ABI comparison with reflection

Reply-to: Saksham Sharma <saksham0808@gmail.com>

Project: Programming Language C++
Date: 2024-02-14
Audience: SG7, LEWG
Document #: P3095R0

Abstract

This paper describes a usage of the reflection API introduced in P2996, for comparing equality of ABI of a type / layout of data between two processes, communicating either over a network or across different modules (perhaps built separately) within the same process (eg: two python extension modules in a single python run).

We propose creating a recursive hash of the ABI by hashing all information whose equality may be relevant when communicating across a boundary where ABI comes into play. We go into the depths of this in this paper. We also discuss a config struct which can be used to compute the hash for different usages and ways to keep this hash forward and backward compatible.

In addition, we discuss other possible approaches to this problem and their trade-offs. We also discuss how standardization of this system as a library may benefit the ecosystem.

Introduction

This paper is meant to demonstrate how reflection P2996 can be used to solve the problem of ABI compatibility of messages and objects passed across various modules communicating with each other. The communication may be across language boundaries (C++ => Python => C++) or over network. Historically a lot of source-generation based tools (think protobuf) have attempted to provide third-party solutions for this, but we will demonstrate how reflection (in particular, value based reflection as explored in P2996) can solve a pertinent problem in this domain.

We will aim to develop a set of meta functions we can use to assert ABI compatibility across modules which are communicating with some data objects. We will be using value based reflection as demonstrated in P2996R1 - Reflection for C++26.

We will constrain ourselves to systems where one process / module (we shall call it ***publisher***) is creating a memory layout to be passed to another process / module (we shall call it ***subscriber***). The subscriber wants to know if the ***data*** sent over by the publisher can be used in the form that the subscriber's code expects to use it.

Two common occurrences of this set of problems are

- Inter-process communication over network / shared memory. A subscriber may care if the fields they want are present and of the same type or not. It may also care about the size of the message.
- Python bindings where one python extension module (written in C++) may create a struct which is passed to a different python extension module. Ostensibly compiled by a different person at an entirely different time with nothing shared except perhaps a header.

- An example may be the numpy library. A user may write their own python extension library that operates on a numpy array created by an entirely different system. Both these C++ modules are glued / connected via a Python process.
- Since the two C++ modules may be in the same process, it is possible to call function pointers of functions in module 1 from module 2. Does this mean we can do virtual functions across modules? Is this safe?

The proposed technique is, using P2996's API, creating a hash of the memory layout of a struct that is being communicated across module boundaries to add a layer of safety in their communication. The proposed API would look like the following with some configurability:

```
constexpr auto HashConfig = meta::abi::ABIHashingConfig{.include_indirections = true};
size_t hash = std::meta::abi::get_abi_hash<MyStruct, HashConfig>();
```

Discussion

Alternative approaches

Manual versioning

Manually versioning a struct you're passing over the network is perhaps the simplest and most common technique by some measure. Developers can just reserve some bytes in their message for a human-maintained "version" field which is compiled into the binary, and can be changed each time a meaningful change is made to the message's layout.

It, however, does not capture compilation-related differences in the publisher and subscriber modules, nor does it do anything to prevent manual oversights (a struct was changed but the version was not incremented).

Protobufs / Apache Avro / other systems

They have a big advantage in that they have cross-language support and are very well documented systems already. That said, these systems are not native to the C++ language and thus require some sort of porting-over of the data's schema into a new language. For a fully native C++ solution, it may be preferable to remain within the language's domain.

Additionally, this does not allow using native C++ features like virtual functions, nor does it allow users to tweak the memory layout in ways they may be familiar with (for example, __attribute__((packed))).

A tangential point is that with reflection as proposed in P2996 and perhaps a few more additions (user defined attributes), we may be able to generate protobuf and avro schemas from native C++ code eventually, similar to how P2911R1 generates python bindings.

Compare the full memory layout serialized as a JSON

This is essentially what Avro does, packing the schema in the message at the start during the handshake process. This has its own place, as the next section demonstrates, but in the absence of handshaking, or when we want something really lightweight (in terms of bytes or overhead), a simple hash of the memory layout might just be enough.

That said, a full representation of some data's memory layout still has its place, but is better done with an ecosystem around it (to ease compatibility in other languages), something that protobuf and Avro already do.

