Accessing object representations

Timur Doumler (papers@timur.audio)
Krystian Stasiowski (sdkrystian@gmail.com)

Document #:	P1839R5
Date:	2022-06-16
Project:	Programming Language C++
Audience:	Core Working Group

Abstract

This paper proposes a wording fix to the C++ standard to allow read access to the object representation (i.e. the underlying bytes) of an object. This is valid in C, and is widely used and assumed to be valid in C++ as well. However, in C++ this is is undefined behaviour under the current specification.

1 Motivation

Consider the following program, which takes an `int` and prints the underlying bytes of its value in hex format:

```
void print_hex(int n) {
  unsigned char* a = (unsigned char*)(&n);
  for (int i = 0; i < sizeof(int); ++i)
  printf("%02x ", a[i]);
}

int main() {
  print_hex(123456);
}
```

In C, this is a valid program. On a little-endian machine where `sizeof(int) == 4`, this will print `40 e2 01 00` . In C++, this is widely assumed to be valid as well, and this functionality is widely used in existing code bases (think of binary file formats, hex viewers, and many other low-level use cases).

However, surprisingly, in C++ this code is undefined behaviour under the current specification. In fact, it is impossible in C++ to directly access the object representation of an object (i.e. to read its underlying bytes), even for built-in types such as `int`. Instead, we would have to use `memcpy` to copy the bytes into a separate array of `unsigned char`, and access them from there. However, this workaround only works for trivially copyable types. It also directly violates one of the fundamental principles of C++: to leave no room for a lower-level language.

The goal of this paper is to provide the necessary wording fixes to make accessing object representations such as in the code above defined behaviour. Existing compilers already assume that this should be valid. The goal of the paper is therefore to *not* require any changes to existing compilers or existing code, but to legalise existing code that already works in practice and was always intended to be valid.

2 The problem

The cast to `unsigned char*`, which performs a `reinterpret_cast`, is fine, because `char`, `unsigned char`, and `std::byte` can alias any other type, so we do not violate the rules for type punning. However, with the current wording, this cast does *not* yield a pointer to the first element of `n`'s object representation (i.e. a pointer to a byte), and in fact it is currently impossible in C++ to obtain such a pointer. This is because this particular `reinterpret_cast` is exactly equivalent to `static_cast<unsigned char*>(static_cast<void*>(n))` as per [expr.reinterpret.cast] p7, and as such, [expr.static.cast] p13 dictates that the value of the pointer is unchanged and therefore it points to the original object (the `int`). When `a` is dereferenced, the behaviour is undefined as per [expr.pre] p4 because the value of the resulting expression would *not* be the value of the first byte, but the value of the whole `int` object (123456), which is not a value representable by `unsigned char`.

Further, even if we ignore this issue, `a` does not point to an array of `unsigned char`, because such an array has never been created, and therefore pointer arithmetics on `a` is undefined behaviour. An object representation as defined by [basic.types] p4 is merely a *sequence* of `unsigned char` objects, not an array, and is therefore unsuitable for pointer arithmetics. No array is ever created explicitly, and no operation is being called in the above code that would implicitly create an array, since casts are not operations that implicitly create objects as per [intro.object] p10.

3 History and context

The intent of CWG has always been that the above code should work, as exemplified by [CWG1314], in which it is stated that access to the object representation is intended to be well-defined. Further, it seems that the above code actually *did* work until C++17, when [P0137R1] was accepted. This proposal fixed an unrelated core issue and included a change to how pointers work, notably that they point to objects, rather than just representing an address. It seems that the proposal neglected to add any provisions to allow access to the object representation of an object, and thus inadvertently broke this functionality. Therefore, this paper is a defect report, not a proposal of a new feature.

Notably, there are even standard library facilities that directly use this functionality and cannot be implemented in standard C++ without fixing it. One such facility is `std::as_bytes` (introduced in C++20), which obtains a `std::span<const std::bytes>` view to the object representation of the elements of another span. Now, we do have a few "magic" functions in the C++ standard library that cannot be implemented in standard C++, but reading the underlying bytes of an object is such basic functionality that it should not fall into this category.

