Accessing Object Representations

Contents

1 Abstract ............................................. 2
2 Revisions ............................................. 2
  2.1 Changes since [P1839R0] .................................. 2
  2.2 Changes since [P1839R1] .................................. 2
    2.2.1 Polls ............................................ 2
3 Motivation ............................................. 2
4 Problem ................................................. 3
5 Changes ................................................. 3
  5.1 Examples ............................................. 3
6 Design Choices .......................................... 4
  6.1 Object contiguity ...................................... 4
  6.2 Preserving reinterpret_cast and static_cast equivalence ............ 4
  6.3 The std::lauder issue .................................. 5
  6.4 “Self-representing” objects ................................ 5
  6.5 Value and access of elements of object representations ............. 5
7 Wording ................................................... 5
  7.1 Memory and object model [intro.object], [intro.memory], [basic.life] . 5
    7.1.1 Memory locations ..................................... 5
    7.1.2 Object representations ................................ 5
    7.1.3 Overlapping objects .................................... 6
    7.1.4 Identical element lifetime ................................ 6
  7.2 Access to object representations via reinterpret_cast [expr.reinterpret.cast] ....... 6
    7.2.1 Obtaining a pointer to the object representation .................. 6
  7.3 Pointer arithmetic [expr.add] ................................ 7
    7.3.1 Differing element and expression type .......................... 7
  7.4 std::lauder [ptr.launder] .................................. 7
    7.4.1 Multiple overlapping objects ............................... 7
8 Acknowledgements .......................................... 7
9 References ................................................ 7
1 Abstract
Allow access to the object representation of an object.

2 Revisions

2.1 Changes since [P1839R0]
— Allow 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::laundry for when there are multiple viable objects.

2.2 Changes since [P1839R1]
— Removed contiguous-layout types from wording, this should be tackled by [P1945R0].

2.2.1 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
Approved.

3 Motivation

This proposal does not intend to introduce anything new, rather to standardize a common existing practice. Accessing the underlying bytes of an object has been a long-standing practice in C and C++ alike, but, in C++, doing so is typically undefined behavior. With current wording, it is impossible to obtain a pointer to an element of the object representation; an expression such as reinterpret_cast<char*>(ka) typically yields a pointer to the original object, and only the type of the expression is changed. This does not represent the intent of CWG, as exemplified by [CWG1314], in which it is stated that access to the object representation is intended to be well-defined.

This has only recently become undefined behavior as of C++17, when [P0137R1] was accepted. This proposal includes a change to how pointers work, notably that they point to objects, rather than just representing an address, and it seems that the proposal neglected to add any provisions to allow access to the object representation of an object.
4 Problem

This issue exists for two primary reasons: casting and pointer arithmetic. Given the following code:

```cpp
int a = 420;
char b = reinterpret_cast<char*>(&a);
```

The pointer does not bind to any `char` object or element of the object representation. This particular `reinterpret_cast` is exactly equivalent to `static_cast<char*>(static_cast<void*>(&a))` 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. When the lvalue-to-rvalue conversion is applied to the initializer expression of `b`, the behavior is undefined as per [expr.pre] p4 because the result of such a conversion would be the value of the `int` object (420), which is not a value representable by `char`.

Additionally, if such wording did exist, an object representation as defined by [basic.types] p4 is a sequence of `unsigned char` objects, not an array, and is unsuitable for pointer arithmetic given the current object model.

5 Changes

— Change object representations to be considered an array if the type of the object they represent is contiguous.

— Objects of type `unsigned char`, `char` and `std::byte` and arrays of such types suffice as being their own object representation to prevent an infinitely recurring property.

— The value of the elements of an object representation of a type other than `unsigned char`, `char` and `std::byte` is unspecified, otherwise the value of the element is the value of the object they represent.

— Allow a pointer to an object representation to be obtained through a `reinterpret_cast` to `unsigned char`, `char` and `std::byte`.

— Allow a pointer to an object representation to be cast back to a pointer to its respective object via `reinterpret_cast`.

— Specify that `std::launder` will prefer to return a pointer to an object that is not an element of an object representation.

— Allow pointer arithmetic to be performed on pointers to elements of an object representation if the type of the expression is `unsigned char*`, `char*` or `std::byte*`.

