Multiple inheritance and the `this` pointer

Quick explanation: When using multiple inheritance, at least one of the base classes will see a different `this` pointer to that that the descendants see.

This occurs because when classes are compiled, the position of their members (both their data fields and their vtables etc.) is fixed - relative to the location of the instance, `this`; when you use multiple inheritance all except one of the base classes must see an offset `this` pointer or they would all put their data and vtables over the top of one another.

The C++ cast operators such as `dynamic_cast` and `static_cast` handle adjusting the `this` pointer automatically; you should *always* use them instead of assuming they have the same address, especially if casting to a `void*` at any point.

Table of contents

Single classes first

To understand how the `this` pointer works in class heirachies that use multiple inheritance, we start by examining how the compiler uses the instance pointer to access the class's members by dissecting a simpler example.

Say we've defined the following:

```
class Foo
{
public:
    int a;
    int b;
};

void modifyFoo(Foo* foo)
{
    foo->a = 1;
    foo->b = 2;
}

int main(int argc, char* argv[])
```

```
{
    Foo* foo = new Foo();
    modifyFoo(foo);
    delete foo;
}
```
When you create an instance of `Foo`, the `new` operator allocates a chunk of memory big enough to hold the class's members. Since `Foo` has no virtual methods and no ancestor classes, the class needs to be simply as large as the data members themselves are (possibly plus padding - adjustable with most compilers, usually will only be applied if you're using types smaller than words).

You can easily check this; `sizeof(Foo)` in the above example will return `sizeof(foo->a)+sizeof(foo->b)`, which is `2*sizeof(int)` - 8 bytes on a 32-bit platform and 16 on a 64-bit platform (we try this out [below](#)).

So for this example, our `foo` pointer in `main()` is really a pointer to an 8-byte or 16-byte chunk of memory containing the data members of Foo; at compile time, the compiler tracks the fact that this is a `Foo*` pointer, but that won't show up in the compiled code. If you have a look at the disassembly for `modifyFoo()`, what you'll find is that the expressions like `foo->a` get translated to '*(foo + the offset of Foo::a vs. the instance pointer, in bytes)' - so for our example, `&foo->a` will be the same pointer as `foo`, and `&foo->b` will be `foo+sizeof(foo->a)`, ie. `foo+4` on 32-bit platforms, `foo+8` on 64-bit platforms.

It's worth pausing to check that this actually works. Adding and running the following method:

```
void Foo::dump()
{
    cout
        << "Foo::dump():" << endl
        << "sizeof(Foo) = " << sizeof(Foo) << endl
        << "sizeof(*this) = " << sizeof(*this) <<
endl
        << "sizeof(a) + sizeof(b) = "
        <<  sizeof(a) + sizeof(b) << endl
        << "offsetof(Foo, a) = " << offsetof(Foo,
a) << endl
        << "offsetof(Foo, b) = " << offsetof(Foo,
b) << endl
        << "this = 0x" << hex << (intptr_t) this
<< endl
        << "&this->a = 0x" << hex << (intptr_t)
&this->a << endl
        << "&this->b = 0x" << hex << (intptr_t)
&this->b << endl
        << dec << endl;
}
```
will, on a 32-bit system with normal 32-bit packing, produce something like:
```
Foo::dump():
sizeof(Foo) = 8
sizeof(*this) = 8
sizeof(a) + sizeof(b) = 8
offsetof(Foo, a) = 0
offsetof(Foo, b) = 4
this = 0x80518a8
&this->a = 0x80518a8
&this->b = 0x80518ac
```
as we expected. (Obviously, the actual memory addresses are meaningless and will change.)

We'll see [later](#) that data members are not the only thing that contribute to the size of an object; polymorphic classes also need a vtable pointer, which is discussed in the section on [polymorphic types and the `dynamic_cast` operator](#).

