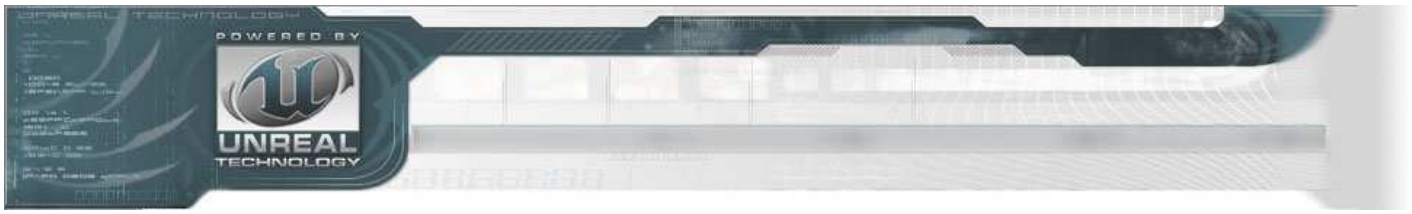

2.1 - UnrealScript Language Reference

Document Summary: Introduction to and short reference of UnrealScript. Original author was Tim Sweeney (EpicGames).

Document Changelog: Created; Maintained over time.

Contents

- Control Structures
  - Repetition Structures
    - For Loops
    - Do Loops
    - While Loops
    - Continue
    - Break
  - Selection Structures
    - If-Then-Else Statements
    - Case Statements
    - Goto
- Language Functionality
  - Built-in operators and their precedence
  - General purpose functions
    - Creating objects
    - Integer functions
    - Floating point functions
    - String functions
    - Vector functions
    - Timer functions
    - Debugging functions
  - UnrealScript preprocessor
  - UnrealScript tools and utilities
    - Script Profiler
    - Script Debugger
- Advanced Language Features
  - Timers
  - States
    - Overview of States
    - State Labels and Latent Functions
    - State inheritance and scoping rules
    - Advanced state programming
      - State Stacking
  - Replication
  - Iteration (ForEach)
  - Function Calling Specifiers
  - Accessing static functions in a variable class
  - Default values of variables
    - Accessing default values of variables
    - Accessing default values of variables through a class reference
    - Specifying default values using the defaultproperties block
      - Syntax
  - Struct Defaults
  - Dynamic Arrays
    - Length Variable
    - Iterating Dynamic Arrays
  - Interface Classes

Introduction

Quick Links

Be sure to check out the UnrealScript CheatSheet and the MasteringUnrealScriptTOC tutorials.

Purpose of this document

This is a technical document describing the UnrealScript language. It's not a tutorial, nor does it provide detailed examples of useful UnrealScript code. For examples of UnrealScript the reader is referred to the source code of the engine, which provides tens of thousands of lines of working UnrealScript code which solves many problems such as AI, movement, inventory, and triggers. A good way to get started is by looking at the "Actor", "Object", "Controller", "Pawn", and "Weapon" scripts.

This document assumes that the reader has a working knowledge of C/C++ or Java, is familiar with object-oriented programming, has played Unreal and has used the UnrealEd editing environment.

For programmers who are new to OOP, I highly recommend going to Amazon.com or a bookstore and buying an introductory book on Java programming. Java is very similar to UnrealScript, and is an excellent language to learn about due to its clean and simple approach.

Design goals of UnrealScript

UnrealScript was created to provide the development team and the third-party Unreal developers with a powerful, built-in programming language that maps naturally onto the needs and nuances of game programming.

The major design goals of UnrealScript are:

- To support the major concepts of time, state, properties, and networking which traditional programming languages don't address. This greatly simplifies UnrealScript code. The major complication in C/C++ based AI and game logic programming lies in dealing with events that take a certain amount of game time to complete, and with events which are dependent on aspects of the object's state. In C/C++, this results in spaghetti-code that is hard to write, comprehend, maintain, and debug. UnrealScript includes native support for time, state, and network replication which greatly simplify game programming.

- To provide Java-style programming simplicity, object-orientation, and compile-time error checking. Much as Java brings a clean programming platform to Web programmers, UnrealScript provides an equally clean, simple, and robust programming language to 3D gaming. The major programming concepts which UnrealScript derives from Java are:
    - a pointerless environment with automatic garbage collection;
    - a simple single-inheritance class graph;
    - strong compile-time type checking;
    - a safe client-side execution "sandbox";
    - and the familiar look and feel of C/C++/Java code.

- To enable rich, high level programming in terms of game objects and interactions rather than bits and pixels. Where design tradeoffs had to be made in UnrealScript, I sacrificed execution speed for development simplicity and power. After all, the low-level, performance-critical code in Unreal is written in C/C++ where the performance gain outweighs the added complexity. UnrealScript operates at a level above that, at the object and interaction level, rather than the bits and pixels level.

During the early development of UnrealScript, several major different programming paradigms were explored and discarded before arriving at the current incarnation. First, I researched using the Sun and Microsoft Java VM's for Windows as the basis of Unreal's scripting language. It turned out that Java

offered no programming benefits over C/C++ in the Unreal context, added frustrating restrictions due to the lack of needed language features (such as operator overloading), and turned out to be unfathomably slow due to both the overhead of the VM task switch and the inefficiencies of the Java garbage collector in the case of a large object graph. Second, I based an early implementation of UnrealScript on a Visual Basic variant, which worked fine, but was less friendly to programmers accustomed to C/C++. The final decision to base UnrealScript on a C++/Java variant was based on the desire to map game-specific concepts onto the language definition itself, and the need for speed and familiarity. This turned out to be a good decision, as it has greatly simplified many aspects of the Unreal codebase.

New in Unreal Engine 3

For those who are already familiar with UnrealScript, here's a short overview of the major changes in UnrealScript since UnrealEngine2.

- Replication - replication statements have changed in UE3:
  - The replication block is only used for variables now
  - Function replication is now defined by means of function specifiers (*Server, Client, Reliable*)
- Stacking states - you are now able to push and pop states onto a stack
- UnrealScript Preprocessor - support for macros and conditional compilation
- Debugging Functions - new debugging related functions have been added
- Default properties - processing of the defaultproperties block has been changed\improved a bit
  - Struct defaults - structs can now also have default properties
  - It is no longer allowed to set default values for config or localized variables
  - Defaultproperties are readonly at runtime, it is no longer allowed to do `class'MyClass'.default.variable = 1`
- Dynamic Arrays - Dynamic arrays now have a new function *find()* to find the index of an element
- Dynamic Array Iterators - The foreach operator now functions on dynamic arrays.
- Delegate function arguments - UE3 now allows delegates to be passed as function arguments
- Interfaces - Support for interfaces has been added
- Accessing constants from other classes: `class'SomeClass'.const.SOMECONST`

- Multiple timer support
- Default function argument values - the default value for optional function arguments can now be specified.
- Tooltip Support - Editor property windows now display tooltips when you mouse over a property, if that property has a comment of the form `/** tooltip text */` above its declaration in UnrealScript.
- Metadata Support - Extend in-game and in-editor functionality by associating properties with various types of metadata.

Example program structure

This example illustrates a typical, simple UnrealScript class, and it highlights the syntax and features of UnrealScript. Note that this code may differ from that which appears in the current Unreal source, as this documentation is not synced with the code.

```
//=============================================================================
// TriggerLight.
// A lightsource which can be triggered on or off.
//=============================================================================
class TriggerLight extends Light;

//-----------------------------------------------------------------------------
// Variables.

var() float ChangeTime; // Time light takes to change from on to off.
var() bool bInitiallyOn; // Whether it's initially on.
var() bool bDelayFullOn; // Delay then go full-on.

var ELightType InitialType; // Initial type of light.
var float InitialBrightness; // Initial brightness.
var float Alpha, Direction;
var actor Trigger;

//-----------------------------------------------------------------------------
```

```
// Engine functions.

// Called at start of gameplay.
function BeginPlay()
{
   // Remember initial light type and set new one.
   Disable( 'Tick' );
   InitialType = LightType;
   InitialBrightness = LightBrightness;
   if( bInitiallyOn )
   {
      Alpha = 1.0;
      Direction = 1.0;
   }
   else
   {
      LightType = LT_None;
      Alpha = 0.0;
      Direction = -1.0;
   }
}

// Called whenever time passes.
function Tick( float DeltaTime )
{
   LightType = InitialType;
   Alpha += Direction * DeltaTime / ChangeTime;
   if( Alpha > 1.0 )
   {
      Alpha = 1.0;
      Disable( 'Tick' );
      if( Trigger != None )
         Trigger.ResetTrigger();
   }
   else if( Alpha < 0.0 )
   {
      Alpha = 0.0;
      Disable( 'Tick' );
      LightType = LT_None;
      if( Trigger != None )
```

```
         Trigger.ResetTrigger();
   }
   if( !bDelayFullOn )
      LightBrightness = Alpha * InitialBrightness;
   else if( (Direction>0 &amp;amp;amp;&amp;amp;amp; Alpha!=1) ||
Alpha==0 )
      LightBrightness = 0;
   else
      LightBrightness = InitialBrightness;
}

//-----------------------------------------------------------------
---
// Public states.

// Trigger turns the light on.
state() TriggerTurnsOn
{
   function Trigger( actor Other, pawn EventInstigator )
   {
      Trigger = None;
      Direction = 1.0;
      Enable( 'Tick' );
   }
}

// Trigger turns the light off.
state() TriggerTurnsOff

{
   function Trigger( actor Other, pawn EventInstigator )
   {
      Trigger = None;
      Direction = -1.0;
      Enable( 'Tick' );
   }
}

// Trigger toggles the light.
state() TriggerToggle
```

```
{
   function Trigger( actor Other, pawn EventInstigator )
   {
      log("Toggle");
      Trigger = Other;
      Direction *= -1;
      Enable( 'Tick' );
   }
}

// Trigger controls the light.
state() TriggerControl
{
   function Trigger( actor Other, pawn EventInstigator )
   {
      Trigger = Other;
      if( bInitiallyOn ) Direction = -1.0;
      else Direction = 1.0;
      Enable( 'Tick' );
   }
   function UnTrigger( actor Other, pawn EventInstigator )
   {
      Trigger = Other;
      if( bInitiallyOn ) Direction = 1.0;
      else Direction = -1.0;
      Enable( 'Tick' );
   }
}
```

The key elements to look at in this script are:

- The class declaration. Each class "extends" (derives from) one parent class, and each class belongs to a "package," a collection of objects that are distributed together. All functions and variables belong to a class, and are only accessible through an actor that belongs to that class. There are no system-wide global functions or variables. More Details
- The variable declarations. UnrealScript supports a very diverse set of variable types including most base C/Java types, object references, structs, and arrays. In addition, variables can be made into editable properties,

which designers can access in UnrealEd without any programming. These properties are designated using the `var()` syntax, instead of `var`. More Details

- The functions. Functions can take a list of parameters, and they optionally return a value. Functions can have local variables. Some functions are called by the Unreal engine itself (such as BeginPlay), and some functions are called from other script code elsewhere (such as Trigger). More Details

- The code. All of the standard C and Java keywords are supported, like `for`, `while`, `break`, `switch`, `if`, and so on. Braces and semicolons are used in UnrealScript as in C, C++, and Java.

- Actor and object references. Here you see several cases where a function is called within another object, using an object reference. More Details

- The "state" keyword. This script defines several "states", which are groupings of functions, variables, and code that are executed only when the actor is in that state. More Details

- Note that all keywords, variable names, functions, and object names in UnrealScript are case-insensitive. To UnrealScript, `Demon`, `demON`, and `demon` are the same thing.

The Unreal Virtual Machine

The Unreal Virtual Machine consists of several components: The server, the client, the rendering engine, and the engine support code.

The Unreal server controls all gameplay and interaction between players and actors. In a single-player game, both the Unreal client and the Unreal server are run on the same machine; in an Internet game, there is a dedicated server running on one machine; all players connect to this machine and are clients.

All gameplay takes place inside a "level", a self-contained environment containing geometry and actors. Though UnrealServer may be capable of running more than one level simultaneously, each level operates independently, and are shielded from each other: actors cannot travel between levels, and actors on one level cannot communicate with actors on another level.

Each actor in a map can either be under player control (there can be many players in a network game) or under script control. When an actor is under script control, its script completely defines how the actor moves and interacts with other actors.

With all of those actors running around, scripts executing, and events occuring in the world, you're probably asking how one can understand the flow of execution in an UnrealScript. The answer is as follows:

To manage time, Unreal divides each second of gameplay into "Ticks". A tick is the smallest unit of time in which all actors in a level are updated. A tick typically takes between 1/100th to 1/10th of a second. The tick time is limited only by CPU power; the faster machine, the lower the tick duration is.

Some commands in UnrealScript take zero ticks to execute (i.e. they execute without any game-time passing), and others take many ticks. Functions that require game-time to pass are called "latent functions". Some examples of latent functions include *Sleep*, *FinishAnim*, and *MoveTo*. Latent functions in UnrealScript may only be called from code within a state (the so called "state code"), not from code within a function (that includes functions define within a state).

While an actor is executing a latent function, that actor's state execution doesn't continue until the latent function completes. However, other actors, or the VM, may call functions within the actor. The net result is that all UnrealScript functions can be called at any time, even while latent functions are pending.

In traditional programming terms, UnrealScript acts as if each actor in a level has its own "thread" of execution. Internally, Unreal does not use Windows threads, because that would be very inefficient (Windows 95 and Windows NT do not handle thousands of simultaneous threads efficiently). Instead, UnrealScript simulates threads. This fact is transparent to UnrealScript code, but becomes very apparent when you write C++ code that interacts with UnrealScript.

All UnrealScripts are executed independently of each other. If there are 100 monsters walking around in a level, all 100 of those monsters' scripts are executing simultaneously and independently each "Tick".

