Introduction

(NB : J'espère que ce tuto vous sera utile et qu'il sera compréhensible par tous. Si vous avez des questions n'hésitez pas à m'envoyer un message sur Facebook « Antoine Gargasson » ou par mail antoine.gargasson@gmail.com)

(NB2 : Ce tuto est réalisé sur la version 4.8.3 de UE4)

Dans ce tuto 0, je vais expliquer les bases du C++ sur UE4 qui reprendra beaucoup de notions de C++.

Je vous conseil également de lire **Tips et Conseils UE4** . Ce document traite des conseils, des problèmes que j'ai rencontré et comment les résoudres, de petites choses à copier/coller dans son code pour plus de rapidité (vous comprendrez plus tard pourquoi).
Il est disponible ici : http://antoinegargasson.allyou.net/5584503

Ce tuto s'appuiera sur beaucoup d'anglais également. Je suis désolé mais c'est un passage obligé quand on veut faire de la programmation. Je ne veux surtout pas effrayer les personnes allergiques à l'anglais. Simplement mes **fonctions**, **variables** et **commentaires dans le code** seront en **anglais**. Toutes mes explications seront, elles, en **français intégrale**.

Je ne peux que vous recommander la **documentation Unreal Engine** (en **anglais**) pour les fonctions, variables et autres précisions qui m'ont déjà été très utiles.
Disponible ici : https://docs.unrealengine.com/latest/INT/

Dans ce tuto je mettrai les choses importantes en **Gras**.

```
//Ceci est du code
Le code dans un cadre et dans une police différente.
```

Une fois encore je précise que je ne possède pas la science infuse, que je peux me tromper, que le moteur UE4 est jeune et en constante évolution. Je fais ces tutos pour le plaisir et surtout pour rendre service aux personnes qui pourront apprendre grâce à ces documents. Il se peut que j'oublie des choses ou que certaines ne soient pas clair. N'hésitez pas à poser des questions ou à demander plus de détail sur une chose que vous ne comprenez pas.

Et encore désolé pour les éventuelles fautes et le manque de couleur parfois évident, je ne suis qu'un programmeur ;)

Plus d'informations ici : www.cplusplus.com

Antoine Gargasson Août 2015

Les Bases

Les types primitifs sont des types de variables que vous utiliserez en permanence et qui doivent être connu **obligatoirement** quand vous codez. Ne vous en faites pas, ça viendra en codant !

Il y a les types primitifs **communs** à tous les langages et ceux **ajoutés** par Unreal pour pouvoir utiliser leur moteur.

Voici les **types primitifs de base** (et leur nom sur UE4):

float ; int(int32) ; string(FString) ; bool ; char ; double ; void ; long ;

Et voici d'autres types que vous utiliserez :

FName ; UClass* ; USoundCue* ; UTexture2D* ; UTexture* ; AActor* ; TSubclassof<class> ; TArray<class> ; USkeletalMeshComponent* ; UCameraComponent*

Cette liste est **exhaustive**. C'est un extrait des **types** que j'ai pu utiliser jusqu'à maintenant.

Lorsque vous voulez écrire quelque chose en C++, vous devez l'écrire dans une C**lasse**. En C++, une classe est représenté par **2 fichiers**. Un fichier **.h** qui est appelé fichier d'en-tête (**header** en anglais) et un fichier **.cpp** qui est le fichier où vous aller écrire la majorité de votre code.

En c++ quand vous voulez ajouter une **variable** ou une **fonction**, vous devez d'abord le **déclarer** dans le **.h** et ensuite le **définir** dans le **.cpp**.

Nommage

Comme nous sommes sur Unreal Engine, il paraît normal de voir des propriétés de nommage. Ainsi quand vous voudrez écrire une fonction, vous l'écrirez de la manière suivant :

```
UFUNCTION()
void Fonction() ;
```

UFUNCTION Signifie au compilateur et donc à UE4 que la fonction qui suit doit être compilée et ajoutée au logiciel.
Retenez simplement que en mettant UFUNCTION() vous aurez accès à cette fonction dans les blueprint.

