Pointers, references, const-ness and virtual function tables

REDPOINT

Pointers & references

REDPOINT

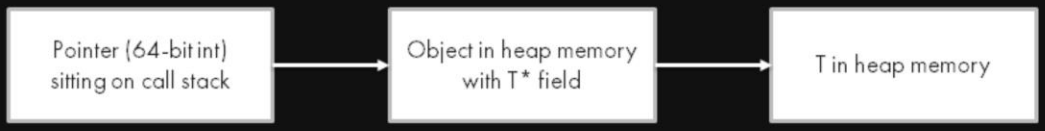
What is a pointer?

- Everything in memory has an address (we saw these in previous workshop)
- Pointer is just an address
- Since addresses are 64-bits on 64-bit machine, this means a pointer is really just a 64-bit unsigned integer under the hood

- Obviously faster to pass a 64-bit int through the call stack than copying the whole object

FYI can also store pointers inside objects on heap

- Everything is just blocks of memory, so totally fine and normal to store pointers to heap memory from heap memory



- Don't have a pointer to heap memory that you previously allocated (i.e. you forgot to call "delete" and you don't have any pointers to that location any more)?
  This is a memory leak

REDPOINT

What is a reference?

- Pointers are awkward
- Code has to deal with the memory address instead of the object just to avoid copies
- Need to dereference every time with * or -> just to get at object data
- Pointers can easily be null – leads to code having to excessively check for null, etc.
- Really in a lot of cases we just want to avoid copies...

What is a reference?

- Implicitly refers to another memory location
- "Invisible" to consuming code: you'll know it's a reference from the & on the type, but code can just treat it as if it's a local value

```
int main() {
    int a = 5;

    int& b = a;
    b = 10;

    printf("%d", b);
    return 0;
}
```

- Does what you expect: this code outputs "10".
- Under the hood "b" is a pointer to "a"'s memory location on the call stack
- But "invisible" to code.

REDPOINT

References vs pointers

References = implicit memory address

```
int main() {
    int a = 5;

    int& b = a;
    b = 10;

    printf("%d", b);
    return 0;
}
```

Reference is invisible – just looks like local variable.

Pointers = explicit memory address

```
int main() {
    int a = 5;

    int* b = &a;
    *b = 10;

    printf("%d", *b);
    return 0;
}
```

Code has to be aware of the pointer.

- Take address of "a" with &a
- Dereference b with *b to actually read/write value

REDPOINT

You can modify pointers, set them to nullptr, etc.

- With a pointer, your code is dealing with a memory address first and foremost.
- Change pointers by setting them to nullptr or taking the address of other values with "&" operator.

```c
int main() {
    int a = 5;
    int c = 10;

    // can set pointers to nullptr (0x00000...)
    int* b = nullptr;

    // set b to the address of a, then print it
    b = &a;
    printf("%d\n", *b);

    // set b to the address of c, then print it
    b = &c;
    printf("%d\n", *b);

    return 0;
}
```

REDPOINT

You can modify pointers, set them to nullptr, etc.

- Pointers can point anywhere!
- Nullptr 0x00000000
- Address of other things
- Any address you want!
- Including previously freed memory
- Or just garbage locations

```
int main() {
    int a = 5;
    int c = 10;

    // weird code, but C++ allows it because you're
    // really just modifying an uint64 underneath...
    int* b = &a;
    b += 1; // move b by sizeof(int) bytes

    // prints "10" because c just *happens* to be
    // after a in the call stack, but you can't
    // actually rely on this... :P
    printf("%d\n", *b);

    return 0;
}
```

REDPOINT

You can modify pointers, set them to nullptr, etc.

- Like seriously you can just put whatever you want in these things :P

```
int main() {
    int a = 5;
    int c = 10;

    int* b = (int*)0x1000100010001000uL;

    // segfaults because b is a pointer to
    // memory address 0x1000100010001000
    // which isn't something we actually own
    printf("%d\n", *b);

    return 0;
}
```

REDPOINT

Type operator comparisons

```
class FTestClass {
public:
    std::string val = "hello";
};

int main() {
    FTestClass test;
    printf("%s\n", test.val.c_str());

    FTestClass& ref = test;
    // notice how we access through ".val" just
    // like we do for "test" itself.
    printf("%s\n", ref.val.c_str());

    // notice how we now need to access via ->
    // because we need to dereference the pointer
    // first
    FTestClass* ptr = &test;
    printf("%s\n", ptr->val.c_str());

    // could also do this
    printf("%s\n", (*ptr).val.c_str());

    return 0;
}
```

