Fast Virtual Functio
for Fun and Profit

Caleb Leak
Jan 21, 2019 · 8 min read

Virtual functions have clear intrinsic value; using them can make architectures significantly more flexible and solve a plethora of design problems.

When it comes to virtual functions (or methods, the terminology distinction isn't fruitful in this context) there are two main schools of thought for performance. The first is that they're slow, always have been, and always will be. Folks of this school will tell you that they're too slow to use in games. The second is that they were once slow, but modern compilers and architectures have made them efficient. Folks of this school will tell you to use them liberally, even in performance critical code, as they're basically free.

So, what's the truth? The unsurprising answer is: it depends. Depends on what? Well, I'm glad you asked. The answer is, like most indirections, it depends mostly on the cache and how your application interacts with it. In this article I'll dig into three things (with a focus on C++):

1. **How to avoid the performance hit of virtual functions.** I'll show how you can call virtual functions directly and avoid the cost of indirection.

2. **How to make large batches of virtual functions fast**. This a common problem in game engines where a scene may have thousands of objects, each with several not necessarily unique update functions. If these updates happen at framerate, then there will be tens to hundreds of thousands of calls per second.

3. **How to detect if a virtual function has been overridden.** This is another common problem in modern game engines where there's a plethora of functions to override. Unity for example has 64 such overrideable methods (they call them *messages*), the majority are no-ops for most objects. The engine can avoid calling them at all when this is detected.

I won't talk about *why* virtual functions are useful.

Understanding virtual functions ar

Before diving into the performance of
look at how virtual functions work. If
this section.

Let's start by creating a class hierarchy without virtual functions. We'll create greeters that simply greet people by their given name:

```cpp
#include <iostream>

using namespace std;

class GenericGreeter {
 public:
  void Greet(const char* name) {
    cout << "Hi " << name << "." << endl;
  }
};

class FriendlyGreeter : public GenericGreeter {
 public:
  void Greet(const char* name) {
    cout << "Hello " << name << "! It's a pleasure to meet you!" << endl;
  }
};

int main() {
  FriendlyGreeter* friendlyGreeter = new FriendlyGreeter;
  GenericGreeter* genericGreeter = (GenericGreeter*)friendlyGreeter;

  friendlyGreeter->Greet("Bob");
  genericGreeter->Greet("Alice");

  delete friendlyGreeter;

  return 0;
}
```

Greeter.cpp hosted with ♡ by GitHub view raw

The output of this is:

```
Hello Bob! It's a pleasure to
Hi Alice.
```

Poor Alice doesn't get a proper greetir

least functions with the same name) c

depending on how we cast it. Why is t

The compiler has no special knowledge about where data came from. To it, a `GenericGreeter*` constructed from `FriendlyGreeter` is the same as one constructed from a `GenericGreeter`, so it simply calls the base function.

Now let's do the same thing with virtual functions:

```cpp
class GenericGreeter {
 public:
  virtual void Greet(const char* name) {
    cout << "Hi " << name << "." << endl;
  }
};

class FriendlyGreeter : public GenericGreeter {
 public:
  virtual void Greet(const char* name) {
    cout << "Hello " << name << "! It's a pleasure to meet you!" << endl;
  }
};
```

GreeterVirtual.cpp hosted with ♡ by GitHub view raw

Now the output is:

```
Hello Bob! It's a pleasure to meet you!
Hello Alice! It's a pleasure to meet you!
```

The derived function is now called — somehow the compiler knows to do this. It still just has a `GenericGreeter` pointer with no special knowledge of how that data was constructed, but now there's more to this pointer than meets the eye.

Hiding inside `GenericGreeter` now is a pointer to a virtual function table or *VTable*. This table is simply a bit of static mem memory is static (and therefore point instance of a particular class will have somewhere it can be shared.

