CTR51-CPP. Use valid references, pointers, and iterators to reference elements of a container

Iterators are a generalization of pointers that allow a C++ program to work with different data structures (containers) in a uniform manner [ISO/IEC 14882-2014]. Pointers, references, and iterators share a close relationship in which it is required that referencing values be done through a valid iterator, pointer, or reference. Storing an iterator, reference, or pointer to an element within a container for any length of time comes with a risk that the underlying container may be modified such that the stored iterator, pointer, or reference becomes invalid. For instance, when a sequence container such as `std::vector` requires an underlying reallocation, outstanding iterators, pointers, and references will be invalidated [Kalev 99]. Use only a valid pointer, reference, or iterator to refer to an element of a container.

The C++ Standard, [container.requirements.general], paragraph 12 [ISO/IEC 14882-2014] states the following:

Unless otherwise specified (either explicitly or by defining a function in terms of other functions), invoking a container member function or passing a container as an argument to a library function shall not invalidate iterators to, or change the values of, objects within that container.

The C++ Standard allows references and pointers to be invalidated independently for the same operation, which may result in an invalidated reference but not an invalidated pointer. However, relying on this distinction is insecure because the object pointed to by the pointer may be different than expected even if the pointer is valid. For instance, it is possible to retrieve a pointer to an element from a container, erase that element (invalidating references when destroying the underlying object), then insert a new element at the same location within the container causing the extant pointer to now point to a valid, but distinct object. Thus, any operation that invalidates a pointer or a reference should be treated as though it invalidates both pointers and references.

The following container functions can invalidate iterators, references, and pointers under certain circumstances.

<table>
<thead>
<tr>
<th>Class</th>
<th>Function</th>
<th>Iterators</th>
<th>References /Pointers</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>std::deque</code></td>
<td><code>insert()</code>, <code>emplace_front()</code>, <code>emplace_back()</code>, <code>emplace()</code>, <code>push_front()</code>, <code>push_back()</code></td>
<td>X</td>
<td>X</td>
<td>An insertion in the middle of the deque invalidates all the iterators and references to elements of the deque. ([deque.modifiers], paragraph 1)</td>
</tr>
<tr>
<td></td>
<td><code>erase()</code>, <code>pop_back()</code>, <code>resize()</code></td>
<td>X</td>
<td>X</td>
<td>An erase operation that erases the last element of a deque invalidates only the past-the-end iterator and all iterators and references to the erased elements. An erase operation that erases the first element of a deque but not the last element invalidates only the erased elements. An erase operation that erases neither the first element nor the last element of a deque invalidates the past-the-end iterator and all iterators and references to all the elements of the deque. ([deque.modifiers], paragraph 4)</td>
</tr>
<tr>
<td></td>
<td><code>clear()</code></td>
<td>X</td>
<td>X</td>
<td>Destroys all elements in the container. Invalidates all references, pointers, and iterators referring to the elements of the container and may invalidate the past-the-end iterator. ([sequence.reqmts], Table 100)</td>
</tr>
<tr>
<td><code>std::forward_list</code></td>
<td><code>erase_after()</code>, <code>pop_front()</code>, <code>resize()</code></td>
<td>X</td>
<td>X</td>
<td><code>erase_after</code> shall invalidate only iterators and references to the erased elements. ([forwardlist.modifiers], paragraph 1)</td>
</tr>
<tr>
<td></td>
<td><code>remove()</code>, <code>unique()</code></td>
<td>X</td>
<td>X</td>
<td>Invalidates only the iterators and references to the erased elements. ([forwardlist.ops], paragraph 12 &amp; paragraph 16)</td>
</tr>
<tr>
<td></td>
<td><code>clear()</code></td>
<td>X</td>
<td>X</td>
<td>Destroys all elements in the container. Invalidates all references, pointers, and iterators referring to the elements of the container and may invalidate the past-the-end iterator. ([sequence.reqmts], Table 100)</td>
</tr>
<tr>
<td><code>std::list</code></td>
<td><code>erase()</code>, <code>pop_front()</code>, <code>push_back()</code>, <code>clear()</code>, <code>remove_if()</code>, <code>remove()</code></td>
<td>X</td>
<td>X</td>
<td>Invalidates only the iterators and references to the erased elements. ([list.modifiers], paragraph 3 and [list.ops], paragraph 15 &amp; paragraph 19)</td>
</tr>
<tr>
<td></td>
<td><code>clear()</code></td>
<td>X</td>
<td>X</td>
<td>Destroys all elements in the container. Invalidates all references, pointers, and iterators referring to the elements of the container and may invalidate the past-the-end iterator. ([sequence.reqmts], Table 100)</td>
</tr>
<tr>
<td><code>std::vector</code></td>
<td><code>reserve()</code></td>
<td>X</td>
<td>X</td>
<td>After <code>reserve()</code>, <code>capacity()</code> is greater or equal to the argument of <code>reserve</code> if reallocation happens and is equal to the previous value of <code>capacity()</code> otherwise. Reallocation invalidates all the references, pointers, and iterators referring to the elements in the sequence. ([vector.capacity], paragraph 3 &amp; paragraph 6)</td>
</tr>
</tbody>
</table>
insert(), emplace_back(), emplace(), push_back() | X | X | Causes reallocation if the new size is greater than the old capacity. If no reallocation happens, all the iterators and references before the insertion point remain valid. ([vector.modifiers], paragraph 1). All iterators and references after the insertion point are invalidated.