If "a" is an FTestClass or FTestClass&

&a = take address of a, returns FTestClass*

a.member = access member inside FTestClass data

If "a" is an FTestClass* (pointer to FTestClass)

*a = dereference pointer, returns FTestClass&

a->member = dereference pointer and access member inside FTestClass data

*Note how *a returns a reference! If it returned FTestClass instead then we'd have to then do a copy to use it...*

REDPOINT

When to use them

References

Function parameters
- To avoid copies should be const &
- Out parameters should be &

When it should not be null

When code doesn't need to modify what the reference points to

When code doesn't need the address of the target object (technically can still get via &a though)

Pointers

Typically members inside classes or structs that are memory addresses of other objects will be T* not T&

Can be nullptr or any address

Referring to raw blocks of memory or things created with new()

Pointer to a pointer T* *

- Does what you expect
- Pointer to a pointer which then refers to T
- Example of memory with 0x1234 being the location of the pointer pointer
  - 0x1234 = 0x5678 (address of pointer)
    0x5678 = 0x1111 (address of object)
    0x1111 = ... actual object data ...
- Not used very often

REDPOINT 

Rvalue references &&

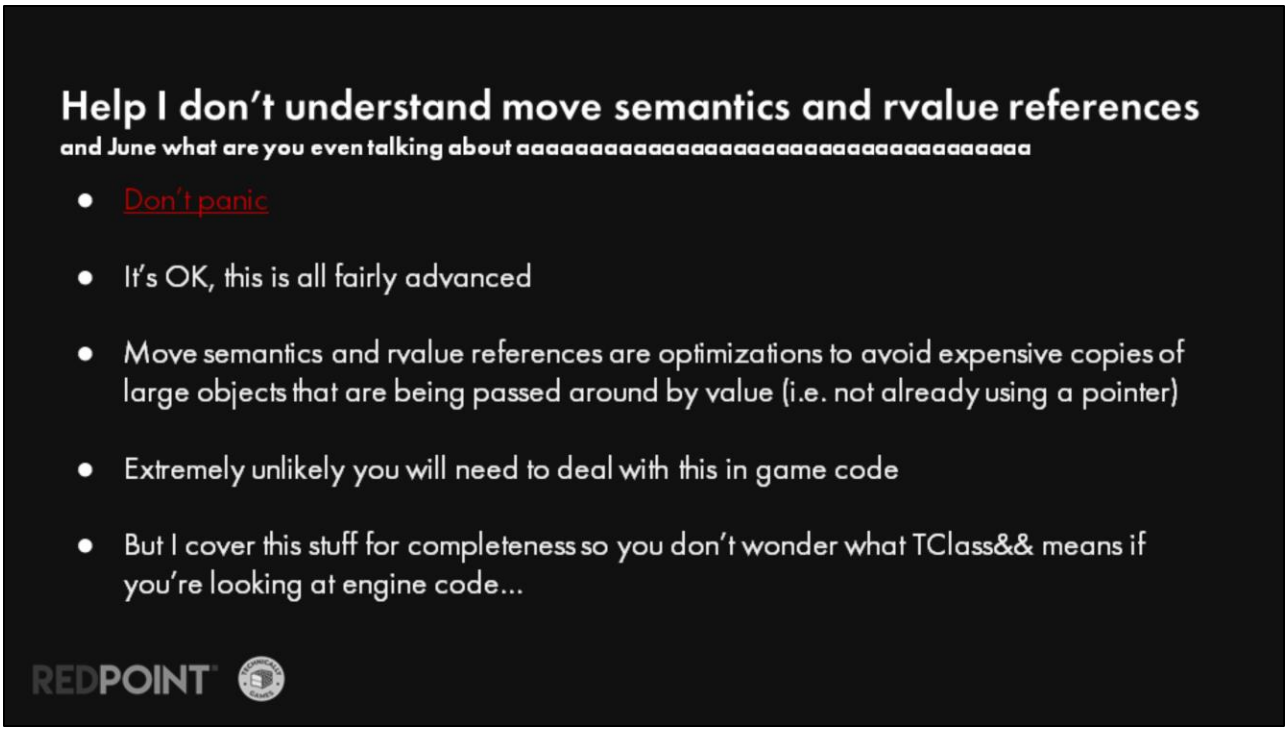
https://godbolt.org/z/baKxq67Wd

- Reference restricted to temporaries and constants
  "things without a memory address"
- Used for move constructors and move assignment operators, since they are more efficient than copying
- Where we want to make sure the target isn't referenced or stored elsewhere so we can safely "steal" it's resources
- `std::move()` for allowing moves of things you normally couldn't; in Unreal Engine use `MoveTemp()` instead.

```
class FClass {
    // image this class has resources
    // that would be expensive to copy...
};

void wants_rvalue_reference(FClass&&)
{
    // imagine this function "steals"
