Unreal Engine 5 - Lesson 2 - C++ VS Blueprint

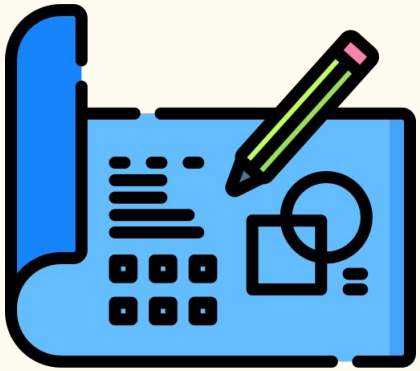
Nicolas Serf - Gameplay Programmer - Wolcen Studio
serf.nicolas@gmail.com

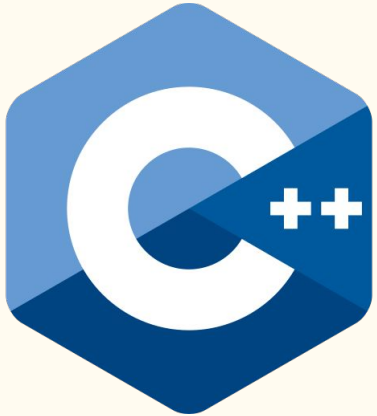
Summary

- 2 ways of programming
- Is the strict distinction usage worth to be discussed ?
- Design concept
- Differences between C++ & Blueprint
- Making both world works together
- Advantages of blueprint
- Advantages of c++
- C++
- Blueprint

2 ways of programming

- **C++**
  - **Writing code using** a general **text based programming language**
  - Access to **any low level** code from the engine
  - Allows to do some **things that are not possible in blueprint**
- **Blueprint**
  - Use **node and graph** in order to program systems
  - **Tailored for higher level game programming**

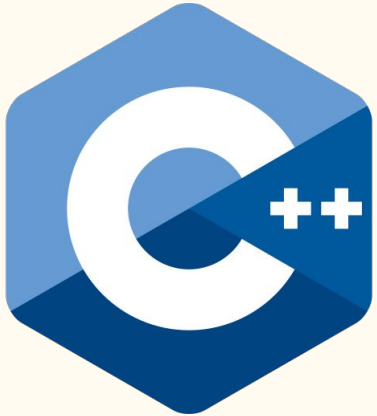




Is the strict distinction usage worth to be discussed ?

- Actually **NO**
- Starting by making the **decision** of **either** using blueprint or C++ is already a **bad decision**
- Unreal is **designed** in such a way that **C++** and **Blueprint** are very **complementary**

- A better question may be : **Where does it make sense to use Blueprint ? Where does it make more sense to use C++ ?**
- This lesson will focus on answering that question with some **highlights**, and understand a bit more what is going on **under the hood**, **performance wise.**

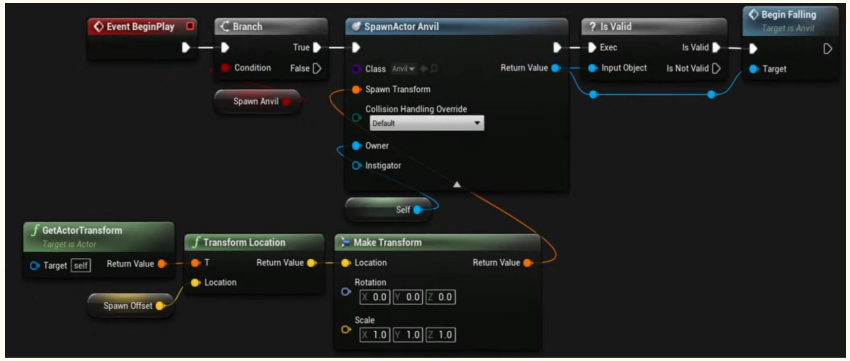


Comparing an Actor spawn between C++ & Blueprint

C++ Version

```cpp
void ACoyote::BeginPlay()
{
    Super::BeginPlay();

    if (bSpawnAnvil)
    {
        const FVector SpawnOffset(100.0f, 0.0f, 1500.0f);
        const FVector SpawnLocation = GetActorTransform().TransformPosition(SpawnOffset);
        const FTransform SpawnTransform(FQuat::Identity, SpawnLocation);

        FActorSpawnParameters SpawnInfo;
        SpawnInfo.Owner = this;

        AAnvil* Anvil = GetWorld()->SpawnActor<AAnvil>(AnvilClass, SpawnTransform, SpawnInfo);
        if (Anvil)
        {
            Anvil->BeginFalling();
        }
    }
}
```