De même pour une variable :

```
UPROPERTY(Propriété1, Propriété2, Propriété3)
TypeDeVariable NomVariable;
```

En mettant UPROPERTY(...) vous aurez accès à cette fonction dans les blueprint <u>suivant certaines conditions</u>.

Les variables

Comme je le disais plus haut, pour utiliser une variable il faut utiliser ça :

```
UPROPERTY(Propriété1, Propriété2, Propriété3)
TypeDeVariable NomVariable;
```

Si vous ne mettez pas **UPROPERTY**(…) votre variable sera **utilisable** mais **uniquement** en c++, il en va de même pour les fonctions.

Voici un exemple de variable :

```
UPROPERTY(VisibleAnywhere, BlueprintReadOnly, Category=Camera)
UCameraComponent* FirstPersonCameraComponent;
```

- Une variable peut être **VisibleAnywhere** (visible dans les defaults et in game) ou **VisibleDefaultsOnly** (visible dans les defaults uniquement) ou **EditAnywhere** (éditable n'importe où) ou **EditDefaultsOnly** (éditable uniquement dans les défaults)

- Une variable peut être **BlueprintReadOnly** (uniquement lisible(**get**) dans les blueprints) ou **BlueprintReadWrite** (lisible(**get**) et scriptable(**set**) dans les blueprints)

- La catégorie est un nom que vous donnerez et qui servira à retrouver votre variable dans les defaults.

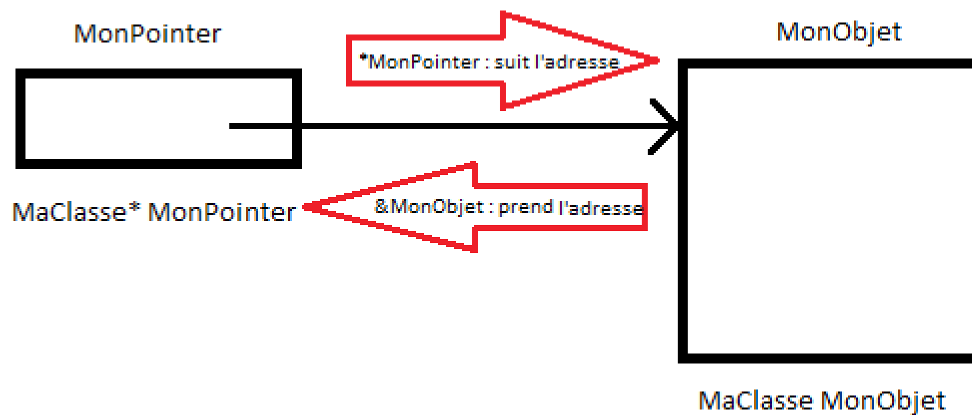
Le **cadre** ci-dessus doit être écrit dans le **.h** . Dans le .cpp vous allez pouvoir utiliser cette variable comme bon vous semble en utilisant simplement son nom **FirstPersonCameraComponent** .

Parlons maintenant de la petite étoile à côté du type **(*)**. Cette étoile signifie que on va utiliser un **pointer** sur le type d'objet.

Qu'est ce qu'un **pointer** ? Me direz vous. Un pointer c'est un **lien, une adresse** vers une **zone mémoire**. Cette **zone** va contenir des **données** que l'on souhaite utiliser. Bien entendu il est préférable que ces données soient **créées par vous** et que vous sachiez ce qu'elles **contiennent**.

Vous pouvez manipuler dans les deux sens. D'un côté, si vous avez un **objet** vous pouvez récupérer **l'adresse** du pointer. D'un autre côté, si vous avez le **pointer** vous pouvez récupérer **l'objet** en question.

Le but d'un **pointer** est **d'économiser** de la place et de pouvoir **réutiliser** un **objet**. Parfois quand vous utilisez un objet directement en **paramètre** d'une **fonction**, le processeur va **recopier** cet objet alors que vous voulez juste une variable sur l'objet. Dans ce cas vous utiliserez un **pointer** qui sera beaucoup **moins lourd** pour votre ordinateur.