Why ABI hashing

An ABI hashing system may be useful in this problem compared to the solutions mentioned above, due to performance and storage considerations:

- Networked systems that communicate over a UDP network may not have a handshake step available to communicate the full object description before communicating without additional overhead. Using a few bytes per message to communicate an ABI should be a good solution that solves most common cases.
- Python extension module libraries like pybind11 and boost::python may want to introduce an unconditional assertion that compares the relatively small ABI hash of a given type between two extension modules, before it allows a type-cast from “extension_module1::MyClass1” to “extension_module2::MyClass2”. Using a larger comparison object here (example: a full class description) may introduce some performance penalties:
    - To be fair it would ideally only have to do that comparison once and later just remember which type conversions are safely allowed.
- When serializing data, ABI hashes may be useful in reducing the amount of bytes that have to be dumped. For large data structures this may reduce the overhead of the serialization format. This point is closely related to the UDP point from before.

Overall, creating a hash of an object's memory layout is not a novel concept, but it is a worthwhile and simple technique that can go a long way in solving some use-cases in a very simple manner.

Decision tree - tasks to solutions

This section serves as the guide rail for the following sections, and covers a number of common tasks that people often have to do that may be assisted with reflection.

1. Inter-process communication (limited to POD data)
    a. **Receiver of data knows the ABI, just wants to validate**

        i. Solution: ABI hashing
            1. **Receiver can handshake with sender to convey what all elements to hash** (member names? member types certainly)
                - Solution: Sender sends the ABI hash computed in the manner the receiver wants.
            2. No handshake is possible, data flow is one-way
                - Solution: Sender sends N different ABI hashes, one each for each kind of hashing technique that is supported.
    b. **Receiver of data is unaware of the ABI, wants to know how to parse the incoming data**
        i. Solution: Sender sends over a fully specified description of the ABI in a premeditated format. We could call this premeditated format the ABI of the ABI, but only for a limited subset of the language. Similar to Avro's schema.
2. Intra-process communication
    a. **Sharing POD structs across different shared modules in the same process**
        i. Same decision tree as under point (1)
            1. Libraries that facilitate this can use ABI hashing automatically to allow implicit object type conversions (example: pybind11)
    b. **Sharing non-POD objects across shared modules**
        i. Sharing function pointers via capsules is already achievable in Python, but verbose.
        ii. Solution: ABI hashing schemes can include a methodology of hashing virtual functions.

Use-cases

Check if all the members are in the right place / right type

The subscriber may have hard-coded assumptions about the layout of data in the struct sent over (both of them may have been compiled with the same header). Such a codebase has two advantages:

- It is easy to write the subscriber code. It just has to include the same header as the publisher.
- The subscriber code can be efficient since it can have hardcoded member offsets in the struct.

There is a disadvantage of writing such code though:

- Subscriber code is dependent on the ABI of the publisher's data and is inflexible / has to be recompiled with the new header whenever there is any *significant* (discussed later) change to the header's members. The only way to work around this is to have some runtime overhead by creating a message parser at runtime using a full schema specification (similar to Avro).
- Even if all the fields the subscriber cares about are present at the right place, it must also assert the size of the message is fixed, because inadvertently the subscriber code might have some implicit

assumptions about the size of the message. This can only happen if the API between the subscriber and publisher has some method which returns an array of these messages.

Call virtual functions on objects owned by another module

This is only applicable in situations like inter-language bindings where the modules communicating are running in the same process. In particular, if I have a python extension written in C++ that exposes an object of this sort:

```
C/C++
class MyStateInterface {
 public:
  virtual void mutate() = 0;
  virtual int get() = 0;
};
```

Passing this object to python can be made possible by adding python bindings for this interface class. That will mean all methods in the class are now contained in a python binding. See P2911R1.

Now say we have another python extension module in the same process and it would like to be able to accept objects of this type in its interface methods:

```
C/C++

void process_state(MyStateInterface* state);
```

In this situation, if python was able to deduce that the ABIHash of the MyStateInterface type matches across both modules, it could allow passing this type safely from one extension module to another (of course, intermediated by the binding library's syntax).

Parse the data format to interpret incoming messages

This is a more flexible style of writing the subscriber code, where it can contain code to parse a ABI textual description so it can better decide how to fetch various fields that it is interested in. This would require the ABI description to have a standard textual.

