Explicitly Implicifying `explicit` Constructors

Document number: P1163R0
Date: 2018-08-31
Project: Programming Language C++, Library Working Group
Reply-to: Nevin “☺” Liber, nevin@cplusplusguy.com *or* nliber@ocient.com

Table of Contents

Revision History 3
P11630R0 3
Introduction 4
Motivation and Scope 5
Impact On the Standard 6
Policy 7
Design Decisions 8
Technical Specifications 9
[template.bitset] 10
[bitset.cons] 11
[basic.string] 13
[string.cons] 15
[deque.overview] 16
[deque.cons] 17
[forwardlist.overview] 18
[forwardlist.cons] 19
[list.overview] 20
[list.cons] 21
[vector.overview] 22
[vector.cons] 23
[map.overview] 24
[map.cons] 25
[multimap.overview] 26
[multimap.cons] 27
[set.overview] 28
[set.cons] 29
[multiset.overview] 30
[multiset.cons] 31
[unord.map.overview] 32
[unord.map.cnstr] 34
[unord.multimap.overview] 35

[unord.multimap.cnstr] **37**
[unord.set.overview] **38**
[unord.set.cnstr] **40**
[unord.multiset.overview] **41**
[unord.multiset.cnstr] **43**
[rand.dist.uni.int] **44**
[rand.dist.uni.real] **45**
[rand.dist.bern.bin] **46**
[rand.dist.bern.negbin] **47**
[rand.dist.pois.gamma] **48**
[rand.dist.pois.weibull] **49**
[rand.dist.pois.extreme] **50**
[rand.dist.norm.normal] **51**
[rand.dist.norm.lognormal] **52**
[rand.dist.norm.cauchy] **53**
[rand.dist.norm.f] **54**
[ios::failure] **55**
[istream::sentry] **56**
[stringbuf] **58**
[stringbuf.cons] **59**
[istringstream] **60**
[istringstream.cons] **61**
[ostringstream] **62**
[ostringstream.cons] **63**
[stringstream] **64**
[stringstream.cons] **65**
[ifstream] **66**
[ifstream.cons] **67**
[ofstream] **68**
[ofstream.cons] **69**
[fstream] **70**
[fstream.cons] **72**
[syncstream.syncbuf.overview] **73**
[fs.class.file_status] **74**
[fs.file_status.cons] **75**
[re.regex] **76**
[re.regex.construct] **77**

Acknowledgements **78**

References **79**

Revision History

P1163OR0

First published, applying all the changes to the draft discussed at the Summer 2018 WG21 Batavia LWG Meeting.

Introduction

This paper is intended to address LWG2307 - Should the Standard Library use explicit only when necessary?

The answer is “yes”. ☺ This paper expands on and applies what is meant by “only when necessary”.

Motivation and Scope

Most `explicit` constructors of more than one parameter which don't take a variadic number of parameters in the library appear to arise out of historical accident rather than intentional design: they are constructors taking two or more parameters, all but the first of which have default arguments, and so in C++98 `explicit` was added to prevent defining an undesirable implicit conversion from the type of the first parameter. At that time, `explicit` simply had no effect in the other cases.

Along came C++11 and list initialization syntax, and `explicit` applies to all parameters when using that syntax. This lead to the inconsistency described in LWG2307, because `explicit` constructors for classes in the standard library where all but the first parameter are defaulted behave differently than other non-variadic non-`explicit` constructors taking multiple parameters in the same class.

This paper is intended as an exhaustive search though n4762 and addressing those constructors, with the follow exceptions:

- Two places where they are not accidental are in [pair] and [tuple], and the author has not looked at any of the constructors in those sections.

- The author also has not modified any classes that are not publicly destructible.

These changes do not affect overload resolution and is more consistent with what we have now.

Impact On the Standard

This splits most `explicit` constructors of more than one parameter where all but the first are defaulted into two constructors: a non-`explicit` constructor taking two or more parameters where all but the first two parameters are defaulted, and an `explicit` constructor taking exactly one parameter which delegates to the non-`explicit` constructor mentioned above. New deduction guides have been added when necessary to retain compatibility.

Note: In [basic.string] the explicit constructor is not delegating to the non-explicit constructor, for ease of specification.

Two other related issues were corrected as a drive-by:

- In [rand.dist.pois.extreme], the second parameter passed to the delegated constructor is 1.0 instead of 1.

- In [syncstream.syncbuf.overview], the default constructor was made non-explicit to reflect the other default constructors corrected by P0935 (basic_syncbuf did not exist when P0935 was applied).

Policy

After applying this paper, the policy for non-variadic constructors not taking tag types is:

- Single parameter constructors are `explicit` and take no defaulted parameters.

- Non-single parameter constructors are not `explicit`.

As always, this policy can be overridden with justification, but without such justification, LWG expects any proposals to follow this policy and will consider it an unintentional error they have the authority to correct.

Design Decisions

Delegating constructors are used to minimize the risk of unintentional breakage.

Note: the explicit single argument constructors delegate by passing exactly two arguments, which in turn may delegate or default other arguments. Another way to do so would be to specify all the rest of the parameters so that we only delegate once, but that was considered more error-prone for both this paper and future edits.

Technical Specifications

The changes are relative to n4762, where ~~red strikethrough~~ indicates removal and green underline indicates additions. Text in black or *blue* is for context and can be ignored as far as applying the edits are concerned.

Editors note: you have complete discretion on applying formatting to the new delegating constructors. If you think of a better way than has been presented in the paper, feel free to apply it.

[template.bitset]

```
namespace std {
  template<size_t N> class bitset {
  public:
    // ...
    // [bitset.cons], constructors
    constexpr bitset() noexcept;
    constexpr bitset(unsigned long long val) noexcept;
    template<class charT, class traits, class Allocator>
      explicit bitset(
        const basic_string<charT, traits, Allocator>& str,
        typename basic_string<charT, traits, Allocator>::size_type
pos = 0,
        typename basic_string<charT, traits, Allocator>::size_type n
          = basic_string<charT, traits, Allocator>::npos,
        charT zero = charT('0'),
        charT one = charT('1'));
    template<class charT, class traits, class Allocator>
      explicit bitset(
        const basic_string<charT, traits, Allocator>& str) :
bitset(str, 0) { }
    template<class charT>
      explicit bitset(
        const charT* str,
        typename basic_string<charT>::size_type n =
basic_string<charT>::npos,
        charT zero = charT('0'),
        charT one = charT('1'));
    template<class charT>
      explicit bitset(
        const charT* str) : bitset(str, basic_string<charT>::npos) {
}

    // ...
  };
}
```

[bitset.cons]

```
template<class charT, class traits, class Allocator>
  explicit bitset(
    const basic_string<charT, traits, Allocator>& str,
    typename basic_string<charT, traits, Allocator>::size_type pos = 0,
    typename basic_string<charT, traits, Allocator>::size_type n
      = basic_string<charT, traits, Allocator>::npos,
    charT zero = charT('0'),
    charT one = charT('1'));
```

Throws: `out_of_range` if `pos > str.size()` or `invalid_argument` if an invalid character is found (see below).

Effects: Determines the effective length `rlen` of the initializing string as the smaller of `n` and `str.size() - pos`.
The function then throws `invalid_argument` if any of the `rlen` characters in `str` beginning at position `pos` is other than `zero` or `one`.

The function uses `traits::eq()` to compare the character values.
Otherwise, the function constructs an object of class `bitset<N>`, initializing the first `M` bit positions to values determined from the corresponding characters in the string `str`.

`M` is the smaller of `N` and `rlen`.

An element of the constructed object has value zero if the corresponding character in `str`, beginning at position `pos`, is `zero`.

Otherwise, the element has the value one.

Character position `pos + M - 1` corresponds to bit position zero.

Subsequent decreasing character positions correspond to increasing bit positions.

