83. Use a checked STL implementation.

Summary

Safety first (see Item 6): Use a checked STL implementation, even if it's only available for one of your compiler platforms, and even if it's only used during pre-release testing.

Discussion

Just like pointer mistakes, iterator mistakes are far too easy to make and will usually silently compile but then crash (at best) or appear to work (at worst). Even though your compiler doesn't catch the mistakes, you don't have to rely on "correction by visual inspection," and shouldn't: Tools exist. Use them.

Some STL mistakes are distressingly common even for experienced programmers:

- *Using an invalidated or uninitialized iterator:* The former in particular is easy to do.
- *Passing an out-of-bounds index:* For example, accessing element 113 of a 100-element container.
- *Using an iterator "range" that isn't really a range:* Passing two iterators where the first doesn't precede the second, or that don't both refer into the same container.
- *Passing an invalid iterator position:* Calling a container member function that takes an iterator position, such as the position passed to **insert**, but passing an iterator that refers into a different container.
- *Using an invalid ordering:* Providing an invalid ordering rule for ordering an associative container or as a comparison criterion with the sorting algorithms. (See [Meyers01] §21 for examples.) Without a checked STL, these would typically manifest at run time as erratic behavior or infinite loops, not as hard errors.

Most checked STL implementations detect these errors automatically, by adding extra debugging and housekeeping information to containers and iterators. For example, an iterator can remember the container it refers into, and a container can remember all outstanding iterators into itself so that it can mark the appropriate iterators as invalid as they become invalidated. Of course, this makes for fatter iterators, containers with extra state, and some extra work every time you modify the container. But it's worth it—at least during testing, and perhaps even during release (remember Item 8; don't disable valuable checks for performance reasons unless and until you know performance is an issue in the affected cases).

Even if you don't ship with checking turned on, and even if you only have a checked STL on one of your target platforms, at minimum ensure that you routinely run your full complement of tests against a version of your application built with a checked STL. You'll be glad you did.

Examples

Example 1: Using an invalid iterator. It's easy to forget when iterators are invalidated and use an invalid iterator (see Item 99). Consider this example adapted from [Meyers01] that inserts elements at the front of a **deque**:

```
deque<double>::iterator current = d.begin();

for( size_t i = 0; i < max; ++i )
  d.insert( current++, data[i] + 41 );          // do you see the bug?
```

Quick: Do you see the bug? You have three seconds.—Ding! If you didn't get it in time, don't worry; it's a subtle and understandable mistake. A checked STL implementation will detect this error for you on the second loop iteration so that you don't need to rely on your unaided visual acuity. (For a fixed version of this code, and superior alternatives to such a naked loop, see Item 84.)

Example 2: Using an iterator range that isn't really a range. An iterator range is a pair of iterators **first** and **last** that refer to the first element and the one-past-the-end-th element of the range, respectively. It is required that **last** be reachable from **first** by repeated increments of **first**. There are two common ways to accidentally try to use an iterator range that isn't actually a range: The first way arises when the two iterators that delimit the range point into the same container, but the first iterator doesn't actually precede the second:

```
for_each( c.end(), c.begin(), Something );      // not always this obvious
```

On each iteration of its internal loop, **for_each** will compare the first iterator with the second for equality, and as long as they are not equal it will continue to increment the first iterator. Of course, no matter how many times you increment the first iterator, it will never equal the second, so the loop is essentially endless. In practice, this will, at best, fall off the end of the container **c** and crash immediately with a memory protection fault. At worst, it will just fall off the end into uncharted memory and possibly read or change values that aren't part of the container. It's not that much different in principle from our infamous and eminently attackable friend the buffer overrun.

The second common case arises when the iterators point into different containers:

```
for_each( c.begin(), d.end(), Something );      // not always this obvious
```

The results are similar. Because checked STL iterators remember the containers that they refer into, they can detect such run-time errors.

References

[Dinkumware-Safe] • [Horstmann95] • [Josuttis99] §5.11.1 • [Metrowerks] • [Meyers01] §21, §50 • [STLport-Debug] • [Stroustrup00] §18.3.1, §19.3.1