What is an ABI and Why is Breaking it a Problem?

Marshall Clow

C++ now

2021
MAY 2-7
Aspen, Colorado, USA

A bit of WG21 history

In the past, several proposed improvements to the standard library have been rejected by the committee using the shorthand phrase, "that's an ABI break". Today, I want to talk about what that actually means.

At the C++ committee meeting in Prague (2020), Titus Winters presented a paper (P1863) arguing that it was time to make changes to some of the classes in the standard library in a non-compatible way. The title of the paper was "ABI - Now or Never".

At that meeting, Titus asked the committee to commit to "breaking the standard library ABI" in some future release of the standard. The committee declined to make such a commitment.

What is an ABI?

ABI stands for "Application Binary Interface", and it deals with the interfacing of C++ code to a particular platform. It includes things like:

1. Structure layout
2. vtable layout
3. Parameter passing conventions
4. Name Mangling
5. Exception Handling methods

ABI and Library changes

In this talk, I'm going to be discussing changes to library code, and how they affect ABI. Not compiler changes.

Strictly speaking, this is not really 'ABI', but that's the term that people use, so I'm going to continue using it.

When I talk about the ABI of piece of code, I mean things like structure layout, parameter passing, etc.

But first, we need to talk about the "One Definition Rule"

C++ has this principle called the "One Definition Rule", which (broadly) says that if there is more than one (non-identical) definition of an entity visible in a program, then the behavior of the program is undefined.

"One Definition Rule" (2)

The actual term that the standard uses is "ill-formed, no diagnostic required."

That means that the toolchain is allowed to produce a program that can do anything, and not tell you that it has done so.

Examples of ODR violations

1. Two different definitions
2. Change the layout of a struct
3. Add a virtual method
4. More subtle things

Two definitions (1a)

```
// "header1.h"
struct Foo {
    int32_t a;
    int32_t b;
};
```

Two definitions (2a)

```
// "header2.h"
struct Foo {
    int32_t a;
    int32_t added;
    int32_t b;
};
```

What happens here?

Code that includes `file1.h` believes that `sizeof(Foo) == 8`, while code that includes `file2.h` believes that `sizeof(Foo) == 12`
Code that includes `file1.h` believes that `b` is at offset 4 in the struct, while code that includes `file2.h` believes that `b` is at offset 8.

If these structures get passed between code that has different expectations, bad things could happen.

This also affects any classes that inherit from `Foo`, or have member variables of type `Foo`. You may have seen something like this as the "fragile base class" problem.

Variations on a theme

1. Removing a member
2. Reordering members
3. `#pragma pack`

Two definitions (1b)

```
// "header1.h"
struct Foo {
    virtual int One (int bar);
    virtual ~Foo ();
};
```

Two definitions (2b)

```
// "header2.h"
struct Foo {
    virtual int One (int bar);
    virtual double Two (std::string s);
    virtual ~Foo ();
};
```

What happens here?

Every variable of type `Foo` contains a pointer to a 'vtable', an array of function pointers (one per virtual function). One set of code believes that the table contains two entries, the other three.

Two definitions (1c)

```
// "header1.h"
template <typename T1, typename T2>
struct pair {
    T1 first;
    T2 second;
    ...
    pair(const pair &p)
    : first(p.first), second(p.second) {}
};
```

Two definitions (2c)

```
// "header2.h"
template <typename T1, typename T2>
struct pair {
    T1 first;
    T2 second;
    ...
    pair(const pair &p) = default;
};
```

What happens here?

We've let the compiler generate the copy constructor for `pair`. Since `pair` is a struct, the defaulted copy constructor does member-wise construction - exactly what we were doing before. It should even generate the same code!

However, with this change, specializations of `pair` can be "trivially copyable", where before they could never be.

On some platforms, parameters of trivially-copyable types which can fit into a register are passed in a register instead of on the stack.

Why can't the compiler diagnose this?

That 'IFNDR' result is scary!

Consider three cases:

1. Two different definitions in the same translation unit
2. Two different definitions in the different translation units, statically linked
3. Two different definitions in the different translation units, dynamically linked

Ok, but how do I get from an ODR violation to an ABI break?

Instead of two files, think about two different versions of the same file.

