CS354P

DR SARAH ABRAHAM

OBJECT-ORIENTED PROGRAMMING

OBJECT-ORIENTED PROGRAMMING

- Treats functionality and data as “objects”
- Objects have their own properties and methods that they can access
  - Objects have notion of “self” (`this` in C++)
- Can use classes to provide property and method definitions
  - Instances (objects) created from these definitions have unique property values based on self
- Prototyping is another way to implement OOP that avoids the class/instance dichotomy
  - Prototypes are **objects** that other **objects inherit from**

FOUR PRINCIPLES OF OOP

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism

ENCAPSULATION

- Mechanism of hiding data implementation details
  - Designed to facilitate using object functionality without having to understand underlying details
  - Prevents side effects that occur when data is manipulated directly
  - Simplifies debugging process if mistakes does occur
- `public` versus `private` and `protected` access modifiers help expose what is necessary to see but hide what is unnecessary to see
  - Make properties private and expose with getters/setters
  - Make helper methods private

OKAY BUT...

- Encapsulation is a pain
- Ideally make everything private then expose to public/protected only when necessary
  - Hard to do when prototyping
  - Easy to just make everything public and imagine you'll rework the class to be better designed once it's done...
- Should you still try to follow best practices from the start?
  - Yes. Think of it as a way to help you organize your thoughts on what the end-user should see/not see
  - Yes. This principle is less important in small bodies of code, but as systems get larger, encapsulation prevents confusion and saves debugging time

ABSTRACTION

- General concept and goal in programming of focusing on the model/design rather than implementation details
- Abstraction in OOP focuses on presenting available functionality to the user while hiding how the functionality is implemented
  - Commonly use concept of **interfaces** to define the functionality that an object implementing the interface will have

INTERFACES

- Programming structure that defines expected properties or behaviors of a class that implements it
- Different languages vary in terms of how interfaces are implemented/what is allowed
- Can allow or not allow state (e.g. properties)
- Can be inheritance-based or use mix-ins (class contains methods but not part of the inheritance chain)

ABSTRACT CLASSES

- C++ uses abstract classes to implement interfaces
- Abstract classes cannot be used to create instances
  - Child classes can instantiate objects
  - Abstract class can be referred to by references and pointers
- A C++ abstract class is any class that has a **pure virtual** function
- Declaring a pure virtual function:

```
virtual returntype functionname() = 0;
```

VIRTUAL AND OVERRIDE

- `virtual` notifies the compiler that the specified function is **virtual** and requires a dynamic binding (i.e. should only be looked up at runtime)
  - Allows the derived class's function implementation to be executed at runtime **overriding** the base class's virtual function
    - Must be defined in base class if it is not a pure virtual function

```
virtual returntype functionname ();
```

- `override` ensures the function is overriding a virtual function from the base class
  - C++11 feature that generates a compiler error if derived class is not correctly overriding base class virtual function

```
returntype functionname () override;
```

VIRTUAL FUNCTIONS IN UNREAL?

- Unreal UObjects do **not** support pure virtual functions but virtual functions are used extensively
  - PURE_VIRTUAL macro makes compiler check that all child classes have implemented the function to “imitate” pure virtual
    - Why no pure virtual functions?
- UClass system requires that all UObjects be instantiated
  - Creates at least one instance of the Class Default Object (CDO)
  - Uses this object as a **prototype** for all objects created from that class
  - **Class constructor only called once to create this object!**

MAKE SENSE?

INHERITANCE

- Mechanism that allows an object or class to be based another object or class
  - Child class/object acquires most properties and behaviors of parent class/object (does not acquire constructor, destructor, etc)
- May be a **subtyping** mechanism that allows classes to express an "is-a" relationship
  - This is the case in C++
- Two broad categories of inheritance:
  - Class-based and prototype-based

CLASS-BASED INHERITANCE

- Use of classes to define properties and behaviors representing the "physical" objects
  - Do not "physically" exist until **instantiated** as objects of that type
- Child classes extend parent classes
  - Abstractions of abstractions
- Child objects inherit properties and behaviors of all previous class abstractions
  - An **instance** of an abstraction of an abstraction

PROTOTYPE-BASED INHERITANCE

- Use of objects to define initial properties and behaviors as well as "physical" instantiation
  - Generalized objects can be cloned and/or extended to make new objects or new types
- To create inheritance, child object is **cloned** from parent object then given properties and behaviors unique to it
- Child objects cloned from this generic child object
  - An **instance** of a generalization

SO IS UNREAL CLASSICAL OR PROTOTYPAL?