4 Non-goals

This paper does not propose to make in-place modification of the object representation valid, i.e. *writing* into the underlying bytes, only *reading* them. The following code will still be undefined behaviour:

```
void increment_first_byte(int* n) {
  auto* a = reinterpret_cast<char*>(n);
  ++(*a);
}
```

It may be desirable to allow such code as well. However, unlike reading the object representation, modifying it was never allowed in C++, so fixing it would be a new feature, not be a defect report. Note that the current rules in [basic.types.general] allow only to `memcpy` the value of an object into an array of `char`, `unsigned char`, or `std::byte` and then `memcpy` it back unmodified, or to `memcpy` the value of one object into another, but not to `memcpy` the value of an object into an array, then modify the elements of the array, and then `memcpy` back the modified value. Writing back a modified value would be a new invention, and modifying the value *in-place* even more so. It can also only ever work for trivially copyable types. Therefore, CWG gave the guidance to reduce the scope of this paper to reading only, and propose the modifying case in a separate paper (not yet published).

This paper also does not propose to subvert existing type punning rules in any way. The proposed changes will not allow type punning between two different types where it was not previously allowed, such as between `int` and `float` (this should be done using `std::bit_cast`). It only allows type punning to `char`, `unsigned char`, and `std::byte`, which are already allowed to alias any other type.

We also do not propose to make accessing the object representation work for *all* types in C++, only for types that are currently guaranteed to occupy contiguous bytes of storage, that is, for trivially copyable or standard-layout types as per [intro.object] p5. On the one hand, this is unnecessarily restrictive: in practice, any sane implementation will have complete objects, array elements, and member subobjects occupying contiguous memory, as the only reason an object would need to be non-contiguous would be if it was a virtual base subobject. On the other hand, making more objects contiguous (and therefore, their object representations accessible) is not in scope for this paper, and is instead tackled in a separate proposal [P1945R0].

5 Proposed solution

For an object `a` of type `T`, we propose to change the definition of *object representation* to be considered an array of `unsigned char`, and not merely a *sequence* of `unsigned char` objects, if `T` is a type that occupies contiguous bytes of storage. We propose that this object representation should be an object in its own right, occupying the same storage as `a` and having the same lifetime. This will make pointer arithmetic work with a pointer to an element of the object representation.

To avoid an infinite recursion of nested object representations, we further specify that an array of `unsigned char` acts as its own object representation. We also need to prevent implicit object creation [P0593R6] within object representations.

We further propose that obtaining a pointer to the object representation should be possible through the use of a cast to `char`, `unsigned char`, or `std::byte`, and allow this pointer to be cast back to a pointer to its respective object. For this, we need to make the appropriate changes to the specification of `static_cast` and to make `a` pointer-interconvertible with its own object representation as well as with the first element thereof. We need to do this in a way that preserves `reinterpret_cast`'s equivalence with `static_cast` with respect to converting object pointers. Simultaneously, if multiple pointer-interconvertible objects exist, we need to specify which one is chosen.

Additionally, we need to make reading an object representation through a pointer to `char` or `std::byte` well-defined, even though it points to an element of the object representation which is of type `unsigned char`. In these cases, we must allow for the type of the expression to differ from that of the object pointed to.

We also need to say something about the values of the elements of an object representation. We propose that for objects of type `char`, `unsigned char`, and `std::byte`, the value of each element is the value of the object it represents. For all other types, the values of the elements of the object representation are unspecified. It seems extremely difficult to specify for the general case what the value of each element would be, but it is also unnecessary, since our goal is only to make reading the elements well-defined, not to specify a particular result (which won't be the same across platforms).

Finally, multiple objects may occupy the same storage, in which case the objects' respective object representations will overlap. We must therefore adjust the specification of `std::launder` to define which object it will return a pointer to.

6 Polls

EWGI

Should accessing the object representation be defined behaviour?

Unanimous consent

Forward P1839R1 as presented to EWG, recommending that this be a core issue?

Unanimous consent

EWG

It should be possible to access the entire object representation through a pointer to a char-like type as a DR.

SF	F	N	A	SA	
10	8	2	0	0	Consensus

7 Proposed wording

The reported issue is intended as a defect report with the proposed resolution as follows. The effect of the wording changes should be applied in implementations of all previous versions of C++ where they apply. The proposed changes are relative to the C++ working draft [N4910].