In particular, if there is a way to communicate the full descriptive textual definition of a class in a parsed form from a publisher to a subscriber, the subscriber can choose to adjust their deserialization code in order to efficiently unpack the received fields and pack them into its own internal format. This provides a lot of flexibility in softwares that communicate with each other, since this makes them only loosely coupled. New changes to the underlying data format can be easily kept backwards compatible. Some of these ideas are explored in Avro already.

For communicating between languages, as long as the description of this textual format is well documented, any language can implement a method to unpack the received information.

That said, ABI hashing does not solve this use-case. But we can envision an approach where one could generate Avro schema files from C++ source code, and those schema files can represent the memory layout while communicating over a network.

Implementation experience / proposed functionality

We propose various methods that can help in standardizing ABIs and their communication. All this is implemented and compiling here: https://godbolt.org/z/sndxcK5qj

ABIHashingConfig

We propose two standard structs with room for future expansion:

```
C/C++
// Common usecase optimized with future extensibility
struct ABIHashingConfig {
  static constexpr int MINIMUM_SUPPORTED_VERSION = 0;  // To allow future rollover
  static constexpr int MAXIMUM_SUPPORTED_VERSION = 0;  // To gracefully error out
  uint8_t version : 4 = 0;
  bool include_nsdm_names : 1 = true;
  bool include_indirections : 1 = false;  // Only relevant in intra-process comparison
} __attribute__((packed));

// Virtual ABI hashing use-case kept as an optional extension
enum class VTableHashingMode : uint8_t {
  NONE, SIGNATURE, SIGNATURE_AND_NAMES, NUM_VTABLE_HASHING_MODES
};
enum class VirtualABI : uint8_t { ITANIUM, MSVC };
struct IndirectionABIHashingConfig {
  uint8_t version : 4 = 0;
  VTableHashingMode virtual_hashing_mode : 4 = VTableHashingMode::NONE;
  VirtualABI abi_mode : 2 = VirtualABI::ITANIUM; // Architecture dependent default
} __attribute__((packed));
```

Serialized representation of ABIHashingConfig can be of this form:

- If **include_indirections** is false:
  - If **version** is up to a certain number in the future:
    - Serialized representation is 1 byte.
  - Else for future extensibility
    - We can introduce an extension struct which can be assumed to be present if version is greater than a certain number in the future
- If **include_indirections** is true:
  - At the end of the ABI hash, we can expect a IndirectionABIHashingConfig struct as well. It has similar semantics for the "version" field as the earlier struct.
  - This struct is not optimized for size since it would never be serialized across process boundaries.

We can bump versions if we have to change the hashing methods in the future, or if we change the input to the hashing methods.

The publisher may choose to serialize multiple ABIHashes sequentially in its messages if it is concerned about backwards compatibility with older subscribers. It can keep hashing with old versions as well for some time. Each such version will basically add an overhead of 1 byte for the config struct, and 8 bytes for the hash (or more, if later versions revise this).

We also provide a "MINIMUM_SUPPORTED_VERSION" constant, which can be bumped in the future to deprecate old versions.

Recursive ABI hashing

This is the simplest technique for asserting some confidence on two modules having a compatible ABI. It is very powerful and simple to use, and solves a large number of common problems, without requiring any fancy new features. On the flip side, it is also not 100% accurate due to the presence of hash collisions and must be utilized with that kept in mind.

The idea is simple, each data type that is communicated across a boundary where ABI comes into question, has an ordered list of members which may be of the following different types:

- Primitive data type members (may come through inheritance or not)
- Non-primitive data type members (may come through inheritance or not)

- Bitfield members (need to hash their offsets and sizes correctly)
- Virtual methods in this class (may be hashed optionally, depends on the use case)
- Members which are pointers or references to other data type objects

We posit that the following algorithm can output a hash value that is generally useful to the wide C++ community for a wide variety of problems.