    // the resources out of FClass and
    // prevents FClass from freeing those
    // stolen resources (just like a move
    // constructor would)
}

FClass returns_instance()
{
    return FClass();
}

int main()
{
    // OK
    wants_rvalue_reference(FClass());
    wants_rvalue_reference(returns_instance());

    // NOT OK
    FClass val;
    wants_rvalue_reference(val);
    // not safe to steal things out of val
    // because could still use it here...

    // OK - std::move escape hatch
    wants_rvalue_reference(std::move(val));
    // promise I won't use val after this point
    // (though I still incorrectly could...)
}
```

REDPOINT

- Example: stealing the pointer to a heap object inside a move constructor
- Imagine "FClass" allocates 1MB of memory, expensive to copy, cheap to move/steal



- I didn't even understand what T&& meant and why they were used in move constructors until I tried things in Godbolt for this slide, got it wrong and then spent 2 hours researching it, so it's fine if you don't understand it

Const-ness

REDPOINT

Const a.k.a readonly

- You can make anything const in C++
  - Variable/parameter types: Makes the type "read-only"
  - Template parameters: So you can have an array of const objects
  - Functions in classes/structs: Declares that the function does not modify the instance and can be called if the caller has the type as const
- Const is for correctness – there's no inherit optimization benefits, but it does make use of references safer

Const examples

```
class FClass {
public:
    void mutating_call() {}
    void const_call() const {}
};

int main()
{
    // pointer to FClass
    FClass* p = new FClass();
    p->mutating_call();
    p->const_call();

    // pointer to const FClass
    const FClass* c = new FClass();
    c->mutating_call(); // not permitted, mutating_call not const
    c->const_call();
}
```

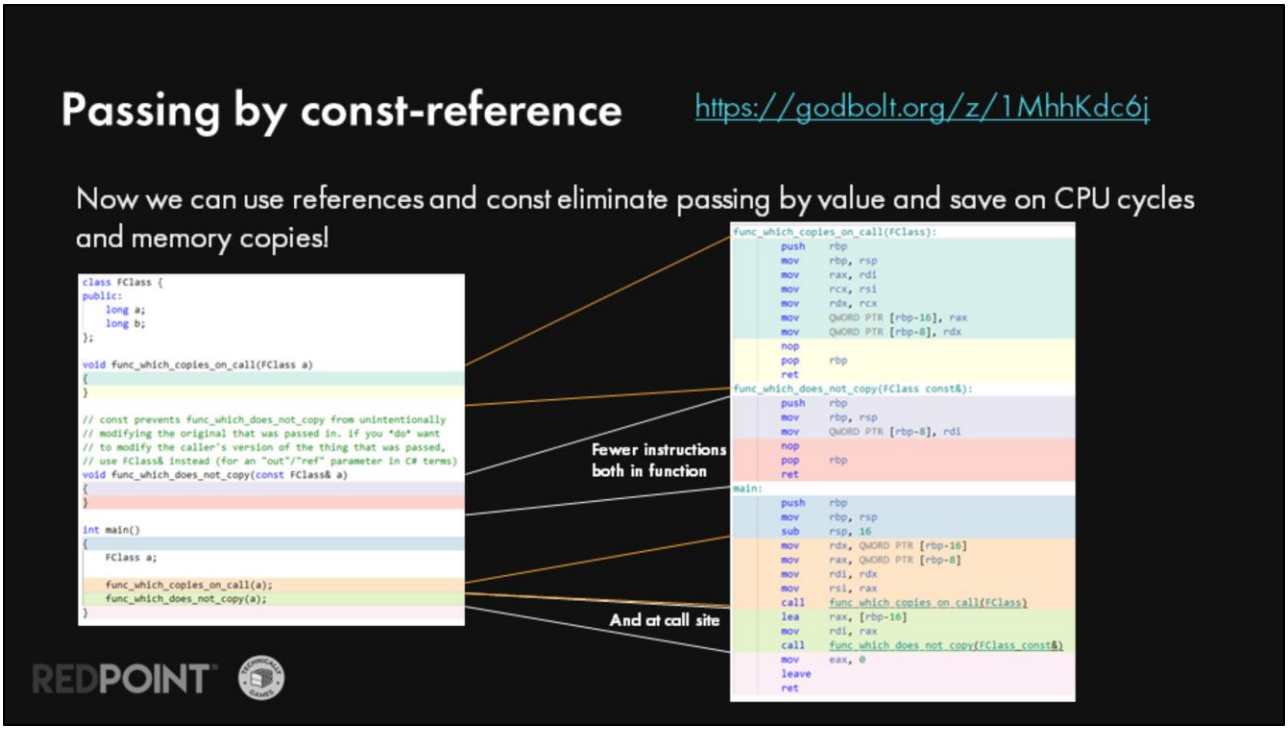
Can't call "mutating_call" – FClass isn't const in that context