Blueprint Version



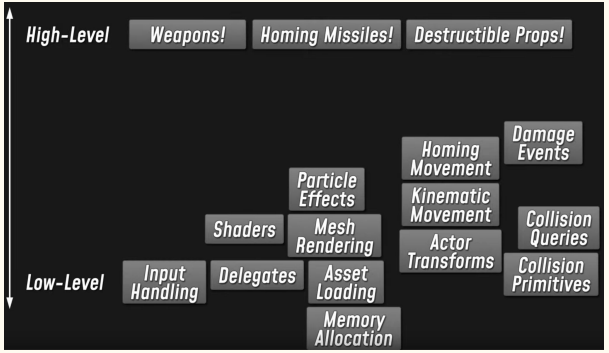
- Being different, they both **produce the exact same result**
- You are **writing code** in both
- You are **dictating** how the program will **behave** at runtime
- You'll deal with **Software design thinking** (class hierarchy, inter-connection, abstraction, etc...) the **exact same way regardless** of what you'll choose

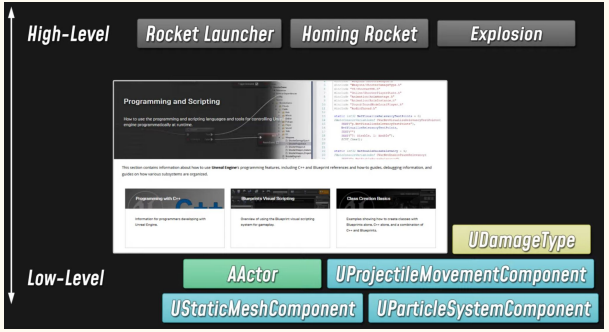
Design concepts

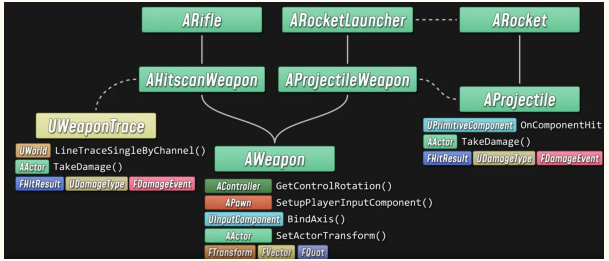
- A video game is a **complex piece of software**, maybe one of the **more complicated** in the industry of programming
  - It needs to be easily **tweakable**, **extendable**
  - It is **maintains** for **years** most of the time
  - It **involves** a lot of **collective works**
  - Team are **most likely big** which complexify everything

- When you think about design, it is helpful to think vertically
  - High-Level : Concrete **inter-connection** between systems, **usage of complexe low level** system, etc...
  - Low-Level : Creating **foundations** for systems to **expand** on and being **used in high level**

- It is our **job** as a **programmer** to take every idea from game design, and think of a way to **organise** the **structures**, **datas** and systems to be **extensible** and **maintainable** as **easy as possible**.



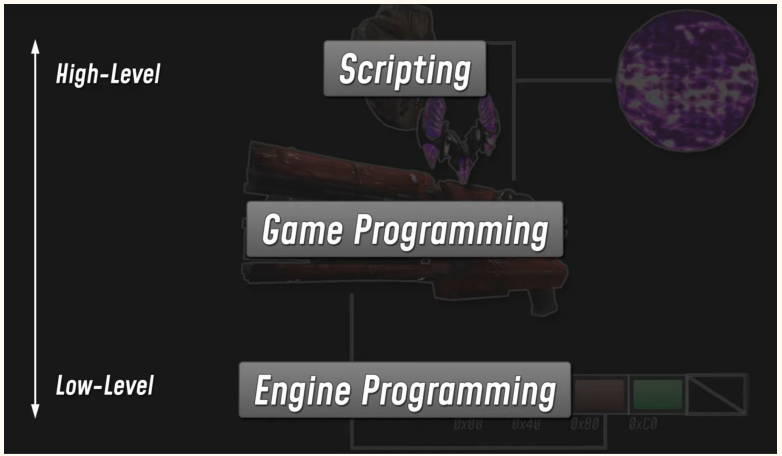




Design concepts - Different level of programming

- We are separate programming domains as 3 different categories
  - Engine Programming
    - How to get the **memory** I need ?
    - How do I **render** a 3D Object ?
    - **Core technologies** that allows to build a **game**, **regardless** of the actual game type, ideas, etc...
  - Game Programming
    - How do I setup a weapon system for my shooter game ?
    - How do I set up inventory system for my ARPG game ?
    - Here, you are **using** and **building** upon the **core technologies** to solve **problem fundamentals** of the **specific game** you are making
  - Scripting
    - **Overall flow** and progression of the game
    - **High-level** interaction between game objects
    - Build **upon** the **systems** for the minute to minute experience of the game

- This is obviously a generic division, in reality we may end up to a 2 division : **Scripting** and **programming**

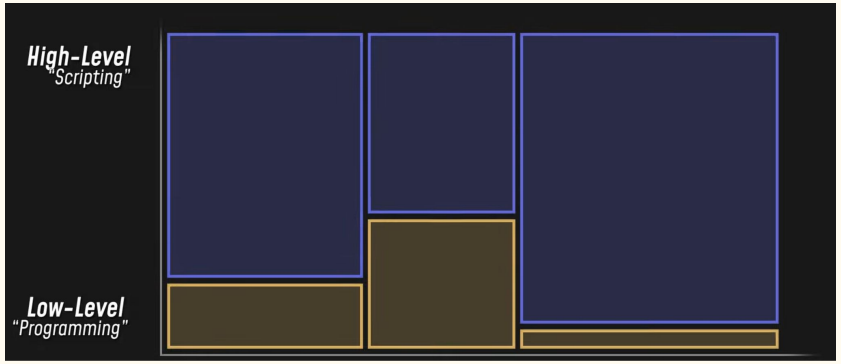


Scripting or programming ?