I think this is best illustrated with an e time we'll create our own VTables:

```
#include <iostream>

using namespace std;

// Our Greet function type.
typedef void (GreetFn)(void*, const char* name);

struct GenericGreeter_VTable {
  GreetFn* greet;
};

// Forward declare VTable instances.
extern GenericGreeter_VTable generic_vtable;
extern GenericGreeter_VTable friendly_vtable;

class GenericGreeter {
 public:
  GenericGreeter_VTable* vtable;

  GenericGreeter() {
    vtable = &generic_vtable;
  }

  static void GreetGeneric(void* _this, const char* name) {
    cout << "Hi " << name << "." << endl;
  }
};

class FriendlyGreeter : public GenericGreeter {
 public:
  FriendlyGreeter() {
    vtable = &friendly_vtable;
  }

  static void GreetFriendly(void* _this, const char* name) {
    cout << "Hello " << name << "! It's a pleasure to meet you!" << endl;
  }
};

// Create the static VTable instances
GenericGreeter_VTable generic_vtable
  (GreetFn*)&GenericGreeter::GreetGen
};

GenericGreeter_VTable friendly_vtable
  (GreetFn*)&FriendlyGreeter::GreetFriendly
};

int main() {
  FriendlyGreeter* friendlyGreeter = new FriendlyGreeter;
  GenericGreeter* genericGreeter = (GenericGreeter*)friendlyGreeter;

  friendlyGreeter->vtable->greet(friendlyGreeter, "Bob");
  genericGreeter->vtable->greet(genericGreeter, "Alice");

  delete friendlyGreeter;

  return 0;
}
```

Again, the output is:

```
Hello Bob! It's a pleasure to meet you!
Hello Alice! It's a pleasure to meet you!
```

And that’s the basics of VTables and virtual functions. With deeper hierarchies and multi-inheritance there can be multiple VTables, but they all follow this pattern.

Poking at the VTable

We saw in the previous section how VTables work, but we didn’t actually access any — not directly. Here we’ll go over locating and making use of the VTable.

Let’s first extend the example from the previous section to include more than one virtual function:

```
class GenericGreeter {
 public:
  virtual void Greet(const char* name) {
    cout << "Hi " << name << "." << endl;
  }

  virtual void Dismiss(const char* na
    cout << "Bye " << name << "." <<
  }
};

class FriendlyGreeter : public Generi
 public:
  virtual void Greet(const char* name) {
    cout << "Hello " << name << "! It's a pleasure to meet you!" << endl;
  }

  virtual void Dismiss(const char* name) {
    cout << "Farewell " << name << "! Until later!" << endl;
  }
};
```

ExtendedGreeter.cpp hosted with ♡ by GitHub view raw

Now let's define what the VTable looks like. Note that we're not implementing it ourselves, but simply making a convenient struct to understand it:

```
typedef void (GreetFn)(void*, const char* name);
struct GenericGreeter_VTable {
  GreetFn* greet;
  GreetFn* dismiss;
};
```

GreeterVTable.cpp hosted with ♡ by GitHub view raw

We're now ready to play with the VTable — but where is it? Well, it's hiding in plain sight. Any class that declares a new virtual function, not just overrides an existing one, gets an implicit pointer to a VTable added along side its member variables. By convention, VTable pointers come before all other members of the class — a stable convention followed by MSVC, gcc, and Clang. This makes them very easy to find. Continuing our example, a function to find a VTable pointer is as easy as:

It's worth noting that in this case taking in a `void*` would work instead of `GenericGreeter*` but is less robust. A VTable pointer is a member of whatever class in the hierarchy makes the VTable, like the example in the last section. As such they follow all the typical rules of multiple inheritance and `class WeirdGreeter : public`

`Weird, public GenericGreeter {…}` would have all of `Weird`'s members *before* all of `GenericGreeter`'s, including the VTabl

the function above does, sidesteps thi

Finally, let's put the pieces together ar
dispatch:

The output of this is:

```
Hello Bob! It's a pleasure to meet you!
Farewell Bob! Until later!
Hello Alice! It's a pleasure to meet you!
Farewell Alice! Until later!
```

Only the overridden functions were called!

When are virtual functions slow and how slow are they?

We've seen how you can call virtual functions directly, now let's see why you might want to.

Virtual functions are slow when you have a cache miss looking them up. As we'll see through benchmarks, they can be *very* slow. They can also be very fast when used carefully — to the point where it's impossible to measure the overhead.