Then simple inheritance

Next, we look at what happens to the `this` pointer if we descend from our base class, `Foo`, like this:

```
class Desc:
    public Foo
{
public:
    void dump();

public:
    int z;
};
```
When you create an instance of `Desc`, the memory allocated will now be just big enough to hold both `Foo`'s members and `Desc`'s members; in fact, the compiler will just put `Desc`'s members straight after `Foo`'s - this:
```
void Desc::dump()
{
    Foo::dump();

    cout
        << "Desc::dump():" << endl
        << "sizeof(Desc) = " << sizeof(Desc) <<
endl
        << "sizeof(*this) = " << sizeof(*this) <<
endl
        << "sizeof(a) + sizeof(b) + sizeof(z) = "
        <<  sizeof(a) + sizeof(b) + sizeof(z) <<
endl
        << "offsetof(Desc, a) = " <<
offsetof(Desc, a) << endl
        << "offsetof(Desc, b) = " <<
offsetof(Desc, b) << endl
        << "offsetof(Desc, z) = " <<
offsetof(Desc, z) << endl
        << "this = 0x" << hex << (intptr_t) this
<< endl
        << "&this->a = 0x" << hex << (intptr_t)
&this->a << endl
        << "&this->b = 0x" << hex << (intptr_t)
&this->b << endl
        << "&this->z = 0x" << hex << (intptr_t)
&this->z << endl
        << dec << endl;
}
```
will, in the same conditions as above, print something like:
```
Foo::dump():
sizeof(Foo) = 8
sizeof(*this) = 8
sizeof(a) + sizeof(b) = 8
offsetof(Foo, a) = 0
offsetof(Foo, b) = 4
this = 0x80518a8
&this->a = 0x80518a8
&this->b = 0x80518ac

Desc::dump():
sizeof(Desc) = 12
sizeof(*this) = 12
sizeof(a) + sizeof(b) + sizeof(z) = 12
offsetof(Desc, z) = 8
this = 0x80518a8
&this->z = 0x80518b0
```
So no suprises there - `Desc` instances are `sizeof(z)` bytes bigger than `Foo` instances, and the `z` member is just placed `sizeof(b)` bytes past the `b` member.

One interesting thing to note is that even though the instance is a `Desc`, `Foo::dump()` still only saw `sizeof(*this) == 8` (`sizeof` is evaluated at compile time, purely in the context of the class itself, so does not include the subclass data).

Multiple inheritance

But we're about to get a nasty surprise. Let's define another simple class, `Bar`, and then make a class, `Multi`, that descends from both `Foo` and `Bar`:

```
class Bar
{
public:
    void dump();

public:
    int c;
};

void Bar::dump()
{
    cout
        << "Bar::dump():" << endl
        << "sizeof(Bar) = " << sizeof(Bar) << endl
        << "sizeof(*this) = " << sizeof(*this) <<
endl
        << "sizeof(c) = " <<  sizeof(c) << endl
        << "offsetof(Bar, c) = " << offsetof(Bar,
c) << endl
        << "this = 0x" << hex << (intptr_t) this
<< endl
        << "&this->c = 0x" << hex << (intptr_t)
&this->c << endl
        << dec << endl;
}


class Multi:
        public Foo,
        public Bar
{
public:
        void dump();

public:
        int y;
};

void Multi::dump()
{
    Foo::dump();
    Bar::dump();

    cout
        << "Multi::dump():" << endl
        << "sizeof(Multi) = " << sizeof(Multi) <<
endl
        << "sizeof(*this) = " << sizeof(*this) <<
endl
        << "sizeof(a) + sizeof(b) + sizeof(c) +
sizeof(y) = "
        <<  sizeof(a) + sizeof(b) + sizeof(c) +
sizeof(y) << endl
        << "offsetof(Multi, y) = " <<
offsetof(Multi, y) << endl
        << "this = 0x" << hex << (intptr_t) this
<< endl
        << "&this->a = 0x" << hex << (intptr_t)
&this->a << endl
        << "&this->b = 0x" << hex << (intptr_t)
&this->b << endl
        << "&this->c = 0x" << hex << (intptr_t)
&this->c << endl
        << "&this->y = 0x" << hex << (intptr_t)
&this->y << endl
        << dec << endl;
}
```

