Types with array-like object representations

Timur Doumler (papers@timur.audio)

Document #:	P1912R1
Date:	2020-01-13
Project:	Programming Language C++
Audience:	Evolution Working Group

Abstract

We propose a new specifier declaring that a given standard-layout type `T` with data members of type `U` has an object representation compatible with an array of `U`s. This allows to convert pointers and references to those types to those of an array 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.

2 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
}
```

3 The remaining type system hole

However, there remains an important use case that is not addressed by any of those proposals.

3.1 No pointer-interconvertibility

Consider a standard-layout type such as

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

The object representation of such a type will typically 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.

Another, related use case is representing numeric types like complex numbers, quaternions, etc. as user-defined types, and then passing them on to a C API such as the GNU scientific library which might expect arrays instead. Again, the existing practice is to use `reinterpret_cast` which is currently undefined behaviour.

3.2 No layout-compatibility

A similar situation occurs when using unions instead of `reinterpret_cast` to pun between the types. Consider the following class (which actually exists in the Qt framework):

```
struct QColor {
    union {
        struct {
            ushort alpha;
            ushort red;
            ushort green;
            ushort blue;
            ushort pad;
        } argb;

        struct {
            ushort alpha;
            ushort hue;
            ushort saturation;
            ushort value;
            ushort pad;
        } ahsv;

        // more structs laid out in the same way...

        ushort array[5];
    } ct;
};
```

This class relies on the user being able to type-pun between the structs `argb`, `ahsv`, etc. on the one hand, and the `ushort[5]` array on the other hand. In practice, this has worked for many years, but as far as the standard is concerned, this is undefined behaviour, as the structs and the array are not layout-compatible, even though they typically have identical object representations. Writing to `argb` followed by reading from the non-active union member `array` is therefore undefined behaviour.

4 The curious case of `std::complex`

Curiously, the C++ standard already allows exactly the kind of interconvertibility that we are proposing (at least for the `reinterpret_cast` case), 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` and `QColor`. 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, and to extend them to layout-compatibility as well as pointer-interconvertibility, so that they work with the union technique as well as with `reinterpret_cast`.

5 Proposed solution

We propose a new specifier that can be added to the declaration of a standard-layout type to specify that the type has an *array-like object representation*. The actual spelling of such a specifier is of course subject to bikeshedding. In the meantime, we use a new keyword `layoutas(`*`type-id`*`)`, where *`type-id`* names an array type, as a placeholder for the final spelling. This 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 a `float[2]`. If the implementation cannot guarantee this, for example because the non-static data members of the type are not exactly two `float`s, the program is ill-formed. Otherwise, we can use this information to allow for relaxed interconvertibility rules, such that `two_floats` and `float[2]` become pointer-interconvertible and layout-compatible.

We can now print all the floats in an array of `two_floats` objects like this, without invoking undefined behaviour:

```
two_floats tfs[100];

for (int i = 0; i < 200; ++i)
    std::cout << reinterpret_cast<float*>(&tfs)[i] << '\n';  // now OK
```

just in the same way it was already allowed for `std::complex`.

Similarly, the class `QColor` can now be declared as follows:

```
struct QColor {
    union {
        struct layoutas(ushort[5]) {
            ushort alpha;
            ushort red;
            ushort green;
            ushort blue;
            ushort pad;
        } argb;

        // more structs laid out in the same way...

        ushort array[5];
    } ct;
};
```

This now makes the members `argb` and `array` layout-compatible, and removes the undefined behaviour present in current programs that use this class.

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, whereas in this case, the removal of `layoutas` may add undefined behaviour to the program.

It is important that this paper does not propose to allow circumventing the type system and the object lifetime rules. It will not allow the user to arbitrarily type-pun between two unrelated types. We propose to only allow this for the case where the other type is an array. The mental model is that the actual underlying object, which exists in memory and is within its lifetime, is the array. The user-defined type simply provides alternative names to the elements of the array through its data members. The user-defined type will be treated as an array for ABI purposes. This proposal therefore essentially introduces strong typedefs for arrays.

Some compilers have modes where the object representation of the array might actually be different from an object that has matching data members, for example "hardening" modes that add extra data to diagnose out-of-bounds array access. In these cases, the `layoutas` specifier will disable this functionality for the type in question.

Note that this functionality does not extend to a `constexpr` context.

6 Wording

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

Document history

— **R0**, 2019-10-06: Initial version.

— **R1**, 2020-01-13: Changed proposal to consider only conversions to an array type, instead of between arbitrary types; added layout-compatibility to enable conversion via unions.

Acknowledgements

Many thanks to Fabian Renn-Giles and Richard Smith for encouraging this proposal. Many thanks to Andrey Davydov for reviewing the paper and suggesting valuable improvements.

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.