Object Hierarchy

Before beginning work with UnrealScript, it's important to understand the high-level relationships of objects within Unreal. The architecture of Unreal is a major departure from that of most other games: Unreal is purely object-oriented (much like COM/ActiveX), in that it has a well-defined object model with support for high-level object oriented concepts such as the object graph, serialization, object lifetime, and polymorphism. Historically, most games have been designed monolithically, with their major functionality hardcoded and unexpandable at the object level, though many games, such as Doom and Quake, have proven to be very expandable at the content level. There is a major benefit to Unreal's form of object-orientation: major new functionality and object types can be added to Unreal at runtime, and this extension can take the form of subclassing, rather than (for example) by modifying a bunch of existing code. This form of extensibility is extremely powerful, as it encourages the Unreal community to create Unreal enhancements that all interoperate.

Object is the parent class of all objects in Unreal. All of the functions in the Object class are accessible everywhere, because everything derives from Object. Object is an abstract base class, in that it doesn't do anything useful. All functionality is provided by subclasses, such as Texture (a texture map), TextBuffer (a chunk of text), and Class (which describes the class of other objects).

Actor (extends Object) is the parent class of all standalone game objects in Unreal. The Actor class contains all of the functionality needed for an actor to move around, interact with other actors, affect the environment, and do other useful game-related things.

Pawn (extends Actor) is the parent class of all creatures and players in Unreal which are capable of high-level AI and player controls.

Class (extends Object) is a special kind of object which describes a class of object. This may seem confusing at first: a class is an object, and a class describes certain objects. But, the concept is sound, and there are many cases where you will deal with Class objects. For example, when you spawn a new actor in UnrealScript, you can specify the new actor's class with a Class object.

With UnrealScript, you can write code for any Object class, but 99% of the time, you will be writing code for a class derived from Actor. Most of the useful UnrealScript functionality is game-related and deals with actors.

Classes

Each script corresponds to exactly one class, and the script begins by declaring the class, the class's parent, and any additional information that is relevent to the class. The simplest form is:

```
class MyClass extends MyParentClass;
```

Here I am declaring a new class named "MyClass", which inherits the functionality of "MyParentClass". Additionally, the class resides in the package named "MyPackage".

Each class inherits all of the variables, functions, and states from its parent class. It can then add new variable declarations, add new functions (or override the existing functions), add new states (or add functionality to the existing states).

The typical approach to class design in UnrealScript is to make a new class (for example a Minotaur monster) which extends an existing class that has most of the functionality you need (for example the Pawn class, the base class of all monsters). With this approach, you never need to reinvent the wheel -- you can simply add the new functionality you want to customize, while keeping all of the existing functionality you don't need to customize. This approach is especially powerful for implementing AI in Unreal, where the built-in AI system provides a tremendous amount of base functionality which you can use as building blocks for your custom creatures.

The class declaration can take several optional specifiers that affect the class:

Native(PackageName)
: Indicates that "this class uses behind-the-scenes C++ support". Unreal expects native classes to contain a C++ implementation in the .EXE. Only native classes are allowed to declare native functions or implement native interfaces. Native classes must always derive from another native class. Native classes create an auto-generated C++ header file with the necessary *glue* to interact with the script variables and specified functions. By default, the *PackageName* is the package that the script class resides in. For example, if the class were in the Engine package, the resulting auto-generated header would be called *EngineClasses.h*.

NativeReplication
: Indicates that replication of variable values for this class is handled in the C++ implementation. Only valid for native classes.

DependsOn(*ClassName*[,*ClassName*,...])

Indicates that *ClassName* is compiled before this class. *ClassName* must specify a class in the same (or a previous) package. Multiple dependency classes can be specified using a single `DependsOn` line delimited by commas, or can be specified using a separate `DependsOn` line for each class.

Abstract

Declares the class as an "abstract base class". This prevents the user from adding actors of this class to the world in UnrealEd or creating instances of this class during the game, because the class isn't meaningful on its own. For example, the "Actor" base class is abstract, while the "Ladder" subclass is not abstract -- you can place a Ladder in the world, but you can't place an Actor in the world. This keyword is propagated to intrinsic child classes, but not to script child classes.

Deprecated

Causes all objects of this class to be loaded but not saved. Any placed instances of deprecated actors will generate warnings for level designers when they load a map in the editor. This keyword is propagated to child classes.

Transient

Says "objects belonging to this class should never be saved on disk". Only useful in conjunction with certain kinds of native classes which are non-persistent by nature, such as players or windows. This keyword is propagated to child classes; child classes can override this flag using the `NotTransient` keyword.

NonTransient

Negates a `Transient` keyword inherited from a base class.

Config(IniName)

Indicates that this class is allowed to store data in the .ini. If there are any configurable variables in the class (declared with "config" or "globalconfig"), causes those variables to be stored in the specified configuration file. This flag is propagated to all child classes and cannot be negated, but child classes can change the .ini file by redeclaring the `Config` keyword and specifying a different IniName. Normally IniName specifies the name of the .ini file to store data in, but several names have a special meaning:

- Config(Engine): Uses the Engine configuration file, which is the name of your game followed by "Engine.ini". For example, ExampleGame's engine configuration file is named ExampleEngine.ini.
- Config(Editor): Uses the Editor configuration file, which is the name of your game followed by "Editor.ini". For example, ExampleGame's editor configuration file is named ExampleEditor.ini.

- Config(Game): Uses the Game configuration file, which is the name of your game followed by "Game.ini". For example, ExampleGame's game configuration file is named ExampleGame.ini.
- Config(Input): Uses the Input configuration file, which is the name of your game followed by "Input.ini". For example, ExampleGame's engine configuration file is named ExampleInput.ini.

PerObjectConfig

Configuration information for this class will be stored per object, where each object has a section in the .ini file named after the object in the format [ObjectName ClassName]. This keyword is propagated to child classes.

PerObjectLocalized

Localized data for this class will be defined on a per-object basis, where each object has a section in the localization file named after the object in the format [ObjectName ClassName]. This keyword is propagated to child classes.

EditInlineNew

Editor. Indicates that objects of this class can be created from the UnrealEd property window (default behavior is that only references to existing objects may be assigned through the property window). This flag is propagated to all child classes; child classes can override this flag using the `NotEditInlineNew` keyword.

NotEditInlineNew

Editor. Negates a `EditInlineNew` keyword inherited from a base class. No effect if no parent classes are using `EditInlineNew`.

Placeable

Editor. Indicates that this class can be created in UnrealEd and placed into a level, UI scene, or kismet window (depending on the class type). This flag is propagated to all child classes; child classes can override this flag using the `NotPlaceable` keyword.

NotPlaceable

Editor. Negates a `Placeable` keyword inherited from a base class. Indicates that this class may not be placed into a level, etc. in UnrealEd.

HideDropDown

Editor. Prevents this class from showing up in UnrealEd property window combo boxes.

HideCategories(*Category*[,*Category*,...])

Editor. Specifies one or more categories that should be hidden in the UnrealEd property window for objects of this class. To hide variables declared with no category, use the name of the class which declares the variable.

ShowCategories(*Category*[,*Category*,...])

Editor. Negates a `HideCategories` keyword inherited from a base class.

AutoExpandCategories(*Category*[,*Category*,...])

Editor. Specifies one or more categories that should be automatically expanded in the UnrealEd property window for objects of this class. To auto-expand variables declared with no category, use the name of the class which declares the variable.

Collapsecategories

Editor. Indicates that properties of this class should not be grouped in categories in UnrealEd property windows. This keyword is propagated to child classes; child classes can override this flag using the `DontCollapseCategories` keyword.

DontCollapseCategories

Editor. Negates a `CollapseCatogories` keyword inherited from a base class.

Within *ClassName*

Advanced. Indicates that objects of this class cannot exist without an instance of *ClassName*. In order to create an object of this class, you must specify an instance of *ClassName* as the `Outer` object. This keyword must be the first to follow the class declaration itself.

Inherits(*ClassName*[,*ClassName*,...])

Advanced. Used for multiple inheritance - specifies the additional base classes. Multiple bases can be specified using a single `Inherits` line delimited by commas, or can be specified using a separate `Inherits` line for each base class. Only valid for native classes. Multiple inheritance from two UObject-derived classes is not supported.

Implements(*ClassName*[,*ClassName*,...])

Advanced. Specifies one of more interface classes which are this class will implement. Multiple interfaces can be specified using a single `Implements` line delimited by commas, or can be specified using a separate `Implements` line for each interface class. Only native classes can implement native interfaces.

NoExport

Advanced. Indicates that this class's C++ declaration should not be included in the automatically-generated C++ header file by the script compiler. The C++ class declaration must be defined manually in a separate header file. Only valid for native classes.

Variables

Variable types

Built-in types

Here are some examples of instance variable declarations in UnrealScript:

```
var int a;               // Declare an integer variable named "A".
var byte Table[64];         // Declare a static array of 64 bytes
named "Table".
var string PlayerName;      // Declare a string variable named
"PlayerName".
var actor Other;         // Declare a variable which can be assigned
a reference to an Actor instance.
var() float MaxTargetDist;   // Declare a float variable named
"MaxTargetDist" and allow its value to be modified from an UnrealEd
property window.
```

Variables can appear in two kinds of places in UnrealScript: Instance variables, which apply to an entire object, appear immediately after the class declarations or within struct declarations. Local variables appear within a function, and are only active while that function executes. Instance variables are declared with the `var` keyword. Local variables are declared with the `local` keyword, such as:

```
function int Foo()
{
   local int Count;
   Count = 1;
   return Count;
}
```

Here are the built-in variable types supported in UnrealScript:

- **byte**: A single-byte value ranging from `0` to `255`.
- **int**: A 32-bit integer value.
- **bool**: A boolean value: either `true` or `false`.
- **float**: A 32-bit floating point number.
- **string**: A string of characters. (see Unreal Strings)
- **constant**: A variable that cannot be modified.

- **enumeration**: A variable that can take on one of several predefined named integer values. For example, the ELightType enumeration defined in the Actor script describes a dynamic light and takes on a value like `LT_None`, `LT_Pulse`, `LT_Strobe`, and so on.

Aggregate data types

array<Type>
A variable length array of `Type`.

struct
Similar to C structures, UnrealScript structs let you create new variable types that contain sub-variables. For example, two commonly-used Unreal structs are `vector`, which consists of an X, Y, and Z component; and `rotator`, which consists of a pitch, yaw, and roll component.

Unreal types

Name
The name of an item in Unreal (such as the name of a function, state, class, etc). Names are stored as an index into the global name table. Names correspond to simple strings of up to 64 characters. Names are not like strings in that they are immutable once created (see Unreal Strings for more information).

Object and Actor references
A variable that refers to another object or actor in the world. For example, the Pawn class has an "Enemy" actor reference that specifies which actor the pawn should be trying to attack. Object and actor references are very powerful tools, because they enable you to access the variables and functions of another actor. For example, in the Pawn script, you can write *Enemy.Damage(123)* to call your enemy's Damage function -- resulting in the enemy taking damage. Object references may also contain a special value called `None`, which is the equivalent of the C `NULL` pointer: it says "this variable doesn't refer to any object".

Delegate
Holds a reference to an unrealscript function.

Variable specifiers

Variables may also contain additional specifiers such as `const` that further describe the variable. Actually, there are quite a lot of specifiers which you wouldn't expect to see in a general-purpose programming language, mainly as a result of wanting UnrealScript to natively support many game- and environment-specific concepts:

config
: This variable will be made configurable. The current value can be saved to the ini file and will be loaded when created. Cannot be given a value in default properties. Implies const.

globalconfig
: Works just like config except that you can't override it in a subclass. Cannot be given a value in default properties. Implies const.

localized
: The value of this variable will have a localized value defined. Mostly used for strings. Implies const. Read more about this in the Localization Reference and Unreal Strings.

const
: Treats the contents of the variable as a constant. In UnrealScript, you can read the value of const variables, but you can't write to them. "Const" is only used for variables which the engine is responsible for updating, and which can't be safely updated from UnrealScript, such as an actor's Location (which can only be set by calling the MoveActor function).

private
: The variable is private, and may only be accessed by the class's script; no other classes (including subclasses) may access it.

protected
: The variable can only be accessed from the class and it's subclasses, not from other classes.

repnotify
: Actors should be notified (via the ReplicatedEvent function) when this value for this property is received via replication.

deprecated
: Indicates that this variable is going to be removed in the near future, and should no longer be accessible in the editor. Deprecated properties are loaded, but not saved.

instanced
Object properties only. When an instance of this class is created, it will be given a unique copy of the object assigned to this variable in defaults. Used for instancing subobjects defined in class default properties.

databinding
This property can be manipulated by the data store system.

editoronly
This property's value will only be loaded when running UnrealEd or a commandlet. During the game, the value for this property is discarded.

notforconsole
This property's value will only be loaded when running on the PC. On consoles, the value for this property is discarded.

editconst
Editor. The variable can be seen in UnrealEd but not edited. A variable that is editconst is *not* implictly "const".

editfixedsize
Editor. Only useful for dynamic arrays. This will prevent the user from changing the length of an array via the UnrealEd property window.

editinline
Editor. Allows the user to edit the properties of the object referenced by this variable within UnrealEd's property inspector (only useful for object references, including arrays of object reference).

editinlineuse
Editor. In addition to the behavior associated with *editinline*, adds a "Use" button next to this object reference in the editor

noclear
Editor. Allows this object reference to be set to None from the editor.

interp
Editor. Indicates that the value can be driven over time by a Float or Vector Property Track in Matinee.

input
Advanced. Makes the variable accessible to Unreal's input system, so that input (such as button presses and joystick movements) can be directly mapped onto it. Only relevent with variables of type "byte" and "float".

transient
Advanced. Declares that the variable is for temporary use, and isn't part of the object's persistent state. Transient variables are not saved to disk. Transient variables are initialized to the class's default value for that variable when an object is loaded.

duplicatetransient
Advanced. Indicates that the variable's value should be reset to the class default value when creating a binary duplicate of an object (via StaticDuplicateObject).

noimport
Advanced. Indicates that this variable should be skipped when importing T3D text. In other words, the value of this variable will not be transferred to new object instances when importing or copy/pasting objects.

native
Advanced. Declares that the variable is loaded and saved by C++ code, rather than by UnrealScript.

export
Advanced. Only useful for object properties (or arrays of objects). Indicates that the object assigned to this property should be exported in its entirety as a subobject block when the object is copied (for copy/paste) or exported to T3D, as opposed to just outputting the object reference itself.

noexport
Advanced. Only useful for native classes. This variable should not be included in the auto-generated class declaration.

nontransactional
Advanced. Indicates that changes to this variable value will not be included in the editor's undo/redo history.

pointer{*type*}
Advanced. This variable is a pointer to *type*. (The *type* is optional). Note the syntax is: pointer varname{*type*}.

init
Advanced. This property should be exported to the header file as an FString or TArray, rather than an FStringNoInit or TArrayNoInit. Only applicable to strings and dynamic arrays declared in native classes. 'Init' properties should not be given default values, as the default value will be cleared when the object is created. (See Unreal Strings and Native Strings)

out
This specifier is only valid for function parameters. See Functions for more details.

coerce
This specifier is only valid for function parameters. See Functions for more details.

optional
This specifier is only valid for function parameters. See Functions for more details.

skip

This specifier is only valid for operator function parameters. Only used for logical operators like && and ||. Prevents evaluation if the outcome of an expression can already be determined. Example: FALSE && ++b==10 (More details).

