Making More Objects Contiguous

Document #:	D1945R0
Date:	2019-10-28
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 potentially non-contiguous objects. A small amendment is made in this proposal, allowing class types with virtual functions to be contiguous-layout types, as this should not affect the contiguity of the object in practice.

2.1 Logical size and `sizeof` incongruence

As a consequence of the large number of objects that can be non-contiguous, `sizeof` can potentially display unintended behavior, as it is specified to return the number of bytes occupied by a non-potentially-overlapping object of the specified type. With the current wording, the value returned by `sizeof` is allowed to be smaller than the difference between the address of the last byte and the first byte of an object of the type of the operand, 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 the actual size of the object.

2.2 Potential ABI breakage

The only context in which a non-contiguous object is absolutely necessary is for a virtual base subobject; all other objects can occupy contiguous storage. In practice, this is the case for the Itanium C++ ABI. The possibility of ABI breakage is only theoretical as it restricts what objects may occupy non-contiguous storage, but should not be an issue for most, if not all implementations.

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 affects 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 to meet the requirement for array elements specified in [dcl.array] p6; these requirements being 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. A permitted layout of a complete object of type `A` would be:

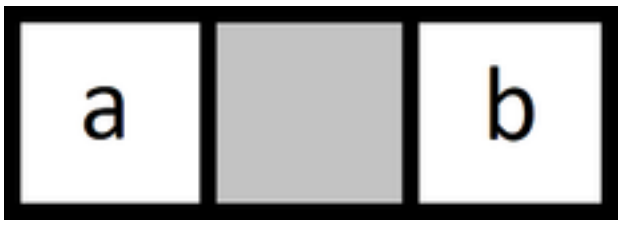

Figure 1: Potential object layout

Where the middle byte is not part of the object, making it occupy non-contiguous storage. The alignment of such an object is implementation defined, but for the purposes of this paper, it is assumed 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` is concerning. For instance, an array of `sizeof(A) unsigned char` would not be large enough to contain a complete object of type `A`. Likewise, an attempt to use `std::memcpy` to copy `sizeof(A)` bytes of the objects

object representation would result in the access of bytes that may not belong to any object, and it may not copy the entirety of the object representation.

4 Changes

The changes required to fix the issues presented in this paper are minimal.

- Introduce *contiguous-layout types* as specified in [P1839R1] without the requirement of no virtual functions.
  - Includes scalar types, and class types without virtual bases and no subobjects of non-contiguous-layout class type or arrays of such types.
- Guarantee that complete objects, array elements, member subobjects, and objects of contiguous-layout type occupy contiguous storage.

5 Wording

5.1 Object model

Changes to [intro.objecct] p8 sentence 5

8 An object ~~of trivially copyable or standard-layout type~~ shall occupy contiguous bytes of storage if it is a complete object, member subobject, array element, or of contiguous-layout type.

5.2 Contiguous-layout types

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 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.

6 References

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