At Epic, we have a few simple coding standards and conventions. This document is not meant to be a discussion or work in progress, but rather, reflects the state of Epic's current coding standards.

Code conventions are important to programmers for a number of reasons:

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Hardly any software is maintained for its whole life by the original author.
- Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.
- If we decide to expose source code to mod community developers, we want it to be easily understood.
- Many of these conventions are actually required for cross-compiler compatibility.

The coding standards below are C++-centric, but the spirit of the standards are expected to be followed no matter which language is used. A section may provide equivalent rules or exceptions for specific languages where it's applicable.

Class Organization

Classes should be organized with the reader in mind rather than the writer. Since most readers will be using the public interface of the class, that should be declared first, followed by the class's private implementation.

Copyright Notice

Any source file (.h, .cpp, .xaml, etc.) provided by Epic for distribution must contain a copyright notice as the first line in the file. The format of the notice must exactly match that shown below:

```
// Copyright Epic Games, Inc. All Rights Reserved.
```

If this line is missing or not formatted properly, CIS will generate an error and fail.

Naming Conventions

- The first letter of each word in a name (such as type name or variable name) is capitalized, and there is usually no underscore between words. For example, `Health` and `UPrimitiveComponent` are correct, but not `lastMouseCoordinates` or `delta_coordinates`.
- Type names are prefixed with an additional upper-case letter to distinguish them from variable names. For example, `FSkin` is a type name, and `Skin` is an instance of a `FSkin`.
  - Template classes are prefixed by T.
  - Classes that inherit from `UObject` are prefixed by U.
  - Classes that inherit from `AActor` are prefixed by A.
  - Classes that inherit from `SWidget` are prefixed by S.
  - Classes that are abstract interfaces are prefixed by I.
  - Enums are prefixed by E.

- Boolean variables must be prefixed by b (for example, `bPendingDestruction`, or `bHasFadedIn`).
- Most other classes are prefixed by F, though some subsystems use other letters.
- Typedefs should be prefixed by whatever is appropriate for that type: F if it's a typedef of a struct, U if it's a typedef of a `UObject` and so on.
    - A typedef of a particular template instantiation is no longer a template, and should be prefixed accordingly, for example:

    ```
    typedef TArray<FMytype> FArrayOfMyTypes;
    ```

- Prefixes are omitted in C#.
- UnrealHeaderTool requires the correct prefixes in most cases, so it's important to provide them.

- Type and variable names are nouns.
- Method names are verbs that describe the method's effect, or describe the return value of a method that has no effect.

Variable, method, and class names should be clear, unambiguous, and descriptive. The greater the scope of the name, the greater the importance of a good, descriptive name. Avoid over-abbreviation.

All variables should be declared one at a time, so that a comment on the meaning of the variable can be provided. Also, the JavaDocs style requires it. You can use multi-line or single line comments before a variable, and the blank line is optional for grouping variables.

All functions that return a bool should ask a true/false question, such as `IsVisible()` or `ShouldClearBuffer()`.

A procedure (a function with no return value) should use a strong verb followed by an Object. An exception is, if the Object of the method is the Object it is in. Then the Object is understood from context. Names to avoid include those beginning with "Handle" and "Process" because the verbs are ambiguous.

Though not required, we encourage you to prefix function parameter names with "Out" if they are passed by reference, and the function is expected to write to that value. This makes it obvious that the value passed in this argument will be replaced by the function.

If an In or Out parameter is also a boolean, put the "b" before the In/Out prefix, such as `bOutResult`.

Functions that return a value should describe the return value. The name should make clear what value the function will return. This is particularly important for boolean functions. Consider the following two example methods:

```
// what does true mean?
bool CheckTea(FTea Tea);

// name makes it clear true means tea is fresh
bool IsTeaFresh(FTea Tea);
```

Examples

```
float TeaWeight;
int32 TeaCount;
```

```
bool bDoesTeaStink;
FName TeaName;
FString TeaFriendlyName;
UClass* TeaClass;
USoundCue* TeaSound;
UTexture* TeaTexture;
```

Portable Aliases for Basic C++ Types

- `bool` for boolean values (NEVER assume the size of bool). `BOOL` will not compile.
- `TCHAR` for a character (NEVER assume the size of TCHAR).
- `uint8` for unsigned bytes (1 byte).
- `int8` for signed bytes (1 byte).
- `uint16` for unsigned "shorts" (2 bytes).
- `int16` for signed "shorts" (2 bytes).
- `uint32` for unsigned ints (4 bytes).
- `int32` for signed ints (4 bytes).
- `uint64` for unsigned "quad words" (8 bytes).
- `int64` for signed "quad words" (8 bytes).
- `float` for single precision floating point (4 bytes).
- `double` for double precision floating point (8 bytes).
- `PTRINT` for an integer that may hold a pointer (NEVER assume the size of PTRINT).