Editability

In UnrealScript, you can make an instance variable "editable", so that users can edit the variable's value in UnrealEd. This mechanism is responsible for the entire contents of the "Actor Properties" dialog in UnrealEd: everything you see there is simply an UnrealScript variable, which has been declared editable.

The syntax for declaring an editable variable is as follows:

```
var() int MyInteger; // Declare an editable integer in the default
                     // category.

var(MyCategory) bool MyBool; // Declare an editable integer in
                             // "MyCategory".
```

You can also declare a variable as `editconst`, which means that the variable should be visible but *not* editable UnrealEd. Note that this only prevents the variable from being changed in the editor, not in script. If you want a variable that is truly `const` but still visible in the editor, you must declare it `const editconst`:

```
// MyBool is visible but not editable in UnrealEd
var(MyCategory) editconst bool MyBool;

// MyBool is visible but not editable in UnrealEd and
// not changeable in script
var(MyCategory) const editconst bool MyBool;

// MyBool is visible and can be set in UnrealEd but
// not changeable in script
var(MyCategory) const bool MyBool;
```

Arrays

Arrays are declared using the following syntax:

```
var int MyArray[20]; // Declares an array of 20 ints.
```

UnrealScript supports only single-dimensional arrays, though you can simulate multidimensional arrays by carrying out the row/column math yourself. For information on Dynamic Arrays, see below in the Advanced Language Features section.

Structs

An UnrealScript struct is a way of cramming a bunch of variables together into a new kind of super-variable called a struct. UnrealScript structs are much like C structs, in that they can contain variables, arrays, and other structs, but UnrealScript structs cannot contain functions.

You can declare a struct as follows:

```
// A point or direction vector in 3D space.
struct Vector
{
   var float X;
   var float Y;
   var float Z;
};
```

Once you declare a struct, you are ready to start declaring specific variables of that struct type:

```
// Declare a bunch of variables of type Vector.
var Vector Position;
var Vector Destination;
```

To access a component of a struct, use code like the following.

```
function MyFunction()
{
   Local Vector A, B, C;

   // Add some vectors.
   C = A + B;

   // Add just the x components of the vectors.
   C.X = A.X + B.X;

   // Pass vector C to a function.
   SomeFunction( C );

   // Pass certain vector components to a function.
   OtherFunction( A.X, C.Z );
}
```

You can do anything with Struct variables that you can do with other variables: you can assign variables to them, you can pass them to functions, and you can access their components.

There are several Structs defined in the Object class, which are used throughout Unreal. You should become familiar with their operation, as they are fundamental building blocks of scripts:

Vector
A unique 3D point or vector in space, with an X, Y, and Z component.

Plane
Defines a unique plane in 3D space. A plane is defined by its X, Y, and Z components (which are assumed to be normalized) plus its W component, which represents the distance of the plane from the origin, along the plane's normal (which is the shortest line from the plane to the origin).

Rotator
A rotation defining a unique orthogonal coordinate system. A rotator contains Pitch, Yaw, and Roll components.

Coords
An arbitrary coordinate system in 3D space.

Color
An RGB color value.
Region
Defines a unique convex region within a level.

Struct specifiers

Structs may also have a few specifiers that affect all instances of the struct.

atomic
Indicates that this struct should always be serialized as a single unit; if any property in the struct differs from its defaults, then all elements of the struct will be serialized.
atomicwhencooked
applies the 'atomic' flag only when working with cooked package data.
immutable
Indicates that this struct uses binary serialization (reduces disk space and improves serialization performance); it is unsafe to add/remove members from this struct without incrementing the package version.
immutablewhencooked
applies the 'immutable' flag only when working with cooked package data.
strictconfig
Indicates that when the struct property has 'config/globalconfig', only properties marked config/globalconfig within this struct can be read from .ini (without this flag, all properties in the struct are configurable if the property is)

Enumerations

Enumerations exist in UnrealScript as a convenient way to declare variables that can contain "one of" a bunch of keywords. For example, the actor class contains the enumeration `EPhysics`, which describes the physics which Unreal should apply to the actor. This can be set to one of the predefined values like `PHYS_None`, `PHYS_Walking`, `PHYS_Falling`, and so on.

Internally, enumerations are stored as byte variables. In designing UnrealScript, enumerations were not seen as a necessity, but it makes code so much easier to read to see that an actor's physics mode is being set to `PHYS_Swimming` than (for example) `3`.

Here is sample code that declares enumerations.

```
// Declare the EColor enumeration, with three values.
enum EColor
{
   CO_Red,
   CO_Green,
   CO_Blue
};

// Now, declare two variables of type EColor.
var EColor ShirtColor, HatColor;

// Alternatively, you can declare variables and
// enumerations together like this:
var enum EFruit
{
   FRUIT_Apple,
   FRUIT_Orange,
   FRUIT_Bannana
} FirstFruit, SecondFruit;
```

In the Unreal source, we always declare enumeration values like `LT_Steady`, `PHYS_Falling`, and so on, rather than as simply "Steady" or "Falling". This is just a matter of programming style, and is not a requirement of the language.

UnrealScript only recognizes unqualified enum tags (like `FRUIT_Apple`) in classes where the enumeration was defined, and in its subclasses. If you need to refer to an enumeration tag defined somewhere else in the class hierarchy, you must "qualify it":

```
FRUIT_Apple         // If Unreal can't find this enum tag...
EFruit.FRUIT_Apple  // Then qualify it like this.
```

Constants

In UnrealScript, you can specify constant literal values for nearly all data types:

- Integer and byte constants are specified with simple numbers, for example: `123`. If you must specify an integer or byte constant in hexadecimal format, use i.e.: `0x123`

- Floating point constants are specified with decimal numbers like: `456.789`
- String constants must be enclosed in double quotes, for example: `"MyString"`
- Name constants must be enclosed in single quotes, for example `'MyName'`
- Vector constants contain X, Y, and Z values like this: `vect(1.0,2.0,4.0)`
- Rotator constants contain Pitch, Yaw, and Roll values like this: `Rot(0x8000,0x4000,0)`
- The `None` constant refers to "no object" (or equivalently, "no actor").
- The `Self` constant refers to "this object" (or equivalently, "this actor"), i.e. the object whose script is executing.
- General object constants are specified by the object type followed by the object name in single quotes, for example: `texture'Default'`
- `EnumCount` gives you the number of elements in an enumeration, for example: `ELightType.EnumCount`
- `ArrayCount` gives you the number of elements in an static array, for example: `ArrayCount(Touching)`

You can use the "const" keyword to declare constants that you can later refer to by name. For example:

```
const LargeNumber=123456;
const PI=3.14159;
const MyName="Tim";
const Northeast=Vect(1.0,1.0,0.0);
```

Constants can be defined within classes or within structs.

To access a constant which was declared in another class, use the "class'classname'.const.constname" syntax, for example:

```
class'Pawn'.const.LargeNumber
```

Object and actor reference variables

You can declare a variable that refers to an actor or object like this:

```
var actor A; // An actor reference.
var pawn P; // A reference to an actor in the Pawn class.
var texture T; // A reference to a texture object.
```

The variable "P" above is a reference to an actor in the Pawn class. Such a variable can refer to any actor that belongs to a subclass of Pawn. For example, P might refer to a Brute, or a Skaarj, or a Manta. It can be any kind of Pawn. However, P can never refer to a Trigger actor (because Trigger is not a subclass of Pawn).

One example of where it's handy to have a variable referring to an actor is the Enemy variable in the Pawn class, which refers to the actor that the Pawn is trying to attack.

When you have a variable that refers to an actor, you can access that actor's variables, and call its functions. For example:

```
// Declare two variables that refer to a pawns.
var pawn P, Q;

// Here is a function that makes use of P.
// It displays some information about P.
function MyFunction()
{
   // Set P's enemy to Q.
   P.Enemy = Q;

   // Tell P to play his running animation.
   P.PlayRunning();
}
```

Variables that refer to actors always either refer to a valid actor (any actor that actually exists in the level), or they contain the value `None`. None is equivalent to the C/C++ `NULL` pointer. However, in UnrealScript, it is safe to access variables and call functions with a `None` reference; the result is always zero.

Note that an object or actor reference "points to" another actor or object, it doesn't "contain" an actor or object. The C equivalent of an actor reference is a pointer to an object in the AActor class (in C, you'd say an AActor*). For example, you could have two monsters in the world, Bob and Fred, who are fighting each other. Bob's "Enemy" variable would "point to" Fred, and Fred's "Enemy" variable would "point to" Bob.

Unlike C pointers, UnrealScript object references are always safe and infallible. It is impossible for an object reference to refer to an object that doesn't exist or is invalid (other than the special-case `None` value). In UnrealScript, when an actor is destroyed, all references to it are automatically set to `None`.

Class Reference Variables

In Unreal, classes are objects just like actors, textures, and sounds are objects. Class objects belong to the class named "class". Now, there will often be cases where you'll want to store a reference to a class object, so that you can spawn an actor belonging to that class (without knowing what the class is at compile-time). For example:

```
var() class C;
var actor A;
A = Spawn( C ); // Spawn an actor belonging to some arbitrary class C.
```

Now, be sure not to confuse the roles of a class C, and an object O belonging to class C (referred to as an "instance" of class C). To give a really shaky analogy, a class is like a pepper grinder, and an instance of that class is like pepper. You can use the pepper grinder (the class) to create pepper (objects of that class) by turning the crank (calling the Spawn function)...BUT, a pepper grinder (a class) is not pepper (an object belonging to the class), so you MUST NOT TRY TO EAT IT!

When declaring variables that reference class objects, you can optionally use the syntax **class<metaclass>** to limit the classes that can be referenced by the variable to classes of type *metaclass* (and its child classes). For example, in the declaration:

```
var class<actor> ActorClass;
```

The variable ActorClass may only reference a class that extends the "actor" class. This is useful for improving compile-time type checking. For example, the Spawn function takes a class as a parameter, but only makes sense when the given class is a subclass of Actor, and the class<classlimitor> syntax causes the compiler to enforce that requirement.

As with dynamic object casting, you can dynamically cast classes like this:

```
// casts the result of SomeFunctionCall() a class of type Actor (or
subclasses of Actor)
class<actor>( SomeFunctionCall() )
```

Expressions

Assignment

To assign a value to a variable, use "=" like this:

```
function Test()
{
   local int i;
   local string s;
   local vector v, q;

   i = 10;       // Assign a value to integer variable i.
   s = "Hello!"; // Assign a value to string variable s.
   v = q;        // Copy value of vector q to v.
}
```

In UnrealScript, whenever a function or other expression requires a certain type of data (for example, a "float"), and you specify a different type of data (for example, an "int), the compiler will try to automatically convert the value you give to the proper type. Conversions among all the numerical data types (byte, int, and float) happen automatically, without any work on your part.

UnrealScript is also able to convert many other built-in data types to other types, if you explicitly convert them in code. The syntax for this is:

```
function Test()
{
   local int i;
   local string s;
   local vector v, q;
   local rotator r;
```

```
   s = string(i);     // Convert integer i to a string, and assign
it to s.
   s = string(v);     // Convert vector v to a string, and assign it
to s.
   v = q + vector(r); // Convert rotator r to a vector, and add q.
}
```

Here is the complete set of non-automatic conversions you can use in UnrealScript:

- String to Byte, Int, Float: Tries to convert a string like `"123"` to a value like `123`. If the string doesn't represent a value, the result is `0`.
- Byte, Int, Float, Vector, Rotator to String: Converts the number to its textual representation.
- String to Vector, Rotator: Tries to parse the vector or rotator's textual representation.
- String to Bool: Converts the case-insensitive words `"True"` or `"False"` to `True` and `False`; converts any non-zero value to `True`; everything else is `False`.
- Bool to String: Result is either `"True"` or `"False"`.
- Byte, Int, Float, Vector, Rotator to Bool: Converts nonzero values to `True`; zero values to `False`.
- Bool to Byte, Int, Float: Converts `True` to `1`; `False` to `0`.
- Name to String: Converts the name to the text equivalent.
- Rotator to Vector: Returns a vector facing "forward" according to the rotator.
- Vector to Rotator: Returns a rotator pitching and yawing in the direction of the vector; roll is zero.
- Object (or Actor) to Int: Returns an integer that is guaranteed unique for that object.
- Object (or Actor) to Bool: Returns `False` if the object is `None`; `True` otherwise.
- Object (or Actor) to String: Returns a textual representation of the object.

Converting object references among classes

Just like the conversion functions above, which convert among simple data types, in UnrealScript you can convert actor and object references among various types. For example, all actors have a variable named "Target", which is a reference to another actor. Say you are writing a script where you need to check

and see if your Target belongs to the "Pawn" actor class, and you need to do something special with your target that only makes sense when it's a pawn -- for example, you need to call one of the Pawn functions. The actor cast operators let you do this. Here's an example:

```
var actor Target;
//...

function TestActorConversions()
{
   local Pawn P;

   // Cast Target to Pawn and assign the result to P.  If Target is
not a Pawn (or subclass of Pawn), then the value assigned to P will
be None.
   P = Pawn(Target);
   if( P != None )
   {
      // Target is a pawn, so set its Enemy to Self.
      P.Enemy = Self;
   }
   else
   {
      // Target is not a pawn.
   }
}
```

To perform an actor conversion, type the class name followed by the actor expression you wish to convert, in parenthesis. Such conversions will either succeed or fail based on whether the conversion is sensible. In the above example, if your Target is referencing a Trigger object rather than a pawn, the expression Pawn(Target) will return "None", since a Trigger can't be converted to a Pawn. However, if your Target is referencing a Brute object, the conversion will successfully return the Brute, because Brute is a subclass of Pawn.

