BEFE - Grey Paper

Class: String

A detailed description of the BEFE `String` class including design overview, issues, motivation, description, and method calls.

Author:	Bruce Ferris
Date:	01 February, 2016
Status:	Draft

Table of Contents

Class: String

The purpose of this paper is to describe the capabilities of the **BEFE** `String` class.

Although the current implementation of `String` is written in C++, the language itself should not get in the way of describing or understanding why the **BEFE** `String` is so different from other implementations you may be more used to.

We figure that since a `String` is such a basic concept and is one of the most fundamental tools required to accomplish many software development projects, it deserves special attention from the very outset.

Overview

The **BEFE** `String` class is one of the most powerful classes in **BEFE**. We've all encountered `String` classes before but, to my knowledge, none quite like this one...

What we find surprising is the number of `String` classes available out there and how poorly designed and implemented the majority of them are. Another surprise is how dated many of the implementations are. This is not only frustrating but it seems, in some sense, almost a crime...

Let's take, for example, the oddly sainted Java and Python implementations of a `String` class... The first "*gotcha*" is that neither of them provide "mutable strings".

What does that mean? It means that the designers of both language implementations skirted the real issues in order to make it "run faster" – with the "real issue" being "garbage collection" or, to put it in realistic terms, "reference counting".

The problem with the exclusively *immutable* approach is 1) immutable strings don't run ANY faster than mutable strings, and 2) this hands the problem on to YOU, the software developer instead of the language developers solving the "real problem" to begin with.

> **Note:** It's a bit surprising how many experienced Java and Python developers I've talked to that hadn't even realised the strings they use day in and day out aren't mutable.

And all this simply because the language developers could not, or were unable to, solve some fundamental problems, and questions, like these...

- What to do when a mutable `String` changes size/length?
- Isn't mutating really going to fragment memory?
- Why go to all that trouble when we can let the end user/programmer work around it instead (over and over again and in an inconsistent manner) and save us the bother?
- Since we don't personally speak any foreign languages, what care do we have for catering to reasonable Unicode standards?

Another thing we here at **BEFE** cannot understand is the proliferation of `UCS-2/UTF-16` surrogate pairs everywhere!!! Although this topic is covered elsewhere we're going to mention it again here since a thorough understanding of it is needed here for practical purposes...

The Great “Surrogate Pair fiasco”

Back before Unicode became an “official standard” developers tended to think in terms of Code Pages when it came to special symbols and “foreign” (meaning “non Latin”) language characters.

> **Note:** The phrase “official standard” is a bit of an oxymoron if you think about it since there is no SINGLE standards body when it comes to computing standards. So, I guess, it would be better to say “accepted standard”. Nevertheless...

This meant that software developers kept a “one byte per character” viewpoint in their heads.

What this meant in practice is that bytes in the range `0x00..0x7F` were treated as “proper ASCII”, while bytes in the range `0x80..0xFF` are “foreign characters” whose exact meaning can only be known if you know what “Code Page” they were created in.

So, Microsoft and many others went crazy with this “Code Page” idea, bought into it whole heartedly, and they quickly became disliked and hated by the majority of software developers around the world. So, what was the big deal about Code Page?...

Firstly, as already stated, you can't know what a “character” greater than 0x7F actually means (or why it's there) unless you know the Code Page context it was put there in.

Secondly, there wasn't any “standardised way” of passing this missing context information along with the bytes themselves.

All this may seem like a bit of a “DUH!!!” and it is. Then, along came Unicode...

Unicode defines characters in the range `0x000000..0x10FFFF`. This means, there are 21 bits in each Unicode character. Clearly, some parts of this 21-bit range are declared “off limits” in the standard (like `0x120000` through `0x1FFFFF` , as obvious examples).

Unfortunately, at the same time Unicode was being developed, people hadn't got their heads around the whole problem, and they starting using the `ISO 10646` standard which defined `UCS` – a 16-bit character standard. As of 1991, the Unicode Consortium – the people that “own” the Unicode standard – decided to co-develop the standard along with `ISO 10646`, including version 2 of `ISO`'s `UCS` (later renamed `UCS-2` because the standard adopted a newer version), and renamed it `UTF-16` – surely you can see the storm clouds forming?

The whole problem with UCS-2 and UTF-16 was that they both introduced “byte ordering” into the equation, causing a set of headaches that weren't even needed.

The secondary problem was that it limited the Unicode Characters to 16-bits which, in the long run, wasn't a reasonable limit.

> **Note:** But, to be fair, some of us remember back to those “good old days” when people like Microsoft couldn't imagine needing more than 640K of the “huge 1 megabyte address space” the 16-bit processors gave you at time. Just imagine! Too bad *they* didn't exercise their imagination!

Then, in January 1993, `UTF-8` was officially presented to “the public” at a USENIX conference in San Diego.

Meanwhile, Microsoft paid no attention – after all, USENIX is a Unix thing for God's sake, what's

that got to do with Microsoft? – and introduced their first version of Windows NT, later in 1993. Windows NT stuck with UCS-2, its 16-bit limit, associated byte order problems, and the additional behaviour of wasting half the space needed to store the text since most text was in the 0x00..0x7F ASCII range.

Now THAT was a DUH moment!!!!

Then, on top of Microsoft's belligerent behaviour, the guys at Sun Microsystems kept this same mindset (UCS-2/UTF-16) when designing Java, even though by that time UTF-8 was clearly the sane alternative, especially in the Unix world that Sun Microsystems lived in. Grrrrr....

Basically, UTF-8 doesn't have byte ordering problems, only contains "multi-byte characters" if and when they're used, and UTF-8 strings sort just fine as if it were a single-byte string – so hard-line C/C++ developers could easily continue to use strlen and friends if they wished.

Not just that, but UTF-8 even allows 0x00 INSIDE THE STRING. Hurrah!!!!

It's no wonder UTF-8 took off, was accepted so eagerly by the sane internet community, and became the *de facto* "minimum requirement" for the majority of the RFCs arising in the internet world.

Still, UTF-16 is still hanging about, a fact that needs to be recognised, handled (and avoided at all costs where possible) as far as **BEFE** is concerned.

The `String` class and UTF-8

Although we've already covered some of this above along with pointing out **BEFE**'s contempt for `UTF-16`, we do need to add a little justification for our embedded implementation of `UTF-8` in the `String` class...

Unlike the "good old days" where everything was ASCII or could be pigeon-holed into externally identified "code pages", we have been "honoured" with an international Unicode standard and encoding scheme: UTF-8.

What UTF-8 does is handle "normal" ASCII text the same as it has always been handled where the first 32 byte values are "Control Characters" in the range `0x00..0x1f` and the "Printable Characters" are in the range `0x20..0x7F`.

But what UTF-8 does is, instead of shoving "foreign characters" in the limited range of 0x80..0xFF, it opts for a multiple byte sequences to represent each "foreign character".

Needless to say, since Unicode specifies characters in the range `0x000000..0x10FFFF`, this means that UTF-8 encodes Unicode characters `0x80` and greater into variable length sequences from one to four bytes long.

> **Note:** This is an important "feature" of UTF-8, because it means that the longest UTF-8 character sequence will fit into a single 32-bit register. Handy, eh?
>
> Plus, what's so neat about UTF-8 is that given an arbitrary position in a UTF-8 string, you can find the beginning or end of the character easily OR tell if it's "invalid" pretty much right off, without having to start at the beginning of the string.

UTF-8 itself is quite clever and ensures that, given two strings, one can do a byte-to-byte comparison and determine whether the strings are equal to each other or if one encoded string is "less than" the other string as far as Unicode is concerned – of course, this "less than" and "greater than" comparison tell you nothing about localised sorting preferences but we'll cover that elsewhere...

The important thing, however, is that *a distinction has to be made*, and available at runtime, between a `String`'s *Byte length* and its *Character Length*.

No biggie, one would think. But what "those other guys" seem to do (Java, Python, Microsoft, et al) is they figure it's easier to keep these characters internally as 16-bit values, even if the value would fit into one byte.

Don't ask me to try and justify why they did what they did, but any bother they saved themselves making their strings immutable, they threw away by having to converting them back and forth between 16-bit values and UTF-8 or other encodings AND they eat up about twice the memory required for most `String` instances. Go figure!!!!

> **Note:** Not only that, by keeping Unicode characters in 16-bit values, they get into the horrid position of having to introduce needless "surrogate pairs" for characters greater than `0xFFFF - things` like musical notations, mathematical symbols, ancient Egyptian Hieroglyphs, Assyrian etc. You know, the good stuff!!!
>
> So what happens in UTF-16 with these characters? Their `String` stops being "fixed length" and becomes "variable length" again (e.g., byte counts don't match character counts). And that's the whole problem there were trying to avoid to begin with. So, they didn't solve "the problem" – they just side-stepped it.

Anyway, what **`BEFE`** does is to keep every `String` in its UTF-8 form while acknowledging the fact that proper `String` management means the byte and character counts may not match. With that in mind it processes the bytes and characters in-place. Problem solved!

Okay, fair enough, it took some head scratching and hard work to get it all working cleanly and bug free, but it was well worth the effort since your `String` contents stay in a clean and consistent form whether they're in memory, on disk, or being sent across the network, and *no wasted resources spent on constant encoding and decoding* or needlessly wasted memory.

End of THAT problem as far as **`BEFE`** is concerned!!!

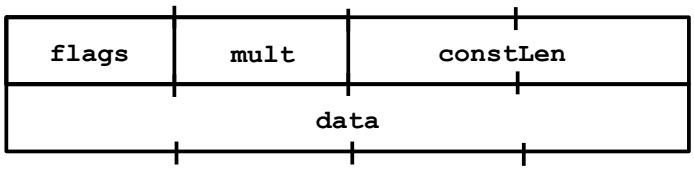
BEFE `String` Design Motivations

The following motivations, observations, and expectations drove the design and implementation of our `String` class...

