



CastGuard

Joe Bialek – Microsoft Offensive Research &
Security Engineering (MORSE)

Twitter: @JosephBialek

Problem Space



Killing Bugs vs. Killing Exploit Techniques

Mitigating exploit techniques has ambiguous long-term value.

Mitigations are typically far enough away from actual bug that bugs are still exploitable using different techniques.

Tradeoffs between performance, compatibility, and mitigation durability are becoming increasingly difficult.

Unclear how many more practical opportunities there are for building meaningful exploit mitigations.

Killing Bugs vs. Killing Exploit Techniques

Microsoft increasingly focused on eliminating vulnerability classes, removing attack surface, and sandboxing code.

Hyper-V vPCI component refactored from a C kernel component to C++ (w/ GSL) user component.

Microsoft investigation of Rust and other safer systems languages, and use of managed languages.

CLFS blocked from sandboxes, Redirection Guard, etc.

Path Forward for C/C++ Code

Four high-level bug classes responsible for majority of memory safety vulnerabilities.

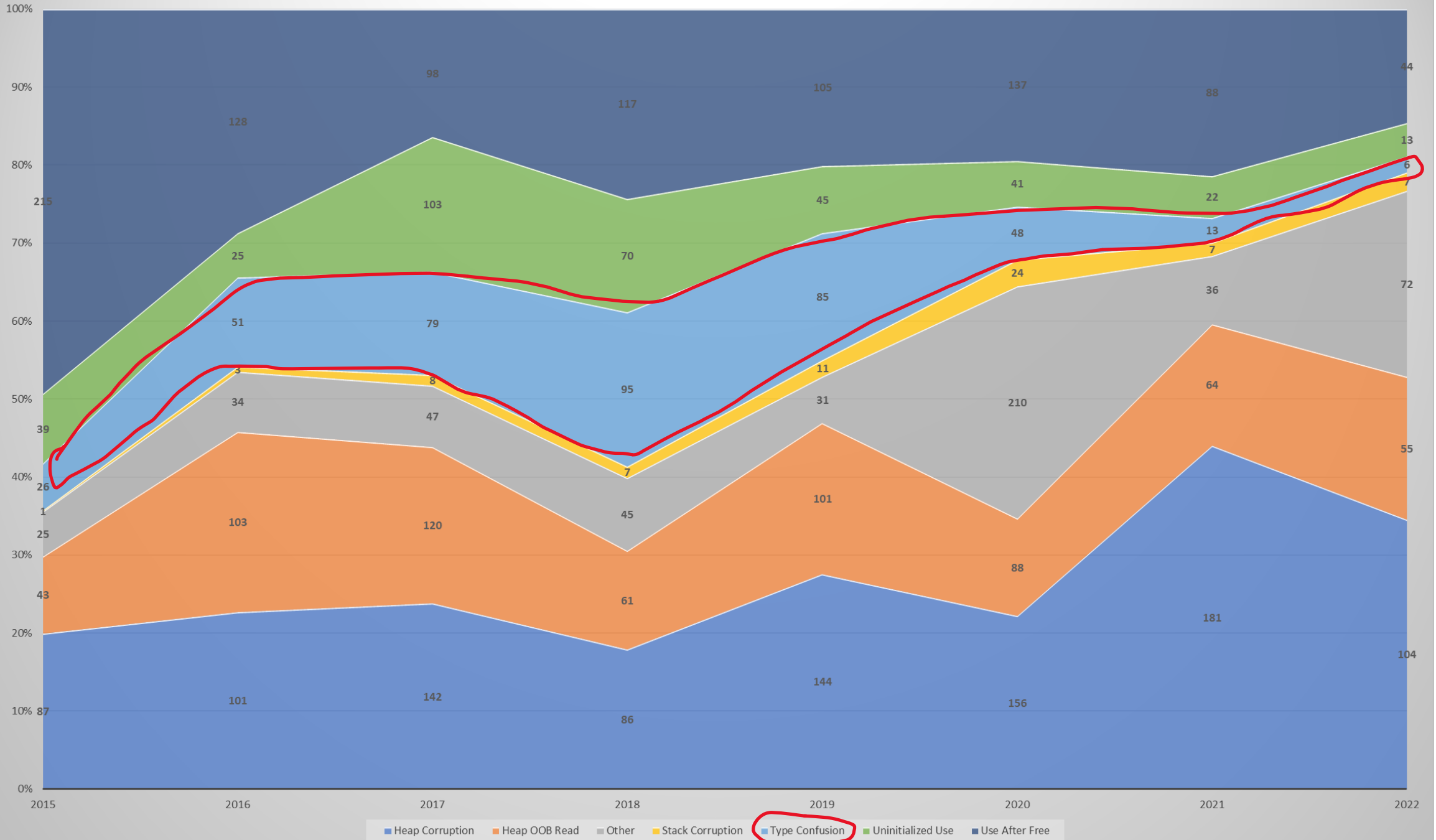
Buffer Overflow & Out-of-Bounds Accesses (i.e. attacker controls array index)

Uninitialized Memory

Type Confusion

Use-After-Free

Root Cause of Memory Safety CVEs by Patch Year



Path Forward for C/C++ Code

Four high-level bug classes responsible for majority of memory safety vulnerabilities.

Buffer overflow / out-of-bounds accesses (spatial safety)

Memory Tagging? / CHERI?

Uninitialized memory

InitAll / Zeroing Allocators

Type confusion

???

Use-after-free

Memory Tagging? / CHERI? / Application Specific Solutions (MemGC)

Not necessarily things Microsoft is committed to using, just illustrating solution space.

Type Confusion

Come in many flavors..

Illegal static downcast (down-casting to the wrong derived type in a class hierarchy).

Improper union use.

Illegal reinterpret_cast (i.e. cast an object of some type to totally different type).

Generic logic issues (i.e. using fields incorrectly).

Offer extremely powerful primitives to attackers and can often lead to breaking mitigations such as Memory Tagging.

Many forms of type confusion are not possible to generically solve 😞.

Illegal Static Downcasts

```
struct Animal {  
    virtual void WhoAmI() {cout << "Animal";} };
```

```
struct Dog : public Animal {  
    virtual void WhoAmI() {cout << "Dog";} };
```

```
struct Cat : public Animal {  
    virtual void WhoAmI() {cout << "Cat";} };
```

```
Animal* myAnimal = new Dog();  
static_cast<Cat>(myAnimal)->WhoAmI(); // Illegal down-cast
```

dynamic_cast?

Code creating the object and code casting object must enable Runtime Type Information (RTTI).

Makes it difficult to automatically convert existing static_cast's to dynamic_cast. Need to control all code to ensure RTTI settings are uniform. We cannot enforce this (3rd party DLL's, etc.).

RTTI causes binary size bloat (may be possible to optimize).

Windows.UI.Xaml.Controls.dll grows 86.5% from RTTI (no dynamic_cast).

dynamic_cast checks have overhead (may be possible to optimize).

Code executed in the success path of a dynamic_cast down-cast in a single-inheritance class hierarchy

```

00007ff7`56771472 c744242000000000 mov     dword ptr [rsp+20h], 0 ss:0000001f`baeff890=567bf330
00007ff7`5677147a 4c8d0ddff070600 lea     r9, [test3!MyChild1`RTTI Type Descriptor' (00007ff7`567d1c60)]
00007ff7`56771481 4c8d0590080600 lea     r8, [test3!MyBase`RTTI Type Descriptor' (00007ff7`567d1d18)]
00007ff7`56771488 33d2 xor     edx, edx
00007ff7`5677148a 488bcb mov     rcx, rbx
00007ff7`5677148d e81ec10400 call    test3!_RTDynamicCast (00007ff7`567bd5b0)

test3!_RTDynamicCast:
00007ff7`567bd5b0 48895c2410 mov     qword ptr [rsp+10h], rbx ss:0000001f`baeff878=0000000000000000
00007ff7`567bd5b5 4889742418 mov     qword ptr [rsp+18h], rsi
00007ff7`567bd5ba 57 push   rdi
00007ff7`567bd5bb 4154 push   r12
00007ff7`567bd5bd 4155 push   r13
00007ff7`567bd5bf 4156 push   r14
00007ff7`567bd5c1 4157 push   r15
00007ff7`567bd5c3 4883ec50 sub     rsp, 50h
00007ff7`567bd5c7 4d8bf9 mov     r15, r9
00007ff7`567bd5ca 4d8be0 mov     r12, r8
00007ff7`567bd5cd 4c63ea movsxd r13, edx
00007ff7`567bd5d0 488bf9 mov     rdi, rcx
00007ff7`567bd5d3 33db xor     ebx, ebx
00007ff7`567bd5d5 4885c9 test    rcx, rcx
00007ff7`567bd5d8 751c jne    test3!_RTDynamicCast+0x46 (00007ff7`567bd5f6) <-- take this jump

00007ff7`567bd5f6 488b01 mov     rax, qword ptr [rcx] ds:000001a4`f0889610=(test3!MyChild1::vftable' (00007ff7`567c9270))
00007ff7`567bd5f9 488b70f8 mov     rsi, qword ptr [rax-8]
00007ff7`567bd5fd 8b4604 mov     eax, dword ptr [rsi+4]
00007ff7`567bd600 4c8bf7 mov     r14, rdi
00007ff7`567bd603 4c2bf0 sub     r14, rax
00007ff7`567bd606 8b5608 mov     edx, dword ptr [rsi+8]
00007ff7`567bd609 482bca sub     rcx, rdx
00007ff7`567bd60c f7da neg     edx
00007ff7`567bd60e 1bc0 sbb     eax, eax
00007ff7`567bd610 2301 and     eax, dword ptr [rcx]
00007ff7`567bd612 4863c8 movsxd rcx, eax
00007ff7`567bd615 4c2bf1 sub     r14, rcx
00007ff7`567bd618 391e cmp     dword ptr [rsi], ebx <-- take this jump

00007ff7`567bd63a 48634614 movsxd rax, dword ptr [rsi+14h] ds:00007ff7`567c9b8c=00059b78
00007ff7`567bd63e 4c8bce mov     r9, rsi
00007ff7`567bd641 4c2bc8 sub     r9, rax
00007ff7`567bd644 48634610 movsxd rax, dword ptr [rsi+10h]
00007ff7`567bd648 428b4c0804 mov     ecx, dword ptr [rax+r9+4]
00007ff7`567bd64d f6c101 test    cl, 1
00007ff7`567bd650 7510 jne    test3!_RTDynamicCast+0xb2 (00007ff7`567bd662) <-- do not take this jump
00007ff7`567bd652 4d8bc7 mov     r8, r15
00007ff7`567bd655 498bd4 mov     rdx, r12
00007ff7`567bd658 488bce mov     rcx, rsi
00007ff7`567bd65b e8bcffff call    test3!FindSITargetTypeInstance (00007ff7`567bd11c) <-- call instruction

