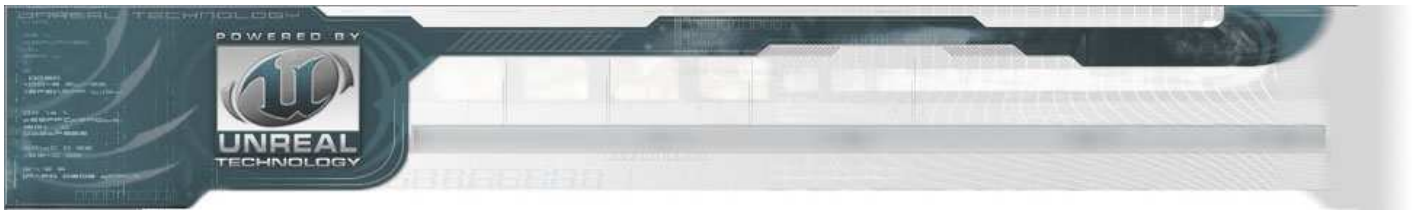

2.3 - Strings In UnrealScript²

Document Summary: This document covers the workings and use of strings in UnrealScript² classes.

Document Changelog: Ported; maintained over time.

Overview

The primary purpose of this document is to cover the workings and use of strings in UnrealScript.

Operators

$ (dollar sign)

```
string $ ( coerce string A, coerce string B )
```

Takes two strings and concatenates them.

Example: `log("Unreal"$"Script"); //prints "UnrealScript"`

$=

```
string $= ( out string A, coerce string B )
```

Concat two strings and assign it to the first.

Example: `MyString $= "add this"; // does the same as: MyString = MyString$"add this"`

@ (at)

```
string @ ( coerce string A, coerce string B )
```

Takes two strings and concatenates them with a space in-between.

Example: `log("a"@"lot"); //prints "a lot"`

@=

```
string @= ( out string A, coerce string B )
```

Analog to $= this operator will concat and assign (but with a space between the two strings).

Example: `MyString @= "concat and assign with a space";`

< (less-than)

```
bool < ( string A, string B )
```

Takes two strings and returns true if the first string is alphabetically before the second string. Note that capital letters are always alphabetically before lower case letters; consider Caps if case does not matter.

Example: `("Monkey" < "Robot") //this is TRUE.`

> (greater-than)

```
bool > ( string A, string B )
```

Takes two strings and returns true if the first string is alphabetically after the second string. Note that capital letters are always alphabetically before lower case letters; consider Caps if case does not matter.

Example: `("Batman" > "Aquaman") //this is TRUE.`

<= (less-than-or-equal-to)

```
bool <= ( string A, string B )
```

Takes two strings and returns true if the first string is alphabetically before or the same as the second string. Note that capital letters are always alphabetically before lower case letters; consider Caps if case does not matter.

Example:

```
("Monkey" &lt;= "Robot")  //this is TRUE.
("Monkey" &lt;= "Monkey") //this is TRUE.
```

>= (greater-than-or-equal-to)

```
bool >= ( string A, string B )
```

Takes two strings and returns true if the first string is alphabetically after or the same as the second string. Note that capital letters are always alphabetically before lower case letters; consider Caps if case does not matter.

Example:

```
("Monkey" &gt;= "Robot")  //this is FALSE.
("Monkey" &gt;= "Monkey") //this is TRUE.
```

== (equal-equal)

```
bool == ( string A, string B )
```

Takes two strings and returns true if the strings are the same. Note that this is a case sensitive compare.

Example:

```
("Monkey" == "Robot")  //this is FALSE.
("Monkey" == "Monkey") //this is TRUE.
("Monkey" == "monkey") //this is FALSE.
```

= (not-equal)

```
bool != ( string A, string B )
```

Takes two strings and returns true if the strings are NOT the same. Note that this is a case sensitive compare.

Example:

```
("Monkey" != "Robot")  //this is TRUE.
("Monkey" != "Monkey") //this is FALSE.
("Monkey" != "monkey") //this is TRUE.
```

~= (tilde-equal)

```
bool ~= ( string A, string B )
```

Takes two strings and returns true if the strings are the same regardless of case.

Example:

```
("Monkey" ~= "Robot")  //this is FALSE.
("Monkey" ~= "Monkey") //this is TRUE.
("Monkey" ~= "monkey") //this is TRUE.
```

-= (minus-equal)

```
string -= ( out string A, coerce string B );
```

Removes all occurences of B from A and assign the result to A;

Example:

```
MyString = "test: this is a test";
MyString -= "test";
log(MyString); // prints: ": this is a ";
```

Functions

Len

```
int Len ( coerce string S )
```

Returns the length of the string.

Example: `Len("this"); //returns 4;`

InStr

```
int InStr ( coerce string S, coerce string t )
```

InStr returns the position of the first occurrence of substring *t* in *S*. If the substring is not found, *InStr* returns -1. Note that the search is case sensitive and so consider calling Caps before using *InStr* if you are not concerned about case.

Example:

```
InStr("These PANTS rock!", "PANTS"); //returns 6
InStr("These PANTS rock!", "pants"); //returns -1
InStr( Caps("These PANTS rock!"), Caps("pants") ); //returns 6
```

Mid

```
string Mid ( coerce string S, int i, optional int j )
```

Mid generates a substring of `S` by starting at character `i` and copying *j* characters. If *j* is omitted, the rest of the string is copied. *i* is clamped between 0 and the length of the string. *j* is clamped between *i* and the length of the string.