- The vast majority of `String`s contain mainly `ASCII` characters
- UTF-16 is something that has to be addressed, and properly, but if you look around you'll find that strange things like Unicode “surrogate pairs” (an archaic left-over from the old UCS-16 days) seem to abound everywhere you look. While we don't want surrogate pairs hanging around inside **BEFE**, we do understand that we'll come across them. So, with a lot of grumbling, we also have to provide the ability to form **BEFE** `Strings` from these silly UCS-2 historical artefacts – but, we don't have to like doing it, do we???!!!
- `Strings` MUST be capable of mutating, unlike Python, Java, and many other implementation.
- Immutable Strings, however must be managed as well since they'll appear, from time to time, from immutable sources like DVDs, CDs, over communication protocols, and even in C/C++ and it's bothersome `const` keyword.

`String` Structure...

The String class is laid out in memory something of like this when illustrated as bytes in memory...



`flags` are described in more detail in the `String.h` header file, but consist of the following...

- `isMutable` - "Is this `String` mutable?"
- `isByte` - "Does this `String` contain only ASCII Bytes?"
- `isChar` - "Does this `String` contain any UTF-8 Characters?"
- `isSensitive` - "Is this `String` case sensitive?"
- `type` - "What kind of `String` is this?"

`mult` indicates number of occurrences for repeating `String`s (more on this later).

`constLen` indicates the length of the `String` in bytes (more on this later).

`data` means something different depending on the type of `String` (see below).

So, each `String` is 8 bytes long in its basic form, but it is the `data` part, in combination with `flags.type` that's the most important. and its meaning depends on the `String`'s type...

`String` Types

The `String` class manages several types, or kinds, of `String` (taken from `String.h`)...

- `Null` `// 0: Null`
- `Const` `// 1: C++ (const char *, NUL terminated)`
- `ConstLen` `// 2: C++ (const char *, Length terminated)`
- `ASCII0` `// 3: ASCII 0 character (Empty)`
- `ASCII1` `// 4: ASCII 1 character`
- `ASCII2` `// 5: ASCII 2 characters`
- `ASCII3` `// 6: ASCII 3 characters`
- `ASCII4` `// 7: ASCII 4 characters`
- `UtfInline` `// 8: 1 Unicode character`
- `Buffer` `// 9: Sbuffer`

Each of these are briefly described below...

The `Null` type is the initial **StartUp** state of all `String`s. It means it will answer `True` to the `IsNull` check AND it means the `String` is **Empty** (see below).

The `Const` and `ConstLen` types sort of mean the same thing but with a subtle difference. They exist to indicate that `data` contains a pointer to the equivalent of a "`const char *`", but...

A `Const` type `String` points to a typical C/C++ `NUL` terminated string like expected by Kernighan and Ritchie's famous (and creaking) C library functions like `strlen` and friends.

A `ConstLen` type String points to a sequence of bytes that we already know, or have already figured out, how many bytes there are.

> **Note:** This distinction helps enormously in the overhead a typical C function causes by calling `strlen` all the time. The point being that you should only need to call `strlen` once and `String` then stores the answer in `constLen` for you. Cool, eh? Especially because you don't have to do anything about it because it *just works*!!!

The `ASCII0` to `ASCII4` types are used to store, inline, really short ASCII strings (quite a common occurrence it turns out)... Note, however, that anything other than ASCII, or longer than 4 bytes, turns into type `Buffer` automatically (see below). The special case of `ASCII0` is implemented so **`BEFE`** can distinguish, semantically, between the two separate concepts: **Null** and **Empty**.

> **Note:** As far as we're concerned, this distinction is often a commonly "grey area" in many developers heads and we'll cover it in detail elsewhere. However, it is important to point out that the distinction between **Null** and **Empty** is an important one in **`BEFE`** even though the two are often mixed together – especially in places like relational databases where `NULL` is a fundamental concept.

The `UtfInline` type is used to indicate a single UTF-8 character. As mentioned earlier, it's quite handy that the maximum encoded length of a UTF-8 character is 4 bytes. So, cleverly, we use `data` to store the UTF-8 character "inline" - hence the type name.

While `UtfInline` is not as common as types ASCII0..ASCII4, it nicely covers cases where your code is messing about with special UTF-8 stuff like line-drawing using UTF-8, including arrows, and other special use symbols in your `String`s.

The final type, `Buffer`, is for the common case of Strings that are long than 4 bytes and, need to be mutable and, such, can justify the overhead of a separate `Buffer` class instance – the `Buffer` class is described elsewhere but, simply enough, it's a pointer to some allocated bytes with a header at the beginning telling the implementation all about the variable length contents of the `Buffer`.

> **Note:** Actually, the `String.data` member is an `SBuffer` instance, but an `SBuffer` is only a minimal specialisation of the `Buffer` class with some extra `String` specific methods thrown in for ease of use in the `String` code. So, just think of it as a `Buffer`.

`String` Methods and the C++ `const` keyword

In the following Method descriptions you will find various `String` methods that are declared `const` but, looking at the name and reading the description, they seem like they shouldn't be...

Those methods generally DO or MAY change the internal state of the `String` instance but we've still declared them to C++ as `const` so that we can easily use them in expressions passed as parameters et cetera (or what the "guru" crowd likes to call an R-value – what complete and utter rubbish!

So, don't take the `const` "literally" because we've applied it to mean the `String` instance's *contents* NOT its *state*.

So, this "off the beaten track" trick let's you do things like this without taking a copy of the `String`...

```
String myString = "Hello";
...
if (myString.Insensitive() == "hello")...
```

And what that does is sets `myString`'s case sensitivity **off** so subsequent comparisons will be insensitive until it's turned back to **on** – BTW, it's still stored in mixed case as a pointer to the `const` C string as `String` type `ConstLen`, but still compares insensitive.

See what we mean about the `const` not referring to *state,* but referring to the *contents* instead?

In case you're wondering... No, we don't feel bad in the slightest about "fooling C++" in that way, that's exactly the sort of crap you have to deal with when you're coding something "properly" in that weird language.

If you want to be really geeky, you could accomplish the same thing without a copy by doing the following but it's not as readable, as far as we're concerned, and takes a teeny bit more runtime overhead...

```
String myString = String("Hello").Insensitive().Consumamble();
...
if (myString == "hello")...
```

String BEFE Lifecycle

This section briefly covers the BEFE Lifecycle as implemented for the String class. The BEFE Lifecycle is covered in a separate BEFE Grey Paper but we figure this is a good place to summarise the details as they apply to the String class.

String StartUp and ShutDown

The **BEFE Lifecycle** `StartUp` and `ShutDown` methods are similar in function to the standard C++ concepts of the Constructor and Destructor but with some important differences...

- Both methods return a Status result instead of throwing exceptions. This occurs throughout the BEFE implementation and is a distinguishing feature compared with how things are expected to perform in C++, Java, etc. We'll cover the justification for this in a separate document.
- `StartUp` is responsible for initialising a new instance “from out of limbo”. However, the state of the instance MUST be such that no external memory resources are allocated outside of the memory where the instance resides.
- The `ShutDown` method is responsible for destroying an existing instance, releasing any allocated memory resources, and returning itself to a “state of limbo”.

And here's where the `Reset` and `Clear` methods come into play...

`String` `Reset`, `Clear`, and Null versus Empty

`Reset` is responsible for achieving the same `String` state as calling `ShutDown` followed by `StartUp`. The reason we've separated it into a single method is to allow you to code specific quirks into it if you wish, separate it from the `Clear` method, and provide a clean point for debugging operations.

In essence, `Reset` is responsible for setting the *Value* to the **Null** state (more on that below).

The `Clear` method is subtly different to `Reset` in that instead of ensuring "no memory resources" on return, it provides a way to set the *Value* in the **Empty** state (more on that below). In the *Empty* state, the *Value* may or may not have external memory resources allocated. More importantly, being *Empty*, it may behave differently than if it were *Null* – but only if that makes sense for the specific class.

> **Note:** In the case of the `String` class, it DOES make sense to have separate *Null* and *Empty* states, so the implementation reflects that difference.

Long time C programmers are used to the nasty and ubiquitous macro `NULL` which is a mindlessly simple way of saying "zero" or "a null pointer".

However, in **`BEFE`**, the concept of **Null** does NOT mean Zero in the "object" or "value" world any more than **NaN** means Zero in the Number world. They are two fairly closely related, but subtly different, concepts. Neither, however, have anything to do with "Zero".

> **Note:** I first encountered **Null** as a formal concept, as distinct from the C `NULL`, in the relational database world at Oracle. However, Oracle's concept of **Null** breaks down a bit when it comes to strings. Hence the importance of mentioning here how **`BEFE`** approaches that problem.

In the context of **`BEFE`**, **Null** should be thought of as it is in the relational database world – as "unknown". Hence, all operations on a *Null* value produce a *Null* result, pretty much regardless of what that operation is.

> **Note:** The obvious exception to that rule is the *Is Null t*est... clearly, if the "is this *Null*?" test returns *Null*, how would you know what that means or worse still, how could you even know it returned *Null* since you couldn't test for it?

Anyway, back to the point... *Null* means "unknown" or, more commonly, "no value". This second meaning comes in handy when trying to keep track of what has or has not been set by the user/caller.

But *Null***,** as a concept, is different to *Empty* when it comes to sequences, sets, collections, lists, arrays and other "containers" – and `String` is conceptually an `Array` of `Char` values.