Modify [basic.types] paragraph 4 as follows:

> The *object representation* of an object `a` of type *cv* `T` is ~~the~~a sequence of *N* *cv* `unsigned char` objects ~~taken up by the object of type `T`~~that occupy the same storage as `a`, where *N* equals `sizeof(T)`. The sequence is considered to be an array of *N* *cv* `unsigned char` if the object of type `T` occupies contiguous bytes of storage ([intro.object]).
>
> For an object of type `unsigned char` or an array of `unsigned char` (ignoring *cv*-qualification), the object representation is the object itself. For an object of type `char`, `std::byte`, or an array of such types (ignoring *cv*-qualification), the value of the elements of the object representation is the value of the object itself. For all other types, the value of the elements of the object representation is unspecified.
>
> The sequence of bytes in the object representation of an object nested within an object `a` is a sub-sequence of the sequence of bytes in the object representation of `a`.

Modify [intro.object] paragraph 9 as follows:

> Two objects with overlapping lifetimes that are not bit-fields may have the same address if one is nested within the other, or if at least one is a subobject of zero size and they are of different types, or if at least one is an element of an object representation; otherwise, they have distinct addresses and occupy disjoint bytes of storage.

Insert a new paragraph below [basic.life] paragraph 2 as follows:

> The lifetime of a reference begins when its initialization is complete. The lifetime of a reference ends as if it were a scalar object requiring storage.
>
> [*Note 1:* [class.base.init] describes the lifetime of base and member subobjects. — *end note*]
>
> The lifetime of the elements of the object representation of an object begins when the lifetime of the object begins. For class types, the lifetime of the elements of the object representation ends when the destruction of the object is completed, otherwise, the lifetime ends when the object is destroyed.

Modify [intro.object] paragraph 13 as follows:

> An operation that begins the lifetime of an array of `char`, `unsigned char`, or `std::byte` other than the lifetime of an object representation implicitly creates objects within the region of storage occupied by the array.

Modify [expr.static.cast] paragraph 13 as follows:

> Otherwise, if the original pointer value points to an object a, and there is an object b of type T (ignoring *cv*-qualification) that is pointer-interconvertible with a, the result is a pointer to b if doing so would give the program defined behavior. Otherwise, the pointer value is unchanged by the conversion.

Modify [basic.compound] paragraph 4.3 as follows:

> Two objects `a` and `b` are *pointer-interconvertible* if:
>
> — they are the same object, or
>
> — one is a union object and the other is a non-static data member of that object ([class.union]), or
>
> — one is a standard-layout class object and the other is the first non-static data member of that object or any base class subobject of that object ([class.mem]), or
>
> — one is the object representation of the other, or the first element thereof, or
>
> — there exists an object c such that a and c are pointer-interconvertible, and c and b are pointer-interconvertible.
>
> If two objects are pointer-interconvertible, then they have the same address, and it is possible to obtain a pointer to one from a pointer to the other via a `reinterpret_cast` ([expr.reinterpret.cast]).
>
> [*Note:* An array object and its first element are not pointer-interconvertible, even though they have the same address, except if the array is an object representation. — *end note*]

Modify [expr.add] paragraph 6 as follows:

> For addition or subtraction, if the expressions `P` or `Q` have type “pointer to *cv* `T`”~~, where T and the array element type are not similar, the behavior is undefined.~~ and point to an object a, one of the following shall hold:
>
> — `T` is similar to the type of a, or
>
> — `T` is similar to `unsigned char`, `char` or `std::byte` and a is an element of an object representation.
>
> Otherwise, the behavior is undefined.

Modify [ptr.launder] paragraph 3 as follows:

> *Returns:* A value of type `T*` that points to the object X that would give the program defined behavior. If no such object exists, the behavior is undefined.