```

Stores: 12
 Loads: 30
 Jumps: 12
 Calls: 2

```

test3!FindSITargetTypeInstance:
00007ff7`567bd11c 488bc4 mov     rax, rsp
00007ff7`567bd11f 48895808 mov     qword ptr [rax+8], rbx
00007ff7`567bd123 48896810 mov     qword ptr [rax+10h], rbp
00007ff7`567bd127 48897018 mov     qword ptr [rax+18h], rsi
00007ff7`567bd12b 48897820 mov     qword ptr [rax+20h], rdi
00007ff7`567bd12f 4156 push   r14
00007ff7`567bd131 48634110 movsxd rax, dword ptr [rcx+10h]
00007ff7`567bd135 498bd8 mov     rbx, r8
00007ff7`567bd138 33c9 xor     ecx, ecx
00007ff7`567bd13a 4c8bf2 mov     r14, rdx
00007ff7`567bd13d 4e635c080c movsxd r11, dword ptr [rax+r9+0Ch]
00007ff7`567bd142 4e8b540808 mov     r10d, dword ptr [rax+r9+8]
00007ff7`567bd147 4d03d9 add     r11, r9
00007ff7`567bd14a 4585d2 test    r10d, r10d
00007ff7`567bd14d 741f je     test3!FindSITargetTypeInstance+0x52 (00007ff7`567bd16e) [br=0]
00007ff7`567bd14f 4d8bc3 mov     r8, r11
00007ff7`567bd152 496310 movsxd rdx, dword ptr [r8]
00007ff7`567bd155 4903d1 add     rdx, r9
00007ff7`567bd158 ffc1 inc     ecx
00007ff7`567bd15a 486302 movsxd rax, dword ptr [rdx]
00007ff7`567bd15d 4903c1 add     rax, r9
00007ff7`567bd160 483bc3 cmp     rax, rbx
00007ff7`567bd163 7467 je     test3!FindSITargetTypeInstance+0xb0 (00007ff7`567bd1cc) [br=1] <-- take this jump

00007ff7`567bd1cc 413bca cmp     ecx, r10d
00007ff7`567bd1cf 73e2 jae    test3!FindSITargetTypeInstance+0x97 (00007ff7`567bd1b3) [br=0]
00007ff7`567bd1d1 4d8d048b lea     r8, [r11+rcx*4]
00007ff7`567bd1d5 496300 movsxd rax, dword ptr [r8]
00007ff7`567bd1d8 42f644081404 test    byte ptr [rax+r9+14h], 4
00007ff7`567bd1de 75d3 jne    test3!FindSITargetTypeInstance+0x97 (00007ff7`567bd1b3) [br=0]
00007ff7`567bd1e0 4a630408 movsxd rax, dword ptr [rax+r9]
00007ff7`567bd1e4 4903c1 add     rax, r9
00007ff7`567bd1e7 493bc6 cmp     rax, r14
00007ff7`567bd1ea 7400 je     test3!FindSITargetTypeInstance+0xdd (00007ff7`567bd1f9) [br=1]

00007ff7`567bd1f9 488bc2 mov     rax, rdx
00007ff7`567bd1fc ebb7 jmp     test3!FindSITargetTypeInstance+0x99 (00007ff7`567bd1b5) <-- jump

00007ff7`567bd1b5 488b5c2410 mov     rbx, qword ptr [rsp+10h] ss:0000001f`baeff7f0=0000000000000000
00007ff7`567bd1ba 488b6c2418 mov     rbp, qword ptr [rsp+18h]
00007ff7`567bd1bf 488b742420 mov     rsi, qword ptr [rsp+20h]
00007ff7`567bd1c4 488b7c2428 mov     rdi, qword ptr [rsp+28h]
00007ff7`567bd1c9 415e pop     r14
00007ff7`567bd1cb c3 ret

; back to test3!_RTDynamicCast
00007ff7`567bd660 eb2d jmp     test3!_RTDynamicCast+0xdf (00007ff7`567bd68f) <-- jump

00007ff7`567bd68f 4c8bc8 mov     r9, rax
00007ff7`567bd692 4885c0 test    rax, rax
00007ff7`567bd695 7510 jne    test3!_RTDynamicCast+0xf7 (00007ff7`567bd6a7) <-- take this jump

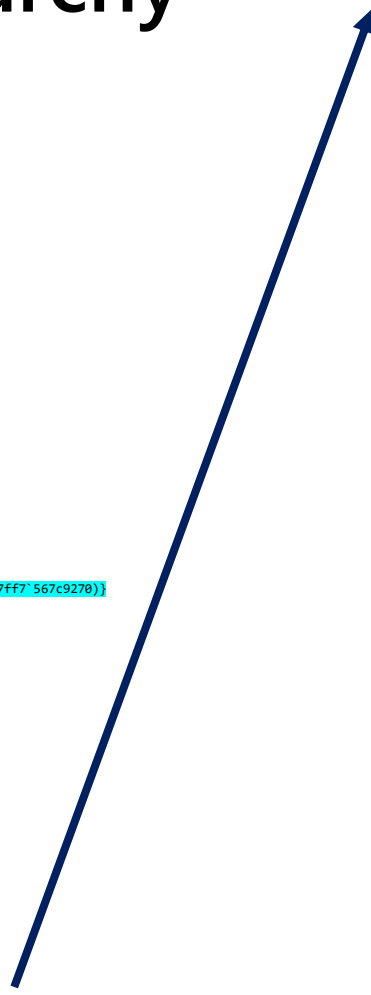
00007ff7`567bd6a7 4139590c cmp     dword ptr [r9+0Ch], ebx
00007ff7`567bd6ab 7c13 jl     test3!_RTDynamicCast+0x110 (00007ff7`567bd6c0) [br=1] <-- take this jump

00007ff7`567bd6c0 49634108 movsxd rax, dword ptr [r9+8] ds:00007ff7`567c9bd0=00000000
00007ff7`567bd6c4 4803c3 add     rax, rbx
00007ff7`567bd6c7 4903c6 add     rax, r14
00007ff7`567bd6ca e900ffff jmp     test3!_RTDynamicCast+0x2c (00007ff7`567bd5dc) <-- jump

00007ff7`567bd5dc 4c8d5c2450 lea     r11, [rsp+50h]
00007ff7`567bd5e1 498b5b38 mov     rbx, qword ptr [r11+38h]
00007ff7`567bd5e5 498b7340 mov     rsi, qword ptr [r11+40h]
00007ff7`567bd5e9 498be3 mov     rsp, r11
00007ff7`567bd5ec 415f pop     r15
00007ff7`567bd5ee 415e pop     r14
00007ff7`567bd5f0 415d pop     r13
00007ff7`567bd5f2 415c pop     r12
00007ff7`567bd5f4 5f pop     rdi
00007ff7`567bd5f5 c3 ret

; dynamic cast check over!

```



CastGuard

Inspired by Clang's [-fsanitize=cfi-derived-cast](#)



Concept



To protect against illegal downcast, object needs a type identifier that can be checked.

We cannot change object layout or we break the world.

Objects with a vtable already have an identifier, the vtable.

Automatically convert all `static_cast` on classes with vtables in to CastGuard protected casts.

Terminology

```
void Foo (Animal* animal) {  
    static_cast<Cat*>(animal);  
}
```

LHS Type: Left-hand side type being cast to (Cat*).

RHS Type: The statically declared right-hand side type (Animal*).

Underlying Type: The actual type of the RHS object (unknown at compile time).

Threat Model / Requirements

Code must be compiled using Link Time Code Generation (LTCG).

Code creating the object in the same LTCG module as code casting the object.

LHS type and RHS type have at least 1 vftable.

Object is valid (i.e. if RHS type is `Animal*`, it is a valid `Animal*`).

If some other component already illegally casted the object, we will not provide protection.

First-order memory safety vulnerability is the type confusion (i.e. attacker doesn't already have memory corruption).

Single Inheritance Example



What The Compiler Knows

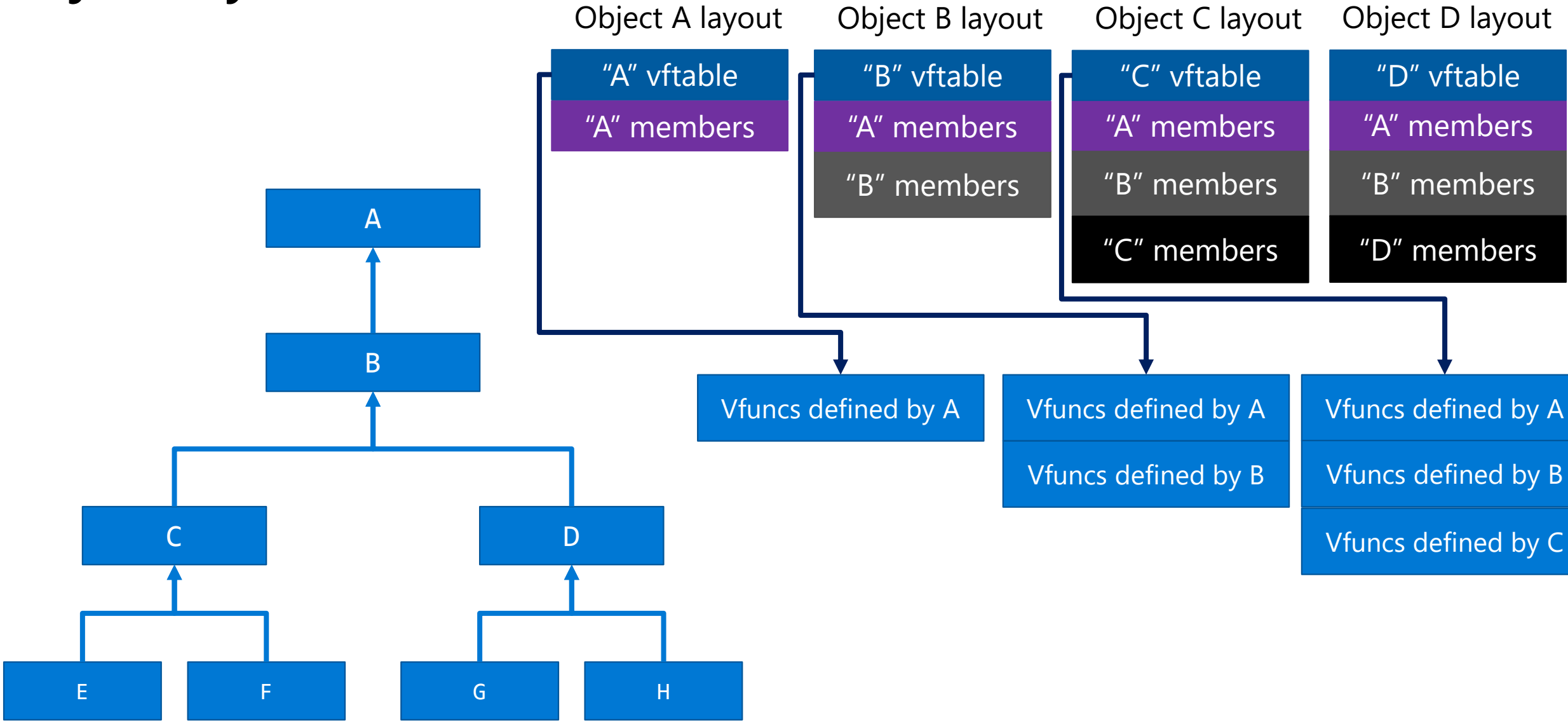
```
void Foo (Animal* animal) {  
    static_cast<Cat*>(animal);  
}
```

This is a static downcast from `Animal*` to `Cat*`.