This is the point where, in our opinion, Oracle and other databases sort of blew it... Some or most of them treat empty strings as the *Null* value.

But hold on, surely just because some value is *Empty* it shouldn't mean the same thing as *Null*, should it?

For example, if you filled your name into a `String` on a form, and one by one shortened it by a

single character, its length gets one less each time. Finally, after you've deleted the last character in it, how long is it? Is it *Null* long or is it *Zero* long? Or, "Is it *Null* or is it *Empty*?"

At **BEFE** we put a stake in the ground here and say if it is *Zero* long it is *Empty.* We expect that if something is *Null* long then it IS NULL**,** and that means nobody ever put anything into it OR they explicitly set it to be *Null*. Once again, problem solved without any rocket science!!!

Subtle difference but well worth pointing out.

String MoveFrom, CopyFrom, and Consumable

The String class implements the standard **BEFE Lifecycle** concepts of **Consumability**.

The concept comes into play when it comes to "context" and a value's "lifetime". Lets say, for example, your code has spent the time and trouble to build up the contents of a local variable so it contains a "whole lot of stuff".

Then, at a proper place in your code, you want to return "all that stuff" to the calling procedure.

That's where *Consumability* comes into play. Unlike the apparently "accepted standard" of "copy semantics" and "reference semantics" through constructors, destructors, and "operator=", **BEFE** handles the problem with the MoveFrom, CopyFrom, Consumable, and NotConsumable, methods in the **BEFE Lifecycle** instead.

What these methods do is allow you to do is, instead of copying, move your entire variable's contents somewhere else (like the caller's variable) because you're finished with it.

These methods are entirely consistent with the standard C++ constructor, copy-constructor, destructor, and the operator = methods but doesn't demand "copying the whole bloody thing"just to move it somewhere else.

As already indicated, this whole subject is discussed in a separate BEFE Grey Paper so we'll leave it there for now.

`String` Mutability

Now, we get into the sometimes contentious subject of `String` mutability. In the context of the **`BEFE` Lifecycle**, the **<Mutability>** concept is not contentious in the slightest since it's a) given, b) well defined, and c) justified. However, in the world of `String` classes as implemented elsewhere, this subject can sometimes cause tempers to flare...

Departing from the "norm" of having immutable Strings, **`BEFE`** lets you decide (within reason), when and where you want your `String` to be mutable and when and where you want it to be immutable.xxxWe do apply some caveats... If you build a String from a "`char const *`" (some people like calling it "`const char *`" but we don't here at BEFE for reasons explained elsewhere), then the C/C++ language especially generates it in "read only"memory (unless you manually cast from a non-`const` to a `const`, in which case its not "real read only", it just acts like it).

The `String` class provides *Mutability Methods* described later here, BUT when applied properly these methods allow you to "change your mind safely" regarding a `String`'s mutability. For example, if a string is of type `ConstLen` (and hence, immutable), you have the option of calling the method `MakeMutable` which will convert it to a `String` of type `Buffer`. Likewise, you may have a `String` of type `Buffer` and call the method `MakeImmutable` – but this will NOT turn it into a `ConstLen` type. Instead, it will simply clear the `String.isMutable` flag and leave it as a type `Buffer`.

The `String` *Mutability Methods* provide the mechanism whereby you can take advantage of immutability with some of your `String` instances while clearing the path for fully mutable in-place `String` manipulation in others – thereby giving you the best of both worlds.

A good example of places where immutable `Strings` are the "best bet" is a scenario where your application code is reading an entire text file into memory in a single memory block of contiguous bytes – this is quite a rational scenario when you're only reading files, are fairly sure they're reasonably "small" (like source files) AND you're going to "throw them away" pretty soon after you're finished with them.

In that scenario, you may conceivably wish to "remove bytes" without "harming" the allocated memory if you're careful – but you certainly can't "insert bytes" into the memory block without risking overflow.

So, if it makes sense to keep the bytes as an immutable `String`, you can do that without any problem. And, if you need to change the bytes in a `String` by adding, inserting, or removing them, you can do that too.

Basically, **`BEFE`** lets YOU decide whether your String should be immutability on a case by case basis, instead of foisting immutability on you like Java, Python, and many other `String` classes do.

`String` Array Indexing

The String class implements several Array-like indexing specifiers including...

- `Int` - Specific Character
- `Range` - Range of Characters
- `Span` - Span of Characters
- `Slice` - Slice of Characters

Each of these are described in detail in a separate BEFE Array Indexing Grey Paper but, for purposes of clarity we'll briefly summarise them again here...

The `Int` indexing indicates a specific Character position where the index value, if zero or positive, indicates Character position zero through `Length()-1`.

If the `Int` index value is negative it wraps from the end of the `String`, where `-1` indicates `Length()-1`, the index `-2` indicates `Length()-2` and so forth.

> **Note:** With this and the other indexing mechanisms, the "negative wrapping" only wraps once – it DOES NOT keep wrapping.
>
> For example, if the String is 5 characters long, index `-1` indicates Character position `4`, index `-2` indicates position `3` and so forth. But, position `-6` will return an `IndexOutOfRange` error.

The `Range` indexing specifiers indicate "inclusive ranges" of Characters in the `String` and follow normal **`BEFE`** `Range` index normalisation rules.

The Span indexing specifiers indicate "non-inclusive ranges" or "spans" of Characters in the String and follow normal BEFE Span index normalisation rules.

The Slice indexing specifiers indicate "non-inclusive stepped ranges" or "Slices" of Characters in the String and follow normal BEFE Slice index normalisation rules.

As far as BEFE Index Normalisation rules go, the following apply...

- `NaN` is equivalent to `Null`, and will return an error for `Int` index specifications
- `NaN` will "fill in the blanks" for `Range`, `Span`, and `Slices` specifications and, when normalised, will result in a meaningful value based on where the `NaN` is positioned in the specification

Examples of `NaN` used in a `Range` specification when normalised to a length of 15 Characters are...

- `"..10"` is equivalent to `"NaN..10"` and `"0..10"` inclusive, resulting in 11 Characters
- `"5.."` is equivalent to `"5..14"` inclusive, resulting in 10 Characters
- `"5..20"` will produce an error
- `"-1..-10"` will produce `"14..5"`, which will indicates a **Reverse Range**, processing the Characters from position 14, backward to and including position 5, resultig in 10 Characters

Examples of `NaN` used in a `Span` specification when normalised to a length of 15 Characters are...

- `":10"` is equivalent to `"NaN:10"` and `"0:10"` exclusive, resulting in 10 Characters
- `"5:"` is equivalent to `"5:14"` exclusive, resulting in 9 Characters
- `"5..20"` will produce an error
- `"-1..-10"` will produce `"14..5"`, which will indicates a **Reverse Range**, processing the Characters from position 14, backward to and excluding position 5, resulting in 9 Characters

Examples of `NaN` used in a `Slice` specification when normalised to a length of 15 Characters are...

- `":10"` is equivalent to `":10"` and `"0:10"` is the same as the equivalent Span (see above)
- `":10:2"` is equivalent to `"0:10:2"`, exclusive, resulting in 5 Characters, step 2 – or every other one
- `"5:20"` will produce an error
- `"-1..-10:-2"` will produce `"14:5:-2"`, which will indicates a **Reverse Range**, processing every other Characters from position 14, backward to and excluding position 5, resulting in 5 Characters

BEFE String Literal Escape Sequences

Before we get into the "real meat" of the `String` methods we need to discuss one final subject, that of `BEFE` **String Literals** and **Escape Sequences**.

> **Note:** The *escape sequences* we're talking about here are typically expressed in terms of the backslash character `'\'` followed by special sequences of characters to represent non-displayable characters like line-feed (`'\n'`), carriage return (`'\r'`) and others such like.

Various places in the current **BEFE** implementation, and many more planned for the near future, use *escape sequences* in *string literals* in a similar but different fashion from what you may be used to in Java, C++, and Python.

These languages basically "inherited" their string literal escape sequences from the C programming language and, through time, added a few more escape sequences here and there – and all in an rather haphazard and inconsistent way.

Well, in **BEFE** we had to attack the same problem for similar reasons but, since we started from "scratch" we had the luxury of being in the position to re-think the problem and do something a bit more sane about *string literals* and *escape sequences*.

> **Note:** It's still one of those great computing mysteries why Microsoft kidnapped the `'\'` character and started using it instead of `'/'` for their file system directory name separator. It's one of those disgusting and frustrating things good old **M$** did that we, unfortunately, just have to live with. Still, it really sucks and we'll happily grumble about it whenever we can.

Yes, like other sane programming environments, we still use the `'\'` character to start escape sequences but, we honour a different set of sequences from the "other languages".

For example, we DO NOT explicitly honour Unicode escape sequences like `"\u+2502"` et cetera because of our intense dislike, as mentioned earlier, of Unicode *surrogate pairs* – plus they're not needed in **BEFE** anyway because we rightly UTF-8 automatically, and UTF-8 has no need for surrogate pairs.

Essentially, we handle the following two-character escape sequence combinations...

- `\a` – `NUL` or `0x00`
- `\b` – Backspace, `BS` or `0x08`
- `\t` – Horizontal Tab, `HT` or `0x09`
- `\n` – Line Feed, `LF` or `0x0A`
- `\v` – Vertical Tab, `VT` or `0x0B`
- `\f` – Form Feed, `FF` or `0x0C`
- `\r` – Carriage Return, `CR` or `0x0D`
- `\e` – Escape, `ESC` or `0x1B`

In addition to these, we also handle a variable length '\x<hexDigits>' sequence where <hexDigits> are any contiguous set of hexadecimal digits 0..9, a..f, and A..F. These are used to represent any arbitrairy set of bytes.