- Let's try to get back on the concept of **C++** and **blueprint** in the **unreal context** of **programming**
  - C++ is a **programming language**
    - Better suited to implement **low level system**
    - It is **out of reach** of **designers** most of the time which may also be a **desire thing** in order to **hide complexity** from them
  - Blueprint is a **scripting language**
    - Naturally better suited for **defining high level behaviors** and **interactions**
    - Integrating assets
    - Fine-tuning cosmetic details

- **Typically** in an unreal project, there is C++ and blueprint
  - Engine is **not there to define how to use them**
  - You may **decide** to use **more heavily** blueprint for **some system**, or for **prototyping** reason

- Unreal has an **API Parity** between C++ and blueprint
  - You are using the **same underlying system** in much the same way
- Unreal is made to make C++ and blueprint have an easy interoperability



Differences between C++ & Blueprint - Clarification

- Let's **clarify** something before digging into some differences between C++ and blueprint

- We'll here speak about **gameplay programming**
- In a wider context of developing an **Unreal project**, there are **plenty** of cases where blueprint are **not a suitable option.** Blueprint is primarily **design** to **write game code**.
- The exact boundaries are sometimes **hard** to defined and you'll sometimes learn it the hard way by seeing there is no way to do what you want in blueprint

- This is some examples
  - Adding new modes to the editor
  - Developing standalone tools
  - Creating subsystems
  - Inheriting from some classes
  - Integrating external libraries
  - Adding custom rendering code
  - Accessing some low-level stuff
  - Etc...

Differences between C++ & Blueprint - Performances

- Probably one of the main concern, let's see what is happening performance wise

- When you **write** a function in **c++**, it end up with **plain text** in a .cpp file. **(.cpp)**
- When you **write** in **blueprint**, you end up in a **bunch of nodes** stored in a **Blueprint asset (.uasset)**

- When you build your project from source
  - The C++ function get compiled to **machine code**
    - It becomes flat listing of **processor instructions** in binary code running **directly on the CPU**
  - The blueprint function get compiled to but **not to machine code**
    - It becomes a **script bytecode** which is a portable intermediate form of your function
    - The **engine's Script VM** will execute it at runtime

Differences between C++ & Blueprint - Performances

- The main problem performance wise lies **into the VM**.
- VM is just a **piece of software engineering** that keep tracks of **callstacks** and ensure to call functions with right parameters.
- Unreal internally ensure to create a **bridge between API and native c++ code** with **helper** functions.
    - Indeed, blueprint node calls are **actually calling some native c++ code** under the hood
    - This interfacing between blueprint and c++ creates an **overload** of **CPU instruction** being run which can get bigger and bigger on big functions

- BUT this **observation** needs to be **taken with care**
    - It is important to take into **consideration** the **context** of the execution of that function
        - If the function is called **once per frame** at a 16 milliseconds frame delta, it is **insignificant**. You may actually end up in **creating** a complex c++ code to maintain for **no benefit**
        - If 1000 actors runs this code every frame, then the overhead might because an **actual issue**

Differences between C++ & Blueprint - Performances

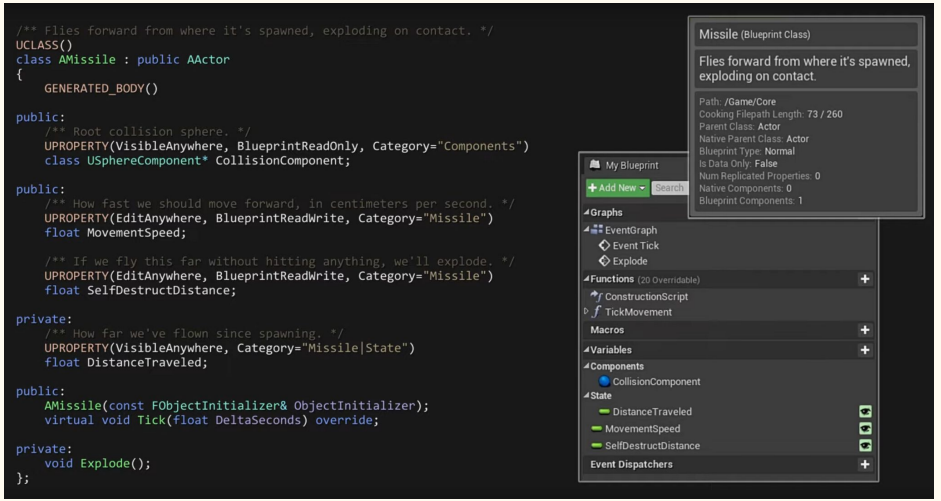
- To conclude about **performance**, while creating a game, you can **reasonably predict** about where script overhead is most likely to be a problem
  - In this case, you may consider moving stuff into C++
  - **Low-level system**
  - **Processing** a lot of **datas** (big array, search, complex lookups, etc...)
  - **Tight loops**
  - Classes with **lof** of **instances**

