Doc no: N2679=08-0189
Date: 2008-06-13
Reply-To: Gabriel Dos Reis
gdr@cs.tamu.edu

Initializer Lists for Standard Containers (Revision 1)

Gabriel Dos Reis Bjarne Stroustrup

Texas A&M University

Abstract

This is a companion paper to the proposal *Initializer lists* (N2215=07-0075). We suggest modifications to the C++ Standard Library to take advantage of generalized initializer lists. Much of the rationale is discussed in that paper.

1 Clause 21: Strings library

Section §21.3 Modify the class template `basic_string` adding the following public member functions:

```
template<class charT, class traits = char_traits<charT>,
         class Allocator = allocator<charT>>
class basic_string {
//...
   basic_string(initializer_list<charT>,
                const Allocator& = Allocator());
   basic_string& operator=(initializer_list<charT>);
   basic_string& operator+=(initializer_list<charT>);
   basic_string& append(initializer_list<charT>);
   basic_string& assign(initializer_list<charT>);
   void insert(iterator, initializer_list<charT>);
   basic_string& replace(iterator, iterator,
                         initializer_list<charT>);
};
```

Section §21.3.2. Add the following paragraphs that describe the semantics of the sequence constructor, and assignment from initializer list:

```
basic_string(initializer_list<charT> s,
             const Allocator& a = Alloctor());
```

Effects: construct a string from the values in the range [`s.begin(),s.end()`) as indicated in the Sequence Requirements table (see 23.1.1).

```
basic_string& operator=(initializer_list<charT> s);
```

Effects: `*this = basic_string(s)` *Returns:* `*this`.

Section §21.3.6.1 Add the following paragraph that describes the semantics of the augmented assignment operator:

```
basic_string& operator+=(initializer_list<charT> s);
```

Returns: The result of `append(s)`.

Section §21.3.6.2 Add the following paragraph that describes the semantics of the `append` member functions:

```
basic_string& append(initializer_list<charT> s);
```

Returns: `append(basic_string<charT, traits, Allocator>(s))`.

Section §21.3.6.3 Add the following paragraph that describes the semantics of the `assign` member functions:

```
basic_string& assign(initializer_list<charT> s);
```

Returns: `assign(basic_string<charT, traits, Allocator>(s))`.

Section §21.3.6.4 Add the following paragraph that describes the semantics of the `insert` member functions:

```
void insert(iterator p, initializer_list<charT> s);
```

Effects: `insert(p, s.begin(), s.end())`.

Section §21.3.6.6 Add the following paragraph that describes the semantics of the `replace` member functions:

```
basic_string& replace(iterator i1, iterator i2,
                      initializer_list<charT> s);
```

Returns: `replace(i1, i2, s.begin(), s.end())`.

2 Clause 23: Containers library

We suggest that all container constructors accepting pairs of input iterators, all container member functions accepting pairs of input iterators be overloaded to accept initializer lists.

Section §23.2.1. The class template `array`, by design, already takes initializer list; so no further modification is proposed here.

Section §23.2.2. Add a sequence constructor to the class template `deque`, along with overloads for assignment operator, `assign`, and `insert` member functions:

```
template<class T, class Allocator = allocator<T>>
class deque {
  //...
  deque(initializer_list<T>,
        const Allocator& = Allocator());
  deque& operator=(initializer_list<T>);
  void assign(initializer_list<T>);
  void insert(const_iterator, initializer_list<T>);
};
```

Section §23.2.2.1. Add the following paragraphs:

```
deque(initializer_list<T> s,
      const Allocator& a = Allocator());
```

Effects: Construct a `deque` equal to `deque(s.begin(), s.end(), a)`.

Complexity: Make `s.size()` calls to copy constructor of `T`.

```
void assign(initializer_list<T> s);
```

Effects: `assign(s.begin(), s.end())`.

```
deque& operator=(initializer_list<T> s);
```

Effects: `assign(s)`.

```
void insert(const_iterator p, initializer_list<T> s);
```

Effects: `insert(p, s.begin(), s.end())`.

Section §23.2.3. Add a sequence constructor to the class template `forward_list`, along with overloads for assignment operator, `assign`, and `insert_after` member functions:

```
template<class T, class Allocator = allocator<T>>
class foward_list {
   //...
   forward_list(initializer_list<T>,
        const Allocator& = Allocator());
   void assign(initializer_list<T>);
   forward_list& operator=(initializer_list<T>);
   void insert_after(const_iterator, initializer_list<T>);
};
```

Section §23.2.3.1. Add the following paragraphs:

```
foward_list(initializer_list<T> s,
            const Allocator& a = Allocator());
```

Effects: Construct a `forward_list` equal to `forward_list(s.begin(),s.end(),a)`.

