Treating the "Character Type" as a "String Type" as of Sibelius Ultimate 2020.9

Bob Zawalich Sep. 16, 20

Sibelius has always treated single-digit/single-character variables differently than other variables, using a special and undocumented format called the **Character** type, and this has caused a number of problems over the years. These problems are discussed in detail in the appendix to this document.

Sibelius Ultimate 2020.9 has added the ability treat single-digit variables as **Strings**, which avoids the many problems caused by the **Character** type.

The **Character type as string** capability is only enabled in plugins that include special "opt-in" code in Initialize(). This is discussed in more detail in the document ***New Optional Plugin Features as of Sibelius Ultimate 2020.9***.Here is an example of the opt-in code that is added by Sibelius to plugins created using **New** in the **Plugin Editor**.

```
// The following enables the latest behavior for the ManuScript interpreter.
// If you intend your plugin to run on previous versions of Sibelius where that functionality
// didn"t exist, you will likely have to revisit the following:

if (Sibelius.ProgramVersion > 20200600) {
   SetInterpreterOption(TreatSingleCharacterAsString);
   SetInterpreterOption(SupportHalfSemitonePitchValues);
}
```

TreatSingleCharacterAsString is the feature that fixes buggy behavior involving single-digit literals. **SupportHalfSemitonePitchValues** is a feature that will allow plugins to create notes that are quartertones. That feature will be discussed in another document. These features can be activated independently.

If a plugin does not have the feature enabled, it will behave as it did prior to 2020.9. If your plugin calls other plugins and you have one of these features enabled, you need to be sure that it will behave the way you want it to.

This feature is an opt-in option for plugins because many existing plugins depend on the buggy behavior of the **Character** type, and if the feature were introduced, many plugins would no longer work. Before you enable this feature in an existing plugin, you must be sure that it does not depend on behavior that will change, and if your new plugin enables the feature, you must not use any code that depends on the earlier behavior. This can be subtle and tricky to figure out.

Finding and fixing undesirable behavior

The most common undesirable behavior is that quoted single-digit literals resolve to their ASCII values. Here are some examples of ASCII values:

```
Asc("2") = 50 Asc("3") = 51 Asc("22") = 50
Asc("a") = 97 Asc("Z") = 90 Asc("?") = 63
```

Note that "22" resolves to the same value as "2", because the Asc() command only handles a single digit.

In arithmetic expressions, both quoted single-character numeric and non-numeric literals are considered to be **Character** type variables, and are converted to their ASCII values. The expression

("2" + 3) resolves to 53, because Asc("2") is 50.
The expression ("a" + 22) = 119 because "a" resolves to its ASCII value 97.

Note this very subtle difference:

```
chChar = "b";
chString = "" & "b"; // converted to type String

chCharNum = "3";
chStringNum = "" & "3";

trace("chChar + 3 = " & (chChar + 3));
trace("chString + 3 = " & (chString + 3));

trace("chCharNum + 3 = " & (chCharNum + 3));
trace("chStringNum + 3 = " & (chStringNum + 3));

************************************************************
chChar + 3 = 101
chString + 3 = 3

chCharNum + 3 = 54
chStringNum + 3 = 6
```

When a non-numeric String variable is used in a arithmetic expression, its value is 0. When a non-numeric Character type variable is used in a arithmetic expression, is value is equivalent to **Asc(chChar**), so in the example above chChar resolves to 98, so 98 + 3 is 101. "3" resolves to 51.

This is generally not what you want. The simple rule of thumb is: **never use non-numeric values in an arithmetic expression. Also, never use numedric literals in quotes in an arithmetic expression.**

These problems, and many more besides, go away if the Character type is removed, and all such variables are treated as strings.

In many plugins, it is not obvious where a variable comes from and whether it happens to be a **Character** type. If you don't know it is wisest to convert the variable to a String

```
ch = ("" & ch);
```

so you know what you are getting.

There are problems using **Character** type variables in comparisons, and switch statements, as well as in arithmetic. Avoid something like

```
ch = "a";
if  (ch > 3)
```

If you really want to do comparisons other than equality tests with non-numeric single-digit variables, use Asc to get a numeric value for the variable.

There are a lot of places where comparisons rely on the conversion of a literal to its ascii value, and if you enable this feature for its many benefits, you must modify such code.

There are 3 examples of this problem in **utils.plg**.

In **IsNumeric()**, there is the code

```
ord = Asc(ch);
if ((ord < "0") or (ord > "9"))
```

where ch was produced by a call to **CharAt**, which returns a **Character** type variable. Suppose that the character ch were a 3. Asc("3") is 51. This is as it should be. We want to compare the variable as a number.

