Doc no: N2220=07-0080
Date: 2007-03-11
Reply-To: Gabriel Dos Reis
gdr@cs.tamu.edu

Initializer Lists for Standard Containers

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: constructs a string from the values in range `[s.begin(),s.end())` [...]

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

Returns: `*this = basic_string(s)`

Section §23.3.6.1 Add the following paragraph that describes the semantics of the augmented assignment operator:

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

Returns: `append(s).`

Section §23.3.6.2 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 §23.3.6.3 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 §23.3.6.4 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 §23.3.6.6 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 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(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(iterator p, initializer_list<T> s);
```

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

Section §23.2.3. Add the following member functions to those listed in paragraph §23.2.3/2:

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

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(iterator, initializer_list<T>);
};
```

Section §23.2.3.1. Add the following paragraphs:

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

Effects: Construct a `list` 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())`.

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

Effects: `assign(s)`.

Section §23.2.3.3. Add the following paragraph:

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

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

Section §23.2.4. 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(iterator, initializer_list<T>);
};
```

Section §23.2.5.1. Add the following paragraphs:

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

Effects: Construct a `vector` 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())`.

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

Effects: `assign(s)`.

Section §23.2.5.4. Add the following paragraph:

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

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

Section §23.2.6. We make no suggestion to change `vector<bool>`.

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<T>);
};
```

Section §23.3.1.1. Add the following paragraphs:

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

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

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

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

```
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<T>);
};
```

Section §23.3.2.1. Add the following paragraphs:

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

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

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

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

```
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<T>);
};
```

Section §23.3.3.1. Add the following paragraphs:

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

Effects: Same as `set(s.begin(), s.end(), comp, a)`.

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

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

```
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<T>);
};
```

Section §23.3.4.1. Add the following paragraphs:

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

Effects: Same as `multiset(s.begin(), s.end(), comp, a)`.

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

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

```
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<T>);
};
```

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: Same as `unordered_map(s.begin(), s.end(), n, h, k, a)`.

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

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

```
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<T>);
};
```

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: Same as `unordered_multimap(s.begin(), s.end(), n, h, k, a)`.

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

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

```
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<T>);
};
```

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: Same as `unordered_set(s.begin(), s.end(), n, h, k, a)`.

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

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

```
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<T>);
};
```

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: Same as `unordered_multiset(s.begin(), s.end(), n, h, k, a)`.

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

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

```
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);
```

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

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: Same as `basic_regex(s.begin(), s.end(), s)`.

Section §28.8.3. Add the following paragraph

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

Returns: Same as `assign(s.begin(), s.end(), s)`.