The offset into `Cat*` and `Animal*` that the vtable pointer is located.

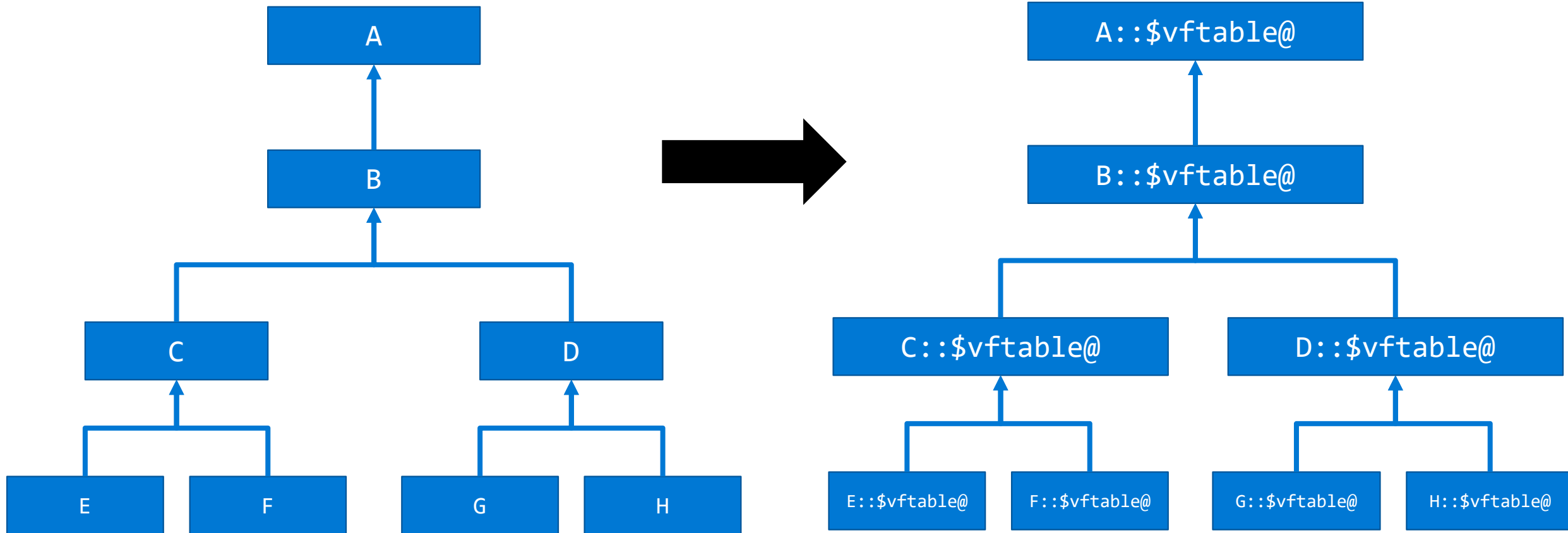
The location (RVA into binary) of the vtables for every type in this hierarchy (i.e. where the vtables are laid out in the binary).

Object Layout



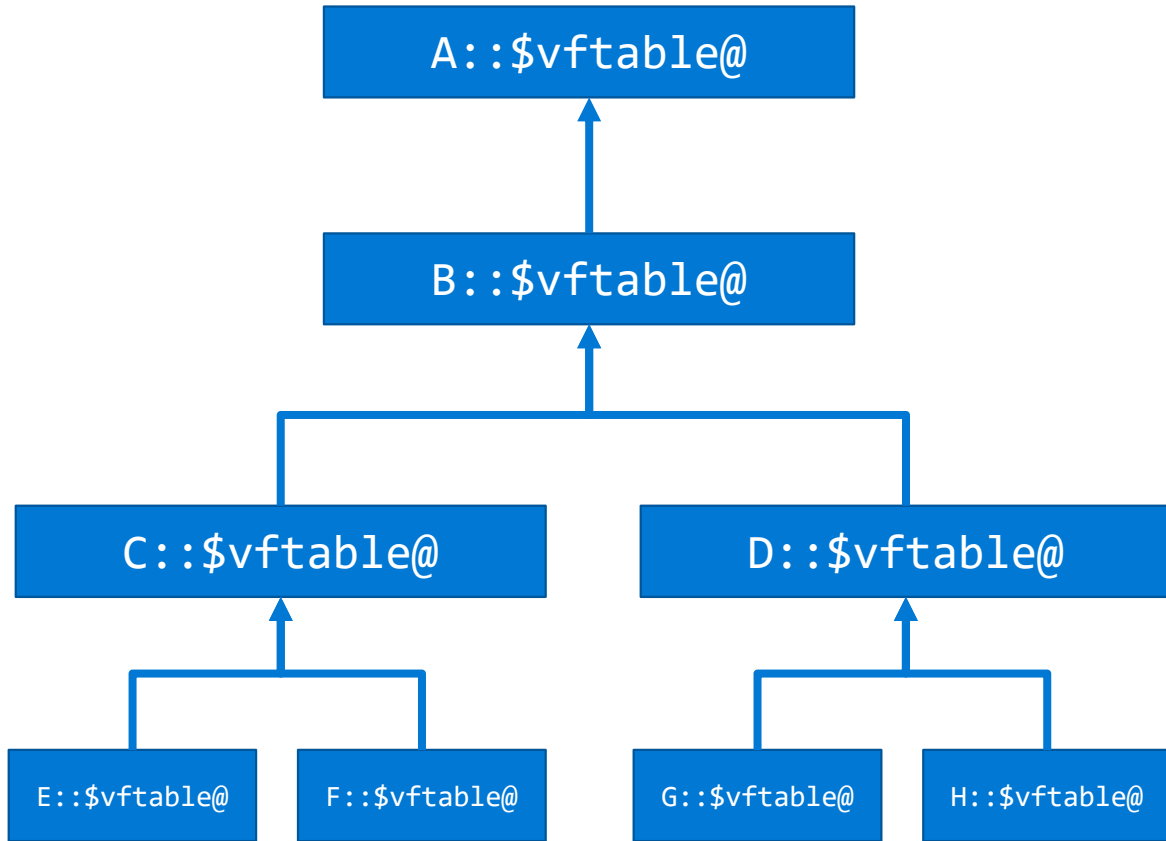
Derived types begin with their base types layout and append their own member variables after the base type.

The Vftable View of the World



It is helpful to think about class hierarchies in terms of the vftables as that is the unique identifier

Legal Underlying Vftables



LHS Type (Type Being Cast To)

	A::\$vtable@	B::\$vtable@	C::\$vtable@	E::\$vtable@	F::\$vtable@	D::\$vtable@	G::\$vtable@	H::\$vtable@
A	✓	✓	✓	✓	✓	✓	✓	✓
B		✓	✓	✓	✓	✓	✓	✓
C			✓	✓	✓			
D						✓	✓	✓
E				✓				
F					✓			
G							✓	
H								✓

This table shows the vftables are that legal for a pointer of some specific type to have

For example, if you cast to an "E" the only legal vtable would be E::\$vtable@

Naïve Check (To Understand Concepts)

User Code:

```
void MyFunction (B* b)
{
    static_cast<C*>(b);
}
```

Compiler Inserts:

```
// Ensure the vftable is one of the legal
// vftables, if not, fast-fail
```

```
if (b != NULL)
{
    if (b->vftable != C::$vftable@ &&
        b->vftable != E::$vftable@ &&
        b->vftable != F::$vftable@)
    {
        fast-fail
    }
}
```

static_cast on a NULL pointer is always allowed.

Naïve Check (To Understand Concepts)

```
void MyFunction (B* b)
{
    static_cast<C*>(b);
}
```

Compiler Inserts CodeGen:

```
// Ensure the vftable is one of the legal
// vftables, if not, fast-fail
if (b != NULL)
{
    if (b->vftable != C::$vftable@ &&
        b->vftable != E::$vftable@ &&
        b->vftable != F::$vftable@)
    {
        fast-fail
    }
}
```

This check would scale terribly with large amounts of vftables

Optimization Step 1: Lay Out Vftables Together in Binary

To make the example simple, assume 64-bit architecture, each vftable has a single virtual function, and no RTTI information. Total size per vftable is 8 bytes.

We'll talk about the global variables `__CastGuardVftableStart` and `__CastGuardVftableEnd` later.

CastGuard Vftable Region

Offset	Name
0x00	<code>__CastGuardVftableStart</code>
0x08	<code>A::\$vftable@</code>
0x10	<code>B::\$vftable@</code>
0x18	<code>C::\$vftable@</code>
0x20	<code>D::\$vftable@</code>
0x28	<code>E::\$vftable@</code>
0x30	<code>F::\$vftable@</code>
0x38	<code>G::\$vftable@</code>
0x40	<code>H::\$vftable@</code>
0x48	<code>__CastGuardVftableEnd</code>

Vftables CastGuard cares about are laid out in their own contiguous region.

Optimization Step 2: Create Bitmaps

Offset	Name
0x00	__CastGuardVftableStart
0x08	A::\$vftable@
0x10	B::\$vftable@
0x18	C::\$vftable@
0x20	D::\$vftable@
0x28	E::\$vftable@
0x30	F::\$vftable@
0x38	G::\$vftable@
0x40	H::\$vftable@
0x48	__CastGuardVftableEnd

Legal Underlying Vftables

LHS Type (Type Being Cast To)

	A::\$vftable@	B::\$vftable@	C::\$vftable@	E::\$vftable@	F::\$vftable@	D::\$vftable@	G::\$vftable@	H::\$vftable@
A	✓	✓	✓	✓	✓	✓	✓	✓
B		✓	✓	✓	✓	✓	✓	✓
C			✓	✓	✓			
D						✓	✓	✓
E				✓				
F					✓			
G							✓	
H								✓

Create a bitmap per LHS Type being cast to which indicates which vftables are legal for that cast

Optimization Step 2: Create Bitmaps

Offset	Name
0x00	__CastGuardVftableStart
0x08	A::\$vftable@
0x10	B::\$vftable@
0x18	C::\$vftable@
0x20	D::\$vftable@
0x28	E::\$vftable@
0x30	F::\$vftable@
0x38	G::\$vftable@
0x40	H::\$vftable@
0x48	__CastGuardVftableEnd

How to Create Bitmap:

Each type being downcast to gets its own bitmap (i.e. B, C, D, etc.).

For each bitmap:

1. Choose a "base vftable". This is the vftable you will compare the underlying types vftable against. It should be the first vftable (lowest RVA) in the binary that is legal for the cast.
2. Compute the offset between this vftable and all other vftables that are legal for the cast.
3. Each legal vftable is "1" in the bitmap. Illegal vftables are "0".