Voici un schéma qui récapitule ce qu'est un **pointer**. Ici le pointer adresse l'objet **MonObjet**. Vous remarquez qu'il faut que le **pointer** et **l'objet** est la **même classe**.

Une autre chose importante, quand vous souhaitez utiliser une méthode ou une variable d'un pointeur, nous n'utilisez pas le . mais la flèche ->

Maintenant que vous connaissez la théorie (et c'était **très important**), voyons l'utilisation pratique sur UE4.

Dans UE4, quand vous utilisez un pointer, vous allez **référencer** un **objet** qui est dans le **monde** (un acteur ou le joueur par exemple). Ainsi vous avez pu voir dans les **blueprint** que le type des objets bleu est **Actor Reference**, ce n'est pas pour rien, ce sont des **pointers** d'objets.

Unreal vous facilite la vie en vous **indiquant** quels types sont des pointers et quels type ne le sont pas (Cf : les types primitifs). Bien sûr vous être libre de **transformer** un non pointer en pointer si vous le souhaitez, mais pas **l'inverse** !

Vous trouverez deux **portée** de variables. La première est la portée **globale**. Cela signifie que votre variable est accessible dans toute votre classe sans exceptions. Ce sont les variables **déclarées** dans le **.h** . La seconde est la portée **locale**. Cela signifie que votre variable est accessible **uniquement** à l'endroit où elle est déclarée (structure ou fonction). C'est le plus **souvent** dans le **.cpp**.

Il existe également 3 **visibilités** à une variable (s'applique aussi aux fonctions). Une variable peut être **public**, **private** ou **protected**.
Une variable **public** peut être utilisée par **d'autres classes** (l'oeil en blueprint).
Une variable **private** est **uniquement** accessible dans la **classe** où elle est **déclarée**.
Une variable **protected** est accessible **uniquement** par les **classes enfants** de la classe qui

la définie.

Pour finir, voici quelques **variables utiles** :

```
// Appelle la fonction Tick() sur ce character à chaque frames.
Si vous n'en avez pas besoin, vous pouvez le désactiver pour
améliorer les performance en le mettant à false.
PrimaryActorTick.bCanEverTick = true;

//Autorise le contrôle de la rotation du pawn
FirstPersonCameraComponent->bUsePawnControlRotation = true;

//Seulement le joueur possédant le pawn verra le mesh
FirstPersonMesh->SetOnlyOwnerSee(true);

//n'utilise pas les ombres pour ce modèle
FirstPersonMesh->bCastDynamicShadow = false;
FirstPersonMesh->CastShadow = false;

//le joueur qui possède ne voit pas le modèle
Mesh->SetOwnerNoSee(true);
```

Les fonctions

Les fonctions sont un peu plus facile à utiliser que les variables.
Lorsque vous **déclarez** une fonction dans le **.h**, vous écrirez ceci :

```
UFUNCTION()
void Fonction() ;
```

Vous le savez maintenant, UFUNCTION() permet de dire à UE4 que votre fonction doit être incluse dans les blueprint.

Tout comme les variables, les fonctions doivent être typées. Ce type sera le type de retour.

Un exemple :

```
int32 GetAge() ;
```

Ici la fonction **GetAge**() **devra** me retourner un **nombre.** Si vous ne le faites pas, vous allez avoir une **erreur de compilation**.

Le type de retour peut être **n'importe** quel **type primitif**.

Une fois votre fonction **déclarée**, vous allez devoir la **définir**. C'est dans le **.cpp** que ça va se passer.

Reprenons notre exemple. Voici la fonction telle qu'elle pourrait être écrite :

```
int32 MaClasse::GetAge()
{
	int32 ValeurRetour  = 0 ;
	ValeurRetour  = 3 + 2 ;
	return ValeurRetour ;
}
```

Qu'est ce que cette fonction nous dit ?

