Doc No: N3173=10-0163
Date: 2010-10-17
Author: Pablo Halpern
Intel, Corp.

phalpern@halpernwightsoftware.com

Terminology for constructing container elements (US115)

Contents

National Body comments and issues 1
Document Conventions 1
Discussion 1
Proposed Wording 2
Example changes 6
References 8

National Body comments and issues

This paper proposes a complete resolution for comment US 115 to the July, 2010 FCD.

Document Conventions

All section names and numbers are relative to the August 2010 WP, N3126.

Existing working paper text is indented and shown in dark blue. Edits to the working paper are shown with ~~red strikeouts for deleted text~~ and green underlining for inserted text within the indented blue original text.

Comments and rationale mixed in with the proposed wording appears as shaded text.

Requests for LWG opinions and guidance appear with light (yellow) shading. It is expected that changes resulting from such guidance will be minor and will not delay acceptance of this proposal in the same meeting at which it is presented.

Discussion

The resolution to LWG issue 704 (http://www.open-std.org/JTC1/SC22/WG21/docs/lwg-defects.html#704) added a number of requirements to the container operations in order to properly constrain the elements of the containers. Unfortunately, the current wording effectively *redefines* the terms CopyConstructible and MoveConstructible and the phrase "constructible with *args*" so that they have different meanings in the containers section than in

the rest of the standard. This use of terminology is not only confusing and vague, it is also not applied correctly through the section. There are some cases, in fact, when the terms CopyConstructible and MoveConstructible are used with their original meanings, but the reader would have no way to know that.

The best solution is to choose an entirely new and more precise set of terms and apply them consistently and correctly in the containers section. This paper presents such a solution. It expands on the partial resolution presented in the Additional Details section for US 115. The suggested terms in this proposal differ from the terms suggested in the Additional Details section for US 115. The following table shows the mapping between the terms used in the current WP, the terms proposed in this paper, and (for reference), the terms originally proposed in US 115:

Current WP/FCD	US 115	Proposed
T is CopyConstructible	X can copy-insert T	T is CopyInsertable into X
T is MoveConstructible	X can move-insert T	T is MoveInsertable into X
T is constructible from args	X can construct-insert T from args	T is EmplaceConstructible into X from args

Proposed Wording

Modify the following rows of 23.2.1, [container.requirements.general] Table 93:

Table 93 – Container requirements

expression	return type	operational semantics	assertion/note pre/post-condition	complexity
...				
`X(a);`			*Requires:* T is ~~`CopyConstructible`~~ `CopyInsertable` into `X` (see below). `a == X(a)`	linear

`X u(a);`
 	
`X u = a;`
			Requires: T is ~~`CopyConstructible`~~ `CopyInsertable` into `X` (see below). post: `u == a;`	linear

expression	return type	operational semantics	assertion/note pre/post-condition	complexity

`X u(rv);`
 	
`X u = rv;`
			~~*Requires:* T is `MoveConstructible`.~~
			post: u shall be equal to the value that `rv` had before this construction.
				(Note B)

I was hoping to avoid making a forward reference to the new term `CopyInsertable`, but could not find a way to do it without rewriting the entire section.

With the exception of `array`, most containers do not require that their elements be `MoveConstructible` in order to do a move. Thus, the last row above does not replace the MoveConstructible requirement with the new MoveInsertable requirement.

Replace 23.2.1 [container.requirements.general], paragraph 15 as follows:

15 ~~The descriptions of the requirements of the type T in this section use the terms CopyConstructible, MoveConstructible, constructible from *i, and constructible from args. These terms are equivalent to the following expression using the appropriate arguments:~~

~~`allocator_traits<allocator_type>::contruct(x.get_allocator(), q, args…);`~~

~~where x is a non-const lvalue of some container type X and q has type X::value_type*. [*Example*: The container is going to move construct a T, so will call:~~

~~`allocator_traits<allocator_type>::construct(x.get_allocator(), q, std::move(t));`~~

~~The default implementation of construct will call:~~

~~`::new (q) T(std::forward<T>(t)); // where forward is the same as move here, cast to rvalue`~~

~~But the allocator author may override the above definition of construct and do the construction of T by some other means. — *end example*]~~

Given a container type X having an `allocator_type` of A and a `value_type` of T and given an lvalue `m` of type A, a pointer `p` of type `T*`, a value v of type T, or a value `rv` of type rvalue-of-T, the following terms are defined. (If X is not allocator-aware, the terms below are defined as if A were `std::allocator<T>`.):

T *is* `CopyInsertable` *into* X means that the following expression is well-formed:

```
allocator_traits<A>::contruct(m, p, v);
```

