<charT, traits, Allocator2>& e,
                 match_flag_type flags = match_default);
template <class charT, class Allocator, class traits, class Allocator2>
bool regex_search(const charT* str, match_results<const charT*, Allocator>& m,
                 const reg_expression<charT, traits, Allocator2>& e,
                 match_flag_type flags = match_default);
template <class BidirectionalIterator, class Allocator, class charT,
          class traits>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                 const reg_expression<charT, traits, Allocator>& e,
                 match_flag_type flags = match_default);
template <class charT, class Allocator, class traits>
bool regex_search(const charT* str
                 const reg_expression<charT, traits, Allocator>& e,
                 match_flag_type flags = match_default);
template <class ST, class SA, class Allocator, class charT,
          class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                 const reg_expression<charT, traits, Allocator>& e,
                 match_flag_type flags = match_default);
template <class ST, class SA, class Allocator, class charT,
          class traits, class Allocator2>
bool regex_search(const basic_string<charT, ST, SA>& s,
                 match_results<typename basic_string<charT, ST, SA>
                                    ::const_iterator,
                               Allocator>& m,
                 const reg_expression<charT, traits, Allocator2>& e,
                 match_flag_type flags = match_default);

template <class OutputIterator, class BidirectionalIterator, class traits,
          class Allocator, class charT>
OutputIterator regex_replace(OutputIterator out,
                             BidirectionalIterator first,
                             BidirectionalIterator last,
                             const reg_expression<charT, traits, Allocator>& e,
                             const basic_string<charT>& fmt,
```

```
                              match_flag_type flags = match_default);
template <class traits, class Allocator, class charT>
basic_string<charT> regex_replace(const basic_string<charT>& s,
                              const reg_expression<charT, traits, Allocator>& e,
                              const basic_string<charT>& fmt,
                              match_flag_type flags = match_default);

// regular expression iterators:
template <class BidirectionalIterator,
          class charT = iterator_traits<BidirectionalIterator>::value_type,
          class traits = regex_traits<charT>,
          class Allocator = allocator<charT> >
class regex_iterator;
template <class BidirectionalIterator,
          class charT = iterator_traits<BidirectionalIterator>::value_type,
          class traits = regex_traits<charT>,
          class Allocator = allocator<charT> >
class regex_token_iterator;

} // namespace tr1
```

7.5 Namespace `std::regex_constants` [tr.re.const]

The namespace `std::regex_constants` acts as a repository for the symbolic constants used by the regular expression library.
The namespace `std::regex_constants` defines five types: `syntax_option_type`, `match_flag_type`, `syntax_type`, `escape_syntax_type` and `error_type`, along with a series of constants of these types.

7.5.1 Bitmask Type `syntax_option_type` [tr.re.synopt]

```
namespace tr1 { namespace regex_constants {

typedef bitmask_type syntax_option_type;
// these flags are required:
static const syntax_option_type normal;
static const syntax_option_type 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 = normal;
static const syntax_option_type JavaScript = normal;
static const syntax_option_type JScript = normal;
// these flags are optional, if the functionality is supported
// then the flags shall take these names.
static const syntax_option_type basic;
static const syntax_option_type extended;
```

```
static const syntax_option_type awk;
static const syntax_option_type grep;
static const syntax_option_type egrep;
static const syntax_option_type sed = basic;
static const syntax_option_type perl;

} // namespace regex_constants
} // namespace tr1
```

The type syntax_option_type is an implementation defined bitmask type (§17.3.2.1.2). Setting its elements has the effects listed in table 7.2, a valid value of type `syntax_option_type` will always have exactly one of the elements `normal`, `basic`, `extended`, `awk`, `grep`, `egrep`, `sed` or `perl` set.

Table 7.2: `syntax_option_type` effects

Element	**Effect if set**
normal	Specifies that the grammar recognized by the regular expression engine uses its normal semantics: that is the same as that given in the ECMA-262, ECMAScript Language Specification, Chapter 15 part 10, RegExp (Regular Expression) Objects .
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	The same as normal.
JavaScript	The same as normal.
JScript	The same as normal.
basic	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX basic regular expressions [3] in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Base Definitions and Headers, Section 9, Regular Expressions .

continued from previous page	
extended	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX extended regular expressions [3] 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 [3] awk in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, awk .
grep	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX utility [3] grep in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, Utilities, grep .
egrep	Specifies that the grammar recognized by the regular expression engine is the same as that used by POSIX utility [3] grep when given the -E option in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX), Shells and Utilities, Section 4, Utilities, grep .
sed	The same as basic.
perl	Specifies that the grammar recognized by the regular expression is an implementation defined extension of the normal syntax.

7.5.2 Bitmask Type `match_flag_type` [tr.re.matchflag]

```
namespace tr1 { namespace regex_constants{

typedef bitmask_type 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_partial;
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_perl;
static const match_flag_type format_no_copy;
```

```
  static const match_flag_type format_first_only;

  } // namespace regex_constants
  } // namespace tr1
```

The type match_flag_type is an implementation defined bitmask type (§17.3.2.1.2). When matching a regular expression against a sequence of characters `[first, last)` then setting its elements has the effects listed in table 7.3

Table 7.3: `match_flag_type` effects when obtaining a match against a character container sequence `[first,last)`.

Element	**Effect if set**
match_default	Specifies that matching of regular expressions proceeds without any modification of the normal rules used in ECMA-262, ECMAScript Language Specification, Chapter 15 part 10, RegExp (Regular Expression) Objects
match_not_bol	Specifies that the expression `"^"` should not be matched against the sub-sequence [first,first).
match_not_eol	Specifies that the expression `"$"` should not be matched against the sub-sequence [last,last).
match_not_bow	Specifies that the expression `"\b"` should not be matched against the sub-sequence [first,first).
match_not_eow	Specifies that the expression `"\b"` should not be matched against the sub-sequence [last,last).
match_any	Specifies that if more than one match is possible then any match is an acceptable result.
match_not_null	Specifies that the expression can not be matched against an empty sequence.
match_continuous	Specifies that the expression must match a sub-sequence that begins at first .
match_partial	Specifies that if no match can be found, then it is acceptable to return a match [from, last) where from!=last, if there exists some sequence of characters [from,to) of which [from,last) is a prefix, and which would result in a full match.
match_prev_avail	Specifies that –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 (RE.7) and iterators (RE.8).

continued from previous page	
format_default	Specifies that when a regular expression match is to be replaced by a new string, that the new string is constructed using the rules used by the ECMAScript replace function in ECMA-262, ECMAScript Language Specification, Chapter 15 part 5.4.11 String.prototype.replace. . In addition during search and replace operations then 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	Specifies that when a regular expression match is to be replaced by a new string, that the new string is constructed using the rules used by the POSIX sed utility [3] in IEEE Std 1003.1-2001, Portable Operating SystemInterface (POSIX), Shells and Utilities.
format_perl	Specifies that when a regular expression match is to be replaced by a new string, that the new string is constructed using an implementation defined superset of the rules used by the ECMAScript replace function in ECMA-262, ECMAScript Language Specification, Chapter 15 part 5.4.11 String.prototype.replace .
format_no_copy	When specified during a search and replace operation, then sections of the character container sequence being searched that do match the regular expression, are not copied to the output string.
format_first_only	When specified during a search and replace operation, then only the first occurrence of the regular expression is replaced.

7.5.3 Implementation defined syntax_type [tr.re.syntype]

```
namespace tr1 { namespace regex_constants{

typedef implementation-defined syntax_type;

static const syntax_type syntax_char;
static const syntax_type syntax_open_mark;
static const syntax_type syntax_close_mark;
static const syntax_type syntax_dollar;
static const syntax_type syntax_caret;
static const syntax_type syntax_dot;
static const syntax_type syntax_star;
static const syntax_type syntax_plus;
static const syntax_type syntax_question;
static const syntax_type syntax_open_set;
static const syntax_type syntax_close_set;
```

```
static const syntax_type syntax_or;
static const syntax_type syntax_escape;
static const syntax_type syntax_dash;
static const syntax_type syntax_open_brace;
static const syntax_type syntax_close_brace;
static const syntax_type syntax_digit;
static const syntax_type syntax_comma;
static const syntax_type syntax_equal;
static const syntax_type syntax_colon;
static const syntax_type syntax_not;