If `M < N`, remaining bit positions are initialized to zero.

```
template<class charT>
  explicit bitset(
    const charT* str,
    typename basic_string<charT>::size_type n =
basic_string<charT>::npos,
    charT zero = charT('0'),
    charT one = charT('1'));
```

Effects: Constructs an object of class `bitset<N>` as if by:

```
  bitset(n == basic_string<charT>::npos
            ? basic_string<charT>(str)
```

```
   : basic_string<charT>(str, n),
0, n, zero, one)
```

[basic.string]

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_string {
  public:
    // ...
    // [string.cons], construct/copy/destroy
    basic_string() noexcept(noexcept(Allocator())) :
basic_string(Allocator()) { }
    explicit basic_string(const Allocator& a) noexcept;
    basic_string(const basic_string& str);
    basic_string(basic_string&& str) noexcept;
    basic_string(const basic_string& str, size_type pos, const
Allocator& a = Allocator());
    basic_string(const basic_string& str, size_type pos, size_type n,
                 const Allocator& a = Allocator());
    template<class T>
      basic_string(const T& t, size_type pos, size_type n, const
Allocator& a = Allocator());
    template<class T>
      explicit basic_string(const T& t, const Allocator& a = =
Allocator());
    template<class T>
      explicit basic_string(const T& t);
    basic_string(const charT* s, size_type n, const Allocator& a =
Allocator());
    basic_string(const charT* s, const Allocator& a = Allocator());
    basic_string(size_type n, charT c, const Allocator& a =
Allocator());
    template<class InputIterator>
      basic_string(InputIterator begin, InputIterator end, const
Allocator& a = Allocator());
    basic_string(initializer_list<charT>, const Allocator& =
Allocator());
    basic_string(const basic_string&, const Allocator&);
    basic_string(basic_string&&, const Allocator&);

    ~basic_string();
    basic_string& operator=(const basic_string& str);
    basic_string& operator=(basic_string&& str)

noexcept(allocator_traits<Allocator>::propagate_on_container_move_ass
ignment::value ||
               allocator_traits<Allocator>::is_always_equal::value);
    template<class T>
      basic_string& operator=(const T& t);
    basic_string& operator=(const charT* s);
    basic_string& operator=(charT c);
    basic_string& operator=(initializer_list<charT>);
```

```
    // ...
  };

  template<class InputIterator,
           class Allocator = allocator<typename
iterator_traits<InputIterator>::value_type>>
    basic_string(InputIterator, InputIterator, Allocator =
Allocator())
      -> basic_string<typename
iterator_traits<InputIterator>::value_type,
                      char_traits<typename
iterator_traits<InputIterator>::value_type>,
                      Allocator>;

  template<class charT,
           class traits,
           class Allocator = allocator<charT>>
    explicit basic_string(basic_string_view<charT, traits>, const
Allocator& = Allocator())
      -> basic_string<charT, traits, Allocator>;
```

[[[Reviewers note: the deduction guide below only has two template parameters, because an allocator isn't passed to it so there is no Allocator type to deduce.]]]

```
  template<class charT,
           class traits>
    explicit basic_string(basic_string_view<charT, traits>)
      -> basic_string<charT, traits>;

  template<class charT,
           class traits,
           class Allocator = allocator<charT>>
    basic_string(basic_string_view<charT, traits>,
                 typename see below::size_type, typename see
below::size_type,
                 const Allocator& = Allocator())
      -> basic_string<charT, traits, Allocator>;
}
```

[string.cons]

[[[Note for reviewers: The Effects and Remarks below will be superceded by the changes in D1148R0a discussed in LWG Batavia 2018, which is why the explicit constructors are added here.]]]

```
template<class T>
  explicit basic_string(const T& t, const Allocator& a = Allocator());
template<class T>
  explicit basic_string(const T& t);
```

Effects: For the second overload, let `a` be `Allocator()`. Creates a variable, `sv`, as if by `basic_string_view<charT, traits> sv = t;` and then behaves the same as `basic_string(sv.data(), sv.size(), a)`.
Remarks: This constructor shall not participate in overload resolution unless

- `is_convertible_v<const T&, basic_string_view<charT, traits>>` is `true` and
- `is_convertible_v<const T&, const charT*>` is `false`.

[[[…]]]

```
template<class charT,
         class traits,
         class Allocator = allocator<charT>>
  explicit basic_string(basic_string_view<charT, traits>, const
Allocator& = Allocator())
    -> basic_string<charT, traits, Allocator>;

template<class charT,
         class traits,
         class Allocator = allocator<charT>>
  basic_string(basic_string_view<charT, traits>,
               typename see below::size_type, typename see
below::size_type,
               const Allocator& = Allocator())
    -> basic_string<charT, traits, Allocator>;
```

Remarks: Shall not participate in overload resolution if `Allocator` is a type that does not qualify as an allocator ([container.requirements.general]).

[deque.overview]

```
namespace std {
  template<class T, class Allocator = allocator<T>>
  class deque {
  public:
    // ...
    // [deque.cons], construct/copy/destroy
    deque() : deque(Allocator()) { }
    explicit deque(const Allocator&);
    explicit deque(size_type n, const Allocator& = Allocator());
    explicit deque(size_type n) : deque(n, Allocator()) { }
    deque(size_type n, const T& value, const Allocator& = Allocator());
    template<class InputIterator>
      deque(InputIterator first, InputIterator last, const Allocator&
= Allocator());
    deque(const deque& x);
    deque(deque&&);
    deque(const deque&, const Allocator&);
    deque(deque&&, const Allocator&);
    deque(initializer_list<T>, const Allocator& = Allocator());

    ~deque();
    deque& operator=(const deque& x);
    deque& operator=(deque&& x)
      noexcept(allocator_traits<Allocator>::is_always_equal::value);
    deque& operator=(initializer_list<T>);
    template<class InputIterator>
      void assign(InputIterator first, InputIterator last);
    void assign(size_type n, const T& t);
    void assign(initializer_list<T>);
    allocator_type get_allocator() const noexcept;

    // iterators
    iterator               begin() noexcept;
    const_iterator         begin() const noexcept;
    iterator               end() noexcept;
    const_iterator         end() const noexcept;
    reverse_iterator       rbegin() noexcept;
    const_reverse_iterator rbegin() const noexcept;
    reverse_iterator       rend() noexcept;
    const_reverse_iterator rend() const noexcept;

    const_iterator         cbegin() const noexcept;
    const_iterator         cend() const noexcept;
    const_reverse_iterator crbegin() const noexcept;
    const_reverse_iterator crend() const noexcept;

    // ...
  };
}
```

[deque.cons]

~~explicit~~ `deque(size_type n, const Allocator&` ~~`= Allocator()`~~`);`

Effects: Constructs a `deque` with `n` default-inserted elements using the specified allocator.
Requires: `T` shall be *Cpp17DefaultInsertable* into `*this`.
Complexity: Linear in `n`.

[forwardlist.overview]

```
namespace std {
  template<class T, class Allocator = allocator<T>>
  class forward_list {
  public:
    // ...
    // [forwardlist.cons], construct/copy/destroy
    forward_list() : forward_list(Allocator()) { }
    explicit forward_list(const Allocator&);
    explicit forward_list(size_type n, const Allocator& = Allocator());
    explicit forward_list(size_type n) : forward_list(n, Allocator())
{ }
    forward_list(size_type n, const T& value, const Allocator& =
Allocator());
    template<class InputIterator>
      forward_list(InputIterator first, InputIterator last, const
Allocator& = Allocator());
    forward_list(const forward_list& x);
    forward_list(forward_list&& x);
    forward_list(const forward_list& x, const Allocator&);
    forward_list(forward_list&& x, const Allocator&);
    forward_list(initializer_list<T>, const Allocator& = Allocator());
    ~forward_list();
    forward_list& operator=(const forward_list& x);
    forward_list& operator=(forward_list&& x)
      noexcept(allocator_traits<Allocator>::is_always_equal::value);
    forward_list& operator=(initializer_list<T>);
    template<class InputIterator>
      void assign(InputIterator first, InputIterator last);
    void assign(size_type n, const T& t);
    void assign(initializer_list<T>);
    allocator_type get_allocator() const noexcept;

    // ...
}
```

[forwardlist.cons]

~~explicit~~ `forward_list(size_type n, const Allocator&` ~~`= Allocator()`~~`);`

Effects: Constructs a `forward_list` object with `n` default-inserted elements using the specified allocator.
Requires: `T` shall be *Cpp17DefaultInsertable* into `*this`.
Complexity: Linear in `n`.

[list.overview]

```
namespace std {
  template<class T, class Allocator = allocator<T>>
  class list {
  public:
    // ...
    // [list.cons], construct/copy/destroy
    list() : list(Allocator()) { }
    explicit list(const Allocator&);
    explicit list(size_type n, const Allocator& = Allocator());
    explicit list(size_type n) : list(n, Allocator()) { }
    list(size_type n, const T& value, const Allocator& = Allocator());
    template<class InputIterator>
      list(InputIterator first, InputIterator last, const Allocator& =
Allocator());
    list(const list& x);
    list(list&& x);
    list(const list&, const Allocator&);
    list(list&&, const Allocator&);
    list(initializer_list<T>, const Allocator& = Allocator());
    ~list();
    list& operator=(const list& x);
    list& operator=(list&& x)
      noexcept(allocator_traits<Allocator>::is_always_equal::value);
    list& operator=(initializer_list<T>);
    template<class InputIterator>
      void assign(InputIterator first, InputIterator last);
    void assign(size_type n, const T& t);
    void assign(initializer_list<T>);
    allocator_type get_allocator() const noexcept;

    // ...
  };
}
```

[list.cons]

~~explicit~~ `list(size_type n, const Allocator&` ~~`= Allocator()`~~`);`

Effects: Constructs a `list` with `n` default-inserted elements using the specified allocator.
Requires: `T` shall be *Cpp17DefaultInsertable* into `*this`.
Complexity: Linear in `n`.

[vector.overview]