- C++ is a classical language
  - Prototypal languages include Javascript and Lua
- UE5's underlying UClass inheritance is prototypal but it looks and behaves much like a classical model
  - Prototypal inheritance is more flexible, dynamic, and potentially efficient than classical inheritance
- Key differences primarily relate to the constructors
  - Class constructors **cannot** contain runtime logic
  - Subobjects **must** be constructed before object is constructed

UE5 C++ OBJECT CONSTRUCTION REDUX

- `CreateDefaultSubobject`
  - **Only callable in the class constructor**
  - Creates the CDO instance
- `NewObject<T>`
  - **Called during gameplay**
  - Convenience template for constructing an object
- `SpawnActor<T>`
  - **Called during gameplay**
  - Convenience template for placing an Actor into a level
  - Wrapper around `NewObject<AActor>`
- All object construction ultimately calls `StaticConstructObject_Internal`

C++ TEMPLATES

- Templated functions operate with generic types
  - Allows for the creation of functionality that exists in only one place but can work on multiple types of objects
- Templated classes have members that use template parameters as types
  - Facilitates the creation of interfaces across multiple derived classes
- Extremely important, and deep, feature of C++ for "simplifying" the issues related to being strongly-typed
  - Simplifying because it allows the writing of generic code once for use by multiple types
  - "Simplifying" because it can be used for metaprogramming, or using programs as data to create new programs

POLYMORPHISM

- The representation of a single entity using multiple types
- Polymorphism types:
  - *Ad hoc* allows arguments of different types (e.g. function overloading)
  - *Parametric* uses generics to handle values of different types while maintaining static type-safety
  - *Subtype* allows instances to have multiple types
- OOP polymorphism usually refers to subtype polymorphism
  - Can achieve the others in an OOP context though (e.g. see discussion on templating)

KNOWING THE OBJECT TYPE

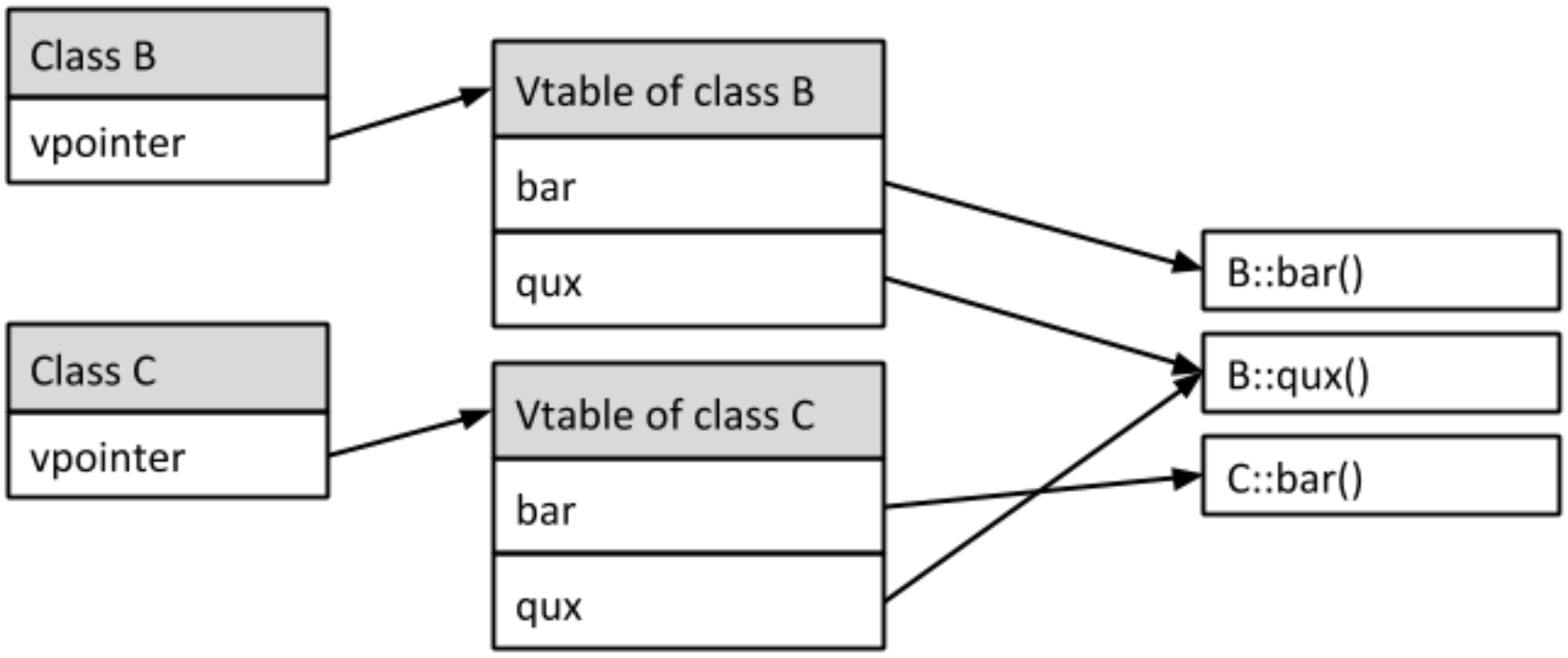
- If a single object can have multiple types, the correct type must be determined at runtime based on context
  - i.e. function is called on the base class type but instances are of the derived class
- How to do this in C++?
- Compiler creates a hidden pointer in the base class that all derived classes inherit
  - Pointer connects to a table of instance's virtual functions