Use of C++'s `int` and unsigned `int` types whose size may vary across platforms is acceptable in code where the integer width is unimportant. Explicitly-sized types must still be used in serialized or replicated formats.

Comments

Comments are communication and communication is vital. The following sections detail some things to keep in mind about comments (from Kernighan & Pike *The Practice of Programming*).

Guidelines

- Write self-documenting code:

```
// Bad:
t = s + l - b;

// Good:
TotalLeaves = SmallLeaves + LargeLeaves - SmallAndLargeLeaves;
```

- Write useful comments:

```
// Bad:

// increment Leaves

++Leaves;


// Good:

// we know there is another tea leaf

++Leaves;
```

- Do not comment bad code - rewrite it:

```
// Bad:

// total number of leaves is sum of

// small and large leaves less the

// number of leaves that are both

t = s + l - b;


// Good:

TotalLeaves = SmallLeaves + LargeLeaves - SmallAndLargeLeaves;
```

- Do not contradict the code:

```
// Bad:

// never increment Leaves!

++Leaves;


// Good:

// we know there is another tea leaf

++Leaves;
```

Const Correctness

Const is documentation as much as it is a compiler directive, so all code should strive to be const-correct.

This includes:

- Passing function arguments by const pointer or reference if those arguments are not intended to be modified by the function,
- Flagging methods as const if they do not modify the object,
- and using const iteration over containers if the loop isn't intended to modify the container.

Example:

```
void SomeMutatingOperation(FThing& OutResult, const TArray<Int32>& InArray)
{
    // InArray will not be modified here, but OutResult probably will be
}

void FThing::SomeNonMutatingOperation() const
{
    // This code will not modify the FThing it is invoked on
}

TArray<FString> StringArray;
for (const FString& : StringArray)
{
    // The body of this loop will not modify StringArray
}
```

Const should also be preferred on by-value function parameters and locals. This tells a reader that the variable will not be modified in the body of the function, which makes it easier to understand. If you do this, make sure that the declaration and the definition match, as this can affect the JavaDoc process.

Example:

```
void AddSomeThings(const int32 Count);

void AddSomeThings(const int32 Count)
{
    const int32 CountPlusOne = Count + 1;
    // Neither Count nor CountPlusOne can be changed during the body of the function
}
```

One exception to this is pass-by-value parameters, which will ultimately be moved into a container (see "Move semantics"), but this should be rare.

Example:

```
void FBlah::SetMemberArray(TArray<FString> InNewArray)
{
    MemberArray = MoveTemp(InNewArray);
}
```

Put the const keyword on the end when making a pointer itself const (rather than what it points to). References can't be "reassigned" anyway, and so can't be made const in the same way.

Example:

```
// Const pointer to non-const object - pointer cannot be reassigned, but T can still be modified
T* const Ptr = ...;

// Illegal
T& const Ref = ...;
```

Never use const on a return type, as this inhibits move semantics for complex types, and will give compile warnings for built-in types. This rule only applies to the return type itself, not the target type of a pointer or reference being returned.

Example:

```
// Bad - returning a const array
const TArray<FString> GetSomeArray();

// Fine - returning a reference to a const array
const TArray<FString>& GetSomeArray();

// Fine - returning a pointer to a const array
const TArray<FString>* GetSomeArray();

// Bad - returning a const pointer to a const array
const TArray<FString>* const GetSomeArray();
```

Example Formatting

We use a system based on JavaDoc to automatically extract comments from the code and build documentation, so there are some specific comment formatting rules that need to be followed.

The following example demonstrates the format of class, method, and variable comments. Remember that comments should augment the code. The code documents the implementation, and the comments document the intent. Make sure to update comments when you change the intent of a piece of code.