Tout d'abord il faut reprendre la ligne du .h qui est la suivante :

```
int32 GetAge() ;
```

Pour que votre fonction soit **affectée** à votre classe, dans le **.cpp** vous devez **rajoutez** le nom de la **classe**. Vous devez le rajouter entre le **type** et le **nom** de la fonction. Et ça donne :

```
int32 MaClasse::GetAge()
```

Votre fonction commence bien telle quel.

Une fonction **exécute** des **actions**. Ces actions doivent être placées entre **accolades** { } .
Pour une meilleure **lisibilité**, placez les **à la ligne** tout le temps.

Ensuite nous avons ceci :

```
1.     int32 ValeurRetour  = 0 ;
2.     ValeurRetour  = 3 + 2 ;
3.     return ValeurRetour ;
```

Sur la première ligne(1.) on créé une **variable locale** de type int32 (un entier) et on lui affecte la valeur 0.
Sur la seconde ligne(2.) on effectue un **calcul** (3+2) et on **l'affecte** à la variable locale.
Sur la troisième ligne(3.) on retourne la **variable locale**.

Une fois passé dans le **return**, on **sort** de la fonction, même s'il y a des choses écrites après.

Vous pourrez par exemple utilisez le **return** pour sortir de la fonction dans un cas non désiré.

```
if ( 3 > 2 )
{
   return ;
}
else
{
   //vous êtes sur de savoir compter ?
}
```

Ici on test si 3 est supérieur à 2 et si c'est le cas on sort de la fonction.

Il existe un **type** particulier qui est le type **void**. Ce type (vide en anglais) signifie que votre fonction ne retourne rien.

Un exemple :

```
void MaClasse::SetAge()
{
     Age = 30 ;
}
```

Ici on ne renvoi **aucune** donnée, le type est donc **void**.

Avec ces deux exemples vous venez de découvrir deux grands **types de fonction** : les **getter** et les **setter**. Le premier sert à **récupérer des données** et sera **toujours typé,** le second sert à **affecter des données** et sera toujours **void**.

Pour finir voici quelques fonctions utiles que vous serez amené à utiliser souvent :

```
//utilisée en début de jeu
virtual void BeginPlay() override;

//utilisée à chaque frames
virtual void Tick( float DeltaSeconds ) override;

//!!utilisée dans les classes qui hérite de AHUD!!
virtual void DrawHUD() override;

//utilisée pour détecter une collision
void OnHit(class AActor* OtherActor, class
UPrimitiveComponent* OtherComp, FVector NormalImpulse, const
FHitResult& Hit);
```

Il existe également une **fonction spéciale** qui se nomme le **constructeur**.
Le constructeur d'une classe sert à **définir des variables** et **créer les bases** de votre objets (en rajoutant des composants par exemple).
Il existe une fonction spéciale dans UE4 qui sert à **initialiser** les variables, rien de nous empêche de le faire dans cette dernière, en revanche l'ajout de composant reste dans le **constructeur** alors autant tout mettre au **même endroit** ;)

Il est **quasi obligatoire** de **déclarer un constructeur** dans chaque classe (je vous encourage à le faire à chaque fois même si il reste vide).
Pour déclarer un constructeur dans une classe, vous devez mettre une visibilité public et écrire comme suit :

```
//dans le .h
AFPSProjectile(const FObjectInitializer& ObjectInitializer);
```

Une fois **déclaré** dans le **.h**, vous devez le **définir** dans le **.cpp**.

```
//dans le .cpp
AFPSProjectile::AFPSProjectile(const FObjectInitializer&
ObjectInitializer) : Super(ObjectInitializer)
{
	...
}
```

Vous aurez remarqué que le constructeur ne prend pas de type, c'est **le seul cas** où une fonction ne prend pas de type. Quand vous devrez faire un constructeur, copiez collez ceci et remplacez `AFPSProjectile` par le nom de votre classe.

Vous avez aussi dû faire attention au `Super(ObjectInitializer)` . C'est une fonction très spéciale que vous verrez **très souvent**. Cela signifie que l'on fait **appel** à la classe mère et que donc nous sommes dans une **classe enfant**.

