FString

Reference for creating, converting, comparing, and more with FStrings in Unreal Engine.

Unlike `FName` and `FText`, `FString` can be searched, modified, and compared against other strings. However, these manipulations can make `FStrings` more expensive than the immutable string classes. This is because `FString` objects store their own character arrays, while `FName` and `FText` objects store an index to a shared character array, and can establish equality based purely on this index value.

Creating FStrings

You can declare an `FString` using the following syntax:

```
FString TestHUDString = FString(TEXT("This is my test FString."));
```

Copy full snippet

Conversions

String Variables

From FString

From	To	Example
FString	FName	`TestHUDName = FName(*TestHUDString);` FString → FName is dangerous as the conversion is lossy as FName's are case insensitive.
FString	FText	`TestHUDText = FText::FromString(TestHUDString);` FString → FText is valid in some cases, but be aware that the FString's content will not benefit from the FText's "auto localization".

To FString

From	To	Example
FName	FString	`TestHUDName = TestHUDText.ToString();` FName → FString is dangerous as it is a potentially lossy conversion for some languages.
FText	FString	`TestHUDString = TestHUDText.ToString();`

From	To	Example
		FText → FString is dangerous as it is a potentially lossy conversion for some languages.

Numeric and Other Variables

To FString

Variable Type	Conversion from String	String Format
float	`FString::SanitizeFloat(FloatVariable);`	
int	`FString::FromInt(IntVariable);`	
bool	`InBool ? TEXT("true") : TEXT("false");`	either 'true' or 'false'
FVector	`VectorVariable.ToString();`	'X= Y= Z='
FVector2D	`Vector2DVariable.ToString();`	'X= Y='
FRotator	`RotatorVariable.ToString();`	'P= Y= R='
FLinearColor	`LinearColorVariable.ToString();`	'(R=,G=,B=,A=)'
UObject	`(InObj != NULL) ? InObj->GetName() : FString(TEXT("None"));`	UObject's FName

For other numeric conversions, you can use the **FString::Printf()** function with the appropriate arguments.

From FString

Conversions also exist from FString to int and float numeric variables, as well as to boolean variables.

Variable Type	Conversion from String	Notes
bool	`TestHUDString.ToBool();`	
int	`FCString::Atoi(*TestHUDString);`	
float	`FCString::Atof(*TestHUDString);`	

Comparisons

The overloaded == operator can be used to compare two FStrings, or to compare an FString to an array of TCHAR*s. There is also the **FString::Equals()** method, which takes the FString to test against and the **ESearchCase** enum for whether or not the comparison should ignore case as arguments. If you want the comparison to ignore case, use **ESearchCase::IgnoreCase**, and if not, use **ESearchCase::CaseSensitive**.

```
TestHUDString.Equals(TEXT("Test"), ESearchCase::CaseSensitive);
```

Copy full snippet

Searching

When searching within FStrings, there are two search types. The first, **FString::Contains()**, returns true if the substring is found, and *false* otherwise. FString::Contains() can search for either an FString or a TCHAR*s substring. The ESearchCase enum can be used to specify whether or not the search should ignore case. Also, the `ESearchDir` enum can be used to specify the direction of the search. The default is to ignore case, and to begin searching at the start.

```
TestHUDString.Contains(TEXT("Test"), ESearchCase::CaseSensitive,
ESearchDir::FromEnd);
```

```
2
```

Copy full snippet

The second, `FString::Find()`, returns the index of the first found instance of the substring. `FString::Find()` can search for either an `FString` or a `TCHAR*`s substring. Just like with `FString::Contains()`, you can specify the case sensitivity and the search direction, and the defaults are to ignore case and begin at the start of the string. You can also optionally set an index within the string where the search should begin. If `FString::Find()` does not find the substring, it returns -1.

```
TestHUDString.Find(TEXT("test"), ESearchCase::CaseSensitive, ESearchDir::FromE
```

Copy full snippet

Building FStrings

There are two methods to construct strings out of substrings or other variable types. The first, concatenation, only takes FStrings as arguments. You will need to convert other types of variables to FStrings before concatenating them. The second, `Printf`, can take numeric inputs like int and float, and also allows you to set the formatting of the inputs as they are added to the string.

