WHAT WE WANT FROM A GAME ENGINE

- Runtime efficiency
- Cross-platform compatibility
- Ease of iterative development
GAME ENGINES BUILT ON C++

- Frostbite
- Unreal Engine
- Lumberyard (formerly CryEngine)
- Source
- Unity
- Godot
- Game Maker
- Cocos2D
GAME ENGINES/FRAMEWORKS BUILT ON SOMETHING ELSE

- id Tech 1-3 (C)
- JMonkeyEngine (Java)
- Scage (Scala)
- Flixe (ActionScript)
- MonoGame (C#)
- Allegro (C)
- Three.js (JavaScript)
C++ IN UE4

- Epic calls their Unreal C++ libraries “assisted C++”
  - Lots of custom functionality, data structures, and types
  - Quality of life “language” features
  - Designed to work with their in-house scripting language
- Can still connect standard C++ libraries but encouraged to use their C++ style for game objects etc
UE4 CUSTOM DATA STRUCTURES

- UE4 is its own ecosystem of classes and functionality
  - Should aim to work with it rather than against it
  - Less about knowing C++ and more knowing how to read documentation and work within a system’s limitations
- Four broad categories of gameplay classes
  - UObject, AActor, UActorComponent, UStruct
- Additional tools for data management
  - Custom object iterators, strings, containers
UOBJECT

- Base class for all UE4 objects (object type defined by UClass)
- Allows for:
  - Reflection of properties and methods
  - Serialization of properties
  - Garbage collection
  - Finding the UObject by name
  - Configurable values for properties
  - Networking support for properties and methods
Reflection is the ability of a program to examine and modify itself at runtime.

Extremely useful feature for editor, serialization, garbage collection, network replication, and Blueprint/C++ communication.

- Basically anything that benefits from being able to assess objects/data at runtime.

C++ does not natively support reflection.

UE4’s reflection system built on UObject/UClass.

Reflection system is opt in:

- `#include "FileName.generated.h"
- `UCLASS()`
- `GENERATED_UCLASS_BODY() / GENERATED_BODY()`

UnrealHeaderTool invoked during build to parse C++ headers for UE4 class meta-data to implement UObject features.
Serialization

- Serialization is the process of formatting data of an object such that it can be stored or transmitted then successfully reconstructed
  - Stored in memory or file system
  - Transmitted across a network
- FArchive is archive base class for loading, saving, and garbage collecting in a byte-order neutral way
  - Many different subclasses for saving and reconstructing game data
  - Saving data is a surprisingly difficult and nuanced issue...
GARBAGE COLLECTION

- Handles memory management for all UObject instances
  - Relies on reflection to inspect objects and determine if they can be safely deleted
- Actor objects automatically garbage collected at the end of a level
- Calling Destroy removes them from game immediately and allows full deletion during next garbage collection

UE4 GC Guidelines:

- All class members should be declared as UPROPERTY
- Member pointers should only point at UObjects
- Any non-UObject pointers must point to something “global” in engine or something within its own UObject
- TArray is only safe container for UObject pointers
**USTRUCT**

- Specialized struct for UE4 purposes
- Marked with `USTRUCT()`
- Not garbage collected
- Passed by value
- Built-in UStructs:
  - `FVector`, `FRotator`, `FQuat`, etc...
OBJECT/ACTOR ITERATORS

- Used to iterate over UObject instances

```cpp
for (TObjectIterator<UObject> It; It; ++It)
{
    UObject* CurrentObject = *It;
    // Do something
}
```

- Can also look for instances of a particular class

```cpp
for (TObjectIterator<UMyClass> It; It; ++It)
{
    // ...
}
```
STRINGS AND TEXT

- Lots of different functionality depending on need:
  - FString, FText, FName

- FString are mutable strings with UE4 specific functionality
  - Created with TEXT macro

- FText are designed for localized text
  - Created with NSLOCTEXT macro
  - Macro takes namespace, key and value for default language

- FName stores recurring string as identifier
  - Fast, space-efficient representation across multiple objects
  - Used for identified bone names in a model’s skeleton, player name, etc
CONTAINERS

- Dynamically sized containers for UE4 C++ objects
- Supports iterators and for-each loops
- Common containers are TArray, TMap, TSet
- TArray similar to std::vector but elements are garbage collected
- TMap similar to std::map (elements are not garbage collected)
- TSet similar to std::set
WHAT ABOUT STANDARD LIBRARY?