> **Note:** No, we DO NOT handle '\o<octalDigits>' because, as far as **BEFE** is concerned, octal went the same way as Digital Equipment Corporation, or DEC, Honeywell, SDS, ICL, and a lot of now defunct computer manufacturers. For historical and not quaint and odd reasons, they liked 6 bit ASCII, 12 and 24 bit words, 48 bit floating point, and other oddities that demanded the use of octal to make sense.
>
> Since BEFE was started well after the "discrete transistor era" ended, we saw absolutely no reason to keep the octal legacy going just for "old times sake".

Once a character is encountered that is not a valid <hexDigit>, the sequence ends.

> **Note:** If the sequence of valid <hexDigit> characters is zero long, the '\x' is simply discarded. So, the string "\xj" is equivalent to "j".
>
> And, if the sequence is an odd number of characters long, a leading zero is assumed. This allows you to specific single digit hexadecimal bytes in strings. So, the string "abc\xadef" is equivalent to "abc\ndef" since "\xa" gets converted to the byte 0x0A.

If a '\' is encountered and it's not followed by one of the listed single escape characters, the '\' is removed from the string and we carry on with the following character as if the '\' wasn't even there.

> **Note:** The only exception to this rule is that in some parts of BEFE, when **BEFE-Wiki** file (discussed elsewhere) is being processed, a '\' at the end of a line, the file, or last non-blank character in an input line. In that case the '\' is left in the String for some other code to interpret as it sees fit – normally as a "line continuation indicator" like C, C++ implements in their preprocessor, and Python implements in all source input lines.

Take, for instance the string "Hello\nWorld!\nSome Hex: \x010203xxx". If you processed this using the Unescape procedure described at the end of this paper, you would find the following bytes would result in bytes in memory...

```
00000000: 48 65 6C 6C 6F 0A 57 6F 72 6C 64 21 0A 53 6F 6D  Hello.World!.Som
00000010: 65 20 48 65 78 3A 20 01 02 03 78 78 78           e Hex: ...xxx
```

Note that the "xxx" terminated the "\x" *escape sequence* in the input string because the 'x' character is not a valid <hexDigit> character.

Note also, that another "trick" is to simply end the sequence with a '\' character followed by a non-single character escape character, because '\' is not a valid <hexDigit>. So, for example, the

string "abc\x202020\def"would result in the string "abc def".

C++ Style Methods

This section covers the C++ style methods. Typically, though we're not overly fond of it, the C++ style is to return "objects" instead of the **BEFE** preferred `Status` return. Still, those of you that are adamant lovers of the C++ style can consider yourself a winner in that respect (for now) except for the following fact...

We DO NOT and WILL NOT implement C++ exceptions in BEFE code. Sorry, but that's a fundamental **BEFE** design mandate and is covered and justified in a separate document...

`String` methods: `C++ Lifecycle`

These methods are here to wrap **BEFE** Lifecycle methods into the C++ way of thinking.

As such, they are pretty small wrappers around the **BEFE** Lifecycle methods described later.

Since we're gearing this document towards experienced C++ developers, we won't bother adding too much here except a quick run-down of the methods we've implemented...

Method	Description
`String()`	Constructor
`~String()`	Destructor
`String(Byte *buf, UInt size)`	Construct from memory (size is bytes)
`String(String const &that)`	Copy Constructor
`String(Char that)`	Construct from **BEFE** `Char`
`String(char const *that)`	Construct from `const` C string (NUL terminated)
`String(char const *that, UInt len)`	Construct from fixed length C string
`String(char *that)`	Construct from non-`const` C string (NUL terminated)
`String(char val)`	Construct from C `char` (ASCII)
`String(int val)`	Construct from C `int` (ie, "`int` to `String`")

`String` methods: C++ `operator =`

This method is fairly straight forward in that it first Resets the `String`, and then copies the contents of the passed parameter into the `String` instance.

The return value is a reference to the `String` itself. Simple.

Each variant is briefly mentioned below...

Method	Description
`String &operator = ( String const &that )`	Replace contents of `this` with a copy of `that` OR replace `this`'s contents with `that` if `that.IsConsumable()`
`String &operator = ( char const *that )`	Replace contents of `this` with a `ConstLen String` pointing to `that`. **Note:** If `Strlen(that) >= 0xFFFF` (64K), it makes a type `Const String` instead and the caller will have to suffer the overhead of constantly using `Strlen` to determine the size – that is an unusual case anyway.
`String &operator = ( char *that )`	*** Same as above ***
`String &operator = ( char that )`	Replace contents of `this` with a single character `String` of type `ASCII1` or `UtfInline`, depending on `that`. **Note:** It turns out that it's highly uncommon for code that was written with **`BEFE`** in mind would use this form except for cases like `String('X')`, or `String('\n')` et cetera. As such, it's highly unlikely the resulting String will be of type `UtfInline`.
`String &operator = ( Char that )`	Replace contents of `this` with a single character `String` of type `ASCII1` or `UtfInline`, depending on `that`. **Note:** It is more likely for the resulting `String` to be type `UtfInline` in this case – especially if `that` is something specified in a form like `Char(0x130B8u)`. **Note:** It is much more common to use `that` in the form `String("♥")`, or simply `"♥"` which automatically gets cast to a `String`, instead of `Char(0x2665u)` which isn't visually obvious.

Method	Description
`String &operator = ( Int that )`	Replace contents of `this` with a `String` representation of signed integer `that`.
`String &operator = ( UInt that )`	Replace contents of `this` with a `String` representation of signed integer `that`.

`String` methods: C++ `operator +` and `operator +=`

These methods append various forms of Character Strings on the end of an `String` instances.

The `operator +=` methods append "in place" and *return a reference* to the destination `String` (and so are not `const` methods), while the `operator +` method *return a copy* of the original `String` with a copy of the specified `String` appended at the end (and so are `const` methods) .

Note: This is the standard **`BEFE`** approach for implementing C++ operators.

Method
`String &operator += (String const &that)`
`String  operator +  (String const &that) const`
`String &operator += (Char that         )`
`String  operator +  (Char that         ) const`
`String &operator += (char const *that  )`
`String  operator +  (char const *that  ) const`
`String &operator += (char *that        )`
`String  operator +  (char *that        ) const`
`String &operator += (char that         )`
`String  operator +  (char that         ) const`
`String &operator += (Int  that         )`
`String  operator +  (Int  that         ) const`
`String &operator += (UInt that         )`
`String  operator +  (UInt that         ) const`

Note: Should an error occur, the `String` will not be modified. That's the problem with the C++ way of doing things... You can only find if the operation failed by catching an exception.

For that reason we highly recommend using the `String Append` methods instead because they return a proper `Status` indicator.

`String` methods: C++ `operator *` and `operator *=`

These methods adjust a `String` instance's `mult` member, which determines the "number of repeating occurrences" of the `String`.

This odd bit of behaviour comes in extremely handy when the `String` is displayed as a monofont and used for spacing, line separators et cetera.

> **Note:** The `mult` member is a `UInt8` value, meaning it can be in the range `0..255`. The special case of Zero means the `String`, regardless of its actual Character contents, will not be displayed by **BEFE**.

The `operator *=` methods adjust `mult` "in place" and return a reference to the destination `String`, while the `operator *` method returns a modified copy of the original `String`.

> **Note:** This is the standard **BEFE** approach for implementing C++ operators.

The `Int` versions of these methods take the absolute value of the specified `mult` before applying it to the `String`.

All versions of these methods truncate the `mult` value to `255` if it is larger than `255`.

A **Null** `mult` value is interpreted as meaning "set to default", so is equivalent to specifying a `mult` of 1.

Method	Description
`String &operator *= ( UInt mult)`	Set a `String` instance's `mult` member value
`String operator * ( UInt mult) const`	Create a repeated copy of a `String`
`String &operator *= ( Int mult)`	Set a `String` instance's `mult` member value
`String operator * (Int mult) const;`	Create a repeated copy of a `String`

The `operator *=` methods DO NOT multiply the `String`'s current `mult` value by the specified parameter. They SET the `mult` value. So, let's say `String X` currently has a `mult` value of `2`, the expression "`X *= 5`" will set `X.mult` to the value `5` and NOT the value `10`.

Should an error occur, the `String` will not be modified.

> **Note:** That's the problem with the C++ way of doing things... You can only find if the operation failed by catching an exception.

`String` methods: C++ `operator ==` and related compares

These methods implement the various C++ comparison methods on `String` instances and return a `Boolean` value indicating the "truth" of the comparison.

> **Note:** For clearly sane reasons, and to play along with C++'s self-imposed ignorance of **Null**, we restrict the returned `Boolean` to be either `true` or `false` even though we do implement a **Null** value in our **`BEFE`** `Boolean` values. What this means, however, is that a **Null String** taking part in a comparison ALWAYS returns false as the result of a comparison. This leads to counter-intuitive cases where `string1 == string2` returns `false` AND `string1 != string2` returns `false`. So, you could say that **Null** breaks formal logic, as it should!!!

Each of these methods will perform a *case insensitive* compare if EITHER `String` (`this` or `that`) are set to an `Insensitive` state, and if BOTH are Sensitive, the operation will perform a *case sensitive* compare...