Cet appel soit être fait quand vous utilisez un **override**.
Qu'est-ce qu'un **override** ?

Un **override** c'est une réécriture (traduit de l'anglais toujours) d'une fonction. Vous **devez** mettre le **Super**. Il permet de faire fonctionner les choses **essentielles** de votre fonction.

Mettons que dans votre classe mère vous avez une fonction qui calcul 3 + 3 et qui l'affecte à une variable. Ça ne vous plaît pas et vous voulez que, dans une classe fille, cette variable est le calcul 3 + 3 mais que vous voulez rajoutez 2. Vous utiliserez le **Super** de votre fonction pour faire d'abord le calcul et puis ajouter 2.

Enfin **Virtual** signifie simplement que votre fonction doit forcément être **réécrite** dans la classe fille.

Les boucles

Revenons sur du basique avec les **boucles**.
Les boucles servent à effectuer des actions à **répétition**.

Pour cela on utilise le mot clé **for**.

Un exemple d'utilisation :

```
int32 increment = 0 ;
for (int32 i = 0 ; i < 10 ; i++)
{
	increment ++ ;
}
```

La première ligne **affecte 0** à une variable **increment**.
La seconde est la **définition** de la **boucle**. On doit **toujours** utiliser une **variable** pour effectuer une **boucle**. Ici on **commence** la boucle à **0**, on la fini quand on arrive juste **avant** 10 et on utilise un **pas** de 1. Cette boucle va donc s'exécuter **9** fois. A la sortie de la boucle, increment vaudra donc 9.

increment ++ est le diminutif de **increment = increment + 1** , ne vous avais-je pas dit que les programmeurs étaient fainéant ? ;)

De même on a les boucles while.

While (tant que), attention cette boucle peut s'effectuer à **l'infini** si vous oubliez **d'incrémenter** la variable de **paramètre** (ici i). Increment vraudra 18 à la fin.

```
int32 increment = 0 ;
int32 i = 0 ;
while(i < 10)
{
	i++ ;
	increment= increment + 2 ;
}
```

Les constantes

Les constantes const, define

Lorsqu'on parle de **constante** il faut avoir en tête une **donnée inchangée**. Lorsque l'on écrit

```
Const int32 var = 6 ;
```

C'est qu'on indique au **compilateur** que cette donnée restera **inchangée**. Ainsi il y a des données comme PI qui sont constantes.

On peut également utiliser les **define**. Un **define** est une **macro** qui est lu par le **préprocesseur**. Lorsque le prépocesseur va lire cette macro, il va remplacer cette macro par sa valeur. Seul bémol, le type de la variable n'est pas vérifié et cela peut ralentir le programme.

```
#define PI 3.14159
```

Le mot clé **define** doit être **précédé** d'un **#** . Généralement on l'écrit **avant** le début de la classe et après les **#include**.

Les opérations

Les **opérations** sont les symboles que vous allez sûrement **le plus** utiliser quand vous allez programmer.

Commençons par les **basiques**. Vous pouvez utiliser tous ces symboles pour **calculer** :

+ (plus), - (moins), * (multiplier), / (diviser), % (modulo)

1+1=2
1-1=0
2*2=4
2/2=1
10%3 = 1 (reste de la division)

Vous pouvez aussi effectuer des **opérations rapides** :

variable += 3 ; (équivaut à : variable = variable + 3;)
variable -= 2 ;
variable *= 2 ;
variable /= 3 ;
variable %= 3 ;

Vous pouvez ensuite utiliser des symboles **d'incrémentation** :

++ (ajoute un à un nombre), **--** (retire un à un nombre)

variable ++ ; (équivaut à : variable = variable + 1;)
variable -- ; (équivaut à : variable = variable – 1;)

Il est possible de faire des **comparaisons** entre nombres et variables :

< (inférieur à), > (supérieur à), >= (supérieur ou égal à), <= (inférieur ou égal à), == (égal à), != (différent de)

Quand vous allez vouloir ajouter des conditions il vous faudra utiliser des **opérations booléennes** :

&& (et), || (ou), ! (non, ex : !true = false)

Pour finir il existe un **opérateur tertiaire** :

condition ? Résultat1 : résultat2

```
7==5 ? 4 : 3 //renverra 3 car n'est pas égal à 5
```

Les conditions

Les conditions if else switch break

Tout au long de votre (grande) carrière de programmeur, vous aurez sans cesse besoin de **tester** des choses et d'ajouter des **conditions**.