T *is* `MoveInsertable` *into* X means that the following expression is well-formed:

```
allocator_traits<A>::contruct(m, p, rv);
```

T *is* `EmplaceConstructible` *into* X *from args*, for zero or more arguments, *args*, means that the following expression is well-formed:

```
allocator_traits<A>::contruct(m, p, args);
```

[*Note:* A container calls `allocator_traits<A>::contruct(m, p, args)` to construct an element at `p` using *args*. The default of `contruct` in `std::allocator` will call `::new((void*) p) T(`*args*`)` but specialized allocators may choose a different definition. – *end note*]

In section 23.2 [container.requirements], starting at Table 96, make the following text replacements. Those entries marked *do not change* are exceptions to the other rules:

Original text, in WP	**Replacement text**
`key_compare` is `CopyConstructible`	*do not change*
`hasher` is `CopyConstructible`	*do not change*
`hasher` and `key_equal` are `CopyConstructible`	*do not change*
`CopyConstructible`	`CopyInsertable` into X
`MoveConstructible`	`MoveInsertable` into X
shall be constructible from	shall be `EmplaceConstructible` into X from
is constructible from	is `EmplaceConstructible` into X from

I have checked that no incorrect replacements will result if these rules are applied only to section 23.2, starting at Table 96. Specifically, the following tables are affected:

- 23.2.1 [container.requirements.general], Table 96 (shown below)
- 23.2.3 [sequence.reqmts], Table 97 (shown below)
- 23.2.3 [sequence.reqmts], Table 98
- 23.2.4 [associative.reqmts], Table 99. **Note to editor**: several entries in Table 99 use the identifier T. I believe all of them should be `value_type`.
- 23.2.5 [unord.req], Table 100. **Note to editor**: several entries in Table 100 use the identifier T. I believe all of them should be `value_type`.

(A couple of example tables can be found in the **Example Changes** section, below, to illustrate the replacements.)

In sections 23.3 [sequences] through 23.4.1 [vector], make the following text substitutions:

Original text, in WP	**Replacement text**

`non-CopyConstructible`	`non-CopyInsertable`
`CopyConstructible`	`CopyInsertable` into `*this`

I have checked that no incorrect replacements will result if these rules are applied only to sections 23.3 through 24.4.1. Specifically, the following paragraphs are affected:

- 23.3.2.1 [deque.cons]/7 (constructor)
- 23.3.2.2 [deque.capacity]/4 (resize)
- 23.3.2.3 [deque.modifiers]/2 (push_back, etc.)
- 23.3.3.1 [forwardlist.cons]/7 (constructor)
- 23.3.4.1 [list.cons]/7 (constructor)
- 23.3.4.2 [list.capacity]/4 (resize)
- 23.4.1.1 [vector.cons]/7 (constructor)
- 23.4.1.2 [vector.capacity]/2 (reserve) and /12 (resize). To make this readable, use "non-`CopyInsertable`" instead of "non-`CopyInsertable` into X."
- 23.4.1.2 [vector.capacity]/10
- 23.4.1.4 [vector.modifiers]/1 (push_back et al.). To make this readable, use "non-`CopyInsertable`" instead of "non-`CopyInsertable` into X."

Note that no changes are needed for the associative and unordered containers. Although these containers make reference to CopyConstructible and MoveConstructible, they are in the context of constructing a temporary value_type object before inserting it into the container.

In section 23.3.1.1 [array.cons], Add move requirements for `array`:

23.3.1.1 array constructors, copy, and assignment [array.cons]

The conditions for an aggregate (8.5.1) shall be met. Class array relies on the implicitly-declared special member functions (12.1, 12.4, and 12.8) to conform to the container requirements table in 23.2. In addition to the requirements specified in the container requirements table, the implicit move constructor and move assignment operator for `array` require that T be `MoveConstructible` or `MoveAssignable`, respectively.

If we eliminate implicitly-defined move operations, will `array` lose its move constructor and move-assignment operator? How will the above paragraph be affected?

Example changes

The changes listed below are covered by the general search-and-replace rules above. They are explicitly shown here as examples to illustrate the nature of the change.