this produces something like:

```
Foo::dump():
sizeof(Foo) = 8
sizeof(*this) = 8
sizeof(a) + sizeof(b) = 8
offsetof(Foo, a) = 0
offsetof(Foo, b) = 4
this = 0x80518e8
```

```
&this->a = 0x80518e8
&this->b = 0x80518ec

Bar::dump():
sizeof(Bar) = 4
sizeof(*this) = 4
sizeof(c) = 4
offsetof(Bar, c) = 0
this = 0x80518f0
&this->c = 0x80518f0

Multi::dump():
sizeof(Multi) = 16
sizeof(*this) = 16
sizeof(a) + sizeof(b) + sizeof(c) + sizeof(y) = 16
offsetof(Multi, y) = 12
this = 0x80518e8
&this->a = 0x80518e8
&this->b = 0x80518ec
&this->c = 0x80518f0
&this->y = 0x80518f4
```

Examine this output carefully, and note:

- **The `this` pointer was the same for `Foo::dump()` and `Multi::dump()`, but different for `Bar::dump()`.** See below for discussion.
- Despite that (in fact, because of it), all the `dump()` methods agreed on the actual memory locations of the fields (`a`, `b`, `c`, and `y`).
- All the `sizeof` values agreed with what we'd expect - the size of any base classes plus the size of the fields.

The clue to what's going on here is that in `Bar::dump()`, the offset of `c` is 0. The `Bar` class doesn't know that you're going to multiple-inherit from it (remember that you could use `Bar` on its own as well, so it has to be compiled independently this way, just the same way that `Foo` was), so it expects `this` to point to the start of its memory.

When you use multiple inheritance and make a call to `Bar::dump()`, the compiler passes a `this` that's been adjusted to point to the start of the `Bar` instance inside our `Multi`. It does this automatically, and normally you don't need to worry about it.

A note regarding `offsetof` and multiple inheritance

Before we move on to the implications of that, you may be wondering why I didn't output the following in `Multi::dump()`:

```
        << "offsetof(Multi, a) = " <<
offsetof(Multi, a) << endl
        << "offsetof(Multi, b) = " <<
offsetof(Multi, b) << endl
        << "offsetof(Multi, c) = " <<
offsetof(Multi, c) << endl
```

The answer is that it doesn't compile with some compilers! And justifiably too, in my opinion. gcc, for example, would output:

```
invalid reference to NULL ptr, use ptr-to-member
instead
```

To understand this error you have to look at the `stddef` definition of `offsetof` (it's a macro, not an operator like `sizeof`):

```
#define offsetof(TYPE, MEMBER) ((size_t) &((TYPE
*)0)->MEMBER)
```

- which basically says, 'hypothetically, if I had a TYPE object at address 0, what address would the MEMBER member of it have' (0 being a convenient choice because then we don't have to subtract anything off to get the offset of MEMBER vs. that object instance address).

gcc recognises that the only useful way to work out what the offsets are when you have multiple inheritance is to do the special casts discussed below, and since these can't possibly be done on the NULL pointer - you can never have an object instance at address 0 - it gives an error.

Some compilers, Borland's for example, are quite happy with the operation and will compile it and Borland's does indeed return the desired result in this case. But you really, really don't want to be addressing members using offsets with multiple inheritance, because you would have to see a different `offsetof` in the different classes - `Multi::dump` would have to see an `offsetof(Multi, c)` of 12, even though `offsetof(Bar, c)` is 0... Which would be an accident waiting to happen; IMHO it's not a bad thing if the compiler prevents this inconsistency from surfacing.

Obviously, you shouldn't normally be using `offsetof` to access members anyway, but it's definitely an even worse idea with MI. Regardless, doing so is not compatible with some common compilers, and should be avoided.

Which class is at zero?

One final note before we move on: why was it `Foo` that shared the same `this` pointer value as `Multi`, and `Bar` that had an offset `this`? The answer is simply that `Foo` was the first class that we listed as a superclass.