Note that two different parameter comment styles are supported, shown by the `Steep` and `Sweeten` methods. The `@param` style used by `Steep` is the traditional multi-line style, but for simple functions it can be clearer to integrate the parameter and return value documentation into the descriptive comment for the function, as in the Sweeten example. Special comment tags like `@see` or `@return` should only be used to start new lines following the primary description.

Method comments should only be included once, where the method is publicly declared. The method comments should only contain information relevant to callers of the method, including any information about overrides of the method that may be relevant to the caller.

Details about the implementation of the method and its overrides, that are not relevant to callers, should be commented within the method implementation.

```
/** The interface for drinkable objects. */
class IDrinkable
{
public:
    /**
     * Called when a player drinks this object.
     * @param OutFocusMultiplier - Upon return, will contain a multiplier to apply to the drinker's focus.
     * @param OutThirstQuenchingFraction - Upon return, will contain the fraction of the drinker's thirst to quench
     * @warning Only call this after the drink has been properly prepared.
     */
    virtual void Drink(float& OutFocusMultiplier, float& OutThirstQuenchingFraction) = 0;
};

/** A single cup of tea. */
class FTea : public IDrinkable
{
public:
    /**
     * Calculate a delta-taste value for the tea given the volume and temperature of water used to steep.
     * @param VolumeOfWater - Amount of water used to brew in mL
     * @param TemperatureOfWater - Water temperature in Kelvins
```

What does a class comment include?

- A description of the problem this class solves.
- Why this class was created.

What do all those parts of the multi-line method comment mean?

1. **Function purpose**: This documents the `problem this function solves.` As has been said above, comments document `intent`, and code documents `implementation`.
2. **Parameter comments**: Each parameter comment should include:
    - units of measure,
    - the range of expected values,
    - "impossible" values,
    - and the meaning of status/error codes.
3. **Return comment**: This documents the expected return value, just as an output variable is documented. To avoid redundancy, an explicit @return comment should not be used if the sole purpose of the function is to return this value and that is already documented in the function purpose.

4. **Extra information:** `@warning`, `@note`, `@see`, and `@deprecated` can optionally be used to document additional relevant information. Each should be declared on their own line following the rest of the comments.

Modern C++ Language Syntax

Unreal Engine is built to be massively portable to many C++ compilers, so we are careful to use features that are compatible with the compilers we might be supporting. Sometimes features are so useful that we will wrap them up in macros and use them pervasively. However, we usually wait until all of the compilers we might be supporting are up to the latest standard.

We are using many C++14 language features that seem to be well-supported across modern compilers, such as range-based-for, move semantics and lambdas with capture initializers. In some cases, we can wrap up usage of these features in preprocessor conditionals (such as rvalue references in containers). However, we might decide to avoid certain language features entirely, until we are confident we won't be surprised by the appearance of a new platform appearing that can't digest the syntax.

Unless specified below, as a modern C++ compiler feature we are supporting, you should not use compiler-specific language features unless they are wrapped in preprocessor macros or conditionals and used sparingly.

static_assert

This keyword is valid for use where you need a compile-time assertion.

override and final

These keywords are valid for use, and their use is strongly encouraged. There might be many places where these have been omitted, but they will be fixed over time.

nullptr

`nullptr` should be used instead of the C-style `NULL` macro in all cases.

One exception to this is that `nullptr` in C++/CX builds (such as for Xbox One) is actually the managed null reference type. It is mostly compatible with `nullptr` from native C++ except in its type and some template instantiation contexts, and so you should use the `TYPE_OF_NULLPTR` macro instead of the more usual `decltype(nullptr)` for compatibility.

The 'auto' Keyword

You shouldn't use `auto` in C++ code, although a few exceptions are listed below. Always be explicit about the type you're initializing. This means that the type must be plainly visible to the reader. This rule also applies to the use of the `var` keyword in C#.

When is it acceptable to use `auto`?

- When you need to bind a lambda to a variable, as lambda types are not expressible in code.
- For iterator variables, but only where the iterator's type is verbose and would impair readability.
- In template code, where the type of an expression cannot easily be discerned. This is an advanced case.

It's very important that types are clearly visible to someone who is reading the code. Even though some IDEs are able to infer the type, doing so relies on the code being in a compilable state. It also won't assist users of merge/diff tools, or when viewing individual source files in isolation, such as on GitHub.

