Interconvertible object representations

Timur Doumler (papers@timur.audio)

Document #: P1912R0
Date: 2019-10-06
Project: Programming Language C++
Audience: Evolution Working Group

Abstract

We propose a new specifier declaring that a given type `T1` has an object representation compatible with some other type `T2`, in cases where the implementation can verify this at compile time. This allows to interconvert pointers and references to those types without invoking undefined behaviour. This facility plugs a very unfortunate hole in the current C++ type system.

As a side benefit, our proposed facility regularises the unusual interconvertibility properties of `std::complex`, such that it is no longer a "magic" type unimplementable without special compiler support.

1 Motivation

It is often useful, especially with fundamental types, to observe the object representation (i.e. the bytes) of an object of type `T1` as if it were the representation of an object of a different type `T2`. Such techniques are sometimes called *type punning* and are widely used in performance-sensitive C++ programs. In some cases, they are essential to achieve acceptable performance.

Typically, this is done using `reinterpret_cast` or `union`. Often, such code compiles fine on all major compilers and achieves the desired results, but is undefined behaviour according to the C++ standard, violating either aliasing rules, object lifetime rules, or both. This very unfortunate situation arises over and over again due to certain holes in the C++ type system.

Recent proposals attempt to plug several of these holes, giving programmers tools to achieve the desired result without running into undefined behaviour. However, one important hole still remains. This paper proposes to fix that last hole.

1.1 Related proposals

[P0476R2], adopted for C++20, introduces `std::bit_cast` as a portable means to type-pun between objects of two types `T1` and `T2`, as long as they are trivially copyable and have equal sizes:

```
float fast_inverse_sqrt(float y)
{
    auto i = std::bit_cast<int>(y);    // type-pun float to int without UB
    i  = 0x5f3759df - ( i >> 1 );
    y = std::bit_cast<float>(i);    // type-pun int back to float without UB
    // ...
}
```

[P0593R4] proposes a mechanism to implicitly create an object of type `T` from a sequence of bytes that holds a valid representation of such an object. This allows conversions which would be undefined behaviour in the current C++ standard:

```
void process(Stream* stream)
{
    std::unique_ptr<char[]> buffer = stream->read();
    auto* foo = reinterpret_cast<Widget*>(buffer.get());// type-pun char[] to Foo without UB
    process_foo(foo);
}
```

Finally, [P1839R0] proposes a fix to the object model that allows the reverse: directly accessing the sequence of bytes that makes up the representation of an existing object of type `T`. This would be again undefined behaviour in the current C++ standard:

```
void print_bytes(float f)
{
    auto* bytes = reinterpret_cast<unsigned char*>(&f);
    for (int i = 0; i < sizeof(float); ++i)
        std::cout << bytes[i];     // print byte i of representation of f without UB
}
```

1.2 The remaining type system hole

However, there remains an important use case that is not addressed by any of those proposals. Consider a standard-layout type such as

```
struct two_floats
{
    float x, y;
};
```

On all platforms known to us, the object representation of such a type will be identical to that of an array of two floats. However, if we wish to access it as such, we immediately invoke undefined behaviour:

```
two_floats* tf = read_data();
auto* f = reinterpret_cast<float*>(tf);
std::cout << f[0] << ',' << f[1];    // UB! Can't do pointer arithmetic on f
```

According to the current language rules, `two_floats` and `float[2]` are not pointer-interconvertible. `std::bit_cast` does not help at all in this case.

In real life, the need for such casts arises e.g. when working with packs of fundamental types that interoperate with a SIMD type such as `__m128`. To pass a data stream across an API boundary, it is often necessary to cast between an array of such packs and an array of objects of the underlying fundamental type.

2 The curious case of `std::complex`

Curiously, the C++ standard already allows exactly the kind of interconvertibility that we are proposing, but only for one specific class in the standard library: `std::complex`. The wording in [complex.numbers]p4 says:

> If `z` is an lvalue of type *cv* `complex<T>` then:
>
> — the expression `reinterpret_cast<`*cv*` T(&)[2]>(z)` shall be well-formed,
>
> — `reinterpret_cast<`*cv*` T(&)[2]>(z)[0]` shall designate the real part of `z`, and
>
> — `reinterpret_cast<`*cv*` T(&)[2]>(z)[1]` shall designate the imaginary part of `z`.

