P2123R0
Extending the Type System to Provide API and ABI Flexibility

Presentation for LWEG on 2021-05-11

Motivation

- Our community wants the flexibility to change the implementation details and interfaces of types in order to realize performance benefits and otherwise enhance functionality.
- Our community also wants the interface stability necessary to compose software from compiled code created by multiple entities over long time spans.
- C++ does not currently provide the language-level tools necessary in order to express different points in this flexibility vs. stability space.

ABI

- In practice, ABIs are composed from different architecture/system-specified pieces, including:
  - The C ABI: structure layouts, calling conventions, TLS implementation details, and so on. Might only apply to extern "C" entities, but is often used for many C++ entities as well.
  - C++ name-mangling rules (i.e., how to encode the overload-identifying name of a function as a symbol name).
  - The representation of C++ primitives (e.g., member pointers).
  - The layout of aggregates that don’t have direct C analogues (e.g., have base classes, virtual functions).
  - The layout of virtual-function tables, RTTI information, and other implementation-internal metadata.
  - The method used to implement exception handling.
- When discussing this issue, we sometimes end up discussing both:
  - The possibility of changing existing types in a way that is ABI-incompatible with existing implementations of those types.
  - The possibility of changing/extending the ABI itself in order to improve functionality.

ABI-Incompatible Changes

- What kinds of changes to a type are ABI-incompatible in this sense? These include:
  - Changing the size of the type, the layout of the type (e.g., the order of the members, alignment, adding [[no_unique_address]], modifying base classes, adding the first virtual function).
  - Changing the size or contents of the virtual-function table (e.g., changing the order in which virtual functions are defined, adding a new virtual function).
  - Changing the return type of a function.
- There are some ABI-changing modifications one might make to a class that are not necessarily ABI incompatible, such as changing the type of a function, because an implementation might use the "provide both the new and old symbol" technique to simultaneously serve both old and new clients.
- Changes to the ABI itself, such as changing the calling convention (e.g., how many registers are used for parameter passing, which register holds the this pointer), are also part of the conversation around ABI changes and might be something a facility for dealing with ABI changes wishes to address.

On extern “C”

- It is also important to be clear about whether we intend only to allow different, ABI-incompatible types of the same name to exist in the same program, which is currently often impossible because of inline-linkage definitions and other shared data (e.g., virtual-function tables), or whether we intend to allow for some translation unit to use both of these types simultaneously.
- It has been suggested that we need to only handle the former case and not the latter. While this may be a helpful step in the right direction, it is not clearly sufficient because this restricts us to a programming model where different parts of the application must use "C" interfaces to communicate across ABI boundaries, including using "C" types to marshal all data across these boundaries (instead of using the C++ types directly).

On Costs

- Dealing with multiple implementations of library types nearly always imposes some kind of cost. Language-level facilities in this space may, and likely should, provide the programmer with a way to incur / trade off these costs in a way that makes sense for each particular application. A programmer might trade off the costs of:
  - Indirection (i.e., performance overhead associated with dynamically-resolved abstraction)
  - Specialization (i.e., overheads associated with generating multiple specializatons)
  - Copying (i.e., the overheads associated with explicitly transitioning between data structures by copying data between one and the other)
  - Technical debt (i.e., only using older types in the application, thus preventing the use of newer features)
  - Ecosystem isolation (i.e., only using the newest types, perhaps preventing integration with other libraries and services)

Some Past Experience

- Some C++ standard-library implementations, such as libc++: inline namespaces: std::vector is really std::__1::vector
  - Can provide unique symbol names when using types from different (inline) namespaces.
  - The inline namespace technique is not viral: it affects only direct uses of the standard-library types. User types that use standard-library types are not automatically tied to the same versioning scheme, and so, the inline namespace technique does not help user-defined types in the same way.
  - On some systems, it doesn't even help standard-library types if they're used only as return types.
  - It's not clear how we might use this technique to manage ABI transitions directly.
    - Overload sets:
      - void foo(std::cxx20::vector<int> &x);
      - // Is this needed or ambiguous with the definition above?
      - void foo(std::cxx23::vector<int> &x);

Some Past Experience

- GCC: viral “ABI tags”, used for the C++11 std::string transition.
- These tags affect the mangling, but also affect the mangling of types that use the tagged types.
  - Unclear what should happen if multiple tags are used.
  - Care is required that forward-declared types are properly annotated.
  - May also need to tag inline-linkage entities based everything used in their definitions, not just those types that appear on the interface.
  - Does not address the problem of allowing the user to name both types in the same translation unit.

Just Recompile!

- Not all users can recompile the “world”. Most can’t.
- Some users can recompile their code as desired, but depend on system libraries and/or system interfaces provided only using some older ABI.
- Some users can recompile their code but depend on third-party libraries that they cannot recompile.
- Some users can recompile their code, but their code is a plugin to a third-party application with a fixed ABI.
- Some users can recompile their code, but their code must load plugins that use the older ABI.
- Some users can recompile their code, but doing so along with a language-version upgrade is a slow, expensive process if it must happen synchronously across the entire organization.

Does It Help With...

- std::regex? - Maybe. Do you just want to improve the algorithms or also change the semantics?
- std::unordered_map? Same. For loosening iterator-stability guarantees? No. For improving the implementation in other ways? Yes.
- std::unique_ptr? Probably. Actually realizing the benefits also requires other changes, however (see the paper).