Optimization Step 2: Create Bitmaps

For cast to "C", "base vftable" is C::\$vftable@

Offset	Name
0x00	__CastGuardVftableStart
0x08	A::\$vftable@
0x10	B::\$vftable@
0x18	C::\$vftable@
0x20	D::\$vftable@
0x28	E::\$vftable@
0x30	F::\$vftable@
0x38	G::\$vftable@
0x40	H::\$vftable@
0x48	__CastGuardVftableEnd

Offset from C::\$vftable@	0x0	0x8	0x10	0x18
C_Bitmap	1	0	1	1

How To Use Bitmap:

`delta = Object->Vftable - C::$vftable@`

`ordinal = delta ROR 3 // shift out low 3 bits`

`C_Bitmap[ordinal] == 1 if cast allowed`

Optimization Step 2: Create Bitmaps

Offset	Name
0x00	__CastGuardVftableStart
0x08	A::\$vftable@
0x10	B::\$vftable@
0x18	C::\$vftable@
0x20	D::\$vftable@
0x28	E::\$vftable@
0x30	F::\$vftable@
0x38	G::\$vftable@
0x40	H::\$vftable@
0x48	__CastGuardVftableEnd

Offset of underlying type vftable from "base vftable"

	0x0	0x8	0x10	0x18	0x20	0x28	0x30	0x38
B_Bitmap	1	1	1	1	1	1	1	
C_Bitmap	1	0	1	1				
D_Bitmap	1	0	0	1	1			
E_Bitmap	1							
F_Bitmap	1							
G_Bitmap	1							
H_Bitmap	1							

Bitmap for LHS Type

Only create bitmap for types that are down-cast to. Minimizes binary size.

Bitmap alignment can change to reduce binary size. This example uses 8-byte alignment (the minimum on 64-bit) but we may increase the alignment of vftables to reduce the size of the bitmap.

Better Check

```
void MyFunction (B* b)
{
    static_cast<C*>(b);
}
```

Compiler Inserts CodeGen:

```
if (b != null) {

    // read the vftable from the object
    uint64 ptr = b->vftable;

    // get offset from the first valid vftable for this cast
    uint64 delta = ptr - &C::$vftable@

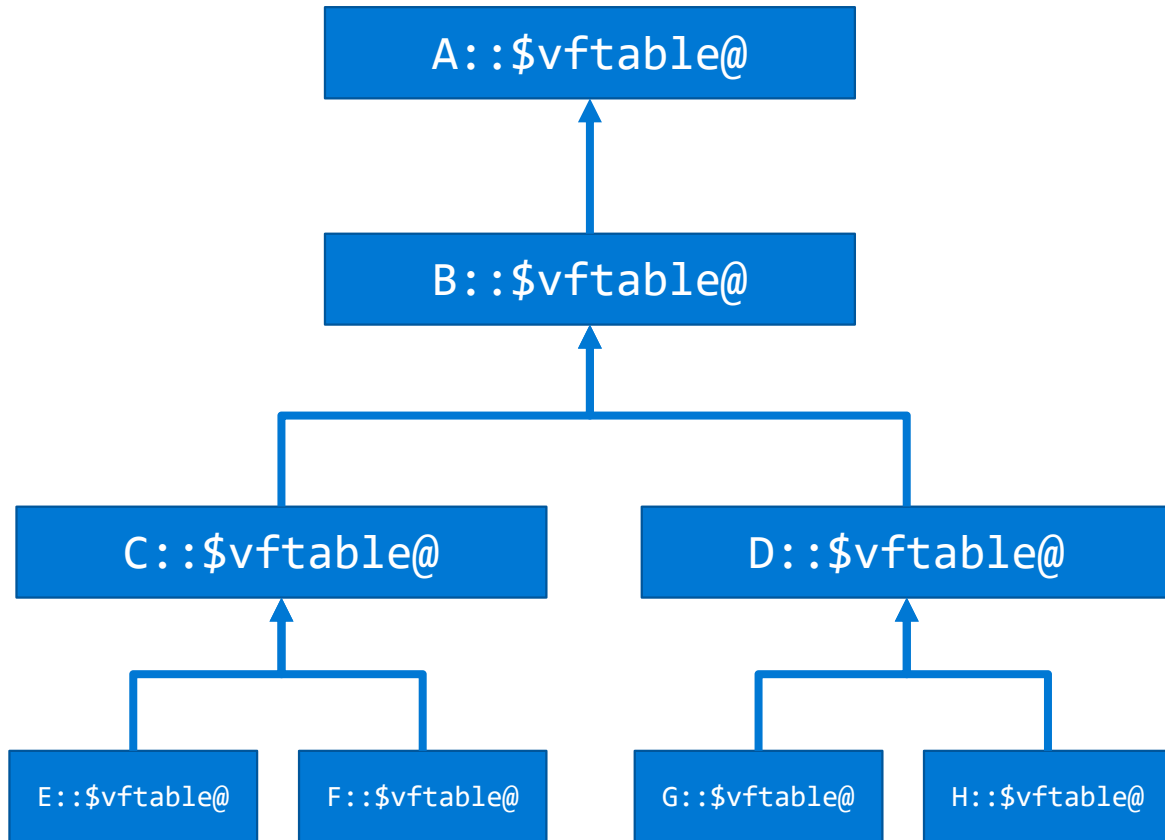
    // vftables are 8 byte aligned
    // if any low 3 bits are set, ROR will shift them to high bits
    uint64 ordinal = delta ROR 3;

    // test the bitmap to see if this is valid
    if (ordinal >= sizeof_in_bits(C_Bitmap))
        !bittest(C_Bitmap, ordinal)
    {
        fast-fail
    }
}
```

More Optimization

- With a few realizations, we can do much better than this.
- Bitmaps are not ideal because:
 - It takes a memory load to consult them.
 - They take up space in the binary.

Order the Vftables Depth First



CastGuard Vftable Region

Offset	Name
0x00	__CastGuardVftableStart
0x08	A::\$vtable@
0x10	B::\$vtable@
0x18	C::\$vtable@
0x20	E::\$vtable@
0x28	F::\$vtable@
0x30	D::\$vtable@
0x38	G::\$vtable@
0x40	H::\$vtable@
0x48	__CastGuardVftableEnd

Create Bitmaps

Offset	Name
0x00	__CastGuardVftableStart
0x08	A::\$vftable@
0x10	B::\$vftable@
0x18	C::\$vftable@
0x20	E::\$vftable@
0x28	F::\$vftable@
0x30	D::\$vftable@
0x38	G::\$vftable@
0x40	H::\$vftable@
0x48	__CastGuardVftableEnd

Offset of underlying type vftable from LHS Type vftable

	0x0	0x8	0x10	0x18	0x20	0x28	0x30	0x38
B_Bitmap	1	1	1	1	1	1	1	
C_Bitmap	1	1	1					
D_Bitmap	1	1	1					
E_Bitmap	1							
F_Bitmap	1							
G_Bitmap	1							
H_Bitmap	1							


Bitmap for LHS Type

Property: When ordered with a DFS, legal vftables are always laid out contiguously (thus you never see 0's in the bitmap)

Range Check

Offset	Name
0x00	__CastGuardVftableStart
0x08	A::\$vftable@
0x10	B::\$vftable@
0x18	C::\$vftable@
0x20	E::\$vftable@
0x28	F::\$vftable@
0x30	D::\$vftable@
0x38	G::\$vftable@
0x40	H::\$vftable@
0x48	__CastGuardVftableEnd

- If bitmap is all ones, no need to check the bitmap. As long as the ordinal is in bounds of the bitmap you succeed.
- Taking it further: Rather than shifting the pointer to calculate the ordinal, just do a range check.



If vftable base address is within 0x10 bytes of C::\$vftable&, this object is a valid "C"

Range Check

```
void MyFunction (B* b)
{
    static_cast<C*>(b);
}
```

Compiler Inserts CodeGen:

```
if (b != null) {

    // read the vftable from the object
    uint64 ptr = b->vftable;

    // get offset from the first valid vftable for this cast

    uint64 offset = ptr - &C::'vftable'

    // can be C, E, or F
    // vftable expected to be 0x0, 0x8, or 0x10 bytes
    // offset from C::$vftable@

    if (offset > 0x10) {
        fast-fail
    }
}
```

Concerns?

What if the vftable is offset 0x9? That would pass the check but is illegal!

Not a concern due to threat model, there is no way a legitimate object could be created with that vftable pointer. We are assuming the first order vulnerability is this `static_cast` so the object must be well formed.

Compiler Inserts CodeGen:

```
if (b != null) {  
  
    // read the vftable from the object  
    uint64 ptr = b->vftable;  
  
    // get offset from the first valid vftable for this cast  
  
    uint64 offset = ptr - &C::'vftable'  
  
    // can be C, E, or F  
    // vftable expected to be 0x0, 0x8, or 0x10 bytes  
    // offset from C::$vftable@  
  
    if (offset > 0x10) {  
        fast-fail  
    }  
}
```

Compatibility

What if the object was created in a different DLL?

The LTCG compiler pass will not know about these vftables. The cast might be legitimate but because the vftable comes from a different DLL it isn't laid out where CastGuard expects.

What if the object was created in a static library?

Mostly similar concern, with caveats. See appendix for more details.

Modified Check for Compatibility

```
...
if (ptr != null) {

    // get offset from the first valid vftable for
    // this cast

    uint64 offset = ptr - &C::$vftable@

    // can be C, E, or F
    // vftable expected to be 0x0, 0x8, or 0x10 bytes
    // offset from C::$vftable@

    if (offset > 0x10 &&
        ptr > __CastGuardVftableStart &&
        ptr < __CastGuardVftableEnd) {
        fast-fail
    }
}
```

Only fast-fail if the underlying vftable is being tracked by CastGuard, otherwise "fail open" for compatibility

AMD64 Assembly

```
; rcx == The right-hand side object pointer.  
; First do the nullptr check. This could be optimized away but is not today.  
; N.B. If the static_cast has to adjust the pointer base, this nullptr check  
; already exists.
```

```
4885c9          test    rcx, rcx  
7416           je     codegentest!DoCast+0x26
```

```
; Next load the RHS vftable and the comparison vftable.
```

```
488b11          mov    rdx, qword ptr [rcx]  
4c8d05ce8f0500 lea   r8, [codegentest!C::~`vftable']
```