- Generally Unreal Engine avoids standard library
  - Faster implementations
  - Additional memory allocation control
  - Consistent codebases and idioms
- UE4 does however use some std features rather than reimplement
  - `atomic`
  - `regex`
- Still possible to use std features but avoid mixing and matching UE4 idioms as much as possible
  - Can cause compiler issues
NOTE: MAKING SIZES EXPLICIT

- `int` and `uint` can be used if width is unimportant
  - Guaranteed at least 32 bits in length
- Cannot be used in serialized or replicated formats
- Use `int32`/`int64` whenever possible
- Enumerations should use `uint8` if they are exposed to Blueprints
UE4 AND C++ LANGUAGE FEATURES

- UE4 favors massive portability to C++ compilers over language features
- Uses many C++14 features but programmers should avoid compiler-specific features unless wrapped in preprocessor macros or conditionals
- Some things you can use:
  - `static_assert` (valid for any compile-time assertions)
  - `override` and `final` (strongly encouraged)
  - `nullptr` (use instead of NULL macro)
WHAT ABOUT AUTO?

- auto keyword tells compiler to deduce its type from the initial expression of the variable
  - Very nice C++11 feature that simplifies type-handling
- Not recommended by Epic for use in UE4 because of readability
  - Doesn’t assist users using merge/diff tools or view source files within a repo
- Acceptable to use if...
  - Binding a lambda to a variable
  - For iterator variables where iterator type is verbose and impairs readability
  - In template code where type cannot be easily discerned
- An example of auto in an iterator:

```cpp
for (auto EnemyIterator = EnemySet.CreateIterator(); EnemyIterator; ++EnemyIterator) { ... }
```
WHAT ABOUT RANGE-BASED FOR LOOPS?

- Range-based for loops execute over the elements within an expression
  - Very nice C++11 feature that encourages safety and readability
- Recommended by Epic for use in UE4
  - Work with TArray, TMap, and TSet
  - Commonly used when finding actors of a certain type in a level
RANGE-BASED LOOP EXAMPLES

TArray< UPrimitiveComponent * > overlappingComponents;
hitBox->GetOverlappingComponents(overlappingComponents);
for (UObject* object : overlappingObjects)
{ ... }

for (auto EnemyIterator = EnemySet.CreateIterator();
EnemyIterator; ++EnemyIterator)
{ ... }

- or rewrite with auto as:

for (const auto& Enemy : EnemySet)
{ ... }
WHAT ABOUT LAMBDAS/ANONYMOUS FUNCTIONS?

- Anonymous functions are unnamed functions that can be passed to higher order functions
  - Very nice C++11 feature that was extended in C++14
- Safe to use in UE4
  - Encouraged to practice good readability and documentation
- UE4 codebase uses a lot of function pointers, which stateful lambdas (i.e. lambdas with capture) can’t be assigned to
WHAT ABOUT SMART POINTERS?

- Smart pointers allow for automatic memory management of objects when pointers are going out of scope
  - Very nice C++11 feature that creates more stable code with fewer memory leaks
- Epic provides a custom implementation of C++11 smart pointers in their own Smart Pointer Library
UE4 SMART POINTER LIBRARY

- Unique Pointers solely and explicitly own the object referenced. Ownership can be transferred but not shared (i.e. no copying)

- Shared Pointers allow multiple owners of the object referenced. Object is reference-counted and deleted when it has no Shared Pointers or Shared References referencing it

- Weak Pointers do not own the object they reference, so object does not maintain a reference counter. Thus it can become null at any time (can produce a Shared Pointer for safety during direct usage)

- Shared References are like Shared Pointers but can only reference non-null objects. A Shared Pointer created from a Shared Reference is guaranteed to not be null
NOTE: SMART POINTER LIMITATIONS

- Not compatible with UObjects which have a separate memory management system!

- Smart Pointers are performant and small (at most 2x a C++ pointer) but creating and copying them has overhead as does reference counting.

- Thread-safe Smart Pointers are slower (atomic reference counting) and must be expressly declared:
  
  - e.g. TSharedPtr&lt;T, ESPMode::ThreadSafe&gt;
CONCLUSION

- C++ is a great language for building UE4 and other game engines but not sufficient for all game development needs.
- The beauty of C++ is its flexibility and efficiency.
- The wisdom of C++ development is knowing when and how to use language features to build for your particular needs.
- Know the project goals before building!
REFERENCES