Concatenation

There are two operators for concatenating strings:

Operator	Description	Usage
`+=`	Appends the supplied string to the FString object.	`StringResult += AddedString;`
`+`	Creates a new FString object and appends the supplied string.	

Printf

FStrings constructed with **FString::Printf** can be stored into FStrings, as well as displayed to the screen with UE_LOG debug messaging. The format argument has the same specifiers as the C++ printf function, as seen in the below example.

```
FString AShooterHUD::GetTimeString(float TimeSeconds)
{
// only minutes and seconds are relevant
const int32 TotalSeconds = FMath::Max(0, FMath::TruncToInt(TimeSeconds) % 3600);
const int32 NumMinutes = TotalSeconds / 60;
const int32 NumSeconds = TotalSeconds % 60;

const FString TimeDesc = FString::Printf(TEXT("%02d:%02d"), NumMinutes, NumSeconds);
return TimeDesc;
}

```

Copy full snippet

When using %s parameters to include FStrings, the `*` operator must be used to return the TCHAR* required for the %s parameter.

Manipulating FStrings

There are many functions for manipulating strings. Some will be covered here, but for the full list of available FString functions, see UnrealString.h or the API documentation on FString. There are functions for copying subsections of strings: Left, Right, and Mid. You can split a string into two strings at the location of a found substring. This is done using the `Split` method. Another method for splitting strings is the **ParseIntoArray**, which splits a string into an array of strings separated by a specified delimiter. Case conversion is done by using **ToUpper** and **ToLower**, converting the string to upper and lower case respectively.

FStrings in HUDs

Canvas

To display an FString in a HUD using Canvas, call the **FCanvas::DrawText()** function:

```
Canvas->DrawText(BigFont, TestHUDString, 110.0f,110.0f);

```

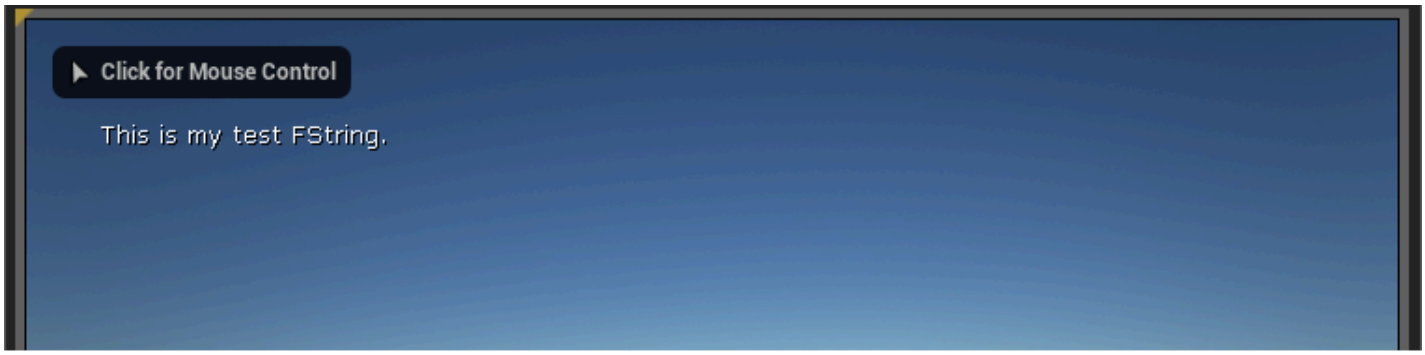
Copy full snippet

> You must call the **DrawText()** function within your HUD class's **DrawHUD()** function, or call it in a function chain that begins with DrawHUD().