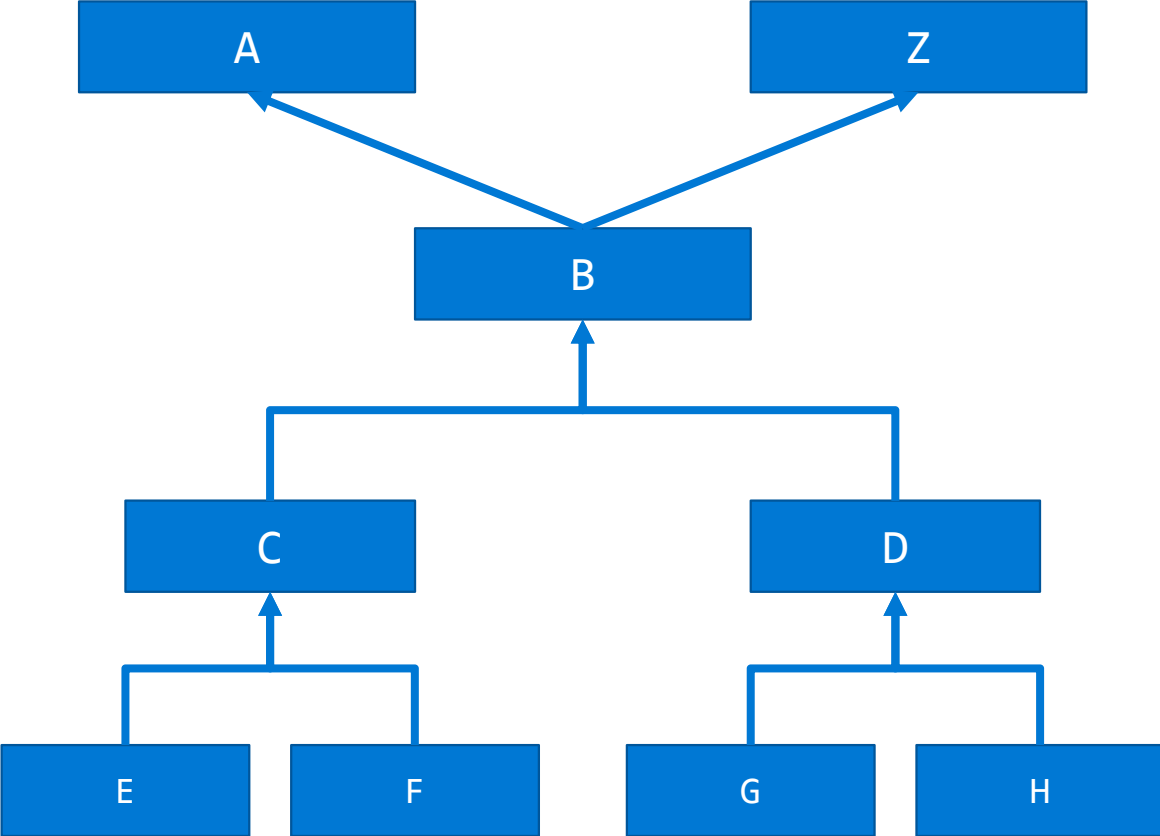
```
; Now do the range check. Jump to the AppCompat check if the range check fails.
```

```
492bc0          sub    rdx, r8  
4883f820        cmp    rdx, 20h  
7715           ja     codegentest!DoCast+0x3b      ; Jump to app-compat check
```

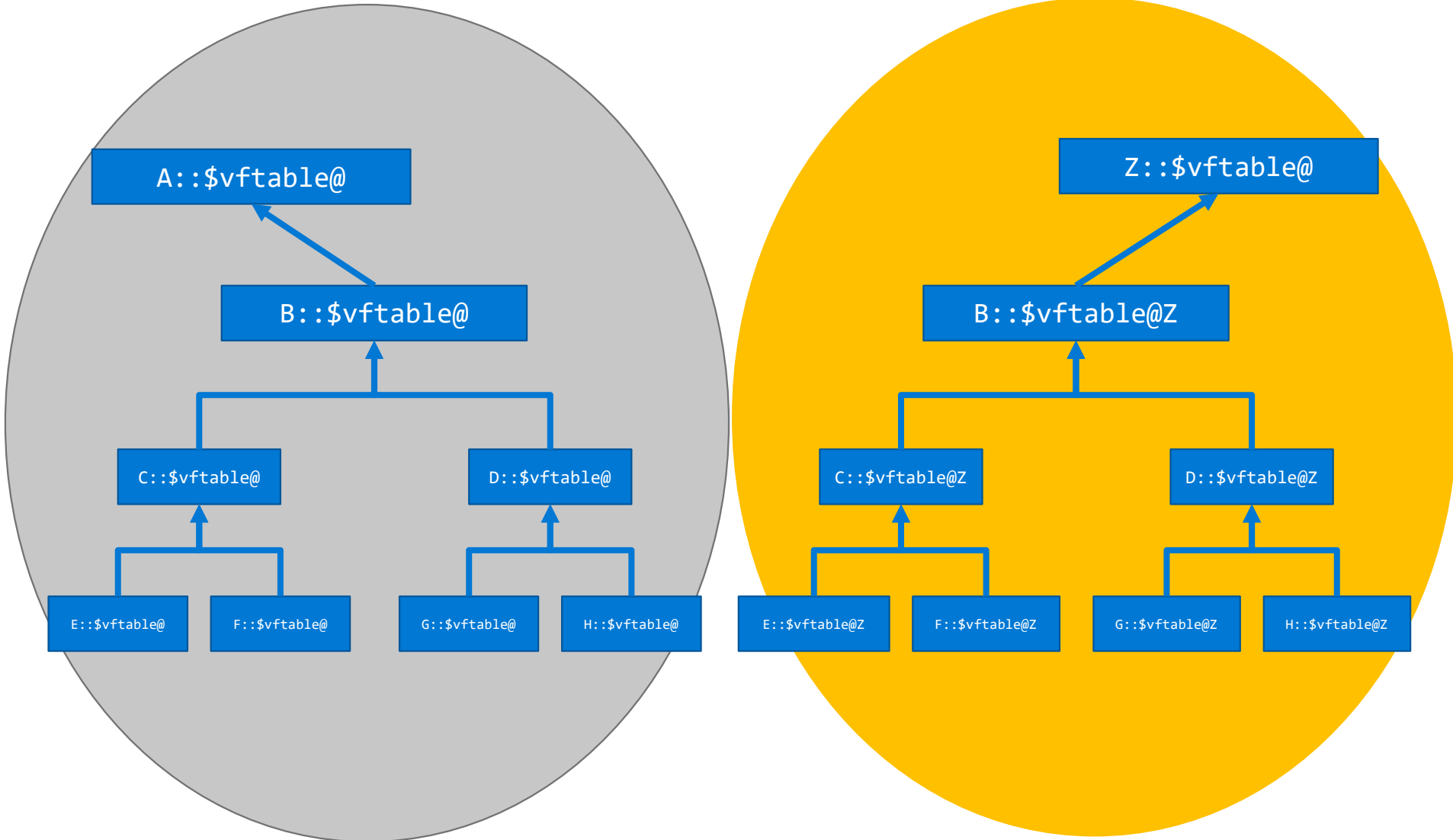
Multiple Inheritance



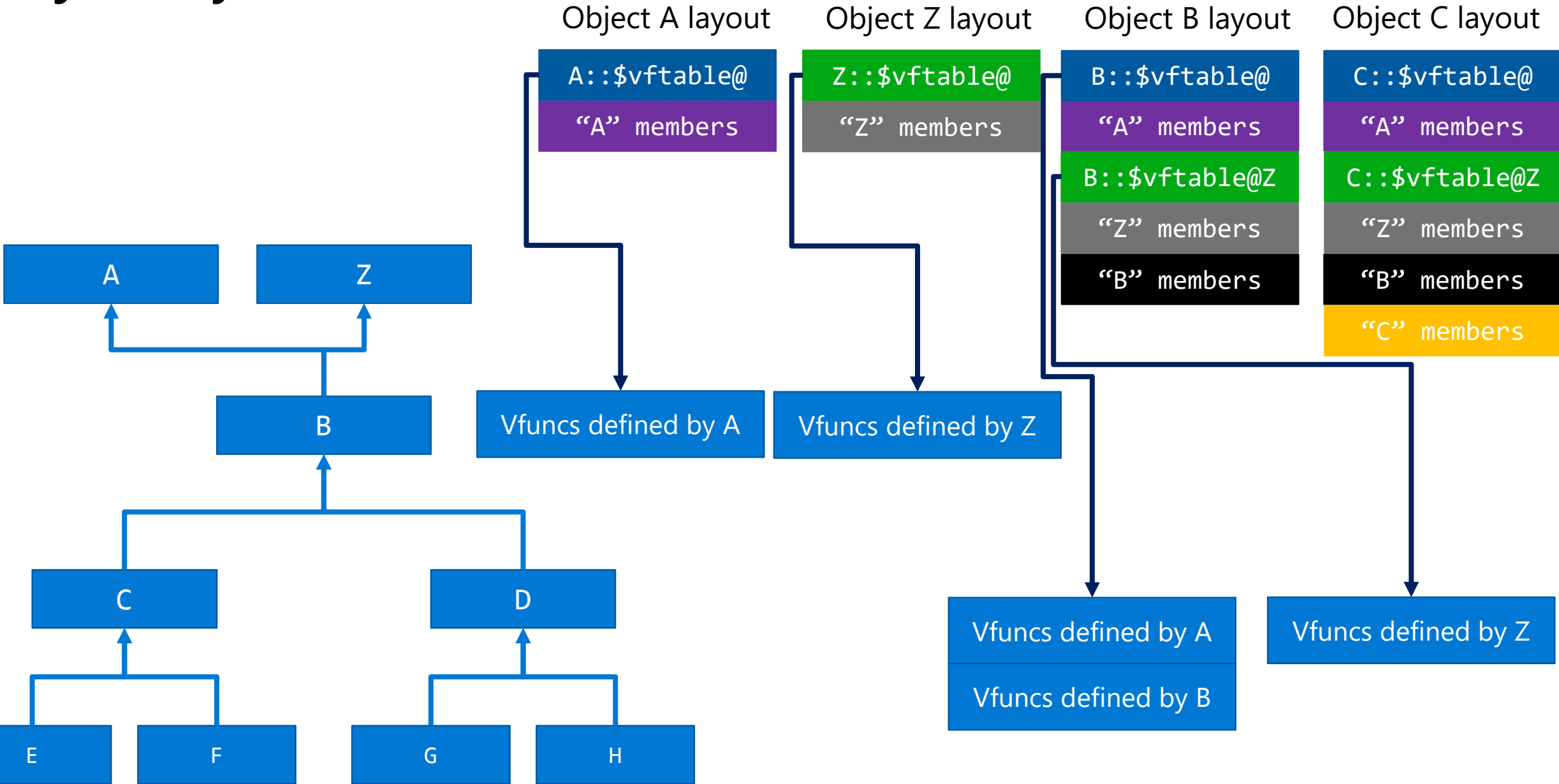
Multiple Inheritance Example



Vftable View



Object Layout



Which Vftable to Use

Depends on what the current RHS type is.

If RHS == "A", need to use vftable that "A" introduced.

If RHS == "Z", need to use vftable that "Z" introduced.

Otherwise, can use either. Prefer the vftable that is closest to the base address of the object to reduce code size.

If vftable is at offset 0 in object, the "this" pointer doesn't need to be adjusted.

Notes

Once you realize there are multiple vftable hierarchies, multiple inheritance becomes identical to single inheritance.

Choose the vftable hierarchy to do checks against based on the RHS type.

Lay out vftable hierarchy using depth-first layout.

Do a simple range check on the vftable.

Virtual Base Hierarchies



Virtual Base

See appendix for full information (not enough time).

A nasty and rarely used C++ feature that allows “more efficiently” doing multiple inheritance when both parent classes inherit from the same base class.

Can make range checks impossible for a vftable hierarchy, need to use bitmap checks.

Interesting Notes



Misc.

Identical Comdat Folding (ICF) must be disabled for vftables in the "CastGuard region".

