# Standard Header

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

\* Filename: databaseinterface.cpp

\* Group Name: TeamNixon

\* Subject: CSCI222

\* Assignment: File Archiver

\* File Description: This module handles all interactions with the database

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

# Origin of coding standard

These standards are heavily influenced by the standards used by Epic in the Unreal Engine. This is a well established code base with a published coding standard. The standard has been tried and tested across many development studios and projects, and has proven to be a clear and workable standard on a large codebase of several million lines of code.

<https://docs.unrealengine.com/latest/INT/Programming/Development/CodingStandard/index.html>

## Naming conventions - update to custom wording

All variable names (except iterator/index variables) should have descriptive names. 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.

|  |
| --- |
| // Bad: t = s + l + b;  // Good: TotalLeaves = SmallLeaves + LargeLeaves - SmallAndLargeLeaves; |

All variables should be declared one at a time so that a comment on the meaning of the variable can be provided.

All functions that return a bool should ask a true/false question, such as IsVisible() or ShouldClearBuffer(). All boolean variables must be prefixed with a "b" character (e.g. bPendingDestruction, or bHasFadedIn).

|  |
| --- |
| **bool** CheckTea(FTea Tea) {...} // what does true mean?  **bool** IsTeaFresh(FTea Tea) {...} // name makes it clear true means tea is fresh |

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"; 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.

## General guidelines - update to custom wording

Comments are communication; communication is vital. Some things to keep in mind about comments (from Kernighan & Pike The Practice of Programming):

Write self-documenting code:

|  |
| --- |
| // Bad: t = s + l + b;  // Good: TotalLeaves = SmallLeaves + LargeLeaves - SmallAndLargeLeaves; |

Write useful comments:

|  |
| --- |
| // Bad: // increment iLeaves ++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 iLeaves! ++Leaves;  // Good: // we know there is another tea leaf ++Leaves; |

# Code Formatting

## Braces { }

All braces should begin on a new line, and all control structures, even 1 line if statements should have braces.

|  |
| --- |
| **if**(bRequireFood)  {  EatSomething();  }  **else**  {  DoSomethingElse();  } |

## Indentation

Indentation should be accomplished with tabs for the majority of indentation. Spaces can be used for the purpose of aligning code in instances such as aligning a multi-line array initialisation statement.

## Switch Statements - update to custom wording

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 a falls-through comment in each case. Other code control-transfer commands (return, continue, etc.) 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**;  } |

## Enumerations - update to custom wording

Always prefix enum types with an "E" character. We require that all enums use namespaces (or empty structs) for scoping. The reason for this is that in C++, enum values are scoped to the same scope as the enum type itself. This can cause naming collisions, which results in a programmer having to create strange names or prefixes for enum values to make their values appear unique. Instead, we always explicitly scope new enums using namespaces. The actual enum type name within the namespace should always be declared as "Type".

|  |
| --- |
| **namespace** EColorChannel {  **enum** Type  {  Red,  Green,  Blue  }; }  FString GetNameForColorChannel(**const** EColorChannel::Type ColorChannel) {  **switch**(ColorChannel)  {  **case** EColorChannel::Red: **return** TEXT("Red");  **case** EColorChannel::Green: **return** TEXT("Green");  **case** EColorChannel::Blue: **return** TEXT("Blue");  **default**: **return** TEXT("Unknown");  } } |

## Pointers - update to custom wording

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

|  |
| --- |
| //Use this:  FShaderType\* Type  //Not These  FShaderType \*Type FShaderType \* Type |

# General Issues - update to custom wording

* 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. Humans are better at looking at a big picture, and drilling down to the interesting details than 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 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 precedes the argument list.
* Address compiler warnings. Compiler warning messages mean something is not as it should be. Fix what the compiler is complaining about. If you absolutely cannot address it, use #pragma to suppress the warning; this is a remedy of last resort.
* Leave a blank line at the end of the file. All .cpp and .h files should include a blank line to play nice with gcc.
* Enforce encapsulation with the protection keywords. Class members should be declared private unless they are part of the public interface to the class.
* 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 even methods that are non-virtual or static, as long as they are implemented inline.
* Use const wherever possible. Particularly on reference parameters and class methods. const is documentation as much as it is a compiler directive.
* Debug code should either be generally useful and polished, or not checked in. Debug code intermixed with other code makes the other code much harder to read.
* 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(); } |