Requirements

- C++ must default to providing flexibility over interface stability. If the programmer does nothing explicit, then they should get the highest possible performance (relative to that provided by other aspects of this proposal). This includes uses of standard-library types and functions.
- C++ must provide a way for types to provide interfaces that prioritize stability over flexibility. This must allow code written using a newer version of C++ to call functions in code written in an ABI-incompatible older version of C++, and code written in an older version of C++ to call functions in code written in an ABI-incompatible newer version of C++. It must be possible to implement this feature such that the ABI of pre-C++-23 code can be selected as required.
- This facility must be generalized, and not specific to different C++ versions. We recognize that the C++ standard library is not the only library with ABI concerns.
- This facility should work naturally with modules and should strive to not require viral annotations while retaining the ability to make expensive operations explicitly visible to readers of the code.

Some Questions

- Does stability imply a lack of inlining? Note that there is a potential separation here between inline linkage and actually allowing compiler inlining. Allowing inlining potentially requires "virtual" data-member access (i.e., making the set of internal data members used by these inline functions part of the abstract, stable interface).
- Does stability imply only additive changes? Adding new functions to the interface? Adding new data members? How about changing the type of existing members? Type changes that change the size vs. type changes that don’t change the size? Replacing a data member with a function of other data members?

“Simple” Wrappers Don’t Work

- “Simple” wrappers don’t work because there is no generic way to tie the lifetime of the wrapper to the underlying object: no **generic** way to wrap!

```
void resilient_abi(cxx20::interface::thing<int> *);
void call_resilient_abi(std::thing<int> *t) {
  // Assuming for simplicity that we know that t is not nullptr...

  // Should we use the copy or the move constructor of thing_wrapper here?
  std::interface::thing_wrapper<int> *w = new std::interface::thing_wrapper<int>(*t);
  resilient_abi(w);
  // Hrmm... should I delete w now? Should I delete t?
}
```

Alternate Library Design

- An alternative is to create two different variants of each type, one normal one, and one also holds a wrapper object of itself.
- See paper for some code showing how this might look.
  - Spoiler: it’s big, complicated, ugly, and still not as efficient as a language-level feature might be (because the compiler can construct offset tables instead of using virtual functions returning data pointers).

Fat Pointers

- Other languages use “fat pointers”
  - Go: go’s interface types are runtime generated fat pointers containing a pointer to the object and a pointer to a fixed-offset table of function pointers that can be referenced just like a vtable in the receiver, but are generated by a linear search of a sorted list of function signatures by name (literally by string comparisons, at runtime). It also uses a fat pointer setup for slices.
  - Swift: Swift uses vtables in all objects by default, and virtual methods actually fall back on a runtime message resolution like Objective-C uses, allowing runtime addition and proxying of methods as well as some interesting resilience and ABI stability enhancements on otherwise plain objects. Where fat pointers come in, in this case implemented by passing an extra implicit table argument in addition to an object, in Protocol objects and APIs. A function that takes a protocol object gets a fixed-offset table of pointers to the methods, and possibly data, that are listed in the protocol specific to the object passed in.
  - Rust: Types are static by default, generics work essentially like templates. Fat pointers are used in a couple of different ways, first they’re used for references to runtime-sized slice types. More significantly, they’re used to implement trait objects. It allows an API writer to select either entirely static replicated implementation with generics and trait bounds or the indirected trait object that has some runtime overhead but will work with any type that implements the protocol now or in the future.

How Do I?

- I have code compiled for C++20 that uses modules. How can I use it in C++23 code?

```
import SomeCXX20Code;
// Now use the exported things as you would expect.
```

How Do I?

- I have code compiled for C++20 that does not use modules. How can I use it in C++23 code?

```
#include <stdinterface>

interface(std::cxx20) {

#include <SomeCXX20Code.h>

}

// Now use the included things as you would expect.
```

How Do I?

- How can the standard library write a class that provides it's existing external interface to C++20 clients, and a new interface to C++23 clients?

```
#include <stdinterface>
...
struct point {
  interface(std::cxx20) {
    int x, y, z;
    interface(std::cxx23) int w;
    int get_x() const { return x; }
    int get_w() const interface(std::cxx23) { return w; }
    interface(std::cxx23) { bool only_for_cxx23() { return true; } }
  }
};
// sizeof(interface(std::cxx20) point) == 12.
// sizeof(interface(std::cxx23) point) == 16;
```

The Proposal

- Interface Tags

```
// Declares an interface tag:
interface v1;
// Declares an interface tag that inherits from the one above.
interface v2 : v1;

namespace std {
  interface cxx20;
  interface cxx23 : cxx20; }
```

The Proposal

- Interface Blocks

```
namespace foo {
  interface(v1) { struct x; }
  interface(v2) {
    struct y; struct x : public y;
  }
  struct z {
    interface(v1) { int x; }
    interface(v2) { int x; int y; }
  };
}
```

The Proposal

- Interface Qualifiers

// A qualifier for the interface tag v1.

interface(v1)

// A qualifier for the interface tag v1 or any interface inheriting from it.

// This is called an unresolved interface.

interface(v1+)

The Proposal

- Unresolved Interfaces
  - Entities cannot have more than one interface, although if an entity has an interface qualifier (explicit or implicit) for both an interface and an interface from which it inherits, the parent interface qualifier is ignored.
  - Rules to ensure interface additivity.
  - Types with unresolved interfaces are distinct from types with resolved interfaces to the same base interface tags.
  - Variables with types with unresolved interfaces may hold values of the same base type and any interface tag that (transitively) inherits from the provided base interface or the base interface itself.
  - Variables with types with unresolved interfaces may hold values of the same base type and any interface tag that (transitively) inherits from the provided base interface or the base interface itself.
  - All interface-public members can be accessed from a type with an unresolved interface, but all types on those interfaces will be act as though they’re also unresolved. If any of these types do not support use with unresolved interfaces, the program is illformed.