Doc No: N3346=12-0036
Date: 2012-01-14
Author: Pablo Halpern
Intel, Corp.

phalpern@halpernwightsoftware.com

Defect Report: Terminology for Container Element Requirements - Rev 1

Contents

Document Conventions

All section names and numbers are relative to the April 2011 FDIS, N3290.

> 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.

National Body comments and issues

This defect report describes an omission in N3173, which resolved comment US 115 to the July, 2010 FCD. The proposed wording in this paper interacts with the resolution of LWG 2033. The wording here assumes that the resolution of LWG 2033 has been applied.

Changes from N3301

- Added the term *default-insertion* to replace *value-initialization* in the WP.
- Added the term `Erasable` to replace `Destructible` in the containers section.
- Fixed typos
- Added `value_type` requirements for associative and unordered containers.

Description of Defect

Adoption of N3173 corrected the misuse of the terms CopyConstructible and MoveConstructible and the phrase "constructible with *args*" in the containers section of the FCD. Unfortunately, the paper missed a few incorrect uses of CopyConstructible and failed to correct similar misuses of the terms DefaultConstructible and Destructible. These errors persist now in the IS and should be corrected by a TC.

The nature of the terminology misuse is that elements of a container are never constructed or destructed directly within the container (except in the case of `array`), but rather are constructed by calling the `construct` member function of the container's allocator and destructed by calling the `destroy` member function of the container's allocator. The allocator is not required to call the element's constructor with exactly the list of arguments supplied to `construct`. The `scoped_allocator_adaptor` is an example of an allocator that modifies the `construct` argument list before calling the element's constructor. Thus, saying that a container's `value_type` is DefaultConstructible is neither necessary nor sufficient for specifying the requirements on that type. The proposed wording below defines precise replacements for the terms DefaultConstructible and Destructible in the containers section just as N3173 did for CopyConstructible and MoveConstructible. The wording also replaces any incorrect uses of DefaultConstructible and Destructible with the new terms and corrects some remaining incorrect uses of CopyConstructible.

Proposed Resolution (formal wording)

Note to the Editor: It is probably easiest to apply the PR of LWG 2033 to the WP before applying these changes, since some of the global search-and-replace will affect text in LWG 2033.

1. Modify the first row of Table 96 in section 23.2.1 [container.requirements.general] as follows:

Expression	Return Type	Operational Semantics	Assertion/note pre-/post-condition	Complexity
`X::value_type`	`T`		*Requires:* `T` is ~~`Destructible`~~`Erasable` from `X` (see [container.requirements.general], below)	compile time

Note to the Editor: The reference to [container.requirements.general] specifically refers to paragraph 13, but I understand that paragraph-level references are not used in the standard.

2. Add a thee new bullets to 23.2.1 [container.requirements.general], paragraph 13 as follows:

Given a container type X having an `allocator_type` identical to A and a `value_type` identical to T and given an lvalue `m` of type A, a pointer `p` of type `T*`, an expression v of type (possibly const) T, and an rvalue `rv` of type T, the following terms are defined. ~~(~~If X is not allocator-aware, the terms below are defined as if `A` were `std::allocator<T>`~~)~~ – no allocator object needs to be created and user specializations of `std::allocator<T>` are not instantiated:

— T is ***DefaultInsertable*** *into X* means that the following expression is well formed:

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

— An element of `X` is ***default-inserted*** if it is initialized by evaluation of the expression

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

where `p` is the address of the uninitialized storage for the element allocated within `X`.

One could argue that the terms *ValueInsertable* and *value-inserted* would be more consistent with the term *value-initialized* which they replace. However, I think it is easier to understand the terms *DefaultInsertable* and *default-inserted* because they typically invoke the default constructor.

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

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

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

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

— `T` is ***Erasable*** *from X* means that the following expression is well formed:

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

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

There are no incorrect uses of `DefaultConstructible`, `CopyConstructible`, `MoveConstructible`, or *constructible from* in section 23.2, including Tables 96 through Tables 103.

3. In section 23.2.4 [associative.reqmts], table 102, modify the top rows as follows:

Expression	Return Type	Assertion/note pre-/post-condition	Complexity
`X::key_type`	`Key`	~~*Requires:* Key is Destructible~~	compile time
`X::mapped_type` (`map` and `multimap` only)	`T`	~~*Requires:* T is Destructible~~	compile time
`X::value_type` (`set` and `multiset` only)	`Key`	*Requires:* `value_type` is `Erasable` from `X`	compile time
`X::value_type` (`map` and `multimap` only)	`pair<const Key, T>`	*Requires:* `value_type` is `Erasable` from `X`	compile time