Let's setup some tests and see for ourselves. We'll test several things:

1. **Repeatedly calling a non-virtual member function with caching allowed.** This will give us a good performance baseline.
2. **Repeatedly calling a virtual member function with caching allowed.** This will demonstrate the best, albeit unlikely, case with virtual functions. The same virtual function is called repeatedly in a tight loop, so we'd expect the VTable pointer, the VTable, and the function code itself to all remain cached.
3. **Repeatedly calling a non-virtual member function without caching.** We'll explicitly flush the cache to achieve another performance baseline. This will be our target performance for non-cached virtual functions.

4. **Repeatedly calling a virtual member function without caching.** This is the worst case performance scenario f and function code will not be in th

5. **Repeatedly calling a virtual mer** hope here is that we'll gain back s and be close to that of non-virtual

To test this this, we'll use a small hierarchy of incrementers. Each will add its value (a randomly initialized int) to a global variable repeatedly via an increment function. Adding from a member variable ensures that the object is loaded into memory, so we're playing fair, and being a random number will prevent the compiler from optimizing the member variable out.

We're now set to run some tests and see just how slow virtual functions are. To do this, we'll create many objects of each type and call their increment functions repeatedly. Below are the results, with number of objects (or calls, since each object's function is called once) vs runtime in seconds:

Amazingly, calling virtual functions directly is the fastest method, though it only marginally beats non-virtual functions. Without the direct calls, and when these functions aren't cached, there's 3x more overhead to the virtual call vs non-virtual.

Keep in mind that we're only comparing overhead. For very long running functions, the overhead cost will be moot.

Avoiding the performance hit of vi

As seen in the previous section, virtua
speed, we'll want to call virtual functi

To call a virtual function directly, we'll go to the VTable and grab a pointer directly to the function we want to call. "But wait," you say, "won't that incur the same penalty as a regular virtual function call?" And you're right, it will. The key here is that we can reuse this function pointer, either to call it with the same instance in the future or (as in the next section) on other instances of the same class.

To get the function pointer, we can first define a generic function to get the VTable:

Then it's as easy as casting the appropriate entry to the desired function pointer:

In the case above, it simply invokes the virtual function in slot 1. Where this technique really becomes useful is when a function is called repeatedly, either as a batch or on its own.

Running batches of virtual functions efficiently

In games and other interactive applications, it's common to run batches of virtual functions at a time. For example, running thousands of `Update()` functions from various scene objects each frame. We can take some of the tools from the previous sections and accomplish this efficiently.

The key insight we can leverage to run these batches efficiently is that cache misses on function calls kill performance. So how can we minimize cache misses? We can group our function calls to keep code in the cache!

One easy way to group virtual functions is with `std::set`, which maintains an ordered set of values. By sorting pairs of { function pointer, object pointer }, the set can easily be iterated over to call everything in sorted order. This is especially convenient because any scene manager would need this list anyways.

Below is a complete example. It uses `SceneObject` as its base class containing the overrideable `Update()` function. `SceneManager` holds these objects and allows all update functions be called as a single batch, grouping update functions.

With a framework like this in place, th

don't call empty functions.

Detecting when virtual functions a

In the previous example, we batched

the same function (but with different objects) for efficiency. This works fine if every child class of `SceneObject` needs an update function, but what if some don't? What if we have a plethora of overrideable functions like `LateUpdate`, `PhysicsUpdate`, and `DrawImmediate`? Surely we don't expect most of those to be overridden in each object, and it would waste cycles to call empty functions. Enter override checking.

To check if a function is overridden, you simply need to compare the function entries in each VTable. The caveat here is that you need an instance of the base object (for which the null-object pattern fits nicely) to compare against.

Let's update the example from the previous section with the new functions we just mentioned:

Now we can update the scene manager to conditionally add new objects to each batch only when they have overridden the appropriate function:

It's that easy! Of course, you'd probably want to start wrapping these up in nicer classes. You can also use some macro magic to guarantee your VTable struct always matches what's actually defined in the class.

Conclusion

Hopefully you’ve seen how useful play
not so hard. I encourage you to try thi
program and happier developers. The
write code like this:

Sign in to medium.com with Google

✕

Lincoln Lin
lincoln123lin@gmail.com

Continue as Lincoln

and not worry about undue overhead of registering functions.

The full source for all examples and tests is on GitHub. Happy hacking!

Programming Performance Game Development Game Engine Cpp