If you're sure you are using `auto` in an acceptable way, always remember to correctly use const, & or * just like you would with the type name. With `auto`, this will coerce the inferred type to be what you want.

Range-Based for

This is preferred to keep the code easier to understand and more maintainable. When you migrate code that uses old `TMap` iterators, be aware that the old `Key()` and `Value()` functions, which were methods of the iterator type, are now simply `Key` and `Value` fields of the underlying key-value `TPair`.

Example:

```
TMap<FString, int32> MyMap;

// Old style
for (auto It = MyMap.CreateIterator(); It; ++It)
{
    UE_LOG(LogCategory, Log, TEXT("Key: %s, Value: %d"), It.Key(), *It.Value());
}

// New style
for (TPair<FString, int32>& Kvp : MyMap)
{
    UE_LOG(LogCategory, Log, TEXT("Key: %s, Value: %d"), *Kvp.Key, Kvp.Value);
}
```

We also have range replacements for some standalone iterator types.

Example:

```
// Old style
for (TFieldIterator<UProperty> PropertyIt(InStruct, EFieldIteratorFlags::IncludeSuper); PropertyIt; ++PropertyIt)
{
    UProperty* Property = *PropertyIt;
    UE_LOG(LogCategory, Log, TEXT("Property name: %s"), *Property->GetName());
}

// New style
for (UProperty* Property : TFieldRange<UProperty>(InStruct, EFieldIteratorFlags::IncludeSuper))
{
    UE_LOG(LogCategory, Log, TEXT("Property name: %s"), *Property->GetName());
}
```

Lambdas and Anonymous Functions

Lambdas can be used freely. The best lambdas should be no more than a couple of statements in length, particularly when used as part of a larger expression or statement, for example as a predicate in a generic algorithm

Example:

```
// Find first Thing whose name contains the word "Hello"
Thing* HelloThing = ArrayOfThings.FindByPredicate([](const Thing& Th){ return Th.GetName().Contains(TEXT("Hello")); }

// Sort array in reverse order of name
Algo::Sort(ArrayOfThings, [](const Thing& Lhs, const Thing& Rhs){ return Lhs.GetName() > Rhs.GetName(); });
```

Be aware that stateful lambdas can't be assigned to function pointers, which we tend to use a lot.

Non-trivial lambdas should be documented in the same manneras regular functions. Don't be afraid to split them over a few more lines in order to include comments.

Explicit captures should be used rather than automatic capture (`[&]` and `[=]`). This is important for readability, maintainability and performance reasons, particularly when used with large lambdas and deferred execution. It declares the intent of the author and so mistakes can more easily be caught during code review. Incorrect captures can have negative consequences which are more likely to become a problem as the code is maintained over time.

- By-reference capture and by-value capture of pointers (including the `this` pointer) can cause accidental dangling references, if execution of the lambda is deferred.
- By-value capture can be a performance concern if it makes unnecessary copies for a non-deferred lambda.
- Accidentally captured UObject pointers are invisible to the garbage collector.Automatic capture captures `this` implicitly if any member variables are referenced, even though `[=]` gives the impression of the lambda having its own copies of everything.

Explicit return types should be used for large lambdas, or when you are returning the result of another function call. These should be considered in the same way as the `auto` keyword:

```
// Without the return type here, the return type is unclear
auto Lambda = []() -> FMyType
{
    return SomeFunc();
}
```

Automatic captures and implicit return types are acceptable for trivial, non-deferred lambdas, such as in Sort calls, where the semantics are obvious and being explicit would make it overly verbose.

The capture initializer feature from C++14 may be used:

```
TUniquePtr<FThing> ThingPtr = MakeUnique<FThing>();
AsyncTask([UniquePtr = MoveTemp(UniquePtr)]()
{
    // Use UniquePtr here
});
```

Strongly - Typed Enums

Enum classes should always be used as a replacement for old-style namespaced enums, both for regular enums and UENUMs. For example:

```
// Old enum
UENUM()
namespace EThing
{
    enum Type
    {
        Thing1,
        Thing2
    };
}

// New enum
UENUM()
enum class EThing : uint8
{
    Thing1,
    Thing2
}
```

These are also supported as UPROPERTYs, and replace the old TEnumAsByte<> workaround. Enum properties can also be any size, not just bytes:

```
// Old property
UPROPERTY()
TEnumAsByte<EThing::Type> MyProperty;

// New property
UPROPERTY()
EThing MyProperty;
```

However, enums exposed to Blueprints must continue to be based on uint8.

