When Empty Base Optimization Goes "Wrong"

This wasn't actually working as designed.

Matt Bisson — cyberbisson.com

February 2023

Occasionally, we use C++ inheritance to "tag" a class for some later purpose, or perhaps to introduce a functional change without copying code (e.g., `boost::noncopyable`). When the inherited class has no data, we (quite reasonably) expect to incur no run-time size overhead, due to Empty Base Optimization (EBO), but there is one surprising case where we must take care, or unintentionally waste memory. In this article, let's explore the issue, and how to guard against it.

1 The Problem

When you write a class with nothing in it, it is very surprising to do some compile-time check, and find that objects are paying a penalty for using that empty class. The C++ Standard (C++20; §6.7.2.8) defines what "empty" really means, but it's generally what you'd expect: if a class has no fields and no virtual functions (ignoring a few other qualifications), you can pretty much expect it to be empty. This means that when one derives from the class, the base should consume zero bytes of space — EBO — but there's a wrinkle (emphasis, mine):

> Two objects with overlapping lifetimes [...] 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** otherwise, they have distinct addresses and occupy disjoint bytes of storage.

This verbiage is dense with meaning, but essentially, if one has a zero-length object at the start of a structure, and a zero-length object as the base class, *and* they are the same type, then the compiler must give space to the base class so it has a distinct memory address. Consider an example:

```
struct empty_class { };

struct smaller_class : empty_class {
    long long int m_x[2];
};

struct bigger_class : empty_class {
    smaller_class m_sc;
};

struct biggest_class : empty_class {
    bigger_class m_bc;
    smaller_class m_sc;
};
```

First, what are we *trying* to have the sizes of these objects be? The obvious expectation is:

- `empty_class`: zero objects, zero size (caveat: `sizeof` must indicate a size of 1 byte).
- `smaller_class`: two 64-bit integers should be 16 bytes total.
- `bigger_class`: since this just wraps `smaller_class`, it must be 16 bytes as well, right?
- `bigger_class`: combining the first two classes, logically means this one is supposed to be 32 bytes.

When we dump the AST (in this case from Clang[1]), we can see that some of the assumptions run into that paragraph from the Standard:

```
*** Dumping AST Record Layout
         0 | struct empty_class (empty)
           | [sizeof=1, dsize=1, align=1,
           |  nvsize=1, nvalign=1]

*** Dumping AST Record Layout
         0 | struct smaller_class
         0 |   struct empty_class (base) (empty)
         0 |   long long[2] m_x
           | [sizeof=16, dsize=16, align=8,
           |  nvsize=16, nvalign=8]

*** Dumping AST Record Layout
         0 | struct bigger_class
         0 |   struct empty_class (base) (empty)
         8 |   struct smaller_class m_sc
         8 |     struct empty_class (base) (empty)
         8 |     long long[2] m_x
           | [sizeof=24, dsize=24, align=8,
           |  nvsize=24, nvalign=8]

*** Dumping AST Record Layout
         0 | struct biggest_class
         0 |   struct empty_class (base) (empty)
         8 |   struct bigger_class m_bc
         8 |     struct empty_class (base) (empty)
        16 |     struct smaller_class m_sc
        16 |       struct empty_class (base) (empty)
        16 |       long long[2] m_x
        32 |   struct smaller_class m_sc
        32 |     struct empty_class (base) (empty)
        32 |     long long[2] m_x
           | [sizeof=48, dsize=48, align=8,
           |  nvsize=48, nvalign=8]
```

The classes, `empty_class` and `smaller_class` are 1 and 16 bytes (respectively), as expected. Notice, however, that in the other two classes, we see `empty_class` as the base class, then the left column (the memory offset from the start of the structure) goes up to 8. This is because the first field inherits again from `empty_class`. It gets even worse for `biggest_class`, which retains the padding from `bigger_class`, and adds its own before it. The size of `bigger_class` is 24 instead of 16, and `biggest_class` is 48 instead of 32!

[1] This is pretty easy in Clang: just invoke "`clang++ -cc1 -emit-llvm -fdump-record-layouts from.cpp > dest.log`". This will work for C or C++. With GCC, you can get similar output with the `-fdump-lang-all` flag, but it only applies to C++. For Clang, if you have code of any complexity, you probably need to pre-process it first, then analyze the pre-processed file. The "CC1" phase of compilation does not generally even know where to find system headers by default.

The point is this: objects in C++ are required to have their own distinct identity. When you have two `empty_class` instances with two different names, the reasons are obvious: even though they have no data, one still needs to be able to differentiate between object #1 and object #2 with the equality operator. Alas, for something like `bigger_class`, the base class doesn't have a distinct name, but it still has an *identity*. Consider this example:

```
bigger_class o1;
void f(empty_class *o);

void g() {
    empty_class *const p1    = &o1;
    empty_class *const p1_sc = &o1.m_sc;
    assert(p1 != p1_sc); // Fundamentally, this...

    f(&o1);
    f(&o1.m_sc);
}
```

If the compiler does not give a different address for the base class, `empty_class` and for the base `empty_class` contained within `m_sc` (a `smaller_class` instance), then things start to break down. The fundamental identity of `o1` and one of its fields would be identical! Stranger still, any function like `f()` would not know which object it is using — probably not a big deal for an empty class, but we don't know if, for example, `f()` memoizes some result based on the identity of the objects it sees.