Method
`Boolean operator == (char const *that)   const`
`Boolean operator == (char *that)         const`
`Boolean operator == (String const &that) const`
`Boolean operator == (String const &that) const`
`Boolean operator != (char const *that)   const`
`Boolean operator != (char *that)         const`
`Boolean operator != (String const &that) const`
`Boolean operator <  (char const *that)   const`
`Boolean operator <  (char *that)         const`
`Boolean operator <  (String const &that) const`
`Boolean operator <= (char const *that)   const`
`Boolean operator <= (char *that)         const`
`Boolean operator <= (String const &that) const`
`Boolean operator >  (char const *that)   const`
`Boolean operator >  (char *that)         const`
`Boolean operator >  (String const &that) const`
`Boolean operator >= (char const *that)   const`
`Boolean operator >= (char *that)         const`

Method
`Boolean operator >= (String const &that) const`

`String` methods: C++ `operator <<=` and `operator <<`

The `operator <<` and `operator <<=` methods both concatenate String representations of various **`BEFE` Values** to an existing `String` value. In that respect, they're similar in function to C++ stream `operator <<`.

Both forms of these methods append to the end of the `String` and return a reference to the same `String` for use in subsequent streaming `operator <<` operations if required.

Method
`String &operator <<= (char  const *str)`
`String &operator <<= (char  const *str)`
`String &operator <<= (char *str)`
`String &operator <<= (void *ptr)`
`String &operator <<= (bool  b)`
`String &operator <<= (char  c)`
`String &operator <<= (Int   I)`
`String &operator <<= (UInt  I)`
`String &operator <<  (char  const *str)`
`String &operator <<  (char *str)`
`String &operator <<  (void *ptr)`
`String &operator <<  (bool  b)`
`String &operator <<  (char  c)`
`String &operator <<  (Int   I)`
`String &operator <<  (UInt  I)`
`String &operator <<  (Char  c)`
`String &operator <<  (String const &str)`

Note: In case you're wondering about why we implemented `operator <<=` on `String` since they're the same as `operator <<`... it's for the simple reason that we didn't want other cases (like bit shifts) to get confused in C++ with the operation on a **`BEFE`** `String`.

Take, for example, the expression "`X <<= 1`". Unless you knew that

X was a String you might read that as “shift X left 1 bit”. So, we decided to implement the <<= operators instead without expecting them to be used a whole lot by any self-respecting developer.

String methods: *Operator <type>*

These minor String methods are used in the odd cases where you wish to cast a String into another primitive type.

The `bool` form means the equivalent to "Is this `String` **Empty**?", where the `int` form is a quick-and-dirty way of turning a `String` into an `Int` if it's valid – sort of like a shorthand for using the standard C `atoi` function.

Method
`operator bool()`
`operator int()`

BEFE Style Methods

This section covers "proper" **BEFE** style methods and, in general, return a `Status` value indicating success or failure.

> **Note:** The odd method or two does return a `Boolean` but, in those cases, we guarantee "success" by ensuring that in unlikely case of internal "failure" we return a `Boolean` value that is "safer" than the alternative.
>
> Other methods may return a `String` or a `String` **Reference**, but those can be put down to "ease of use in C++" instead of a fundamental part of the **BEFE** design. This whole issue is covered in a separate Grey Paper: **BEFE Calling Sequences**
>
> So, in general, when you think of **BEFE** calling sequences, the first bet is it returns a `Status` value.

The vast majority of the `String` class is written in a **BEFE** style whereas the C++ operator methods are typically simple wrappers around their **BEFE** counterparts.

`String` methods: `BEFE Lifecycle`

The **`BEFE` Lifecycle** methods are responsible for maintaining a `String` instance as a "proper" **`BEFE` Value**.

The entire set of **`BEFE`** *Lifecycle* methods are discussed in detail in a separate Grey Paper: **`BEFE` Lifecycle**, but for clarity's sake are summarised here along with the rest of the `String` methods...

Method	Description
`Status StartUp()`	Initialise uninitialised instance
`Status ShutDown()`	Finish initialised instance
`Status Reset()`	Reset (Nullify) the String
`Status Clear()`	Clear (Empty) the String
`Boolean IsNull() const`	"Is This `String` **Null**?"
`Status SetNull()`	Set this `String` to **Null**
`Boolean IsEmpty() const`	"Is this `String` **Empty**?"
`Status SetEmpty()`	Set this `String` to **Empty** (clear contents)
`Status MoveFrom( String const &that)`	Move contents from another `String`
`Status CopyFrom( String const &that)`	Copy contents from another `String`, move if **Consumable**
`String &Consumable() const`	Set this `String` to **Consumable**
`String &NotConsumable() const`	Set this `String` to be **Not Consumable**
`Boolean IsConsumable() const`	"Is this `String` **Consumable**?"
`UInt Length() const`	Get number of *Characters* in the `String`
`UInt Size() const`	Get number of *Bytes* in the `String`

`String` methods: Type `Morphing`

There are two morphing methods not commonly used but are described here for completeness sake.

From time to time, **BEFE** wishes treat a `String` is if it were a `Bytes` **Array** instance. So, this method is used to discard the **`String`** specific nature of the `String` contents, and treat them as a `Bytes` instance instead.

```
Bytes    MorphToBytes();
```

Once this method has been called, all the methods available to a `Bytes` Array are available.

> **Note:** It is important, however, to remember that while one gains the ability to treat it as a `Bytes` instance, `String` specific attributes are lost in the conversion. So, a call to `String.MorphToBytes` followed by `Bytes.MorphToString` loses information.

The other uncommonly used method is `MakeBuffer` which forces the String to be a mutable String type `Buffer`.

> **Note:** When this is called, it may need to take a copy of the original `String` contents (in case it was of `String` type `Const`, `ConstLen`, etc.) and allocate a new `Buffer` instance for it.

`String` methods: State Query

These methods either passively return various information about the *state* of a `String` instance or change the instance's behaviour but do NOT alter the instance's *contents*...

Method	Description
`Boolean IsASCII() const`	"Is this `String` only ASCII Characters?"
`Boolean IsUTF8() const`	"Does this `String` have any non-ASCII Characters?"
`Boolean IsUnicode() const`	*** Same as above ***
`Boolean IsSensitive() const`	"Does this `String` compare case sensitive?"
`Boolean IsInsensitive() const`	"Is this `String` compare case insensitive?"
`String &Sensitivity( Boolean sense) const`	Turn this String's comparison case sensitivity on/off (`true` = On, `false` = Off)
`String &Sensitive() const`	Turn this `String`'s comparison case sensitivity on
`String &Insensitive() const`	Turn this `String`'s comparison case sensitivity off
`UInt GetType() const`	Get this `String`'s type (see `String.h`)
`UInt Multipler() const`	Get this `String`'s occurrence multiple
`UInt32 Hash() const`	Get this `String`'s Hash value (based on sensitivity)
`UInt32 HashSensitive() const`	Get this `String`'s case sensitive `Hash` value
`UInt32 HasnInsensitive() const`	Get this `String`'s case insensitive `Hash` value

`String` methods: `Comparison`

The String comparison methods come in two flavours, ones that return an Int to determine “relative order”, and ones that return a Boolean to determine if their contents are “equal” or “the same”.

Comparison of order and sameness depends on whether you wish to take case sensitivity into account.

For example, a String containing “aardvark” is “equal” to a String containing “Aardvark” if you DO NOT consider case but they are “not equal” if you DO consider case. This also means that a String containing “b” is “less than” a String containing “X” if you DO consider case but what about if you DO NOT consider case? That's where it can become a bit counter-intuitive...

You might mean two different things by asking these comparison questions... the first meaning could be “Did I type the same letters to produce the Strings?”, where the second meaning could be “Would I find them in the same place in the dictionary?” So, case sensitivity is an important point to be taken into account here.

> **Note:** A lot of people don't really think about the fact that their friend, the dictionary, does NOT take character case into consideration when ordering words. This is clearly because the word's position in the dictionary would then depend on type and/or usage... so the relative order in a dictionary would depend on if it was a proper noun or even if it started a sentence or not.
>
> So, the English speakers can praise and thank Robert Cawdrey (not Samuel Johnson) for their first alphabetical dictionary, and Americans can despise and curse Noah Webster for theirs.

So, what the BEFE String comparison mechanisms do is to provide you with the ability to specify which meaning you really want when comparing two String instances.

The default comparison is case insensitive if EITHER of the two String instances are flagged as case insensitive. But you an determine an exact comparison by not using the default comparison.

> **Note:** This is an extremely handy bit of behaviour in the case of our `String` instances because it allows the C++ `operator ==` and friends to behave according to how the instances are flagged instead of having to use a separate procedure to do case insensitive compares!!!

The `Int` methods are used to compare two `String` instances and determine their relative order. If `this` is “before” `that`, `-1` is returned, if `this` is “at the same place” as `that`, `0` is returned, and if `this` is “after” `that`, `+1` is returned.

The `Boolean` methods are used to see if two `String` instance's contents are “equal” or “in the same place”. If the two are “equal”, `true` is returned, otherwise `false` is returned.

These methods are summarised as follows...

Method	Description

`Int Compare(`
`  String const &that`
`) const`
	Compare `this` with `that` using default case sensitivity and return relative order

`Int Compare(`
`  String const &that,`
`  Boolean  sensitive`
`) const`
	Compare `this` with `that` using specified case sensitivity and return relative order
	(`false` = insensitive, `true` = sensitive)

`Int CompareSensitive(`
`  String const &that`
`) const`
	Compare `this` with `that`, case sensitive, and return relative order

`Int CompareInsensitive(`
`  String const &that`
`) const`
	Compare `this` with `that`, case insensitive, and return relative order

`Boolean IsEqual(`
`  String const &that`
`) const`
	“Is `this` 'equal to' or 'the same as' `that`?” using default case sensitivity
	(***Provided for **nmemonic** purposes only***)

`Boolean CompareEquals(`
`  String const &that`
`) const`
	“Is `this` 'equal to' or 'the same as' `that`?” using default case sensitivity

`Boolean CompareEquals(`
`  String const &that,`
`  Boolean sensitive`
`) const`
	“Is `this` 'equal to' or 'the same as' `that`?” using specified case sensitivity
	(`false` = insensitive, `true` = sensitive)

`Boolean CompareEqualsSensitive(`
`  String const &that`
`) const`
	“Is `this` 'equal to' or 'the same as' `that`?” using case sensitivity

`Boolean CompareEqualsInsensitive(`
`  String const &that`
`) const`
	“Is `this` 'equal to' or 'the same as' `that`?” using case insensitivity

String methods: *Byte Formatting*

These methods are used to format a `String` instance into bytes (in C string format) at in a given memory location.