Thus, actor conversions have two purposes: First, you can use them to see if a certain actor reference belongs to a certain class. Second, you can use them to convert an actor reference from one class to a more specific class. Note that these conversions don't affect the actor you're converting at all -- they just enable UnrealScript to treat the actor reference as if it were a more specific type

and allow you access the properties and methods declared in the more derived class.

Another example of conversions lies in the Inventory script. Each Inventory actor is owned by a Pawn, even though its Owner variable can refer to any Actor (because Actor.Owner is a variable of type Actor). So a common theme in the Inventory code is to cast Owner to a Pawn, for example:

```
// Called by engine when destroyed.
function Destroyed()
{
    // Remove from owner's inventory.
    if( Pawn(Owner)!=None )
        Pawn(Owner).DeleteInventory( Self );
}
```

Functions

Declaring Functions

In UnrealScript, you can declare new functions and write new versions of existing functions (overwrite functions). Functions can take one or more parameters (of any variable type UnrealScript supports), and can optionally return a value. Though most functions are written directly in UnrealScript, you can also declare functions that can be called from UnrealScript, but which are implemented in C++ and reside in a DLL. The Unreal technology supports all possible combinations of function calling: The C++ engine can call script functions; script can call C++ functions; and script can call script.

Here is a simple function declaration. This function takes a vector as a parameter, and returns a floating point number:

```
// Function to compute the size of a vector.
function float VectorSize( vector V )
{
   return sqrt( V.X * V.X + V.Y * V.Y + V.Z * V.Z );
}
```

The word `function` always precedes a function declaration. It is followed by the optional return type of the function (in this case, `float`), then the function name, and then the list of function parameters enclosed in parenthesis.

When a function is called, the code within the brackets is executed. Inside the function, you can declare local variables (using the `local` keyword), and execute any UnrealScript code. The optional `return` keyword causes the function to immediately return a value.

You can pass any UnrealScript types to a function (including arrays), and a function can return any type.

By default, any local variables you declare in a function are initialized to zero.

Function calls can be recursive. For example, the following function computes the factorial of a number:

```
// Function to compute the factorial of a number.
function int Factorial( int Number )
{
   if( Number <= 0 )
      return 1;
   else
      return Number * Factorial( Number - 1 );
}
```

Some UnrealScript functions are called by the engine whenever certain events occur. For example, when an actor is touched by another actor, the engine calls its *Touch* function to tell it who is touching it. By writing a custom *Touch* function, you can take special actions as a result of the touch occurring:

```
// Called when something touches this actor.
function Touch( actor Other )
{
   Log( "I was touched!")
   Other.Message( "You touched me!" );
}
```

The above function illustrates several things. First of all, the function writes a message to the log file using the *Log* command (which is the equivalent of Basic's "print" command and C's "printf", with the exception on formatting rules).

Second, it calls the "Message" function residing in the actor Other. Calling functions in other actors is a common action in UnrealScript, and in object-oriented languages like Java in general, because it provides a simple means for actors to communicate with each other.

Function parameter specifiers

When you normally call a function, UnrealScript makes a local copy of the parameters you pass the function. If the function modifies some of the parameters, those don't have any effect on the variables you passed in. For example, the following program:

```
function int DoSomething( int x )
{
   x = x * 2;
   return x;
}
function int DoSomethingElse()
{
   local int a, b;

   a = 2;
   log( "The value of a is " $ a );

   b = DoSomething( a );
   log( "The value of a is " $ a );
   log( "The value of b is " $ b );
}
```

Produces the following output when DoSomethingElse is called:

```
The value of a is 2
The value of a is 2
The value of b is 4
```

In other words, the function DoSomething was futzing with a local copy of the variable "a" which was passed to it, and it was not affecting the real variable "a".

The `out` specifier lets you tell a function that it should actually modify the variable that is passed to it, rather than making a local copy. This is useful, for example, if you have a function that needs to return several values to the caller.

You can juse have the caller pass several variables to the function which are `out` values. For example:

```
// Compute the minimum and maximum components of a vector.
function VectorRange( vector V, out float Min, out float Max )
{
   // Compute the minimum value.
   if ( V.X<V.Y &amp;amp;&amp;amp; V.X<V.Z ) Min = V.X;
   else if( V.Y<V.Z ) Min = V.Y;
   else Min = V.Z;

   // Compute the maximum value.
   if ( V.X>V.Y &amp;amp;&amp;amp; V.X>V.Z ) Max = V.X;
   else if( V.Y>V.Z ) Max = V.Y;
   else Max = V.Z;
}
```

Without the `out` keyword, it would be painful to try to write functions that had to return more than one value. Out parameters are passed by reference so modifying the parameter's value in the function will immediately affect the original. This can also be used as an optimization for large values, similiarly to C++, by specifying "const out".

With the `optional` keyword, you can make certain function parameters optional, as a convenience to the caller. For UnrealScript functions, optional parameters which the caller doesn't specify are set to the default value given in the function declaration, or zero (e.g. 0, false, "", none) if no value was specified in the function signature. For native functions, the default values of optional parameters depends on the function. For example, the Spawn function takes an optional location and rotation, which default to the spawning actor's location and rotation. The default value of optional arguments can be specified by adding `= value`. For example `function myFunc(optional int x = -1)`.

The `coerce` keyword forces the caller's parameters to be converted to the specified type (even if UnrealScript normally would not perform the conversion automatically). This is useful for functions that deal with strings, so that the parameters are automatically converted to strings for you. (See Unreal Strings)

Function overriding

"Function overriding" refers to writing a new version of a function in a subclass. For example, say you're writing a script for a new kind of monster called a Demon. The Demon class, which you just created, extends the Pawn class. Now, when a pawn sees a player for the first time, the pawn's "SeePlayer" function is called, so that the pawn can start attacking the player. This is a nice concept, but say you wanted to handle "SeePlayer" differently in your new Demon class. How do you do this? Function overriding is the answer.

To override a function, just cut and paste the function definition from the parent class into your new class. For example, for SeePlayer, you could add this to your Demon class.

```
// New Demon class version of the Touch function.
function SeePlayer( actor SeenPlayer )
{
   log( "The demon saw a player" );
   // Add new custom functionality here...
}
```

Function overriding is the key to creating new UnrealScript classes efficiently. You can create a new class that extends an existing class. Then, all you need to do is override the functions that you want to be handled differently. This enables you to create new kinds of objects without writing gigantic amounts of code.

Several functions in UnrealScript are declared as `final`. The `final` keyword (which appears immediately before the word `function`) says "this function cannot be overridden by child classes". This should be used in functions that you know nobody would want to override, because it results in faster script code. For example, say you have a *VectorSize* function that computes the size of a vector. There's absolutely no reason anyone would ever override that, so declare it as `final`. On the other hand, a function like *Touch* is very context-dependent and should not be final.

Advanced function specifiers

Static

A static function acts like a C global function, in that it can be called without having a reference to an object of the class. Static functions can call other

static functions, and can access the default values of variables. Static functions cannot call non-static functions and they cannot access instance variables (since they are not executed with respect to an instance of an object). Unlike languages like C++, static functions are virtual and can be overridden in child classes. This is useful in cases where you wish to call a static function in a variable class (a class not known at compile time, but referred to by a variable or an expression).

Singular

The `singular` keyword, which appears immediately before a function declaration, prevents a function from calling itself recursively. The rule is this: If a certain actor is already in the middle of a singular function, any subsequent calls to singular functions will be skipped over. This is useful in avoiding infinite-recursive bugs in some cases. For example, if you try to move an actor inside of your *Bump* function, there is a good chance that the actor will bump into another actor during its move, resulting in another call to the *Bump* function, and so on. You should be very careful in avoiding such behavior, but if you can't write code with complete confidence that you're avoiding such potential recursive situations, use the `singular` keyword.

Native

You can declare UnrealScript functions as `native`, which means that the function is callable from UnrealScript, but is actually implemented (elsewhere) in C++. For example, the Actor class contains a lot of native function definitions, such as:

```
native(266) final function bool Move( vector Delta );
```

The number inside the parenthesis after the `native` keyword corresponds to the number of the function as it was declared in C++ (using the `AUTOREGISTER_NATIVE` macro), and is only required for operator functions. The native function is expected to reside in the DLL named identically to the package of the class containing the UnrealScript definition.

NoExport

Only used for native functions. Declares that the C++ function declaration for this native function should not be exported. Only the declaration for the glue version of the function will be exported.

Exec

Indicates that this function can be executed by typing the name of the function into the console. Only valid in certain classes.

Latent

Declares that a native function is latent, meaning that it can only be called from state code, and it may return after some game-time has passed.

Iterator

Declares that a native function is an iterator, which can be used to loop through a list of actors using the `foreach` command.

Simulated

Declares that a function may execute on the client-side when an actor is either a simulated proxy or an autonomous proxy. All functions that are both native and final are automatically simulated as well.

Server

Declares that a function should be sent to the server for execution instead of running on the local client.

Client

Declares that a function should be sent to the owning client for execution instead of running on the server. This flag also implicitly sets the **simulated** flag for the function.

Reliable

Declares that a replicated function (marked with either **server** or **client**) should be reliably sent, meaning it will always eventually reach the other end in order relative to other replication on that Actor.

Unreliable

Declares that a replicated function (marked with either **server** or **client**) should be unreliably sent, meaning that it is not guaranteed to reach the other end in any particular order or at all and may be skipped completely if there is insufficient bandwidth available.

Private, Protected

These keywords have the same meaning as the corresponding variable keywords.

Operator, PreOperator, PostOperator

These keywords are for declaring a special kind of function called an operator (equivalent to C++ operators). This is how UnrealScript knows about all of the built-in operators like "+", "-", "==", and "||". I'm not going into detail on how operators work in this document, but the concept of operators is similar to C++, and you can declare new operator functions and keywords as UnrealScript functions or native functions.

Event

The `event` keyword has the same meaning to UnrealScript as `function`". `However, when you export a C++ header file from Unreal using =unreal -make -h`, UnrealEd automatically generates a C++ -> UnrealScript

calling stub for each "event". This automatically keeps C++ code synched up with UnrealScript functions and eliminates the possibility of passing invalid parameters to an UnrealScript function. For example, this bit of UnrealScript code:

```
event Touch( Actor Other )
{ ... }
```

Generates code similar to the following in EngineClasses.h:

```
void eventTouch(class AActor* Other)
{
    FName N("Touch",FNAME_Intrinsic);
    struct {class AActor* Other; } Parms;
    Parms.Other=Other;
    ProcessEvent(N, &Parms);
}
```

Thus enabling you to call the UnrealScript function from C++ like this:

```
AActor *SomeActor, *OtherActor;
SomeActor->eventTouch(OtherActor);
```

Const

Can only be used with native declared function and this specifier is added *after* the function declaration. This specifier will determine whether this function should be exported as 'const' in the generated header. Example usage:

```
native function int doSomething(string myData) const;
```

Control Structures

UnrealScript supports all the standard flow-control statements of C/C++/Java:

Repetition Structures

For Loops

"For" loops let you cycle through a loop as long as some condition is met. For example:

```
// Example of "for" loop.
function ForExample()
{
   local int i;
   log( "Demonstrating the for loop" );
   for( i=0; i<4; i++ )
   {
      log( "The value of i is " $ i );
   }
   log( "Completed with i=" $ i);
}
```

The output of this loop is:

```
Demonstrating the for loop
The value of i is 0
The value of i is 1
The value of i is 2
The value of i is 3
Completed with i=4
```

In a for loop, you must specify three expressions separated by semicolons. The first expression is for initializing a variable to its starting value. The second expression gives a condition which is checked before each iteration of the loop executes; if this expression is true, the loop executes. If it's false, the loop terminates. The third condition gives an expression which increments the loop counter.

Though most "for" loop expressions just update a counter, you can also use "for" loops for more advanced things like traversing linked lists, by using the appropriate initialization, termination, and increment expressions.

In all of the flow control statements, you can either execute a single statement, without brackets, as follows:

```
for( i=0; i<4; i++ )
   log( "The value of i is " $ i );
```

Or you can execute multiple statements, surrounded by brackets, like this:

```
for( i=0; i<4; i++ )
{
   log( "The value of i is" );
   log( i );
}
```

Do Loops

"Do" loops let you cycle through a loop while some ending expression is true. Note that Unreal uses `do-until` syntax, which differs from C/Java (which use `do-while`).

```
// Example of "do" loop.
function DoExample()
{
   local int i;
   log( "Demonstrating the do loop" );
   do
   {
      log( "The value of i is " $ i );
      i = i + 1;
   } until( i == 4 );
   log( "Completed with i=" $ i);
}
```

The output of this loop is:

```
Demonstrating the do loop
The value of i is 0
The value of i is 1
The value of i is 2
The value of i is 3
Completed with i=4
```

While Loops

"While" loops let you cycle through a loop while some starting expression is true.

```
// Example of "while" loop.
function WhileExample()
{
   local int i;
   log( "Demonstrating the while loop" );
   while( i < 4 )
   {

      log( "The value of i is " $ i );
      i = i + 1;
   }
   log( "Completed with i=" $ i);
}
```

The output of this loop is:

```
Demonstrating the do loop
The value of i is 0
The value of i is 1
The value of i is 2
The value of i is 3
Completed with i=4
```

Continue

The "continue" command will jump back to the beginning of the loop, so everything after the continue command isn't executed. This can be used to skip the loop code in certain cases.

```
function ContinueExample()
{
   local int i;
   log( "Demonstrating continue" );
   for( i=0; i<4; i++ )
   {
      if( i == 2 )
         continue;
```

```
      log( "The value of i is " $ i );
   }
   log( "Completed with i=" $ i );
}
```

The output of this loop is:

```
Demonstrating break
The value of i is 0
The value of i is 1
The value of i is 3
Completed with i=4
```

Break

The "break" command exits out of the nearest loop ("For", "Do", or "While").

```
function BreakExample()
{
   local int i;
   log( "Demonstrating break" );
   for( i=0; i<10; i++ )
   {
      if( i == 3 )
         break;
      log( "The value of i is " $ i );
   }
   log( "Completed with i=" $ i );
}
```

The output of this loop is:

```
Demonstrating break
The value of i is 0
The value of i is 1
The value of i is 2
Completed with i=3
```

Note that the "break" command can also be used to skip the remainder of a conditional statement("switch").