2 Just Make It Get Smaller, Please

Just to take the problem away, we could consider moving the actual data away from the empty base class. Here, we set up implementation classes that we can use to define the other classes, noted by the "`_impl`" suffix. They do not derive from `empty_class`, and therefore everything is exactly the size it's supposed to be:

```
struct smaller_class_impl {
    long long int m_x[2];
};
struct smaller_class : empty_class, smaller_class_impl { };

struct bigger_class_impl {
    smaller_class_impl m_sc;
};
struct bigger_class : empty_class, bigger_class_impl { };

struct biggest_class_impl {
    bigger_class_impl m_bc;
    smaller_class_impl m_sc;
};
struct biggest_class : empty_class, biggest_class_impl { };
```

To prove it, here's what Clang says:

```
*** Dumping AST Record Layout
         0 | struct biggest_class
         0 |   struct empty_class (base) (empty)
         0 |   struct biggest_class_impl (base)
         0 |     struct bigger_class_impl m_bc
         0 |       struct smaller_class_impl m_sc
         0 |         long long[2] m_x
        16 |     struct smaller_class_impl m_sc
        16 |       long long[2] m_x
           | [sizeof=32, dsize=32, align=8,
           |  nvsize=32, nvalign=8]
```

Unfortunately, this actually makes life pretty complicated for an interface of any complexity. Consider if the interface to `bigger_class` had a function that returned an instance of `bigger_class` — but now everything has to live in `bigger_class_impl` — should it return the tagged class, or the implementation class? Things have to be forward-declared, perhaps, but then it has to be separate if it's `inline`. Or should it be in `bigger_class`, but then it's separate from the data in `_impl`. Yuck! This is certainly not a drop-in fix for such a code-base.

So while this approach does confirm out reasoning — that EBO is indeed affected by the repeated base class — we can do better.

3 A Much Better Answer

The solution I propose here is painfully simple. *Make the empty base class a class template.* If the problem is that the same address for two objects cannot have the same type, let's change the type so that it's based on the most-derived type!

```
template<typename T> struct empty_class { };

struct smaller_class : empty_class<smaller_class> {
    long long int m_x[2];
};

struct bigger_class : empty_class<bigger_class> {
    smaller_class m_sc;
};

struct biggest_class : empty_class<biggest_class> {
    bigger_class m_bc;
    smaller_class m_sc;
};
```

The empty base class came from a class *template*, but a template isn't a "real" type until it is instantiated. By handing the derived class to the template, the template can also perhaps include special functions for specific classes, but this is probably not the point. The point is that at compile time, it's still pretty easy to check if a class derives from `empty_class<T>` for any `T`. Meanwhile, `is_same<empty_class<bigger_class>, empty_class<biggest_class>>` is `false`, and the compiler doesn't need to provide a unique address for adjacent instances. Problem solved!

```
*** Dumping AST Record Layout
         0 | struct bigger_class
         0 |   struct empty_class<struct bigger_class> (base) (empty)
         0 |   struct smaller_class m_sc
         0 |     struct empty_class<struct smaller_class> (base) (empty)
         0 |     long long[2] m_x
           | [sizeof=16, dsize=16, align=8,
           |  nvsize=16, nvalign=8]

*** Dumping AST Record Layout
         0 | struct biggest_class
         0 |   struct empty_class<struct biggest_class> (base) (empty)
         0 |   struct bigger_class m_bc
         0 |     struct empty_class<struct bigger_class> (base) (empty)
         0 |     struct smaller_class m_sc
         0 |       struct empty_class<struct smaller_class> (base) (empty)
         0 |       long long[2] m_x
        16 |   struct smaller_class m_sc
        16 |     struct empty_class<struct smaller_class> (base) (empty)
        16 |     long long[2] m_x
           | [sizeof=32, dsize=32, align=8,
           |  nvsize=32, nvalign=8]
```

Revisiting the "identity" example above, the `assert` no longer compiles because `empty_class` is a class template, and if you provide the correct template parameters so that it accepts the provided addresses, it then complains about comparing distinct pointer types. The function, `f()` now must be

a function template, and the two invocations in the example call two distinct functions, with no possibility of confusing the identities of the class and its member data.

4 Final Thoughts

This kind of problem is one that you might believe is working as expected for a years before realizing it's not quite right. It's not (generally) a serious issue, as we're talking about bytes, but the lost data is completely useless and wasted in many cases. I offer the following suggestions:

1. If it's at all reasonable, validate the size of your class with `static_assert`. For a class with a lot of data, or something algorithmic and frequently changing, this can be tedious and quite unnecessary. For a data class (think, a UUID class), however, you probably expect a specific size by design. Validate it.
2. Check out things that people tend to add to classes to adjust the interface — like `boost::noncopyable`. They may not be as harmless as they seem.
3. Change any "tag" classes in the code, and convert them to a template when reasonable. Perhaps, too, by using `std::enable_if`, the tag is not even needed.