Audience: EWG, SG14: D0130R1  
Date: 28-02-2016

Reply to: Scott Wardle, Roberto Parolin  
[swardle@gmail.com](mailto:swardle@gmail.com), [robertoparolin@gmail.com](mailto:robertoparolin@gmail.com)

**Comparing Virtual Methods**

# I. Summary

It would be useful to compare if an instance of a class has a particular function in its virtual table. I will show we can do this using the GCC extension -Wno-pmf-conversions.

# II. Motivation

Entity systems in c++ are used often used in games. A simple (and maybe naïve) entity would be a class with an update virtual function. A simple (and maybe naïve) entity systems would be a vector of base Entity pointers. The main loop would call update on each Entity one per frame by looping though each entity and calling update.

Once you had the entity system programmers (or more likely content creators) would then add their own entity like soldiers or zombies or basketball players to the list of entities and you would update the whole system of basketball players, zombies and soldiers just by calling update. Programmers would probably setup the entity type content creators would say what entity would be part of the world.

Each game wants to have new types of entity as a basketball player does not move like a zombie. However the game team don’t want to handle all problems writing entity that walk is very hard. You need a whole animation system to make walk cycles etc… So games share large library of function between each other called an engine.

Engine programmers might write walking entity for a first person zombies and soldiers game. So even the basic walking entity with some different tuning can look good enough. The basketball player however maybe needs a better system as in a basketball game you are focused on how the player moves and looking right at him.

This story I think explains the 2 types of objects that are in entity systems.

* The “in engine” entity like the zombie and soldier
* The “game team overridden” entity for basketball player.

Now the game team that made the basketball player wants to add zombies to their basketball game they can.

The engine programming team does not know about the basketball player but wants to make zombies faster as everyone likes to have lots of zombies in their games. What they would like to do is take a group of “walking entity” and make a zombies group update function out of that. But basketball player that engine team didn’t write might still want to use the old walking entity functionality. This is where comparing virtual function come in.

We could separate each entities based on the update function in the virtual table moving these overridden objects out of the fast path and have one function that updated many non-overridden entities. Then update each overridden entity slowly afterwards.

Here are some advantages:

* No new data is necessary.
* Decrease i-cache misses
* Improve branch prediction, if statements can be predicted on more hardware then virtual calls.
* Allow for grouping objects even if we started out with poor maybe naïve design.
* Let’s the programmer reuse vtable as data not just functions. So do more with the same data.
* Let’s the user make a sort key that is as complicated as an objects vtable or as simple as one element of it.

To show the power of this idea. I created 3 loops see below for source code. (The full source is in cpp\_entity\_example\ \*.cpp \*.h)

1. Slow loop
   1. Calling a virtual function for each loop
2. Slow and complicated loop
   1. Trying to avoid calling “expensive” function with GetType virtual function.
3. Fast loop
   1. Trying to avoid calling “expensive” function with pointer to a 64 bit int like what would happen if I could compare to a method in the virtual table.

The function in my example are not very expensive so I can see virtual function call overhead clearly but here are the numbers in my example.

Speed Test Chart 1

## One Virtual Function

mytimer timer;

for (float t = 0.0f; t < 1.0; t += 0.05f) {

for (auto &a : entity\_vec) {

a->Update(t);

}

}

gSlowSimpleUpdateExampleTimers.emplace\_back(timer.stop());

## Two Virtual Functions

mytimer timer;

for (float t = 0.0f; t < 1.0; t += 0.05f) {

for (auto &a : entity\_vec) {

if (a->GetType() != entity\_lerp\_fast::type) {

a->Update(t);

}

}

entity\_lerp\_fast::UpdateAll(t);

}

gSlowComplicatedUpdateExampleTimers.emplace\_back(timer.stop());

## Enums

mytimer timer;

for (float t = 0.0f; t < 1.0; t += 0.05f) {

for (auto &a : entity\_vec) {

if (\*a->m\_typedata != entity\_lerp\_fast::type) {

a->Update(t);

}

}

entity\_lerp\_fast::UpdateAll(t);

}

gFastUpdateExampleTimers.emplace\_back(timer.stop());