ICF will eliminate duplicate copies of data (i.e. vftables that are identical) but CastGuard requires all vftables are unique since they are used as identifiers.

OptRef (remove unreferenced symbols) linker optimization also disabled for CastGuard vftables.

If the linker deletes an unreferenced vftable in the CastGuard region, it changes the layout of the region but we already generated code based on the expected layout.

Luckily LTCG does a pretty good job up-front at determining if a symbol is unreferenced and we won't lay out the symbol in the first place.

Curiously Recurring Template Pattern (CRTP)




```
struct A
{
    A::A(){}

    virtual void Entry()
    {
        return;
    }
};
```

```
template <class T>
struct B : A
{
    virtual void Entry()
    {
        static_cast<T*>(this)->WhoAmI();
    }

    void DoStuff()
    {
        DoOtherStuff(static_cast<T*>(this));
    }
};
```

An object of type "B" is never created.
Only derived types are created.

Derived type specifies itself as a
templated parameter when inheriting
from "B" class.

```
struct C : B<C>
{
    void WhoAmI()
    {
        PrintWhoIAm();
    }

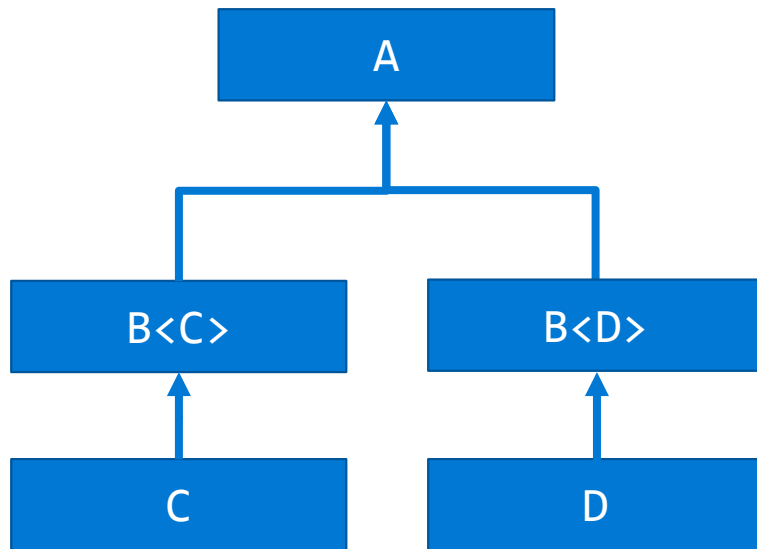
    static void PrintWhoIAm()
    {
        printf("C");
    }
};
```

```
struct D : B<D>
{
    void WhoAmI()
    {
        PrintWhoIAm();
    }

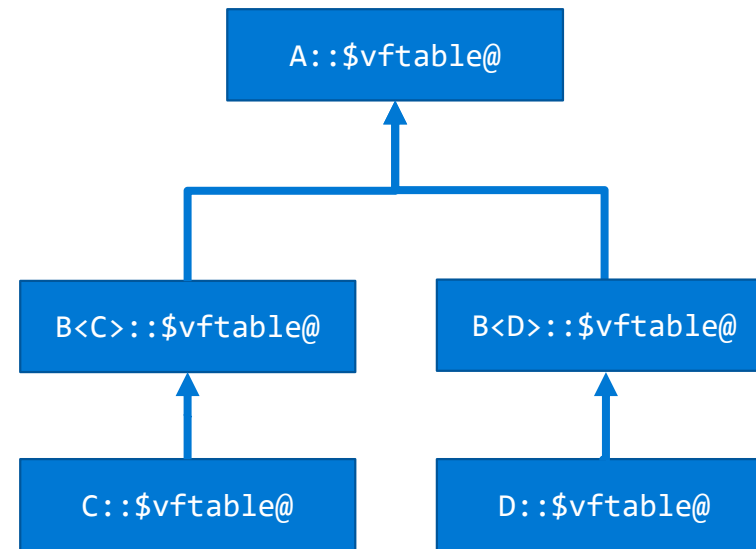
    static void PrintWhoIAm()
    {
        printf("D");
    }
};
```

Because "B" is specialized based on the type inheriting from it, any given "B" type (such as B<C>) only has a single derived type.

Class Hierarchy



Vftable Hierarchy



Each specialized version of "B" is its own unique class and has a unique vftable.

```
struct A
{
    A::A(){}

    virtual void Entry()
    {
        return;
    }
};
```

```
template <class T>
struct B : A
{
    virtual void Entry()
    {
        static_cast<T*>(this)->WhoAmI();
    }

    void DoStuff()
    {
        DoOtherStuff(static_cast<T*>(this));
    }
};
```

"DoStuff" for B<C> and B<D> are identical. Linker will de-duplicate them.

Vftable for C & B<C>, D & B<D> are identical. Linker will de-duplicate.

```
struct C : B<C>
{
    void WhoAmI()
    {
        PrintWhoIAM();
    }

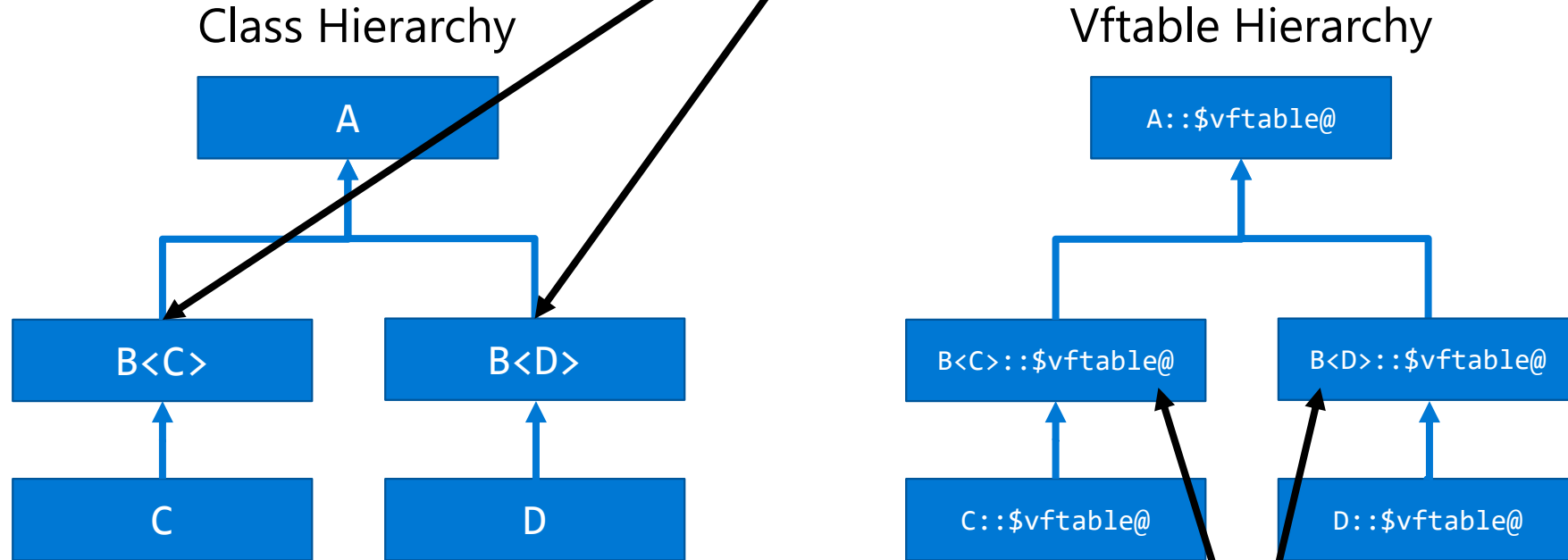
    static void PrintWhoIAM()
    {
        printf("C");
    }
};
```

```
struct D : B<D>
{
    void WhoAmI()
    {
        PrintWhoIAM();
    }

    static void PrintWhoIAM()
    {
        printf("D");
    }
};
```

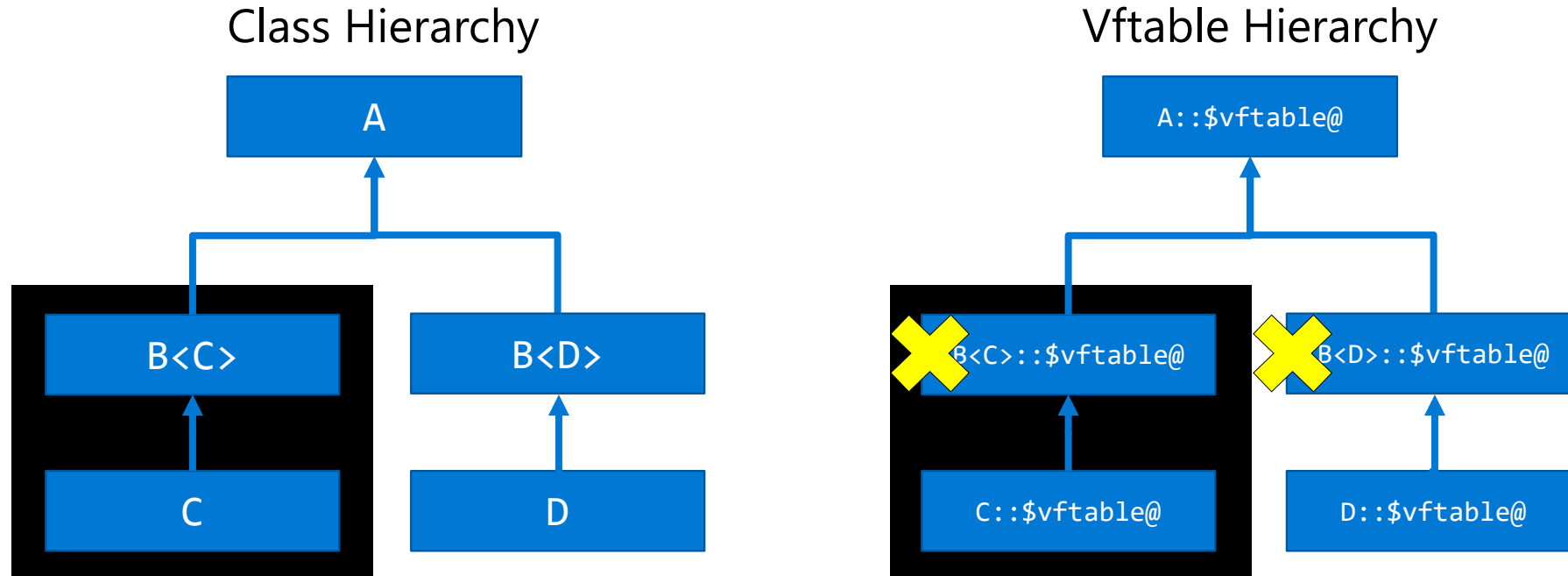
CastGuard checks make "DoStuff" implementations different (type specific cast checks) and reference vftables so they can no longer be de-duplicated. Some WinRT binaries got ~20% bigger.

These types are never created, so a valid object will NEVER have their vtables.



No reason to include these vtables in checks since an object can never have them.

```
void Demo(B<C>* MyPtr) {  
    static_cast<C*>(MyPtr);  
}
```



A "B<C>*" either points to a "B<C>" or a "C". We know "B<C>" was never created, must be "C".

