Memory Layout for Multiple and Virtual Inheritance

(By Edsko de Vries, January 2006)

Warning. This article is rather technical and assumes a good knowledge of C++ and some assembly language.

In this article we explain the object layout implemented by `gcc` for multiple and virtual inheritance. Although in an ideal world C++ programmers should not need to know these details of the compiler internals, unfortunately the way multiple (and especially virtual) inheritance is implemented has various non-obvious consequences for writing C++ code (in particular, for downcasting pointers, using pointers to pointers, and the invocation order of constructors for virtual bases). If you understand how multiple inheritance is implemented, you will be able anticipate these consequences and deal with them in your code. Also, it is useful to understand the cost of using virtual inheritance if you care about efficiency. Finally, it is interesting :-)

Multiple Inheritance

First we consider the relatively simple case of (non-virtual) multiple inheritance. Consider the following C++ class hierarchy.

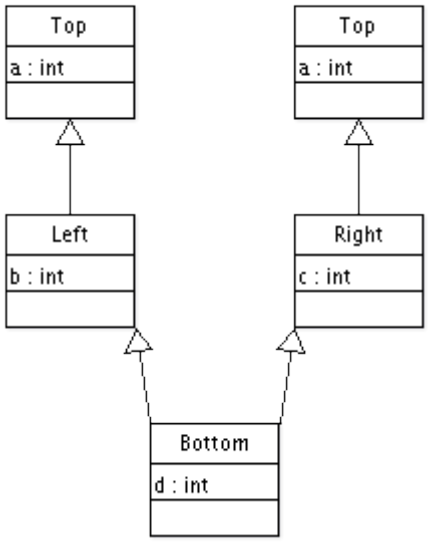
```
class Top
{
public:
   int a;
};

class Left : public Top
{
public:
   int b;
};

class Right : public Top
{
public:
   int c;
};

class Bottom : public Left, public Right
{
public:
   int d;
};
```

Using a UML diagram, we can represent this hierarchy as



Note that `Top` is inherited from *twice* (this is known as *repeated inheritance* in Eiffel). This means that an object `bottom` of type `Bottom` will have *two* attributes called `a` (accessed as `bottom.Left::a` and `bottom.Right::a`).

How are `Left`, `Right` and `Bottom` laid out in memory? We show the simplest case first. `Left` and `Right` have the following structure:

Left	**Right**
Top::a	Top::a
Left::b	Right::c

Note that the first attribute is the attribute inherited from `Top`. This means that after the following two assignments

```
Left* left = new Left();
Top* top = left;
```

`left` and `top` can point to the exact same address, and we can treat the `Left` object as if it were a `Top` object (and obviously a similar thing happens for `Right`). What about `Bottom`? `gcc` suggests

Bottom
Left::Top::a
Left::b
Right::Top::a

Right::c
Bottom::d

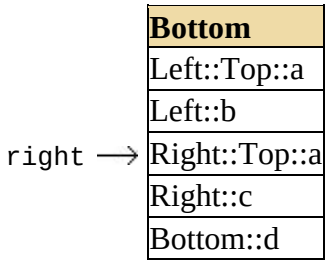
Now what happens when we upcast a `Bottom` pointer?

```
Bottom* bottom = new Bottom();
Left* left = bottom;
```

This works out nicely. Because of the memory layout, we can treat an object of type `Bottom` as if it were an object of type `Left`, because the memory layout of both classes coincide. However, what happens when we upcast to `Right`?

```
Right* right = bottom;
```

For this to work, we have to adjust the pointer value to make it point to the corresponding section of the `Bottom` layout:



After this adjustment, we can access `bottom` through the `right` pointer as a normal `Right` object; however, `bottom` and `right` now point to *different* memory locations. For completeness' sake, consider what would happen when we do

```
Top* top = bottom;
```

Right, nothing at all. This statement is ambiguous: the compiler will complain

```
error: `Top' is an ambiguous base of `Bottom'
```

The two possibilities can be disambiguated using

```
Top* topL = (Left*) bottom;
Top* topR = (Right*) bottom;
```

After these two assignments, `topL` and `left` will point to the same address, as will `topR` and `right`.