In section 23.2.1 [container.requirements.general], Table 96, modify the third column of selected rows as follows:

Table 96 – Allocator-aware container requirements

expression	return type	assertion/note pre/post-condition	complexity
…			

`X(t,m)`
`X u(t,m);`
		Requires: ~~`CopyConstructible`~~ `CopyInsertable` into `X`.
		`post: u == a,`
		`get_allocator() == m`
			linear

…			

`X(rv,m)`
 	
`X u(rv,m);`
		requires: T shall be ~~`MoveConstructible`~~ `MoveInsertable` into `X`.
		
		post: `u` shall have the same elements, or copies of the elements, that `rv` had before this construction,
		`get_allocator() == m`
			constant if `m == rv.get_allocator()`, else linear

`a = t`	`X&`	*Requires:* T is ~~`CopyConstructible`~~ `CopyInsertable` into `X` and `CopyAssignable`. post: `a == t`	linear
`a = rv`	`X&`	*Requires*: If `allocator_traits<allocator_type>::propagate_on_container_move_assignment::value` is `false`, T is ~~`MoveConstructible`~~ `MoveInsertable` into `X` and `MoveAssignable`. All existing elements of a are either move assigned to or destroyed. post: a shall be equal to the value that `rv` had before this assignment.	linear

In section 23.2.3 [sequence.reqmts], modify rows of table 97 as follows:

Table 97 – Sequence container requirements (in addition to container)

Expression	Return type	Assertion/note pre-/post-condition
`X(n, t)` `X a(n, t)`		*Requires:* T shall be ~~CopyConstructible~~ `CopyInsertable` into `X`. post: `distance(begin(),end())== n` Constructs a sequence container with n copies of t
`X(i, j)` `X a(i, j)`		*Requires*: T shall be ~~constructible~~ `EmplaceConstructible` into X from *i. For `vector`, if the iterator does not meet the forward iterator requirements (24.2.5), T shall also be ~~MoveConstructible~~ `MoveInsertable` into X. Each iterator in the range `[i,j)` shall be dereferenced exactly once. post: distance `(begin(), end()) == distance(i, j)` Constructs a sequence container equal to the range `[i, j)`
`X(il);`		Equivalent to `X(il.begin(),il.end())`
`a = il;`	`X&`	*Requires*: T is ~~CopyConstructible~~ `CopyInsertable` into X and `CopyAssignable`. Assigns the range `[il.begin(),il.end())` into `a`. All existing elements of `a` are either assigned to or destroyed. *Returns*: `*this`.
`a.emplace(p,args)`	`iterator`	*Requires*: T is ~~constructible~~ `EmplaceConstructible` into X from `args`. For `vector` and `deque`, T is also ~~MoveConstructible~~ `MoveInsertable` and `MoveAssignable`. *Effects*: Inserts an object of type T constructed with `std::forward<Args>(args)...` before `p`.
`a.insert(p,t)`	`iterator`	*Requires*: T shall be ~~CopyConstructible~~ `CopyInsertable` into `X`. For `vector` and `deque`, T shall also be `CopyAssignable`. *Effects*: Inserts a copy of t before p.
`a.insert(p,rv)`	`iterator`	*Requires*: T shall be ~~MoveConstructible~~ `MoveInsertable` into X. For vector and deque, T shall also be `MoveAssignable`. *Effects*: Inserts a copy of `rv` before `p`.
`a.insert(p,n,t)`	`iterator`	*Requires*: T shall be ~~CopyConstructible~~ `CopyInsertable` into X and `CopyAssignable`. *Effects:* Inserts `n` copies of `t` before `p`.

`a.insert(p,i,j)`	`iterator`	*Requires:* T shall be ~~constructible~~ `EmplaceConstructible` into X from `*i`. For `vector`, if the iterator does not meet the forward iterator requirements (24.2.5), T shall also be ~~MoveConstructible~~ `MoveInsertable` into X and MoveAssignable. Each iterator in the range `[i,j)` shall be dereferenced exactly once. pre: `i` and `j` are not iterators into `a`. *Effects:* Inserts copies of elements in [i, j) before p
`a.insert(p, il);`	`iterator`	`a.insert(p,il.begin(),il.end())`

…

`a.assign(i,j)`	`void`	*Requires*: T shall be ~~constructible~~ `EmplaceConstructible` into X from *i and assignable from *i. For `vector`, if the iterator does not meet the forward iterator requirements (24.2.5), T shall also be ~~MoveConstructible~~ `MoveInsertable` into X. Each iterator in the range `[i,j)` shall be dereferenced exactly once. pre: `i, j` are not iterators into a. *Effects:* Replaces elements in `a` with a copy of `[i, j)`.
`a.assign(il)`	`void`	`a.assign(il.begin(),il.end()).`
`a.assign(n,t)`	`void`	*Requires*: T shall be ~~CopyConstructible~~ `CopyInsertable` into X and `CopyAssignable`. pre: t is not a reference into a. *Effects:* Replaces elements in a with n copies of t.

Editorial note: This table is inconsistent in its used of the *Effects* label. I added them as I ran across them, but they are unimportant to the content of this proposal.

References

N3102: ISO/IEC FCD 14882, C++0X, National Body Comments