Implementation experience

I was able to implement the proposed ideas in the following manner. Note that albeit the final implementation must formalize the hashing methodology, at present the paper does not aim to optimize things like reduction of the number of unique (and some potentially redundant) hash operations.

```
C/C++
template <typename T, ABIHashingConfig config = ABIHashingConfig{},
          IndirectionABIHashingConfig indirection_cfg = IndirectionABIHashingConfig{}>
consteval size_t get_abi_hash() {
    // TODO: Handle recursive cycles.
    size_t hash = 0;
    // Need to keep a local lambda else it does not recognize it is consteval.
    auto hash_combine = [](size_t seed, size_t v) -> size_t {
        return seed ^ (v + 0x9e3779b9 + (seed << 6) + (seed >> 2));
    };
    [: expand(nonstatic_data_members_of(^T)) :] >> [&]<auto e>{
        constexpr auto elem_type = meta::type_of(e);
        constexpr auto is_indirect_ref = (
            meta::test_type(^std::is_pointer_v, elem_type) ||
            meta::test_type(^std::is_reference_v, elem_type)
        );
        if constexpr (!config.include_indirections && is_indirect_ref) {
            return;
        }

        if constexpr(config.include_nsdm_names) {
            auto name = std::string(meta::name_of(e));
            hash = hash_combine(hash, HASH_STR(name.c_str()));
        }
        if constexpr(meta::is_bit_field(e)) {
            hash = hash_combine(hash, std::meta::offset_of(e));
            hash = hash_combine(hash, std::meta::bit_offset_of(e));
```

```
                hash = hash_combine(hash, std::meta::bit_size_of(e));
            } else {
                hash = hash_combine(hash, std::meta::offset_of(e));
                hash = hash_combine(hash, std::meta::size_of(e));
            }
            if constexpr (meta::test_type(^std::is_class_v, elem_type)) {
                using T2 = typename[:meta::type_of(e):];
                hash = hash_combine(hash, get_abi_hash<T2, config, indirection_cfg>());
            } else {
                if constexpr (meta::test_type(^std::is_pointer_v, elem_type)) {
                    if constexpr(config.include_indirections) {
                        // Hash the underlying type's ABI recursively.
                        constexpr auto ctype = std::meta::substitute(
                            ^std::remove_pointer, {elem_type});
                        using T2 = typename[:ctype:];
                        hash = hash_combine(hash, get_abi_hash<T2, config, indirection_cfg>());
                    }
                } else if constexpr (meta::test_type(^std::is_reference_v, elem_type)) {
                    if constexpr(config.include_indirections) {
                        // Hash the underlying type's ABI recursively.
                        constexpr auto ctype = std::meta::substitute(
                            ^std::remove_reference, {elem_type});
                        using T2 = typename[:ctype:];
                        hash = hash_combine(hash, get_abi_hash<T2, config, indirection_cfg>());
                    }
                } else {
                    // Returns empty string as of right now, perhaps there is a better way?
                    // Debate if this is even useful? Since we hashed the size and name earlier.
                    auto name = std::string(meta::name_of(meta::type_of(e)));
                    hash = hash_combine(hash, HASH_STR(name.c_str()));
                }
            }
        };
        if constexpr(config.include_indirections) {
            if constexpr(indirection_cfg.virtual_hashing_mode != VTableHashingMode::NONE) {
                // TODO
            }
        }
```

```
    // Attributes like `packed` may change the size of the struct.
    // Not everyone would be concerned with padding at the end though.
    // Can consider making this optional via a config param.
    hash = hash_combine(hash, sizeof(T));
    return hash;
}
```

As for implementation of the HASH_STR function, a constexpr version of that will work for our use-cases.

```C/C++
// See https://stackoverflow.com/a/9842857 for implementation
// Better implementations are possible
// Full implementation can be found here: https://godbolt.org/z/sndxcK5qj
#define HASH_STR(x) (crc32<sizeof(x) - 2>(x) ^ 0xFFFFFFFF)
```

cvref / pointers

One could argue that recursing into types which are referred to via pointers or references are irrelevant when hashing the ABI. "constness" is certainly irrelevant. But pointers / references might still deserve a recursion if the object is being shared within the same process (two python extension modules communicating).

A question that may come up is, in cases like pointers, indirections of memory layout, and such, what is the software contract between the two modules sharing this object? To elaborate, who manages this memory and who is allowed to modify it? And can reflection have anything to do with that? My view is that such ideas must be conveyed through the function names / API names that pass these objects between the two modules. For instance, if an object wants to transfer ownership of a dynamically allocated memory to a different module, it must convey this through the function name, and this should not be included in the ABI of the modules, and thus is ignored from the hash (That said, transferring ownership this way would certainly require this module to leak some memory which is questionable at best).

vtable internals

The code to hash the virtual function ordering has been left out in the above implementation, but it can be implemented in a relatively easy manner.

- Iterate all base classes of type T in order, and recursively do the following:
    - Iterate all members in order to find virtual methods via is_virtual (pure or not).
    - For each virtual method, first hash the name of the method.

