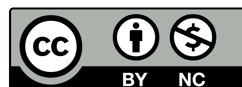
ABI

Or: Am Barely Interested

Core C++ Meetup, Feb. 2020

Yehezkel Bernat

YehezkelShB@gmail.com, @YehezkelShB



Intro

- You may have heard discussions about ABI, proposals, conference talks, blog posts
- You may have heard that “this is why we can’t have nice things”
- We’ll see:
  - What is ABI
  - Where it affects us
  - Why this is a real problem in the real world

ABI – What

API

- API – Application Programming Interface
- E.g. changing function name in a library breaks library’s API compatibility
- API is about *source compatibility*
- “API-compatible change” means that no source code change is required
- E.g. when changing the implementation of any entity (function or type)

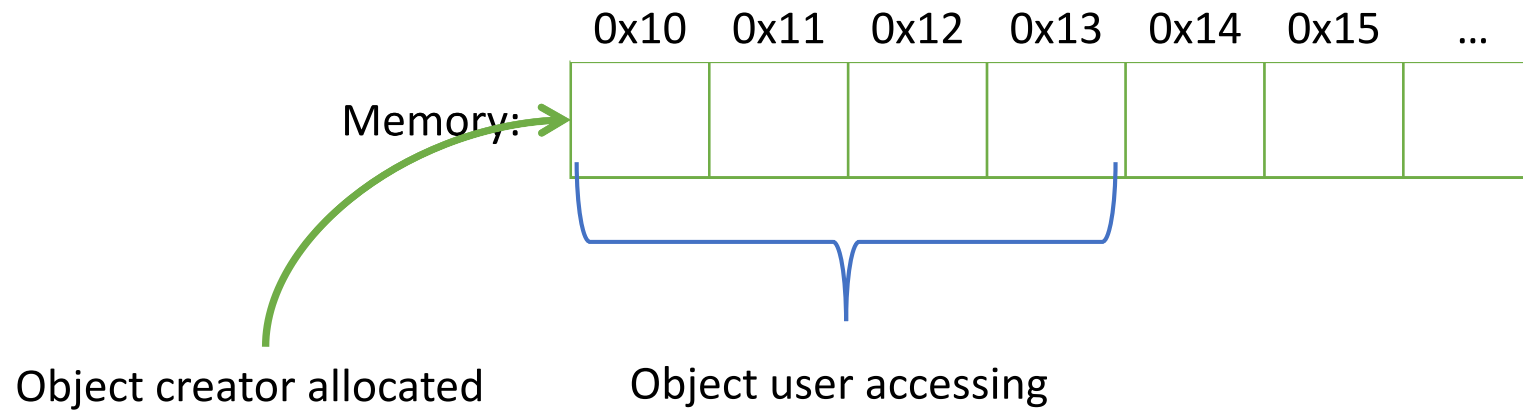
ABI

- ABI – Application **Binary** Interface
- ABI compatibility is about keeping things compatible *without recompiling*
- Breaking API causes ABI breakage too, that's trivial
  - Except for very limited use-cases, maybe
- But there are many changes that break ABI even while keeping API compatibility

Object size changes

- Changing a **private** field in a type
- No API break, it’s still source compatible, the public interface hasn’t changed
- But it’s an ABI break, it isn’t binary compatible
- (Code examples)

Object size changes – example



Object fields change

- Similarly, changing the order or the meaning of the fields, breaks ABI

v1

```
struct Student {
    std::string m_id;
    std::string m_name;
};
```

v2

```
struct Student {
    std::string m_name;
    std::string m_id;
};
```

- This is source compatible change (API compatible) but ABI is broken
- Information will be swapped
- (Changing order of base classes is also considered as data order change)