Example:

```
Mid("These PANTS rock!", 6, 5); //returns "PANTS"
Mid("These PANTS rock!", 6); //returns "PANTS rock!"
```

Left

```
string Left ( coerce string S, int i )
```

Left returns the *i* leftmost character in the give string *S*. This returns the string to the left of index *i* but not including that character.

Example: `Left("These PANTS rock!", 5); //returns "These"`

Right

```
string Right ( coerce string S, int i )
```

This returns the *i* rightmost character in the give string *S*.

Example: `Right("These PANTS rock!", 5); //returns "rock!"`

Caps

```
string Caps ( coerce string S )
```

Caps returns the capitalized version of the given string *S*.

Example: `Caps("wooo"); //returns "WOOO"`

Locs

```
string Locs ( coerce string S )
```

Locs returns the lowercase version of the given string *S*.

Example: `Locs("WoOo"); //returns "wooo"`

Chr

```
string Chr ( int i )
```

Chr returns the string representation of the given int. This can be anything in the Unicode range 0 - 65535.

Example: `Chr(65); //returns "A"`

Asc

```
int Asc ( string S )
```

Asc returns the numeric Unicode representation of the first letter of the given string *S*.

Example: `Asc("A"); //returns 65`

Divide

```
bool   Divide ( coerce string Src, string Divider, out string
LeftPart, out string RightPart)
```

Split a string into two parts with *Divider* as the cut-off point. Returns true when the string was divided.

Example:

```
Divide("Key=Value", "=", X, Y); // X = "Key"; Y = "Value"
Divide("Key=Value=And More", "=", X, Y); // X = "Key"; Y =
"Value=And More"
```

Split

```
int Split ( coerce string Src, string Divider, out array Parts )
```

Split a string into pieces using *Divider* as cut-off point. Returns the number of elements of the resulting array.

Example:

```
Split("Key=Value=And More", "=", Result);
// Result[0]="Key"
// Result[1]="Value"
// Result[2]="And more"
```

StrCmp

```
int StrCmp ( coerce string S, coerce string T, optional int Count,
optional bool bCaseSensitive )
```

UnrealScript equivalent of the C strcmp/strncmp function. *Count* specifies the max number of characters to compare (strncmp), by default the whole string is compared. *bCaseSensitive* will set the case sensitivity of the compare.

Example:

```
StrCmp("true", "false"); // returns >0
StrCmp("test", "tests", 4); // returns 0 (strings match up to 4
chars)
StrCmp("test", "TEST",,true); // returns not 0
StrCmp("test", "TEST",,false); // returns 0
```

Repl

```
string Repl ( coerce string Src, coerce string Match, coerce string
With, optional bool bCaseSensitive )
```

Replaces all occurences of *Match* with *With* in *Src*.

Example;

```
Repl("This is a test", "is", "was"); // produces "Thwas was a test";
Repl("Two be or not two be", "two", "to", true); // returns "Two be
or not to be"
Repl("Two be or not two be", "two", "to", false); // returns "to be
or not to be"
```

Eval

```
string Eval   ( bool Condition, coerce string ResultIfTrue, coerce
string ResultIfFalse )
```

Returns either *ResultIfTrue* or *ResultIfFalse* based on the *Condition*

Example:

```
Eval(6*9==42, "The answer to life", "the universe and everything");
// returns "the universe and everything";
```

ReplaceText

```
function ReplaceText(out string Text, string Replace, string With)
```

Similar to *Repl* except that the result will be saved to the input string. Str = "This is a test"; Repl(Str, "is", "was"); // Str contains "Thwas was a test";

Str = "Two be or not two be"; Repl(Str, "two", "to"); // Str contains "Two be or not to be"

EatStr

```
EatStr(out string Dest, out string Source, int Num)
```

Moves *Num* elements from *Source* to *Dest*.

Example:

```
Y = "Hello world!";
EatStr(X, Y, 5);
// X = "Hello";
// Y = " world!";
```

GetItemName

```
String GetItemName( string FullName )
```

Accepts a "Package.Item" string and returns the "Item" part of it.

Example:

```
GetItemName(string(self)); // returns the class name
GetItemName("Package.Group.bla.Item"); // return "Item"
```

Special Considerations

String Concatenation and Assignment

You'll often see script code that looks like this:

```
for ( i = 0; i < Count; i++ )
{
      if ( MyString != "" )
      {
            MyString = MyString + ", ";
      }
      MyString = MyString + NextArrayValue[i];
}
```

There are currently two string operators for performing string concatenation and assignment:

```
native(322) static final operator(44) string $= ( out  string A,
coerce string B );
native(323) static final operator(44) string @= ( out  string A,
coerce string B);
```

The above code could be rewritten to look like:

```
for ( i = 0; i < Count; i++ )
{
      if ( MyString != "" )
      {
            MyString $= ", ";
      }
      MyString $= NextArrayValue[i];
}
```

The reason is that the `$=` operator is faster!

Here is what is actually happening for each version:

```
MyString = MyString + NextArrayValue[i];
```

1. Evaluate the left side; look up the address for the `MyString` variable.

2. Evaluate the right side; invokes the **+** operator (`execString_Concat`)
3. Look up the address of the `MyString` variable; copy its value into a temporary buffer for use by the **+** operator.
4. Look up the address of the `NextArrayValue` (`execArrayElement`); copy its value into a temporary buffer for use by the **+** operator.
5. Add the two temporary buffers together; copy that string into `MyString`.

```
MyString $= NextArrayValue[i];
```

1. Look up the address for the `MyString` variable.
2. Look up the address for the `NextArrayValue[i]` variable. Append the value directly to `MyString`.