```
namespace std {
  template<class T, class Allocator = allocator<T>>
  class vector {
  public:
    // ...
    // [vector.cons], construct/copy/destroy
    vector() noexcept(noexcept(Allocator())) : vector(Allocator()) { }
    explicit vector(const Allocator&) noexcept;
    explicit vector(size_type n, const Allocator& = Allocator());
    explicit vector(size_type n) : vector(n, Allocator()) { }
    vector(size_type n, const T& value, const Allocator& =
Allocator());
    template<class InputIterator>
      vector(InputIterator first, InputIterator last, const Allocator&
= Allocator());
    vector(const vector& x);
    vector(vector&&) noexcept;
    vector(const vector&, const Allocator&);
    vector(vector&&, const Allocator&);
    vector(initializer_list<T>, const Allocator& = Allocator());
    ~vector();
    vector& operator=(const vector& x);
    vector& operator=(vector&& x)

noexcept(allocator_traits<Allocator>::propagate_on_container_move_ass
ignment::value ||
                allocator_traits<Allocator>::is_always_equal::value);
    vector& operator=(initializer_list<T>);
    template<class InputIterator>
      void assign(InputIterator first, InputIterator last);
    void assign(size_type n, const T& u);
    void assign(initializer_list<T>);
    allocator_type get_allocator() const noexcept;

    // ...
  };
}
```

[vector.cons]

~~explicit~~ `vector(size_type n, const Allocator&` ~~`= Allocator()`~~`);`

Effects: Constructs a `vector` with `n` default-inserted elements using the specified allocator.

Requires: `T` shall be *Cpp17DefaultInsertable* into `*this`.

Complexity: Linear in `n`.

[map.overview]

```
namespace std {
  template<class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T>>>
  class map {
  public:
    // …
    // [map.cons], construct/copy/destroy
    map() : map(Compare()) { }
    explicit map(const Compare& comp, const Allocator& = Allocator());
    explicit map(const Compare& comp) : map(comp, Allocator()) { }
    template<class InputIterator>
      map(InputIterator first, InputIterator last,
          const Compare& comp = Compare(), const Allocator& =
Allocator());
    map(const map& x);
    map(map&& x);
    explicit map(const Allocator&);
    map(const map&, const Allocator&);
    map(map&&, const Allocator&);
    map(initializer_list<value_type>,
      const Compare& = Compare(),
      const Allocator& = Allocator());
    template<class InputIterator>
      map(InputIterator first, InputIterator last, const Allocator& a)
        : map(first, last, Compare(), a) { }
    map(initializer_list<value_type> il, const Allocator& a)
      : map(il, Compare(), a) { }
    ~map();
    map& operator=(const map& x);
    map& operator=(map&& x)
      noexcept(allocator_traits<Allocator>::is_always_equal::value &&
               is_nothrow_move_assignable_v<Compare>);
    map& operator=(initializer_list<value_type>);
    allocator_type get_allocator() const noexcept;

    // …
  };
}
```

[map.cons]

~~explicit~~ `map(const Compare& comp, const Allocator&` ~~`= Allocator()`~~`);`

Effects: Constructs an empty `map` using the specified comparison object and allocator.
Complexity: Constant.

[multimap.overview]

```
namespace std {
  template<class Key, class T, class Compare = less<Key>,
           class Allocator = allocator<pair<const Key, T>>>
  class multimap {
  public:
    // …
    // [multimap.cons], construct/copy/destroy
    multimap() : multimap(Compare()) { }
    explicit multimap(const Compare& comp, const Allocator& = 
Allocator());
    explicit multimap(const Compare& comp) : multimap(comp, 
Allocator()) { }
    template<class InputIterator>
      multimap(InputIterator first, InputIterator last,
               const Compare& comp = Compare(),
               const Allocator& = Allocator());
    multimap(const multimap& x);
    multimap(multimap&& x);
    explicit multimap(const Allocator&);
    multimap(const multimap&, const Allocator&);
    multimap(multimap&&, const Allocator&);
    multimap(initializer_list<value_type>,
      const Compare& = Compare(),
      const Allocator& = Allocator());
    template<class InputIterator>
      multimap(InputIterator first, InputIterator last, const 
Allocator& a)
        : multimap(first, last, Compare(), a) { }
    multimap(initializer_list<value_type> il, const Allocator& a)
      : multimap(il, Compare(), a) { }
    ~multimap();
    multimap& operator=(const multimap& x);
    multimap& operator=(multimap&& x)
      noexcept(allocator_traits<Allocator>::is_always_equal::value &&
               is_nothrow_move_assignable_v<Compare>);
    multimap& operator=(initializer_list<value_type>);
    allocator_type get_allocator() const noexcept;

    // …
  };
}
```

[multimap.cons]

~~explicit~~ `multimap(const Compare& comp, const Allocator&` ~~`= Allocator()`~~`);`

Effects: Constructs an empty `multimap` using the specified comparison object and allocator.
Complexity: Constant.

[set.overview]

```
namespace std {
  template<class Key, class Compare = less<Key>,
           class Allocator = allocator<Key>>
  class set {
  public:
    // …
    // [set.cons], construct/copy/destroy
    set() : set(Compare()) { }
    explicit set(const Compare& comp, const Allocator& =
Allocator());
    explicit set(const Compare& comp) : set(comp, Allocator()) { }
    template<class InputIterator>
      set(InputIterator first, InputIterator last,
          const Compare& comp = Compare(), const Allocator& =
Allocator());
    set(const set& x);
    set(set&& x);
    explicit set(const Allocator&);
    set(const set&, const Allocator&);
    set(set&&, const Allocator&);
    set(initializer_list<value_type>, const Compare& = Compare(),
        const Allocator& = Allocator());
    template<class InputIterator>
      set(InputIterator first, InputIterator last, const Allocator&
a)
        : set(first, last, Compare(), a) { }
    set(initializer_list<value_type> il, const Allocator& a)
      : set(il, Compare(), a) { }
    ~set();
    set& operator=(const set& x);
    set& operator=(set&& x)
      noexcept(allocator_traits<Allocator>::is_always_equal::value &&
                is_nothrow_move_assignable_v<Compare>);
    set& operator=(initializer_list<value_type>);
    allocator_type get_allocator() const noexcept;

    // …
  };
}
```

[set.cons]

~~explicit~~ `set(const Compare& comp, const Allocator&` ~~`= Allocator()`~~`);`

Effects: Constructs an empty `set` using the specified comparison objects and allocator.
Complexity: Constant.

[multiset.overview]

```
  namespace std {
    template<class Key, class Compare = less<Key>,
             class Allocator = allocator<Key>>
    class multiset {
    public:
      // …
      // [multiset.cons], construct/copy/destroy
      multiset() : multiset(Compare()) { }
      explicit multiset(const Compare& comp, const Allocator& =
  Allocator());
      explicit multiset(const Compare& comp) : multiset(comp,
  Allocator()) { }
      template<class InputIterator>
        multiset(InputIterator first, InputIterator last,
                 const Compare& comp = Compare(), const Allocator& =
  Allocator());
      multiset(const multiset& x);
      multiset(multiset&& x);
      explicit multiset(const Allocator&);
      multiset(const multiset&, const Allocator&);
      multiset(multiset&&, const Allocator&);
      multiset(initializer_list<value_type>, const Compare& =
  Compare(),
               const Allocator& = Allocator());
      template<class InputIterator>
        multiset(InputIterator first, InputIterator last, const
  Allocator& a)
          : multiset(first, last, Compare(), a) { }
      multiset(initializer_list<value_type> il, const Allocator& a)
        : multiset(il, Compare(), a) { }
      ~multiset();
      multiset& operator=(const multiset& x);
      multiset& operator=(multiset&& x)
        noexcept(allocator_traits<Allocator>::is_always_equal::value &&
                 is_nothrow_move_assignable_v<Compare>);
      multiset& operator=(initializer_list<value_type>);
      allocator_type get_allocator() const noexcept;

      // …
    };
}
```

[multiset.cons]

~~explicit~~ `multiset(const Compare& comp, const Allocator&` ~~`= Allocator()`~~`);`

Effects: Constructs an empty `multiset` using the specified comparison object and allocator.
Complexity: Constant.

[unord.map.overview]

