# Development strategies: You've written a library - now what?

Marshall Clow

C++ Alliance

April 10, 2019

# About me

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

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

I work for the C++Alliance, a US-based non-profit organization.

Contact info:

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

# Documentation

Does your library have any documentation? Even just a README?

In many cases, the documentation is how you make a first impression. Good docs can draw people in.

Many people evaluate a library by looking at the documentation.

# What kinds of documentation should you have?

1. Overview

# What kinds of documentation should you have?

1. Overview
2. Getting Started

# What kinds of documentation should you have?

1. Overview
2. Getting Started
3. Examples

# What kinds of documentation should you have?

1. Overview
2. Getting Started
3. Examples
4. Tutorials

# What kinds of documentation should you have?

1. Overview
2. Getting Started
3. Examples
4. Tutorials
5. Reference

# Tests

Disclaimer: I **really** like tests.

A good set of tests can help you over and over again:

1. Make sure that you've fixed a bug
2. Catch regressions
3. Pinpoint problems with someone's installation or configuration.
4. Porting to a new platform

# A good test suite is:

1. Easy to run
2. Always green
3. Comprehensive
4. Run often - preferably on every change.
5. Easy to automate

# Example: Boost

The boost documentation explains how to run the boost tests on your machine.

To run a library's regression tests, run Boost's `b2` utility
from the `<boost-root>/libs/<library>/test` directory.

To run every library's regression tests, run `b2` from the `<boost-root>/status` directory.

Source: https://www.boost.org/development/running_regression_tests.html

# Tools

There are an amazing number of tools available to developers today:

1. Compilers

# Tools

There are an amazing number of tools available to developers today:

1. Compilers
2. Static Analyzers

# Tools

There are an amazing number of tools available to developers today:

1. Compilers
2. Static Analyzers
3. Dynamic Analyzers

# Tools

There are an amazing number of tools available to developers today:

1. Compilers
2. Static Analyzers
3. Dynamic Analyzers
4. Code Coverage Tools

# Tools

There are an amazing number of tools available to developers today:

1. Compilers
2. Static Analyzers
3. Dynamic Analyzers
4. Code Coverage Tools
5. Fuzzers

# Compiler warnings

Different compilers will warn on different things.
Sometimes they point out problems in your code:

```
int val = <some expression>;
if (val < INT_MIN) { ... }
```

or

```
if (a = b) { ... }
```

but sometimes the compiler is just wrong:

```
auto foo = std::make_unique<unsigned char>(0);
```

# Static Analyzers

The "static" part of static analysis means that it does not happen while your program is running. It happens inside of a separate program, which builds a model of your code and then analyses that mode.

Every compiler has a (simple) static analyzer inside of it; that's how they generate the "tautological comparison" warnings.

There are several commercial static analysis tools. Coverity, Fortify and KlocWork seem to be the most popular.

LLVM has a static analysis framework, and a tool (`clang-tidy`) that uses it.

# Static analysis example from clang-tidy

```
void foo(int a, double b);
...
foo(1.0, 3);
```

This is perfectly legal code.

# Dynamic Analyzers

Dynamic Analyzers perform their work while your program is running.

1. Assertions

# Dynamic Analyzers

Dynamic Analyzers perform their work while your program is running.

1. Assertions
2. "Debug Mode"

# Dynamic Analyzers

Dynamic Analyzers perform their work while your program is running.

1. Assertions
2. "Debug Mode"
3. Sanitizers

# Fuzzers

Fuzzers create random-looking inputs to your program, and then see if your program misbehaves and/or crashes.

1. libFuzzer from clang
2. American Fuzzy Lop
3. OSS-Fuzz aka "Fuzzing as a service"

# Dealing with Users

# Dealing with users

What do users do?

1. Ask Questions / make Comments
2. File bug reports
3. Make feature Requests
4. Offer contributions
5. Port to new systems

# Thank them!

If someone has gone to the trouble to learn your library enough to send you a comment, or file a bug report, or make a contribution, you should be grateful.

1. Questions can hopefully be answered with a link to your documentation.
2. Bugs should be fixed (duh!) or explained why it's not a bug.
3. Contributions can be evaluated and discussed with the submitter.

# Bug Example: libc++

Bug report: `multimap<T>::clear()` missing an exception specifier.
It should be marked `noexcept`, but it is not.