- Always think about Pareto principle or 80/20 rules
  - **80/20 rules** : 80% of execution time is spent running 20% of the code
- Think about using a **profiler** to see where every millisecond of the frame is going and based your optimization decision only on that data and **not on speculation**.

Differences between C++ & Blueprint - Design decision

- We'll not go through **programming design decision** in this lesson has it is **not in scope** and suppose to be **already known**
- But the idea is still important, programmers create **classes**, and design the interface on that class with **private** and **public accessibility**, the methods it offers, etc...

- Both **C++** and **blueprint** allows to do the job quite equivalently in the **definition** itself of the **classes** and **responsibilities** of that class.
- First difference between the two are **dependencies** management
    - Dependency is the **process** of having **various classes**, **enums**, **structs**, etc.. **interacting** with each other by **knowing** each others
    - When the project **grows** bigger and bigger, **dependencies** become **more and more complex** and may cause the project to be less and **less maintainable**.

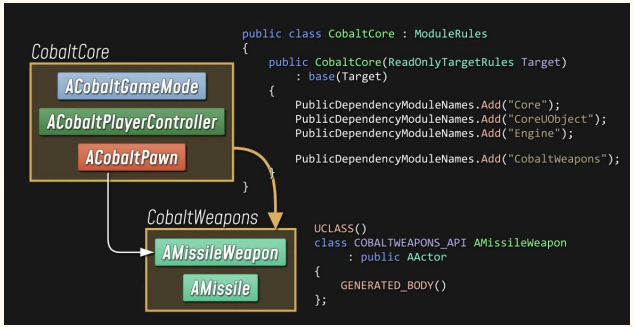


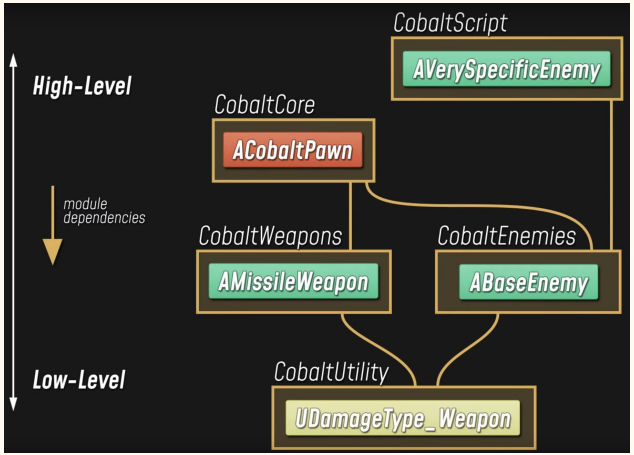
Differences between C++ & Blueprint - Dependencies in C++

- A way to handle dependencies in C++ is to use **modules**
- Your core game mechanismes are located in a **simple core module** contains the **core** classes like **Pawn**, **Controller**, **GameMode**, etc...

- When the project grows, it is possible to **separate** the systems in **different modules**
  - In the project module, it is then possible to add **dependencies** on **modules**
  - Module dependency is **supposed** to be always **strictly one way**. It naturally leads to a **layered architecture**

- Doing this separation in module will
  - **Ensure** that while you are coding in C++, you'll **not be able to reference some element** in a module that in not following that **layered architecture** and **dependency decision** you made. It promotes **separation of concerns**
  - **Speed up builds** by only relinking what's changed
  - Facilitates **organisation** and **ownership** of modules