4. In section 23.2.5 [unord.req], table 103, modify the top rows as follows:

Expression	Return Type	Assertion/note pre-/post-condition	Complexity
X::key_type	Key	~~*Requires:* Key shall be Destructible~~	compile time
X::mapped_type (unordered_map and unordered_multimap only)	T	~~*Requires:* T is Destructible~~	compile time
X::value_type (unordered_set and unordered_multiset only)	Key	*Requires:* value_type is Erasable from X	compile time
X::value_type (unordered_map and unordered_multimap only)	pair<const Key, T>	*Requires:* value_type is Erasable from X	compile time

5. In sections 23.3.3 [deque] through 23.5 [unord], make the following text replacements:

Original text, in FDIS	Replacement text
T shall be DefaultConstructible	T shall be DefaultInsertable into *this
value-initialized elements	default-inserted elements
key_type shall be CopyConstructible	key_type shall be CopyInsertable into *this
mapped_type shall be DefaultConstructible	mapped_type shall be DefaultInsertable into *this
mapped_type shall be CopyConstructible	mapped_type shall be CopyInsertable into *this
mapped_type shall be MoveConstructible	mapped_type shall be MoveInsertable into *this
Key shall be CopyConstructible	Key shall be CopyInsertable into *this
value_type is constructible from	value_type is EmplaceConstructible into *this from

Notes to the editor: The above are carefully selected phrases that can be used for global search-and-replace within the specified sections without accidentally making changes to correct uses of DefaultConstructible et. al.. Please ensure that the resolution of 2033 is applied before applying these changes, otherwise, the use of `DefaultConstructible` in that resolution will be incorrect.

6. Modify section 23.3.4.5 [forwardlist.modifiers], split paragraphs 27 and 28 into four paragraphs as follows:

`void resize(size_type sz);`

~~`void resize(size_type sz, const value_type& c);`~~

Effects: If `sz < distance(begin(), end())`, erases the last `distance(begin(), end()) - sz` elements from the list. Otherwise, inserts `sz - distance(begin(), end())` default-inserted elements at the end of the list. ~~For the first signature the inserted elements are value-initialized, and for the second signature they are copies of c.~~

Requires: `T` shall be `DefaultInsertable` into `*this`. ~~`DefaultConstructible` for the first form and it shall be `CopyInsertable` into `*this` for the second form.~~

`void resize(size_type sz, const value_type& c);`

Effects: If `sz < distance(begin(), end())`, erases the last `distance(begin(), end()) - sz` elements from the list. Otherwise, inserts `sz - distance(begin(), end())` such that each new element, `e`, is initialized by a method equivalent to calling `allocator_traits<allocator_type>::construct(get_allocator(), std::addressof(e), c).`

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

7. Fix section 23.3.6.3 [vector.capacity] paragraph 10 as shown:

void resize(size_type sz);

9 *Effects*: If `sz <= size()`, equivalent to `erase(begin() + sz, end());`. If `size() < sz`, appends `sz - size()` ~~value-initialized~~default-inserted elements to the sequence.

10 *Requires*: T shall be ~~Copy~~`MoveInsertable` into *this and `DefaultInsertable` into *this.

Note to the editor: The change to paragraph 10 supersedes a similar change proposed in the resolution of LWG 2033. The change to paragraph 9 is orthogonal to a separate change to the same paragraph in LWG 2033.

Separable issue: In 23.4.4.2 map constructor `map(first, last)`, has an incomplete *requires* clause. It describes what the requirement is *if* `*first` is `pair<key_type,mapped_type>` but doesn't say what requirement is otherwise. What should the requirement be? Does `*this` have to be a pair, or merely pair-like? What are the actual requirements on `first->first` and `first->second`? I believe that the requirement should be fairly broad but complex: the iterator's value type must have members `first` and `second`, where `key_type` is EmplaceConstructible into `*this` from `first->first` and `mapped_type` is EmplaceConstructible into *`this` from `first->second`. However, it might be sufficient and simplest to say that `value_type` is EmplaceConstructible into `*this` from `*first`. The same issue applies to the `insert` member 23.4.4.4 [map.modifiers]. In the latter case, the range insert version should probably be separated from the other two and each one's requirements precisely described (some use of forward<> might be needed). It is also confusing that the requirements for `insert` describes things that are *not* required. Same issue for multimap (23.4.5.3).

Separable issue: `operator[](key_type&&)` is missing a requirement that `key_type` be `MoveInsertable` into `*this`.

Acknowledgements

Thanks for Daniel Krugler for pointing out omissions and errors in N3301 and for helping me revise it.

References

N3301: Defect Report: Terminology for Container Element Requirements

N3290: Final Draft International Standard: Programming Languages C++, 2011-04-11

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

N3173: Terminology for constructing container elements

LWG 2033: Preconditions of `reserve`, `shrink_to_fit`, and `resize` functions