} // namespace regex_constants
} // namespace tr1
```

The type syntax_type is an implementation defined enumeration type (§17.3.2.1.2). Values of type syntax_type represent how individual characters should be interpreted within a localized regular expression grammar, table 7.4 shows which special characters defined in ECMA-262, ECMAScript Language Specification, Chapter 15 part 10, RegExp (Regular Expression) Objects, are equivalent to which syntax_type values in the C locale:

Table 7.4: `syntax_type` values in the C locale

Value	Equivalent character(s) in the syntax specified by ECMA-262, ECMAScript Language Specification, Chapter 15 part 10, RegExp (Regular Expression) Objects.
`syntax_char`	Any character not listed below.
`syntax_open_mark`	`(`
`syntax_close_mark`	`)`
`syntax_dollar`	`$`
`syntax_caret`	`^`
`syntax_dot`	`.`
`syntax_star`	`*`
`syntax_plus`	`+`
`syntax_question`	`?`
`syntax_open_set`	`[`
`syntax_close_set`	`]`
`syntax_or`	`|`
`syntax_escape`	`\`
`syntax_dash`	`-`
`syntax_open_brace`	`{`
`syntax_close_brace`	`}`
`syntax_digit`	`0123456789`
`syntax_comma`	`,`
`syntax_equal`	`=`
`syntax_colon`	`:`

continued from previous page	
`syntax_not`	`!`

7.5.4 Implementation defined `escape_syntax_type` [tr.re.escsyn]

```
namespace tr1 { namespace regex_constants {

typedef implementation-defined escape_syntax_type;

static const escape_syntax_type escape_type_word_assert;
static const escape_syntax_type escape_type_not_word_assert;
static const escape_syntax_type escape_type_control_f;
static const escape_syntax_type escape_type_control_n;
static const escape_syntax_type escape_type_control_r;
static const escape_syntax_type escape_type_control_t;
static const escape_syntax_type escape_type_ascii_control;
static const escape_syntax_type escape_type_hex;
static const escape_syntax_type escape_type_unicode;
static const escape_syntax_type escape_type_identity;
static const escape_syntax_type escape_type_backref;
static const escape_syntax_type escape_type_decimal;
static const escape_syntax_type escape_type_class;
static const escape_syntax_type escape_type_not_class;

} // namespace regex_constants
} // namespace tr1
```

The type `escape_syntax_type` is an implementation defined enumeration type (§17.3.2.1.2). Values of type `escape_syntax_type` represent how individual escaped characters should be interpreted within a localized regular expression grammar, table 7.5 shows which special characters defined in ECMA-262, ECMAScript Language Specification, Chapter 15 part 10, RegExp (Regular Expression) Objects, are equivalent to which `escape_syntax_type` values in the C locale:

Table 7.5: `escape_syntax_type` values in the C locale

Value	Equivalent character(s) in syntax specified in ECMA-262, ECMAScript Language Specification, Chapter 15 part 10, RegExp (Regular Expression) Objects
`escape_type_word_assert`	`b`
`escape_type_not_word_assert`	`B`
`escape_type_control_f`	`f`
`escape_type_control_n`	`n`
`escape_type_control_r`	`r`
`escape_type_control_t`	`t`
`escape_type_ascii_control`	`c`

continued from previous page	
`escape_type_hex`	`x`
`escape_type_unicode`	`u`
`escape_type_identity`	`\ |^$*+?{}.()[]`
`escape_type_backref`	`123456789`
`escape_type_decimal`	`0`
`escape_type_not_class`	Any upper case character not listed above.
`escape_type_class`	Any non-upper case character not listed above.

7.5.5 Implementation defined `error_type` [tr.re.err]

```
namespace tr1 { namespace regex_constants {

typedef implementation defined error_type;

static const error_type error_collate;
static const error_type error_ctype;
static const error_type error_escape;
static const error_type error_subreg;
static const error_type error_brack;
static const error_type error_paren;
static const error_type error_brace;
static const error_type error_badbrace;
static const error_type error_range;
static const error_type error_space;
static const error_type error_badrepeat;
static const error_type error_complexity;
static const error_type error_stack;

} // namespace regex_constants
} // namespace tr1
```

The type `error_type` is an implementation defined enumeration type (§17.3.2.1.2). Values of type `error_type` represent the error conditions as described in table 7.6:

Table 7.6: `escape_syntax_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_subreg`	The expression contained an invalid backreference.
`error_brack`	The expression contained mismatched `[` and `]`.

continued from previous page	
`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 `bad_expression` [tr.re.badexp]

```
class bad_expression : public std::runtime_error
{
public:
   explicit bad_expression(const std::string& what_arg);
};
```

The class `bad_expression` defines the type of objects thrown as exceptions to report errors during the conversion from a string representing a regular expression to a finite state machine.

```
bad_expression(const string& what_arg );
```

Effects: Constructs an object of class `bad_expression`.
Postcondition: `strcmp(what(), what_arg.c_str()) == 0`.

7.7 Class template `regex_traits` [tr.re.traits]

```
template <class charT>
struct regex_traits
{
public:
   typedef charT                          char_type;
   typedef std::size_t                    size_type;
   typedef std::basic_string<char_type>   string_type;
   typedef std::locale                    locale_type;
   typedef bitmask_type                   char_class_type;

   struct sentry
```

```
    {
       sentry(regex_traits<charT>&);
       operator void*();
    };

    regex_traits();
    static size_type length(const char_type* p);
    regex_constants::syntax_type syntax_type(charT c)const;
    regex_constants::escape_syntax_type escape_syntax_type(charT c) const;
    charT translate(charT c, bool icase) const;
    string_type transform(const string_type& in) const;
    string_type transform_primary(const string_type& in) const;
    char_class_type lookup_classname(const string_type& name) const;
    string_type lookup_collatename(const string_type& name) const;
    bool is_class(charT c, char_class_type f) const;
    template<class InputIterator>
    int toi(InputIterator& first, InputIterator last, int radix) const;
    locale_type imbue(locale_type l);
    locale_type getloc()const;
    std::string error_string(regex_constants::error_type) const;
};
```

The class template `regex_traits` is capable of being specialized for the types `char` and `wchar_t` and satisfies the requirements for a regular expression traits class (7.2).

```
typedef bitmask_type                 char_class_type;
```

The type `char_class_type` is used to represent a character classification and is capable of holding an implementation defined superset of the values held by `std::ctype_base::mask` (22.2.1).

```
struct sentry
{
   sentry(regex_traits<charT>&);
   operator void*();
};
```

An object of type `regex_traits<charT>::sentry` shall be constructed from a `regex_traits` object, and tested to be not equal to null, before any of the member functions of that object other than `length`, `getloc`, and `imbue` shall be called. Type `sentry` performs implementation defined initialization of the traits class object, and represents an opportunity for the traits class to cache data obtained from the locale object.

```
static size_type length(const char_type* p);
```

Effects: returns `char_traits<charT>::length(p);`

```
regex_constants::syntax_type syntax_type(charT c) const;
```

Effects: for a character `c` in the right hand column of table reftab:re:syntaxtype, returns the corresponding `regex_constants::syntax_type` value.

```
regex_constants::escape_syntax_type escape_syntax_type(charT c) const;
```

Effects: For a character `c` in the right hand column of table 7.5, returns the corresponding `regex_constants::escape_syntax_type` value.

```
charT translate(charT c, bool icase) const;
```

Effects: returns `(icase ? use_facet<ctype<charT> >(getloc()).tolower(c) : c)`

```
string_type transform(const string_type& in) const;
```

Effects: returns `use_facet<collate<charT> >(getloc()).transform(in.begin(), in.end())`.

```
string_type transform_primary(const string_type& in) const;
```

Effects: if `typeid(use_facet<collate<charT> >) == typeid(collate_byname<charT>)` and the form of the sort key returned by `collate_byname<charT>::transform` is known and can be converted into a primary sort key, then returns that key, otherwise returns a empty string.

```
char_class_type lookup_classname(const string_type& name) const;
```

Effects: returns an implementation defined value that represents the character classification `name`. If name is not recognized then returns a value that compares equal to 0. At least the names `"d"`, `"w"`, `"s"`, `"alnum"`, `"alpha"`, `"blank"`, `"cntrl"`, `"digit"`, `"graph"`, `"lower"`, `"print"`, `"punct"`, `"space"`, `"upper"` and `"xdigit"`, shall be recognized. The value returned shall be independent of the case of the characters in `name`.

```
string_type lookup_collatename(const string_type& name) const;
```

Effects: returns the sequence of one or more characters that represents the collating element `name`. Returns an empty string if name is not recognized. At least the names specified in IEEE Std 1003.1-2001, Portable Operating System Interface (POSIX) [3], Base Definitions and Headers, Section 6.1, Portable Character Set shall be recognized.

```
bool is_class(charT c, char_class_type f) const;
```

Effects: determines if the character `c` is a member of the character classification represented by `f`.
Returns: converts `f` into a value `m` of type `std::ctype_base::mask` in an implementation defined manner, and returns `true` if `use_facet<ctype<charT> >(getloc()).is(c, m)` is `true`. Otherwise returns `true` if `f & lookup_classname("w") == lookup_classname("w") and c == '_'`, otherwise returns `false`.

```
template <class InputIterator>
int toi(InputIterator& first, InputIterator last, int radix) const;
```

Precondition: argument `radix` shall take one of the values 8, 10 or 16.
Effects: constructs an object `result` of type `int`. If `first == last` or if `is_class(*first, lookup_classname("d")) == false` then sets `result` equal to -1. Otherwise constructs a `basic_istream<charT>` object which uses an implementation defined stream buffer type which represents the character sequence `[first,last)`, and sets the format flags on that object as appropriate for argument radix. Performs unsigned integer formatted input on the `basic_istream<charT>`

object setting the value `result` to the value obtained, and updates `first` to point to the first non-digit character in the sequence `[first,last)`.
Returns: the value `result`.
Postcondition: `is_class(*first, lookup_classname("d")) == false`.