```
namespace std {
  template<class Key,
           class T,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Allocator = allocator<pair<const Key, T>>>
  class unordered_map {
  public:
    // ...
    // [unord.map.cnstr], construct/copy/destroy
    unordered_map();
    explicit unordered_map(size_type n,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a =
allocator_type());
explicit unordered_map(size_type n) : unordered_map(n, hasher()) { }
    template<class InputIterator>
      unordered_map(InputIterator f, InputIterator l,
                    size_type n = see below,
                    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&&);
    explicit unordered_map(const Allocator&);
    unordered_map(const unordered_map&, const Allocator&);
    unordered_map(unordered_map&&, const Allocator&);
    unordered_map(initializer_list<value_type> il,
                  size_type n = see below,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
    unordered_map(size_type n, const allocator_type& a)
      : unordered_map(n, hasher(), key_equal(), a) { }
    unordered_map(size_type n, const hasher& hf, const
allocator_type& a)
      : unordered_map(n, hf, key_equal(), a) { }
    template<class InputIterator>
      unordered_map(InputIterator f, InputIterator l, size_type n,
const allocator_type& a)
        : unordered_map(f, l, n, hasher(), key_equal(), a) { }
    template<class InputIterator>
      unordered_map(InputIterator f, InputIterator l, size_type n,
const hasher& hf,
                    const allocator_type& a)
        : unordered_map(f, l, n, hf, key_equal(), a) { }
```

```
    unordered_map(initializer_list<value_type> il, size_type n, const
allocator_type& a)
      : unordered_map(il, n, hasher(), key_equal(), a) { }
    unordered_map(initializer_list<value_type> il, size_type n, const
hasher& hf,
                  const allocator_type& a)
      : unordered_map(il, n, hf, key_equal(), a) { }
    ~unordered_map();
    unordered_map& operator=(const unordered_map&);
    unordered_map& operator=(unordered_map&&)
      noexcept(allocator_traits<Allocator>::is_always_equal::value &&
               is_nothrow_move_assignable_v<Hash> &&
               is_nothrow_move_assignable_v<Pred>);
    unordered_map& operator=(initializer_list<value_type>);
    allocator_type get_allocator() const noexcept;

    // …
  };
}
```

[unord.map.cnstr]

```
unordered_map() : unordered_map(size_type(see below)) { }
explicit unordered_map(size_type n,
                       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 predicate, and allocator, and using at least `n` buckets. For the default constructor, the number of buckets is implementation-defined. `max_load_factor()` returns `1.0`.

Complexity: Constant.

[unord.multimap.overview]

```
namespace std {
  template<class Key,
           class T,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Allocator = allocator<pair<const Key, T>>>
  class unordered_multimap {
  public:
    // ...
    // [unord.multimap.cnstr], construct/copy/destroy
    unordered_multimap();
    explicit unordered_multimap(size_type n,
                                const hasher& hf = hasher(),
                                const key_equal& eql = key_equal(),
                                const allocator_type& a =
allocator_type());
    explicit unordered_multimap(size_type n) : unordered_multimap(n,
hasher()) { }
    template<class InputIterator>
      unordered_multimap(InputIterator f, InputIterator l,
                         size_type n = see below,
                         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&&);
    explicit unordered_multimap(const Allocator&);
    unordered_multimap(const unordered_multimap&, const Allocator&);
    unordered_multimap(unordered_multimap&&, const Allocator&);
    unordered_multimap(initializer_list<value_type> il,
                       size_type n = see below,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
    unordered_multimap(size_type n, const allocator_type& a)
      : unordered_multimap(n, hasher(), key_equal(), a) { }
    unordered_multimap(size_type n, const hasher& hf, const
allocator_type& a)
      : unordered_multimap(n, hf, key_equal(), a) { }
    template<class InputIterator>
      unordered_multimap(InputIterator f, InputIterator l, size_type
n, const allocator_type& a)
        : unordered_multimap(f, l, n, hasher(), key_equal(), a) { }
    template<class InputIterator>
      unordered_multimap(InputIterator f, InputIterator l, size_type
n, const hasher& hf,
                         const allocator_type& a)
        : unordered_multimap(f, l, n, hf, key_equal(), a) { }
```

```
    unordered_multimap(initializer_list<value_type> il, size_type n,
const allocator_type& a)
      : unordered_multimap(il, n, hasher(), key_equal(), a) { }
    unordered_multimap(initializer_list<value_type> il, size_type n,
const hasher& hf,
                       const allocator_type& a)
      : unordered_multimap(il, n, hf, key_equal(), a) { }
    ~unordered_multimap();
    unordered_multimap& operator=(const unordered_multimap&);
    unordered_multimap& operator=(unordered_multimap&&)
      noexcept(allocator_traits<Allocator>::is_always_equal::value &&
               is_nothrow_move_assignable_v<Hash> &&
               is_nothrow_move_assignable_v<Pred>);
    unordered_multimap& operator=(initializer_list<value_type>);
    allocator_type get_allocator() const noexcept;

    // ...
  };
}
```

[unord.multimap.cnstr]

```
unordered_multimap() : unordered_multimap(size_type(see below)) { }
explicit unordered_multimap(size_type n,
                            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 predicate, and allocator, and using at least `n` buckets. For the default constructor, the number of buckets is implementation-defined. `max_load_factor()` returns `1.0`.

Complexity: Constant.

[unord.set.overview]

```
namespace std {
  template<class Key,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Allocator = allocator<Key>>
  class unordered_set {
  public:
    // ...
    // [unord.set.cnstr], construct/copy/destroy
    unordered_set();
    explicit unordered_set(size_type n,
                           const hasher& hf = hasher(),
                           const key_equal& eql = key_equal(),
                           const allocator_type& a =
allocator_type());
    explicit unordered_set(size_type n) : unordered_set(n, hasher())
{ }
    template<class InputIterator>
      unordered_set(InputIterator f, InputIterator l,
                    size_type n = see below,
                    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&&);
    explicit unordered_set(const Allocator&);
    unordered_set(const unordered_set&, const Allocator&);
    unordered_set(unordered_set&&, const Allocator&);
    unordered_set(initializer_list<value_type> il,
                  size_type n = see below,
                  const hasher& hf = hasher(),
                  const key_equal& eql = key_equal(),
                  const allocator_type& a = allocator_type());
    unordered_set(size_type n, const allocator_type& a)
      : unordered_set(n, hasher(), key_equal(), a) { }
    unordered_set(size_type n, const hasher& hf, const
allocator_type& a)
      : unordered_set(n, hf, key_equal(), a) { }
    template<class InputIterator>
      unordered_set(InputIterator f, InputIterator l, size_type n,
const allocator_type& a)
        : unordered_set(f, l, n, hasher(), key_equal(), a) { }
    template<class InputIterator>
      unordered_set(InputIterator f, InputIterator l, size_type n,
const hasher& hf,
                    const allocator_type& a)
      : unordered_set(f, l, n, hf, key_equal(), a) { }
```

```
    unordered_set(initializer_list<value_type> il, size_type n, const
allocator_type& a)
      : unordered_set(il, n, hasher(), key_equal(), a) { }
    unordered_set(initializer_list<value_type> il, size_type n, const
hasher& hf,
                  const allocator_type& a)
      : unordered_set(il, n, hf, key_equal(), a) { }
    ~unordered_set();
    unordered_set& operator=(const unordered_set&);
    unordered_set& operator=(unordered_set&&)
      noexcept(allocator_traits<Allocator>::is_always_equal::value &&
               is_nothrow_move_assignable_v<Hash> &&
               is_nothrow_move_assignable_v<Pred>);
    unordered_set& operator=(initializer_list<value_type>);
    allocator_type get_allocator() const noexcept;

    // …
  };
}
```

[unord.set.cnstr]

```
unordered_set() : unordered_set(size_type(see below)) { }
explicit unordered_set(size_type n,
                       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 predicate, and allocator, and using at least `n`buckets. For the default constructor, the number of buckets is implementation-defined. `max_load_factor()` returns `1.0`.

Complexity: Constant.