Enum classes used as flags can take advantage of the ENUM_CLASS_FLAGS(EnumType) macro to automatically define all of the bitwise operators:

```
enum class EFlags
{
    None = 0x00,
    Flag1 = 0x01,
    Flag2 = 0x02,
    Flag3 = 0x04
};

ENUM_CLASS_FLAGS(EFlags)
```

The one exception to this is the use of flags in a *truth* context - this is a limitation of the language. Instead, all flags enums should have an enumerator called `None` which is set to 0 for comparisons:

```
// Old
if (Flags & EFlags::Flag1)

// New
if ((Flags & EFlags::Flag1) != EFlags::None)
```

Move Semantics

All of the main container types `TArray, TMap, TSet, FString\` have move constructors and move assignment operators. These are often used automatically when passing/returning these types by value, but can be explicitly invoked by using `MoveTemp`, which is UE4's equivalent of `std::move`.

Returning containers or strings by value can be a win for expressivity, without the usual cost of temporary copies. Rules around pass-by-value and use of `MoveTemp` are still being established, but can already be found in some optimized areas of the codebase.

Default Member Initializers

Default member initializers can be used to define the defaults of a class inside the class itself:

```
UCLASS()
class UTeaOptions : public UObject
{
    GENERATED_BODY()

public:
    UPROPERTY()
    int32 MaximumNumberOfCupsPerDay = 10;

    UPROPERTY()
    float CupWidth = 11.5f;

    UPROPERTY()
    FString TeaType = TEXT("Earl Grey");

    UPROPERTY()
    EDrinkingStyle DrinkingStyle = EDrinkingStyle::PinkyExtended;
};
```

Code written like this has the following benefits:

- It doesn't need to duplicate initializers across multiple constructors.
- It isn't possible to mix the initialization order and declaration order.
- The member type, property flags and default values are all in one place, which helps readability and maintainability.

However, there are also some downsides:

- Any change to the defaults will require a rebuild of all dependent files.
- Headers can't change in patch releases of the engine, so this style can limit the kinds of fixes that are possible.
- Some things can't be initialized in this way, such as base classes, `UObject` subobjects, pointers to forward-declared types, values deduced from constructor arguments, and members initialized over multiple steps.
- Putting some initializers in the header, and the rest in constructors in the .cpp file, can reduce readability and maintainability.

Use your best judgment when deciding whether to use them. As a rule of thumb, default member initializers make more sense in game code than engine code. Also consider using config files for default values.

Third Party Code

Whenever you modify the code to a library that we use in the engine, be sure to tag your changes with a //@UE4 comment, as well as an explanation of why you made the change. This makes merging the changes into a new version of that library easier, and lets licensees easily find any modifications we have made.

Any third party code included in the engine should be marked with comments formatted to be easily searchable. For example:

```
// @third party code - BEGIN PhysX
#include <physx.h>
// @third party code - END PhysX
// @third party code - BEGIN MSDN SetThreadName
// [http://msdn.microsoft.com/en-us/library/xcb2z8hs.aspx]
// Used to set the thread name in the debugger
...
//@third party code - END MSDN SetThreadName
```

Code Formatting

Braces

Brace wars are foul. Epic has a long standing usage pattern of putting braces on a new line. Please adhere to that usage.

Always include braces in single-statement blocks. For example.:

```
if (bThing)
{
    return;
}
```

If - Else

Each block of execution in an if-else statement should be in braces. This is to prevent editing mistakes - when braces are not used, someone could unwittingly add another line to an if block. The extra line wouldn't be controlled by the if expression, which would be bad. It's also bad when conditionally compiled items cause if/else statements to break. So always use braces.

```
if (bHaveUnrealLicense)
{
    InsertYourGameHere();
}
else
{
    CallMarkRein();
}
```

A multi-way if statement should be indented with each else if indented the same amount as the first if; this makes the structure clear to a reader:

```
if (TannicAcid < 10)
{
    UE_LOG(LogCategory, Log, TEXT("Low Acid"));
}
else if (TannicAcid < 100)
{
    UE_LOG(LogCategory, Log, TEXT("Medium Acid"));
}
else
{
    UE_LOG(LogCategory, Log, TEXT("High Acid"));
}
```

Tabs and Indenting

Here are the standards for indenting your code.

