Accessing Object Representations

Document #:	P1839R1
Date:	2019-09-28
Project:	Programming Language C++ Core Working Group

Reply-to:	Krystian Stasiowski <sdkrystian@gmail.com>

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::launder` for when there are multiple viable objects.

3 Motivation

This proposal does not intend to introduce anything new, but 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, with an expression such as `reinterpret_cast<char*>(&a)` typically yielding a pointer to the original object, with only the type of the expression being 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 due to two primary reasons: casting and pointer arithmetic. Given the following code:

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

There exist no provisions in the current wording for the pointer to 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 be unchanged, leaving it pointing to the original object. When the lvalue-to-rvalue conversion is applied to the initializer expression when initializing `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

- Introduce contiguous-layout types, a classification of types that encompass scalar types, and class types without virtual functions or virtual bases and no subobjects of non-contiguous-layout class type or arrays of such types.
- Specify that contiguous-layout types are guaranteed to be contiguous.
- Change object representations to be considered an array if the type of the object they represent is a contiguous-layout type.
  - 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:

Before

```
using T = unsigned char*;
int a = 0;
T b = reinterpret_cast<T>(&a);
// Pointer value unchanged, still
// points to the int object
T c = ++b;
// UB, expression type differs
// from element type
```

After

```
using T = unsigned char*;
int a = 0;
T b = reinterpret_cast<T>(&a);
// Pointer now points to the first unsigned
// char element of the object representation
T c = ++b;
// This is now a pointer to the second
// element of the object representation
++(*c); // OK
```

Another example for arrays:

Before	After

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

6 Design Choices

6.1 Contiguous-layout types

A major limitation for this proposal is that only objects of trivially-copyable or standard-layout type are guaranteed to occupy contiguous storage. This presents the challenge of not being able to define an object representation as an array for the purpose of pointer arithmetic for many types. This limitation is demonstrated by this example:

```
struct S
{
  S(const S& value) : a(value.a) { }
  int a;

private:
  int b;
};
```

This type is not guaranteed to occupy contiguous storage, although in practice it always will. As such, this proposal intends to define a new category for types, contiguous-layout types, which are effectively trivially-copyable types without the requirement for trivial special member functions, as in a reasonable implementation, special member functions would not impact the layout of the class.

With the definition of objects guaranteed to be contiguous less restrictive, the object representation of an object may be defined to be a sequence, that is treated as an array if the type of the object the object representation is associated with is a contiguous-layout type, allowing for pointer arithmetic to be performed on pointers that point to elements of an object representation. This grants a fair bit more latitude for when one wants to access the object representation of an object.

6.2 Preserving `reinterpret_cast` and `static_cast` equivalence

There also exists the question of preserving the current `reinterpret_cast` and `static_cast` equivalence for object pointer types. The proposed wording does not do so when `reinterpret_cast` is used to cast a pointer to `unsigned char*`, `char*`, and `std::byte*` as doing so would present a conflict with resulting in a pointer to the object representation or if it should follow the pointer-interconvertibility rules used by `static_cast`.

For example:

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

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

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 elelment of the object representation of `b`. However, not preserving 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

Since multiple objects may occupy the same storage, there exists an issue that elements of these object's respective object representations will overlap, and will present the issue of having multiple objects that `std::launder` can return a pointer to. This proposal remedies the issue by prioritizing objects that are not elements of an object representation, and if such an object is not 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 representation, such as object 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 will are specified to have their own value, as expected, and 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

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

Changes to [intro.memory] p3 sentence 1

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

Insert a new paragraph below [intro.object] p1

2 The *object representation* of an object `a` of type `cv T` is a sequence of `N cv unsigned char` objects that occupy the same storage as `a`, where `N` is equal to `sizeof(T)`. The sequence is considered to be an array of

`N T` if `T` is a contiguous-layout type. The object representation of an object of type `unsigned char`, `char`, `std::byte`, or an array of such types (ignoring cv-qualification), is itself. Unless an object representation is of an object of type `unsigned char`, `char` or `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`.

Changes to [intro.object] p8

8 [...] Unless it is a bit-field, an object with nonzero size shall occupy one or more bytes of storage, including every byte that is occupied in full or in part by any of its subobjects. An object of ~~trivially copyable or standard-layout~~contiguous-layout type shall occupy contiguous bytes of storage.

Changes to [intro.object] p9

9 [...] 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] p2

3 The lifetime of the elements of the object representation of an object begins when the lifetime of the object begins and ends when the lifetime of the object ends.

7.2 Contiguous-layout types [basic.types], [class.prop]

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)`.

Append a sentence to [basic.types] p9

9 [...] Scalar types, standard-layout class types, arrays of such types and cv-qualified versions of these types are collectively called *standard-layout types*. Scalar types, contiguous-layout class types, (possibly multi-dimensional) arrays of such types and cv-qualified versions of these types are collectively called *contiguous-layout types.*

Insert a new paragraph below [class.prop] p7

8 A class is a *contiguous-layout class* if it has no virtual functions, no virtual base classes, no non-static data members of non-contiguous-layout class type (or array of such types), and no base classes of non-contiguous-layout class type.

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

Replace [expr.reinterpret.cast] p7

7 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 `static_cast<cv T*>(static_cast<cv void*>(v))`.

7 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:

(7.1) — If `T1` is a contiguous-layout type and `T2` is `unsigned char`, `char` or `std::byte`, the result is a pointer to the first element of the object representation of `a`.

(7.2) — 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`.

(7.3) — Otherwise, the result is `static_cast<cv2 T2*>(static_cast<cv2 void*>(v))`.

7.4 Pointer arithmetic [expr.add]

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 addtion and subtraction where `P` or `Q` have type "pointer to `cv T`" and point to an object `o`, one of the following must hold true:

(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.

7.5 `std::launder` [ptr.launder]

Changes to [ptr.launder] p3

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 which object in the set the result points to.

8 Acknowledgements

Thank you to Jason Cobb, John Iacino, Marcell Kiss, and Killian Long, and everyone who participated on the std-proposals mailing list for the countless reviews and suggestions. Addtionally, I would like to thank Professor Ben Woodard for his grammatical review.

9 References

[CWG1314] Nikolay Ivchenkov. 2011. Pointer arithmetic within standard-layout objects.
https://wg21.link/cwg1314

[P0137R1] Richard Smith. 2016. Core Issue 1776: Replacement of class objects containing reference members.
https://wg21.link/p0137r1