[unord.multiset.overview]

```
namespace std {
  template<class Key,
           class Hash = hash<Key>,
           class Pred = equal_to<Key>,
           class Allocator = allocator<Key>>
  class unordered_multiset {
  public:
    // ...
    // [unord.multiset.cnstr], construct/copy/destroy
    unordered_multiset();
    explicit unordered_multiset(size_type n,
                                const hasher& hf = hasher(),
                                const key_equal& eql = key_equal(),
                                const allocator_type& a =
allocator_type());
    explicit unordered_multiset(size_type n) : unordered_multiset(n,
hasher()) { }
    template<class InputIterator>
      unordered_multiset(InputIterator f, InputIterator l,
                         size_type n = see below,
                         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&&);
    explicit unordered_multiset(const Allocator&);
    unordered_multiset(const unordered_multiset&, const Allocator&);
    unordered_multiset(unordered_multiset&&, const Allocator&);
    unordered_multiset(initializer_list<value_type> il,
                       size_type n = see below,
                       const hasher& hf = hasher(),
                       const key_equal& eql = key_equal(),
                       const allocator_type& a = allocator_type());
    unordered_multiset(size_type n, const allocator_type& a)
      : unordered_multiset(n, hasher(), key_equal(), a) { }
    unordered_multiset(size_type n, const hasher& hf, const
allocator_type& a)
      : unordered_multiset(n, hf, key_equal(), a) { }
    template<class InputIterator>
      unordered_multiset(InputIterator f, InputIterator l, size_type
n, const allocator_type& a)
        : unordered_multiset(f, l, n, hasher(), key_equal(), a) { }
    template<class InputIterator>
      unordered_multiset(InputIterator f, InputIterator l, size_type
n, const hasher& hf,
                         const allocator_type& a)
      : unordered_multiset(f, l, n, hf, key_equal(), a) { }
```

```
    unordered_multiset(initializer_list<value_type> il, size_type n,
const allocator_type& a)
      : unordered_multiset(il, n, hasher(), key_equal(), a) { }
    unordered_multiset(initializer_list<value_type> il, size_type n,
const hasher& hf,
                       const allocator_type& a)
      : unordered_multiset(il, n, hf, key_equal(), a) { }
    ~unordered_multiset();
    unordered_multiset& operator=(const unordered_multiset&);
    unordered_multiset& operator=(unordered_multiset&&)
      noexcept(allocator_traits<Allocator>::is_always_equal::value &&
               is_nothrow_move_assignable_v<Hash> &&
               is_nothrow_move_assignable_v<Pred>);
    unordered_multiset& operator=(initializer_list<value_type>);
    allocator_type get_allocator() const noexcept;

    // …
  };
}
```

[unord.multiset.cnstr]

```
unordered_multiset() : unordered_multiset(size_type(see below)) { }
explicit unordered_multiset(size_type n,
                            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 predicate, and allocator, and using at least `n` buckets. For the default constructor, the number of buckets is implementation-defined. `max_load_factor()` returns `1.0`.

Complexity: Constant.

[rand.dist.uni.int]

```
template<class IntType = int>
  class uniform_int_distribution {
  public:
    // types
    using result_type = IntType;
    using param_type  = unspecified;

    // constructors and reset functions
    uniform_int_distribution() : uniform_int_distribution(0) {}
    explicit uniform_int_distribution(IntType a, IntType b =
numeric_limits<IntType>::max());
    explicit uniform_int_distribution(IntType a) :
uniform_int_distribution(a, numeric_limits<IntType>::max()) { }
    explicit uniform_int_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    result_type a() const;
    result_type b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

```
explicit uniform_int_distribution(IntType a, IntType b =
numeric_limits<IntType>::max());
```

Requires: a≤b

Effects: Constructs a `uniform_int_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

[rand.dist.uni.real]

```
template<class RealType = double>
  class uniform_real_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructors and reset functions
    uniform_real_distribution() : uniform_real_distribution(0.0) {}
    explicit uniform_real_distribution(RealType a, RealType b = 1.0);
    explicit uniform_real_distribution(RealType a) :
uniform_real_distribution(a, 1.0) { }
    explicit uniform_real_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    result_type a() const;
    result_type b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };

explicit uniform_real_distribution(RealType a, RealType b = 1.0);
```

Requires: $a \le b$ and $b - a \le$ `numeric_limits<RealType>::max()`.

Effects: Constructs a `uniform_real_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

[rand.dist.bern.bin]

```
  template<class IntType = int>
    class binomial_distribution {
    public:
      // types
      using result_type = IntType;
      using param_type  = unspecified;

      // constructors and reset functions
      binomial_distribution() : binomial_distribution(1) {}
      explicit binomial_distribution(IntType t, double p = 0.5);
      explicit binomial_distribution(IntType t) :
  binomial_distribution(t, 0.5) { }
      explicit binomial_distribution(const param_type& parm);
      void reset();

      // generating functions
      template<class URBG>
        result_type operator()(URBG& g);
      template<class URBG>
        result_type operator()(URBG& g, const param_type& parm);

      // property functions
      IntType t() const;
      double p() const;
      param_type param() const;
      void param(const param_type& parm);
      result_type min() const;
      result_type max() const;
    };

explicit binomial_distribution(IntType t, double p = 0.5);
```

Requires: $0 \le p \le 1$ and $0 \le t$.

Effects: Constructs a `binomial_distribution` object; `t` and `p` correspond to the respective parameters of the distribution.

[rand.dist.bern.negbin]

```
template<class IntType = int>
  class negative_binomial_distribution {
  public:
    // types
    using result_type = IntType;
    using param_type  = unspecified;

    // constructor and reset functions
    negative_binomial_distribution() :
negative_binomial_distribution(1) {}
    explicit negative_binomial_distribution(IntType k, double p = 0.5);
    explicit negative_binomial_distribution(IntType k) :
negative_binomial_distribution(k, 0.5) { }
    explicit negative_binomial_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    IntType k() const;
    double p() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };

explicit negative_binomial_distribution(IntType k, double p = 0.5);
```

Requires: $0 < p \leq 1$ and $0 < k$.

Effects: Constructs a `negative_binomial_distribution` object; `k` and `p` correspond to the respective parameters of the distribution.

[rand.dist.pois.gamma]

```
template<class RealType = double>
  class gamma_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructors and reset functions
    gamma_distribution() : gamma_distribution(1.0) {}
    explicit gamma_distribution(RealType alpha, RealType beta = 1.0);
    explicit gamma_distribution(RealType alpha) :
gamma_distribution(alpha, 1.0) { }
    explicit gamma_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType alpha() const;
    RealType beta() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };

explicit gamma_distribution(RealType alpha, RealType beta = 1.0);
```

Requires: 0<`alpha` and 0<`beta`.

Effects: Constructs a `gamma_distribution` object; `alpha` and `beta` correspond to the parameters of the distribution.

[rand.dist.pois.weibull]

```
template<class RealType = double>
  class weibull_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructor and reset functions
    weibull_distribution() : weibull_distribution(1.0) {}
    explicit weibull_distribution(RealType a, RealType b = 1.0);
    explicit weibull_distribution(RealType a) : weibull_distribution
(a, 1.0) { }
    explicit weibull_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType a() const;
    RealType b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };

explicit weibull_distribution(RealType a, RealType b = 1.0);
```

Requires: 0<a and 0<b.

Effects: Constructs a `weibull_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

[rand.dist.pois.extreme]

```
template<class RealType = double>
  class extreme_value_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructor and reset functions
    extreme_value_distribution() : extreme_value_distribution(0.0) {}
    explicit extreme_value_distribution(RealType a, RealType b = 1.0);
    explicit extreme_value_distribution(RealType a) : extreme_value_distribution(a, 1.0) { }
    explicit extreme_value_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType a() const;
    RealType b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };

explicit extreme_value_distribution(RealType a, RealType b = 1.0);
```

Requires: 0<b.

Effects: Constructs an `extreme_value_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

[rand.dist.norm.normal]

```
template<class RealType = double>
  class normal_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructors and reset functions
    normal_distribution() : normal_distribution(0.0) {}
    explicit normal_distribution(RealType mean, RealType stddev = 1.0);
    explicit normal_distribution(RealType mean) : normal_distribution(mean, 1.0) { }
    explicit normal_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType mean() const;
    RealType stddev() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };

explicit normal_distribution(RealType mean, RealType stddev = 1.0);
```

Requires: 0<`stddev`.

Effects: Constructs a `normal_distribution` object; `mean` and `stddev` correspond to the respective parameters of the distribution.

[rand.dist.norm.lognormal]

```
template<class RealType = double>
  class lognormal_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructor and reset functions
    lognormal_distribution() : lognormal_distribution(0.0) {}
    explicit lognormal_distribution(RealType m, RealType s = 1.0);
    explicit lognormal_distribution(RealType m) :
lognormal_distribution(m, 1.0) { }
    explicit lognormal_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType m() const;
    RealType s() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };

explicit lognormal_distribution(RealType m, RealType s = 1.0);
```

Requires: 0<s.

Effects: Constructs a `lognormal_distribution` object; `m` and `s` correspond to the respective parameters of the distribution.

[rand.dist.norm.cauchy]

```
template<class RealType = double>
  class cauchy_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructor and reset functions
    cauchy_distribution() : cauchy_distribution(0.0) {}
    explicit cauchy_distribution(RealType a, RealType b = 1.0);
    explicit cauchy distribution(RealType a) : cauchy distribution(a,
1.0) { }
    explicit cauchy_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType a() const;
    RealType b() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };
```

```
explicit cauchy_distribution(RealType a, RealType b = 1.0);
```

Requires: 0<b.

Effects: Constructs a `cauchy_distribution` object; `a` and `b` correspond to the respective parameters of the distribution.