- Indent code by execution block.
- Use tabs, not spaces, for whitespace at the beginning of a line. Set your tab size to 4 characters. However, spaces are sometimes necessary and allowed for keeping code aligned regardless of the number of spaces in a tab. For example, when you are aligning code that follows non-tab characters.
- If you are writing code in C#, please also use tabs, and not spaces. The reason for this is that programmers often switch between C# and C++, and most prefer to use a consistent setting for tabs. Visual Studio defaults to using spaces for C# files, so you will need to remember to change this setting when working on Unreal Engine code.

Switch Statements

Except for empty cases (multiple cases having identical code), switch case statements should explicitly label that a case falls through to the next case. Either include a break, or include a falls-through comment in each case. Other code control-transfer commands (return, continue, and so on) are fine as well.

Always have a default case, and include a break just in case someone adds a new case after the default.

```
switch (condition)
{
    case 1:
        ...
        // falls through

    case 2:
        ...
        break;

    case 3:
        ...
        return;

    case 4:
    case 5:
        ...
        break;

    default:
        break;
```

Namespaces

You can use namespaces to organize your classes, functions and variables where appropriate. If you do use them, follow the rules below.

- Unreal code is currently not wrapped in a global namespace. Be careful to avoid collisions in the global scope, especially when using or including third party code.
- `Using` declarations:
    - Do not put `using` declarations in the global scope, even in a .cpp file (it will cause problems with our "unity" build system.)
    - It's okay to put `using` declarations within another namespace, or within a function body.
    - If you put `using` declarations within a namespace, this will carry over to other occurrences of that namespace in the same translation unit. As long as you are consistent, it will be fine.
    - You can only use `using` declarations in header files safely if you follow the above rules.
- Note that forward-declared types need to be declared within their respective namespace. If you don't do this, you will get link errors.
- If you declare a lot of classes/types within a namespace, it can be difficult to use those types in other global-scoped classes (for example, function signatures will need to use explicit namespace when appearing in class declarations).

- You can use `using` declarations to only alias specific variables within a namespace into your scope (for example, using `Foo::FBar`). However, we don't usually do that in Unreal code.
- Namespaces are not supported by UnrealHeaderTool, so they should not be used when defining `UCLASSes`, `USTRUCTs` and so on.

Physical Dependencies

- File names should not be prefixed where possible; for example, `Scene.cpp` instead of `UScene.cpp`. This makes it easy to use tools like Workspace Whiz, or Visual Assist's Open File in Solution, by reducing the number of letters needed to identify the file you want.
- All headers should protect against multiple includes with the `#pragma once` directive. Note that all compilers we use support `#pragma once`.

```
#pragma once

//<file contents>
```

- In general, try to minimize physical coupling.
- If you can use forward declarations instead of including a header, do so.
- When including, be as fine grained as possible. For example, do not include Core.h--include the specific headers in Core that you need definitions from.
- Try to include every header you need directly, to make fine-grained inclusion easier.
- Don't rely on a header that is included indirectly by another header you include.
- Don't rely on being included through another header. Include everything you need.
- Modules have Private and Public source directories. Any definitions that are needed by other modules must be in headers in the Public directory. Everything else should be in the Private directory. Note that in older Unreal modules, these directories may still be called "Src" and "Inc", but those directories are meant to separate private and public code in the same way, and are not meant to separate header files from source files.
- Don't worry about setting up your headers for precompiled header generation. UnrealBuildTool can do a better job of this than you can.
- Split up large functions into logical sub-functions. One area of compilers' optimizations is the elimination of common subexpressions. The bigger your functions are, the more work the compiler has to do to identity them. This leads to greatly inflated build times.
- Don't use too many inline functions, because they force rebuilds even in files which don't use them. Inline functions should only be used for trivial accessors and when profiling shows there is a benefit to doing so.
- Be even more conservative in the use of `FORCEINLINE`. All code and local variables will be expanded out into the calling function, and this will cause the same build time problems caused by large functions.

Encapsulation

Enforce encapsulation with the protection keywords. Class members should almost always be declared private unless they are part of the public/protected interface to the class. Use your best judgment, but always be aware that a lack of accessors makes it hard to refactor later without breaking plugins and existing projects.