```
locale_type imbue(locale_type loc);
```

Effects: Imbues `this` with a copy of the locale `loc`. The effect of calling `imbue` is to invalidate all cached data held by `this`. No member functions other than `length`, `getloc`, and `imbue` may be called until an object of type `sentry` has been copy-constructed from `*this`.
Returns: if no locale has been previously imbued then a copy of the global locale in effect at the time of construction of `this`, otherwise a copy of the last argument passed to `imbue`.
Postcondition: `getloc() == loc`.

```
locale_type getloc()const;
```

Returns: if no locale has been imbued then a copy of the global locale in effect at the time of construction of `this`, otherwise a copy of the last argument passed to `imbue`.

```
std::string error_string(regex_constants::error_type e) const;
```

Returns: a human readable error string for error condition `e`.

7.8 Class template `basic_regex` [tr.re.regex]

For a char-like type `charT` , the template class `basic_regex` describes objects that represent a regular expression constructed from a sequence of `charT`s. In the rest of this clause, `charT` denotes such a given char-like type. Storage for the regular expression is allocated and freed as necessary by the member functions of class `basic_regex`.
The template class `basic_regex` conforms to the requirements of a Sequence, as specified in [lib.sequence.reqmts], except that only operations defined for const-qualified Sequences are supported. Objects of type specialization of `basic_regex` are responsible for converting the sequence of `charT` objects to an internal representation. It is not specified what form this representation takes, nor how it is accessed by algorithms that operate on regular expressions. [*Note:* implementations will typically declare some function template as friends of `basic_regex` to achieve this *—end note*]
The regular expression grammar recognized by type specialization of basic_regex is described in ECMA-262 [2], ECMAScript Language Specification, Chapter 15 part 10, RegExp (Regular Expression) Objects (FWD.1), and is modified according to any `syntax_option_type` flags specified when constructing the object or assigning a regular expression to it (RE.3.1.1). In addition to the features specified in ECMA-262, the following features shall also be recognized:

- The character classes `\d` `\D` `\w` `\W` `\s` `\S` are sensitive to the locale encapsulated by the traits class.
- Expressions of the form `[[:class-name:]]` are recognized, and are sensitive to the locale encapsulated by the traits class. The range of values for class-name is determined by the traits class, but at least the following names are recognized: `alnum`, `alpha`, `blank`, `cntrl`, `digit`, `graph`, `lower`, `print`, `punct`, `space`, `upper`, `xdigit`
- Expressions of the form `[[.collating-name.]]` are recognized, the range of values for collating-name is determined by the traits class.

- Expressions of the form `[[=collating-name=]]` are recognized, the range of values for collating-name is determined by the traits class.
- Expressions of the form `[a-b]` are locale sensitive when the flag `regex_constants::collate` is passed to the regular expression constructor.

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; no `basic_regex` object shall call any locale dependent C or C++ API, including the formatted string input functions, instead it shall call the appropriate traits member function to achieve the required effect.
The transformation from a sequence of characters to a finite state machine is accomplished by first by transforming the sequence of characters to the sequence of tokens obtained by calling `traits_inst.syntax_type(c)` for each input character `c`. The regular expression grammar is then applied to the sequence of tokens in order to construct the finite state machine [*Note:* this is to allow the regular expression syntax to be localized to a specific character set; for example to use Far Eastern ideographs, rather than Latin characters —*end note*]. Where `traits_inst.syntax_type(c)` returns `syntax_escape`, then the implementation shall call `traits_inst.escape_syntax_type(c)` for the character following the escape, in order to determine the grammatical meaning of the escape sequence. When `traits_inst.escape_syntax_type(c)` returns `escape_type_class` or `escape_type_not_class`, then the single character following the escape is converted to a string `n`, and passed to `traits_inst.lookup_classname(n)`, to determine the character classification to be matched against. If `traits_inst.lookup_classname(n)` returns null, then the escape is treated as an identity escape.
Where the regular expression grammar requires the conversion of a sequence of characters to an integral value, this is accomplished by calling `traits::toi`.
Where the regular expression grammar requires the conversion of a sequence of characters representing a named collating element to the character(s) it represents, this is accomplished by calling `traits::lookup_collatename`.
Where the regular expression grammar requires the conversion of a sequence of characters representing a named character class to an internal representation, this is accomplished by calling `traits::lookup_classname`. The results from subsequent calls to this function can be bitwise OR'ed together and subsequently passed to `traist::is_class`.
The behavior of the internal finite state machine representation, when used to match a sequence of characters is as described in ECMAScript Language Specification [2], Chapter 15 part 10, RegExp (Regular Expression) Objects. The behavior is modified according to any `match_flag_type` flags specified (RE.3.1.2) when using the regular expression object in one of the regular expression algorithms 7.10. The behavior is also localized by interaction with the traits class template parameter as follows:
During matching of a regular expression finite state machine against a sequence of characters, two characters *c* and *d* are compared using `traits_inst.translate(c, getflags() & regex_constants::icase) == traits_inst.translate(d, getflags() & regex_constants::icase)`.
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 `getflags() & regex_constants::collate` is true, then the character `c` is matched if `traits_inst.transform(string_type(1,c1)) <= traits_inst.transform(string_type(1,c)) && traits_inst.transform(strin`

`type(1,c)) <= traits_inst.transform(string_type(1,c2))`, otherwise `c` is matched if `c1 <= c && c <= c2`.

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 a sort keys using `traits::transform_primary`, and then comparing the sort keys for equality.

During matching of a regular expression finite state machine against a sequence of characters, a character `c` is a member of character class `some_name`, if `traits_inst.is_class(c, traits_inst.lookup_classname("some_name"))`.

The functions described in this clause can report errors by throwing exceptions of type `bad_expression`.

```
template <class charT,
          class traits = regex_traits<charT>,
          class Allocator = allocator<charT> >
class basic_regex
{
public:
   // types:
   typedef          charT                                value_type;
   typedef          implementation-defined               const_iterator;
   typedef          const_iterator                       iterator;
   typedef typename Allocator::reference                 reference;
   typedef typename Allocator::const_reference           const_reference;
   typedef typename Allocator::difference_type           difference_type;
   typedef typename Allocator::size_type                 size_type;
   typedef          Allocator                            allocator_type;
   typedef          regex_constants::syntax_option_type  flag_type;
   typedef typename traits::locale_type                  locale_type;

   // constants:
   static const regex_constants::syntax_option_type normal = regex_constants::normal
   static const regex_constants::syntax_option_type icase = regex_constants::icase;
   static const regex_constants::syntax_option_type nosubs = regex_constants::nosubs
   static const regex_constants::syntax_option_type optimize = regex_constants::opti
   static const regex_constants::syntax_option_type collate = regex_constants::colla
   static const regex_constants::syntax_option_type ECMAScript = normal;
   static const regex_constants::syntax_option_type JavaScript = normal;
   static const regex_constants::syntax_option_type JScript = normal;
   // these flags are optional, if the functionality is supported
   // then the flags shall take these names.
   static const regex_constants::syntax_option_type basic = regex_constants::basic;
   static const regex_constants::syntax_option_type extended = regex_constants::exte
   static const regex_constants::syntax_option_type awk = regex_constants::awk;
   static const regex_constants::syntax_option_type grep = regex_constants::grep;
   static const regex_constants::syntax_option_type egrep = regex_constants::egrep;
   static const regex_constants::syntax_option_type sed = basic = regex_constants::s
```

```
static const regex_constants::syntax_option_type perl = regex_constants::perl;

// construct/copy/destroy:
explicit basic_regex(const Allocator& a = Allocator());
explicit basic_regex(const charT* p, flag_type f = regex_constants::normal,
                     const Allocator& a = Allocator());
basic_regex(const charT* p1, const charT* p2, flag_type f = regex_constants::norm
            const Allocator& a = Allocator());
basic_regex(const charT* p, size_type len, flag_type f,
            const Allocator& a = Allocator());
basic_regex(const basic_regex&);
template <class ST, class SA>
explicit basic_regex(const basic_string<charT, ST, SA>& p,
                     flag_type f = regex_constants::normal,
                     const Allocator& a = Allocator());
template <class InputIterator>
basic_regex(InputIterator first, inputIterator last,
            flag_type f = regex_constants::normal,
            const Allocator& a = Allocator());

~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);

// iterators:
const_iterator begin() const;
const_iterator end() const;
// capacity:
size_type size() const;
size_type max_size() const;
bool empty() const;
unsigned mark_count() const;