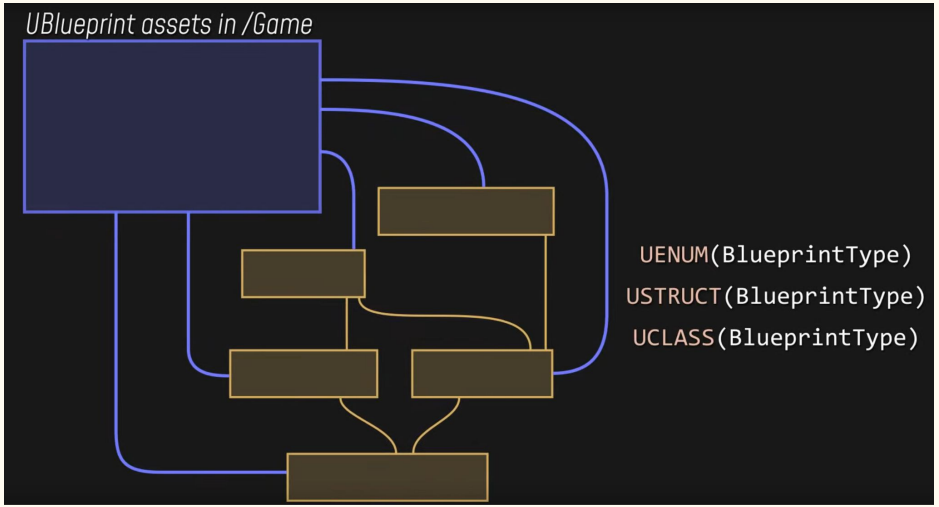




Differences between C++ & Blueprint - Dependencies in Blueprint

- There is **no modules concept** in Blueprint
- When you creates stuff in blueprint, you can consider that it all goes into a **“Game”** module which is accessible by **anything**, **anywhere** in blueprint, and can **reference** as they **want source module** when they are blueprint type (**UENUM**, **UCLASS**, **USTRUCT**).

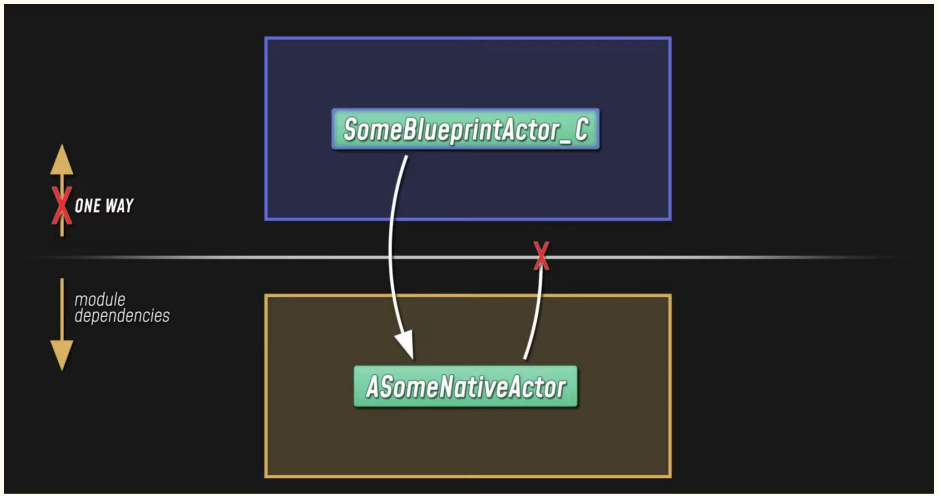
- It will naturally **leads** to a **bigger chance** for making system **tightly coupled**
- **Reference viewer** of Unreal is a **great tool** to understand how coupled your blueprint assets are



Differences between C++ & Blueprint - One way dependency

- Probably one of the **most important** thing to keep in mind while **designing** your **architecture** and **choosing** between C++ or Blueprint for a **system or functionality** is **One way dependency**

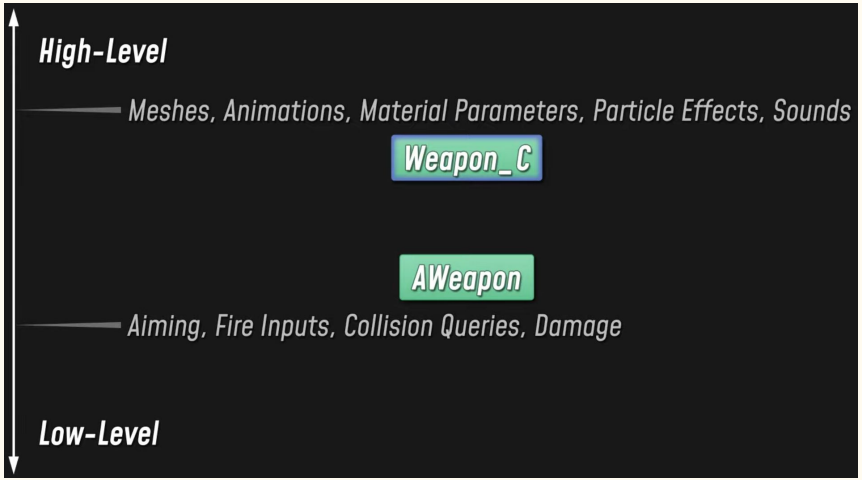
- It is **possible** in blueprint to reference, inherit and use stuff from c++ source modules.
- It is **impossible** in c++ to reference, inherit or access stuff from blueprint