Virtual Inheritance

To avoid the repeated inheritance of `Top`, we must inherit *virtually* from `Top`:

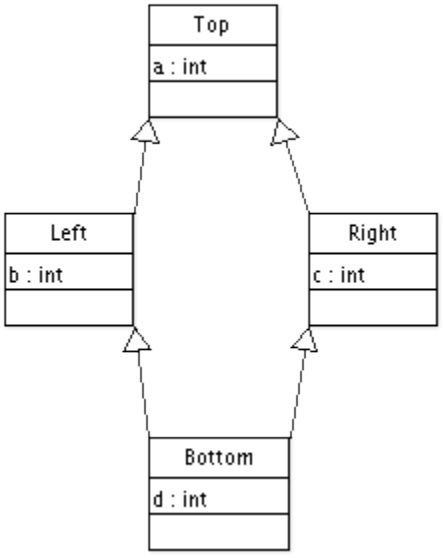
```
class Top
{
public:
   int a;
};

class Left : virtual public Top
{
public:
   int b;
};

class Right : virtual public Top
{
public:
   int c;
};

class Bottom : public Left, public Right
{
public:
   int d;
};
```

This yields the following hierarchy (which is perhaps what you expected in the first place)



while this may seem more obvious and simpler from a programmer's point of view, from the compiler's point of view, this is vastly more complicated. Consider the layout of `Bottom` again. One (non) possibility is

Bottom

Left::Top::a

Left::b

Right::c

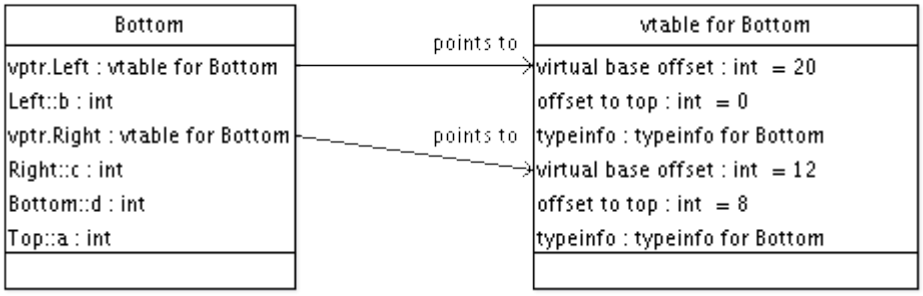
Bottom::d

The advantage of this layout is that the first part of the layout collides with the layout of `Left`, and we can thus access a `Bottom` easily through a `Left` pointer. However, what are we going to do with

```
Right* right = bottom;
```

Which address do we assign to `right`? After this assignment, we should be able to use `right` as if it were pointing to a regular `Right` object. However, this is impossible! The memory layout of `Right` itself is completely different, and we can thus no longer access a "real" `Right` object in the same way as an upcasted `Bottom` object. Moreover, no other (simple) layout for `Bottom` will work.

The solution is non-trivial. We will show the solution first and then explain it.



You should note two things in this diagram. First, the order of the fields is completely different (in fact, it is approximately the reverse). Second, there are these new `vptr` pointers. These attributes are automatically inserted by the compiler when necessary (when using virtual inheritance, or when using virtual functions). The compiler also inserts code into the constructor to initialise these pointers.

The `vptrs` (virtual pointers) index a "virtual table". There is a `vptr` for every virtual base of the class. To see how the virtual table (**vtable**) is used, consider the following C++ code.

```
Bottom* bottom = new Bottom();
Left* left = bottom;
int p = left->a;
```

The second assignment makes `left` point to the *same address* as `bottom` (i.e., it points to the "top" of the `Bottom` object). We consider the compilation of the last assignment (slightly

simplified):

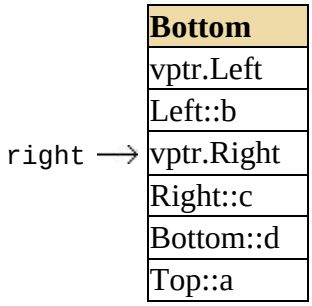
```
movl  left, %eax        # %eax = left
movl  (%eax), %eax      # %eax = left.vptr.Left
movl  (%eax), %eax      # %eax = virtual base offset
addl  left, %eax        # %eax = left + virtual base offset
movl  (%eax), %eax      # %eax = left.a
movl  %eax, p           # p = left.a
```