//
// modifiers:
basic_regex& assign(const basic_regex& that);
basic_regex& assign(const charT* ptr, flag_type f = regex_constants::normal);
basic_regex& assign(const charT* first, const charT* last,
                    flag_type f = regex_constants::normal);
template <class string_traits, class A>
basic_regex& assign(const basic_string<charT, string_traits, A>& s,
                    flag_type f = regex_constants::normal);
template <class InputIterator>
basic_regex& assign(InputIterator first, InputIterator last,
```

```
                        flag_type f = regex_constants::normal);

    // const operations:
    Allocator get_allocator() const;
    flag_type getflags() const;
    basic_string<charT> str() const;
    int compare(basic_regex&) const;
    // locale:
    locale_type imbue(locale_type loc);
    locale_type getloc() const;
    // swap
    void swap(basic_regex&) throw();
};
```

7.8.1 `basic_regexp` constants [tr.re.regex.const]

```
static const regex_constants::syntax_option_type normal = regex_constants::normal;
static const regex_constants::syntax_option_type icase = regex_constants::icase;
static const regex_constants::syntax_option_type nosubs = regex_constants::nosubs;
static const regex_constants::syntax_option_type optimize = regex_constants::optimiz
static const regex_constants::syntax_option_type collate = regex_constants::collate;
static const regex_constants::syntax_option_type ECMAScript = normal;
static const regex_constants::syntax_option_type JavaScript = normal;
static const regex_constants::syntax_option_type JScript = normal;
// these flags are optional, if the functionality is supported
// then the flags shall take these names.
static const regex_constants::syntax_option_type basic = regex_constants::basic;
static const regex_constants::syntax_option_type extended = regex_constants::extende
static const regex_constants::syntax_option_type awk = regex_constants::awk;
static const regex_constants::syntax_option_type grep = regex_constants::grep;
static const regex_constants::syntax_option_type egrep = regex_constants::egrep;
static const regex_constants::syntax_option_type sed = basic = regex_constants::sed;
static const regex_constants::syntax_option_type perl = regex_constants::perl;
```

The static constant members are provided as synonyms for the constants declared in namespace `regex_constants`; for each constant of type syntax_option_type declared in namespace `regex_constants` then a constant with the same name, type and value shall be declared within the scope of `basic_regex`.

7.8.2 `basic_regexp` constructors [tr.re.regex.construct]

In all basic_regex 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.

```
basic_regex(const Allocator& a = Allocator());
```

Effects: Constructs an object of class `basic_regex` . The postconditions of this function are indicated in Table 7.7

Table 7.7: `basic_regex(const Allocator&)` effects

Element	Value
`empty()`	`true`
`size()`	`0`
`str()`	`basic_string<charT>()`

```
basic_regex(const charT* p,
            flag_type f = regex_constants::normal,
            const Allocator& a = Allocator());
```

Requires: *p* shall not be a null pointer.
Throws: `bad_expression` if *p* is not a valid regular expression.
Effects: Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the null-terminated string *p*, and interpreted according to the flags specified in *f*. The postconditions of this function are indicated in Table 7.8.

Table 7.8: `basic_regex(const charT* p, flag_type f, const Allocator&)` effects

Element	Value
`empty()`	`false`
`size()`	`char_traits<charT>::length(p)`
`str()`	`basic_string<charT>(p)`
`getflags()`	`f`
`mark_count()`	The number of marked sub-expressions within the expression.

```
basic_regex(const charT* p1, const charT* p2,
            flag_type f = regex_constants::normal,
            const Allocator& a = Allocator());
```

Requires: `p1` and `p2` are not null pointers, `p1 < p2`.
Throws: `bad_expression` if `[p1,p2)` is not a valid regular expression.
Effects: Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the sequence of characters `[p1,p2)`, and interpreted according the flags specified in *f*. The postconditions of this function are indicated in Table 7.9.

Table 7.9: `basic_regex(const charT* p1, const charT* p2, flag_type f, const Allocator&)` effects

Element	Value
`empty()`	`false`
`size()`	`std::distance(p1,p2)`
`str()`	`basic_string<charT>(p1,p2)`

continued from previous page	
`getflags()`	`f`
`mark_count()`	The number of marked sub-expressions within the expression.

```
basic_regex(const charT* p, size_type len,
            flag_type f,
            const Allocator& a = Allocator());
```

Requires: *p* shall not be a null pointer, `len < max_size()`.
Throws: `bad_expression` if *p* is not a valid regular expression.
Effects: Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the sequence of characters `[p, p+len)`, and interpreted according the flags specified in *f*. The postconditions of this function are indicated in Table 7.10

Table 7.10: `basic_regex(const charT* p1, size_type len, flag_type f, const Allocator&)` effects

Element	**Value**
`empty()`	`false`
`size()`	`len`
`str()`	`basic_string<charT>(p,len)`
`getflags()`	`f`
`mark_count()`	The number of marked sub-expressions within the expression.

```
basic_regex(const basic_regex& e);
```

Effects: Constructs an object of class `basic_regex` as a copy of the object `e`. The postconditions of this function are indicated in Table 7.11

Table 7.11: `basic_regex(const basic_regexp& e)` effects

Element	**Value**
`empty()`	`e.empty`
`size()`	`e.size()`
`str()`	`e.str()`
`getflags()`	`e.getflags()`
`mark_count()`	`e.mark_count()`

```
template <class ST, class SA>
basic_regex(const basic_string<charT, ST, SA>& s,
            flag_type f = regex_constants::normal,
            const Allocator& a = Allocator());
```

Throws: `bad_expression` if `s` is not a valid regular expression.
Effects: Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the string `s`, and interpreted according to the flags specified in `f`. The postconditions of this function are indicated in Table 7.12

Table 7.12: `basic_regex(const basic_string&)` effects

Element	Value
`empty()`	`false`
`size()`	`s.size()`
`str()`	`s`
`getflags()`	`f`
`mark_count()`	The number of marked sub-expressions within the expression.

```
template <class ForwardIterator>
basic_regex(ForwardIterator first, ForwardIterator last,
            flag_type f = regex_constants::normal,
            const Allocator& a = Allocator());
```

Throws: `bad_expression` if the sequence *[first, last)* is not a valid regular expression.
Effects: Constructs an object of class `basic_regex`; the object's internal finite state machine is constructed from the regular expression contained in the sequence of characters `[first, last)`, and interpreted according to the flags specified in `f`. The postconditions of this function are indicated in Table 7.13.

Table 7.13: `basic_regex(ForwardIterator first, ForwardIterator last, flag_type f, const Allocator&)` effects

Element	Value
`empty()`	`false`
`size()`	`std::distance(first, last)`
`str()`	`std::basic_string<charT>(first, last)`
`getflags()`	`f`
`mark_count()`	The number of marked sub-expressions within the expression.

```
basic_regex& operator=(const basic_regex& e);
```

Effects: Returns the result of `assign(e.str(), e.getflags())`.

```
basic_regex& operator=(const charT* ptr);
```

Requires: `ptr` shall not be a null pointer.
Effects: Returns the result of `assign(ptr)`.

```
template <class ST, class SA>
basic_regex& operator=(const basic_string<charT, ST, SA>& p);
```

Effects: Returns the result of `assign(p)`.

7.8.3 `basic_regexp` iterators [tr.re.regex.iter]

```
const_iterator begin() const;
```

Effects: Returns a starting iterator to a sequence of characters representing the regular expression.

```
const_iterator end() const;
```

Effects: Returns termination iterator to a sequence of characters representing the regular expression.

7.8.4 `basic_regexp` capacity [tr.re.regex.cap]

```
size_type size() const;
```

Effects: Returns the length of the sequence of characters representing the regular expression.

```
size_type max_size() const;
```

Effects: Returns the maximum length of the sequence of characters representing the regular expression.

```
bool empty() const;
```

Effects: Returns true if the object does not contain a valid regular expression, otherwise false .

```
unsigned mark_count() const;
```

Effects: Returns the number of marked sub-expressions within the regular expresion.

7.8.5 `basic_regexp` assign [tr.re.regex.assign]

```
basic_regex& assign(const basic_regex& that);
```

Effects: Returns `assign(that.str(), that.getflags())`.

```
basic_regex& assign(const charT* ptr, flag_type f = regex_constants::normal);
```

Effects: Returns `assign(string_type(ptr), f)`.

```
basic_regex& assign(const charT* first, const charT* last,
                    flag_type f = regex_constants::normal);
```

Effects: Returns `assign(string_type(first, last), f)`.

```
template <class string_traits, class A>
basic_regex& assign(const basic_string<charT, string_traits, A>& s,
                    flag_type f = regex_constants::normal);
```

Throws: `bad_expression` if `s` is not a valid regular expression.
Returns: `*this`.
Effects: Assigns the regular expression contained in the string `s`, interpreted according the flags specified in `f`. The postconditions of this function are indicated in Table 7.14.

Table 7.14: `basic_regex& assign(const basic_string<charT, string_traits, A>& s, flag_type f)` effects

Element	Value
`empty()`	`false`
`size()`	`s.size()`
`str()`	`s`
`getflags()`	`f`
`mark_count()`	The number of marked sub-expressions within the expression.