Complexity: Make `s.size()` calls to copy constructor of `T`.

```
void assign(initializer_list<T> s);
```

Effects: `assign(s.begin(), s.end())`.

```
forward_list& operator=(initializer_list<T> s);
```

Effects: `assign(s)`.

Returns: `*this`

Section §23.2.3.3. Add the following paragraph:

```
void insert_after(const_iterator p, initializer_list<T> s);
```

Effects: `insert_after(p, s.begin(), s.end())`.

Section §23.2.4. Add a sequence constructor to the class template `list`, along overloads for assignment operator, `assign`, and `insert` member functions:

```
template<class T, class Allocator = allocator<T>>
class list {
   //...
   list(initializer_list<T>,
        const Allocator& = Allocator());
   void assign(initializer_list<T>);
   list& operator=(initializer_list<T>);
   void insert(const_iterator, initializer_list<T>);
};
```

Section §23.2.4.1. Add the following paragraphs:

```
list(initializer_list<T> s,
     const Allocator& a = Allocator());
```

Effects: Construct a `list` equal to `list(s.begin(), s.end(), a)`.

Complexity: Make `s.size()` calls to copy constructor of `T`.

```
void assign(initializer_list<T> s);
```

Effects: `assign(s.begin(), s.end())`.

```
list& operator=(initializer_list<T> s);
```

Effects: `assign(s)`.

Returns: `*this`

Section §23.2.4.3. Add the following paragraph:

```
void insert(const_iterator p, initializer_list<T> s);
```

Effects: `insert(p, s.begin(), s.end())`.

Section §23.2.5. No proposed change to container adaptors.

Add a sequence constructor to the class template `vector`, along overloads for assignment operator, `assign`, and `insert` member functions:

```
template<class T, class Allocator = allocator<T>>
class vector {
   //...
   vector(initializer_list<T>,
          const Allocator& = Allocator());
   void assign(initializer_list<T>);
   vector& operator=(initializer_list<T>);
   void insert(const_iterator, initializer_list<T>);
};
```

Section §23.2.6.1. Add the following paragraphs:

```
vector(initializer_list<T> s,
       const Allocator& a = Allocator());
```

Effects: Construct a `vector` equal to `vector(s.begin(), s.end(), a)`.

Complexity: Make `s.size()` calls to copy constructor of `T`.

```
void assign(initializer_list<T> s);
```

Effects: `assign(s.begin(), s.end())`.

```
vector& operator=(initializer_list<T> s);
```

Effects: `assign(s)`.

Section §23.2.6.4. Add the following paragraph:

```
void insert(const_iterator p, initializer_list<T> s);
```

Effects: `insert(p, s.begin(), s.end())`.

Add a sequence constructor to the class template `vector<bool>`, along overloads for assignment operator, `assign`, and `insert` member functions:

```
template<class Allocator>
class vector<bool,Allocator> {
   //...
   vector(initializer_list<bool>,
```

```
        const Allocator& = Allocator());
  void assign(initializer_list<bool>);
  vector& operator=(initializer_list<bool>);
  void insert(const_iterator, initializer_list<bool>);
};
```

Note to LWG: the semantics description are inherited from general provisions for the primary template `vector`.

Section §23.3.1. Add a sequence constructor to the class template `map`, along with overloads for assignment operator, and `insert`:

```
template<class Key, class T, class Compare = less<Key>,
         class Allocator = allocator<pair<const Key, T>>>
class map {
  //...
  map(initializer_list<value_type>,
      const Compare& = Compare(),
      const Allocator& = Allocator());
  map& operator=(initializer_list<value_type>);
  void insert(initializer_list<value_type>);
};
```

Section §23.3.1.1. Add the following paragraphs:

```
map(initializer_list<value_type> s,
    const Compare& comp = Compare(),
    const Allocator& a = Allocator());
```

Effects: Constructs an empty `map` using the specified comparison object and allocator, and inserts elements from [`s.begin()`,`s.end()`).

Complexity: Linear in N if the range [`s.begin()`,`s.end()`) is already sorted using `comp`, and otherwise $N \log N$, where N is `s.size()`.

```
map& operator=(initializer_list<value_type> s);
```

Effects: `*this = map(s)`.

Returns: `*this`

```
void insert(initializer_list<value_type> s);
```

Effects: `insert(s.begin(), s.end())`.

Section §23.3.2. Add a sequence constructor to the class template `multimap`, along with new assignment operator, and overload of `insert`:

```
template<class Key, class T, class Compare = less<Key>,
         class Allocator = allocator<pair<const Key, T>>>
class multimap {
   //...
   multimap(initializer_list<value_type>,
            const Compare& = Compare(),
            const Allocator& = Allocator());
   multimap& operator=(initializer_list<value_type>);
   void insert(initializer_list<value_type>);
};
```

Section §23.3.2.1. Add the following paragraphs:

```
multimap(initializer_list<value_type> s,
         const Compare& comp = Compare(),
         const Allocator& a = Allocator());
```

Effects: Constructs an empty `multimap` using the specified comparison object and allocator, and inserts elements from [`s.begin()`,`s.end()`).