5.1 Examples

Here is an example demonstrating the difference:

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>using T = unsigned char*;</td>
<td>using T = unsigned char*;</td>
</tr>
<tr>
<td>int a = 0;</td>
<td>int a = 0;</td>
</tr>
<tr>
<td>T b = reinterpret_cast&lt;T&gt;(&amp;a);</td>
<td>T b = reinterpret_cast&lt;T&gt;(&amp;a);</td>
</tr>
<tr>
<td>// Pointer value unchanged, still</td>
<td>// Pointer now points to the first unsigned</td>
</tr>
<tr>
<td>// points to the int object</td>
<td>// char element of the object representation</td>
</tr>
<tr>
<td>T c = ++b;</td>
<td>T c = ++b;</td>
</tr>
<tr>
<td>// UB, expression type differs</td>
<td>// This is now a pointer to the second</td>
</tr>
<tr>
<td>// from element type</td>
<td>// element of the object representation</td>
</tr>
<tr>
<td></td>
<td>++(*c); // OK</td>
</tr>
</tbody>
</table>
Another example for arrays:

<table>
<thead>
<tr>
<th>Before</th>
<th>After</th>
</tr>
</thead>
<tbody>
<tr>
<td>using T = unsigned char*; int a[5]{}; T b = reinterpret_cast&lt;T&gt;(&amp;a); // Pointer value unchanged, still // points to the array object for (int i = 0; i &lt; sizeof(int) * 5; ++i) b[i] = 0; // UB, expression type differs // from element type</td>
<td>using T = unsigned char*; int a[5]{}; T b = reinterpret_cast&lt;T&gt;(&amp;a); // Pointer now points to the first // unsigned char element of the // object representation of the array for (int i = 0; i &lt; sizeof(int) * 5; ++i) b[i] = 0; // OK</td>
</tr>
</tbody>
</table>

6 Design Choices

6.1 Object contiguity

For pointer arithmetic to work with a pointer to an element of the object representation, it is necessary that the object representation be an array. However, since not all objects are guaranteed to occupy contiguous bytes of storage, the object representation may only be an array if the corresponding object occupies contiguous bytes of storage. It would be useful to expand the guarantee for which objects occupy contiguous storage, and therefore the subset of objects that can have their object representation accessed, however, this will be addressed in later paper [P1945R0].

6.2 Preserving reinterpret_cast and static_cast equivalence

The proposed wording does not preserve the current reinterpret_cast and static_cast equivalence for object pointer types when reinterpret_cast is used to cast a pointer to unsigned char*, char*, and std::byte*. In these cases, reinterpret_cast will no longer follow the pointer-interconvertibility rules used by static_cast, instead resulting in a pointer to the object representation.

For example:

```cpp
struct S
{
    unsigned char a;
};

void f()
{
    S b{0};
    unsigned char* c = reinterpret_cast<unsigned char*>(&b);
}
```

The current approach taken by this proposal is to only allow for reinterpret_cast to convert &b into a pointer that points to an element of the object representation of b. Preserving the equivalence leads to the question of whether the cast should follow the pointer-interconvertibility rules and result in a pointer to b.a, or result in a pointer to the first element of the object representation of b. However, breaking the equivalence will prevent std::memcpy from being implemented by the user with standard C++, as casting from a void* to unsigned char* will not result in a pointer pointing to an element of the object representation. It is unclear which of these it should be and is best to be decided on by CWG.
6.3 The std::launder issue

Multiple objects may occupy the same storage, in which case the objects’ respective object representations will overlap. This presents the issue of deciding to which object std::launder returns a pointer. This proposal remedies the issue by prioritizing objects that are not elements of an object representation, and if no such object is found, then a pointer to an unspecified element of the set of viable objects is returned.

6.4 “Self-representing” objects

Certain objects are suitable to act as their own object representations, such as objects of type unsigned char, char and std::byte and arrays of these types. This is to prevent infinite recursion of objects having object representations, as happens with the current word if read pedantically.

6.5 Value and access of elements of object representations

“Self-representing” elements of an object representation of non-array type are specified to have their own value; all other elements of an object representation have an unspecified value. The reasoning for this is quite obvious, as it would be extremely difficult to specify what the value of each element would be. Access of the elements is intended to be well-defined, and is under the proposed wording, however it is up to CWG whether it should be specified explicitly.

7 Wording

All wording is relative to N4835.

7.1 Memory and object model [intro.object], [intro.memory], [basic.life]

7.1.1 Memory locations

Modifying an element of an object representation concurrently with another memory location that it overlaps should be a data race. This sentence specifies that the element of the object representation is the same memory location.

Changes to [intro.memory] p3 sentence 1

3 A memory location is either an object of scalar type or a maximal sequence of adjacent bit-fields all having nonzero width, and any overlapping elements of an object representation.

7.1.2 Object representations

The type of the elements of the object representation should have the same cv-qualification as that of the object they represent to prevent accidental modification of the object indirectly. Additionally, if the object occupies contiguous bytes of storage, then we could consider the object representation to be an array, and thereby make pointer arithmetic well-defined. Certain objects are said to represent themselves so that the object representation does not have an object representation of its own. The value of the elements that do not represent themselves is left unspecified, as specifying it would be effectively impossible. Lastly, it is specified that an object representation appears in an enclosing object representation to make it useful for introspection.