```
template <class InputIterator>
basic_regex& assign(InputIterator first, InputIterator last,
                    flag_type f = regex_constants::normal);
```

Requires: The type `InputIterator` corresponds to the Input Iterator requirements (§24.1.1).
Effects: Returns `assign(string_type(first, last), f)`.

7.8.6 `basic_regexp` constant operations [tr.re.regex.operations]

```
Allocator get_allocator() const;
```

Effects: Returns a copy of the Allocator that was passed to the object's constructor.

```
flag_type getflags() const;
```

Effects: Returns a copy of the regular expression syntax flags that were passed to the object's constructor, or the last call to `assign`.

```
basic_string<charT> str() const;
```

Effects: Returns a copy of the character sequence passed to the object's constructor, or the last call to `assign`.

```
int compare(basic_regex& e)const;
```

Effects: If `getflags() == e.getflags()` then returns `str().compare(e.str())`, otherwise returns `getflags() - e.getflags()`.

7.8.7 `basic_regexp` locale [tr.re.regex.locale]

```
locale_type imbue(locale_type l);
```

Effects: Returns the result of `traits_inst.imbue(l)` where `traits_inst` is a (default initialized) instance of the template parameter `traits` stored within the object. Calls to `imbue` invalidate any currently contained regular expression.
Postcondition: `empty() == true`.

```
locale_type getloc() const;
```

Effects: Returns the result of `traits_inst.getloc()` where `traits_inst` is a (default initialized) instance of the template parameter `traits` stored within the object.

7.8.8 `basic_regexp` swap [tr.re.regex.swap]

```
void swap(basic_regex& e) throw();
```

Effects: Swaps the contents of the two regular expressions.
Postcondition: `*this` contains the characters that were in `e`, `e` contains the regular expression that was in `*this`.
Complexity: constant time.

7.8.9 `basic_regex` non-member functions [tr.re.regex.nonmemb]

7.8.9.1 `basic_regex` non-member comparison operators [tr.re.regex.comp]

```
template <class charT, class traits, class Allocator>
bool operator == (const basic_regex<charT, traits, Allocator>& lhs,
                  const basic_regex<charT, traits, Allocator>& rhs);
```

Effects: Returns `lhs.compare(rhs) == 0`.

```
template <class charT, class traits, class Allocator>
bool operator != (const basic_regex<charT, traits, Allocator>& lhs,
                  const basic_regex<charT, traits, Allocator>& rhs);
```

Effects: Returns `lhs.compare(rhs) != 0`.

```
template <class charT, class traits, class Allocator>
bool operator < (const basic_regex<charT, traits, Allocator>& lhs,
                 const basic_regex<charT, traits, Allocator>& rhs);
```

Effects: Returns `lhs.compare(rhs) < 0`.

```
template <class charT, class traits, class Allocator>
bool operator <= (const basic_regex<charT, traits, Allocator>& lhs,
                  const basic_regex<charT, traits, Allocator>& rhs);
```

Effects: Returns `lhs.compare(rhs) <= 0`.

```
template <class charT, class traits, class Allocator>
bool operator >= (const basic_regex<charT, traits, Allocator>& lhs,
                  const basic_regex<charT, traits, Allocator>& rhs);
```

Effects: Returns `lhs.compare(rhs) >= 0`.

```
template <class charT, class traits, class Allocator>
bool operator > (const basic_regex<charT, traits, Allocator>& lhs,
                 const basic_regex<charT, traits, Allocator>& rhs);
```

Effects: Returns `lhs.compare(rhs) > 0`.

7.8.9.2 `basic_regex` inserter [tr.re.regex.inserter]

```
template <class charT, class io_traits, class re_traits,
          class Allocator>
basic_ostream<charT, io_traits>&
   operator << (basic_ostream<charT, io_traits>& os
                const basic_regex<charT, re_traits, Allocator>& e);
```

Effects: Returns `(os << e.str())`.

7.8.9.3 `basic_regex` non-member swap [tr.re.regex.nmswap]

```
template <class charT, class traits, class Allocator>
void swap(basic_regex<charT, traits, Allocator>& lhs,
          basic_regex<charT, traits, Allocator>& rhs);
```

Effects: calls `lhs.swap(rhs)`.

7.8.9.4 Class template `sub_match` [tr.re.submatch]

Class template `sub_match` denotes the sequence of characters matched by a particular marked sub-expression.

When the marked sub-expression denoted by an object of type `sub_match<>` participated in a regular expression match then 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` contained undefined values.

If an object of type `sub_match<>` represents sub-expression 0 - that is to say the whole match - then member `matched` is always true, unless a partial match was obtained as a result of the flag `match_partial` being passed to a regular expression algorithm, in which case member `matched` is false, and members `first` and `second` represent the character range that formed the partial match.

```
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;
```

```
};
```

7.8.10 `sub_match` members [tr.re.submatch.members]

```
static difference_type length();
```

Returns: `(matched ? 0 : distance(first, second))`.

```
operator basic_string<value_type>()const;
```

Returns: `(matched ? basic_string<value_type>(first, second) : basic_string<value_type>())`.

```
basic_string<value_type> str()const;
```

Returns: `(matched ? basic_string<value_type>(first, second) : basic_string<value_type>())`.

```
int compare(const sub_match& s)const;
```

Returns: `str().compare(s.str())`.

```
int compare(const basic_string<value_type>& s)const;
```

Returns: `str().compare(s)`.

```
int compare(const value_type* s)const;
```

Returns: `str().compare(s)`.

7.8.11 `sub_match` non-member operators [tr.re.submatch.op]

```
template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs.compare(rhs) == 0`.

```
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs.compare(rhs) != 0`.

```
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs.compare(rhs) < 0`.

```
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs.compare(rhs) <= 0`.

```
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs.compare(rhs) >= 0`.

```
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs.compare(rhs) > 0`.

```
template <class BidirectionalIterator>
bool operator == (typename iterator_traits<BidirectionalIterator>::value_type const
                  const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs == rhs.str()`.

```
template <class BidirectionalIterator>
bool operator != (typename iterator_traits<BidirectionalIterator>::value_type const
                  const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs != rhs.str()`.

```
template <class BidirectionalIterator>
bool operator < (typename iterator_traits<BidirectionalIterator>::value_type const*
                 const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs < rhs.str()`.

```
template <class BidirectionalIterator>
bool operator > (typename iterator_traits<BidirectionalIterator>::value_type const*
                 const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs > rhs.str()`.

```
template <class BidirectionalIterator>
bool operator >= (typename iterator_traits<BidirectionalIterator>::value_type const
                  const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs >= rhs.str()`.

```
template <class BidirectionalIterator>
bool operator <= (typename iterator_traits<BidirectionalIterator>::value_type const
                  const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs <= rhs.str()`.

```
template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const
```

Returns: `lhs.str() == rhs`.

```
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const*
```

Returns: `lhs.str() != rhs`.

```
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const*
```

Returns: `lhs.str() < rhs`.

```
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const*
```

Returns: `lhs.str() > rhs`.

```
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const*
```

Returns: `lhs.str() >= rhs`.

```
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const*
```

Returns: `lhs.str() <= rhs`.

```
template <class BidirectionalIterator>
bool operator == (typename iterator_traits<BidirectionalIterator>::value_type const&
                  const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs == rhs.str()`.

```
template <class BidirectionalIterator>
bool operator != (typename iterator_traits<BidirectionalIterator>::value_type const&
                  const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs != rhs.str()`.

```
template <class BidirectionalIterator>
bool operator < (typename iterator_traits<BidirectionalIterator>::value_type const&
                 const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs < rhs.str()`.

```
template <class BidirectionalIterator>
bool operator > (typename iterator_traits<BidirectionalIterator>::value_type const&
                 const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs > rhs.str()`.

```
template <class BidirectionalIterator>
bool operator >= (typename iterator_traits<BidirectionalIterator>::value_type const&
                  const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs >= rhs.str()`.

```
template <class BidirectionalIterator>
bool operator <= (typename iterator_traits<BidirectionalIterator>::value_type const&
                  const sub_match<BidirectionalIterator>& rhs);
```

Returns: `lhs <= rhs.str()`.

```
template <class BidirectionalIterator>
bool operator == (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const&
```

Returns: `lhs.str() == rhs`.

```
template <class BidirectionalIterator>
bool operator != (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const&
```

Returns: `lhs.str() != rhs`.

```
template <class BidirectionalIterator>
bool operator < (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const&
```

Returns: `lhs.str() < rhs`.

```
template <class BidirectionalIterator>
bool operator > (const sub_match<BidirectionalIterator>& lhs,
                 typename iterator_traits<BidirectionalIterator>::value_type const&
```

Returns: `lhs.str() > rhs`.

```
template <class BidirectionalIterator>
bool operator >= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const&
```

Returns: `lhs.str() >= rhs`.

```
template <class BidirectionalIterator>
bool operator <= (const sub_match<BidirectionalIterator>& lhs,
                  typename iterator_traits<BidirectionalIterator>::value_type const&
```

Returns: `lhs.str() <= rhs`.

```
template <class charT, class traits, class BidirectionalIterator>
basic_ostream<charT, traits>&
   operator << (basic_ostream<charT, traits>& os
                const sub_match<BidirectionalIterator>& m);
```

Returns: `(os << m.str())`.