The only vftable a "B<C>*" pointer could have is "C::\$vftable@" which is legal when casting to "C", no point in doing a cast check.

Optimizing for CRTP

Optimization can also help non-CRTP related casts.

We were able to statically prove away EVERY cast check in Windows.UI.Xaml.Controls.dll, going from 20% binary size regression to 0%.

Similar CRTP optimization issues exist for other technologies.

XFG (Extended Flow Guard) – Caused a 43% binary size regression on “Windows.UI.Xaml.Controls.dll” due to fine-grain indirect call signature checks. We got this fixed by making function signature ignore template specializations.

Clang CFI – Breaks CRTP code sharing due to fine-grain indirect call signature checks and cast checks (when derived-cast checks enabled). Likely a similar regression to CastGuard and XFG.

Performance



Near-zero runtime overhead

- Spec 2006 showed no regression.
- No overhead detected in Windows components.

Binary size impact under 1%

- Components without downcasting have no overhead.
- CRTP optimization can statically prove safety of many casts.

Future Possibilities



Strict Mode

- Mark hierarchies as “strict” indicating they should never have an app compat check, all failures are fatal.
- Could force full bitmap checks (defend against type confusions caused by memory corruption).

Acceleration for dynamic_cast

- Dynamic_cast hot path uses CastGuard style check, only does the full check in the app compat check path.

Just ideas – nothing committed.

Conclusion



It is possible to provide performant cast checks to prevent certain types of type confusion. May even be possible to use CastGuard tech in the hot-path of `dynamic_cast`.

CastGuard is flighting in Hyper-V code in Windows Insider Preview builds.

Additional Windows components will use CastGuard in the future.

Acknowledgements



CastGuard would not be possible without:

- Inspiration from Clang/LLVM `-fsanitize=cfi-derived-cast`.
- Many folks across Windows, Visual Studio, and MSRC.

Appendix



Virtual Base Inheritance



Overview

Inheritance works efficiently because offsets can be computed statically.

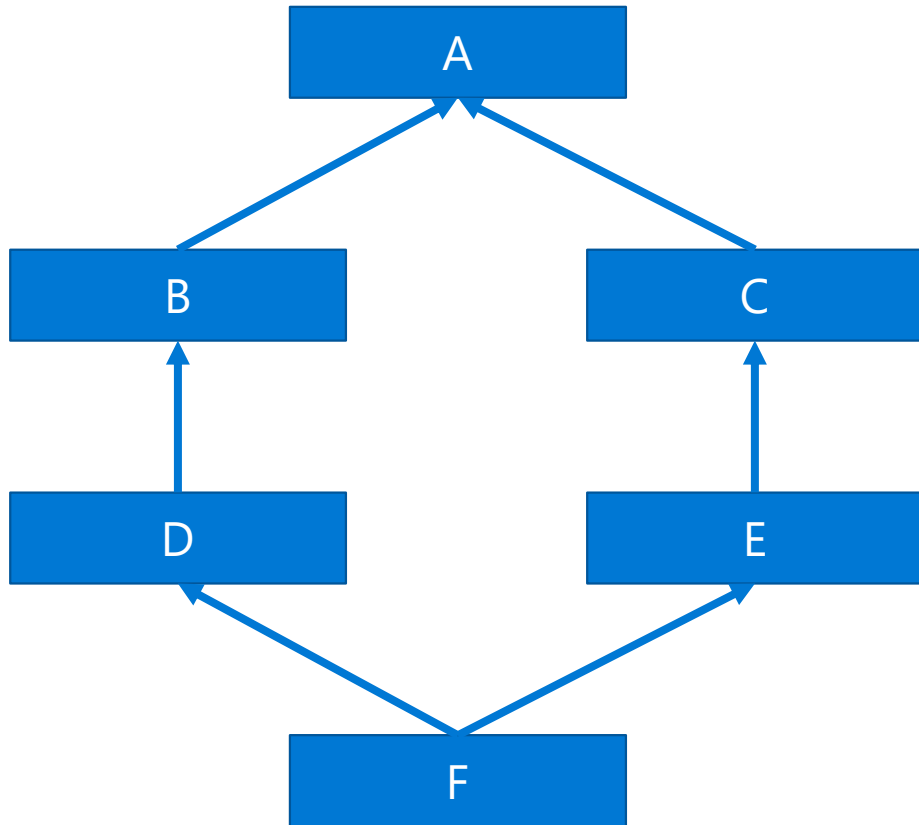
A pointer to an object can be upcast or downcast using simple pointer arithmetic based on where the type being casted to is “laid out” in the object’s memory.

The same is generally true for multiple inheritance.

Object layout is known at compile time and any `static_cast` can be statically computed at compile time.

Things get complicated when the same base class is inherited from multiple times (i.e. diamond pattern).

Non-Virtual Base Diamond Pattern



Object "F" layout:

```
0x0  vtable of A
0x8  <members of A>
0x?  <members of B>
0x?  <members of D>
0x?  vtable of A
0x?  <members of A>
0x?  <members of C>
0x?  <members of E>
0x?  <members of F>
```

The object "A" is inherited from twice and so there are two copies of it in "F". This may not be desirable.

Note: Cast checks for this pattern are identical to single inheritance / multiple inheritance

Diamond Pattern Issues

An object of type "F" contains two copies of the type "A".

"A" is inherited from by both "B" and "C". Which means "B" and "C" have their own copy of "A".

C++ does not automatically determine "you are using multiple inheritance and inheriting from the same base class twice so I will de-duplicate the object". Sometimes having multiple copies of the same base object is actually desired.

Can lead to bizarre behavior.

You cannot directly `static_cast` a pointer of type "A" to a pointer of type "F". "F" has two copies of "A" so you need to first `static_cast` to either "B" or "C" and then cast to "F" so the compiler knows which copy of "A" you are trying to access.

Virtual Base Inheritance Offers a Solution

Allows the compiler to de-duplicate the base class "A".

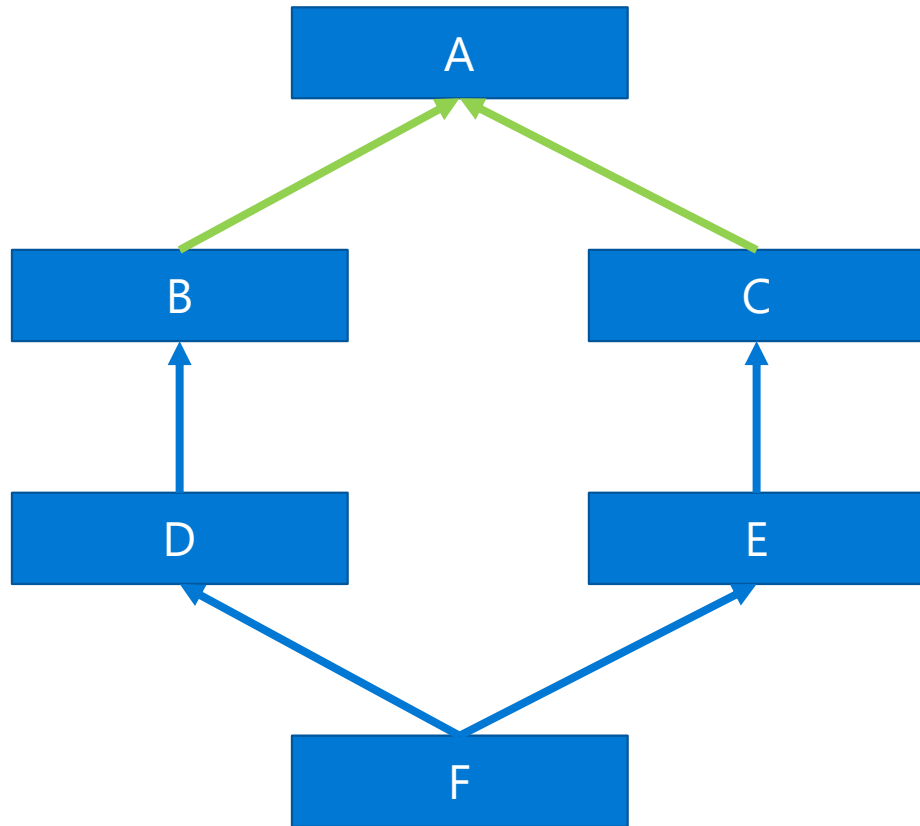
Anything that inherits from "A" cannot trivially cast to it since the location of "A" depends on the layout of the underlying type that was created.

i.e. an object of type "B" and type "F" may have "A" laid out at different positions.

A virtual base table is created in each object that contains the offset of "A" from the current "this" pointer.

Casting to "A" requires looking up how to adjust the current "this" pointer by reading from the virtual base table.

Virtual Base Diamond Pattern



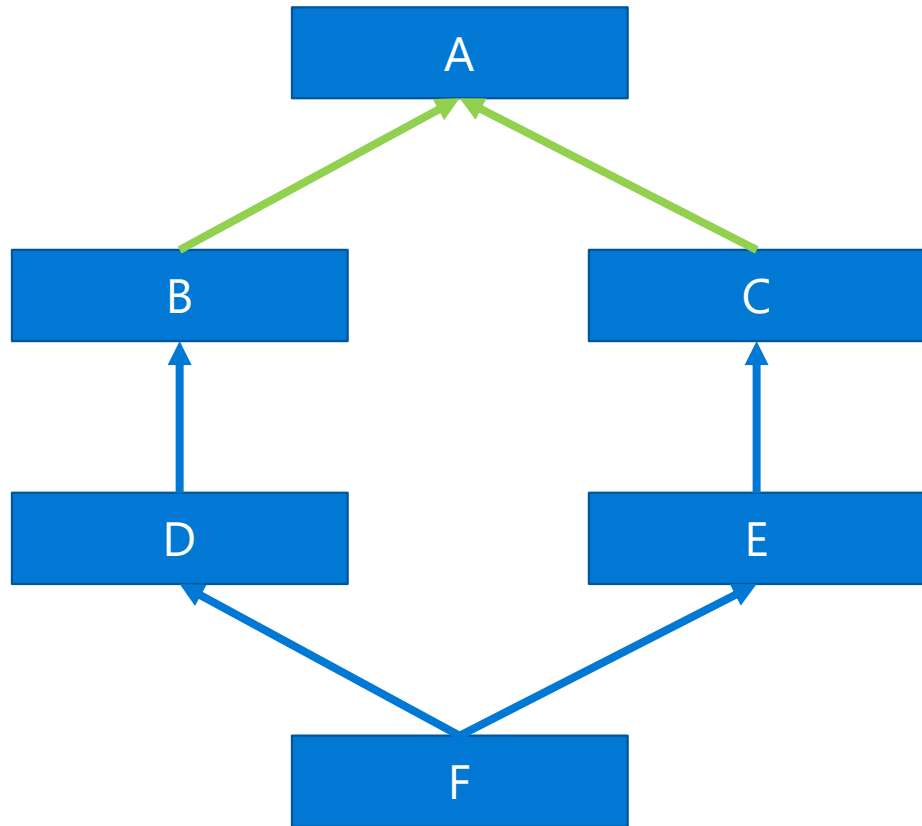
Object "F" layout:

```
0x0    <vbase table of B>  
0x?    <members of B>  
0x?    <members of D>  
0x?    vtable of A  
0x?    <members of A>  
0x?    <vbase table of C>  
0x?    <members of C>  
0x?    <members of E>  
0x?    <members of F>
```

There is a single copy of "A" in the object "F"

The object has a "virtual base table" to identify where "A" is relative to the object base.