[rand.dist.norm.f]

```
template<class RealType = double>
  class fisher_f_distribution {
  public:
    // types
    using result_type = RealType;
    using param_type  = unspecified;

    // constructor and reset functions
    fisher_f_distribution() : fisher_f_distribution(1) {}
    explicit fisher_f_distribution(RealType m, RealType n = 1);
    explicit fisher_f_distribution(RealType m) :
fisher_f_distribution(m, 1.0) { }
    explicit fisher_f_distribution(const param_type& parm);
    void reset();

    // generating functions
    template<class URBG>
      result_type operator()(URBG& g);
    template<class URBG>
      result_type operator()(URBG& g, const param_type& parm);

    // property functions
    RealType m() const;
    RealType n() const;
    param_type param() const;
    void param(const param_type& parm);
    result_type min() const;
    result_type max() const;
  };

explicit fisher_f_distribution(RealType m, RealType n = 1);
```

Requires: 0<m and 0<n.

Effects: Constructs a `fisher_f_distribution` object; `m` and `n` correspond to the respective parameters of the distribution.

[ios::failure]

```
namespace std {
  class ios_base::failure : public system_error {
  public:
    explicit failure(const string& msg, const error_code& ec =
io_errc::stream);
    explicit failure(const string& msg) : failure(msg,
io_errc::stream) { }
    explicit failure(const char* msg, const error_code& ec =
io_errc::stream);
    explicit failure(const char* msg) : failure(msg, io_errc::stream)
{ }
  };
}
```

```
explicit failure(const string& msg, const error_code& ec =
io_errc::stream);
```

Effects: Constructs an object of class `failure` by constructing the base class with `msg` and `ec`

```
explicit failure(const char* msg, const error_code& ec =
io_errc::stream);
```

Effects: Constructs an object of class `failure` by constructing the base class with `msg` and `ec`.

[istream::sentry]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_istream<charT, traits>::sentry {
    using traits_type = traits;
    bool ok_; // exposition only
  public:
    explicit sentry(basic_istream<charT, traits>& is, bool noskipws = false);
    explicit sentry(basic_istream<charT, traits>& is) : sentry(is, false) { }
    ~sentry();
    explicit operator bool() const { return ok_; }
    sentry(const sentry&) = delete;
    sentry& operator=(const sentry&) = delete;
  };
}
```

`explicit sentry(basic_istream<charT, traits>& is, bool noskipws = false);`

Effects: If `is.good()` is `false`, calls `is.setstate(failbit)`.

Otherwise, prepares for formatted or unformatted input.

First, if `is.tie()` is not a null pointer, the function calls `is.tie()->flush()` to synchronize the output sequence with any associated external C stream.

Except that this call can be suppressed if the put area of `is.tie()` is empty.

Further an implementation is allowed to defer the call to `flush` until a call of `is.rdbuf()->underflow()` occurs.

If no such call occurs before the `sentry` object is destroyed, the call to `flush` may be eliminated entirely.

If `noskipws` is zero and `is.flags() & ios_base::skipws` is nonzero, the function extracts and discards each character as long as the next available input character `c` is a whitespace character.

If `is.rdbuf()->sbumpc()` or `is.rdbuf()->sgetc()` returns `traits::eof()`, the function calls `setstate(failbit | eofbit)` (which may throw `ios_base::failure`).

Remarks: The constructor

`explicit sentry(basic_istream<charT, traits>& is, bool noskipws = false)`

uses the currently imbued locale in `is`, to determine whether the next input character is whitespace or not.

To decide if the character `c` is a whitespace character, the constructor performs as if it executes the following code fragment:

```
const ctype<charT>& ctype = use_facet<ctype<charT>>(is.getloc());
if (ctype.is(ctype.space, c) != 0)
  // c is a whitespace character.
```

If, after any preparation is completed, `is.good()` is `true`, `ok_ != false` otherwise, `ok_ == false`.

During preparation, the constructor may call `setstate(failbit)` (which may throw `ios_base::failure` ([iostate.flags])

[stringbuf]

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_stringbuf : public basic_streambuf<charT, traits> {
  public:
    // ...
    // [stringbuf.cons], constructors
    basic_stringbuf() : basic_stringbuf(ios_base::in | ios_base::out)
{}
    explicit basic_stringbuf(ios_base::openmode which);
    explicit basic_stringbuf(
      const basic_string<charT, traits, Allocator>& str,
      ios_base::openmode which = ios_base::in | ios_base::out);
    explicit basic_stringbuf(
      const basic_string<charT, traits, Allocator>& str) :
basic_stringbuf(str, ios_base::in | ios_base::out) { }
    basic_stringbuf(const basic_stringbuf& rhs) = delete;
    basic_stringbuf(basic_stringbuf&& rhs);

    // ...
  };
}
```

[stringbuf.cons]

```
explicit basic_stringbuf(ios_base::openmode which);
```

Effects: Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()` ([streambuf.cons]), and initializing `mode` with `which`.

Ensures: `str() == ""`.

```
explicit basic_stringbuf(
  const basic_string<charT, traits, Allocator>& s,
  ios_base::openmode which = ios_base::in | ios_base::out);
```

Effects: Constructs an object of class `basic_stringbuf`, initializing the base class with `basic_streambuf()` ([streambuf.cons]), and initializing `mode` with `which`.

Then calls `str(s)`.

[istringstream]

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_istringstream : public basic_istream<charT, traits> {
  public:
    using char_type      = charT;
    using int_type       = typename traits::int_type;
    using pos_type       = typename traits::pos_type;
    using off_type       = typename traits::off_type;
    using traits_type    = traits;
    using allocator_type = Allocator;

    // [istringstream.cons], constructors
    basic_istringstream() : basic_istringstream(ios_base::in) {}
    explicit basic_istringstream(ios_base::openmode which);
    explicit basic_istringstream(
      const basic_string<charT, traits, Allocator>& str,
      ios_base::openmode which = ios_base::in);
    explicit basic_istringstream(
      const basic_string<charT, traits, Allocator>& str) :
basic_istringstream(str, ios_base::in) { }
    basic_istringstream(const basic_istringstream& rhs) = delete;
    basic_istringstream(basic_istringstream&& rhs);

    // [istringstream.assign], assign and swap
    basic_istringstream& operator=(const basic_istringstream& rhs) =
delete;
    basic_istringstream& operator=(basic_istringstream&& rhs);
    void swap(basic_istringstream& rhs);

    // [istringstream.members], members
    basic_stringbuf<charT, traits, Allocator>* rdbuf() const;

    basic_string<charT, traits, Allocator> str() const;
    void str(const basic_string<charT, traits, Allocator>& s);
  private:
    basic_stringbuf<charT, traits, Allocator> sb; // exposition only
  };

  template<class charT, class traits, class Allocator>
    void swap(basic_istringstream<charT, traits, Allocator>& x,
              basic_istringstream<charT, traits, Allocator>& y);
}
```

[istringstream.cons]

```
explicit basic_istringstream(ios_base::openmode which);
```

Effects: Constructs an object of class `basic_istringstream<charT, traits>`, initializing the base class with `basic_istream<charT, traits>(&sb)` ([istream]) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which | ios_base::in)` ([stringbuf.cons]).

```
explicit basic_istringstream(
  const basic_string<charT, traits, Allocator>& str,
  ios_base::openmode which = ios_base::in);
```

Effects: Constructs an object of class `basic_istringstream<charT, traits>`, initializing the base class with `basic_istream<charT, traits>(&sb)` ([istream]) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::in)` ([stringbuf.cons]).

[ostringstream]

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_ostringstream : public basic_ostream<charT, traits> {
  public:
    using char_type      = charT;
    using int_type       = typename traits::int_type;
    using pos_type       = typename traits::pos_type;
    using off_type       = typename traits::off_type;
    using traits_type    = traits;
    using allocator_type = Allocator;

    // [ostringstream.cons], constructors
    basic_ostringstream() : basic_ostringstream(ios_base::out) {}
    explicit basic_ostringstream(ios_base::openmode which);
    explicit basic_ostringstream(
      const basic_string<charT, traits, Allocator>& str,
      ios_base::openmode which = ios_base::out);
    explicit basic_ostringstream(
      const basic_string<charT, traits, Allocator>& str) :
basic_ostringstream(str, ios_base::out) { }
    basic_ostringstream(const basic_ostringstream& rhs) = delete;
    basic_ostringstream(basic_ostringstream&& rhs);

    // [ostringstream.assign], assign and swap
    basic_ostringstream& operator=(const basic_ostringstream& rhs) =
delete;
    basic_ostringstream& operator=(basic_ostringstream&& rhs);
    void swap(basic_ostringstream& rhs);

    // [ostringstream.members], members
    basic_stringbuf<charT, traits, Allocator>* rdbuf() const;

    basic_string<charT, traits, Allocator> str() const;
    void str(const basic_string<charT, traits, Allocator>& s);
   private:
    basic_stringbuf<charT, traits, Allocator> sb; // exposition only
  };

  template<class charT, class traits, class Allocator>
    void swap(basic_ostringstream<charT, traits, Allocator>& x,
              basic_ostringstream<charT, traits, Allocator>& y);
}
```

[ostringstream.cons]

```
explicit basic_ostringstream(ios_base::openmode which);
```

Effects: Constructs an object of class `basic_ostringstream<charT, traits>`, initializing the base class with `basic_ostream<charT, traits>(&sb)` ([ostream]) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which | ios_base::out)` ([stringbuf.cons]).

```
~~explicit~~ basic_ostringstream(
  const basic_string<charT, traits, Allocator>& str,
  ios_base::openmode which~~ = ios_base::out~~);
```

Effects: Constructs an object of class `basic_ostringstream<charT, traits>`, initializing the base class with `basic_ostream<charT, traits>(&sb)` ([ostream]) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which | ios_base::out)` ([stringbuf.cons]).