Selection Structures

If-Then-Else Statements

"If", "Else If", and "Else" let you execute code if certain conditions are met.

```
// Example of simple "if".
if( LightBrightness < 20 )
   log( "My light is dim" );

// Example of "if-else".
if( LightBrightness < 20 )
   log( "My light is dim" );
else
   log( "My light is bright" );

// Example if "if-else if-else".
if( LightBrightness < 20 )
   log( "My light is dim" );
else if( LightBrightness < 40 )
   log( "My light is medium" );
else if( LightBrightness < 60 )
   log( "My light is kinda bright" );
else
   log( "My light is very bright" );

// Example if "if" with brackets.
if( LightType == LT_Steady )
{
   log( "Light is steady" );
}
else
{
   log( "Light is not steady" );
}
```

Case Statements

"Switch", "Case", "Default", and "Break" let you handle lists of conditions easily.

```
// Example of switch-case.
function TestSwitch()
{
   // Executed one of the case statements below, based on
   // the value in LightType.
   switch( LightType )
   {
      case LT_None:
         log( "There is no lighting" );
         break;
      case LT_Steady:
         log( "There is steady lighting" );
         break;
      case LT_Backdrop:
         log( "There is backdrop lighting" );
         break;
      default:
         log( "There is dynamic" );
         break;
   }
}
```

A "switch" statement consists of one or more "case" statements, and an optional "default" statement. After a switch statement, execution goes to the matching "case" statement if there is one; otherwise execution goes to the "default" statement; otherwise execution continues past the end of the "select" statement.

After you write code following a "case" label, you must use a "break" statement to cause execution to go past the end of the "switch" statement. If you don't use a "break", execution "falls through" to the next "case" handler.

```
// Example of switch-case.
function TestSwitch2()
{
   switch( LightType )
   {
```

```
      case LT_None:
         log( "There is no lighting" );
         break;
      case LT_Steady:   // will "fall though" to the LT_Backdrop
case
      case LT_Backdrop:
         log( "There is lighting" );
         break;
      default:
         log( "Something else" );
         break;
   }
}
```

Goto

The "Goto" command goes to a label somewhere in the current function or state.

```
// Example of "goto".
function GotoExample()
{
   log( "Starting GotoExample" );
   goto Hither;
Yon:
   log( "At Yon" );
   goto Elsewhere;
Hither:
   log( "At Hither" );
   goto Yon;
Elsewhere:
   log( "At Elsewhere" );
}
```

The output is:

```
Starting GotoExample
At Hither
At Yon
At Elsewhere
```

Language Functionality

Built-in operators and their precedence

UnrealScript provides a wide variety of C/C++/Java-style operators for such operations as adding numbers together, comparing values, and incrementing variables. The complete set of operators is defined in Object.u, but here is a recap. Here are the standard operators, in order of precedence. Note that all of the C style operators have the same precedence as they do in C.

Operator	Types it applies to	Meaning
@	string	String concatenation, with an additional space between the two strings. "string1"@"string2" = "string1 string2"
@=	string	String concatenation, with an additional space between the two strings, concat and assign (v3323 and up)
$	string	String concatenation
$=	string	String concatenation, concat and assign (v3323 and up)
*=	byte, int, float, vector, rotator	Multiply and assign
/=	byte, int, float, vector, rotator	Divide and assign
+=	byte, int, float, vector	Add and assign
-=	byte, int, float, vector	Subtract and assign
||	bool	Logical or
&&	bool	Logical and
^^	bool	Exclusive or
&	int	Bitwise and
|	int	Bitwise or
^	int	Bitwise exlusive or (XOR)
!=	All	Compare for inequality
==	All	Compare for equality

<	byte, int, float, string	Less than
>	byte, int, float, string	Greater than
<=	byte, int, float, string	Less than or equal to
>=	byte, int, float, string	Greater than or equal to
~=	float, string	Approximate equality (within 0.0001), case-insensitive equality.
<<	int, vector	Left shift (int), Forward vector transformation (vector)
>>	int, vector	Right shift (int), Reverse vector transformation (vector)
>>>		same as >>
+	byte, int, float, vector	Add
-	byte, int, float, vector	Subtract
%	float, int, byte	Modulo (remainder after division)
*	byte, int, float, vector, rotator	Multiply
/	byte, int, float, vector, rotator	Divide
Dot	vector	Vector dot product
Cross	vector	Vector cross product
**	float	Exponentiation
ClockwiseFrom	int (rotator elements)	returns true when the 1st is clockwise from the 2nd argument

The above table lists the operators in order of precedence (with operators of the same precedence grouped together). When you type in a complex expression like "1*2+3*4", UnrealScript automatically groups the operators by precedence. Since multiplication has a higher precedence than addition, the expression is evaluated as "(1*2)+(3*4)".

The "&&" (logical and) and "||" (logical or) operators are short-circuited: if the result of the expression can be determined solely from the first expression (for

example, if the first argument of && is false), the second expression is not evaluated.

Regarding forward and reverse vector transformations, revers >> transforms from local space to world space and forward << transforms back again.

For example, if you have a vector facing 64 units ahead of a player, vect(64,0,0) is in local player-space. If you want it in world space, you have to transform it to world space using the player's rotation so you'd calculate it with the following:

```
myWorldSpaceVect = my64Vect >> playerPawn.rotation;
```

You'd want to use the forward rotation in cases when you have a world-space vector and you want it in local player space. As an example, you might want to convert a car actor's world-space velocity to local space so you can grab the X (forward velocity) to print it to a HUD.

In addition to the standard operators, UnrealScript supports the following unary operators:

- ! (bool) Logical not.
- - (int, float) negation.
- ~ (int) bitwise negation.
- ++, -- Decrement (either before or after a variable).

New operators are added to the engine from time to time. For a complete list of operators, check the latest UnrealScript source - specifically the Object class.

General purpose functions

Creating objects

In order to create a new object instance in UnrealScript, you'll use one of two functions depending on whether the object is an Actor or not. For Actors, you must use the Spawn function, which is declared in Actor.uc. For non Actor-derived classes, you must use the `new` operator. The syntax for the new operator is unlike that of any other function. In addition to an optional parameter list, you must specify the class of the new object and an optional template object. There is no UnrealScript declaration for the new operator, but here's what the function signature would look like:

```
native final operator function coerce Object new
(
   object   InOuter,
   name     InName,
   int      InFlags,
   class    InClass,
   object   InTemplate
);
```

InOuter
: (optional) the object to assign as the *Outer* for the newly created object. If not specified, the object's Outer will be set to a special package which exists only while the game is running, called the "transient package".

InName
: (optional) the name to give the new object. If not specified, the object will be given a unique name in the format ClassName_##, where ## is incremented each time an instance of this class is created.

InFlags
: (optional, currently broken since object flags are now 64 bits) the object flags to use when creating the object. The valid values are:

  - 0x0000000100000000: Supports editor undo/redo. (RF_Transactional)
  - 0x0000000400000000: Can be referenced by external files. (RF_Public)
  - 0x0000400000000000: Cannot be saved to disk. (RF_Transient)
  - 0x0010000000000000: Don't load object on the game client. (RF_NotForClient)
  - 0x0020000000000000: Don't load object on the game server. (RF_NotForServer)
  - 0x0040000000000000: Don't load object in the editor. (RF_NotForEdit)
  - 0x0008000000000000: Keep object around for editing even if unreferenced. (RF_Standalone)

InClass
: the class to create an instance of

InTemplate
: the object to use for initializing the new object's property values

The actual syntax for the new operator is as follows:

```
ObjectVar = new[(InOuter, InName, InFlags)]
<class'InClass'>[(InTemplate)];
```

Create an object of class LightFunction:

```
function CreateALight()
{
   local LightFunction NewObj;

   NewObj = new class'Engine.LightFunction';
}
```

Create a new LightFunction object named "NewLight", assigning this object as its Outer

```
function CreateALight()
{
   local LightFunction NewObj;

   NewObj = new(Self,'NewLight') class'Engine.LightFunction';
}
```

Create an new LightFunction object named "NewLight" in the transient package, using the object assigned as the value of the LightFunctionTemplate variable for initializing the new object's properties:

```
var LightFunction LightFunctionTemplate;

function CreateALight()
{
   local LightFunction NewObj;

   NewObj = new(None,'NewLight') class'Engine.LightFunction'
(LightFunctionTemplate);
}

defaultproperties
{
   Begin Object Class=LightFunction Name=MyLightFunctionArchetype
   End Object
   LightFunctionTemplate=MyLightFunctionArchetype
}
```

Integer functions

- int Rand(int Max); Returns a random number from 0 to Max-1.
- int Min(int A, int B); Returns the minimum of the two numbers.
- int Max(int A, int B); Returns the maximum of the two numbers.
- int Clamp(int V, int A, int B); Returns the first number clamped to the interval from A to B.

Warning - Unlike the C or C++ equivalents, Min and Max work on integers. There is no warning for using them on floats - your numbers will just be silently rounded down! For floats, you need to use FMin and FMax.

Floating point functions

- float Abs(float A); Returns the absolute value of the number.
- float Sin(float A); Returns the sine of the number expressed in radius.
- float Cos(float A); Returns the cosine of the number expressed in radians.
- float Tan(float A); Returns the tangent of the number expressed in radians.
- float ASin(float A); Returns the inverse sine of the number expressed in radius.
- float ACos(float A); Returns the inverse cosine of the number expressed in radius.
- float Atan(float A); Returns the inverse tangent of the number expressed in radians.
- float Exp(float A); Returns the constant "e" raised to the power of A.
- float Loge(float A); Returns the log (to the base "e") of A.
- float Sqrt(float A); Returns the square root of A.
- float Square(float A); Returns the square of A = A*A.
- float FRand(); Returns a random number from 0.0 to 1.0.
- float FMin(float A, float B); Returns the minimum of two numbers.
- float FMax(float A, float B); Returns the maximum of two numbers.
- float FClamp(float V, float A, float B); Returns the first number clamped to the
- interval from A to B.
- float Lerp(float A, float B, float Alpha); Returns the linear interpolation between A and B.
- float Smerp(float Alpha, float A, float B); Returns an Alpha-smooth nonlinear interpolation between A and B.
- float Ceil (float A); Rounds up

- float Round (float A); Rounds normally

String functions

- int Len(coerce string S); Returns the length of a string.
- int InStr(coerce string S, coerce string t); Returns the offset into the first string of the second string if it exists, or -1 if not.
- string Mid (coerce string S, int i, optional int j); Returns the middle part of the string S, starting and character i and including j characters (or all of them if j is not specified).
- string Left (coerce string S, int i); Returns the i leftmost characters of s.
- string Right (coerce string] S, int i); Returns the i rightmost characters of s.
- string Caps (coerce string S); Returns S converted to uppercase.
- string Locs (coerce string S); Returns the lowercase representation of S (v3323 and up)
- string Chr (int i); Returns a character from the ASCII table
- int Asc (string S); Returns the ASCII value of a character (only the first character from the string is used)
- string Repl (coerce string Src, coerce string Match, coerce string With, optional bool bCaseSensitive); Replace `Match` with `With` in the source. (v3323 and up)
- string Split(coerce string Text, coerce string SplitStr, optional bool bOmitSplitStr); Splits `Text` on the first occurrence of `SplitStr` and returns the remaining part of `Text`. If `bOmitSplitStr` is true, `SplitStr` will be omitted from the returned string.
- array SplitString(string Source, optional string Delimiter=",", optional bool bCullEmpty); Wrapper for splitting a string into an array of strings using a single expression.
- JoinArray(array StringArray, out string out_Result, optional string delim = ",", optional bool bIgnoreBlanks = true); Create a single string from an array of strings, using the delimiter specified, optionally ignoring blank members.
- ParseStringIntoArray(string BaseString, out array Pieces, string Delim, bool bCullEmpty); Breaks up a delimited string into elements of a string array.
- A == B; Comparison that returns true if both strings are the same (Case Sensitive).
- A ~= B; Comparison that returns true if both strings are the same (NOT Case Sensitive).

- A != B; Comparison that returns true if the strings are different (Case Sensitive).

See Unreal Strings for more information.

Vector functions

- vector vect(float X, float Y, float Z); Creates a new vector with the given components.
- float VSize(vector A); Returns the euclidean size of the vector (the square root of the sum of the components squared).
- vector Normal(vector A); Returns a vector of size 1.0, facing in the direction of the specified vector.
- Invert (out vector X, out vector Y, out vector Z); Inverts a coordinate system specified by three axis vectors.
- vector VRand ( ); Returns a uniformly distributed random vector.
- vector MirrorVectorByNormal(vector Vect, vector Normal); Mirrors a vector about a specified normal vector.

Timer functions

Timer functions are only available to Actor subclasses.

You can create multiple timers with each a different rate. Each timer has a unique target function (defaults to *Timer()*).

- function SetTimer(float inRate, optional bool inbLoop, optional Name inTimerFunc); Start a timer that is triggered after *inRate* seconds. If *inbLoop* is true the timer will loop. *inTimerFunc* defines the function to call, by default this is the function *Timer()*, this value is also used to identify to multiple timers.
- ClearTimer(optional Name inTimerFunc); stops a running timer.
- bool IsTimerActive(optional Name inTimerFunc); returns true if the given timer is active
- float GetTimerCount(optional Name inTimerFunc); returns the counter value of the timer, e.g. the number of seconds since the last time the timer was executed. Returns -1 if the timer is not active.
- float GetTimerRate(optional name TimerFuncName = 'Timer'); returns the rate of timer, `GetTimerRate('SomeTimer') - GetTimerCount('SomeTimer')` will return the remaining time for the timer.