If particular fields are only intended to be usable by derived classes, make them private and provide protected accessors.

Use final if your class is not designed to be derived from.

General Style Issues

- Minimize dependency distance. When code depends on a variable having a certain value, try to set that variable's value right before using it. Initializing a variable at the top of an execution block, and not using it for a hundred lines of code, gives lots of space for someone to accidentally change the value without realizing the dependency. Having it on the next line makes it clear why the variable is initialized the way it is and where it is used.
- Split methods into sub-methods where possible. It is easier for someone to look at a big picture, and then drill down to the interesting details, than it is to start with the details and reconstruct the big picture from them. In the same way, it is easier to understand a simple method, that calls a sequence of several well-named sub-methods, than it is to understand an equivalent method that simply contains all the code in those sub-methods.
- In function declarations or function call sites, do not add a space between the function's name and the parentheses that precede the argument list.
- Address compiler warnings. Compiler warning messages mean something is wrong. Fix what the compiler is warning you about. If you absolutely can't address it, use `#pragma` to suppress the warning, but this should only be done as a last resort.
- Leave a blank line at the end of the file. All .cpp and .h files should include a blank line, to coordinate with gcc.
- Debug code should either be generally useful and polished, or not checked in. Debug code that is intermixed with other code makes the other code harder to read.
- Always use the `TEXT()` macro around string literals. Without it, code that constructs `FStrings` from literals will cause an undesirable string conversion process.
- Avoid repeating the same operation redundantly in loops. Move common subexpressions out of loops to avoid redundant calculations. Make use of statics in some cases, to avoid globally-redundant operations across function calls, such as constructing an `FName` from a string literal.
- Be mindful of hot reload. Minimize dependencies to cut down on iteration time. Don't use inlining or templates for functions which are likely to change over a reload. Only use statics for things which are expected to remain constant over a reload.
- Use intermediate variables to simplify complicated expressions. If you have a complicated expression, it can be easier to understand if you split it into sub-expressions, that are assigned to intermediate variables, with names describing the meaning of the sub-expression within the parent expression. For example:

```
if ((Blah->BlahP->WindowExists->Etc && Stuff) &&
    !(bPlayerExists && bGameStarted && bPlayerStillHasPawn &&
    IsTuesday())))
{
    DoSomething();
}
```

`should be replaced with`

```
const bool bIsLegalWindow = Blah->BlahP->WindowExists->Etc && Stuff;
const bool bIsPlayerDead = bPlayerExists && bGameStarted && bPlayerStillHasPawn && IsTuesday();
if (bIsLegalWindow && !bIsPlayerDead)
{
    DoSomething();
}
```

- Pointers and references should only have one space, which is to the right of the pointer or reference. This makes it easy to quickly use **Find in Files** for all pointers or references to a certain type.
- *Use this:*

```
FShaderType* Ptr
```

Not these:

```
FShaderType *Ptr
FShaderType * Ptr
```

- Shadowed variables are not allowed. C++ allows variables to be shadowed from an outer scope, but this makes usage ambiguous to a reader. For example, there are three usable `Count` variables in this member function:

```
class FSomeClass
{
public:
    void Func(const int32 Count)
    {

        for (int32 Count = 0; Count != 10; ++Count)
        {
            // Use Count
        }
    }

private:
    int32 Count;
}
```

- Avoid using anonymous literals in function calls. Prefer named constants which describe their meaning:

```
// Old style
Trigger(TEXT("Soldier"), 5, true);.

// New style
const FName ObjectName                = TEXT("Soldier");
const float CooldownInSeconds         = 5;
const bool bVulnerableDuringCooldown  = true;
Trigger(ObjectName, CooldownInSeconds, bVulnerableDuringCooldown);
```

This makes intent more obvious to a casual reader as it avoids the need to look up the function declaration to understand it.

