Switching absl::optional to std::optional

David Benjamin
Last updated: Sep 25, 2023
https://crbug.com/1373619
https://crrev.com/c/4469528 (though see discussion of binary size below)

Although we've long moved to C++17, Chromium still reconfigures `absl::optional` to use a polyfill over a `std::optional` typedef. This document discusses switching to the typedef, as the first step to migrating to `std::optional`.

The main decision point here is a binary size vs debuggability tradeoff. **Do we want better debuggability and worse binary size (+19K) or worse debuggability and better binary size (-94K)?** More on this below in the OPEN QUESTION section.

Motivations

`#include <optional>` is a bit shorter than `#include "third_party/abseil-cpp/absl/types/optional.h"`.

External C++ libraries may not depend on Abseil and thus use `std::optional`, such as the certificate verifier getting extracted to BoringSSL. Even Google projects that do depend on Abseil will use `std::optional` as the rest of Google C++ stopped needing the polyfill after C++17. If there is some way `std::optional` is worse for us (e.g. bad binary size or memory usage), we'll need to understand and resolve it anyway.

Likewise, as `std::optional` appears in API boundaries for those libraries, we will need to utter `std::optional`. Our PRESUBMIT warnings will then get in the way and need exceptions. `std::optional` is less common at API boundaries than `std::string_view`, but it will still be a source of development friction.

QUICHE is currently blocked on aligning with Google C++ style because of the Chromium preference, so this will also help them align things.

Finally, Abseil will eventually require C++17. As the Abseil types exist to be polyfills over the STL ones, they may drop their polyfill at that point. Switching to the typedef avoids putting us in a bind if that happens.

Safety checks

We once preferred Abseil types over libc++ because Abseil had hardening checks. libc++ has since caught up and we enable libc++'s "safe mode". There is no security difference between the two types:

The only hardening checks to add to `std::optional` / `absl::optional` are whether `operator->` and `operator*` check that the value exists. (`optional::value` is already required to throw `bad_optional_access`.) Both libc++ and Abseil do this:
https://source.chromium.org/chromium/chromium/src/+/main:third_party/libc++/src/include/optional;l=998-1050;drc=491ba7b7c3c12d39022bc1fb30f85545eafdb3b4
https://source.chromium.org/chromium/chromium/src/+/main:third_party/abseil-cpp/absl/types/optional.h;l=412-444;drc=5dd7dd46d5ae09b28f5553ca24221eea0072b66f

We also are not giving up significant possibilities of future improvements or safety checks. `absl::optional` is maintained as a polyfill for `std::optional`, so the Abseil team won't diverge from `std::optional` anyway, for better or worse. Abseil or libc++, if we want to improve on our optional type, we need to stay within the STL's API contract (which we can upstream to libc++), or fork the type.

Windows memory usage

libc++'s `std::optional` and `std::variant` were originally larger on Windows. I fixed that in libc++ upstream, so this is no longer a concern.

Binary size

Trying to switch to `std::optional` initially reported significant size regressions. I believe the causes of these are now fixed or understood:

Stack canaries

The majority of the size increase (90K!) was triggered by a change in the field order. Abseil uses `bool; T` while libc++ uses `T; bool`. These two orders have the same size. Some offsets change, but this is bizarre. It turned out this was a Clang bug.

Chromium built with `-fstack-protector`, which adds a stack canary to functions that allocate `char` buffers on the stack above some size. By default that threshold is 8, but Chromium previously set it to 4.

4 makes us more likely to trip a Clang bug: in some cases (I think it's when LLVM's padding and Clang's padding differ?), Clang lowers a C type into an LLVM IR type with explicit padding. `struct { bool; uint64_t; }` becomes `type { i8, i64 }`, while `struct { uint64_t; bool; }` becomes `type <{ i64, i8, [7 x i8] }>`. This explicit padding is then misinterpreted by `-fstack-protector` as a C character array! Whenever there is an `optional<T>` on the stack such that `T` has alignment >= 8, we would add a stack protector with `std::optional` where there may not have been one with `absl::optional`. This issue is compounded by Chrome still shipping 32-bit Arm, and 32-bit Arm is particularly bad at stack canaries. (Very small offset space for PC-relative loads forces an extra indirection to load `__stack_chk_guard`.)

I filed an LLVM bug at https://github.com/llvm/llvm-project/issues/66709. In the meantime, this issue was worked around by using the default 8-byte threshold, saving 278,544 bytes. With that fixed, the Abseil and libc++ field orders are comparable. They do change offsets, but the difference attributable to the ordering is overall in libc++'s favor by 9,349 bytes.

OPEN QUESTION: How to crash

There is still a size increase by 19,432 bytes. This seems to come from crash sequences.

Abseil uses `__builtin_trap` to abort. libc++ calls `_LIBCPP_VERBOSE_ABORT` with a format string (more on this below), which we define to `__libcpp_verbose_abort` and then implement with `base::ImmediateCrash`. That, in turn, is some inline asm.