7.9 Class template `match_results` [tr.re.results]

Class template `match_results` denotes a collection of character sequences representing the result of a regular expression match. Storage for the collection is allocated and freed as necessary by the member functions of class `match_results`.

The class template `match_results` conforms to the requirements of a Sequence, as specified in [lib.sequence.reqmts], except that only operations defined for const-qualified Sequences are supported.

```
template <class BidirectionalIterator,
          class Allocator = allocator<sub_match<BidirectionalIterator> >
class match_results
{
public:
   typedef sub_match<BidirectionalIterator>
           value_type;
   typedef const value_type&
           const_reference;
   typedef const_reference
           reference;
   typedef implementation-defined
           const_iterator;
   typedef const_iterator
           iterator;
   typedef typename iterator_traits<BidirectionalIterator>::difference_type
           difference_type;
   typedef typename Allocator::size_type
           size_type;
   typedef Allocator
           allocator_type;
   typedef typename iterator_traits<BidirectionalIterator>::value_type
           char_type;
   typedef basic_string<char_type>
           string_type;

   // construct/copy/destroy:
   explicit match_results(const Allocator& a = Allocator());
   match_results(const match_results& m);
   match_results& operator=(const match_results& m);
   ~match_results();

   // size:
   size_type size() const;
   size_type max_size() const;
   bool empty() const;
   // element access:
```

```
    difference_type length(int sub = 0) const;
    difference_type position(unsigned int sub = 0) const;
    string_type str(int sub = 0) const;
    const_reference operator[](int n) const;

    const_reference prefix() const;

    const_reference suffix() const;
    const_iterator begin() const;
    const_iterator end() const;
    // format:
    template <class OutputIterator>
    OutputIterator format(OutputIterator out,
                          const string_type& fmt,
                          match_flag_type flags = format_default) const;
    string_type format(const string_type& fmt,
                       match_flag_type flags = format_default) const;

    allocator_type get_allocator() const;
    void swap(match_results& that);
};
```

7.9.1 `match_results` constructors [tr.re.results.const]

In all match_results constructors, a copy of the Allocator argument is used for any memory allocation performed by the constructor or member functions during the lifetime of the object.

```
match_results(const Allocator& a = Allocator());
```

Effects: Constructs an object of class `match_results`. The postconditions of this function are indicated in Table 7.15

Table 7.15: `match_results(const Allocator&)` effects

Element	Value
`empty()`	`true`
`size()`	`0`
`str()`	`basic_string<charT>()`

```
match_results(const match_results& m);
```

Effects: Constructs an object of class `match_results`, as a copy of `m`.

```
match_results& operator=(const match_results& m);
```

Effects: Assigns `m` to `*this`. The postconditions of this function are indicated in Table 7.16

Table 7.16: `match_results` assignment operator effects

Element	Value
`empty()`	`m.empty()`
`size()`	`m.size()`
`str(n)`	`m.str(n)` for all integers `n < m.size`
`prefix()`	`m.prefix()`
`suffix()`	`m.suffix()`
`(*this)[n]`	`m[n]` for all integers `n < m.size`
`length(n)`	`m.length(n)` for all integers `n < m.size`
`position(n)`	`m.position(n)` for all integers `n < m.size`

7.9.2 `match_results` size [tr.re.results.size]

```
size_type size() const;
```

Returns: the number of `sub_match` elements stored in `*this`.

```
size_type max_size()const;
```

Returns: the maximum number of `sub_match` elements that can be stored in `*this`.

```
bool empty()const;
```

Returns: `size() == 0`.

7.9.3 `match_results` element access [tr.re.results.acc]

```
difference_type length(int sub = 0)const;
```

Returns: `(*this)[sub].length()`.

```
difference_type position(unsigned int sub = 0)const;
```

Returns: `std::distance(prefix().first, (*this)[sub].first)`.

```
string_type str(int sub = 0)const;
```

Returns: `string_type((*this)[sub])`.

```
const_reference operator[](int n) const;
```

Returns: A reference to the `sub_match` object representing the character sequence that matched marked sub-expression `n`. If `n == 0` then returns a reference to a `sub_match` object representing the character sequence that matched the whole regular expression.

```
const_reference prefix()const;
```

Returns: a reference to the `sub_match` object representing the character sequence from the start of the string being matched/searched, to the start of the match found.

```
const_reference suffix()const;
```

Returns: a reference to the `sub_match` object representing the character sequence from the end of the match found to the end of the string being matched/searched.

```
const_iterator begin()const;
```

Returns: a starting iterator that enumerates over all the marked sub-expression matches stored in `*this`.

```
const_iterator end()const;
```

Returns: a terminating iterator that enumerates over all the marked sub-expression matches stored in `*this`.
rSec2[tr.re.results.reform]`match_results` reformatting

```
OutputIterator format(OutputIterator out,
                      const string_type& fmt,
                      match_flag_type flags = format_default);
```

Requires: The type `OutputIterator` conforms to the Output Iterator requirements [24.1.2] .
Effects: Copies the character sequence `[fmt.begin(), fmt.end())` to OutputIterator `out`. For each format specifier or escape sequence in fmt , replace that sequence with either the character(s) it represents, or the sequence of characters within `*this` to which it refers. The bitmasks specified in `flags` determines what format specifiers or escape sequences are recognized, by default this is the format used by ECMA-262 [2], ECMAScript Language Specification, Chapter 15 part 5.4.11 String.prototype.replace.
Returns: `out`.

```
string_type format(const string_type& fmt,
                   match_flag_type flags = format_default);
```

Effects: Returns a copy of the string `fmt`. For each format specifier or escape sequence in `fmt`, replace that sequence with either the character(s) it represents, or the sequence of characters within `*this` to which it refers. The bitmasks specified in `flags` determines what format specifiers or escape sequences are recognized, by default this is the format used by ECMA-262 [2], ECMAScript Language Specification, Chapter 15 part 5.4.11 String.prototype.replace.
rSec2[tr.re.results.all]`match_results` allocator

```
allocator_type get_allocator() const;
```

Effects: Returns a copy of the Allocator that was passed to the object's constructor.
rSec2[tr.re.results.swap]`match_results` swap

```
void swap(match_results& that);
```

Effects: Swaps the contents of the two sequences.
Postcondition: `*this` contains the sequence of matched sub-expressions that were in `that`, `that` contains the sequence of matched sub-expressions that were in `*this`.
Complexity: constant time.

7.10 Regular expression algorithms [tr.re.alg]

7.10.1 `regex_match` [tr.re.alg.match]

```
template <class BidirectionalIterator, class Allocator, class charT,
          class traits, class Allocator2>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
                 match_results<BidirectionalIterator, Allocator>& m,
                 const reg_expression<charT, traits, Allocator2>& e,
                 match_flag_type flags = match_default);
```

Requires: Type `BidirectionalIterator` meets the requirements of a Bidirectional Iterator (§24.1.4).
Effects: Determines whether there is an exact match between the regular expression `e`, and all of the character sequence `[first, last)`, parameter `flags` is used to control how the expression is matched against the character sequence. Returns `true` if such a match exists, `false` otherwise.
Postconditions: If the function returns `false`, then the effect on parameter `m` is undefined, otherwise the effects on parameter mare given in table 7.17

Table 7.17: Effects of `regex_match` algorithm

Element	Value
`m.size()`	`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` if it was a partial match (found as a result of the `match_partial` flag being set).
`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, class Allocator2>
bool regex_match(BidirectionalIterator first, BidirectionalIterator last,
```

```
                const reg_expression<charT, traits, Allocator2>& e,
                match_flag_type flags = match_default);
```

Effects: Behaves "as if" by constructing an instance of `match_results< BidirectionalIterator > what` , and then returning the result of `regex_match(first, last, what, e, flags)`.

```
template <class charT, class Allocator, class traits, class Allocator2>
bool regex_match(const charT* str, match_results<const charT*, Allocator>& m,
                const reg_expression<charT, traits, Allocator2>& e,
                match_flag_type flags = match_default);
```

Returns: `regex_match(str, str + char_traits<charT>::length(str), m, e, flags)`.

```
template <class ST, class SA, class Allocator, class charT,
          class traits, class Allocator2>
bool regex_match(const basic_string<charT, ST, SA>& s,
                match_results<typename basic_string<charT, ST, SA>::const_iterator,
                const reg_expression<charT, traits, Allocator2>& e,
                match_flag_type flags = match_default);
```

Returns: `regex_match(s.begin(), s.end(), m, e, flags)`.