Const examples

```
int main()
{
    // pointer to FClass
    // - can change the FClass
    // - can change what the pointer points to
    FClass* p = nullptr;

    // pointer to const FClass
    // - can't change the FClass
    // - BUT can set the pointer to point at something else!
    const FClass* c = nullptr;

    // const pointer to FClass
    // - can't change the pointer
    // - BUT you can change the FClass
    FClass* const cp = nullptr;

    // const pointer to const FClass
    // - can't change either
    const FClass* const cc = nullptr;
}
```

Various examples on how "const" modifies the type depending on where it is

```
int main()
{
    FClass a;

    // same const-semantics as pointer examples
    FClass& p = a;
    const FClass& c = a;

    // these don't work because references can't be changed
    // to point to something else later, so there's no
    // const variants at the reference level like there is
    // for pointers
    FClass& const cp = a;
    const FClass& const cc = a;
}
```

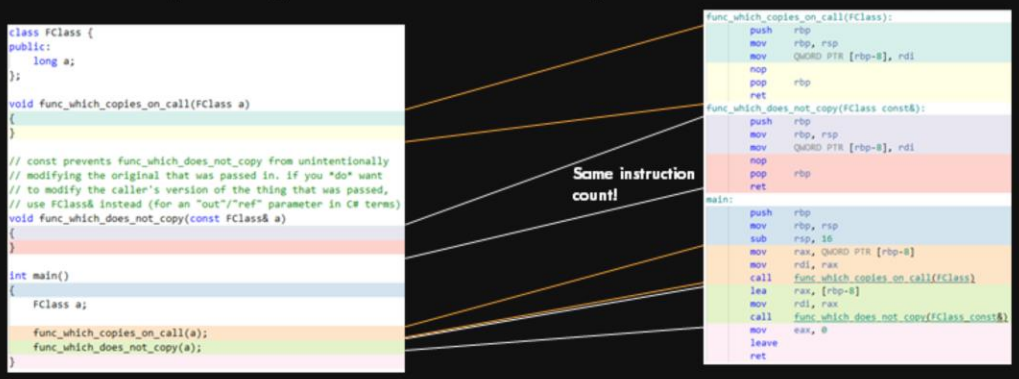



- Const reference: this is a reference to a const object

Passing by const-reference

https://godbolt.org/z/KWo9GsbKz

Note: No benefit if the thing being passed is same size as a memory address, since it will fit in the same space regardless of whether it's copied.



REDPOINT

Virtual function tables

a.k.a. how overrides *really* work

REDPOINT

Let's think about this example

```
class BaseClass
{
public:
    virtual void A() {}
};

class DerivedClass : public BaseClass
{
public:
    virtual void A() override {}
};

int main()
{
    BaseClass* b = new DerivedClass();

    // How do we know which "A" to call at runtime?
    b->A();

    return 0;
}
```

- C++ compiled ahead of time
- Can't do dynamic lookups or anything like that
- So how *does* this actually work?