Debugging functions

The following functions can aid you in debugging you code

- LogEx(ELoggingSeverity Severity, name Category, coerce string Msg); Log a message with a given severity and category. This function gives more control that the standard *log()* function. It allows you to filter log messages based on severity and category in runtime.
- LogFatal(name Category, coerce string Msg); shorthand for calling *LogEx(LOG_FATAL, Category, Msg)*
- LogError(name Category, coerce string Msg);
- function LogWarn(name Category, coerce string Msg);
- LogInfo(name Category, coerce string Msg);
- LogDebug(name Category, coerce string Msg);
- LogTrace(name Category, coerce string Msg);

Note that as of changelist 134102, the above logging functions are no longer available. They have been replaced by a logging macro, which is handled by the UnrealScript Preprocessor.

- ScriptTrace(); Dumps the current script call stack to the log file
- Name GetFuncName(); Returns the current calling function's name
- DumpStateStack(); Logs the current state stack

UnrealScript preprocessor

For more details, see the Unrealscript Preprocessor page.

UnrealScript tools and utilities

Script Profiler

The Script Profiler can help with understanding what areas of script execution are taking the most time.

Script Debugger

See the Unreal Debugging Tools page for more information.

Advanced Language Features

Timers

Timers are used as a mechanism for scheduling an event to occur, or reoccur, over time. In this manner, an Actor can set a timer to register itself with the game engine to have a `Timer()` function called either once, or recurring, after a set amount of time has passed.

UnrealScript timers are just implemented as an array of structs inside each Actor (an Actor can have multiple timers pending). The struct contains the amount of time remaining before the timer expires, the function to call on expiry, etc.

The game loop normally ticks each Actor once per frame, and part of each Ator's `Tick()` function includes a call to `UpdateTimers()` which will check for any expired timers and call their appropriate UnrealScript function.

The granularity is limited to the frame delta time, but there are no hardware or OS resources required. All of this is implemented in C++ so you could safely update hundreds of UnrealScript timers without any cause for concern. Of course you wouldn't want them all expiring simultaneously or every frame because they execute (slow) script code when they're activated.

States

Overview of States

Historically, game programmers have been using the concept of states ever since games evolved past the "pong" phase. States (and what is known as "state machine programming") are a natural way of making complex object behaviour manageable. However, before UnrealScript, states have not been supported at the language level, requiring developers to create C/C++ "switch" statements based on the object's state. Such code was difficult to write and update.

UnrealScript supports states at the language level.

In UnrealScript, each actor in the world is always in one and only one state. Its state reflects the action it wants to perform. For example, moving brushes have several states like "StandOpenTimed" and "BumpOpenTimed". Pawns have several states such as "Dying", "Attacking", and "Wandering".

In UnrealScript, you can write functions and code that exist in a particular state. These functions are only called when the actor is in that state. For example, say

you're writing a monster script, and you're contemplating how to handle the "SeePlayer" function. When you're wandering around, you want to attack the player you see. When you're already attacking the player, you want to continue on uninterrupted.

The easiest way to do this is by defining several states (Wandering and Attacking), and writing a different version of "Touch" in each state. UnrealScript supports this.

Before delving deeper into states, you need to understand that there are two major benefits to states, and one complication:

- Benefit: States provide a simple way to write state-specific functions, so that you can handle the same function in different ways, depending on what the actor is doing.

- Benefit: With a state, you can write special "state code", using the entire regular UnrealScript commands plus several special functions known as "latent functions". A latent function is a function that executes "slowly" (i.e. non-blocking), and may return after a certain amount of "game time" has passed. This enables you to perform time-based programming -- a major benefit which neither C, C++, nor Java offer. Namely, you can write code in the same way you conceptualize it; for example, you can write a script that says the equivalent of "open this door; pause 2 seconds; play this sound effect; open that door; release that monster and have it attack the player". You can do this with simple, linear code, and the Unreal engine takes care of the details of managing the time-based execution of the code.

- Complication: Now that you can have functions (like *Touch*) overridden in multiple states as well as in child classes, you have the burden of figuring out exactly which "Touch" function is going to be called in a specific situation. UnrealScript provides rules which clearly delineate this process, but it is something you must be aware of if you create complex hierarchies of classes and states.

Here is an example of states from the TriggerLight script:

```
// Trigger turns the light on.
state() TriggerTurnsOn
{
```

```
   function Trigger( actor Other, pawn EventInstigator )
   {
      Trigger = None;
      Direction = 1.0;
      Enable( 'Tick' );
   }
}

// Trigger turns the light off.
state() TriggerTurnsOff
{
   function Trigger( actor Other, pawn EventInstigator )
   {
      Trigger = None;
      Direction = -1.0;
      Enable( 'Tick' );
   }
}
```

Here you are declaring two different states (TriggerTurnsOn and TriggerTurnsOff), and you're writing a different version of the Trigger function in each state. Though you could pull off this implementation without states, using states makes the code far more modular and expandable: in UnrealScript, you can easily subclass an existing class, add new states, and add new functions. If you had tried to do this without states, the resulting code would be more difficult to expand later.

A state can be declared as editable, meaning that the user can set an actor's state in UnrealEd, or not. To declare an editable state, do the following:

```
state() MyState
{
   //...
}
```

To declare a non-editable state, do this:

```
state MyState
{
   //...
}
```

You can also specify the automatic, or initial state that an actor should be in by using the "auto" keyword. This causes all new actors to be placed in that state when they first are activated:

```
auto state MyState
{
   //...
}
```

State Labels and Latent Functions

In addition to functions, a state can contain one or more labels followed by UnrealScript code. For example:

```
auto state MyState
{
Begin:
   Log( "MyState has just begun!" );
   Sleep( 2.0 );
   Log( "MyState has finished sleeping" );
   goto('Begin');
}
```

The above state code prints the message "`MyState has just begun!`", then it pauses for two seconds, then it prints the message "`MyState has finished sleeping`". The interesting thing in this example is the call to the latent function "Sleep": this function call doesn't return immediately, but returns after a certain amount of game time elapses. Latent functions can only be called from within state code, and not from within functions. Latent functions let you manage complex chains of events that include the passage of time.

All state code begins with a label definition; in the above example the label is named "Begin". The label provides a convenient entry point into the state code. You can use any label name in state code, but the "Begin" label is special: it is the default starting point for code in that state.

There are three main latent functions available to all actors:

- Sleep(float Seconds) pauses the state execution for a certain amount of time, and then continues.

- FinishAnim() waits until the current animation sequence you're playing completes, and then continues. This function makes it easy to write animation-driven scripts, scripts whose execution is governed by mesh animations. For example, most of the AI scripts are animation-driven (as opposed to time-driven), because smooth animation is a key goal of the AI system.
- FinishInterpolation() waits for the current InterpolationPoint movement to complete, and then continues.

The Pawn class defines several important latent functions for actions such as navigating through the world and short-term movement. See the separate AI docs for descriptions of their usage.

Three native UnrealScript functions are particularly useful when writing state code:

- The "Goto('LabelName')" function (similar to the C/C++/Basic goto) within a state causes the state code to continue executing at the specified label.
- The special Goto('') command within a state causes the state code execution to stop. State code execution doesn't continue until you go to a new state, or go to a new label within the current state.
- The "GotoState" function causes the actor to go to a new state, and optionally continue at a specified label (if you don't specify a label, the default is the "Begin" label). You can call GotoState from within state code, and it goes to the destination immediately. You can also call GotoState from within any function in the actor, but that does not take effect immediately: it doesn't take effect until execution returns back to the state code.

Here is an example of the state concepts discussed so far:

```
// This is the automatic state to execute.
auto state Idle
{
   // When touched by another actor...
   function Touch( actor Other )
   {
      log( "I was touched, so I'm going to Attacking" );
      GotoState( 'Attacking' );
      Log( "I have gone to the Attacking state" );
   }
```

```
Begin:
   log( "I am idle..." );
   sleep( 10 );
   goto 'Begin';
}

// Attacking state.
state Attacking
{
Begin:
   Log( "I am executing the attacking state code" );
   //...
}
```

When you run this program and then go touch the actor, you will see:

```
I am idle...
I am idle...
I am idle...
I was touched, so I'm going to Attacking
I have gone to the Attacking state
I am executing the attacking state code
```

Make sure you understand this important aspect of GotoState: When you call GotoState from within a function, it does not go to the destination immediately, rather it goes there once execution returns back to the state code.

State inheritance and scoping rules

In UnrealScript, when you subclass an existing class, your new class inherits all of the variables, functions and states from its parent class. This is well-understood.

However, the addition of the state abstraction to the UnrealScript programming model adds additional twists to the inheritance and scoping rules. The complete inheritance rules are:

- A new class inherits all of the variables from its parent class.
- A new class inherits all of its parent class's non-state functions. You can override any of those inherited non-state functions. You can add entirely new non-state functions.

- A new class inherits all of its parent class's states, including the functions and labels within those states. You can override any of the inherited state functions, and you can override any of the inherited state labels, you can add new state functions, and you can add new state labels.

Here is an example of all the overriding rules:

```
// Here is an example parent class.
class MyParentClass extends Actor;

// A non-state function.
function MyInstanceFunction()
{
   log( "Executing MyInstanceFunction" );
}

// A state.
state MyState
{
   // A state function.
   function MyStateFunction()
   {
      Log( "Executing MyStateFunction" );
   }
// The "Begin" label.
Begin:
   Log("Beginning MyState");
}

// Here is an example child class.
class MyChildClass extends MyParentClass;

// Here I'm overriding a non-state function.
function MyInstanceFunction()
{
   Log( "Executing MyInstanceFunction in child class" );
}

// Here I'm redeclaring MyState so that I can override
MyStateFunction.
```

```
state MyState
{
   // Here I'm overriding MyStateFunction.
   function MyStateFunction()
   {
      Log( "Executing MyStateFunction" );
   }
// Here I'm overriding the "Begin" label.
Begin:
   Log( "Beginning MyState in MyChildClass" );
}
```

When you have a function that is implemented globally, in one or more states, and in one or more parent classes, you need to understand which version of the function will be called in a given context. The scoping rules that resolve these complex situations are:

- If the object is in a state, and an implementation of the function exists somewhere in that state (either in the actor's class or in some parent class), the most-derived state version of the function is called.
- Otherwise, the most-derived non-state version of the function is called.

Advanced state programming

If a state doesn't override a state of the same name in the parent class, then you can optionally use the "extends" keyword to make the state expand on an existing state in the current class. This is useful, for example, in a situation where you have a group of similar states (such as MeleeAttacking and RangeAttacking) that have a lot of functionality in common. In this case you could declare a base Attacking state as follows:

```
// Base Attacking state.
state Attacking
{
   // Stick base functions here...
}

// Attacking up-close.
state MeleeAttacking extends Attacking
{
```

```
   // Stick specialized functions here...
}

// Attacking from a distance.
state RangeAttacking extends Attacking
{
   // Stick specialized functions here...
}
```

A state can optionally use the `ignores` specifier to ignore functions while in a state. The syntax for this is:

```
// Declare a state.
state Retreating
{
   // Ignore the following messages...
   ignores Touch, UnTouch, MyFunction;

   // Stick functions here...
}
```

You can tell what specific state an actor is in from its "state" variable, a variable of type "name".

It is possible for an actor to be in "no state" by using *GotoState('')*. When an actor is in "no state", only its global (non-state) functions are called.

Whenever you use the *GotoState* command to set an actor's state, the engine can call two special notification functions, if you have defined them: *EndState()* and *BeginState()*. *EndState* is called in the current state immediately before the new state is begun, and *BeginState* is called immediately after the new state begins. These functions provide a convenient place to do any state-specific initialization and cleanup that your state may require.

State Stacking

With normal state changing you go from one state to the other without being able to return to the previous state as it was left. With state stacking this is possible. Calling the function *PushState* will change to a new state putting it on top of the stack. The current state will be frozen. When *PopState* is called the previous state will be restored and continue it's execution from the point where

PushState was called. *PushState* will act as a latent function when possible (only inside of state code), so code execution behavior is different if you call *PushState* from within a function. Calling it from a function will not interrupt code execution (much like GotoState from within a function), whereas calling it from within state code will pause execution until the child state is popped (again, similar to GotoState from within state code).

A state can be put on the stack only once, trying to push the same state on the stack a second time will fail. *PushState* works just like *GotoState*, it takes the state name and an optional label for the state's entry point. The new state will receive an *PushedState* event, the current state receives a *PausedState* event. After calling *PopState* the current state receives a *PoppedState* event and the new state (the one that was next on the stack) will receive *ContinuedState*.

```
state FirstState
{
   function Myfunction()
   {
      doSomething();
      PushState('SecondState');
      // this will be executed immediately since we're inside of a
function (no latent functionality)
      JustPushedSecondState();
   }

Begin:
   doSomething();
   PushState('SecondState');
   // this will be executed once SecondState is popped since we're
inside of a state code block (latent functionality)
   JustPoppedSecondState();
}

state SecondState
{
   event PushState()
   {
      // we got pushed, push back
      PopState();
   }
```

```
}
```

Using the function *IsInState* you will be able to check if a certain state is on the stack. This function only checks the name of the states and therefor can not be used to check on parent states. For example:

```
state BaseState
{
   ...
}

state ExtendedState extends BaseState
{
   ...
}
```

If the active state is `ExtendedState` then *IsInState('BaseState')* will return false. Ofcourse calling *IsInState('BaseState', true)* will return true if *BaseState* is on the stack.

Replication

For more details on replication in UnrealScript, see the Networking Overview page.

Iteration (ForEach)

UnrealScript's `foreach` command makes it easy to deal with large groups of actors, for example all of the actors in a level, or all of the actors within a certain distance of another actor. "foreach" works in conjunction with a special kind of function called an "iterator" function whose purpose is to iterate through a list of actors.

Here is a simple example of `foreach`:

```
// Display a list of all lights in the level.
function Something()
{
   local actor A;

   // Go through all actors in the level.
```

```
   log( "Lights:" );
   foreach AllActors( class 'Actor', A )
   {
      if( A.LightType != LT_None )
         log( A );
   }
}
```

The first parameter in all `foreach` commands is a constant class, which specifies what kinds of actors to search. You can use this to limit the search to, for example, all Pawns only.