[stringstream]

```
namespace std {
  template<class charT, class traits = char_traits<charT>,
           class Allocator = allocator<charT>>
  class basic_stringstream : public basic_iostream<charT, traits> {
  public:
    using char_type      = charT;
    using int_type       = typename traits::int_type;
    using pos_type       = typename traits::pos_type;
    using off_type       = typename traits::off_type;
    using traits_type    = traits;
    using allocator_type = Allocator;

    // [stringstream.cons], constructors
    basic_stringstream() : basic_stringstream(ios_base::out |
ios_base::in) {}
    explicit basic_stringstream(ios_base::openmode which);
    explicit basic_stringstream(
      const basic_string<charT, traits, Allocator>& str,
      ios_base::openmode which = ios_base::out | ios_base::in);
    explicit basic_stringstream(
      const basic_string<charT, traits, Allocator>& str) :
basic_stringstream(str, ios_base::out | ios_base::in) { }
    basic_stringstream(const basic_stringstream& rhs) = delete;
    basic_stringstream(basic_stringstream&& rhs);

    // [stringstream.assign], assign and swap
    basic_stringstream& operator=(const basic_stringstream& rhs) =
delete;
    basic_stringstream& operator=(basic_stringstream&& rhs);
    void swap(basic_stringstream& rhs);

    // [stringstream.members], members
    basic_stringbuf<charT, traits, Allocator>* rdbuf() const;
    basic_string<charT, traits, Allocator> str() const;
    void str(const basic_string<charT, traits, Allocator>& str);

  private:
    basic_stringbuf<charT, traits> sb;  // exposition only
  };

  template<class charT, class traits, class Allocator>
    void swap(basic_stringstream<charT, traits, Allocator>& x,
              basic_stringstream<charT, traits, Allocator>& y);
}
```

[stringstream.cons]

```
explicit basic_stringstream(ios_base::openmode which);
```

Effects: Constructs an object of class `basic_stringstream<charT, traits>`, initializing the base class with `basic_iostream<charT, traits>(&sb)` ([iostream.cons]) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(which)`.

```
~~explicit~~ basic_stringstream(
  const basic_string<charT, traits, Allocator>& str,
  ios_base::openmode which~~ = ios_base::out | ios_base::in~~);
```

Effects: Constructs an object of class `basic_stringstream<charT, traits>`, initializing the base class with `basic_iostream<charT, traits>(&sb)` ([iostream.cons]) and initializing `sb` with `basic_stringbuf<charT, traits, Allocator>(str, which)`.

[ifstream]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_ifstream : public basic_istream<charT, traits> {
  public:
    using char_type   = charT;
    using int_type    = typename traits::int_type;
    using pos_type    = typename traits::pos_type;
    using off_type    = typename traits::off_type;
    using traits_type = traits;

    // [ifstream.cons], constructors
    basic_ifstream();
    explicit basic_ifstream(const char* s,
                            ios_base::openmode mode = ios_base::in);
    explicit basic_ifstream(const char* s) : basic_ifstream(s,
ios_base::in) { }
    explicit basic_ifstream(const filesystem::path::value_type* s,
                            ios_base::openmode mode = ios_base::in);
// wide systems only; see [fstream.syn]
    explicit basic_ifstream(const filesystem::path::value_type* s) :
basic_ifstream(s, ios_base::in) { }  // wide systems only; see [fstream.syn]
    explicit basic_ifstream(const string& s,
                            ios_base::openmode mode = ios_base::in);
    explicit basic_ifstream(const string& s) : basic_ifstream(s,
ios_base::in) { }
    explicit basic_ifstream(const filesystem::path& s,
                            ios_base::openmode mode = ios_base::in);
    explicit basic_ifstream(const filesystem::path& s) :
basic_ifstream(s, ios_base::in) { }
    basic_ifstream(const basic_ifstream& rhs) = delete;
    basic_ifstream(basic_ifstream&& rhs);


    // ...
  };
}
```

[ifstream.cons]

```
explicit basic_ifstream(const char* s,
                        ios_base::openmode mode = ios_base::in);
explicit basic_ifstream(const filesystem::path::value_type* s,
                        ios_base::openmode mode = ios_base::in);   // wide
```
systems only; see [fstream.syn]

Effects: Constructs an object of class `basic_ifstream<charT, traits>`, initializing the base class with `basic_istream<charT, traits>(&sb)` ([istream.cons]) and initializing `sb` with `basic_filebuf<charT, traits>()` ([filebuf.cons]), then calls `rdbuf()->open(s, mode | ios_base::in)`.

If that function returns a null pointer, calls `setstate(failbit)`.

```
explicit basic_ifstream(const string& s,
                        ios_base::openmode mode = ios_base::in);
explicit basic_ifstream(const filesystem::path& s,
                        ios_base::openmode mode = ios_base::in);
```

Effects: The same as `basic_ifstream(s.c_str(), mode)`.

[ofstream]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_ofstream : public basic_ostream<charT, traits> {
  public:
    using char_type   = charT;
    using int_type    = typename traits::int_type;
    using pos_type    = typename traits::pos_type;
    using off_type    = typename traits::off_type;
    using traits_type = traits;

    // [ofstream.cons], constructors
    basic_ofstream();
    explicit basic_ofstream(const char* s,
                            ios_base::openmode mode = ios_base::out);
    explicit basic_ofstream(const char* s) : basic_ofstream(s,
ios_base::out) { }
    explicit basic_ofstream(const filesystem::path::value_type* s,
                            ios_base::openmode mode = ios_base::out);
// wide systems only; see [fstream.syn]
    explicit basic_ofstream(const filesystem::path::value_type* s) :
basic_ofstream(s, ios_base::out) { }  // wide systems only; see [fstream.syn]
    explicit basic_ofstream(const string& s,
                            ios_base::openmode mode = ios_base::out);
    explicit basic_ofstream(const string& s) : basic_ofstream(s,
ios_base::out) { }
    explicit basic_ofstream(const filesystem::path& s,
                            ios_base::openmode mode = ios_base::out);
    explicit basic_ofstream(const filesystem::path& s) :
basic_ofstream(s, ios_base::out) { }
    basic_ofstream(const basic_ofstream& rhs) = delete;
    basic_ofstream(basic_ofstream&& rhs);
    // ...
  };
}
```

[ofstream.cons]

```
explicit basic_ofstream(const char* s,
                        ios_base::openmode mode = ios_base::out);
explicit basic_ofstream(const filesystem::path::value_type* s,
                        ios_base::openmode mode = ios_base::out); // wide
```

systems only; see [fstream.syn]

Effects: Constructs an object of class `basic_ofstream<charT, traits>`, initializing the base class with `basic_ostream<charT, traits>(&sb)` ([ostream.cons]) and initializing `sb` with `basic_filebuf<charT, traits>()` ([filebuf.cons]), then calls `rdbuf()->open(s, mode | ios_base::out)`.

If that function returns a null pointer, calls `setstate(failbit)`.

```
explicit basic_ofstream(const string& s,
                        ios_base::openmode mode = ios_base::out);
explicit basic_ofstream(const filesystem::path& s,
                        ios_base::openmode mode = ios_base::out);
```

Effects: The same as `basic_ofstream(s.c_str(), mode)`.

