Accessing Object Representations

Document #:	P1839R0
Date:	2019-07-30
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 Motivation

This proposal does not intend to introduce anything new, but rather 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, the best you can get is access to the first byte through a `reinterpret_cast` to `char*`, `unsigned char*` or `std::byte*` due to aliasing rules, however, almost any pointer arithmetic performed afterward will result in undefined behavior, as said pointer will typically still point to the original object.

3 Changes

The following changes proposed are the most effective way to fix this issue using existing wording and without introducing bloat:

- Introduce contiguous-layout types, a classification of types that encompasses scalar types, and class types without virtual functions or virtual bases and no subobjects of non-contiguous-layout type.
- 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 it's respective object via `reinterpret_cast`.

An example of what will change:

Before	After

`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`
	`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`

An 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`

4 Design Choices

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

5 Wording

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 if `T` is of 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 of nonzero size 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.

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.

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, arrays of such types and cv-qualified versions of these types are collectively called *contiguous-layout types.*

Remove [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))`.

Insert a new paragraph below [expr.reinterpret.cast] p6

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))`.

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, and no base classes of non-contiguous-layout class type.

6 Acknowledgements

Thank you to Jason Cobb, John Iacino, Marcell Kiss, and Killian Long for the countless reviews and suggestions.