In words, we use `left` to index the virtual table and obtain the "virtual base offset" (**vbase**). This offset is then added to `left`, which is then used to index the `Top` section of the `Bottom` object. From the diagram, you can see that the virtual base offset for `Left` is 20; if you assume that all the fields in `Bottom` are 4 bytes, you will see that adding 20 bytes to `left` will indeed point to the `a` field.

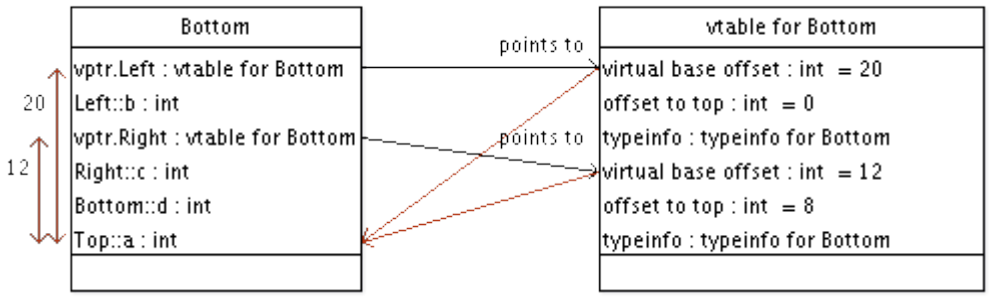
With this setup, we can access the `Right` part the same way. After

```
Bottom* bottom = new Bottom();
Right* right = bottom;
int p = right->a;
```

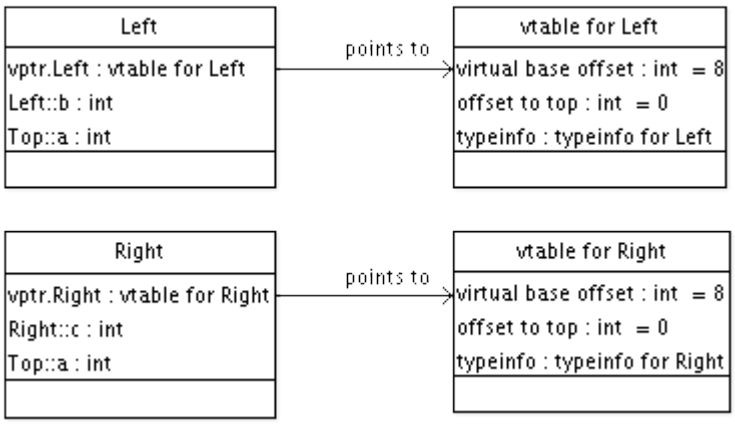
`right` will point to the appropriate part of the `Bottom` object:



The assignment to `p` can now be compiled in the *exact same way* as we did previously for `Left`. The only difference is that the `vptr` we access now points to a different part of the virtual table: the virtual base offset we obtain is 12, which is correct (verify!). We can summarise this visually:



Of course, the point of the exercise was to be able to access real `Right` objects the same way as upcasted `Bottom` objects. So, we have to introduce `vptrs` in the layout of `Right` (and `Left`) too:



Now we can access a `Bottom` object through a `Right` pointer without further difficulty. However, this has come at rather large expense: we needed to introduce virtual tables, classes needed to be extended with one or more virtual pointers, and a simple attribute lookup in an object now needs two indirections through the virtual table (although compiler optimizations can reduce that cost somewhat).

