More C++ Idioms/Empty Base Optimization

Contents

Empty Base Optimization

Intent

Optimize storage for data members of empty class types

Also Known As

- EBCO: Empty Base Class Optimization
- Empty Member Optimization

Motivation

Empty classes come up from time to time in C++. C++ requires empty classes to have non-zero size to ensure object identity. For instance, an array of `EmptyClass` below has to have non-zero size because each object identified by the array subscript must be unique. Pointer arithmetic will fall apart if `sizeof(EmptyClass)` is zero. Often the size of such a class is one.

```
class EmptyClass {};
EmptyClass arr[10]; // Size of this array can't be zero.
```

When the same empty class shows up as a data member of other classes, it may consume more than a single byte. Compilers align data depending on members sizes according to architecture requirements. The padding bytes are inserted to satisfy the requirements but serve no other useful purpose. Avoiding wastage of memory is generally desirable and may be critical in certain cases: deriving structure with single member of any type from empty base or including empty member will typically make it 2x bigger because of alignment requirements.

Solution and Sample Code

C++ makes special exemption for empty classes when they are inherited from. The compiler is allowed to flatten the inheritance hierarchy in a way that the empty base class does not consume space. For instance, in the following example, `sizeof(AnInt)` is 4 on 32 bit architectures and `sizeof(AnotherEmpty)` is 1 byte even though both of them inherit from the `EmptyClass`

```
class AnInt  : public EmptyClass
{
   int data;
};   // size = sizeof(int)

class AnotherEmpty : public EmptyClass {};  // size =  1
```

EBCO makes use of this exemption in a **systematic** way. It may not be desirable to naively move the empty classes from member-list to base-class-list because that may expose interfaces that are otherwise hidden from the users. For instance, the following way of applying EBCO will apply the optimization but may have undesirable side-effects: The signatures of the functions (if any in E1, E2) are now visible to the users of class `Foo` (although they can't call them because of private inheritance).

```
class E1 {};
class E2 {};

// **** before EBCO ****

class Foo {
  E1 e1;
  E2 e2;
  int data;
}; // sizeof(Foo) = 8

// **** after EBCO ****

class Foo : private E1, private E2 {
  int data;
}; // sizeof(Foo) = 4
```

A practical way of using EBCO is to combine the empty members into a single member that flattens the storage. The following template `BaseOptimization` applies EBCO on its first two type parameter. The `Foo` class above has been rewritten to use it.

```
template <class Base1, class Base2, class Member>
struct BaseOptimization : Base1, Base2
{
   Member member;
   BaseOptimization() {}
   BaseOptimization(Base1 const& b1, Base2 const & b2, Member const& mem)
       : Base1(b1), Base2(b2), member(mem) { }
};

class Foo {
  BaseOptimization<E1, E2, int> data;
}; // sizeof(Foo) = 4
```

With this technique, there is no change in the inheritance relationship of the `Foo` class. And, because `BaseOptimization` declares no member functions, it also avoids the problem of accidentally overriding a function from the base classes. Note that in the approach shown above it is critical that the base classes do not conflict with each other. That is, `Base1` and `Base2` are part of independent hierarchies.

Caveat

Object identity issues do not appear to be consistent across compilers. The addresses of the empty objects may or may not be the same. For instance, consider the below.

```
template <class Base1, class Base2, class Member>
struct BaseOptimizationWithFunctions : Base1, Base2
{
   Member member;
   BaseOptimizationWithFunctions() {}
   BaseOptimizationWithFunctions(Base1 const& b1, Base2 const & b2, Member
const& mem)
       : Base1(b1), Base2(b2), member(mem) { }
   Base1 * first()  { return this; }
   Base2 * second() { return this; }
};
```

The pointers returned by the `first` and `second` member functions may be the same on some compilers and different on others. See more discussion on StackOverflow (http://stackoverflow.com/questions/7694158/boost-compressed-pair-and-addresses-of-empty-objects)

Known Uses

- `boost::compressed_pair` (http://www.boost.org/doc/libs/1_47_0/libs/utility/compressed_pair.htm) makes use of this technique to optimize the size of the pair.

- A C++03 emulation of unique_ptr (http://home.roadrunner.com/~hinnant/unique_ptr03.html) also uses this idiom.

Related Idioms

References

- The Empty Base Class Optimization (EBCO) (http://www.informit.com/articles/article.aspx?p=31473&seqNum=2)
- The "Empty Member" C++ Optimization (http://www.cantrip.org/emptyopt.html)
- Internals of boost::compressed_pair (http://www.ibm.com/developerworks/aix/library/au-boostutilities/index.html)