The libc++ and //base abort inhibits compiler optimization in a way the Abseil one does not. Between the format string parameters and the inline assembly (depending on whether LTO inlines it), different aborts in the same function cannot be coalesced together. This means a function that includes many safety checks needs to emit lots of abort spots. For //base, this was intentional to improve debuggability, because it means we know which safety check failed in a crash dump. It is unclear whether this was intentional for upstream libc++. (Though we can make it intentional by defining `_LIBCPP_VERBOSE_ABORT` to what we want.)

In contrast, the Abseil version is a compiler intrinsic, so the compiler can freely merge them.

Switching libc++'s `_LIBCPP_VERBOSE_ABORT` macro to `__builtin_trap` reduces binary size by 85,392 bytes. That is, independent of `std::optional`, we're paying 85K for this debuggability already. Switching `absl::optional` to `std::optional` saves an additional 9,140 bytes. That aligns with the size savings from switching just the field order, above, which implies this is the rest of the difference.

This isn't a good reason to stick with `absl::optional`. We should evaluate this tradeoff consistently. That is, we should **pick one of these two:**

1. Either we believe the debuggability is worth the size hit. We should then switch `absl::optional` to `std::optional` to improve debuggability and pay the 19K for our belief.
2. Or we believe the debuggability is not worth the size hit, at least for STL-like libraries with lots of inlined safety checks. We should then switch `_LIBCPP_VERBOSE_ABORT` to `__builtin_trap`, save 85K, then `absl::optional` to `std::optional` for an additional 9K savings from the better field order. For those savings, we'll pay some debuggability on STL safety checks.

Which is our preference?

IWYU

Removing Abseil's polyfill means `third_party/abseil-cpp/absl/types/optional.h` pulls in fewer transitive Abseil headers. Most of the CLs on https://crbug.com/1373619 are to tidy that up.

Future work and related issues

LSC

After this change lands, we should do an LSC to replace references to `absl::optional` to `std::optional`, but this can be a later project. In the meantime, the two types will interoperate cleanly as they'll be aliases for each other. This document does not propose a particular way to do this, but tooling for it would sure be nice.

std::variant

Early attempts at `std::variant` showed a similar binary size cost, but they were likely all analogous to the `std::optional` causes. There's also an outstanding bug that we'll need fixed. After that, and after we've decided how to handle `std::optional`, we can run a couple experiments to confirm `std::variant` adds no new size issues. If not, we can apply the same decision to `std::variant`.

Verbose aborts

Keen eyes may have noticed above that `_LIBCPP_VERBOSE_ABORT` seems to currently be worse than `base::ImmediateCrash`. It passes in a format string, `__FILE__`, and `__LINE__`, and then ignores it. When the function isn't inlined, those end up in the binary!

This is https://crbug.com/1411831. Unfortunately, fixing it either with an inline `base::ImmediateCrash` or an out-of-line version still increased binary size altogether. The two I tried were:

- https://crrev.com/c/4874969 - 33,811 byte size increase
- https://crrev.com/c/4873454 - 48,165 byte size increase

I'm not sure exactly what's going on here. Looking at the breakdown, it seems some functions got smaller, but others got significantly larger, particularly one function that creates many `std::string`s (more below). I didn't dig very carefully, but it seems making aborts cheaper caused Clang to be more willing to inline things, but then that caused it to generate larger code. I suspect the inline asm blocks also throw Clang off. (Even out-of-line, I still saw inlining, possibly from LTO.)

That said, some of the size increase might also be from increased debuggability. I think the new way to compile the function above actually fixed a "problem" where the distinct `std::string` constructor calls got deduped.

This needs more exploration, but I think it should be treated as orthogonal.

Unoptimized non-overlapping checks

That constructing `std::string`s requires a safety check is surprising. The function in question was https://source.chromium.org/chromium/chromium/src/+/main:v8/src/asmjs/asm-types.cc;drc=32e6b48fca669a4a5dce7844396dbc9040649bc6;l=28. Underneath all the macros, it's just doing:

```
None
std::string AsmType::Name() {
  ...
  switch (some_integer) {
    case foo:
       return "foo";
    case bar:
       return "bar";
    ...
  }
  ...
}
```

There's no reason for constructing a `std::string` out of a string literal to require a safety check. This comes from the constructor calling `std::char_traits<char>::copy` under the hood, which has a non-overlapping requirement. Clang seems unable to optimize it out. Granted, aliasing analysis is hard, but this should be an easy case. I filed https://github.com/llvm/llvm-project/issues/66953 for this.

Another fix could be to disable some of the checks. libc++ is moving to having a separate "hardened" mode from "safe" mode that disables some checks. We currently use "safe" mode. However, there are still many checks that are uncategorized (which we'd lose) and it is unclear whether we want to lose all the checks that "safe" mode loses.

TODO(davidben): There's the bug above about LLVM not optimizing something, but file a more general bug about specifically `std::string` being full of these checks. That could be resolved by making LLVM smarter, or we could change libc++ to not do these checks in some (which?) cases.