Pour **tester** une **valeur** et décider quoi faire comme **action** vous allez pouvoir utiliser le if, elseif et else.

```
int32 variable = 7 ;
if(variable == 5)
{
   // faire une action
}
else if(variable == 3)
{
   //faire une autre action
}
else
{
   //faire une autre action dans tous les autres cas
}
```

C'est la forme la plus courante que vous trouverez et que vous écrirez.

Il est possible à tout moment **d'arrêter** une **boucle**, une **condition**. Pour cela il faut **utiliser** le mot clé **break**. Il fonctionne comme un **return** mais sort juste **d'un** état. Si par exemple vous avez deux boucles imbriquées, le break arrêtera la boucle la plus **basse**.

```
boucle1
   boucle2
      break ;
   fin boucle2
fin boucle1
```

Ici le **break** arrêtera la **boucle 2**.

Il existe une autre façon de tester des conditions. Moins **courante** et plus **adaptée** aux **states**, il s'agit du **Switch**.

Le **Switch** va tester un par un des cas et arriver dans un cas **default** si aucun cas ne correspond.

Le **switch** utilise le **break** pour fonctionner. Voici un exemple :

```
int32 variable = 10 ;
switch(variable)
{
   case : 1
   {
      //faire une action si variable vaut 1
      break ;
   }
   case : 3
   {
      //faire une action si variable vaut 3
      break ;
   }
   case : 50
   {
      //faire une action si variable vaut 50
      break ;
   }
   default :
   {
      //dans tous les autres cas
      break ;
   }
}
```

Le **default** se met toujours en **dernier**.

Les tableaux

Les **tableaux** (**array** en anglais) sont une donnée très utilisée en programmation. On les utilises pour manipuler des **données** de même type en **grand nombre**.

Vous pouvez faire des tableaux de plusieurs **dimensions**. Sachez tout de même que chaque type de tableau à ses **particularités**.

Commençons par le **tableau statique**. Le tableau statique a une **taille** qui ne **varie pas**. Il vous suffit de donner le **nombre** de **cases** que vous souhaitez et toutes ses cases sont **initialisées** à sa création. C'est **important** car cela signifie que vos cases contiennent des données qui ne sont pas les vôtres. Vous devez donc les **remplacer** par des données que vous **maîtrisez** et que vous **connaissez** avant de les **manipuler**.

Pour initialiser un tableau il suffit de faire comme suit :

NomTableau Type[NombreDeCases]

```
int32[5] JoliTableau ;
```

Ici mon **JoliTableau** aura donc **5 cases** qui contiennent des **int**. Les cases sont **initialisées** mais avec des données qui ne vous **appartiennent** pas. Pour **initialiser** un tableau vous pouvez faire comme suit :

```
int32[5] JoliTableau = {0,1,2,3,4};
```

Ou :

```
int32[5] JoliTableau ;
JoliTableau[0] = 0 ;
JoliTableau[1] = 30 ;
etc.
```

Sachez également, et c'est très **important**, que votre **tableau démarre** à **0**.

Il existe **beaucoup** de **méthodes** pour **manipuler** les **données** de votre tableau.

```
JoliTableau[1] ; //renverra 30
```

Plus d'informations ici : www.cplusplus.com

Pour créer un tableau à **plusieurs dimensions** il faut procéder comme suit :

```
int32[][] GrandTableau ;
GrandTableau[0][5] = 30 ;
GrandTableau[0][5] ; //renverra 30
```

Sachez que à chaque fois que vous allez **rajouter** une **dimension**, votre tableau va prendre une **taille exponentielle** et que cela risque de **ralentir** votre jeu. Préférez un tableau à 1 dimension où vous **maîtrisez** la **taille** et la **longueur**.