What did I do?

1. Check: Do we have a test for this?
2. Write a test
3. Watch it fail
4. Add `noexcept`
5. Run the tests again
6. Watch them pass
7. Check in the new test and the fix

# Design Example: Howard Hinnant's Calendar library

The library has types named `day`, `month`, `weekday`, `year`, which are thin wrappers over a numeric value, with a bit of logic.
Originally, these were not default constructible; since what is the value of a default-constructed day?

But people wanted to do this:

```
day d;
somestream >> d;
```

and

```
std::vector<weekday> v(12); // 12 weekdays
```

and then read them in from a file.

# Listen to your users!

They will use your library in ways that you did not anticipate.

# Do you have explicit releases?

Do you expect your users to "live at head", or will they use a particular version of your library?

People who are using your library for important things want to change their infrastructure at times of their choosing.

You should consider having milestones, which are intended to be stable for a while.

# What goes into a release?

1. An announcement
2. Release notes
3. A method for obtaining the release

# Release Notes

What kinds of things should go into release notes?

1. New features since the last release.
2. Problems fixed since the last release.
3. Changes that affect users.
4. News about the project - maybe future plans.

# Managing Change

As your user base grows *(as you gain understanding of the problem domain)*, you'll see places where your library can be improved; either on your own own or via suggestions from other people.

Making these kinds of changes is a tricky process, because it involves change in both the library code and in the code of the people who use it.

# Managing Change (2)

It is important to consider both the costs and the benefits of a change.

The benefits are clear to you; they're right there in the code.
The costs are less clear, because much of them are not borne by you.
Your users are the ones that will have to change.

# Managing Change (3)

How can you mitigate the costs of breaking changes?

1. Make changes only when the benefit is compelling

# Managing Change (3)

How can you mitigate the costs of breaking changes?

1. Make changes only when the benefit is compelling
2. Documentation

# Managing Change (3)

How can you mitigate the costs of breaking changes?

1. Make changes only when the benefit is compelling
2. Documentation
3. Provide both old and new interfaces for a period of time

# Managing Change (3)

How can you mitigate the costs of breaking changes?

1. Make changes only when the benefit is compelling
2. Documentation
3. Provide both old and new interfaces for a period of time
4. Example conversions

# Managing Change (3)

How can you mitigate the costs of breaking changes?

1. Make changes only when the benefit is compelling
2. Documentation
3. Provide both old and new interfaces for a period of time
4. Example conversions
5. Automated code conversion tool

# Conclusions

1. Good documentation can help attract new users

# Conclusions

1. Good documentation can help attract new users
2. Tests can help keep your code quality high

# Conclusions

1. Good documentation can help attract new users
2. Tests can help keep your code quality high
3. There are a lot of tools out help you improve your code

# Conclusions

1. Good documentation can help attract new users
2. Tests can help keep your code quality high
3. There are a lot of tools out help you improve your code
4. Listen to your users - treat them kindly

# Conclusions

1. Good documentation can help attract new users
2. Tests can help keep your code quality high
3. There are a lot of tools out help you improve your code
4. Listen to your users - treat them kindly
5. Take field experience into account

# Conclusions

1. Good documentation can help attract new users
2. Tests can help keep your code quality high
3. There are a lot of tools out help you improve your code
4. Listen to your users - treat them kindly
5. Take field experience into account
6. Tell them what you've done.

# Conclusions

1. Good documentation can help attract new users
2. Tests can help keep your code quality high
3. There are a lot of tools out help you improve your code
4. Listen to your users - treat them kindly
5. Take field experience into account
6. Tell them what you've done.
7. Think about how changes will affect your users

Thank you

# Links

1. C++Alliance: https://www.cppalliance.org
2. Sanitizers in clang: https://clang.llvm.org/docs/UsersManual.html#controlling-code-generation
3. Fuzzers in clang: https://llvm.org/docs/LibFuzzer.html
4. clang-tidy: http://clang.llvm.org/extra/clang-tidy/
5. Kostya's Fuzzing talk: https://www.youtube.com/watch?v=k-Cv8Q3zWNQ
6. OSS-Fuzz: https://github.com/google/oss-fuzz
7. Blog post about changing API: https://cplusplusmusings.wordpress.com/2016/02/01/sometimes-you-get-things-wrong/