> **Note:** These methods are a bit like the standard C `itoa` function except that they are given a buffer size which tells them the maximum number of bytes to output.

Each of these methods will, if possible, `NUL` (`0x00`) terminate the bytes if there is enough room in the buffer. If there is not enough room in the buffer, a non-zero Status will be returned.

Method	Description

`Status ToBytes(`
`  Byte *buf,`
`  UInt  maxLen,`
`  UInt &usedLen`
`) const`
	Convert contents to a C string (`NUL` terminated)

`Status ToEllipsis(`
`  Byte *buf,`
`  UInt  maxLen`
`) const`
	Convert contents to a C string (`NUL` terminated) indicating number of bytes used with trailing "`...`" if too long to fit

`Status ToEllipsis(`
`  Byte *buf,`
`  UInt  maxLen,`
`  UInt &usedLen`
`) const`
	Convert contents to a C string (`NUL` terminated) indicating number of bytes used with trailing "`...`" if too long to fit

> **Note:** The `usedLen` parameter, if given, will be updated to contain the number of bytes output into the buffer – excluding the `NUL`. So, it will contain the equivalent of `strlen(buf)` – unless it was too long to fit the `NUL` at the end.
>
> With this in mind, we recommend you pass one less byte than the real buffer size in the `maxLen` parameter, and to always put a `NUL` in the last byte of the buffer to ensure the resulting C string is guaranteed to be `NUL` terminated. For example...

```
Status status;
Byte   myBuf[10];
String myString;
UInt   usedLen;
...
myBuf[sizeof(myBuf)-1] = 0;
myString.ToEllipsis(myBuf,sizeof(myBuf)-1);
```

`String` methods: Content Query

These methods are used to get a copy (partial or complete) of a String instance's contents .

The first form of methods is the recommended form, and is passed the location to store the `String` portion and returns a `Status` value...

Method	Description

`Status Get(`
`  Int  index,`
`  Char &c`
`) const`
	Retrieve the character at the specified position in the `String` instance.

`Status Get(`
`  Span const &span,`
`  String  &outSpan`
`) const`
	Retrieve a portion, specified as a `Span`, of the `String` instance.

`Status Get(`
`  Range const &range,`
`  String  &outRange`
`) const`
	Retrieve a portion, specified as a `Range`, of the `String` instance.

`Status Get(`
`  Slice const &slice,`
`  String  &outSlice`
`) const`
	Retrieve a portion, specified as a `Slice`, of the `String` instance.

The second form of method, not recommended for use in “critical code”, returns the indicated portion of the `String` as a return value...

Method	Description
`Char Get()` `  const`	Retrieve the first character in `String` instance – or **Null** if none
`Char Get(Int index)` `  const`	Retrieve a portion, specified as a `Span`, of the `String` instance.
`String Get(Span const &span)` `  const`	Retrieve a portion, specified as a `Range`, of the `String` instance.
`String Get(Range const &range)` `  const`	Retrieve a portion, specified as a `Slice`, of the `String` instance.
`String Get(Slice const &slice)` `  const`	Retrieve a portion, specified as a `Span`, of the `String` instance.

`String` methods: `Append`

These methods append various forms of character and string data to the end of a `String` instance...

Method	Description

`Status Append(`
`  Char thechar`
`)`
	Append a single `Char` to a `String` instance

`Status Append(`
`  char thechar`
`)`
	Append a single C/C++ `char` to a `String` instance

`Status Append(`
`  Byte *buf,`
`  UInt  size`
`)`
	Append a specified number of bytes to a `String` instance

`Status Append(`
`  String const& that`
`)`
	Append a copy of a `String` instance's contents to a `String` instance

`Status Append(`
`  void const *that,`
`  UInt  thatSize`
`)`
	Append a specified number of bytes to a `String` instance

`Status Append(`
`  char const *that`
`)`
	Append a C string (`NUL` terminated) to a `String` instance

`Status Append(`
`  char *that`
`)`
	Append a C string (`NUL` terminated) to a `String` instance

`String` methods: `Insert`

These methods insert various forms of character and string data to a given position in a `String` instance...

Method	Description

`Status Insert(`
`  Int  index,`
`  Char that`
`)`
	Insert a single `Char` into a `String` instance

`Status Insert(`
`  Int  index,`
`  char that`
`)`
	Insert a single C/C++ `char` into a `String` instance

`Status Insert(`
`  Int  index,`
`  String const &that)`
	Insert a copy of a `String` instance's contents into a `String` instance

`String` methods: `Remove`

These methods remove a range of *contiguous* characters in a `String` instance...

Method	Description
`Status Remove(Int index)`	Remove a single character from a String instance
`Status Remove(Range const &range)`	Remove a given Range (inclusive) of characters from a `String` instance
`Status Remove(Span  const &span)`	Remove a given Span (exclusive) of characters from a `String` instance

Note: The indexes indicated by the `index`, `range`, and `span` parameters may be positive or negative. Negative indexes are adjusted by adding the `String` instance's length to them.

Note: Index relative order (`idx1` versus `idx2`) in the `Range` and `Span` parameters do NOT effect the order in which the single characters are removed, but only effect the identified range of characters. So, for example, a `Span '0:10'` means "the first 10 characters" or "characters `0..9` inclusive", whereas a `Span '10:0'` means characters in positions `10..1` inclusive. Subtle but important difference!

Note: We have intentionally NOT implemented a `Slice` version of this method because that might not indicate a set of *contiguous* characters. Quite honestly, if you want to do that when write a bloody loop – or shout real loud to convince us to throw that in the next version!!!

`String` methods: `Replace`

These methods replace a range of *contiguous* characters in a `String` instance with various forms of character and string data.

You could accomplish the same thing by calling `Remove` followed by `Insert` (see notes above) but these methods allow you do it in one go without incurring additional unneeded memory reallocations...

Method	Description

`Status Replace(`
`  Int  index,`
`  String const &that`
`)`
	Replace a single character at the specified `index` with a copy of another `String` instance's contents.

`Status Replace(`
`  Int  index,`
`  Char that)`
	Replace a single character at the specified `index` with a given *Unicode Character.*

`Status Replace(`
`  Int  index,`
`  char that)`
	Replace a single character at the specified `index` with a given C character – we don't advise non-ASCII values here.

`Status Replace(`
`  Range  const &range,`
`  String const &that`
`)`
	Replace a `Range` of characters with a copy of another `String` instance's contents.

`Status Replace(`
`  Range const &range,`
`  Char  that`
`)`
	Replace a `Range` of characters with a given *Unicode Character.*

`Status Replace(`
`  Range const &range,`
`  char  that`
`)`
	Replace a `Range` of characters with a given C character – we don't advise non-ASCII values here.

`Status Replace(`
`  Span  const &span,`
`  String const &that`
`)`
	Replace a `Span` of characters with a copy of another `String` instance's contents.

`Status Replace(`
`  Span const &span,`
`  Char  that`
`)`
	Replace a `Span` of characters with a given *Unicode Character.*

`Status Replace(`
`  Span const &span,`
`  char  that`
`)`
	Replace a `Span` of characters with a given C character – we don't advise non-ASCII values here.

`String` methods: `Find`

These methods help you to search a `String` instance's contents for occurrences of various kinds of character data – single character and multiple characters.

Note: Since the character positions in a `Range` and a `Span` are specified as `Int` values, we also stick with `Int` to indicate a specific character *position* or *index*. These `Int` *index* values, since they are signed, will have the `String` instance's `Length` added to them before they are used -- so they are considered to mean "before the end" if they are negative.

To remain consistent with the input parameter type `Int`, some of these methods return an `Int` indicating where the specified character(s) was/were located. However...

If the character(s) is/are not found, we return a `Null` position and NOT a value of `-1`.

Important: Just keep that in mind when using these methods.

The various groupings of `Find` method are described separately below based on what type of data (e.g., `Char`, `String` et cetera) is being searched for...

The first set of `Find` methods search for a given `Char` in a specified range of character positions in a `String`...

Method	Description

`Status Find(`
`  Char  c,`
`  Range const &range,`
`  Int  &foundPos`
`) const`
	Find a given *Unicode Character* in a specified `Range` in the `String` instance – return the position found in the `foundPos` output parameter.

`Status Find(`
`  Char  c,`
`  Span const &span,`
`  Int  &foundPos`
`) const`
	Find a given *Unicode Character* in a specified `Span` in the `String` instance – return the position found in the `foundPos` output parameter.

`Status Find(`
`  Char c,`
`  Int &foundPos`
`) const`
	Find a given *Unicode Character* in a specified *index* position in the `String` instance – return the position found in the `foundPos` output parameter.

`Int Find(`
`  Char  c,`
`  Range const &range`
`) const`
	Find a given *Unicode Character* in a specified `Range` in the `String` instance – return the position found as the method's *return result*.