Il existe un autre type de tableau, il s'agit des **Tableaux dynamiques**.
Ces tableaux peuvent **s'ajuster** aux données qu'ils **contiennent**. On les appels **TArray**.

On les **initialise** comme suit :

```
TArray<int32> UnTableauDynamique ;
UnTableauDynamique [0] ; //récupère la donnée en 0
```

Ici pour **ajouter** ou **supprimer** des données, par exemple, il faudra utiliser les **fonctions add** et **remove**.

Pour les tableaux dynamiques, référez vous à ce lien pour plus d'infos :
https://wiki.unrealengine.com/Dynamic_Arrays

Les structures

Les **structures** sont des données importantes car elles permettent de **regrouper** plusieurs **types** de données. C'est d'ailleurs cet outils que j'utilise pour faire mon tuto sur l'inventaire.

Une structure peut contenir autant de **données différentes** que vous souhaitez.

Pour créez une structure il faut procéder comme suit dans un **.h** :

```
USTRUCT()
struct FUneStructure
{
    GENERATED_USTRUCT_BODY()

    UPROPERTY()
    int32 ID ;

    UPROPERTY()
    FVector UnVecteur;

    void SetID(const int32 NewID)
    {
            ID = NewID ;
    }

    FUneStructure()
    {
            ID = 0 ;
            UnVecteur = FVector(1,1,1) ;
    }
} ;
```

Il est **important** de ne pas oublier le **USTRUCT**() pour que la structure soit **utilisable** dans les **blueprint**. Vous pouvez créer des **fonctions** dans les structures, rajouter des **propriétés** et n'oubliez pas le **constructeur** pour **initialiser** les **variables**.

Attention, votre nom de structure doit toujours **commencer** par un **F** (majuscule) et votre structure doit être écrite **en dehors** d'une **class**.

Vous disposez ensuite des structures comme vous **souhaitez**. Vous pouvez faire des **tableaux** ou autre. Les **structures** se comportent comme des **variables** ensuite.

Les conversions

Les **conversions** sont très pratiques et vous aurez l'occasion de les utiliser pour du **debug** ou autre. Il faut savoir que pour écrire ou faire du debug, il faut utiliser le type **TEXT** qui n'autorise que le **Text** et le **String** comme type. Il faut donc faire des **conversions** pour **afficher** des **nombres** ou autre.

Il y a tellement de **type** et de **conversions** que je vais simplement vous mettre 2 exemples pour que vous compreniez et ça sera sûrement les plus **utiles** ;)

```
Fstring::FromInt(int32) ; //converti un int32 en string

FString::SanitizeFloat(YourFloat); //converti un float en string
```

Il suffit de **remplacer** « **int32** » ou « **YourFloat** » par vos **données** et le tour est joué !

N'hésitez pas à **fouiller** sur **internet** pour trouver les **conversions** dont vous avez besoin, ça se trouve plutôt **facilement** en général.

Il nous reste à voir les **Cast** maintenant. Qu'est-ce qu'un **cast** ?

Un **cast** c'est le fait de **transformer** un **type** d'objet en un **autre** qui nous arrange afin d'utiliser les fonctions du second type.

Un exemple ? C'est prévu ;)

Vous voulez récupérer **l'objet touché** par une **balle**. Vous avez un récupéré un **Actor** après une **collision**. Vous savez qu'il y a de grandes chance pour que ça soit un **joueur**. Vous allez donc caster pour vérifier qu'il s'agit bien d'un **joueur**. Si c'est le cas, le type de votre **Actor** deviendra le type de votre **joueur**. Vous pourrez alors **utiliser tout** ce qui se trouve sur le **joueur**. Si le **cast échoue**, vous pourrez effectuer une action comme un debug.

```
auto MyPC = Cast<AFPSCharacter>(OtherActor);
if (MyPC)
{
   //cast réussi
}
else
{
   //cast échoué
}
```

Ici je tente de **caster** mon **OtherActor** en **AFPSCharacter**. Si le **cast fonctionne**, je pourrai utiliser toutes les **méthodes** et **variables** de **AFPSCharacter**.
(**auto** signifie que **j'affecte automatique** le **bon type**, ici AFPSCharacter)

Autre

chemin, etc. include

Il est temps de **conclure** avec les **choses inclassables**.