erase(), pop_back(), resize() | X | X | Invalidates iterators and references at or after the point of the erase. ([vector.modifiers], paragraph 3)

clear() | X | X | Destroys all elements in the container. Invalidates all references, pointers, and iterators referring to the elements of the container and may invalidate the past-the-end iterator. ([sequence.reqmts], Table 100)

std::set, std::multiset, std::map, std::multimap

erase(), clear() | X | X | Invalidates only iterators and references to the erased elements. ([associative.reqmts], paragraph 9)

std::unordered_set, std::unordered_multiset, std::unordered_map, std::unordered_multimap

erase(), clear() | X | X | Invalidates only iterators and references to the erased elements. ([unord.req], paragraph 14)

insert(), emplace() | X | The insert and emplace members shall not affect the validity of iterators if \((N+n) < z \times B\), where \(N\) is the number of elements in the container prior to the insert operation, \(n\) is the number of elements inserted, \(B\) is the container’s bucket count, and \(z\) is the container’s maximum load factor. ([unord.req], paragraph 15)

rehash(), reserve() | X | Rehashing invalidates iterators, changes ordering between elements, and changes which buckets the elements appear in but does not invalidate pointers or references to elements. ([unord.req], paragraph 9)

std::valarray

resize() | X | Resizing invalidates all pointers and references to elements in the array. ([valarray.members], paragraph 12)

A std::basic_string object is also a container to which this rule applies. For more specific information pertaining to std::basic_string containers, see STR52-CPP. Use valid references, pointers, and iterators to reference elements of a basic_string.

Noncompliant Code Example

In this noncompliant code example, \(\text{pos}\) is invalidated after the first call to \(\text{insert()}\), and subsequent loop iterations have undefined behavior.

```cpp
#include <deque>

void f(const double *items, std::size_t count) {
    std::deque<double> d;
    auto pos = d.begin();
    for (std::size_t i = 0; i < count; ++i, ++pos) {
        d.insert(pos, items[i] + 41.0);
    }
}
```

Compliant Solution (Updated Iterator)

In this compliant solution, \(\text{pos}\) is assigned a valid iterator on each insertion, preventing undefined behavior.

```cpp
#include <deque>

void f(const double *items, std::size_t count) {
    std::deque<double> d;
    auto pos = d.begin();
    for (std::size_t i = 0; i < count; ++i, ++pos) {
        pos = d.insert(pos, items[i] + 41.0);
    }
}
Compliant Solution (Generic Algorithm)

This compliant solution replaces the handwritten loop with the generic standard template library algorithm `std::transform()`. The call to `std::transform()` accepts the range of elements to transform, the location to store the transformed values (which, in this case, is a `std::inserter` object to insert them at the beginning of `d`), and the transformation function to apply (which, in this case, is a simple lambda).

```cpp
#include <algorithm>
#include <deque>
#include <iterator>

void f(const double *items, std::size_t count) {
  std::deque<double> d;
  std::transform(items, items + count, std::inserter(d, d.begin()),
                 [](double d) { return d + 41.0; });
}
```

Risk Assessment

Using invalid references, pointers, or iterators to reference elements of a container results in undefined behavior.

<table>
<thead>
<tr>
<th>Rule</th>
<th>Severity</th>
<th>Likelihood</th>
<th>Remediation Cost</th>
<th>Priority</th>
<th>Level</th>
</tr>
</thead>
<tbody>
<tr>
<td>CTR51-CPP</td>
<td>High</td>
<td>Probable</td>
<td>High</td>
<td>P6</td>
<td>L2</td>
</tr>
</tbody>
</table>

Automated Detection

<table>
<thead>
<tr>
<th>Tool</th>
<th>Version</th>
<th>Checker</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Astree</td>
<td>22.10</td>
<td>overflow_upon_dereference</td>
<td></td>
</tr>
<tr>
<td>CodeSonar</td>
<td>7.2p0</td>
<td>ALLOC.UAF</td>
<td>Use After Free</td>
</tr>
<tr>
<td>Helix QAC</td>
<td>2022.3</td>
<td>C++4746, C++4747, C++4748, C++4749</td>
<td></td>
</tr>
<tr>
<td>Klocwork</td>
<td>2022.3</td>
<td>ITER.CONTAINER.MODIFIED</td>
<td></td>
</tr>
<tr>
<td>Parasoft C/C++test</td>
<td>2022.1</td>
<td>CERT_CPP-CTR51-a</td>
<td>Do not modify container while iterating over it</td>
</tr>
<tr>
<td>Polyspace Bug Finder</td>
<td>R2022b</td>
<td>CERT C++: CTR51-CPP</td>
<td>Checks for use of invalid iterator (rule partially covered).</td>
</tr>
<tr>
<td>PVS-Studio</td>
<td>7.22</td>
<td>V783</td>
<td></td>
</tr>
</tbody>
</table>

Related Vulnerabilities

Search for vulnerabilities resulting from the violation of this rule on the CERT website.

Related Guidelines

SEI CERT C++ Coding Standard

STR52-CPP. Use valid references, pointers, and iterators to reference elements of a basic_string

Bibliography

Subclause 24.2.1, "In General"


[Meyers 2001] Item 43, "Prefer Algorithm Calls to Handwritten Loops"

[Sutter 2004] Item 84, "Prefer Algorithm Calls to Handwritten Loops"