Making both world works together

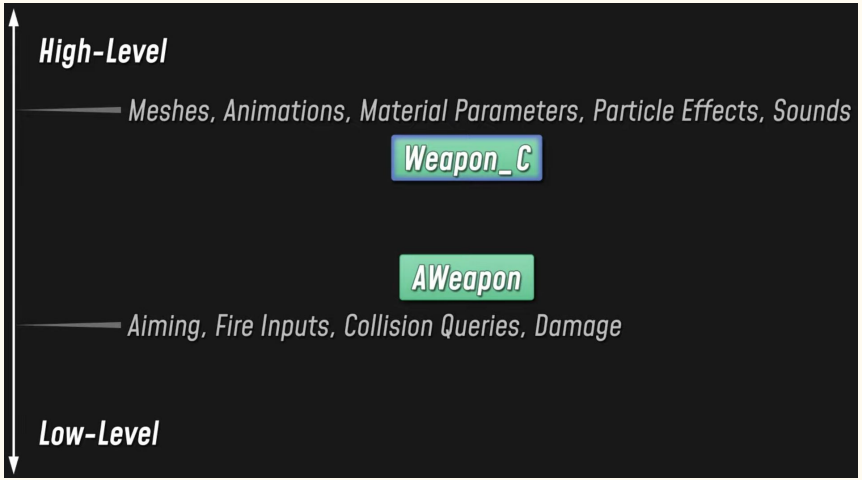
- The following is an example as there is **10 ways to solve 1 problem**
- We know that **high-level stuff** and fine **tuning** are more easily **doable** and **tweakable** in **blueprint**
  - It also **allows designer** and **artists** to **configure** as they want stuff
  - They can even **slightly modify** rendering behavior to suits their needs

- A good approach may be
  - Create low level ground stuff of a feature in c++.
    - In may even not care at all about **visual aspect** of a weapon, but just **ensure** to shoot when an input is received
  - Create high level implementation details and visual customization into blueprint
    - You'll **inherit** from the **c++ class** as a blueprint class and have access to think like **BlueprintImplementableEvent function**, **event**, etc...
  - It create a layered architecture with an higher level blueprint based on a lower level c++



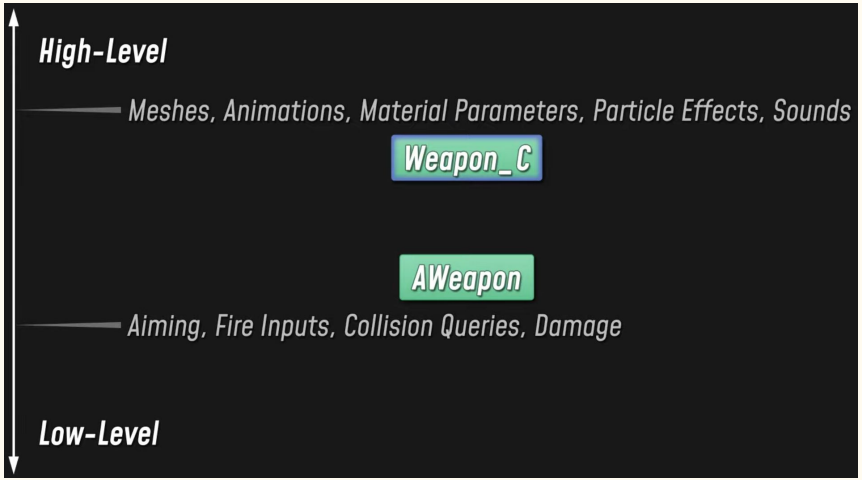
Wrap up - Advantages of blueprint

- **Assets and visuals**
  - Blueprint are better to deal with assets and visual effects thanks to viewport.
  - Blueprint being asset, they are easy access to any asset.
- **Scripted behaviors**
  - You have easy access thanks to event graph to **event**, **asynchronous function**, **sequencers**, **timelines**, etc...
  - Ease the process of create high level behavior
- **Iteration speed**
  - Blueprint allows to **easy** and **fastly** test and **iterate** over.
  - Blueprint directly takes **place** in the **editor** and have an native **debugger** running with it
- **Accessibility**
  - It allows people with **no** or **limited knowledge** in C++ to actually **code** and **tweak** the **gameplay**.
  - Useful for **designers** or **artists** for example



Wrap up - Advantages of c++

- **Runtime performances**
  - C++ code can be fully optimize for platforms it going to run on with no overhead from blueprint
- **Fundamental code**
  - There is a reason to put some fundamental code in c++
  - System that are meant to be really performant, only modifiable by programmers
  - More readable and maintainable for complex system
  - It allows to hide complexity for user of the system
- **Engine functionality**
  - There is a lot to say, but you can take full advantage of **logging system**, use **asserts**, use **custom stats categories** to tracks, use **raw TCP and UDP** socket and **communicate** with **web API**, customize **serialization**, create **editor modules**, use **engine delegate**
- **External Libraries**
  - Extremely important to import some **API** from **console developers**
- **Diffing and Merging**
  - Last but not least, blueprint are asset so binaries file impossible to merge when c++ is just plain text.