(ord < "0") compares 51 to 48, since "0" gets resolved to its ascii value of 48. If we get rid of the **Character** type, this will fail, because "0" will resolve to 0.

To fix this, rewrite the code to be

```
if ((ord < Asc("0")) or (ord > Asc("9")))
```

which will work correctly, whether or not **TreatSingleCharacterAsString** is enabled.

Similar code exists in **utils.UpperCase** and **utils.LowerCase**. In **utils.UpperCase** we have

```
ord = Asc(ch);
if ((ord >= "a") and (ord <= "z"))
```

and this should also be rewritten as

```
if ((ord >= Asc("a")) and (ord <= Asc("z")))
```

Most likely you will not have a lot of code that behaves this way, and activating this feature will only bring you joy. But if you are enabling it in a complex existing plugin, you should examine the code closely for places where such code may have snuck in, and root it out.

Remember that as **String** variables are converted to numbers, "a" converts to the number 0, and "9" converts to 9, which is as it should be.

If you read through the appendix, you will see why it is a good idea to enable this feature if possible.

Appendix: ManuScript bugs involving the Character type object in comparisons, switch statements, and arithmetic

There are long-standing bugs in ManuScript where comparisons, including **switch** statements, evaluate incorrectly when an operand contains a single digit or character. These are incredibly difficult to debug. It totally blew my mind when I finally grasped what was going on.

Here is the gist of the problem.

In ManuScript, besides the objects listed in the chart at the end of the Reference, there are **strings**, integers, floating point numbers, booleans, arrays, hashes, and an undocumented type called **Character**.

These are single-character entities which can be created by the method **CharAt**, or when you have a single-digit literal, either directly ("x") or as a variable assigned to a literal (ch = "x"). If you use **Substring**() to get a single character, its type is **String**, not **Character,** and it does not present the same kind of problems.

These are examples of **Characters**

ch = CharAt(str, 0);
cha = "a";
ch3 = 3;

Executive Summary

Character variables cause failures in **if statements, switch statements, and arithmetic expressions**. Avoid creating such variables if possible. To be sure **if** and **switch** statements will work, cast all operands to **String**s (prepend ""& to the variable). (e.g., cha = "" & "a").

Arithmetic expressions involving single digit literals are problematic. This is explained in more detail below.

Quoted single-digit numeric literals resolve to their ASCII values. Here are some examples of ASCII values:

Asc("2") = 50 Asc("3") = 51 Asc("22") = 50
Asc("a") = 97 Asc("Z") = 90 Asc("?") = 63

So unfortunately, the expression ("2" + 3) resolves to 53, because Asc("2") is 50.

You can fix this by converting "2" to a **String** before adding. If I use a method "tostr" to do that I get:
tostr("2") + 3 = 5

Arithmetic expressions involving single-character non-numeric literals (such as "a") are also a problem, as the also resolve to their Ascii values (Asc("a") = 97)

if you want them treated as **Strings** it is best to remove them from the expression, but, if cast to a **String**, they will resolve to 0 (zero). If you really want to use the ASCII value of the literal, use **Asc**(literal) instead.

("a" + 22) = 119 --- "a" resolves to its ASCII value 97
(tostr("a") + 22) = 22 --- String "a" resolves to 0.

Failures in If Statements

In an **if** statement, the right operand is always converted to the type of the left operand. If the ***left*** operand is a **Character**, then the comparison will **convert the *right* operand to a Character** as well, which will **truncate that operand to a single character**, even if it had been longer.

Here are examples of **if** statements that will resolve incorrectly:

```
if ("a" = "abc")
...
ch3 = 3;
if (ch3 = "3foo")
```

Integers are tricky, though. These examples will all return **False**. Apparently, if both operands are completely numeric, even if one of them is quoted, there is no conversion to **Character** type, and the normal numeric comparison is done. They must have caught that one early on.

```
if (ch3 = 345)
if (3 = "345")
if (3 = 345)
if ("3" = 345)
```

if (3 = "3foo") would return True, though, because "3foo" would be truncated to "3". You really can't rely on an **if statement** when the left operand is a single digit character or digit. With numeric operands they sometimes do the right thing, though.

Workarounds for if statements

In general, avoid **Character** variables. Use **Substring**() instead of **CharAt**(), or if you really want **CharAt**(), cast the result to a **String**. If you have a variable var1, use ("" & var1) when you want to use it. Be sure to parenthesize it to avoid problems with evaluation order. I sometimes create a method **tostr**() that does that very concatentation, and find **tostr(var1)** easier to read than **("" & var1)**. The result is the same though.