The second parameter in the `foreach` command is a variable that is assigned an actor for the duration of each iteration through the `foreach` loop.

Here are all of the iterator functions that work with "foreach".

- `AllActors ( class<actor> BaseClass, out actor Actor, optional name MatchTag )`
  Iterates through all actors in the level. If you specify an optional MatchTag, only includes actors that have a "Tag" variable matching the tag you specified.

- `DynamicActors( class<actor> BaseClass, out actor Actor )`
  Iterates through all the actors that have been spawned since the level started, ignoring the ones placed in the level.

- `ChildActors( class<actor> BaseClass, out actor Actor )`
  Iterates through all actors owned by this actor.

- `BasedActors( class<actor> BaseClass, out actor Actor )`
  Iterates through all actors which use this actor as a base.

- `TouchingActors( class<actor> BaseClass, out actor Actor )`
  Iterates through all actors which are touching (interpenetrating) this actor.

- `TraceActors( class<actor> BaseClass, out actor Actor, out vector HitLoc, out vector HitNorm, vector End, optional vector Start, optional vector Extent )`
  Iterates through all actors which touch a line traced from the Start point to the End point, using a box of collision extent Extent. On each iteration,

HitLoc is set to the hit location, and HitNorm is set to an outward-pointing hit normal.

- `OverlappingActors( class<actor> BaseClass, out actor Actor, float Radius, optional vector Loc, optional bool bIgnoreHidden )`
  Iterates through all actors within a specified radius of the specified location (or if none is specified, this actor's location).

- `VisibleActors( class<actor> BaseClass, out actor Actor, optional float Radius, optional vector Loc )`
  Iterates through a list of all actors who are visible to the specified location (or if no location is specified, this actor's location).

- `VisibleCollidingActors ( class<actor> BaseClass, out actor Actor, float Radius, optional vector Loc, optional bool bIgnoreHidden );`
  returns all colliding (bCollideActors==true) actors within a certain radius for which a trace from Loc (which defaults to caller's Location) to that actor's Location does not hit the world. Much faster than *AllActors()* since it uses the collision hash.

- `CollidingActors ( class<actor> BaseClass, out actor Actor, float Radius, optional vector Loc );`
  returns colliding (bCollideActors==true) actors within a certain radius. Much faster than *AllActors()* for reasonably small radii since it uses the collision hash

Note: The iterator functions are all members of particular class so if you want to use an iterator from within a function in a non-Actor, you must have an actor variable and use the following syntax:

- `foreach ActorVar.DynamicActors(class'Pawn', P)`

So, from with an Interaction class, you could do:

- `foreach ViewportOwner.Actor.DynamicActors(class'Pawn', P)`

Note: Iterators now support dynamic arrays as well, which you can find in the Advanced Language Features section.

Function Calling Specifiers

In complex programming situations, you will often need to call a specific version of a function, rather than the one that's in the current scope. To deal with these cases, UnrealScript provides the following keywords:

Global
Calls the most-derived global (non-state) version of the function.

Super
Calls the corresponding version of the function in the parent class. The function called may either be a state or non-state function depending on context.

Super(classname)
Calls the corresponding version of the function residing in (or above) the specified class. The function called may either be a state or non-state function depending on context.

It is not valid to combine multiple calling specifiers (i.e. *Super(Actor).Global.Touch*).

Here are some examples of calling specifiers:

```
class MyClass extends Pawn;

function MyExample( actor Other )
{
   Super(Pawn).Touch( Other );
   Global.Touch( Other );
   Super.Touch( Other );
}
```

As an additional example, the BeginPlay() function is called when an actor is about to enter into gameplay. The BeginPlay() function is implemented in the Actor class and it contains some important functionality that needs to be executed. Now, say you want to override BeginPlay() in your new class MyClass, to add some new functionality. To do that safely, you need to call the version of BeginPlay() in the parent class:

```
class MyClass extends Pawn;

function BeginPlay()
```

```
{
   // Call the version of BeginPlay in the parent class (important).
   Super.BeginPlay();

   // Now do custom BeginPlay stuff.
   //...
}
```

Accessing static functions in a variable class

Static functions in a variable class may be called using the following syntax.

```
var class C;
var class<Pawn> PC;

class'SkaarjTrooper'.static.SomeFunction(); // Call a static
function
                                                // in a specific class.

PC.static.SomeFunction(); // Call a static function in a variable
class.

class<Pawn>(C).static.SomeFunction(); // Call a static function in a
                                      //casted class expression.
```

Default values of variables

Accessing default values of variables

UnrealEd enables level designers to edit the "default" variables of an object's class. When a new actor is spawned of the class, all of its variables are initialized to those defaults. Sometimes, it's useful to manually reset a variable to its default value. For example, when the player drops an inventory item, the inventory code needs to reset some of the actor's values to its defaults. In UnrealScript, you can access the default variables of a class with the "Default." keyword. For example:

```
var() float Health, Stamina;
//...

// Reset some variables to their defaults.
function ResetToDefaults()
{
   // Reset health, and stamina.
   Health = Default.Health;
   Stamina = Default.Stamina;
}
```

Accessing default values of variables through a class reference

If you have a class reference (a variable of `class` or `class<classlimitor>` type), you can access the default properties of the class it references, without having an object of that class. This syntax works with any expression that evaluates to class type.

```
var class C;
var class<Pawn> PC;

Health = class'Spotlight'.default.LightBrightness; // Access the
default value of
                                                   //
LightBrightness in the Spotlight class.

Health = PC.default.Health; // Access the default value of Health in
                           // a variable class identified by PC.

Health = class<Pawn>(C).default.Health; // Access the default value
                                        // of Health in a casted
class
                                        // expression.
```

Specifying default values using the defaultproperties block

In addition to setting the default value for an Actor's properties using a property window in UnrealEd, you can also assign default values for member variables by placing special assignment expressions inside the class's defaultproperties block.

- Statements are not allowed in the defaultproperties block, which the exception of dynamic array operations
- Semi-colons can be placed at the end of each line, but are not required
- Default values are inherited by child classes. Values specified in child classes' defaultproperties override values specified in parent classes.

Syntax

The syntax of the defaultproperties block is slightly different from the standard unrealscript syntax:

- Simple Types (Ints, Floats, Bools, Bytes):
  - VarName=Value

- Static Array:
  - `ArrayProp(0)=Value1`
    `ArrayProp(1)=Value2`
    OR
  - `ArrayProp[0]=Value1`
    `ArrayProp[1]=Value2`

- Dynamic Arrays:
  - `ArrayProp=(Value1,Value2,Value3)`
    OR
  - `ArrayProp(0)=Value1`
    `ArrayProp(1)=Value2`
    `ArrayProp(2)=Value3`
    OR
  - `ArrayProp.Add(Value1)`
    `ArrayProp.Add(Value2)`
    `ArrayProp.Add(Value3)`

- Names

    - `NameProp='Value'`
      OR
    - `NameProp=Value`

- Objects
    - `ObjectProp=ObjectClass'ObjectName'`

- Subobjects
    - ```
      Begin Object Class=ObjectClass Name=ObjectName
          VarName=Value
          ...
      End Object
      ObjectProperty=ObjectName
      ```

- Structs (including Vectors):
    - `StructProperty=(InnerStructPropertyA=Value1,InnerStructPropertyB=Value2)`
      OR
    - ```
      StructProperty={(
                      InnerStructPropertyA=Value1,
                      InnerStructPropertyB=Value2
                      )}
      ```

NOTE: Some types require different syntax when used inside a struct default value.

- Inline static arrays must be declared like so (Notice that brackets "[]" are used here for the array delimiter instead of parenthesis "()"):
  `StructProperty=(StaticArray[0]=Value,StaticArrayProp[1]=Value)`
- Inline dynamic arrays must be declared using the single line syntax:
  `StructProperty=(DynamicArray=(Value,Value))`
- Inline name variables must be wrapped with quotes:
  `StructProperty=(NameProperty="Value")`

- Dynamic Array Operations. These can be used to modify the contents of a dynamic array, which can be inherited by a parent.
    - `Array.Empty` - clears the whole array
    - `Array.Add(element)` - adds element to the end of the array
    - `Array.Remove(element)` - removes the element from the array, this will remove all occurences of the element

- `Array.RemoveIndex(index)` - removes the element at the given index
- `Array.Replace(elm1, elm2)` - replaces elm1 with elm2. All occurences will be replaced. A warning is produced with elm1 is not found.

Consider the following example (based on Actor.uc):

```
defaultproperties
{
   // objects
   MessageClass=class'LocalMessage'

    // declare an inline subobject of class SpriteComponent named
"Sprite"
   Begin Object Class=SpriteComponent Name=Sprite
       // values specified here override SpriteComponent's own
defaultproperties
      Sprite=Texture2D'EngineResources.S_Actor'
      HiddenGame=true
   End Object
   //todo
   Components.Add(Sprite)

    // declare an inline subobject of class CylinderComponent named
"CollisionCylinder"
   Begin Object Class=CylinderComponent Name=CollisionCylinder
       // values specified here override CylinderComponent's own
defaultproperties
      CollisionRadius=10
      CollisionHeight=10
      AlwaysLoadOnClient=True
      AlwaysLoadOnServer=True
   End Object
   //todo
   Components.Add(CollisionCylinder)

   CollisionComponent=CollisionCylinder

    // floats (leading '+' and trailing 'f' characters are ignored)
```

```
   DrawScale=00001.000000
   Mass=+00100.000000
   NetPriority=00001.f

    // ints
   NetUpdateFrequency=100

    // enumerations
   Role=ROLE_Authority
   RemoteRole=ROLE_None

    // structs
   DrawScale3D=(X=1,Y=1,Z=1)

    // bools
   bJustTeleported=true
   bMovable=true
   bHiddenEdGroup=false
   bReplicateMovement=true

   // names
   InitialState=None

   // dynamic array (in this case, a dynamic class array)
   SupportedEvents(0)=class'SeqEvent_Touch'
   SupportedEvents(1)=class'SeqEvent_UnTouch'
   SupportedEvents(2)=class'SeqEvent_Destroyed'
   SupportedEvents(3)=class'SeqEvent_TakeDamage'
}
```

Struct Defaults

When you declare a struct in UnrealScript, you can optionally specify default values for the struct's properties. Anytime the struct is used in UnrealScript, it's members are initialized with these values. The syntax is identical to the defaultproperties block for a class - the only exception is that you must name the block **structdefaultproperties**. For example:

```
struct LinearColor
{
   var() config float R, G, B, A;

   structdefaultproperties
   {
      A=1.f
   }
};
```

Anytime you declare a LinearColor variable in UnrealScript, the value of its *A* property will be set to 1.f. Something else that is important to understand when using structdefaultproperties is that class defaults override struct defaults. If you have a class member variable of type LinearColor, any value you assign to that member variable in the class defaultproperties will override the value in the struct's defaults.

```
var LinearColor NormalColor, DarkColor;

defaultproperties
{
    NormalColor=(R=1.f,B=1.f,G=1.f)       // value of A will be 1.0f
for this property
    DarkColor=(R=1.f,B=1.f,G=1.f,A=0.2f)  // value of A will be 0.2f
for this property
}
```

Dynamic Arrays

Previously, we covered Arrays, which were static. What that means is that the size (how many elements are in the array) is set at compile time and cannot be changed. Dynamic Arrays and Static Arrays share the following common characteristics :

- constant seek time - the time code spends accessing any given element of the array is the same, regardless of how many elements are in the array
- unlimited element type - you can have an array of anything - ints, vectors, Actors, etc. (with the exception that booleans are only valid for dynamic arrays)

- access behavior - you can access any element with an index into the array, and conversely, attempting to access an element at an index that is outside the bounds of the array will throw an accessed none.

Dynamic Arrays provide a way of having Static Array functionality with the ability to change the number of elements during run-time, in order to accommodate changing needs. In order use Dynamic Arrays, we need to know a few things.

The first is variable declaration. Declaring a dynamic array is much like declaring any other unrealscript variable (i.e. var/local *type varname*). For dynamic arrays, the type is specified with the *array* keyword, followed by the array type wrapped in angle brackets. If the array type contains angle brackets as well (such as *class*<*Actor*>), you must place a space between the closing bracket of the type and the closing bracket of the array wrapper or the compiler resolves the double closing brackets as the >> operator. For example:
Declare a dynamic array of ints named IntList: **`var array<int> IntList;`**
Declare a dynamic array of type class<PlayerController> named Players: **`var array<class<PlayerController> > Players;`**

When script starts, IntList will start with 0 elements. There are methods supported by Dynamic Arrays that allow us to add elements to the array, take elements out, and increase or decrease the length of the array arbitrarily. The syntax for calling these methods is (using our IntList example): `IntList.MethodName()`. The following dynamic array methods are available:

- Add(**int** *Count*): extends the length of the array by *Count*, identical to FArray::AddZeroed().
- Insert(**int** *Index*, **int** *Count*): where *Index* is the array index to being inserting elements, and *Count* is the number of elements to insert. Any existing elements at that location in the array are shifted up, and new elements are created and inserted into the specified location. Inserting 5 elements at index 3 will shift up (in index value) all elements in the array starting at index 3 and up by 5. The element previously located at index 3 will now be located at index 8, element 4 will now be element 9, and so on. Newly added elements are all initialized to their default values (zero/null for all types except structs containing structdefaultproperties).
- Remove(**int** *Index*, **int** *Count*): where *Index* is the array index to begin removing elements from, and *Count* is the number of elements to remove. This allows us to remove a group of elements from the array starting at any valid index within the array. Note that any indexes that are higher

than the range to be removed will have their index values changed, keep this in mind if you store index values into dynamic arrays.