    - Then, for each type in the virtual method’s signature, hash it recursively using this algorithm.

Cycles in hashing

Once include_indirections is enabled, thanks to forward declarations we could easily have cycles in the algorithm. To avoid this, the get_abi_hash method must keep a set of all types that are being hashed right now, to avoid an infinite loop. This is not yet implemented in the linked implementation but can be done using a variadic template function.

Enums

Computing the hash of enums can be widely useful, and is pretty simple to accomplish without much effort. It does not allow forwards or backwards compatibility though in its most basic form. In most cases users would want to publish the enum's full map as a JSON or equivalent format across processes and validate the required values only. Constructing that representation is simple with the examples from P2996R1 and is omitted here.

One interesting topic pending further thought is, if a type that we are hashing contains an element that is an enum, perhaps we should include the hash of the enum in that? That may indicate that the get_abi_hash method might be augmented to include the enum class' hash as well.

Standardization

The techniques we propose here can be implemented plainly on top of value based reflection by any user. As such, this does not present any obstacles to the standardization of reflection (P2996) in any manner. That said, there is a case to be made for standardizing these techniques.

Benefits

If the programmatic description of the ABI of a data type can be standardized, open source software authors can start to depend on it safely, massively reducing software code that has to check validity of various information received from a publisher. In the long run this can be a very valuable addition to the open source ecosystem.

Downsides

The ABI hash or the ABI textual description will itself become a part of the ABI or API of the language itself if we go down this route. So if in the future one of these has to evolve, it will cause an ABI break of a different manner. This can be addressed in a different form:

- ABI breakages ought to only happen with new language standards, since that is when it is most likely for the structure of a struct or class to change. Many C++ modules already use standard library objects as a part of their API, which means such modules are already subject to ABI breakage on new standards releases. Thus it follows that they must already tag the standard in their release / deployment process, making this concern slightly less concerning.
- Changes to hash functions / textual representation must be rolled out in a backwards compatible manner. If we ever need to make such changes, the language must offer a way to continue to compute the older version of the hash / textual description, such that library authors can bundle both old and new versions of the hash / textual description in their library's API.

Our discussion in the section on ABIHashingConfig mentions an approach where the ABIHashingConfig itself is bundled with the ABIHash, allowing most applications to remain forwards and backwards compatible. For applications where the message size is important and 8 more bytes are not acceptable, they may have to accept an ABI break if versions of this ABI hashing technique mismatch.

Additionally, we must notice that the hashing choices we have made here are rather sweeping (example:" include all or none of the data member names in the hash). These sweeping choices may not solve every use-case, but the hope is that we solve 90% of the use-cases in a natural and efficient manner.

Cross language support

Modules written in other languages may wish to compute the expected hash of a struct on their end as well. To that point, C++ may benefit from keeping in mind that exposing an API for the get_abi_hash function in different languages (with a different form of input, even if it is runtime) would be beneficial. This can be implemented by the community once a standard hashing scheme is decided upon, for each version of the hashing config.

Questions / next steps

- Is ABI hashing using reflection a worthwhile proposal for:
  - The standard library?
  - A non-standard utility (Boost?)
- Should libraries like Avro and Protobuf be able to take advantage of P2996 to reduce boilerplate for C++ users?
- Should python bindings generator systems use an ABI hashing technique like this to make type conversions across modules safer? (As of right now, it is not possible making all loaded symbols globally visible since type checking is really strict at the moment)

Acknowledgements

Thanks to Inbal Levi for introducing me to the discussion on Reflection, and to Jagrut Dave, Dan Katz, and Adam Lach for their papers which led to this thought process, and for their comments during the initial discussion on this idea.

Citations

- P2996 - Reflection for C++26 - Childers, Dimov, Revzin, Sutton, Vali, Vandevoorde. 2023 https://isocpp.org/files/papers/P2996R1.html
- P2320 - The Syntax of Static Reflection - Sutton, Childers, Vandevoorde https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2320r0.pdf
- N3980 - Types don't know - H. Hinnant, V. Falco, J. Byteway. 2014-05-24. https://wg21.link/n3980
- P2911R1 - Python Bindings with Value-Based Reflection - Lach, Adam, and Jagrut Dave. 2023. https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2023/p2911r1.pdf
- P1240 - Scalable Reflection in C++ - Sutton, Vali, Vandevoorde https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p1240r0.pdf