API Design Guidelines

- `bool` function parameters should be avoided, particularly for flags passed to functions. These have the same anonymous literal problem as mentioned previously, but they also tend to multiply over time as APIs get extended with more behavior. Instead, prefer an enum (see the advice on use of enums as flags in the Strongly-Typed Enums section):

```
// Old style
FCup* MakeCupOfTea(FTea* Tea, bool bAddSugar = false, bool bAddMilk = false, bool bAddHoney = false, bool bAddL
FCup* Cup = MakeCupOfTea(Tea, false, true, true);

// New style
enum class ETeaFlags
{
    None,
    Milk  = 0x01,
    Sugar = 0x02,
    Honey = 0x04,
    Lemon = 0x08
};
ENUM_CLASS_FLAGS(ETeaFlags)
```

This form prevents the accidental transposing of flags, avoids accidental conversion from pointer and integer arguments, removes the need to repeat redundant defaults, and is more efficient.

It is acceptable to use `bools` as arguments when they are the complete state to be passed to a function like a setter, such as `void FWidget::SetEnabled(bool bEnabled)`. Though consider refactoring if this changes.

- Avoid overly-long function parameter lists. If a function takes many parameters then consider passing a dedicated struct instead:

```
// Old style
TUniquePtr<FCup[]> MakeTeaForParty(const FTeaFlags* TeaPreferences, uint32 NumCupsToMake, FKettle* Kettle, ETeaT

// New style
struct FTeaPartyParams
{
    const FTeaFlags* TeaPreferences       = nullptr;
    uint32           NumCupsToMake        = 0;
    FKettle*         Kettle               = nullptr;
    ETeaType         TeaType              = ETeaType::EnglishBreakfast;
    float            BrewingTimeInSeconds = 120.0f;
};
TUniquePtr<FCup[]> MakeTeaForParty(const FTeaPartyParams& Params);
```

- Avoid overloading functions by `bool` and `FString`, as this can have unexpected behavior:

```
void Func(const FString& String);
void Func(bool bBool);

Func(TEXT("String")); // Calls the bool overload!
```

- Interface classes (prefixed with "I") should always be abstract, and must not have member variables. Interfaces are allowed to contain methods that are not pure-virtual, and can even contain methods that are non-virtual or static, as long as they are implemented inline.

- Use the `virtual` and `override` keywords when declaring an overriding method. When declaring a virtual function in a derived class, that overrides a virtual function in the parent class, you must use both the `virtual` and the `override` keywords. For example:

```
class A

{

public:

    virtual void F() {}

};


class B : public A

{

public:

    virtual void F() override;

}
```

NOTE

There is a lot of existing code that doesn't follow this yet, due to the recent addition of the `override` keyword. The `override` keyword should be added to that code when convenient.

Platform-Specific Code

Platform-specific code should always be abstracted and implemented in platform-specific source files in appropriately named subdirectories, for example:

```
Source/Runtime/Core/Private/[PLATFORM]/[PLATFORM]Memory.cpp
```

In general, you should avoid adding any uses of `PLATFORM_[`PLATFORM`]` (e.g `PLATFORM_XBOXONE`) to code outside of a directory named [PLATFORM].

Instead, extend the hardware abstraction layer to add a static function, e.g. in FPlatformMisc:

```
FORCEINLINE static int32 GetMaxPathLength()
{
    return 128;
}
```

Platforms can then override this function, returning either a platform-specific constant value or even using platform APIs to determine the result.

If you force-inline the function it will have the same performance characteristics as using a define.

In cases where a define is absolutely necessary, create new #defines that describe particular properties that can apply to a platform, for example `PLATFORM_USE_PTHREADS`. Set the default value in Platform.h and override for any platforms which require it in the platform-specific Platform.h file.

For example, in Platform.h we have:

```
#ifndef PLATFORM_USE_PTHREADS
    #define PLATFORM_USE_PTHREADS 1
#endif
```

Windows/WindowsPlatform.h has:

```
#define PLATFORM_USE_PTHREADS 0
```

Cross-platform code can then use the define directly without needing to know the platform.

```
#if PLATFORM_USE_PTHREADS
    #include "HAL/PThreadRunnableThread.h"
#endif
```

Reasoning: centralizing the platform-specific details of the engine allow for such details to be contained entirely within platform-specific source files. Doing so makes it easier to maintain the engine across multiple platforms and also to port the code to new platforms without the need to scour the codebase for platform-specific defines.

Keeping platform code in platform-specific folders is also a requirement for NDA platforms such as PS4, XboxOne and Nintendo Switch.

It is important to ensure the code compiles and runs regardless of whether the `[PLATFORM]` subdirectory is present. In other words, cross platform code should never be dependent on platform-specific code.