```
template <class charT, class traits, class Allocator2>
bool regex_match(const charT* str,
                const reg_expression<charT, traits, Allocator2>& e,
                match_flag_type flags = match_default);
```

Returns: `regex_match(str, str + char_traits<charT>::length(str), e, flags)`

```
template <class ST, class SA, class charT, class traits, class Allocator2>
bool regex_match(const basic_string<charT, ST, SA>& s,
                const reg_expression<charT, traits, Allocator2>& e,
                match_flag_type flags = match_default);
```

Returns: `regex_match(s.begin(), s.end(), e, flags)`.

7.10.2 `regex_search` [tr.re.alg.search]

```
template <class BidirectionalIterator, class Allocator, class charT,
          class traits, class Allocator2>
bool regex_search(BidirectionalIterator first, BidirectionalIterator last,
                 match_results<BidirectionalIterator, Allocator>& m,
                 const reg_expression<charT, traits, Allocator2>& e,
                 match_flag_type flags = match_default);
```

Requires: Type `BidirectionalIterator` meets the requirements of a Bidirectional Iterator (24.1.4).

Effects: Determines whether there is some sub-sequence within `[first,last)` that matches the regular expression `e`, parameter `flags` is used to control how the expression is matched against the character sequence. Returns `true` if such a sequence exists, `false` otherwise.

Postconditions: If the function returns `false`, then the effect on parameter `m` is undefined, otherwise the effects on parameter `m` are given in table 7.18.

Table 7.18: Effects of `regex_search` algorithm

Element	Value
`m.size()`	`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 full match was found, and `false` if it was a partial match (found as a result of the `match_partial` flag being set).
`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 nparticipated in the match, `false` otherwise.

```
template <class charT, class Allocator, class traits, class Allocator2>
bool regex_search(const charT* str, match_results<const charT*, Allocator>& m,
                  const reg_expression<charT, traits, Allocator2>& e,
                  match_flag_type flags = match_default);
```

Returns: the result of `regex_search(str, str + char_traits<charT>::length(str), m, e, flags)`.

```
template <class ST, class SA, class Allocator, class charT,
          class traits, class Allocator2>
bool regex_search(const basic_string<charT, ST, SA>& s,
                  match_results<typename basic_string<charT, ST, SA>::const_iterator
                  const reg_expression<charT, traits, Allocator2>& e,
                  match_flag_type flags = match_default);
```

Returns: the result of `regex_search(s.begin(), s.end(), m, e, flags)`.

```
template <class iterator, class Allocator, class charT,
          class traits>
bool regex_search(iterator first, iterator last,
                  const reg_expression<charT, traits, Allocator>& e,
                  match_flag_type flags = match_default);
```

Effects: Behaves "as if" by constructing an instance of `match_results< BidirectionalIterator > what`, and then returning the result of `regex_search(first, last, what, e, flags)`.

```
template <class charT, class Allocator, class traits>
bool regex_search(const charT* str
                  const reg_expression<charT, traits, Allocator>& e,
                  match_flag_type flags = match_default);
```

Returns: `regex_search(str, str + char_traits<charT>::length(str), e, flags)`

```
template <class ST, class SA, class Allocator, class charT,
          class traits>
bool regex_search(const basic_string<charT, ST, SA>& s,
                  const reg_expression<charT, traits, Allocator>& e,
                  match_flag_type flags = match_default);
```

Returns: `regex_search(s.begin(), s.end(), e, flags)`.

7.10.3 `regex_replace` [tr.re.alg.replace]

```
template <class OutputIterator, class BidirectionalIterator, class traits,
          class Allocator, class charT>
OutputIterator regex_replace(OutputIterator out,
                             BidirectionalIterator first,
                             BidirectionalIterator last,
                             const reg_expression<charT, traits, Allocator>& e,
                             const basic_string<charT>& fmt,
                             match_flag_type flags = match_default);
```

Effects: Finds all the non-overlapping matches `m` of type `match_results<BidirectionalIterator>` that occur within the sequence `[first, last)`. If no such matches are found and `!(flags & format_no_copy)` then calls `std::copy(first, last, out)`. Otherwise, for each match found, if `!(flags & format_no_copy)` calls `std::copy(m.prefix().first, m.prefix().last, out)`, and then calls `m.format(out, fmt, flags)`. Finally if `!(flags & format_no_copy)` calls `std::copy(last_m.suffix().first, last_m,suffix().last, out)` where `last_m` is a copy of the last match found. If `flags & format_first_only` is non-zero then only the first match found is replaced.
Returns: `out`.

```
template <class traits, class Allocator, class charT>
basic_string<charT> regex_replace(const basic_string<charT>& s,
                                  const reg_expression<charT, traits, Allocator>& e,
                                  const basic_string<charT>& fmt,
```

```
                                match_flag_type flags = match_default);
```

Effects: Constructs an object `basic_string<charT> result`, calls `regex_replace(back_inserter(result), s.begin(), s.end(), e, fmt, flags)`, and then returns `result`.

7.11 Regular expression Iterators [tr.re.iter]

7.11.1 Class template `regex_iterator` [tr.re.regiter]

The class template `regex_iterator` is an iterator adapter; that is to say it represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence. regex_iterator finds (using `regex_search`) successive regular expression matches within the sequence from which it was constructed. After it is constructed, and every time `operator++` is used, the iterator finds and stores a value of `match_results<BidirectionalIterator>`. If the end of the sequence is reached (`regex_search` returns `false`), the iterator becomes equal to the end-of-sequence iterator value. The constructor with no arguments `regex_iterator()` always constructs an end of sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of `operator*` on an end of sequence is not defined. For any other iterator value a const `match_results<BidirectionalIterator>&` is returned. The result of `operator->` on an end of sequence is not defined. For any other iterator value a `const match_results<BidirectionalIterator>*` is returned. It is impossible to store things into `regex_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 = iterator_traits<BidirectionalIterator>::value_type,
          class traits = regex_traits<charT>,
          class Allocator = allocator<charT> >
class regex_iterator
{
public:
   typedef basic_regex<charT, traits, Allocator>
           regex_type;
   typedef match_results<BidirectionalIterator>
           value_type;
   typedef typename iterator_traits<BidirectionalIterator>::difference_type
           difference_type;
   typedef typename iterator_traits<BidirectionalIterator>::pointer
           pointer;
   typedef typename iterator_traits<BidirectionalIterator>::reference
           reference;
   typedef std::forward_iterator_tag
           iterator_category;

   regex_iterator();
   regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
                  const regex_type& re,
```

```
                match_flag_type m = match_default);
   regex_iterator(const regex_iterator&);
   regex_iterator& operator=(const regex_iterator&);
   bool operator==(const regex_iterator&);
   bool operator!=(const regex_iterator&);
   const value_type& operator*();
   const value_type* operator->();
   regex_iterator& operator++();
   regex_iterator operator++(int);
private:
   match_results<BidirectionalIterator> what;  // exposition only
   BidirectionalIterator                end;   // exposition only
   const regex_type*                    pre;   // exposition only
   match_flag_type                      flags; // exposition only
};
```

7.11.1.1 `regex_iterator` constructors [tr.re.regiter.cnstr]

```
regex_iterator();
```

Effects: default constructs the members `what`, `end` and `flags`, and sets the member `pre` equal to the null-pointer constant.

```
regex_iterator(BidirectionalIterator a, BidirectionalIterator b,
               const regex_type& re,
               match_flag_type m = match_default);
```

Effects: default constructs all data members, and then calls `regex_search(a, b, what, re, m)`. If this returns `false` then sets `*this` equal to the end of sequence iterator, otherwise sets `end` equal to `b`, `pre` equal to `&re`, and `flags` equal to `m`.

```
regex_iterator(const regex_iterator& that);
```

Effects: constructs all data members as a copy of `that`.
Postconditions: `*this == that`.

```
regex_iterator& operator=(const regex_iterator& that);
```

Effects: sets all data members equal to those in `that`.
Postconditions: `*this == that`.

7.11.1.2 `regex_iterator` comparisons [tr.re.regiter.comp]

```
bool operator==(const regex_iterator& that);
```

Effects: if `(pre == 0) && (that.pre == 0)` then returns `true`, otherwise returns the result of `(pre == that.pre) && (end == that.end) && (flags == that.flags) && (what[0].first == that.what[0].first) && (what[0].second == that.what[0].second)`.

```
bool operator!=(const regex_iterator&);
```

Effects: returns `!(*this == that)`.

7.11.1.3 `regex_iterator` dereference [tr.re.regiter.deref]

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

Returns: what.

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

Returns: &what.

7.11.1.4 `regex_iterator` increment [tr.re.regiter.incr]

```
regex_iterator& operator++();
```

Effects: if what.prefix().first != what[0].second and if the element match_prev_avail is not set in flags then sets it. Then calls regex_search(what[0].second, end, what, *pre, ((what[0].first == what[0].second) ? flags match_not_null : flags))—, and if this returns false then sets *this equal to the end of sequence iterator.
Returns: *this.

```
regex_iterator operator++(int);
```

Effects: constructs a copy result of *this, then calls ++(*this).
Returns: result.