`Int Find(`
`  Char  c,`
`  Span const &span`
`) const`
	Find a given *Unicode Character* in a specified `Span` in the `String` instance – return the position found as the method's *return result*.

`Int Find(`
`  Char c`
`) const`
	Find a given *Unicode Character* in a specified *index* position in the `String` instance – return the position found as the method's *return result*.

The second set of Find methods, instead of searching for a given Char , search for a C char in a specified range of character positions in a String.

Note: We had to add these Find(char...) methods because both String and Char have a C++ char constructor, so if you do something like "...myString.Find('a')..." in your code, the compiler will tell you it is ambiguous because it can't decide between two options...

```
String.Find(Char...)
      or
String.Find(String const &...)
```

And that code snippet shows such a common thing you would want to do, we had to provide these extra six methods to let you get away with it. So, a few extra methods because of a basic and frustrating flaw in the C++ language. It isn't the first time and it won't be the last!!!

Method	Description

`Status Find(`
`  char  c,`
`  Range const &range,`
`  Int  &foundPos`
`) const`
	Find a given C char in a specified Range in the String instance – return the position found in the foundPos output parameter.

`Status Find(`
`  char  c,`
`  Span const &span,`
`  Int  &foundPos`
`) const`
	Find a given C char in a specified Span in the String instance – return the position found in the foundPos output parameter.

`Status Find(`
`  char  c,`
`  Int  &foundPos`
`) const`
	Find a given C char in a specified *index* position in the String instance – return the position found in the foundPos output parameter.

`Int Find(`
`  char  c,`
`  Range const &range`
`) const`
	Find a given *C* char in a specified Range in the String instance – return the position found as the method's *return result*.

`Int Find(`
`  char  c,`
`  Span const &span`
`) const`
	Find a given C char in a specified Span in the String instance – return the position found as the method's *return result*.

`Int Find(`
`  char c`
`) const`
	Find a given C char in a specified *index* position in the String instance – return the position found as the method's *return result*.

The third set of `Find` methods search for a given `String` in a specified range of character positions in the `String` instance.

Note: This method uses same comparison semantics as the rest of the `String` methods. That is to say that if either `String` is flagged as **insensitive**, the comparison is **insensitive**.

Method	Description

`Status Find(`
`  String const &string,`
`  Range  const &range,`
`  Int          &foundPos`
`) const`
	Find a given `String` in a specified `Range` in the `String` instance – return the position found in the `foundPos` output parameter.

`Status Find(`
`  String const &string,`
`  Span const   &span,`
`  Int          &foundPos`
`) const`
	Find a given `String` in a specified `Span` in the `String` instance – return the position found in the `foundPos` output parameter.

`Status Find(`
`  String const &string,`
`  Int          &foundPos`
`) const`
	Find a given `String` in a specified *index* position in the `String` instance – return the position found in the `foundPos` output parameter.

`Int Find(`
`  String const &string,`
`  Range  const &range`
`) const`
	Find a given `String` in a specified `Range` in the `String` instance – return the position found as the method's *return result*.

`Int Find(`
`  String const &string,`
`  Span   const &span`
`) const`
	Find a given `String` in a specified `Span` in the `String` instance – return the position found as the method's *return result*.

`Int Find(`
`  String const &string`
`) const`
	Find a given `String` in a specified *index* position in the `String` instance – return the position found as the method's *return result*.

The final set of Find-like methods search for a specific *positional condition* in a String...

Method	Description
Int FirstNonBlank() const	Find position of first non-blank (not 0x20) character in the String.
Int LastNonBlank() const	Find position of last non-blank (not 0x20) character in the String.
Int FirstNonWhite() const	Find position of first non-white (see below) character in the String.
Int FirstNonWhite() const	Find position of first non-white (see below) character in the String.

Note: By "non-white" we mean not **white space** – which we take to mean any Unicode character greater than 0x20.

This means ASCII Control characters like '\n' and '\r' are considered **white space** by these methods.

That may seem a bit over-extending to you in the "normal" sense of *white space* but, quite honestly, if that's not what you want then don't use these methods!!!

`String` methods: Unicode Case Folding

The following methods perform **Unicode Case Folding** operations. Although the *Unicode Consortium* calls this "case folding", users of western European Latin based alphabets typically think of it as "uppercase" and "lowercase" but that's a bit of a long story...

Unicode Case Folding specifications are for turning a *Unicode Character* into what we think of as lowercase – but they do NOT have the reverse specification for converting lowercase to uppercase – that's an odd bit of missing specification that we can't figure out.

But we figured that, given their specification file (see `BEFE_CaseFolding.txt` in the `BEFE_Home` directory), we could easily reverse the order with a couple of exceptions and by applying some well behaved interpretation of their input file. So what we do is take the FIRST occurrence of any folding target and using that one for the reverse operation (see the file to get a better understanding).

> **Note:** There's really not enough room to get into this a whole lot more here so we will describe *Unicode Case Folding* in a separate *Grey Paper* at some point. But we WILL say that if you leave it alone, the standard **`BEFE`** configuration will produce proper uppercase to lowercase conversions and vice versa properly across most languages in use today.

The `String` *Unicode Case Folding* methods are as follows...

Method	Description
`String &ToLowercase() const`	Convert, in-place, all characters in a `String` instance to lowercase (e.g. "fold it")
`String &ToUppercase() const`	Convert, in-place, all characters in a `String` instance to uppercase (e.g. "unfold it")
`String  Lowercase()`	Convert, out-of-place, all characters in a `String` instance to lowercase (e.g. "make a lowercase copy and fold it")
`String  Uppercase()`	Convert, out-of-place, all characters in a `String` instance to lowercase (e.g. "make an uppercase copy and unfold it")
`Status  Fold(Boolean doFold)`	Fold or unfold, in-place, all characters in a `String` instance (`true` = Fold, `false` = Unfold)

`String` methods: Content manipulation

We have provided various groups of String content manipulation methods.

Each of these method groups are describe separately below.

Trimming...

These methods are useful for removing whitespace from a `String` instance.

Once again, by "whitespace" we mean **`BEFE`** *Whitespace*, which is any characters less than or equal to `0x20`, so this means *ASCII Control Characters* as well. This is exceedingly handy getting rid of the bothersome `'\r'` that Microsoft Windows applications like putting on the end of text file lines...

Method	Description
`String &Trim()`	Remove all leading and trailing whitespace from the `String` instance
`String &LeftTrim()`	Remove all leading whitespace from the `String` instance.
`String &RightTrim()`	Remove all leading whitespace from the `String` instance.

Truncating...

These methods are helpful when you wish to force a `String` instance to be no MORE than a specific number of characters.

> **Note:** If the `String` instance is already less than the specified number of characters then it will not be modified.

Method	Description
`String &Truncate()`	Remove all leading and trailing whitespace from the `String` instance
`String &Truncate(UInt len)`	Remove all leading whitespace from the `String` instance.

Padding...

These methods are helpful when you wish to force a `String` instance to be no LESS than a specific number of characters.

> **Note:** If the `String` instance is already mopre than the specified number of characters then it will not be modified.

Method	Description
`String &Pad(UInt len)`	Pad a `String` instance with trailing `' '` until it is at least `len` characters in length

`String &Pad(`
`  UInt len,`
`  Char padChar)`
	Pad a `String` instance with trailing `padChar` until it is at least `len` characters in length

`String &RightPad(UInt len)`	Pad a `String` instance with trailing `' '` until it is at least `len` characters in length

`String &RightPad(`
`  UInt len,`
`  Char padChar)`
	Pad a `String` instance with trailing `padChar` until it is at least `len` characters in length

`String &LeftPad(UInt len)`	Pad a `String` instance with leading `' '` until it is at least `len` characters in length

`String &LeftPad(`
`  UInt len,`
`  Char padChar)`
	Pad a `String` instance with leading `padChar` until it is at least `len` characters in length

Ellipsis...

The following methods are used to create a new `String` containing an **ellipsis** ("**. . .**", three ASCII `'.'` characters) at the beginning, middle, or end of the source `String` instance if it exceeds a specified number of characters.

If the source String does NOT exceed the specified length, a copy of it is returned...

Method	Description
`String  Ellipsis(UInt len)`	Create a truncated copy of a `String` instance appending an `ellipsis` if it is longer than `len` characters, otherwise copy the `String` instance – same as `PostEllipsis` below
`String  PreEllipsis(UInt len)`	Create a truncated copy of a `String` instance prepending an `ellipsis` if it is longer than `len` characters, otherwise copy the `String` instance
`String  MidEllipsis(UInt len)`	Create a truncated copy of a `String` instance with an `ellipsis` embedded in the middle if it is longer than `len` characters, otherwise copy the `String` instance
`String  PostEllipsis(UInt len)`	Create a truncated copy of a `String` instance appending an `ellipsis` if it is longer than `len` characters, otherwise copy the `String` instance

Pad OR Ellipsis...

The following methods are used to create a new `String` of a specified number of characters and performing one of the following exclusive actions based on the `String` instance's length in characters...

- Create a copy of the `String` instance, truncating and optionally adding an **ellipsis**
- Padding a copy of the `String` instance with a specified character
- Creating a copy of the `String` instance if it is already the correct size.

Method	Description
`String  PadOrEllipsis(UInt len)`	Pad with `' '` or truncate with **ellipsis to** `len` characters

`String  PadOrEllipsis(`
`  UInt len,`
`  Char padChar)`
	Pad with `padChar` or truncate with **ellipsis** to `len` characters

`String &TruncateOrPad(UInt len)`	Pad with `' '` or truncate (no **ellipsis**) to `len` characters

`String &TruncateOrPad(`
`  UInt len`
`  Char padChar)`
	Pad with `padChar` or truncate (no **ellipsis**) to `len` characters

`String` methods: `Join`

These methods are used to create a new String instance which contains the join all the strings in a string **Array** (a `Strings` instance) with the source `String` (`this`) as a separator.