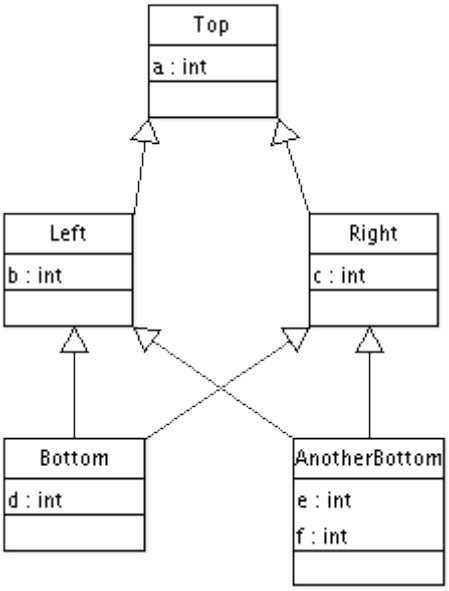
Downcasting

As we have seen, casting a pointer of type `DerivedClass` to a pointer of type `SuperClass` (in other words, upcasting) may involve adding an offset to the pointer. One might be tempted to think that downcasting (going the other way) can then simply be implemented by subtracting the same offset. And indeed, this is the case for non-virtual inheritance. However, virtual inheritance (unsurprisingly!) introduces another complication.

Suppose we extend our inheritance hierarchy with the following class.

```
class AnotherBottom : public Left, public Right
{
public:
   int e;
   int f;
};
```

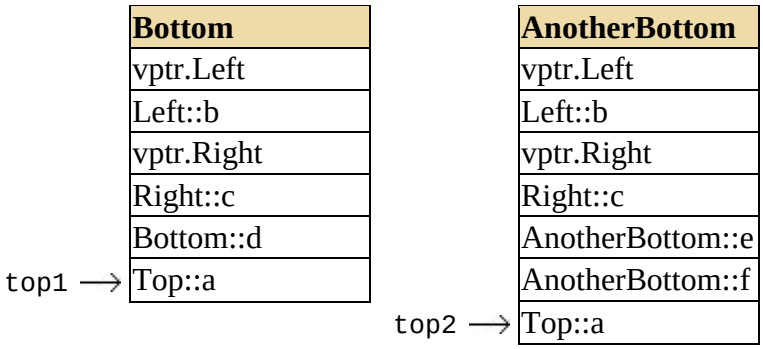
The hierarchy now looks like



Now consider the following code.

```
Bottom* bottom1 = new Bottom();
AnotherBottom* bottom2 = new AnotherBottom();
Top* top1 = bottom1;
Top* top2 = bottom2;
Left* left = static_cast<Left*>(top1);
```

The following diagram shows the layout of `Bottom` and `AnotherBottom`, and shows where `top` is pointing after the last assignment.



Now consider how to implement the *static* cast from `top1` to `left`, while taking into account that we do not know whether `top1` is pointing to an object of type `Bottom` or an object of type `AnotherBottom`. It can't be done! The necessary offset depends on the runtime type of `top1` (20 for `Bottom` and 24 for `AnotherBottom`). The compiler will complain:

```
error: cannot convert from base `Top' to derived type `Left'
via virtual base `Top'
```

Since we need runtime information, we need to use a dynamic cast instead:

```
Left* left = dynamic_cast<Left*>(top1);
```

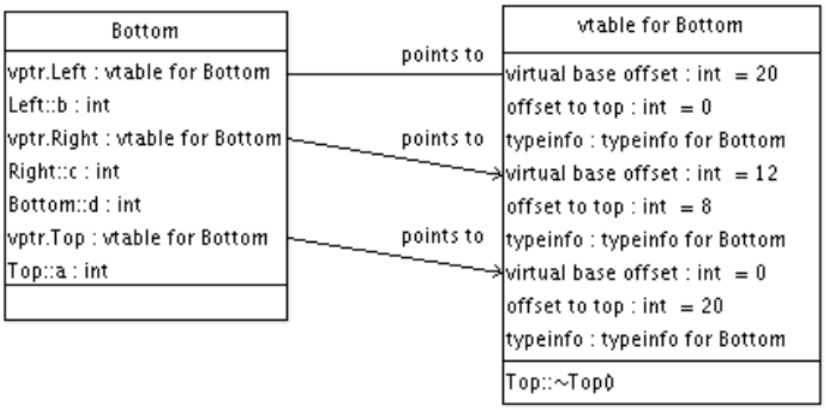
However, the compiler is still unhappy:

```
error: cannot dynamic_cast `top' (of type `class Top*') to type
   `class Left*' (source type is not polymorphic)
```