- AddItem(*Item*): adds *Item* to the end of the array, extending the array length by one.
- RemoveItem(*Item*): removes any instances of *Item* using a linear search.
- InsertItem(**int** *Index*, *Item*): inserts *Item* into the array at *Index*, extending the array length by one.
- Find(...) - finds the index of an element in the array. There are two versions of Find: standard find for matching entire element values, and a specialized version for matching a struct based on the value of a single property of the struct
    - Find(*Value*): where *Value* is the value to search for. Returns the index for the first element found in the array which matches the value specified, or -1 if that value wasn't found in the array. *Value* can be represented using any valid expression.
    - Find(*PropertyName*, *Value*): where *PropertyName* is the name of property in the struct to search against (must be of type 'Name'), and *Value* is the value to search for. Returns the index for the first struct in the array that has a value matching the value specified for a property named *PropertyName*, or -1 if the value wasn't found. *Value* can be any valid expression.
- Sort(*SortDelegate*) - uses *SortDelegate* to sort the contents of the array in-place. *SortDelegate* should have signature matching the following:
    - delegate int ExampleSort(ArrayType A, ArrayType B) { return A < B ? -1 : 0; } // a negative return value indicates the items should be swapped

Length Variable

Dynamic Arrays also have a variable called `Length`, which is the current length (number of elements) of the dynamic array. To access Length, using our example array, we would say: `IntList.Length` . We can not only read the Length variable, but we can also directly set it, allowing us to modify the number of elements in the array. When you modify the Length variable directly, all changes in array length happen at the 'end' of the array. For example, if we set IntList.Length = 5, and then we set IntList.Length = 10, the extra 5 elements we just added were added to the end of the array, maintaining our original 5 elements and their values. If we decreased the Length, the elements would be taken off the end as well. Note that when you add elements to the array, either by Insert() or by increasing Length, the elements are initialized to the variable

type's default value (0 for ints, None for class references, etc). It is also noteworthy to know that you can increase the length of a dynamic array by setting an element index that is greater than the array's current Length value. This will extend the array just as if you had set Length to the larger value.

```
OldLength = Array.length
Array.Length = OldLength + 1
Array[OldLength] = NewValue

Array[Array.Length] = NewValue

Array.AddItem(NewValue)
```

are all equivalent forms of the same operation.

Note however that you cannot both increase the length of an array and access its members at the same time.

```
Array[Array.length].myStructVariable = newVal
```

does not work.

A word of caution - the Length member of a dynamic array should never be incremented / decremented by '++', '--', '+=', or '-=', nor should you pass Length to a function as an out parameter (where the function can change the value of it). Doing these things will result in memory leaks and crashes due to Length not being accurate any more; only setting the Length via the '=' operator (and setting an element at an index larger than Length) modifies the actual length of the dynamic array properly.

NOTE: `array<bool>` **is not a supported type!**

A final note - dynamic arrays are **not** replicated. You could get around this by having a function that replicates and has two arguments, an index into the dynamic array and the value to store there. However, you would also have to consider consequences of elements not being the same within a space of a tick on client and server.

Iterating Dynamic Arrays

Dynamic arrays now support the 'foreach' operator to allow simple iterations. The basic syntax is 'foreach ArrayVariable(out ArrayItem,optional out ItemIndex)

{}', where each iteration will increment the index and write out the item as well as the index if a property is supplied.

```
function IterateThroughArray(array<string> SomeArray)
{
    local string ArrayItem;
    local int Index;
    foreach SomeArray(ArrayItem)
    {
       `log("Array iterator test #1:"@ArrayItem);
    }
    foreach SomeArray(ArrayItem,Index)
    {
        `log("Array iterator test #2:"@ArrayItem@Index);
    }
}
```

Interface Classes

For more information on Interfaces, see the UnrealScript Interfaces page.

Function Delegates

For more information on how to use Delegates, see the UnrealScript Delegates page.

Native Classes

For more information on native classes, see the Compiling Native Classes and Creating Native Classes pages.

MetaData Support

In-game and in-editor functionality can be extended via property meta-data.

Metadata Overview

Arbitrary metadata can be linked to a property in UnrealScript as follows:

For a variable:

```
var float MyVar<TAG=VALUE>
```

For an enum:

```
enum EMyEnum
{
   EME_ValA<TAG=VALUE>,
   EME_ValB<TAG=VALUE>,
};
```

For a class:

See the following UnProg3 mailing list thread:
https://udn.epicgames.com/lists/showpost.php?id=24834&list=unprog3

Using Multiple MetaData Specifications

You can use multiple metadata specifications for the same property by separating them by a **|** character.

For example:

```
var()   LinearColor    DrawColor<DisplayName=Draw
Color|EditCondition=bOverrideDrawColor>;
```

Available MetaData Specifications

Here are the currently supported metadata tags and what they do:

<ToolTip=TEXT_STRING>

Makes `TEXT_STRING` appear as the tooltip when the mouse is placed over the corresponding property in an editor property window.

NOTE: Support was recently added so that `/** VALUE */` comments are automatically translated into ToolTip metadata by the script compiler.

<DisplayName=TEXT_STRING>

Makes a property's name appear as `TEXT_STRING` in the editor property window instead of its actual name.

Example:

```
Var() bool bEnableSpawning<DisplayName=Spawning Enabled>;
```

WARNING: *The use of DisplayName in enums will cause problems if you modify UPropertyInputCombo to sort enums in editor combo boxes.*

See the following UnProg3 mailing list thread:
https://udn.epicgames.com/lists/showpost.php?list=unprog3&id=24302

<EditCondition=ConditionalPropertyName>

This allows you to make the editability status of an editor property be enabled or disabled based on the value of another (Boolean) property.

For example, you could have the following setup in the MyPackage.MyClass UnrealScript class:

```
/** Enable or disable spawning */
Var() bool bEnableSpawning;

/** Set the rate at which AIs are spawned. Has no effect unless
bEnableSpawning = TRUE */
Var() float RespawnsPerSecond<EditCondition=bEnableSpawning>;
```

And then RespawnsPerSecond would be greyed-out in the editor whenever bEnableSpawning is false. This helps make things less confusing for designers.

Important: This metadata setting requires that the controlled variable (RespawnsPerSecond) utilize a custom property item binding (WxCustomPropertyItem_ConditionalItem).

In order to enable this, you need to hook it up in Editor.ini, as follows:

```
[UnrealEd.CustomPropertyItemBindings]
CustomPropertyClasses=(PropertyPathName=" MyPackage.MyClass:
RespawnsPerSecond
",PropertyItemClassName="WxCustomPropertyItem_ConditionalItem")
```

For more info on this, see
https://udn.epicgames.com/lists/showpost.php?list=unprog3&id=30824

<FriendlyName=TEXT_STRING>

NOT SURE IF THIS IS ONLY USED INTERNALLY AND NOT INTENDED FOR USE IN UNREALSCRIPT. PERHAPS EPIC CAN CLARIFY?

<AllowAbstract>

If present on a Class property, editor drop-down boxes for editing that property will include abstract classes. If not present, they will only contain concrete classes. There is no need to specify a value such as True or False in this metadata specification.

<AutoComment=BOOLEAN_VALUE>

When added to a property of a Kismet Sequence Action, the property and its current value will automatically appear as a comment above that action. To see this in action place a new "Gate" sequence action in a script. In this class both bOpen and AutoCloseCount use this metadata option.

Advanced Technical Issues

UnrealScript Implementation

For more detailed information on how UnrealScript works under the covers - from the Compile Process to Execution to Byte Code representaiton - see the UnrealScript Implementation page.

UnrealScript binary compatibility issues

UnrealScript is designed so that classes in package files may evolve over time without breaking binary compatibility. Here, binary compatibility means "dependent binary files may be loaded and linked without error"; whether your modified code functions as designed is a separate issue. Specifically, the kinds of modifications when may be made safely are as follows:

- The .uc script files in a package may be recompiled without breaking binary compatibility.
- Adding new classes to a package.
- Adding new functions to a class.
- Adding new states to a class.

- Adding new variables to a class.
- Removing private variables from a class.

Other transformations are generally unsafe, including (but not limited to):

- Adding new members to a struct.
- Removing a class from a package.
- Changing the type of any variable, function parameter, or return value.
- Changing the number of parameters in a function.

Technical notes

Garbage collection. All objects and actors in Unreal are garbage-collected using a tree-following garbage collector similar to that of the Java VM. The Unreal garbage collector uses the UObject class's serialization functionality to recursively determine which other objects are referenced by each active object. As a result, object need not be explicitly deleted, because the garbage collector will eventually hunt them down when they become unreferenced. This approach has the side-effect of latent deletion of unreferenced objects; however it is far more efficient than reference counting in the case of infrequent deletion. See the Garbage Collection page for more details.

UnrealScript is bytecode based. UnrealScript code is compiled into a series of bytecodes similar to p-code or the Java bytecodes. This makes UnrealScript platform-neutral; this porting the client and server components of Unreal to other platforms, i.e. the Macintosh or Unix, is straightforward, and all versions can interoperate easily by executing the same scripts.

Unreal as a Virtual Machine. The Unreal engine can be regarded as a virtual machine for 3D gaming in the same way that the Java language and the built-in Java class hierarchy define a virtual machine for Web page scripting. The Unreal virtual machine is inherently portable (due to splitting out all platform-dependent code in separate modules) and expandable (due to the expandable class hierarchy). However, at this time, there are no plans to document the Unreal VM to the extent necessary for others to create independent but compatible implementations.

The UnrealScript compiler is three-pass. Unlike C++, UnrealScript is compiled in three distinct passes. In the first pass, variable, struct, enum, const, state and function declarations are parsed and remembered; the skeleton of each class is built. In the second pass, the script code is compiled to byte codes.

This enables complex script hierarchies with circular dependencies to be completely compiled and linked in two passes, without a separate link phase. The third phase parses and imports default properties for the class using the values specified in the `defaultproperties` block in the .uc file.

Persistent actor state. It is important to note that in Unreal, because the user can save the game at any time, the state of all actors, including their script execution state, can be saved only at times when all actors are at their lowest possible UnrealScript stack level. This persistence requirement is the reason behind the limitation that latent functions may only be called from state code: state code executes at the lowest possible stack level, and thus can be serialized easily. Function code may exist at any stack level, and could have (for example) C++ native functions below it on the stack, which is clearly not a situation which one could save on disk and later restore.

Unrealfiles are Unreal's native binary file format. Unrealfiles contain an index, serialized dump of the objects in a particular Unreal package. Unrealfiles are similar to DLL's, in that they can contain references to other objects stored in other Unrealfiles. This approach makes it possible to distribute Unreal content in predefined "packages" on the Internet, in order to reduce download time (by never downloading a particular package more than once).

Why UnrealScript does not support static variables. While C++ supports static (per class-process) variables for good reasons true to the language's low-level roots, and Java support static variables for reasons that appear to be not well thought out, such variables do not have a place in UnrealScript because of ambiguities over their scope with respect to serialization, derivation, and multiple levels: should static variables have "global" semantics, meaning that all static variables in all active Unreal levels have the same value? Should they be per package? Should they be per level? If so, how are they serialized -- with the class in its .u file, or with the level in its .unr file? Are they unique per base class, or do derived versions of classes have their own values of static variables? In UnrealScript, we sidestep the problem by not defining static variables as a language feature, and leaving it up to programmers to manage static-like and global-like variables by creating classes to contain them and exposing them in actual objects. If you want to have variables that are accessible per-level, you can create a new class to contain those variables and assure they are serialized with the level. This way, there is no ambiguity. For examples of classes that serve this kind of purpose, see LevelInfo and GameInfo.

UnrealScript programming strategy

Here I want to cover a few topics on how to write UnrealScript code effectively, and take advantage of UnrealScript's strengths while avoiding the pitfalls.

UnrealScript is a slow language compared to C/C++. A typical C++ program runs about 20X faster than UnrealScript. The programming philosophy behind all of our own script writing is this: Write scripts that are almost always idle. In other words, use UnrealScript only to handle the "interesting" events that you want to customize, not the rote tasks, like basic movement, which Unreal's physics code can handle for you. For example, when writing a projectile script, you typically write a HitWall(), Bounce(), and Touch() function describing what to do when key events happen. Thus 95% of the time, your projectile script isn't executing any code, and is just waiting for the physics code to notify it of an event. This is inherently very efficient. In our typical level, even though UnrealScript is comparably much slower than C++, UnrealScript execution time averages 5-10% of CPU time.

Exploit latent functions (like FinishAnim and Sleep) as much as possible. By basing the flow of your script execution on them, you are creating animation-driven or time-driven code, which is fairly efficient in UnrealScript.

Keep an eye on the Unreal log while you're testing your scripts. The UnrealScript runtime often generates useful warnings in the log that notify you of nonfatal problems that are occuring.

Be wary of code that can cause infinite recursion. For example, the "Move" command moves the actor and calls your Bump() function if you hit something. Therefore, if you use a Move command within a Bump function, you run the risk of recursing forever. Be careful. Infinite recursion and infinite looping are the two error conditions which UnrealScript doesn't handle gracefully.

Spawning and destroying actors are fairly expensive operations on the server side, and are even more expensive in network games, because spawns and destroys take up network bandwidth. Use them reasonably, and regard actors as "heavy weight" objects. For example, do not try to create a particle system by spawning 100 unique actors and sending them off on different trajectories using the physics code. That will be sloooow.

Exploit UnrealScript's object-oriented capabilities as much as possible. Creating new functionality by overriding existing functions and states leads to

clean code that is easy to modify and easy to integrate with other peoples' work. Avoid using traditional C techniques, like doing a switch() statement based on the class of an actor or the state, because code like this tends to break as you add new classes and modify things.

UnrealScript .u packages are compiled strictly in the order specified by the .ini file's EditPackages list, so each package can only reference other objects in itself and in previously-compiled packages, and never in subsequently-compiled packages. If you find that a need for circular references between packages arises, the two solutions are:

1.
    1. Factor the classes into a set of base classes compiled in the first .u package, and a set of child classes compiled in the second .u package, making sure that the base classes never reference the child classes. This is a good programming practice anyway, and it usually works.
       Note that if a given class C needs to reference a class or object O in a later-compiled package, you can often factor that class into two parts: an abstract base class definition C that defines a variable MyO in the first package (but doesn't contain a default value for MyO in its default properties), and a subclass D in the second package that specifies the proper default value for MyO which can only be done from within that second package.
    2. If the two .u packages are inextricably intertwined by references, then merge them into a single package. This is reasonable because packages are intended as a unit of code modularity, and there's not a real benefit (such as memory savings) to separating these inseparable sets of classes into multiple packages.