Remove [basic.types] p4 sentence 1

4 The object representation of an object of type T is the sequence of N unsigned char objects taken up by the object of type T, where N equals sizeof(T).
The object representation of an object \( a \) of type \( \text{cv } T \) is a sequence of \( N \) \( \text{cv } \text{unsigned char} \) objects that occupy the same storage as \( a \), where \( N \) is equal to \( \text{sizeof}(T) \). The sequence is considered to be an array of \( N \) \( T \) if the object occupies contiguous bytes of storage. The object representation of an object of type \( \text{unsigned char} \), \( \text{char} \), \( \text{std::byte} \), or an array of such types (ignoring cv-qualification), is itself. Unless an object representation is of an object of type \( \text{unsigned char} \), \( \text{char} \) or \( \text{std::byte} \) (ignoring cv-qualification), the value of the elements of the object representation is unspecified. The object representation of an object nested within an object \( o \) is guaranteed to appear in the object representation of \( o \).

### 7.1.3 Overlapping objects

This ensures that an object representation and its elements may exist concurrently with the object they represent, as they occupy the same storage.

Changes to [intro.object] p9

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.

### 7.1.4 Identical element lifetime

Specifying the lifetime of an object representation explicitly ensures that its lifetime will begin and end with that of its corresponding object meaning it need not be preserved after the object it represents is destroyed. The lifetime does not begin during construction to match the wording of [class.cdtor] p2.

7.2 Access to object representations via reinterpret_cast [expr.reinterpret.cast]

#### 7.2.1 Obtaining a pointer to the object representation

As the current wording for \texttt{reinterpret\_cast} will result in an unchanged pointer value when attempting to access an object representation, adding a special case to \texttt{reinterpret\_cast} is necessary to be able to obtain a pointer to the object representation, and vice versa. However, this should only happen if the object representation of the object we are casting has an object representation on which we can perform pointer arithmetic.

Replace [expr.reinterpret.cast] p7

An object pointer can be explicitly converted to an object pointer of a different type. When a prvalue \( v \) of object pointer type is converted to the object pointer type “pointer to cv T”, the result is \( \text{static\_cast<cv T*>>(\text{static\_cast<cv void*>(v))} \).

A prvalue \( v \) of object pointer type “pointer to cv1 T1” pointing to an object \( a \) can be explicitly converted to an object pointer of a different type “pointer to cv2 T2”, where cv2 is the same cv-qualification as, or greater cv-qualification than cv1, the result of which is defined as follows:

\begin{enumerate}
\item If \( a \) occupies contiguous bytes of storage and \( T2 \) is \( \text{unsigned char} \), \( \text{char} \) or \( \text{std::byte} \), the result is a pointer to the first element of the object representation of \( a \).
\item Otherwise, if \( a \) points to the object representation of an object \( b \) of type \( T2 \) (ignoring cv-qualification), or the first element thereof, the result is a pointer to \( b \).
\item Otherwise, the result is \( \text{static\_cast<cv2 T2*>>(\text{static\_cast<cv2 void*>(v))} \).
\end{enumerate}
7.3 Pointer arithmetic [expr.add]

7.3.1 Differing element and expression type

In order to make accessing an object representation using a type other than unsigned char well-defined, it must be allowed for the type of the expression to differ from that of the object pointed to in cases where the type of the pointer is char* or std::byte*, as the pointer points to an object of type unsigned char.

Replace [expr.add] p6

6 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.

6 For addition and subtraction where P or Q have type “pointer to cv T” and point to an object o, one of the following shall hold:

(6.1) T is similar to the type of the o, or
(6.2) T is similar to unsigned char, char or std::byte and o is an element of an object representation.

Otherwise, the behavior is undefined.

7.4 std::launder [ptr.launder]

7.4.1 Multiple overlapping objects

Since multiple elements of an object representation may exist in the same storage, it is unclear which one std::launder would return if such a situation were to arise. Specifying that it is implementation-defined (or possibly unspecified) gives this defined, but unclear behavior.

Changes to [ptr.launder] p3

3 Returns: A value of type T* that points to X. If multiple such objects exist, the result is the object in the set of possible objects that is not an element of an object representation. Otherwise, it is implementation-defined to which object in the set the result points.

8 Acknowledgements

Thank you to Jason Cobb, John Iacino, Marcell Kiss, Killian Long, Theodoric Stier, and everyone who participated on the std-proposals mailing list for the countless reviews and suggestions. Additionally, I would like to thank Professor Ben Woodard for his grammatical review. Lastly, I’d like to thank Timur Doumler for presenting this paper at Belfast, among other things.

9 References

https://wg21.link/cwg1314

https://wg21.link/p0137r1

https://wg21.link/p1945r0