The problem is that a dynamic cast (as well as use of `typeid`) needs runtime type information about the object pointed to by `top1`. However, if you look at the diagram, you will see that all we have at the location pointed to by `top1` is an integer (`a`). The compiler did not include a `vptr.Top` because it did not think that was necessary. To force the compiler to include this `vptr`, we can add a virtual destructor to `Top`:

```
class Top
{
public:
   virtual ~Top() {}
   int a;
};
```

This change necessitates a `vptr` for `Top`. The new layout for `Bottom` is



(Of course, the other classes get a similar new `vptr.Top` attribute). The compiler now inserts a library call for the dynamic cast:

```
left = __dynamic_cast(top1, typeinfo_for_Top, typeinfo_for_Left, -1);
```

This function `__dynamic_cast` is defined in `libstdc++` (the corresponding header file is `cxxabi.h`); armed with the type information for `Top`, `Left` and `Bottom` (through `vptr.Top`), the cast can be executed. (The -1 parameter indicates that the relationship between `Left` and `Top` is

presently unknown). For details, refer to the implementation in tinfo.cc.

Concluding Remarks

Finally, we tie a couple of loose ends.

(In)variance of Double Pointers

This is were it gets slightly confusing, although it is rather obvious when you give it some thought. We consider an example. Assume the class hierarchy presented in the last section (Downcasting). We have seen previously what the effect is of

```
Bottom* b = new Bottom();
Right* r = b;
```

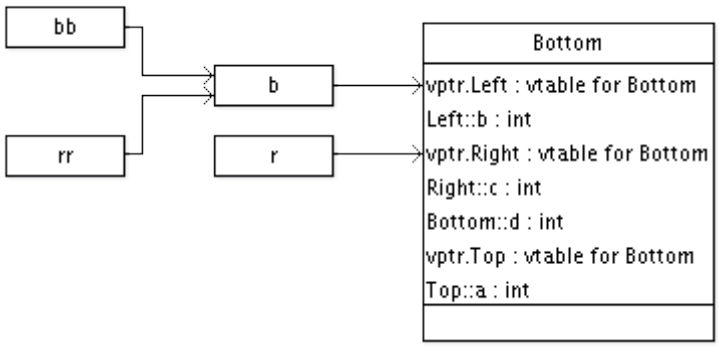
(the value of `b` gets adjusted by 8 bytes before it is assigned to `r`, so that it points to the `Right` section of the `Bottom` object). Thus, we can legally assign a `Bottom*` to a `Right*`. What about `Bottom**` and `Right**`?

```
Bottom** bb = &b;
Right** rr = bb;
```

Should the compiler accept this? A quick test will show that the compiler will complain:

```
error: invalid conversion from `Bottom**' to `Right**'
```

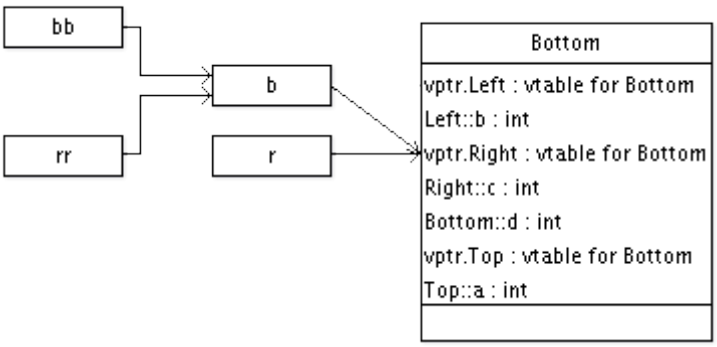
Why? Suppose the compiler would accept the assignment of `bb` to `rr`. We can visualise the result as:



So, `bb` and `rr` both point to `b`, and `b` and `r` point to the appropriate sections of the `Bottom` object. Now consider what happens when we assign to `*rr` (note that the type of `*rr` is `Right*`, so this assignment is valid):

```
*rr = b;
```

This is essentially the same assignment as the assignment to `r` above. Thus, the compiler will implement it the same way! In particular, it will adjust the value of `b` by 8 bytes before it assigns it to `*rr`. But `*rr` pointed to `b`! If we visualise the result again:



This is correct as long as we access the `Bottom` object through `*rr`, but as soon as we access it through `b` itself, all memory references will be off by 8 bytes — obviously a very undesirable situation.