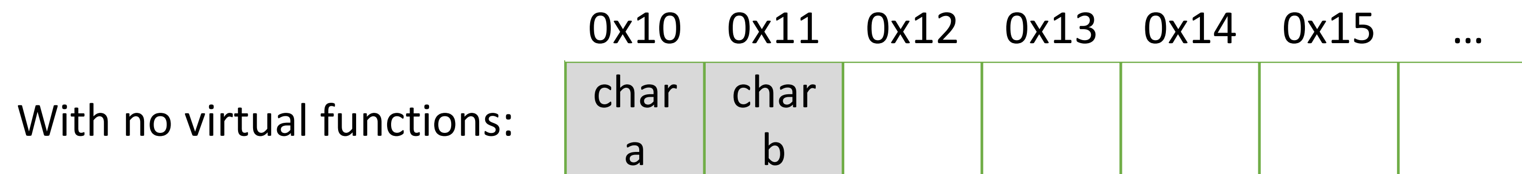
Platform differences

- Differences in object layout can come from various environment settings
- 32/64 bit environments:
  - Type sizes may differ
  - Alignment requirements may differ, and thus the layout and padding
- Endianness
  - Order of bytes inside the type

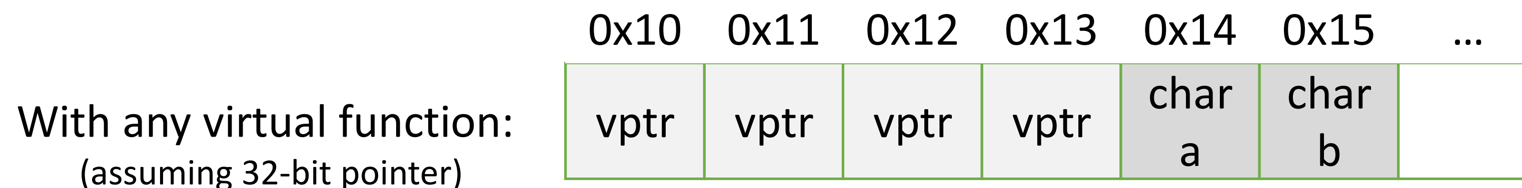
No-so-obvious way to change object size

- Adding the first `virtual` function to a type
- It adds `vptr` to the object
  - In the common implementation
- Moving all the other fields further

With no virtual functions:



With any virtual function:
(assuming 32-bit pointer)



Speaking of virtual functions…

- Adding a new member function anywhere or reordering member functions doesn’t affect ABI
  - Normal functions aren’t part of the object layout
- Adding a new **`virtual`** function, breaks ABI if it wasn’t added as the last one in the type
  - (Even if it’s never called!)
- Because it pushes all the rest of the pointers down in the `vtbl`

Adding a virtual function at the end is safe!

So adding a virtual function at the end is safe

- If the class is used as a base class, now it pushes further the virtual pointer *of the derived class*
- Well, at least if the type is declared `final` we can add virtual functions at the end
  - But what's the point in a `virtual` function if no one can override it?

ABI – What

Things that aren’t object size or meaning

Platform ABI considerations – Name mangling

- In C, the symbol name of a function in the compiled binary is simply its name
- C++ compilers use name mangling to encode the function parameters (and more) into the function name
  - To support function overloading without complicating the linkers very much
- Change to function parameters or return type results in different function name and ABI break
  - Class name (for member functions) and namespace name are also embedded there
- Similarly, type names are built to include the namespace (and then used in turn to build the function name mangling)

Platform ABI considerations – Name mangling

Code	MSVC	gcc/Clang (Itanium ABI)
void f(char)	?f@@YAXD@Z	_Z1fc
void f(int)	?f@@YAXH@Z	_Z1fi
void g(const std::lock_guard<std::mutex>&)	?g@@YAXAEBV?$lock_guard@Vmutex@std@@@std@@@Z	_Z1gRKSt10lock_guardISt5mutexE
void g(const std::scoped_lock<std::mutex>&)	?g@@YAXAEBV?$scoped_lock@Vmutex@std@@@std@@@Z	_Z1gRKSt11scoped_lockIJSt5mutexEE
std::lock_guard<std::mutex>::lock_guard(std::mutex&)	??0?$lock_guard@Vmutex@std@@@std@@QEAA@AEAVmutex@1@@Z	_ZNSt10lock_guardISt5mutexEC2ERS0_
std::scoped_lock<std::mutex>::scoped_lock(std::mutex&)	??0?$scoped_lock@Vmutex@std@@@std@@QEAA@AEAVmutex@1@@Z	_ZNSt11scoped_lockIJSt5mutexEEC2ERS0_