8 Known issues

There are a number of known issues with the proposed wording that need to be resolved before this paper can make any further progress:

- The current proposed wording says that the first element of an object representation is pointer-interconvertible with the object that is represented by the representation. Combined with the existing rules, this means that first elements of some object representations are pointer-interconvertible with first elements of other object representations. In other words, a `reinterpret_cast` of a value to its own type is no longer equivalent to its operand. Example:

  ```
  struct A {
    unsigned char x[5];
    unsigned char y[5];
    // Note: arrays of unsigned char are their own object representation
  } a;

  int main(void) {
    unsigned char* p = &a.x[0];
    ++reinterpret_cast<unsigned char*>(p)[7]; // OK
    --p[7]; // not OK
  }
  ```

  This changes the existing behaviour of `reinterpret_cast` and potentially breaks array bounds checking.

- The current proposed wording for `std::launder` (returning a pointer to the object that would give the program defined behaviour) doesn't quite work because there is possibly more than one such object, such as elements of different object representations. Example:

  ```
  struct A {
    unsigned char x;
    unsigned char y;
  } a;

  auto f(bool b) {
    unsigned char *pastRepX = 1 + &a.x;
    unsigned char *pastRepA = sizeof(A) + reinterpret_cast<unsigned char*>(&a);
    unsigned char *ambiguous = reinterpret_cast<unsigned char*>(&a);
    return (b ? pastRepX : pastRepA) // ambiguous;
  }
  ```

- The current proposed wording allows `reinterpret_cast` or `std::launder` to `unsigned char*` to produce a pointer to the first element of an object representation. The same cannot be said for `std::byte*`.

- The current proposed wording does not allow `reinterpret_cast` of pointers past-the-end of an object to produce pointers past-the-end of the object representation. For cases involving pointer arithmetic, such as looping over the entire object representation (an important use case), such conversions are necessary.

- In [expr.static.cast], as well as in [ptr.launder], there might be multiple such objects that would give the program defined behaviour. We don't know how to specify which one is returned.

Document history

— **R0**, 2019-07-30: Initial version.

— **R1**, 2019-09-28: Allowed pointer arithmetic on expressions of type `unsigned char*`, `char*` and `std::byte*` when pointing to objects of different type. Removed exclusion of the object representation of objects of zero size from appearing in the object representation of their containing object. Added multi-dimensional arrays of contiguous-layout types to the definition of contiguous-layout types. Slight change to the behavior of `std::launder` for when there are multiple viable objects.

— **R2**, 2019-11-20: Removed contiguous-layout types from wording, this should be tackled by [P1945R0].

— **R3**, 2022-02-15: Moved wording for casts to the rules of pointer-interconvertibility. Changed the wording for `std::launder` to bind to the best candidate object.

— **R4**, 2022-03-16: Changed the wording to fix ambiguous usage of N in object representations specification.

— **R5**, 2022-06-16: Reduced scope of paper to only reading object representations, not writing. Completely rewrote rationale. Added wording to prevent implicit object creation within object representations. Added cross-reference to types with contiguous storage ([intro.object]) in the wording. Fixed inconsistency in the wording by defining that only `unsigned char` is its own object representation, not `char` or `std::byte`. Removed erroneous wording regarding memory locations. Added list of known issues.

Acknowledgements

Many thanks to Jens Maurer and Hubert Tong for their help with the wording. Thanks to Jason Cobb, John Iacino, Marcell Kiss, Killian Long, Theodoric Stier, and everyone who participated on the std-proposals mailing list for their countless reviews and suggestions for earlier revisions of this paper. Thanks to Professor Ben Woodard for his grammatical review of an earlier revision of this paper.

References

[CWG1314] Nikolay Ivchenkov. Core Issue 1314: Pointer arithmetic within standard-layout objects. `https://www.open-std.org/jtc1/sc22/wg21/docs/cwg_closed.html#1314`, 2011-05-06.

[N4910] Thomas Köppe. Working Draft, Standard for Programming Language C++. `http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2022/n4910.pdf`, 2022-03-17.

[P0137R1] Richard Smith. Core Issue 1776: Replacement of class objects containing reference members. `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0137r1.html`, 2019-10-28.

[P0593R6] Richard Smith. Implicit creation of objects for low-level object manipulation. `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p0593r6.html`, 2020-02-14.

[P1945R0] Krystian Stasiowski. Making More Objects Contiguous. `https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1945r0.pdf`, 2019-10-28.