> **Note:** This works exactly the same way as Python's string `join` method – except that it produces a mutable `String` of course!!!

Method	Description

`Status Join(`
`  Strings const &strings,`
`  String &outString`
`) const`
	Join a `Strings` instance with the contents of `this` and place result in `outString`

`String Join(`
`  Strings const &strings`
`) const`
	Join a `Strings` instance with the contents of `this` and return the resulting `String` instance

Take, for example, the following code snippet...

```
Strings arr;
arr.Append("abc");
arr.Append("def");
arr.Append("ghi");
Cout << String('.').Join(arr) << '\n';
```

If you ran this code you would get the following `Console` output...

```
abc.def.ghi
```

`String` methods: Counting Characters

These methods search a `String` instance and returns the number of times a given `char`, `Char`, or `String` instance occur in it..

Method	Description
`UInt Count(Char c) const`	Count occurrences of `Char` in `this`
`UInt Count(char c) const`	Count occurrences of `char` in `this`
`UInt Count(String const &s) const`	Count occurrences of `String` in `this`

`String` methods: Splitting

The Split methods are used split a String instance into various forms of Arrays like Strings and and Spans.

As such, they come in various related grouping, each covered below.

Note: You may have notice below that we do provide a mechanism for splitting to `Spans` but NOT to `Ranges`. This is because the concept of an **Empty** `Range` is typically avoided in **`BEFE`** since a `Range` is **inclusive**. As such, the concept of an **Empty** `Range` becomes weird, because what does it INCLUDE then? So it's dangerous thinking to implement the concept of an **Empty** `Range` because you quickly get onto shaky ground there!!!

Note: Each grouping of Split method below creates a new `Array` (`Strings` or `Spans`). We explicitly mention this here to point out that they act differently to the various **`BEFE`** `ToStrings` methods you may have encountered which append to the `Strings` array. But these methods `Clear` the output `Array` first.

`Split` methods

These methods split a `String` instance's contents into a `Strings` array instance.

> **Note:** These are the exact equivalent to the `SplitToStrings` methods but we'll duplicate the definitions here just because they exist AND are cleaner to read in your code...

Method	Description

`Strings Split(`
`  Char searchChar`
`) const`
	Split a `String` instance on a given `Char` returning a `Strings` array

`Strings Split(`
`  char searchChar`
`) const`
	Split a `String` instance on a given `char` returning a `Strings` array

`Strings Split(`
`  String const &searchString`
`) const`
	Split a `String` instance on a given `String` returning a `Strings` array

`Status  Split(`
`  Char  searchChar,`
`  Strings &splitArray`
`) const`
	Split a `String` instance on a given `Char` returning the result in the `splitArray` parameter

`Status  Split(`
`  char searchChar,`
`  Strings &splitArray`
`) const`
	Split a `String` instance on a given `char` returning the result in the `splitArray` parameter

`Status  Split(`
`  String const &searchString,`
`  Strings &splitArray`
`)`
	Split a `String` instance on a given `String` returning the result in the `splitArray` parameter

`SplitToStrings` methods

These methods split a `String` instance's contents into a `Strings` array instance...

Method	Description

`Strings SplitToStrings(`
`  Char searchChar`
`) const`
	Split a `String` instance on a given `Char` returning a `Strings` array

`Strings SplitToStrings(`
`  char searchChar`
`) const`
	Split a `String` instance on a given `char` returning a `Strings` array

`Strings SplitToStrings(`
`  String const &searchString`
`) const`
	Split a `String` instance on a given `String` returning a `Strings` array

`Status  SplitToStrings(`
`  Char  searchChar,`
`  Strings &splitArray`
`) const`
	Split a `String` instance on a given `Char` returning the result in the `splitArray` parameter

`Status  SplitToStrings(`
`  char searchChar,`
`  Strings &splitArray`
`) const`
	Split a `String` instance on a given `char` returning the result in the `splitArray` parameter

`Status  SplitToStrings(`
`  String const &searchString,`
`  Strings &splitArray`
`)`
	Split a `String` instance on a given `String` returning the result in the `splitArray` parameter

SplitToSpans methods

These methods split a `String` instance's contents into a `Spans` array instance. Each instance in the `Spans` array indicates a distinct `Span` of characters the original `String` instance. These `Spans` can be used to examine or extract parts of the `String` instance at your own leisure...

Method	Description

`Spans SplitToSpans(`
`  Char searchChar`
`) const`
	Split a `String` instance on a given `Char` returning a `Spans` array

`Spans SplitToSpans(`
`  char searchChar`
`) const`
	Split a `String` instance on a given `char` returning a `Spans` array

`Spans SplitToSpans(`
`  String const &searchString`
`) const`
	Split a `String` instance on a given `String` returning a `Spans` array

`Status SplitToSpans(`
`  Char  searchChar,`
`  Spans &splitArray`
`) const`
	Split a `String` instance on a given `Char` returning the result in the `splitArray` parameter

`Status SplitToSpans(`
`  char  searchChar,`
`  Spans &splitArray`
`) const`
	Split a `String` instance on a given `char` returning the result in the `splitArray` parameter

`Status SplitToSpans(`
`  String const &searchString,`
`  Spans  &splitArray`
`) const`
	Split a `String` instance on a given `String` returning the result in the `splitArray` parameter

`String` methods: Escape Character Manipulation

These methods are simply String method wrappers around the general purpose BEFE Procedures Escape and Unescape described in more detail later in this paper...

Method	Description
`String Escape() const`	Turn a UTF-8 form of `String` instance into an *externalised escape* (e.g. C, Java, Python, etc.) form

`Status Escape(`
`  String &outStr`
`) const`
	Turn a UTF-8 form of `String` instance into an *externalised escape* (e.g. C, Java, Python, etc.) form

`String Unescape() const`	Turn an *externalised escape* form of `String` instance (e.g. C, Java, Python, etc.) into a UTF-8 form of `String` instance

`Status Unescape(`
`  String &outStr`
`) const`
	Turn an *externalised escape* form of `String` instance (e.g. C, Java, Python, etc.) into a UTF-8 form of `String` instance

BEFE Procedures

This section describes various String related procedures that aren't String *Methods*. Instead, they are passed the String(s) in question as parameters and, hence, are called *Procedures* not instead.

Most of these have equivalent String *Methods* but are provided mainly for ease of use and readability purposes.

> **Note:** Those procedures marked as BEFE_INLINE are declared that way in C++ to enable inline code generation instead of separate calls for efficiency purposes.

String Null and Empty

The following BEFE_INLINE String procedures are made available for ease of use and checking your *Class's* members or your *local variables*...

Method	Description

`Boolean IsNull(`
`  String const &rThis`
`)`
	"Is String rThis **Null**?"

`Status SetNull(`
`  String &rThis`
`)`
	Set the String rThis to the **Null** state

`Boolean IsEmpty(`
`  String const &rThis`
`)`
	"Is String rThis **Empty**?"

`Status SetEmpty(`
`  String &rThis`
`)`
	Set the String rThis to the **Empty** state

String Substitution

The following `Substitute` **Procedure** may be used to substitute occurrences of **Substitution Variables** from a `NamedStrings` instance...

```
Status Substitute(
  String       const &inStr,     // IN:  Input String
  NamedStrings const &substVars, // IN:  Named Substitution Variables
  String       const &startTag,  // IN:  Start Tag (eg "%" "$(", etc)
  String       const &endTag,    // IN:  End Tag (eg "%", ")", etc)
  String             &outStr     // OUT: Resulting String
)
```

The `inStr` parameter contains the "input string". This String instance is not modified by this procedure.

The `substVars` is a `NamedStrings` map instance (a `StringMapValue<String>`) indicating the named *Substitution Variables* to look for and substitute – think of these as stuff like your OS environment variables and you won't be caught too far off kilter.

The `startTag` and `endTag` parameters are "tag" `String` insances indicating the start and end delimeters for substitution markers. Examples include "`%`" and "`%`" for Windows, maybe "$(" and "`)`" for command shells, and things like "`<`" and "`>`" if for simple C++ templates usages.

The `outStr` parameter is updated to contain the fully recursive string substitution.

> **Note:** The only likely error to be returned, other than OutOfMemory, is most likely due to recursion in your Substitution Variable values, like `A="%B%"` and `B="%A%"` etc.

If a non-zero error is returned, the `outStr` parameter will remain untouched.

This allows you to pass the same `String` in the `inStr` and `outStr` parameters with the understanding that `outStr` will only contain valid substitutions if `inStr` does not contain recursive *Substitution Variable* references.

> **Note:** Any `startTag` encountered without a matching `endTag` or any `startTag`/`endTag` occurrences encountered that are not in the `substVars` map, will be left untouched with the original `startTag` and `endTag` intact.

`String` Escape Character Manipulation

The final two procedures are used to do BEFE String Literal Escape Sequence processing.

These methods are already fairly well described earlier in this paper but they are declared as follows...

Method	Description

`Status Escape(`
`  String const &inStr,`
`  String &outStr`
`)`
	Convert a `String` instance to a **`BEFE`** *String Literal Escape Sequence* formatted String – e.g., turn `'\'` into `'\\'` et cetera

`Status Unescape(`
`  String const &inStr,`
`  String &outStr`
`)`
	Convert a **`BEFE`** *String Literal Escape Sequence* formatted String instance to a "proper" BEFE `String` instance – e.g., turn `'\\'` into `'\'` et cetera