Making More Objects Contiguous

Document #:	DXXXXR0
Date:	2019-10-21
Project:	Programming Language C++ Core Working Group

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

1 Abstract

Expand the guarantee for which objects are contiguous, and fix `sizeof`.

2 Motivation

There exists only a relatively small subset of types that are currently guaranteed to be contiguous, being trivially-copyable and standard-layout types. Allowing such a large subset of objects to occupy non-contiguous needlessly complicates the standard and introduces subtle bugs in the wording. 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. In addition to this, the specification of contiguous-layout types as defined in [P1839R1] would further reduce the number of objects that are potentially non-contiguous.

2.1 Logical size and `sizeof` incongruence

Consequently, `sizeof` can potentially display behavior that is unintended, as it is specified to return the number of bytes occupied by a non-potentially-overlapping. With the current wording, the value returned by `sizeof` is allowed to by smaller than the difference between the address of the last byte and the first byte, which introduces the problem that an array of `sizeof(T) unsigned char` may not necessarily be large enough to contain an object of type `T`. Consider the following:

```
struct S
{
  S(float f) : b(f) { }
  S(const S&);
  int a;
  float& b;
};

void f()
{
    alignas(alignof(S)) unsigned char arr[sizeof(S)];
    new (&arr) S(42.0f);
}
```

In the above code snippet, the standard does not guarantee that the lifetime of the `S` object will begin, since the size of the storage may not be of suitable size due to the aforementioned issue with `sizeof`, which is in direct violation of [basic.life] p1. Furthermore, `std::memcpy` is not guaranteed to work, as it is defined to copy `N` contiguous bytes of storage, which may be smaller than actual size of the object.

3 Problem

3.1 Object Contiguity

[intro.object] p8 guarantees that all objects of trivially-copyable or standard-layout type will occupy contiguous storage, which is a fairly limited scope of types. This weak requirement on which objects are guaranteed to be contiguous also effects how `sizeof` determines the size of an object. [expr.sizeof] p2 states:

> When [`sizeof` is] applied to a class, the result is the number of bytes in an object of that class including any padding required for placing objects of that type in an array.

This effectively says that `sizeof` applied to a class type will yield the number of bytes occupied by an non-potentially-overlapping object of that class type, plus any additional padding bytes that would be required in order to meet the requirement for array elements specified in [dcl.array] p6; these requirements are that the elements are allocated contiguously. Consider the following:

```
struct A
{
  A(const A&);
  char a;
private:
  char b;
};
```

`A` is neither trivially-copyable, nor a standard-layout type, which means that it can potentially occupy non-contiguous bytes of storage. According to the standard, a fully standard conforming layout of complete object of type `A` would be:

```
+-----------------------+
|       |       |       |
|   a   |       |   b   |
|       |       |       |
+-----------------------+
```

Where the middle byte is not part of the object. The alignment of such an object is implementation defined, but for the purposes of this paper, this paper will assume the alignment of `A` is `1`. As previously mentioned, the only padding bytes that are counted by `sizeof` are those that would be required to meet the requirements of [dcl.array] p6. In this case, `0` padding bytes would be required to allocate array elements of type `A` contiguously, as we previously established that the alignment of `A` is `1`. Since `sizeof` only counts the number of bytes occupied by an object of the specified type, the final number that `sizeof` would come up with is `2`.

3.2 Implications

Given the possible memory layout of `A` that was previously defined, the possibility that `sizeof(A)` could yield `2` with this possible is concerning. For instance, an array of `sizeof(A) unsigned char` would not be large enough to contain such an object. Likewise, an attempt to use `std::memcpy` to copy `sizeof(A)` bytes of the objects object representation would result in access of bytes that may not belong to any object, and it may not copy the entirety of the object representation.

4 Changes

5 Wording

6 References

[P1839R1] Krystian Stasiowski. 2019. Accessing Object Representations.
https://wg21.link/p1839r1