Complexity: Linear in N if the range [`s.begin()`,`s.end()`) is already sorted using `comp`, and otherwise $N \log N$, where N is `s.size()`.

```
multimap& operator=(initializer_list<value_type> s);
```

Effects: `*this = multimap(s)`.

Returns: `*this`.

```
void insert(initializer_list<value_type> s);
```

Effects: `insert(s.begin(), s.end())`.

Section §23.3.3. Add a sequence constructor to the class template `set`, along with new assignment operator, and overload of `insert`:

```
template<class Key, class T, class Compare = less<Key>,
         class Allocator = allocator<pair<const Key, T>>>
class set {
   //...
   set(initializer_list<value_type>,
       const Compare& = Compare(),
```

```
        const Allocator& = Allocator());
    set& operator=(initializer_list<value_type>);
    void insert(initializer_list<value_type>);
};
```

Section §23.3.3.1. Add the following paragraphs:

```
set(initializer_list<value_type> s,
    const Compare& comp = Compare(),
    const Allocator& a = Allocator());
```

Effects: Constructs an empty `set` using the specified comparison object and allocator, and inserts elements from [`s.begin()`,`s.end()`).

Complexity: Linear in N if the range [`s.begin()`,`s.end()`) is already sorted using `comp`, and otherwise $N \log N$, where N is `s.size()`.

```
set& operator=(initializer_list<value_type> s);
```

Effects: `*this = set(s)`. *Returns:* `*this`.

```
void insert(initializer_list<value_type> s);
```

Effects: `insert(s.begin(), s.end())`.

Section §23.3.4. Add a sequence constructor to the class template `multiset`, along with overloads for assignment operator, and `insert`:

```
template<class Key, class T, class Compare = less<Key>,
         class Allocator = allocator<pair<const Key, T>>>
class multiset {
   //...
   multiset(initializer_list<value_type>,
            const Compare& = Compare(),
            const Allocator& = Allocator());
   multiset& operator=(initializer_list<value_type>);
   void insert(initializer_list<value_type>);
};
```

Section §23.3.4.1. Add the following paragraphs:

```
multiset(initializer_list<value_type> s,
         const Compare& comp = Compare(),
         const Allocator& a = Allocator());
```

Effects: Constructs an empty `multiset` using the specified comparison object and allocator, and inserts elements from [`s.begin()`,`s.end()`).

Complexity: Linear in N if the range [`s.begin()`,`s.end()`) is already sorted using `comp`, and otherwise $N \log N$, where N is `s.size()`.

```
multiset& operator=(initializer_list<value_type> s);
```

Effects: `*this = set(s)`.

Returns: `*this`.

```
void insert(initializer_list<value_type> s);
```

Effects: `insert(s.begin(), s.end())`.

Section §23.3.5. No proposed change to the class template `bitset`

Section §23.4.1. Add a sequence constructor to the class template `unordered_map`, along with overloads for assignment operator, and `insert`:

```
template<class Key, class T, class Compare = less<Key>,
         class Allocator = allocator<pair<const Key, T>>>
class unordered_map {
   //...
   unordered_map(initializer_list<value_type>,
                 size_type = implementation-defined,
                 const hasher& = hasher(),
                 const key_equal& = key_equal(),
                 const Allocator& = Allocator());
   unordered_map& operator=(initializer_list<value_type>);
   void insert(initializer_list<value_type>);
};
```

Section §23.4.1.1. Add the following paragraphs:

```
unordered_map(initializer_list<value_type> s,
              size_type n = implementation-defined,
              const hasher& h = hasher(),
              const key_equal& k = key_equal(),
              const Allocator& a = Allocator());
```

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 `[s.benin(),s.end())`. `max_load_factor()` returns 1.0.

```
unordered_map& operator=(initializer_list<value_type> s);
```

Effects: `*this = unordered_map(s)`. *Returns:* `*this`.

```
void insert(initializer_list<value_type> s);
```

Effects: `insert(s.begin(), s.end())`.