## Member Functions Compare Using GCC extension -Wno-pmf-conversions

typedef void (entity::\*memfun)(float y) const;

memfun mf = &entity::Update;

as\_normfun snf = (as\_normfun)(&entity\_lerp\_fast\_impl::Update);

for (float t = 0.0f; t < 1.0; t += 0.05f) {

for (auto &a : entity\_vec) {

const entity& e = \*(&(\*a));

as\_normfun dnf = (as\_normfun)(e.\*mf);

if (snf != dnf) {

a->Update(t);

}

}

entity\_lerp\_fast::UpdateAll(t);

}

So why not just add an enum like the like in the fast example? The main problem is this can be a large refactor job. The simplest code is the one virtual function version. Once you see this as a hot spot in some profiler you will want to check code around that hot spot and not to change every single class if posable.

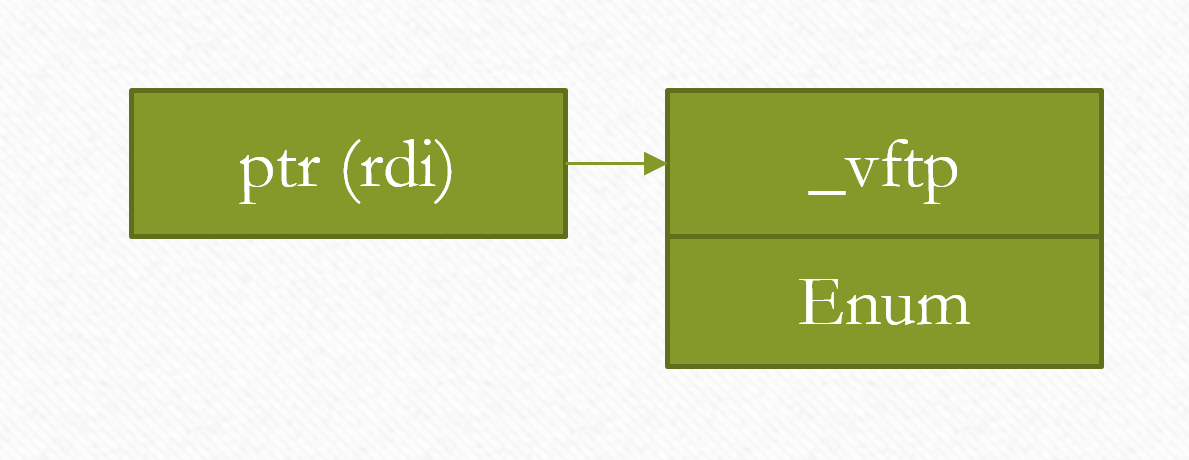
The ASM output of Enum version vs Member Function Compare is very close.