[fstream]

```
namespace std {
  template<class charT, class traits = char_traits<charT>>
  class basic_fstream : public basic_iostream<charT, traits> {
  public:
    using char_type   = charT;
    using int_type    = typename traits::int_type;
    using pos_type    = typename traits::pos_type;
    using off_type    = typename traits::off_type;
    using traits_type = traits;

    // [fstream.cons], constructors
    basic_fstream();
    explicit basic_fstream(
      const char* s,
      ios_base::openmode mode = ios_base::in | ios_base::out);
    explicit basic_fstream(const char* s) : basic_fstream(s,
ios_base::in | ios_base::out) { }
    explicit basic_fstream(
      const filesystem::path::value_type* s,
      ios_base::openmode mode = ios_base::in|ios_base::out);   // wide
systems only; see [fstream.syn]
    explicit basic_fstream(const filesystem::path::value_type* s) :
basic_fstream(s, ios_base::in|ios_base::out) { }   // wide systems only; see
[fstream.syn]
    explicit basic_fstream(
      const string& s,
      ios_base::openmode mode = ios_base::in | ios_base::out);
    explicit basic_fstream(const string& s) : basic_fstream(s,
ios_base::in | ios_base::out) { }
    explicit basic_fstream(
      const filesystem::path& s,
      ios_base::openmode mode = ios_base::in | ios_base::out);
    explicit basic_fstream(const filesystem::path& s) :
basic_fstream(s, ios_base::in | ios_base::out) { }
    basic_fstream(const basic_fstream& rhs) = delete;
    basic_fstream(basic_fstream&& rhs);

    // [fstream.assign], assign and swap
    basic_fstream& operator=(const basic_fstream& rhs) = delete;
    basic_fstream& operator=(basic_fstream&& rhs);
    void swap(basic_fstream& rhs);

    // [fstream.members], members
    basic_filebuf<charT, traits>* rdbuf() const;
    bool is_open() const;
    void open(
      const char* s,
      ios_base::openmode mode = ios_base::in | ios_base::out);
    void open(
```

```
      const filesystem::path::value_type* s,
      ios_base::openmode mode = ios_base::in|ios_base::out);  // wide
```
systems only; see [fstream.syn]
```
    void open(
      const string& s,
      ios_base::openmode mode = ios_base::in | ios_base::out);
    void open(
      const filesystem::path& s,
      ios_base::openmode mode = ios_base::in | ios_base::out);
    void close();

  private:
    basic_filebuf<charT, traits> sb; // exposition only
  };

  template<class charT, class traits>
    void swap(basic_fstream<charT, traits>& x,
              basic_fstream<charT, traits>& y);
}
```

[fstream.cons]

```
explicit basic_fstream(
  const char* s,
  ios_base::openmode mode = ios_base::in | ios_base::out);
explicit basic_fstream(
  const filesystem::path::value_type* s,
  ios_base::openmode mode = ios_base::in | ios_base::out);   // wide systems
```
only; see [fstream.syn]

Effects: Constructs an object of class `basic_fstream<charT, traits>`, initializing the base class with `basic_iostream<charT, traits>(&sb)` ([iostream.cons]) and initializing `sb` with `basic_filebuf<charT, traits>()`.

Then calls `rdbuf()->open(s, mode)`.

If that function returns a null pointer, calls `setstate(failbit)`.

```
explicit basic_fstream(
  const string& s,
  ios_base::openmode mode = ios_base::in | ios_base::out);
explicit basic_fstream(
  const filesystem::path& s,
  ios_base::openmode mode = ios_base::in | ios_base::out);
```

Effects: The same as `basic_fstream(s.c_str(), mode)`.

[syncstream.syncbuf.overview]

```
namespace std {
  template<class charT, class traits, class Allocator>
  class basic_syncbuf : public basic_streambuf<charT, traits> {
  public:
    using char_type      = charT;
    using int_type       = typename traits::int_type;
    using pos_type       = typename traits::pos_type;
    using off_type       = typename traits::off_type;
    using traits_type    = traits;
    using allocator_type = Allocator;

    using streambuf_type = basic_streambuf<charT, traits>;
```

// [[[Note for reviewers: The change to the default constructor below is from applying P0935, but this class was not in the working paper when P0935R0 was applied.]]]

```
    // [syncstream.syncbuf.cons], construction and destruction
    basic_syncbuf()
      : basic_syncbuf(nullptr) { }
    explicit basic_syncbuf(streambuf_type* obuf = nullptr)
      : basic_syncbuf(obuf, Allocator()) {}
    basic_syncbuf(streambuf_type*, const Allocator&);
    basic_syncbuf(basic_syncbuf&&);
    ~basic_syncbuf();

    // [syncstream.syncbuf.assign], assignment and swap
    basic_syncbuf& operator=(basic_syncbuf&&);
    void swap(basic_syncbuf&);

    // [syncstream.syncbuf.members], member functions
    bool emit();
    streambuf_type* get_wrapped() const noexcept;
    allocator_type get_allocator() const noexcept;
    void set_emit_on_sync(bool) noexcept;

  protected:
    // [syncstream.syncbuf.virtuals], overridden virtual functions
    int sync() override;

  private:
    streambuf_type* wrapped;      // exposition only
    bool emit_on_sync{};          // exposition only
  };

  // [syncstream.syncbuf.special], specialized algorithms
  template<class charT, class traits, class Allocator>
    void swap(basic_syncbuf<charT, traits, Allocator>&,
              basic_syncbuf<charT, traits, Allocator>&);
}
```

[fs.class.file_status]

```
namespace std::filesystem {
  class file_status {
  public:
    // [fs.file_status.cons], constructors and destructor
    file_status() noexcept : file_status(file_type::none) {}
    explicit file_status(file_type ft,
                         perms prms = perms::unknown) noexcept;
    explicit file_status(file_type ft) noexcept : file_status(ft,
perms::unknown) { }
    file_status(const file_status&) noexcept = default;
    file_status(file_status&&) noexcept = default;
    ~file_status();

    // assignments
    file_status& operator=(const file_status&) noexcept = default;
    file_status& operator=(file_status&&) noexcept = default;

    // [fs.file_status.mods], modifiers
    void        type(file_type ft) noexcept;
    void        permissions(perms prms) noexcept;

    // [fs.file_status.obs], observers
    file_type   type() const noexcept;
    perms       permissions() const noexcept;
  };
}
```

[fs.file_status.cons]

~~explicit~~ `file_status(file_type ft, perms prms` ~~`= perms::unknown`~~`) noexcept;`

Ensures: `type() == ft` and `permissions() == prms`.

[re.regex]

```
namespace std {
  template<class charT, class traits = regex_traits<charT>>
    class basic_regex {
    public:
      // …
      // [re.regex.construct], construct/copy/destroy
      basic_regex();
      explicit basic_regex(const charT* p, flag_type f =
regex_constants::ECMAScript);
      explicit basic_regex(const charT* p) : basic_regex(p,
regex_constants::ECMAScript) { }
      basic_regex(const charT* p, size_t len, flag_type f =
regex_constants::ECMAScript);
      basic_regex(const basic_regex&);
      basic_regex(basic_regex&&) noexcept;
      template<class ST, class SA>
        explicit basic_regex(const basic_string<charT, ST, SA>& p,
                             flag_type f =
regex_constants::ECMAScript);
      template<class ST, class SA>
        explicit basic_regex(const basic_string<charT, ST, SA>& p) :
basic_regex(p, regex_constants::ECMAScript) { }
      template<class ForwardIterator>
        basic_regex(ForwardIterator first, ForwardIterator last,
                    flag_type f = regex_constants::ECMAScript);
      basic_regex(initializer_list<charT>, flag_type =
regex_constants::ECMAScript);

      ~basic_regex();

      basic_regex& operator=(const basic_regex&);
      basic_regex& operator=(basic_regex&&) noexcept;
      basic_regex& operator=(const charT* ptr);
      basic_regex& operator=(initializer_list<charT> il);
      template<class ST, class SA>
        basic_regex& operator=(const basic_string<charT, ST, SA>& p);

      // …
    };
}
```

[re.regex.construct]

```
explicit basic_regex(const charT* p, flag_type f = 
regex_constants::ECMAScript);
```

Requires: `p` shall not be a null pointer.

Throws: `regex_error` 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 array of `charT` of length `char_traits<charT>::length(p)` whose first element is designated by `p`, and interpreted according to the flags `f`.

Ensures: `flags()` returns `f`.

`mark_count()` returns the number of marked sub-expressions within the expression.

[…]

```
template<class ST, class SA>
  explicit basic_regex(const basic_string<charT, ST, SA>& s,
                       flag_type f = regex_constants::ECMAScript);
```

Throws: `regex_error` 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`.

Ensures: `flags()` returns `f`.

`mark_count()` returns the number of marked sub-expressions within the expression.

Acknowledgements

Thanks to Tim Song for the advice to use delegating constructors (as well as his original paper where I cribbed much of the discusssion from); Marshall Clow for prodding me to write this paper; Alisdair Meredith and Casey Carter for reviewing the changes made after discussing this at the LWG meeting in Batavia; Nico Josuttis for helping to formulate the policy; Carol Brown for the delicious baked goodies that kept us going all week in Batavia (and gave me the energy to write this paper). As always, thank them / blame me.

References

N4762 - Working Draft, Standard for Programming Language C++
P0935R0 - Eradicating unnecessarily explicit default constructors from the standard library – *Tim Song*
LWG2307 - Should the Standard Library use explicit only when necessary?
D1148R0a – Cleaning up Clause 20 – *Tim Song* (LWG Batavia 2018)