Section §23.4.2. Add a sequence constructor to the class template `unordered_multimap`, along with overloads for assignment operator, and `insert`:

```
template<class Key, class T, class Compare = less<Key>,
         class Allocator = allocator<pair<const Key, T>>>
class unordered_multimap {
   //...
   unordered_multimap(initializer_list<value_type>,
                      size_type = implementation-defined,
                      const hasher& = hasher(),
                      const key_equal& = key_equal(),
                      const Allocator& = Allocator());
   unordered_multimap& operator=(initializer_list<value_type>);
   void insert(initializer_list<value_type>);
};
```

Section §23.4.2.1. Add the following paragraphs:

```
unordered_multimap(initializer_list<value_type> s,
                   size_type n = implementation-defined,
                   const hasher& h = hasher(),
                   const key_equal& k = key_equal(),
                   const Allocator& a = Allocator());
```

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 `[s.benin(),s.end())`. `max_load_factor()` returns 1.0.

```
unordered_multimap& operator=(initializer_list<value_type> s);
```

Effects: `*this = unordered_multimap(s).`

Returns: `*this.`

```
void insert(initializer_list<value_type> s);
```

Effects: `insert(s.begin(), s.end()).`

Section §23.4.3. Add a sequence constructor to the class template `unordered_set`, along with new assignment operator, and overload of `insert`:

```
template<class Key, class T, class Compare = less<Key>,
         class Allocator = allocator<pair<const Key, T>>>
class unordered_set {
   //...
   unordered_set(initializer_list<value_type>,
                 size_type = implementation-defined,
                 const hasher& = hasher(),
                 const key_equal& = key_equal(),
                 const Allocator& = Allocator());
   unordered_set& operator=(initializer_list<value_type>);
   void insert(initializer_list<value_type>);
};
```

Section §23.4.3.1. Add the following paragraphs:

```
unordered_set(initializer_list<value_type> s,
              size_type n = implementation-defined,
              const hasher& h = hasher(),
              const key_equal& k = key_equal(),
              const Allocator& a = Allocator());
```

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 `[s.benin(),s.end())`. `max_load_factor()` returns 1.0.

```
unordered_set& operator=(initializer_list<value_type> s);
```

Effects: `*this = unordered_set(s).`

Returns: `*this.`

```
void insert(initializer_list<value_type> s);
```

Effects: `insert(s.begin(), s.end())`.

Section §23.4.4. Add a sequence constructor to the class template `unordered_multiset`, along overloads for assignment operator, and `insert`:

```
template<class Key, class T, class Compare = less<Key>,
         class Allocator = allocator<pair<const Key, T>>>
class unordered_multiset {
   //...
   unordered_multiset(initializer_list<value_type>,
                      size_type = implementation-defined,
                      const hasher& = hasher(),
                      const key_equal& = key_equal(),
                      const Allocator& = Allocator());
   unordered_multiset& operator=(initializer_list<value_type>);
   void insert(initializer_list<value_type>);
};
```

Section §23.4.4.1. Add the following paragraphs:

```
unordered_multiset(initializer_list<value_type> s,
                   size_type n = implementation-defined,
                   const hasher& h = hasher(),
                   const key_equal& k = key_equal(),
                   const Allocator& a = Allocator());
```

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 `[s.benin(),s.end())`. `max_load_factor()` returns 1.0.

```
unordered_multiset& operator=(initializer_list<value_type> s);
```

Effects: `*this = unordered_multiset(s)`.

Returns: `*this`.

```
void insert(initializer_list<value_type> s);
```

Effects: `insert(s.begin(), s.end())`.

3 Clause 25: Algorithms library

We do not propose any change at this moment. However, we do recommend that if overloads for algorithms on containers are added, then the non-mutating algorithms must also be added for `initializer_list`.

4 Clause 26: Numerics library

Section §26.5.2. Add a sequence constructor to the class template `valarray`, along with assignment operator from initializer list:

```
template<class T>
class valarray {
   // ...
   valarray(initializer_list<T>);
   valarray& operator=(initializer_list<T>);
};
```

Section §26.5.2.1. Add the following paragraph

```
valarray(initializer_list<T> s);
```

Effects: Same as `valarray(s.begin(), s.size())`.

Section §26.5.2.1. Add the following paragraph

```
valarray& operator=(initializer_list<T> s);
```

Effects: Same as `*this = valarray(s)`.

Returns: `*this`.

5 Clause 28: Regular expressions library

Add a sequence constructor to the class template `basic_regex`

```
template<class charT,
         class traits = regex_traits<charT>>
class basic_regex {
   // ...
```

```
    basic_regex(initializer_list<charT>,
                flag_type = regex_constants::ECMAScript);
    basic_regex& assign(initializer_list<charT>,
                        flag_type = regex_constants::ECMAScript);
};
```

Section §28.8.2. Add the following paragraph

```
basic_regex(initializer_list<charT> s,
            flag_type f = regex_constants::ECMAScript);
```

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 `[s.begin(),s.end())`, and interpreted according to the flags specified in f.

Section §28.8.3. Add the following paragraph

```
basic_regex&
assign(initializer_list<charT> s,
       flag_type f = regex_constants::ECMAScript);
```

Effects: Same as `assign(s.begin(), s.end(), f)`. *Returns:* `*this`.