C++

- Let's try to highlight **key concept** when you are developing in c++
- It is not a **lesson** about **programming in c++**, it is something you are supposed to know already
- We'll check mainly what is different between a **standard c++** class and a C++ though in **Unreal way**
- We'll mainly see in the following slides **Metadata Specifiers** that Unreal provides and **allows** to make our **c++ interact** with **blueprint** and the **engine**
- An important point to understand is that **metadata specifiers** only exists in the **editor.** You should **not write** any code that access this **metadata**
- Here's the list of metadata that you may use
    - UCLASS
    - UENUM
    - UINTERFACE
    - USTRUCT
    - UFUNCTION
    - UPROPERTY
- We'll obviously not see each **metadata** that can be added in an **Metadata Specifier** as it would offer no much value, **documentation** is here for that

UPROPERTY

- If you want to have **detail** about a specifier, knows how it works etc... You can consult Unreal Documentation. Take care as **Unreal** Documentation is **not exhaustive**, for a more complete one, you can check BenUI page
- Let's now see most common specifier
  - **BlueprintReadOnly** : This property can be **read** by **Blueprints**, but **not modified**.
  - **Blueprint ReadWrite** : This property can be **read** or **written** from a **Blueprint**.
  - **Visible Anywhere** : Indicates that this property is **visible** in all **property windows**, but **cannot** be **edited**.
  - **VisibleDefaultsOnly** : Indicates that this property is **only visible** in property windows for **archetypes**, and **cannot** be **edited**.
  - **VisibleInstanceOnly** : Indicates that this property is only **visible** in property **windows** for **instances**, not for **archetypes**, and **cannot** be **edited**.
  - **EditAnywhere** : Indicates that this property can be **edited** by **property windows**, on **archetypes** and **instances**.
  - **EditInline** : Allows the user to edit the properties of the **Object** referenced by this **property** within Unreal Editor's property inspector
  - **Instanced** : Object (UCLASS) properties only. When an instance of this class is created, it will be given a unique copy of the Object assigned to this property in defaults. Used for instancing subobjects defined in class default properties. Implies **EditInline** and **Export**.

```
UPROPERTY(EditAnywhere, BlueprintReadOnly, Category = "General")
FGameplayTag Identifier;
```

UFUNCTION

- If you want to have **detail** about a specifier, knows how it works etc... You can consult Unreal Documentation. Take care as **Unreal** Documentation is **not exhaustive**, for a more complete one, you can check BenUI page
- Let's now see most common specifier
    - **BlueprintCallable** : The function can be **executed** in a **Blueprint** or Level Blueprint graph.
    - **BlueprintImplementableEvent** : The function can be **implemented** in a **Blueprint** or Level Blueprint graph.
    - **BlueprintNativeEvent** : This function is designed to be **overridden** by a **Blueprint**, but also has a **default native** implementation. Declares an additional function named the same as the main function, but with **_Implementation** added to the end, which is where code should be written. The **autogenerated** code will call the _Implementation method **if** no Blueprint override is found.
    - **BlueprintPure** : The function does not affect the owning object in any way and can be executed in a Blueprint or Level Blueprint graph.

```
UFUNCTION(BlueprintCallable, Category = "Ares|Movement")
void WalkTo(FVector Destination, float ForcedSpeed = 0.f);
```

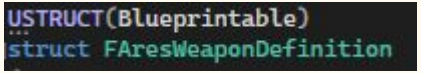
UCLASS

- If you want to have **detail** about a specifier, knows how it works etc... You can consult Unreal Documentation. Take care as **Unreal** Documentation is **not exhaustive**, for a more complete one, you can check BenUI page
- Let's now see most common specifier
    - **Abstract** : The **Abstract** Specifier declares the class as an "abstract base class", preventing the user from adding Actors of this class to Levels.
    - **Blueprintable** : Exposes this class as an acceptable base class for creating Blueprints. The default is **NotBlueprintable**, unless inherited otherwise. This Specifier is inherited by subclasses.
    - **BlueprintType** : Exposes this class as a type that can be used for **variables** in Blueprints.
    - **EditInlineNew** : Indicates that Objects of this class can be **created** from the Unreal Editor **Property window**, as opposed to being referenced from an existing Asset. The default behavior is that only references to existing **Objects** may be assigned through the Property window). This Specifier is propagated to all child classes; child classes can override this with the **NotEditInlineNew** Specifier.

```
UCLASS()
class ENGINE_API UCharacterMovementComponent : public UPawnMovementComponent, public IRVOAvoidanceInterface, public INetworkPredictionInterface
```

USTRUCT

- If you want to have **detail** about a specifier, knows how it works etc... You can consult Unreal Documentation. Take care as **Unreal** Documentation is **not exhaustive**, for a more complete one, you can check BenUI page
- Let's now see most common specifier
    - **BlueprintType** : Exposes this struct as a type that can be used for variables in Blueprints.