https://gcc.godbolt.org/z/Vf-jd2

Name mangling – good news and bad news

- The bad news are that each change to function signature or to type definition breaks ABI
- The good news are that such a change is caught on link time
  - Unlike C
  - But more on this later

And more and more…

- Calling conventions like:
- In what order argument are passed
  - right-to-left or left-to-right
- Where they are passed
  - Registers? Which ones? Stack? Depends on argument type?
- Who cleans the stack frame
  - The caller or the callee?
  - (see MSVC stdcall vs. cdecl vs. fastcall)

What Does The Standard Say?

https://me.me/embed/i/cec6fb33d9f643519a16a0ea4948b200

The standard doesn’t mention ABI!

- This is the common reaction

ODR – One Definition Rule

- Most (if not all) of the examples fall under ODR
- Here are relevant quotes from [basic.def.odr] (emphases mine)
  - https://eel.is/c++draft/basic.def.odr#10

Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program outside of a discarded statement; no diagnostic required.

The definition can appear explicitly in the program, it can be found in the standard or a user-defined library, or (when appropriate) it is implicitly defined

IF-NDR

- Ill Formed; No Diagnostic Required
- Some ODR violations are hard or impossible to diagnose
- The standard accepts this by making ODR violation IF-NDR

Rejecting changes to the standard

- The standard still could decide on changes that would require ABI break or make implementors life much harder when trying to keep ABI stable
- Not doing such changes is also a way to say something about ABI
  - (See Titus Winters’ paper linked in the references)

The standard doesn’t mention ABI! - Fixed

- To correct the common reaction
- The standard doesn’t *demand* any *specific* ABI
  - Keeping platform and implementer freedom
- But it’s well aware of ABI

Nice Things We Can’t Have ☹

std::scoped_lock

- C++17 draft planned to change std::lock_guard to a variadic template
- NB comments mentioned this is an ABI break
- The change has been reverted, and a new type was introduced, std::scoped_lock
- (See references at the end)

std::default_order

- Replacing std::less as the default for ordered associative containers
- Removed from C++17 draft due to NB comment that it’s an ABI break
- The original paper discussed ABI effects!
  - But overlooked the one mentioned in the NB comment
- (See references at the end)

Zero-cost std::unique_ptr

- std::unique_ptr type was designed carefully to be a zero-cost replacement to raw pointers…
- … for the object size
- ABIs still mandate passing object-with-non-trivial-d-tor on the stack
- While raw pointer is passed in a register
- Can’t be changed due to ABI break
  - (there are also issues with destruction order, but let’s don’t get into it)
- (See Chandler Carruth’s CppCon talk; link in the references)

ABI – Where?

Where ABI break affects us and how?

- When name mangling changed:
- Linking objects or static libraries fails
- Loading dynamic libraries (DLL/so/dylib) – it's a linkage failure that happens in runtime (usually load-time)
- When name mangling isn't changed (e.g. type content changed):
- Undefined Behavior™ in its worst!
- IF-NDR

ABI – How To Fix It

Inline namespaces!