In C++ when you use multiple inheritance, the order in which you list the superclasses does matter: the superclasses will be constructed in that order (and therefore destroyed in the reverse order), and the members will be laid out in that order too, meaning that the first superclass will normally have 0 offset from the instance pointer (I say 'normally' because there is an exception if the subclass is polymorphic but the first superclass wasn't - we'll see why later).

The danger of unsafe casts

From the above discussion, one thing that's not clear is *why you should care*. The compiler correctly adjusts everything so that the base classes find their own members correctly, and the subclass has no problem accessing them; you don't need to do anything special when accessing the class from outside either. So why did I feel the need to write up a page explaining all this?

The answer is: **unsafe casts will not work correctly with multiple inheritance**.

C++ introduced a number of new casting operators: `static_cast`, `dynamic_cast`, `reinterpret_cast`, and `const_cast` (they look more like templates than operators, but that's what they're called). Each of these is different to C-style casts, and so we actually have 5 different cast operators.

A full discussion of what each of these casts does is beyond the scope of this document (if I get enough requests, I'll write up a seperate article), so we will confine ourselves to considering what happens when we apply these cast operators to instances of objects with multiple inheritance. `const_cast` is therefore irrelevant to our discussion (it just adds or removes `const` and/or `volatile` qualification), leaving us four.

Let's start by trying them. Given our `multi` instance in the program above, we get these pointers out of our casts:

Original `multi` pointer: `0x80518e8`	**Cast to `Multi*`**	**Cast to `Foo*`**	**Cast to `Bar*`**
C-style cast	0x 80518e8	0x 80518e8	0x 80518f0
C-style cast to `void*`, then C-style cast to type	0x 80518e8	0x 80518e8	0x 80518e8
`reinterpret_cast`	0x 80518e8	0x 80518e8	0x 80518e8

static_cast	0x 80518e8	0x 80518e8	0x 80518f0
dynamic_cast	0x 80518e8	0x 80518e8	0x 80518f0

(As usual, actual values are irrelevant - just look at which values are *different*. To try these out yourself, download the [example code](#) at the end of the article.)

The first surprise in the above results is that if you use the old, C-style casts (for example, '`(Bar*) multi`', the compiler will adjust the pointer value, as it does for `static_cast` and `dynamic_cast`. In other words, in C++, C-style casts do not just do a plain copy-the-appropriate-number-of-bits as they did in C; it may actually involve adjustment of the pointer value (I certainly didn't expect that!).

But the other bit of important news here is that if we do a C-style cast to `void*` and then cast the result of that to our second type, the compiler cannot perform its address-adjustment magic, because it has no way of knowing that the `void*` is actually a `Multi*`; and worse, the same problem occurs if you `static_cast` to `void*` and then `static_cast` to our desired type (eg '`static_cast<Bar*>(static_cast<void*>(multi))`') - that cast to `Bar*` returns the wrong result!.

`reinterpret_cast`, we can see, doesn't do any adjustment, it just reinterprets the literal bits of our `multi` pointer as another type of pointer completely. That will return the wrong value when casting classes with multiple inheritance, as we'd expect from the description of `reinterpret_cast`

These results give us the most important conclusions of this article: **Never use C-style casts to convert pointers or references between object types**, and secondly, **Avoid using `static_cast` to downcast, and never use it with multiple inheritance**. Instead:

- If you want to convert the pointer value literally, without adjustment and without real type checks, use `reinterpret_cast`. This will not come up very often.
- If you want to cast an instance pointer to a superclass, use `static_cast`; `dynamic_cast` will also work, but is unnecessary. `static_cast` checks the type relationships at compile-time, and has no unnecessary runtime overhead; `dynamic_cast` has runtime overhead and also imposes the extra requirement discussed below - but you might still prefer to use it for consistency sometimes.
- **If you want to convert a superclass pointer down to a descendant type, always use `dynamic_cast`**; it will check at runtime that the pointer is in fact an instance of the descendant type, and will adjust it if necessary, making this the only option that works with downcasting objects with multiple inheritance. See the section on [downcasting](#) below, which explains [one change](#) you may have to make to make this compile.