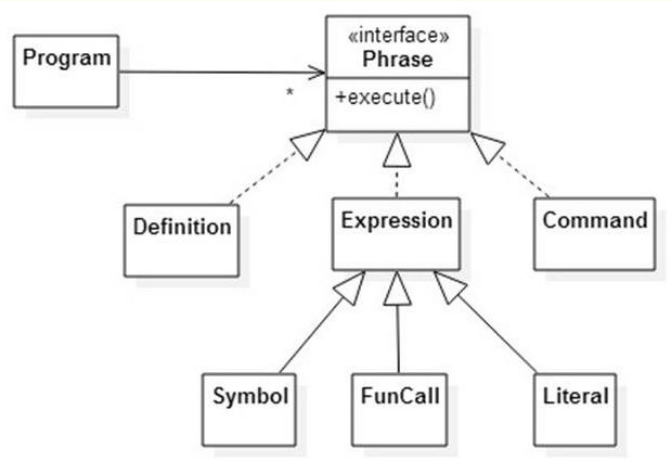
UENUM

- If you want to have **detail** about a specifier, knows how it works etc... You can consult Unreal Documentation. Take care as **Unreal** Documentation is **not exhaustive**, for a more complete one, you can check BenUI page
- Let's now see most common specifier
    - **BlueprintType** : Exposes this struct as a type that can be used for variables in Blueprints.

```
UENUM(Blueprintable)
enum class EAresSkillEffectType : uint8
```

UINTERFACE

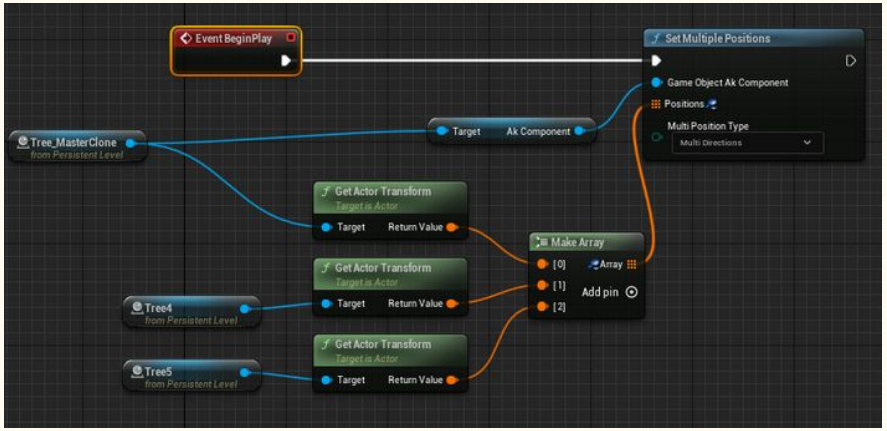
- Interface are a bit specific to implement in Unreal, in order to have more details about it, check this Unreal Documentation
- To declare an Interface, we need a macro UINTERFACE() on top of the interface and inheriting from **UInterface**
- **The class you define with above state is not your actual interface. It is needed for reflection system of unreal but naming is important.** Notice that you name the class starting with a **U** like every UObject.
- The actual interface **must** be named like the declaration above, but replace **U** with an **I**




```cpp
#pragma once

#include "ReactToTriggerInterface.generated.h"

UINTERFACE(MinimalAPI, Blueprintable)
class UReactToTriggerInterface : public UInterface
{
    GENERATED_BODY()
};

class IReactToTriggerInterface
{
    GENERATED_BODY()

public:
    /** Add interface function declarations here */
};
```

Blueprint

- **Blueprint** being way **more visual** than C++, we'll highlight key concept by doing an **interactive demonstration** directly in the engine
- Globally speaking, you'll don't have to **worry** about all **macros** etc... which allow **editor** to **interact** with c++ as you'll be writing blueprint which is **directly** in the engine
- You'll have access to **everything** needed but have **limitation** about the **specifier** which are not directly available, mainly **metadata** which offer **cool feature** like hidden a variable in blueprint based on condition



Time to.... highlight a concept

Communication is key

Practice

- **General**
  - Create a class inheriting from Actor called ATestingLesson2
    - Create some variables with UPROPERTY
    - Create some function with UFUNCTION
    - Create an enum with ENUM which is a field in ATestingLesson2
    - Create an interface which is implemented by ATestingLesson2
      - Create a Dummy function in it to be implemented by subclasses
  - Create a Blueprint class which implement the C++ interface
  - Create a class inheriting from UObject and make it EditInlineNew
    - Modify the data asset you create in C++, create a field of the UObject above which is Instanced
    - Create a blueprint class inheriting from your UObject c++ class
    - See the result in the DA based on the C++ data asset
  - Generally, play with all specifiers in order to understand them, primarily the visibility / edit one, function specifiers, etc...

- **Follow-through project**
  - Think design-wise how you would like to separate your project into a C++ / Blueprint modular model
  - Modify any already created stuff to match that design
  - Extend your Character in order to manage parametrization of movement speed, and differentiating logic behavior which may be implementation detail in C++ and cosmetic and fine tuning which should be in Blueprint
  - Create your structure for an NPC, think your global architecture... Common behavior between an NPC and Player, think about Controllers, etc...