This is interesting, because it allows interconvertibility between `std::complex<T>` and `T[2]` by fiat, regardless of how `std::complex<T>` is defined, because the implementation "knows" that the two types have the same layout.

If `std::complex<T>` is implemented as having a data member of type `T[2]`, the above cast would work due to existing pointer-interconvertibility rules, which allow to type-pun between a standard-layout class object and its first non-static data member.

However, if `std::complex<T>` is implemented as having two data members of type `T`, like our `struct two_floats` above (notably, the libc++ implementation of `std::complex` does exactly this), there is no way to reconcile the above requirement on `std::complex` with the language rules, except by allowing the specific casts *by fiat* for this specific type. This is exactly what the C++ standard is doing here – arguably, an unfortunate solution.

The subsequent part of the wording in [complex.numbers]p4 is even more interesting:

> Moreover, if `a` is an expression of type *cv* `complex<T>*` and the expression `a[i]` is well-defined for an integer expression `i`, then:
>
> — `reinterpret_cast<`*cv* `T*>(a)[2*i]` shall designate the real part of `a[i]`, and
>
> — `reinterpret_cast<`*cv* `T*>(a)[2*i + 1]` shall designate the imaginary part of `a[i]`.

This essentially allows to arbitrarily pointer-interconvert between `std::complex<T>*` and `T*`. Unlike the first part of [complex.numbers]p4, there is no way to make this work with the current C++ language rules *regardless* of how `std::complex<T>` is implemented under the hood, except by fiat for this specific type.

Crucially, taken together, the conversions above are exactly what we want to allow for our user-defined types such as `two_floats`. Our task is therefore to turn these rules that currently exist only for `std::complex` into a generic tool that a C++ programmer can use for their own types.

3 Proposed solution

We propose a new specifier that can be added to the declaration of a type to specify an *object representation compatibility requirement*. The actual spelling of such a specifier is of course subject to bikeshedding. In the meantime, we use a new keyword `layoutas(`*`type-id`*`)` as a placeholder for the final spelling, since the new specifier has properties similar to the existing `alignas` specifier specifying an alignment requirement, and should occupy a similar place in the grammar. Using this temporary placeholder syntax, we can now express our intent as follows:

```
struct two_floats layoutas(float[2])
{
    float x, y;
};
```

This declares that an object of type `two_floats` has an object representation compatible with that of an object of type `float[2]`. If the implementation cannot prove that this is actually the case, the program is ill-formed. Otherwise, we can use this information to allow for relaxed interconvertibility rules to allow a pointer or reference to an object of type `two_floats` to be cast to a pointer or reference to `float[2]`.

In general, if a type `T1` is declared with `layoutas(T2)`, we can say that an object of type `T1` is *object-representation-compatible* with an object of type `T2`, and allow the following conversions:

— If `z` is an lvalue of type *cv* `T1`, then `reinterpret_cast<`*cv* `T2&>(z)` shall be well-formed,

— If `a` is an expression of type *cv* `T1*` and the expression `a[i]` is well-defined for an integer expression `i`, then `reinterpret_cast<`*cv* `T2*>(a)` shall be well-formed.

We can make that relation transitive by saying that, further, an object a is object-representation-compatible with an object b, if there exists an object c such that a is object-representation-compatible with c, and c is object-representation-compatible with b.

Obviously, two objects are object-representation-compatible if they are the same object.

Finally, we can remove the wording in [complex.numbers]p4, and instead simply add the specifier `layoutas(T[2])` to the definition of `std::complex`. As a result, `std::complex` is no longer a "magic" type unimplementable without special compiler support.

Note that `layoutas` cannot be an attribute, because the removal of an attribute is not allowed to alter the semantic meaning of a valid C++ program.

4 Wording

The formal wording for this proposal will be provided in a future revision.

Acknowledgements

Many thanks to Fabian Renn-Giles and Richard Smith for encouraging this proposal.

References

[P0476R2] JF Bastien. Bit-casting object representations. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/p0476r2.html, 2017-11-10.

[P0593R4] Richard Smith. Implicit creation of objects for low-level object manipulation. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p0593r4.html, 2019-06-16.

[P1839R0] Krystian Stasiowski. Accessing Object Representations. http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2019/p1839r0.pdf, 2019-07-30.