ASM listing 1 Enum vs Member Functions Compare

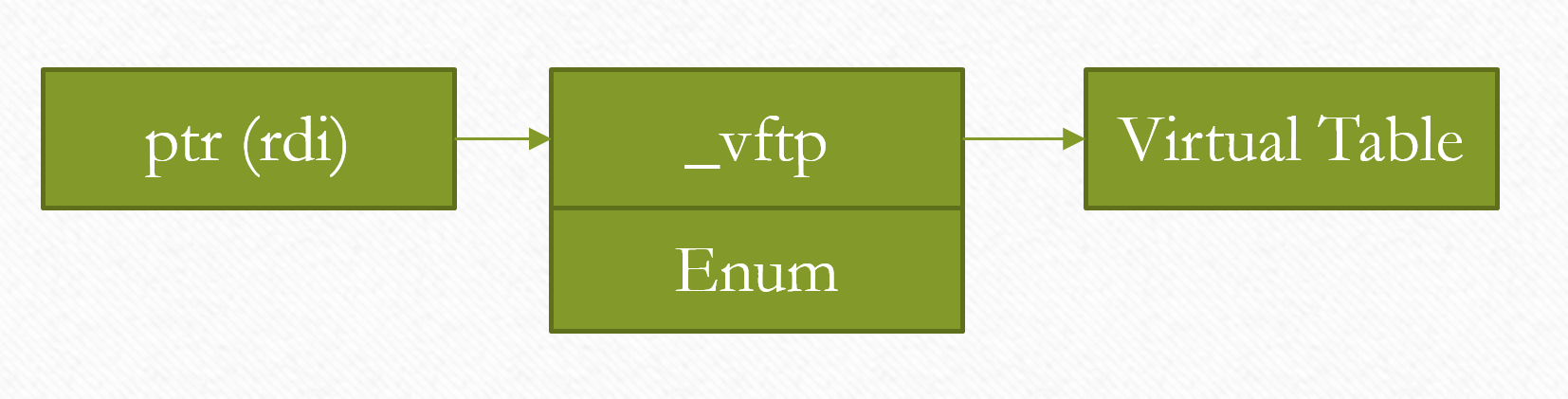
|  |  |
| --- | --- |
| Enum | Member Functions Compare |
| if (\*a->m\_typedata != entity\_lerp\_fast::type) {  mov 0x8(%rdi),%rax  cmp %r12,(%rax)  je 404417 <\_Z23FastUpdateExampleTimersv+0x4d7>  a->Update(t);  mov (%rdi),%rax  movss 0xc(%rsp),%xmm0  callq \*(%rax) | const entity& e = \*(&(\*a));  as\_normfun dnf = (as\_normfun)(e.\*mf);  mov (%rdi),%rax  mov (%rax),%rdx  if (snf != dnf) {  cmp $0x402c80,%rdx  je 403e0a <\_Z32MethodPointerUpdateExampleTimersv+0x4ca>  a->Update(t);  movss 0xc(%rsp),%xmm0  callq \*%rdx |

As one would expect the main difference is the enum is fetched from within the data inside the class while the member function gets data that is stored inside the virtual function table. See the 2 diagrams below.

**Enum structure diagram 1**



**Member Functions Compare Structure Enum structure diagram 2**



The extra level of indirection is a mixed blessing. It is one more pointer to chase but compared to chasing this pointer then calling for even the simplest of functions it comparing the function pointer itself to a value is cheaper.

Entity system are very important structure in games. One many games you will have up to 300 content people and only 30 programmers. The content people will make entities and components for these entities. The programmers will have little say in how many of any entity are used. They will see what type of prototype scene content people are trying to make and then try to update their game code to make these scenes faster as needed. But they should try and not change the engine itself or at least make sure the changes are small. The idea is to only make these type of changes where it was worth it to get the cycles back otherwise we would just stick to simple virtual functions.

For entities you can think of them as things that do stuff in the world and components as the tools entities can use. Entity could be very complicated like a car in a race game or a soldier in battle field or could be very simple like a tree. You might want to support lot and lots of simple objects like trees and they could be updated very quickly and uniformly with SIMD matrix and vector math. With large objects like soldier or race car these virtual function will not mater. The nodes in skeleton animation hierarchy in a soldier entity have a similar problems to the entity one I talk about mind you so a sports game that has more animating players then battle field might apply this tech there to more effect. The entity problem is just one easy to understand where we would use it.

In some cases we could sort the entity based on type and update them. This would give us good saving in these cases as we would get less branch misprediction however this is not always posable. Even in these cases comparing data and not calling a virtual function would be better.

A static typing systems could be writing in some cases but the entity hierarchy is a good example of a hard case. Since each main player is quite different soldier verses rocket ship each team will write their own entities so the whole system will be 100s of files over 20 development teams. So the changes cause integrations problems when we have to change the type system.

So the basic problem with type systems is you don’t know you need a better one until you see the need for optimization. You don’t know that you need optimization until you see you scene. Once you have your scene you written a lot of code the wrong way and you need to refactor it to make it faster to make the scene look good.

# III Issues with this system

One interesting problem is comdat folding that some linker do. Optimized code with comdat folding means simple function like the 2 below would become the same function in the linker and get the same address.

size\_t foo () { 0 return;}

void \*bar() { 0 return;}

You would have to be very careful. It does reduce the number of cases that this feature is useful for example a stub destructor would be the same address as all other stub destructor. Some way to work around this problem would be very helpful. I am not sure how we should work around this problem. Most ideas I have are a pain and compiler specific like adding some inline asm to access a pretend to access a function static var. I would still want this feature even if a perfect solution can’t be found but this would mean this feature could only be used by “experts”.