Ici vous allez trouver tout ce que je n'ai pu classer. Des choses plutôt **utiles** !

Pour écrire un message de **debug** à **l'écran** vous pouvez écrire ce qui suit :

```
if (GEngine)
{
   GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Yellow, 
TEXT("HELLO WORLD"));
}
```

Ici j'écris en **jaune**, pendant **5** secondes, **HELLO WORLD**.

Pour créer un objet en c++, rien de plus facile !

```
CollisionComp = 
ObjectInitializer.CreateDefaultSubobject<USphereComponent>(this, 
TEXT("SphereComp"));
RootComponent = CollisionComp;
```

Ici je **créé** une **UsphereComponent** que j'appelle **SphereComp**. Ensuite je **l'affecte** en tant que **RootComponent** sur mon objet.

Pour utiliser une **classe** en tant que telle et non en tant **qu'objet** vous pouvez procéder comme suit :

```
HUDClass = AFPSHUD::StaticClass();
```

Ici j'affecte la classe (**StaticClass**) à ma classe de HUD.

Pour retrouver la **classe** d'un **blueprint** que vous avez créé sur UE4 il faut tout d'abord retrouver son **chemin**.

Pour ça faites un **clique droit** sur l'objet et **Copy Reference**.

```
//set the default pawn class to our Blurprinted character
static ConstructorHelpers::FClassFinder<APawn>
PlayerPawnObject(TEXT("Pawn'/Game/Blueprint/BP_FPSCharacter.BP_FPS
Character_C'"));
if (PlayerPawnObject.Class != NULL)
{
   // L'objet est bien récupéré
   DefaultPawnClass = PlayerPawnObject.Class;
}
```

Ici j'utilise le **FclassFinder** pour retrouver le **Pawn**. Quand j'ai fais un **Copy Reference**, j'ai obtenu ça :

```
Pawn'/Game/Blueprint/BP_FPSCharacter.BP_FPSCharacter_C'
```

Pour **retrouver** un **objet** que vous avez **créé** en **Blueprint**, vous pouvez aussi utiliser ça :

```
// Set the crosshair texture static
ConstructorHelpers::FObjectFinder<UTexture2D>
CrosshairTextObj(TEXT("Texture2D'/Game/models/crosshair.crosshair'
"));
CrosshairTex = CrosshairTextObj.Object;
```

Ici je récupère une **Texture2D** et je **l'affecte** à **CrosshairTex**.

Pour créer un **composant** et **l'attacher** utiliser la méthode suivante :

```
//create a CameraComponent
FirstPersonCameraComponent =
ObjectInitializer.CreateDefaultSubobject<UCameraComponent>(this,
TEXT("FirstPersonCamera"));
//attach it to the root component
FirstPersonCameraComponent->AttachParent = CapsuleComponent;
```

Ici on a créé une **caméra** comme **composant** et on **l'attache** à une **capsule**.

C'est la fin de ce tuto, j'espère ne pas trop vous avoir embrouillé, que vous avez appris des choses et que vous êtes encore plus motivé à entrer dans le vif du sujet !

Avant de **commencer** à réellement coder, vous devez, si ce n'est pas déjà fait, **vous munir** de **UE4** et de **Visual Studio 2013**. Il vous faut **absolument** la **version 2013** pour VS(Visual Studio).

A vos projets !

Pour toute éventuelles questions, remarques, détails ou demandes, n'hésitez pas à prendre contact avec moi ! (Cf : début du tuto)

Antoine Gargasson Août 2015