Debug Messaging

FStrings can be printed to the **Viewport** as well as to the **Output Log** for debugging purposes.

Print to Viewport



To print debug messages to the **Viewport**, use **UEngine::AddOnScreenDebugMessage()**. This function takes the following four parameters (in order):

Parameter Name	Parameter Type	Description
Key	int	A unique key to prevent the same message from being added multiple times. Use -1 as the key to have your debug message be transient.
TimeToDisplay	float	How long to display the message, in seconds.
DisplayColor	FColor	The color to display the text in.
DebugMessage	FString	The message to display (FString).

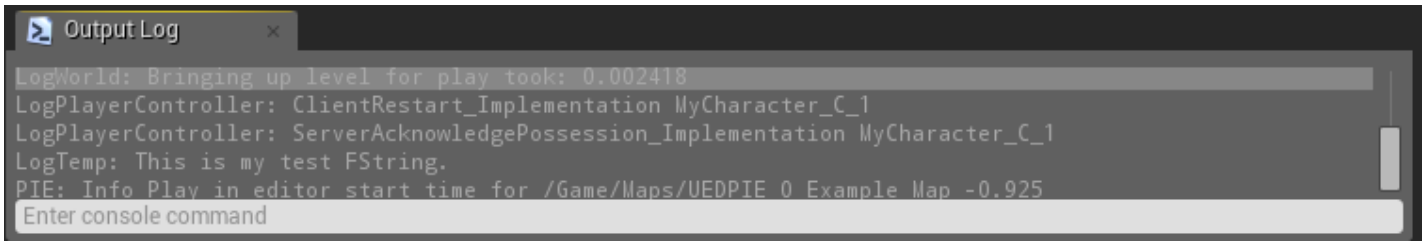
Example:

```
GEngine->AddOnScreenDebugMessage(-1, 5.f, FColor::Blue, TestHUDString);

```

Copy full snippet

Print to Output Log



UE_LOG uses printf markup for parameterization.

```
UE_LOG(LogClass, Log, TEXT("This is a testing statement. %s"), *TestHUDString);
```

Copy full snippet

- LogClass is the log category. You can use an existing category (set with the DECLARE_LOG_CATEGORY_EXTERN macro in OutputDevices.h) or define your own using DEFINE_LOG_CATEGORY_STATIC.
- Log is the verbosity level to use. Verbosity is defined in the **ELogVerbosity** enum. Valid values are Fatal, Error, Warning, Display, Log, Verbose, or VeryVerbose.
- The next argument is the text you wish to output, including the markup for the parameters.

> This example uses a %s parameter, so the `*` operator is used to return the TCHAR* required for a %s parameter.

Messages printed with UE_LOG can be found in the **Output Log** (**Window > Output Log** in Unreal Editor).

Encoding Conversion Macros

The FString class is built upon a TArray of TCHARs. There are multiple macros available to convert an application string (TCHAR*) to either ANSI or UNICODE character sets and vice versa. The macro definitions are found in Engine\Source\Runtime\Core\Public\Containers\StringConv.h.

If the string is relatively small, the allocation is made on the stack as part of the converter class; otherwise the heap is used to allocate a temporary buffer. The size before using the heap is a template parameter, so you can tune this to your application. This is safe within loops because the scoping of the class pops off the stack allocation.

The common conversion macros are:

- TCHAR_TO_ANSI - Converts an engine string (TCHAR*) to an ANSI one.
- ANSI_TO_TCHAR - Converts an ANSI string to an engine string (TCHAR*).

> The objects these macros declare have very short lifetimes. They are meant to be used as parameters to functions. You cannot assign a variable to the contents of the converted string as the object will go out of scope and the string will be released.

> The parameter you pass in must be a proper string, as the parameter is typecast to a pointer. If you pass in a TCHAR instead of a TCHAR*, it will compile and then crash at runtime.
>
> Usage: **SomeApi(TCHAR_TO_ANSI(SomeUnicodeString));**