Re-implementing VFTs to show how they work

VFT holds function pointers for all the virtual functions in a class:

```
struct BaseClass_VFT
{
    // pointer to function with signature void().
    void (*A_ptr)(class BaseClass* self) = nullptr;
};
```

https://godbolt.org/z/d919vf74o

Class then contains the VFT as part of it's data:

```
class BaseClass
{
private:
    static void BaseClass_A_internal(BaseClass* self) {}

public:
    // The VFT for "BaseClass". Whenever we call a "virtual"
    // function, we'll look it up in this table.
    BaseClass_VFT VFT;

    BaseClass()
    {
        // Set up the VFT in the base class.
        VFT = {&BaseClass::BaseClass_A_internal};
    }
};
```


Re-implementing VFTs to show how they work

Derived classes just change the VFT when they're constructed:

```
class DerivedClass : public BaseClass
{
private:
    static void DerivedClass_A_internal(BaseClass* self_d) {}

public:
    DerivedClass() : BaseClass()
    {
        // Override the VFT to point elsewhere.
        VFT = {&DerivedClass::DerivedClass_A_internal};
    }
};
```

Whenever we call a virtual function, we call it by looking up the pointer in the VFT:

```
int main()
{
    BaseClass* b = new DerivedClass();

    // Call via the VFT.
    b->VFT.A_ptr(b);

    return 0;
}
```

REDPOINT 

Differences between our re-implementation & real VFT

- VFTs are generated at compile time and stored in the built executable, not generated in the constructor
- VFT entry in class is a *pointer to VFT*, not the VFT data inline – this means the VFT data is always just a single pointer inside memory allocated for a class

Actual VFT example (as compiled by C++)

https://godbolt.org/z/6jTP89Er7

Things to note:

- Function table pointer being set in class during constructor
- Call against rdx – immediately prior assembly is looking stuff up in that function table
- First field no longer matches object address, this is because VFT pointer is now at the start of the memory block

Why virtual functions are slower

Non-virtual function addresses are known at compile time:

- If you call a non-virtual function, the compiler just emits "call" to jump straight into the non-virtual function.

Virtual functions have been looked up in the VFT at runtime:

- Get VFT pointer inside class
- Resolve pointer and look up function offset inside VFT
- Call to the address that the VFT indicates

Not that much slower, but will be noticeable in performance critical code like iterating over lots of data...

How does multiple inheritance work?

C++ doesn't have interfaces as a first-class type; interfaces in C++ are just pure virtual classes.

So if we want to do interfaces, we need to inherit from multiple classes.

Each base class has it's own VFT, so when we upcast to an interface, how does the calling code know the offset of the VFT and data?

Example problem: Where is BaseB's VFT in Derived?

```
class BaseA
{
private:
    long a;

public:
    virtual void A() {}
};

class BaseB
{
private:
    long b;

public:
    virtual void B() {}
};

class Derived : public BaseA, public BaseB
{
public:
    virtual void A() override {}
    virtual void B() override {}
};

int main()
{
    BaseB* b = new Derived();

    // How can this code know where the VFT for BaseB
    // is? All it has is a BaseB* pointer and knows
    // nothing of Derived.
    b->B();
}
```

https://godbolt.org/z/jPdWWqve9

- Where do we put BaseB's virtual function table pointer?
- Can't put it at the start because BaseA's VFT pointer is there
- How can code using BaseB* know where the VFT is inside Derived?

REDPOINT

Solution: Change the pointer when casting ☺

Memory layout of "Derived":

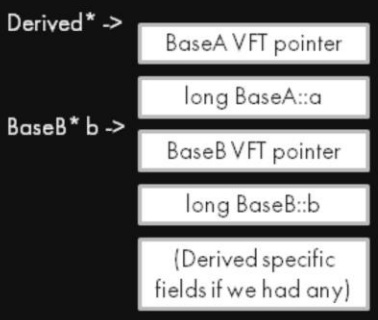
- When we cast from Derived* to BaseB*, we adjust the returned pointer by the size of BaseA.

https://godbolt.org/z/7nf9cea4q

REDPOINT

Solution: Change the pointer when casting ☺

Memory layout of "Derived":



- When we cast from Derived* to BaseB*, we adjust the returned pointer by the size of BaseA.
- When we cast from BaseB* to Derived* we do the reverse ☺

https://godbolt.org/z/7nf9cea4q

REDPOINT

THE SUPER IMPORTANT SUMMARY SLIDE

- Make sure you understand how pointers and references work – you will be using them a lot.
- Use const references to pass things into function parameters if they're larger than a memory address (e.g. strings, etc.)
  - No need to pass by const& for things like int, long, bool, etc. These are all same or smaller than memory address.
- Helpful to know how virtual functions work under the hood so you know what the memory layout of things will look like, but not critical to writing C++.

REDPOINT