7.11.2 Class template `regex_token_iterator` [tr.re.tokiter]

The class template regex_token_iterator is an iterator adapter; that is to say it represents a new view of an existing iterator sequence, by enumerating all the occurrences of a regular expression within that sequence, and presenting one or more new strings for each match found. Each position enumerated by the iterator is a string that represents what matched a particular sub-expression within the regular expression. When class regex_token_iterator is used to enumerate a single sub-expression with index -1, then the iterator performs field splitting: that is to say it enumerates one string for each section of the character container sequence that does not match the regular expression specified.
After it is constructed, the iterator finds and stores a value of match_results<BidirectionalIterator> what (by calling regex_search) and sets the internal count N to zero. Every time operator++ is used the count N is incremented; if N exceeds or equals this->subs.size(), then the iterator finds and stores the next value of match_results<BidirectionalIterator> and sets count N to zero.
If the end of sequence is reached (regex_search returns false), the iterator becomes equal to the *end-of-sequence* iterator value, unless the sub-expression being enumerated has index -1: In which case the iterator enumerates one last string that contains all the characters from the end of the last regular expression match to the end of the input sequence being enumerated, provided that this would not be an empty string.
The constructor with no arguments, regex_iterator(), always constructs an end of sequence iterator object, which is the only legitimate iterator to be used for the end condition. The result of operator* on an end of sequence is not defined. For any other iterator value a const basic_string<charT>& is returned (obtained by calling this->what[subs[N]]). The result of operator-> on an end of sequence is not defined. For any other iterator value a const basic_string<charT>* is returned. It is impossible to store things into regex_iterators. Two end-of-sequence iterators are always equal. An end-of-sequence iterator is not equal to a non-end-of-sequence iterator. Two non-end-of-sequence iterators are equal when they are constructed from the same arguments.

```
template <class BidirectionalIterator,
          class charT = iterator_traits<BidirectionalIterator>::value_type,
          class traits = regex_traits<charT>,
          class Allocator = allocator<charT> >
class regex_token_iterator
{
public:
   typedef basic_regex<charT, traits, Allocator>
           regex_type;
   typedef basic_string<charT>
           value_type;
   typedef typename iterator_traits<BidirectionalIterator>::difference_type
           difference_type;
   typedef typename iterator_traits<BidirectionalIterator>::pointer
           pointer;
   typedef typename iterator_traits<BidirectionalIterator>::reference
           reference;
   typedef std::forward_iterator_tag
           iterator_category;

   regex_token_iterator();
   regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const rege
                        int submatch = 0, match_flag_type m = match_default);
   regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const rege
                        const std::vector<int>& submatches, match_flag_type m = matc
   template <std::size_t N>
   regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b, const rege
                        const int (&submatches)[N], match_flag_type m = match_defau
   regex_token_iterator(const regex_token_iterator&);
   regex_token_iterator& operator=(const regex_token_iterator&);
   bool operator==(const regex_token_iterator&);
   bool operator!=(const regex_token_iterator&);
   const value_type& operator*();
   const value_type* operator->();
   regex_token_iterator& operator++();
   regex_token_iterator operator++(int);
private:
   match_results<BidirectionalIterator> what;   // exposition only
   BidirectionalIterator                end;    // exposition only
   const regex_type*                    pre;    // exposition only
   match_flag_type                      flags;  // exposition only
   basic_string<charT>                  result; // exposition only
   std::size_t                          N;      // exposition only
   std::vector<int>                     subs;   // exposition only
};
```

7.11.2.1 `regex_token_iterator` constructors [tr.re.tokiter.cnstr]

```
regex_token_iterator();
```

Effects: constructs an end of sequence iterator, by default constructing all data members.
Postconditions: `pre == 0`.

```
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                     const regex_type& re,
                     int submatch = 0, match_flag_type m = match_default);
```

Preconditions: `!re.empty()`.
Effects: default constructs all data members and pushes the value `submatch` onto `subs`. Then if `regex_search(a, b, what, re, m) == true` sets `N` equal to zero, `flags` equal to `m`, `pre` equal to `&re`, `end` equal to `b`, and sets `result` equal to `((submatch == -1) ? value_type(what.prefix().str()) : value_type(what[submatch].str()))`. Otherwise if the call to `regex_search` returns `false`, then if `(submatch == -1) && (a != b)` sets `result` equal to `value_type(a, b)` and `N` equal to -1. Otherwise sets `*this` equal to the end of sequence iterator.

```
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                     const regex_type& re,
                     const std::vector<int>& submatches,
                     match_flag_type m = match_default);
```

Preconditions: `submatches.size() && !re.empty()`.
Effects: copy constructs member `subs` from `submatches`, and default constructs the other data members. Then if `regex_search(a, b, what, re, m) == true` sets `N` equal to zero, `flags` equal to `m`, `pre` equal to `&re`, `end` equal to `b`, and sets `result` equal to `((submatch[N] == -1) ? value_type(what.prefix().str()) : value_type(what[submatch[N]].str()))`. Otherwise if the call to `regex_search` returns `false`, then if `(submatch[0] == -1) && (a != b)` sets `result` equal to `value_type(a, b)` and `N` equal to -1. Otherwise sets `*this` equal to the end of sequence iterator.

```
template <std::size_t N>
regex_token_iterator(BidirectionalIterator a, BidirectionalIterator b,
                     const regex_type& re,
                     const int (&submatches)[R],
                     match_flag_type m = match_default);
```

Preconditions: `!re.empty()`.
Effects: copy constructs member `subs` from the iterator sequence `[submatches, submatches+R)`, and default constructs the other data members. Then if `regex_search(a, b, what, re, m) == true` sets `N` equal to zero, `flags` equal to `m`, `pre` equal to `&re`, `end` equal to `b`, and sets `result` equal to `((subs[N] == -1) ? value_type(what.prefix().str()) : value_type(what[subs[N]].str()))`. Otherwise if the call to `regex_search` returns `false`, then if `(subs[0] == -1) && (a != b)` sets `result` equal to `value_type(a, b)` and `N` equal to -1. Otherwise sets `*this` equal to the end of sequence iterator.

```
regex_token_iterator(const regex_token_iterator& that);
```

Effects: constructs all data members as a copy of `that`.
Postconditions: `*this == that`.

```
regex_token_iterator& operator=(const regex_token_iterator& that);
```

Effects: sets all data members equal to those in `that`.
Postconditions: `*this == that`.

7.11.2.2 `regex_token_iterator` comparisons [tr.re.tokiter.comp]

```
bool operator==(const regex_token_iterator&);
```

Effects: if `(pre == 0) && (that.pre == 0)` then returns `true`, otherwise returns the result of `(pre == that.pre) && (end == that.end) && (flags == that.flags) && (N == that.N) && (what[0].first == that.what[0].first) && (what[0].second == that.what[0].second)`.

```
bool operator!=(const regex_token_iterator&);
```

Returns: `!(*this == that)`.

7.11.2.3 `regex_token_iterator` deference [tr.re.tokiter.deref]

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

Effects: returns `result`.

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

Effects: returns `&result`.

7.11.2.4 `regex_token_iterator` increment [tr.re.tokiter.incr]

```
regex_token_iterator& operator++();
```

Effects: if `N == -1` then sets `*this` equal to the end of sequence iterator. Otherwise if `N+1 < subs.size()`, then increments `N` and sets `result` equal to `((subs[N] == -1) ? value_type(what.prefix().str()) : value_type(what[subs[N]].str()))`. Otherwise if `what.prefix().first != what[0].second` and if the element `match_prev_avail` is not set in `flags` then sets it. Then if `regex_search(what[0].second, end, what, *pre, ((what[0].first == what[0].second) ? flags` match_not_null : flags)) == true— sets `N` equal to zero, and sets `result` equal to `((subs[N] == -1) ? value_type(what.prefix().str()) : value_type(what[subs[N]].str()))`. Otherwise if the call to `regex_search` returns `false`, then let `last_end` be the value of `what[0].second` prior to the call to `regex_search`. Then if `last_end != end` and `subs[0] == -1` sets `N` equal to -1 and sets `result` equal to `value_type(last_end, end)`. Otherwise sets `*this` equal to the end of sequence iterator.
Returns: `*this`.

```
regex_token_iterator& operator++(int);
```

Effects: constructs a copy `result` of `*this`, then calls `++(*this)`.
Returns: `result`.

Bibliography

[1] Abramowitz, Milton, 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] Ecma International, *ECMAScript Language Specification*, Standard Ecma-262, third edition, 1999.

[3] IEEE, *Information Technology—Portable Operating System Interface (POSIX)*, IEEE Standard 1003.1-2001.

[4] International Standards Organization: *Quantities and units, Third edition*. International Standard ISO 31-11:1992. ISBN 92-67-10185-4.

[5] International Standards Organization: *Programming Languages – C, Second edition*. International Standard ISO/IEC 9899:1999.

[6] International Standards Organization: *Programming Languages – C++*. International Standard ISO/IEC 14882:1998.