Virtual Base Diamond Pattern



Virtual base inheritance allows for a single copy of the A.

Virtual base table contains the offset from the "this" pointer of the current object to the base of "Object A" inside the object.

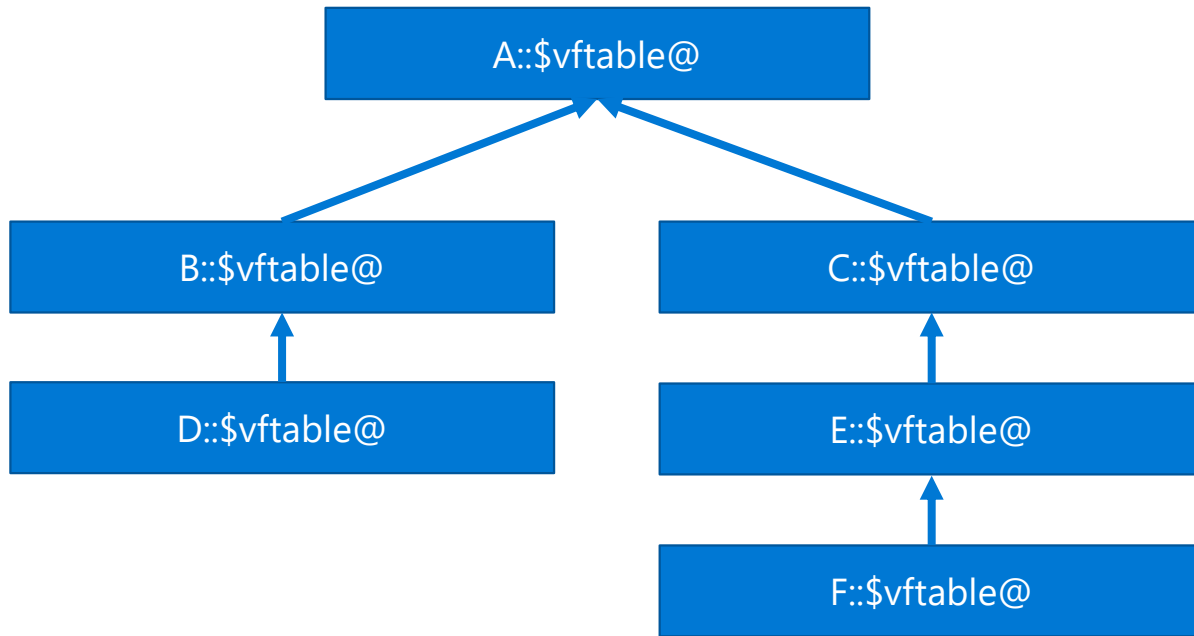
Object "F" layout:

```
0x0    <vbase table of B>
0x?    <members of B>
0x?    <members of D>
0x?    <vbase table of C>
0x?    <members of C>
0x?    <members of E>
0x?    <members of F>
0x?    vftable of A
0x?    <members of A>
```

Object "B" layout:

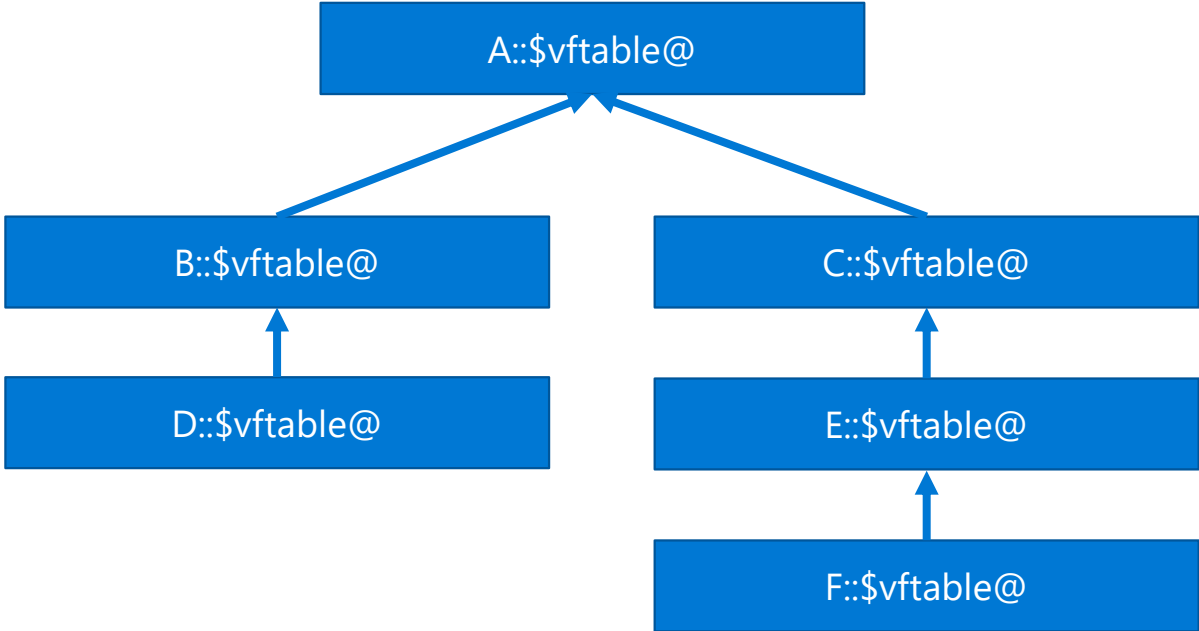
```
0x0    <vbase table of B>
0x?    <members of B>
0x?    vftable of A
0x?    <members of A>
```

Virtual Base Vftable View



Offset	Name
0x00	__CastGuardVftableStart
0x08	A::\$vftable@
0x10	B::\$vftable@
0x18	D::\$vftable@
0x20	C::\$vftable@
0x28	E::\$vftable@
0x30	F::\$vftable@
0x38	__CastGuardVftableEnd

Cast Validity



Legal Underlying Vftables

LHS Type (Type Being Cast To)

	A::\$vftable@	B::\$vftable@	D::\$vftable@	C::\$vftable@	E::\$vftable@	F::\$vftable@
A	✓	✓	✓	✓	✓	✓
B		✓	✓			✓
C				✓	✓	✓
D			✓			✓
E					✓	✓
F						✓

Bitmap Creation

LHS Type (Type Being Cast To)

Bitmap bits relative to start of object						
A_Bitmap	1	1	1	1	1	1
B_Bitmap	1	1	0	0	1	
C_Bitmap	1	1	1			
D_Bitmap	1	0	0	1		
E_Bitmap	1	1				
F_Bitmap	1					

Legal Underlying Vftables

LHS Type (Type Being Cast To)

	A::\$vftable@	B::\$vftable@	D::\$vftable@	C::\$vftable@	E::\$vftable@	F::\$vftable@
A	✓	✓	✓	✓	✓	✓
B		✓	✓			✓
C				✓	✓	✓
D			✓			✓
E					✓	✓
F						✓

Notes

- Bitmap has zeros in it, cannot be optimized to a range check.
- With complicated virtual base hierarchies, there may be no way to order the vftables that results in bitmaps of all 1's.
 - We don't even attempt to brute force alternate vftable orderings since very little code uses virtual base, just pay the overhead.

```

; Start of CastGuard check
; rcx == the right-hand side object pointer.
; First do the nullptr check. This could be optimized away but is not today.
; N.B. If the static_cast has to adjust the pointer base, this nullptr check
; already exists.
4885c9      test    rcx, rcx
7433       je     CodeGenTest!DoCast+0x3e

; Load the virtual base table
488b01      mov     rax, qword ptr [rcx]

; Right-hand side pointer adjustment (not part of CastGuard)
488d59e8    lea    rbx, [rcx-18h]

; Read from virtual-base table
8b5004     mov     edx, dword ptr [rax+4]

; Load vftable to compare against
488d057c6c0500 lea    rax, [CodeGenTest!MyGrandChild1:.`vftable']

; Add the offset read from the virtual-base table to the object pointer
4803ca     add     rcx, rdx

; Read the vftable
488b11     mov     rdx, qword ptr [rcx]

; Do the ordinal check using an ROL instruction to force alignment
; Low bits below the alignment get shifted to high bits making the
; value huge.
482bd0     sub     rdx, rax
48c1c240   rol    rdx, 3dh ; shift out low 3 bits
4883fa01   cmp     rdx, 3 ; ordinal range check

; Jump to app compat check if the range check fails
7336      jae    CodeGenTest!DoCast+0x65

; Load the bit vector and do a bit test against it using the ordinal
computed
488d059af20500 lea    rax, [CodeGenTest!CastGuardBitVector]
480fa310   bt     qword ptr [rax], rdx

; Jump to a failure stub if the bitmap test fails
7330      jae    CodeGenTest!DoCast+0x6c
eb02     jmp    CodeGenTest!DoCast+0x40

; End of CastGuard check

```

Optimizations

Bitmaps that are all 1's can be turned in to range checks

LHS Type (Type Being Cast To)

Bitmap bits relative to start of object						
A_Bitmap	1	1	1	1	1	1
B_Bitmap	1	1	0	0	1	
C_Bitmap	1	1	1			
D_Bitmap	1	0	0	1		
E_Bitmap	1	1				
F_Bitmap	1					

Optimizations

Bitmaps where all 1's are at fixed offsets from each other don't need a bitmap (but do need to enforce alignment)

LHS Type (Type Being Cast To)

Bitmap bits relative to start of object						
A_Bitmap	1	1	1	1	1	1
B_Bitmap	1	1	0	0	1	
C_Bitmap	1	1	1			
D_Bitmap	1	0	0	1		
E_Bitmap	1	1				
F_Bitmap	1					

1. Find pointer delta ($\text{current_vftable} - \text{address_of_D_vftable}$)
2. Compute the ordinal by shifting the delta
3. Do a range check on the ordinal

Optimizations

Prefer to not use vtables that have virtual base inheritance.

Example:

RHS type and LHS type each have 2 vtable's

- 1 vtable is part of a virtual-base class hierarchy

- 1 vtable is part of a normal inheritance class hierarchy

Do the CastGuard check on the vtable from the normal inheritance hierarchy so that no virtual base overhead is needed

Misc



One Definition Rule (ODR) Violations

Sometimes can be detected at compile time (i.e. ODR violation occurs in LTCG code)

To ease adoption, CastGuard will not protect these hierarchies but won't error.

If detected at link time (i.e. static lib introduces ODR violation):

If static lib introduces a smaller or identically sized vtable, we keep the vtable already placed in CastGuard region by the compiler.

If static lib introduces a bigger vtable, linker must select this vtable (documented behavior). This would break CastGuard, so we throw a linker error.