- C++11 introduced inline namespaces
- Lib code: namespace MyLib { inline namespace v1 { class MyType; }}
- User code: MyLib::MyType;
- The compiler (and linker) sees it as MyLib::v1::MyType;
- When lib changed, it removes "inline" from v1 and adds:
- namespace MyLib { inline namespace v2 {
  class MyType; /* shiny new one! */ }}
- Old user binaries keep loading v1 type from the DLL (the mangled name hasn't changed)

Inline namespaces ☹

- It does help the user code <-> type provider
- It doesn't help when the type is used in ABI boundaries (passed between two different binaries)
- Either it affects the user code name mangling…
  - And then user code doesn't link to 3rd party lib anymore
- … or it doesn't affect it (type is used a member of user-defined type)
  - And then, guess what?
  - IF-NDR

Just recompile everything from source!

- (Probably part of the reasoning behind Titus Winters' paper)
- If you can't recompile everything you use, you are doomed anyway!
  - (-random r/cpp rant)
- Is it?

Just recompile everything from source – RLY?

mpyne 5 points · 10 days ago

> For me it is not a valid argument. If someone is not able to recompile whole code, he is s****'ed no matter the ABI stability.

Have none of you ever used Linux distributions where you *actually* have to recompile code?

I use Gentoo, and have since 2006, and ABI breakages, while theoretically resolvable by "just compiling affected software", are often nightmarish affairs.

Binary distributions

- Distributing a binary lib becomes impossible
  - We had enough with MSVC v10, v11, etc. * debug/release * static/dynamic
- How many variations of the same lib you are going to provide?
- How many variations of the same lib the Linux distro is going to provide?
- How many variations of the same lib the user has to have on the disk for everything to play?
- How many variations of the same installer your favorite game company is going to release?
- (How many variations of this question I'm going to bother you with?)
- And we still haven't solved the usage of 3rd party libs

gcc and C++11

- C++11 required a few ABI incompatible changes
- std::string can’t use COW (copy on write)
  - Complexity of write to a char must be O(1)
  - Thread safety is required
- std::list must store the size
  - std::list::size() must be O(1) action
- gcc (actually libstdc++) had to break ABI!
- It took a long time to find a solution

gcc and C++11 – abi_tag

- gcc introduced abi_tag
- An attribute that can be added to a type
- It becomes part of the mangling of it
- Kind of inline namespace…
- … with a twist
- The abi_tag is automatically applied to every function that returns this type
- Warning when a type includes subobject with abi_tag but the type itself isn’t tagged
    - Making it viral

abi_tag – what it doesn’t solve

- It prevents silence break, everything becomes name mangling incompatibility ☺
- But if the library doesn’t load, the user still suffers ☹
  - (as discussed)
- Still can’t afford ABI change

Modules solve everything!!1!

- Nope ☹
- Modules don’t affect linkage
- They make it harder to create ODR violation in the same source
- They have no effect when two different DLLs use two versions of the same module

References

References

- National Body comments for C++17 (see GB 61, FI 8, FI 18)
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4664.pdf
- Mangling dependent parameter types, or, what happened to std::default_order (Arthur O'Dwyer's blog)
  - https://quuxplusone.github.io/blog/2019/08/08/why-default-order-failed/
- Ordered By Default (std::default_order proposal)
  - http://wg21.link/p0181
- r/cpp comment on Prague trip report about ABI break in Gentoo
  - https://www.reddit.com/r/cpp/comments/f47x4o/202002_prague_iso_c_committee_trip_report_c20_is/fhpcds8/

References

- Titus Winters papers from 2020-01 pre-Prague mailing:
  - ABI – Now or Never – http://wg21.link/p1863
  - What is ABI, and What Should WG21 Do About It? – http://wg21.link/p2028
- CppCon 2019: Chandler Carruth “There Are No Zero-cost Abstractions” - https://youtu.be/rHIkrotSwcc
- abi_tag, gcc documentation
  - https://gcc.gnu.org/onlinedocs/gcc/C_002b_002b-Attributes.html
- Dual ABI (C++11 situation), gcc documentation
  - https://gcc.gnu.org/onlinedocs/libstdc++/manual/using_dual_abi.html