(Note that this summary discusses only casting object instance pointers between related types - there are many other things you might do with them, outside the scope of this document.)

Downcasting

'Downcasting' is the term used to describe casting a pointer or reference to a class 'down' the class heirachy - to one of its subclasses.

Why `reinterpret_casts` can't downcast

When you `reinterpret_cast` an instance pointer, the operator simply makes a pointer of the requested type with exactly the same address as the original. Therefore while `reinterpet_cast` will work fine if you are downcasting from a superclass to a subclass with no multiple inheritance anywhere in it's ancestry, it won't work in the general case - it'll compile and run, but will produce the wrong pointer.

For the example above, that pointer will be right for casting a `void*` down to a `Foo*`, or a `Foo*` down to a `Multi*`, but that's only because it happens that both `Foo` and `Multi` start at offset 0 (and even that can't be relied upon - if `Multi` is later made polymorphic, it will no longer be true); it won't work if you try to downcast a `void*` that actually points to a `Multi*` to a `Bar*`, because while `Bar` starts at offset 0 on its own, when it's a part of a `Multi`, it starts at offset 8 (or whatever).

(Note that this document makes no attempt to provide a general explanation of the utility or otherwise of `reinterpret_cast`, discussing only what's relevant to the matter of multiple inheritance.)

Why `static_casts` can't *safely* downcast

One might at first hope that `static_cast` could do the job. Sadly however, when starting with a pointer to a `Foo` or a `Bar`, there's no way to know, at compile time, from that type alone (which is all `static_cast` inspects) that it's actually a pointer to an object that's not only a `Foo` or a `Bar` but is in fact a `Multi`.

So the compiler will essentially do the same as it did for `reinterpret_cast`; if you cast our `multi` instance to a `void*` and then try to `static_cast` it back down to a `Bar`, it'll compile, but return the wrong result, because it doesn't know that this is not really a `Bar` - it's a `Multi`, which puts the `Bar` superclass instance at a nonzero offset. The compiler can't even check that the pointer you're casting is composed of a `Bar` instance at all, let alone know where the `Bar` is placed inside; so it just unsafely does the cast (I wish it gave an error - after all you can get that behaviour with other operators, if you really want it).

Again, note that this document makes no attempt to provide a general explanation of the use and limitations of `static_cast`; there's a lot more to know about this operator not directly relevant here.

Polymorphic types: Why `dynamic_casts` *can* downcast

The only operator that can successfully downcast is `dynamic_cast`. However, there is one catch: in order for downcasts to be possible, the base type you are casting from must be *polymorphic*.

Polymorphic is a general OOP term with which I assume readers are familiar. However, in the context of the C++ language, it has a specific and tangible meaning: *polymorphic classes* are those that have at least one `virtual` method (including destructors, and also including pure virtual methods - so you can just define a dummy private pure virtual method if you want to force a base class to be polymorphic but have no other `virtuals`).

The implication of a C++ class being polymorphic is that it has a *vtable*. A vtable ('virtual(s) table') is simply a static data structure (one per class, not per instance - all `Foo` instances share the same vtable, all `Bar` instances share another) that lists a number of things, including the table of all the virtual methods, and some metadata regarding the class itself and its ancestors, the latter being the part that is of use to us here.

vtables are created by the compiler and generally remain hidden to the application; you shouldn't ever need to access them directly. A fully detailed discussion of the contents of vtables falls outside the scope of this document, but thankfully, you don't need to know exactly what's in them for the problem at hand; we'll just look at how they're placed and how they help with downcasts.

If we make, say, our `Foo` class polymorphic, which we could do by (for example) adding a `virtual` tag to our definition of `dump()`, we notice that the `sizeof(Foo)` increases by the size of a pointer (4 bytes on my 32-bit PC), and that `offsetof(Foo, a)` and `offsetof(Foo, b)` will both increase by the size of a pointer too.

