Hardening the C++ Standard Template Library

C++ Russia, St. Petersburg

Marshall Clow

November 1, 2019

About me

I have been working on LLVM for eight years, and on libc++ for about six, and I am the "code owner" for libc++.

I am also the chairman of the Library Working Group of the C++ Standards Committee.

Contact info:

1. Email: mclow.lists@gmail.com
2. Slack: marshall
3. IRC: mclow

What is libc++?

Libc++ (https://libcxx.llvm.org) is an implementation of the C++ standard library for LLVM/clang.

It contains (among other things) commonly used facilities like `vector`, `shared_ptr`, `sort` and so on.

Why do we care about hardening the standard library?

Every C++ program depends on a standard library implementation. This means that libc++ is at the "bottom" of their dependency graph.

It is vital that this library be correct and performant.

How do we harden a library?

Some of the techniques we use are:

1. A comprehensive test suite
2. Warning eradication
3. Precondition checking
4. Static analysis
5. Dynamic analysis
6. Fuzzing

Test Suite

Who would think of developing a software package today that does not include a reasonably comprehensive test suite?

libc++ is no exception here. The libc++ test suite can best described as “a good start”.

We also have a build bot that generates coverage metrics from the test suite.

Warning Eradication

Users of libc++ have many requirements; some of them don't care about errors, others compile with `-Werror -Wall -Wextra`. Our goal is to support all of these users.

This is an ongoing job; people are adding warnings to clang all the time.

Precondition checking

Most of the calls in the standard library have preconditions. Some of these preconditions are cheap to check. Others are expensive. Others are not possible to check at all.

libc++ has a macro named `_LIBCPP_ASSERT`, which is defined to expand to nothing. Users can define this macro themselves to catch some precondition violations.

Precondition checking - easy

```
template<class T>
  T& vector<T>::front();
```

*R*equires: `*this` shall not be empty.

```
_LIBCPP_ASSERT(!empty(), "front() called for empty vector");
```

Precondition checking - expensive

```
template<class RanIter, class Compare>
  void sort(RanIter first, RanIter last, Compare comp);
```

*R*equires: comp shall induce a strict weak ordering on the values.

Precondition checking - not possible

```
template<class InIter>
  typename iterator_traits<InIter>::difference_type
    distance(InIter first, InIter last);
```

*R*equires: `last` shall be reachable from `first`.

Static analysis

We don't do near as much static analysis as I would like.

The current static analysis tools that we've tried are very "noisy"; reporting many of problems, most of which are false positives.

Dynamic analysis

libc++ was an early user of the sanitizers. See
http://blog.llvm.org/2013/03/testing-libc-with-address-sanitizer.html and
http://blog.llvm.org/2013/04/testing-libc-with-fsanitizeundefined.html

The have caught many bugs that would have plagued users in the field.

libc++ also has ASAN integration for `std::vector`, so ASAN can tell if you read or write off the end of the vector (even if its not off the end of the memory block that vector manages)

Now we have bots that run the tests under ASAN/UBSAN all the time.

Fuzzing

libc++ is an excellent target for fuzzing; it has many (independent) entry points. I did some fuzzing experiments, and found a few bugs in libc++.

Now we use OSS-Fuzz https://github.com/google/oss-fuzz for this.

Fuzzing example (4)

```
int test_partition(const uint8_t *data, size_t size)
{
    vector<uint8_t> working(data, data + size);
    auto iter = partition(working.begin(), working.end(),
                          is_even<uint8_t>());

    if (!all_of (working.begin(), iter, is_even<uint8_t>()))
        return 1;
    if (!none_of(iter,   working.end(), is_even<uint8_t>()))
        return 2;
    if (!is_permutation(data, data + size, working.cbegin()))
        return 99;
    return 0;
}
```

Questions?

Thank you