You install a new version of a shared library that you use, and it comes with an updated header file, with a different declaration for a struct/class that you use in your code, and that code does not get rebuilt to incorporate that change.

Alternately, you install a system update, and it comes with an updated shared library that you use.

An ABI Break is just an ODR violation in time.

How can I avoid this?

1. Don't change things that affect ABI
2. Don't have "stale binaries"
3. Have only one definition for everything

Do we have examples of ABI breaks "in the wild"? (1)

For C++17, the committee changed `emplace_back` to return a reference to the newly emplaced object, instead of `void`.

This did not cause any problems that I am aware of.

Do we have examples of ABI breaks "in the wild"? (2)

libstdc++ changed the layout and behavior (and hence, the ABI) of `std::basic_string` for C++11, due to changes in the standard. Their old string class implemented "copy-on-write" semantics, while the new one did not. They provided a "back door", (`_GLIBCXX_USE_CXX11_ABI`) so that people could keep the old layout and behavior, for compatibility with old software.

This caused much anguish and confusion, which persists to this day (a decade later) in spite of heroic efforts by the libstdc++ people to minimize the pain.

Jonathan Wakely wrote (when asked about how libstdc++ handled the change):
By inventing new compiler magic and spending a lot of time and energy to ensure it could be a *transition* (on the users' schedules) not a hard break. The old and new ABIs can co-exist in a single process.

An example of a change rejected because of an "ABI break"?

For C++20, there was a proposal (P0192) to add a "`half float`" type, a smaller (16 bit) floating-point type that would play well with GPUs.

This was not adopted for C++20, and one reason was that adding iostreams support would involve adding virtual functions to `num_put` and `num_get`. There were other reasons that it was not adopted as well.

Why is WG21 talking about this? (1)

Theoretically, WG21 has no reason to poke into this. At any given time, there is only one C++ standard. The only mention of older standards is in Annex C, and that's only "informative".

Traditionally, it has been implementers who have prevented ABI breaks - mostly speaking up in committee when a change could cause an ABI break, or reporting defects for ABI breaks that are discovered.

Why is WG21 talking about this? (2)

However, some members of WG21 wish to make changes that involve ABI changes, and so have started a meta-discussion about when ABI breaks are allowed and/or desirable.

See P2028: What is ABI, and What Should WG21 Do About It?

How could this be detectible?

One of the suggestions in P2028 is that if WG21 were to sanction an ABI break (for, say C++26) is to request implementers change the name mangling scheme starting in that release.

This would mean that object files built with `-std=c++26` would not link with object files built with previous standards, and executables that used shared libraries would fail to load.

How could this be detectible? (2)

The real problem is that the "system" is never put together until you actually launch the program. That's when all the binaries (executables, shared libraries, plugins, etc) are all together.

What would this mean to developers?

Assuming some kind of "fat binary" packaging scheme, developers that ship binaries would have to choose between supporting "old standards", "new standards" or both.

What would this mean to users?

1. If you have source to every bit of software that you use, and are willing to rebuild it, then this is not a problem for you.
2. If you never use any third-party software, then this is not a problem for you - your OS vendor will resolve any issues.
3. Otherwise, if you have binaries that use C++ internally, then this would affect you.

User Story - A PhotoShop User

Imagine that you are a graphic artist, and a heavy PhotoShop user. You upgrade your system to a new OS version, and that comes with a new standard library dylib, which has a different ABI. Fortunately, Adobe is on top of this, and they have a new version of PhotoShop ready to go, which uses that new ABI.

You launch the new version of PhotoShop, and none of your third party plugins load. (Maybe they crash, but not-loading is the best possible outcome)

Now you have to disable all those plugins, and contact the 15 different developers that make the 40+ plugins that you use on a regular basis. Some will have free updates ready to go. Some will want $$. Some will say "I'll get right on that". Some will say "Yeah, one of these days". Some will not answer your phone/emails.

Summing up

1. There's a real problem here
2. Historically, the committee has prized stability
3. "We" would like a solution that will allow us to make changes
4. We do not have such a solution today

If we were to change the ABI of the standard library, would this be a one-time thing, or would we want to do it more than once?

Questions?

Thank you