This is not a coincidence: **when the compiler compiles the polymorphic class, it creates a vtable for it, and it now stores a pointer to this vtable at the very start of every instance** (storing that pointer is one of the many jobs that is performed automatically by the constructor).

When the compiler compiles a normal subclass of a polymorphic class, it will make a new vtable for that subclass, but the instances of this subclass don't have to have pointers to both the superclass vtables and the subclass vtables separately, because the vtable for the subclass includes everything that the vtable for the superclass contained. So the size overhead of being polymorphic for a normal class is just one pointer per instance, regardless of how deep in the ancestry it is (again, the vtable itself is just one per class).

But this isn't quite enough if we're using multiple inheritance. Making any of the superclasses polymorphic is enough to make `Multi` polymorphic. But, if we made `Bar` polymorphic too, then it too would always have a vtable pointer at the start of its instance data. And since all `Multi` instances have what is effectively a standalone `Bar` instance embedded somewhere inside them (remember that you can cast the subclass instance to that superclass, so it must be able to work just like a real standalone `Bar` would), `Multi` will now have to have two vtable pointers - at the very start of our `Multi` instance will be the pointer to the combined `Foo+Multi` vtable, and at the offset of the `Bar` inside the `Multi` will be the pointer to the combined `Bar+Multi` vtable.

You don't need to know that, but if you're interested in how vtables work, convince yourself that this is so. You can test this theory out by noting that adding virtual methods to `Multi` adds no more instance size overhead once the `Foo` superclass is polymorphic, but that making `Bar` polymorphic as well *does* increase the size of `Multi` instances by one pointer.

Let's get back on track. We've said that all instances of polymorphic classes have a vtable pointer at the start of the instance. This is great because now, **if we have a pointer to *any* polymorphic class, we can inspect the vtable and determine what the 'actual' type of that object is**. So, even if we were given a `Foo*` or a `Bar*`, if we inspected the vtable, we'd be able to see if this is actually a `Multi` instance, not just a plain `Foo` or `Bar` instance.

And this is just what the `dynamic_cast` operator does for us: **`dynamic_cast` inspects the vtable of the polymorphic class instance you pass it to check that is actually an instance of whatever type you are trying to convert to - and if so, then it calculates if an offset is required to do the conversion and does it**.

The offset for converting a `Foo*` to a `Multi*` would in the simplest case be 0 (and likewise for going back the other way). But to convert a `Bar*` to a `Multi*`, `dynamic_cast` would find from the vtable that the `Bar` instance is embedded some distance into the `Multi` (12 bytes when I run my test code), and it therefore subtracts that many bytes to find the correct address of the enclosing `Multi` from the `Bar*` pointer it started with.

So there you have it: **`dynamic_cast` is the solution to the downcasting problem; the unavoidable cost is that your classes must be polymorphic**. That constraint may annoy you sometimes because it means that you can't directly downcast from types like `void*`, but in practice you should find that even if you're stuffing your instance pointers into `void*`s for a callback or whatever from a C library you're using, you will at least know what the base class is, so you can statically cast to that, and then `dynamic_cast` to perform the downcast; if not, you'll just have to declare one. That one issue aside, the overhead is generally small enough to not be an issue.

Conclusion

Whether you are using multiple inheritance or not, remember:

- Don't rely on the `this` pointer having the same value everywhere if you're using multiple inheritance - cast it properly if necessary to get a 'canonical' pointer.
- Never hardcode the offset of members relative to the instance pointer, and avoid using `offsetof` unless strictly necessary (rare).
- Don't use C-style casts to convert pointers or references between object types - use `reinterpret_cast` if you want an unsafe, unadjusted conversion, or `static_cast` or `dynamic_cast` as below.
- If you want to cast an instance pointer to a superclass, use `static_cast` or `dynamic_cast`; the former is more efficient.
- If you want to convert a superclass pointer down to a descendant type, always use `dynamic_cast`.

Example code

I encourage readers to download the example code for this article and try out the variations and effects for themselves.

Postscript

As always, feedback is welcome.

Also cash. Cash is welcome too.

Actually, cash is probably *more* welcome than feedback, come to think of it.