VPTR AND VTABLES

- Vptr is the pointer created at compile time for each **instance**
- Vtable is the static table for each base/derived **class** containing function pointers to all virtual functions of base class
  - Function pointers point to the most derived version of the function
- Each instance's vptr points to its most derived class's vtable
- When polymorphic functions are called (e.g. are virtual), vptr is accessed and the correct version of the function is accessed based on the vtable pointers
  - Does have some overhead over non-virtual functions

EXAMPLE OF CLASSES AND THEIR VTABLES

- From this image, what do we know about the relationship of B and C, which functions are virtual, and which functions are implemented by what class?



https://pabloariasal.github.io/2017/06/10/understanding-virtual-tables/

CHECKING TYPES IN UE5

- Since Unreal implements reflection (the ability of an object to examine itself), we can also efficiently check object type at runtime
  - `instance->IsA(OtherClass::StaticClass());`
- Allows for more nuanced, flexible interactions with objects than just using virtual functions
  - e.g. you get back an array of all PrimitiveComponents colliding with an Actor, but you only need to perform an operation on the ones of a given type

UNREAL INTERFACES

- Unreal has UInterface which facilitates implementation of interfaces without an abstract base class
- Derives from `UInterface` rather than `UObject`
  - Use `UINTERFACE` macro rather than `UCLASS` macro
- Exposes this interface to Unreal reflection system

INTERFACES VS COMPONENTS?

- Fundamentally both tackle the same problem:
  - How do I have shared functionality between unrelated objects?
- Deciding between a component and an interface can be tricky and is largely based on personal experience and preference.
- In general I prefer components but use interfaces if:
  - The functionality is between totally unrelated objects
  - The functionality serves an unrelated purpose between these objects
- More info here: https://dev.epicgames.com/documentation/en-us/unreal-engine/interfaces-in-unreal-engine

CRITICISMS OF OOP

- OOP is quite contentious these days
  - Many fervent supporters and many fervent detractors
- General arguments against are that it is:
  - Too complex in practice
  - Too focused on types and data
  - Not as flexible as other approaches
  - Too simplistic in its modeling

SO WHY OOP?

- OOP paradigm meshes well with the modeling of real-world concepts of objects and object interactions (i.e. what we want in most video games)
- C++ is a highly efficient, feature-rich language with great cross-platform compiler support
- Broad specifications of OOP means language implementations can be more or less efficient and more or less legible
  - Not necessarily the right solution for all problems but useful when applied in a domain-specific way

UNREAL AND OOP

- Unreal does take an object-oriented approach to its architecture
  - Built around the fundamental principles of OOP
  - Built on an object-oriented language
- Unreal doesn't necessarily look like a "typical" OOP implementation for something built on C++
  - Overlap with Javascript and other dynamic languages
  - Takes the efficiency of C++ and applies it in a more dynamic way for the class of problems it is built to solve