Asserts

Reference for Unreal Engine's Assert Functionality

In C and C++ programming, `assert` helps to detect and diagnose unexpected or invalid runtime conditions during development. These conditions are often checks that a pointer is non-null, a divisor is non-zero, a function isn't running recursively, or other important assumptions that the code requires, but that would be inefficient to check every time. In some cases, `assert` discovers bugs that cause delayed crashes before the actual crash happens, such as deleting an object that will be required in a future tick, assisting the developer at discovering the root cause of an eventual crash. A key feature of `assert` is that it doesn't exist in shipping code, meaning it has no performance impact on a shipped product, but also must not have any side effects. The simplest way to think of `assert` is that whatever is "asserted" must be true, or the program will stop running.

Unreal Engine provides three different families of `assert` equivalents: `check`, `verify`, and `ensure`. If you would like to examine the code behind these features, you can find the relevant macros in `Engine/Source/Runtime/Core/Public/Misc/AssertionMacros.h`. Each one behaves slightly differently, but they all serve the same general role as diagnostic tools for use during development.

Check

The Check family is the closest to the base `assert`, in that members of this family halt execution when the first parameter evaluates to a false value, and do not run in shipping builds by default. The following Check macros are available:

Macro	Parameters	Behavior
`check` or `checkSlow`	`Expression`	Halts execution if `Expression` is false

Macro	Parameters	Behavior
`checkf` or `checkfSlow`	`Expression`, `FormattedText`, `...`	Halts execution if `Expression` is false and outputs `FormattedText` to the log
`checkCode`	`Code`	Executes `Code` within a do-while loop structure that runs once; primarily useful as a way to prepare information that another Check requires
`checkNoEntry`	(none)	Halts execution if the line is ever hit, similar to `check(false)`, but intended for code paths that should be unreachable
`checkNoReentry`	(none)	Halts execution if the line is hit more than once
`checkNoRecursion`	(none)	Halts execution if the line is hit more than once without leaving scope
`unimplemented`	(none)	Halts execution if the line is ever hit, similar to `check(false)`, but intended for virtual functions that should be overridden and not called

By default, check macros only operate in Debug and Development builds. "Slow" macros only operate in Debug builds. Setting `USE_CHECKS_IN_SHIPPING=1` enables check macros in Test and Shipping builds. This can be useful if you suspect that code inside a Check macro is altering a value, or you have Shipping-only bugs that are hard to track down, and that you suspect would be caught by your existing Check macros. Projects should ship with `USE_CHECKS_IN_SHIPPING` set to its default value of `0`.

Verify

The Verify family behaves identically to the Check family in most builds. However, Verify macros evaluate their expressions even in builds where Check macros are disabled. This means that you should use Verify macros only when the expression needs to run independently of diagnostic checks. For example, if you have a function that performs an action and then returns a `bool` indicating whether that action succeeded or failed, you should use Verify rather than Check to make sure that the action was successful. This is because, in shipping builds, Verify will ignore the return value, but will still perform the action. Check, however, will simply not call the function at all in shipping builds, resulting in different behavior.

Macro	Parameters	Behavior
`verify` or `verifySlow`	`Expression`	Halts execution if `Expression` is false
`verifyf` or `verifyfSlow`	`Expression`, `FormattedText`, `...`	Halts execution if `Expression` is false and outputs `FormattedText` to the log

Verify macros operate fully in Debug, Development, Test, and Shipping Editor builds, except those ending in "Slow", which only operate in Debug builds. Defining `USE_CHECKS_IN_SHIPPING` to hold a true value, generally `1`, overrides this behavior. In all other cases, Verify macros will evaluate their expressions, but will not halt execution or output text to the log.

Ensure

The Ensure family is similar to the Verify family, but works with non-fatal errors. This means that if an Ensure macro's expression evaluates as false, the Engine will inform the crash reporter, but will continue running. In order to avoid flooding the crash reporter, Ensure macros will only report once per Engine or Editor session. If your use case requires that an Ensure macro report every time its expression evaluates as false, use the "Always" version of the macro.

Macro	Parameters	Behavior
`ensure`	`Expression`	Notifies the crash reporter on the first time `Expression` is false
`ensureMsgf`	`Expression`, `FormattedText`, `...`	Notifies the crash reporter and outputs `FormattedText` to the log on the first time `Expression` is false
`ensureAlways`	`Expression`	Notifies the crash reporter if `Expression` is false
`ensureAlwaysMsgf`	`Expression`, `FormattedText`, `...`	Notifies the crash reporter and outputs `FormattedText` to the log if `Expression` is false

Ensure macros evaluate their expressions in all builds, but only contact the crash reporter in Debug, Development, Test, and Shipping Editor builds.

Usage Examples

The following hypothetical situations demonstrate use cases where Check, Verify, and Ensure can help to clarify your code or assist with debugging.

```
// This function should never be called with a null JumpTarget. Halt the program if it happens.
void AMyActor::CalculateJumpVelocity(AActor* JumpTarget, FVector& JumpVelocity)
{
check(JumpTarget != nullptr);
// (Compute velocity we need to land on JumpTarget. We are now confident that JumpTarget is non-null.)
}
```

Copy full snippet

```
// This sets the value of Mesh and expects it to be non-null. Halt the program if Mesh is null afterward.
// We use verify instead of check because our expression has a side effect (setting Mesh).
verify((Mesh = GetRenderMesh()) != nullptr);
```

Copy full snippet

```
// This line of code catches a minor error that could happen in the shipping version of the product.
// The error is minor enough that we can handle it without needing to halt execution.
// We may think that we've fixed the bug, but still want to know if it ever happens.
void AMyActor::Tick(float DeltaSeconds)
{
Super::Tick(DeltaSeconds);
// Ensure that bWasInitialized is true before proceeding. If it is false, log that the bug hasn't been fixed yet.
if (ensureMsgf(bWasInitialized, TEXT("%s ran Tick() with bWasInitialized == false"), *GetActorLabel()))
{
// (Do something that requires a properly-initialized AMyActor.)
}
}
```

Copy full snippet

```
// This code halts if we add a new shape type, but forget to handle it in this switch block.
switch (MyShape)
{
case EShapes::S_Circle:
// (Handle circles.)
break;
case EShapes::S_Square:
// (Handle squares.)
```

```
break;
default:
// There should be a case for every shape type, so this should never happen.
checkNoEntry();
break;
}
```

Copy full snippet

```
// This UObject has a test function, IsEverythingOK, that has no side effects, but returns false if there's a problem.
// If this happens, terminate with a fatal error.
// Because the code has no side effects and only serves a diagnostic purpose, it does not need to run in shipping builds.
checkCode(
if (!IsEverythingOK())
{
UE_LOG(LogUObjectGlobals, Fatal, TEXT("Something is wrong with %s! Terminating."), *GetFullName());
}
);
```

Copy full snippet

```
// There should never be a cycle in this list, and our program will hang if there is one. However, checking for cycles can be slow, so we only want to do it in debug builds.
checkfSlow(!MyLinkedList.HasCycle(), TEXT("Found a cycle in the list!"));
// (Walk through the list, running some code on each element.)
```

Copy full snippet