So, in summary, even if `*a` and `*b` are related by some subtyping relation, `**a` and `**b` are *not*.

Constructors of Virtual Bases

The compiler must guarantee that all virtual pointers of an object are properly initialised. In particular, it guarantees that the constructor for all virtual bases of a class get invoked, and get invoked only once. If you don't explicitly call the constructors of your virtual superclasses (independent of how far up the tree they are), the compiler will automatically insert a call to their default constructors.

This can lead to some unexpected results. Consider the same class hierarchy again we have been considering so far, extended with constructors:

```
class Top
{
public:
   Top() { a = -1; }
   Top(int _a) { a = _a; }
   int a;
};

class Left : public Top
{
public:
   Left() { b = -2; }
   Left(int _a, int _b) : Top(_a) { b = _b; }
   int b;
};
```

```
class Right : public Top
{
public:
   Right() { c = -3; }
   Right(int _a, int _c) : Top(_a) { c = _c; }
   int c;
};

class Bottom : public Left, public Right
{
public:
   Bottom() { d = -4; }
   Bottom(int _a, int _b, int _c, int _d) : Left(_a, _b), Right(_a, _c)
  {
      d = _d;
  }
   int d;
};
```

(We consider the non-virtual case first.) What would you expect this to output:

```
Bottom bottom(1,2,3,4);
printf("%d %d %d %d %d\n", bottom.Left::a, bottom.Right::a,
   bottom.b, bottom.c, bottom.d);
```

You would probably expect (and get)

```
1 1 2 3 4
```

However, now consider the virtual case (where we inherit virtually from `Top`). If we make that single change, and run the program again, we instead get

```
-1 -1 2 3 4
```

Why? If you trace the execution of the constructors, you will find

```
Top::Top()
Left::Left(1,2)
Right::Right(1,3)
Bottom::Bottom(1,2,3,4)
```

As explained above, the compiler has inserted a call to the default constructor in `Bottom`, before the execution of the other constructors. Then when `Left` tries to call its superconstructor (`Top`), we find that `Top` has already been initialised and the constructor does not get invoked.

To avoid this situation, you should explicitly call the constructor of your virtual base(s):

```
Bottom(int _a, int _b, int _c, int _d)
: Top(_a), Left(_a,_b), Right(_a,_c)
{
   d = _d;
}
```

Pointer Equivalence

Once again assuming the same (virtual) class hierarchy, would you expect this to print “Equal”?

```
Bottom* b = new Bottom();
Right* r = b;

if(r == b)
   printf("Equal!\n");
```

Bear in mind that the two addresses are not *actually* equal (`r` is off by 8 bytes). However, that should be completely transparent to the user; so, the compiler actually subtracts the 8 bytes from `r` before comparing it to `b`; thus, the two addresses are considered equal.

Casting to `void*`

Finally, we consider what happens we can cast an object to `void*`. The compiler must guarantee that a pointer cast to `void*` points to the “top” of the object. Using the vtable, this is actually very easy to implement. You may have been wondering what the *offset to top* field is. It is the offset from the `vptr` to the top of the object. So, a cast to `void*` can be implemented using a single lookup in the vtable. Make sure to use a dynamic cast, however, thus:

```
dynamic_cast<void*>(b);
```

References

[1] CodeSourcery, in particular the C++ ABI Summary, the Itanium C++ ABI (despite the name, this document is referenced in a platform-independent context; in particular, the structure of the vtables is detailed here). The `libstdc++` implementation of dynamic casts, as well RTTI and name unmangling/demangling, is defined in tinfo.cc.

[2] The libstdc++ website, in particular the section on the C++ Standard Library API.

[3] C++: Under the Hood by Jan Gray.

[4] Chapter 9, “Multiple Inheritance” of *Thinking in C++ (volume 2)* by Bruce Eckel. The author has made this book available for download.