Determining the Size of a Class Object

By Girish Shetty

There are many factors that decide the size of an object of a class in C++. These factors are:

1. Size of all non-static data members
2. Order of data members
3. Byte alignment or byte padding
4. Size of its immediate base class
5. The existence of virtual function(s) (Dynamic polymorphism using virtual functions).
6. Compiler being used
7. Mode of inheritance (virtual inheritance)

Size of all non-static data members

Only non-static data members will be counted for calculating sizeof class/object.

```
class A {
private:
        float iMem1;
        const int iMem2;
        static int iMem3;
        char iMem4;
};
```

For an object of class A, the size will be the size of float iMem1 + size of int iMem2 + size of char iMem4. Static members are really not part of the class object. They won't be included in object's layout.

Order of data members

The order in which one specifies data members also alters the size of the class.

```
class C {
        char c;
        int int1;
        int int2;
        int i;
        long l;
        short s;
};
```

The size of this class is 24 bytes. Even though char c will consume only 1 byte, 4 bytes will be allocated for it, and the remaining 3 bytes will be wasted (holes). This is because the next member is an int, which takes 4 bytes. If we don't go to the next (4th) byte for storing this integer member, the memory access/modify cycle for this integer will be 2 read cycles. So the compiler will do this for us, unless we specify some byte padding/packing.

If I re-write the above class in different order, keeping all my data members like below:

```
class C {
        int int1;
```

```
        int int2;
        int i;
        long l;
        short s;
        char c;
};
```

Now the size of this class is 20 bytes.

In this case, it is storing c, the char, in one of the slots in the hole in the extra four bytes.

Byte alignment or byte padding

As mentioned above, if we specify 1 byte alignment, the size of the class above (class C) will be 19 in both cases.

Size of its immediate base class

The size of a class also includes size of its immediate base class.

Let's take an example:

```
Class B {
...
        int iMem1;
        int iMem2;
}

Class D: public B {
...
        int iMem;
}
```

In this case, sizeof(D) is will also include the size of B. So it will be 12 bytes.

The existence of virtual function(s)

Existence of virtual function(s) will add 4 bytes of virtual table pointer in the class, which will be added to size of class. Again, in this case, if the base class of the class already has virtual function(s) either directly or through its base class, then this additional virtual function won't add anything to the size of the class. Virtual table pointer will be common across the class hierarchy. That is

```
class Base {
public:
...
        virtual void SomeFunction(...);
private:
        int iAMem
};

class Derived : public Base {
...
        virtual void SomeOtherFunction(...);
private:
        int iBMem
};
```

In the example above, sizeof(Base) will be 8 bytes--that is sizeof(int iAMem) + sizeof(vptr). sizeof(Derived) will be 12 bytes, that is sizeof(int iBMem) + sizeof(Derived). Notice that the existence of virtual functions in class Derived won't add

anything more. Now Derived will set the vptr to its own virtual function table.

Compiler being used

In some scenarios, the size of a class object can be compiler specific. Let's take one example:

```
class BaseClass {
        int a;
        char c;
};

class DerivedClass : public BaseClass {
        char d;
        int i;
};
```

If compiled with the Microsoft C++ compiler, the size of DerivedClass is 16 bytes. If compiled with gcc (either c++ or g++), size of DerivedClass is 12 bytes.

The reason for sizeof(DerivedClass) being 16 bytes in MC++ is that it starts each class with a 4 byte aligned address so that accessing the member of that class will be easy (again, the memory read/write cycle).

Mode of inheritance (virtual inheritance)

In C++, sometimes we have to use virtual inheritance for some reasons. (One classic example is the implementation of final class in C++.) When we use virtual inheritance, there will be the overhead of 4 bytes for a virtual base class pointer in that class.

```
class ABase{
        int iMem;
};

class BBase : public virtual ABase {
        int iMem;
};

class CBase : public virtual ABase {
        int iMem;
};

class ABCDerived : public BBase, public CBase {
        int iMem;
};
```

And if you check the size of these classes, it will be:

- Size of ABase : 4
- Size of BBase : 12
- Size of CBase : 12
- Size of ABCDerived : 24

Because BBase and CBase are derived from ABase virtually, they will also have an virtual base pointer. So, 4 bytes will be added to the size of the class (BBase and CBase). That is sizeof ABase + size of int + sizeof Virtual Base pointer.

Size of ABCDerived will be 24 (not 28 = sizeof (BBase + CBase + int member)) because it will maintain only one Virtual Base pointer (Same way of maintaining virtual table pointer).