For global variables or array values you will be casting to a **String** anyway, since these return addresses, not values. That casting will prevent **Character** types.

If you get a variable and do not know its internal structure, just cast it to a String and you will not need to worry about it failing. Casting both operands of an *if* statement to Strings is fail-safe.

If you know that one of the arguments is a **Character** and the other is not, you can make the non-**Character** variable the left operand in the expression. The right operand will be cast to something that is not a **Character**, and nothing will be truncated. I find few situations where I can do this, though.

Failures in Switch statements

Switch statements can fail if the value in a **case** statement is a **Character**.
If you have a switch statement like this:

```
**********************************************
switch (myDynamic)
{
```

```
	case ("f")
	{
		Xoffset = "f";
	}
	case ("ff")
	{
		Xoffset = "ff";
	}
	case ("fff")
	{
		Xoffset = "fff";
	}
}

return (Xoffset);
```

---

the first **case**() has a **Character** type operand, and any string that starts with "f" will match that case..

If **myDynamic** is a set of strings like "f", "ff", "fff"..., **each of those strings will match the first case**. A switch is effectively a nested **if** structure where the value in the **switch** is the **right** operand, and the values in the **case** statements are the **left** operands. A **left operand of type Character will truncate the right operator** to a single character before comparing.

Workarounds for switch

The values for the case() statements cannot be Characters. Cast them to **String**s by concatenating "" & to the start of the values. You can cast the **switch** statements values as well, or not, but it might be simpler to cast both so you don't need to remember which one worked.

As with all these problems, forcing any arguments of **switch/case** statements to be **Strings** should result in correct code.

This is an example of a properly working **switch**. The method **tostr(ch)** casts the **Character** to a **String**. You could just add ""& to the start, but this seems a bit more readable to me.

```
switch (myDynamic)
{
case (tostr("f"))
{
	Xoffset = "f";
}

case (tostr("ff"))
{
	Xoffset = "ff";
}
case (tostr("fff"))
{
	Xoffset = "fff";
}
}
```

return (Xoffset);

Failures in Arithmetic

Arithmetic expressions involving single digit literals are problematic.

Quoted **single-digit numeric** literals **resolve to their ASCII values**. Here are some examples of ASCII values:

Asc("2") = 50 Asc("3") = 51 Asc("22") = 50
Asc("a") = 97 Asc("Z") = 90 Asc("?") = 63

Note that if the argument of **Asc** is more than 1 numeric digit, only the first digit is processed, thus **Asc**(22) = 50 just the same as **Asc**(2). It is mentioned in the Reference that **Asc** expects a single-digit, but the result is not specified.

The expression **("2" + 3)** resolves to **53**, because the ASCII value of ("2") is 50.

Quoted **multi-digit numeric** literals (which are **String**s), or single-digit quoted numeric literals that are cast to **String**s, **resolve to their correct numeric values**.

You can fix the example above by converting "2" to a **String** before adding. If I use a method "**tostr**" to cast to a **String** I get:

tostr("2") + 3 = 5.

Arithmetic expressions involving single-character **non-numeric** literals (such as "a") are also a problem, as they also resolve to their Ascii values (**Asc**("a") = 97)

if you want them treated as **String**s it is best to remove them from the expression, but, if cast to a **String**, they will resolve to 0 (zero). If you really want to use the ASCII value of the literal, use **Asc**(literal) instead.

("a" + 22) = 119 --- character "a" resolves to its ASCII value 97
(tostr("a") + 22) = 22 --- string "a" resolves to 0.

Before **Asc**() and **Chr**() were available, around the time of Sibelius 5, code fairly often took advantage of quoted single digit literals to access ASCII values. Here are some examples from **utils.plg**, which was post-**Asc**() but still uses ASCII single-digit literals, even though it also uses Asc() explicitly.

1. utils.IsNumeric contains these lines
   ```
   ord = Asc(ch);
   if ((ord < '0') or (ord > '9'))
   ```
   and this only works because the resolutions of '0' and 0 are different. This would work as well and be more explicit as
   ```
   ord = Asc(ch);
   if ((ord < Asc('0')) or (ord > Asc('9')))
   ```
2. utils.UpperCase contains these lines

```
c = CharAt(str, i);
ord = Asc(c);
if (ord >= 'a' and ord <= 'z') {
ord = ord - 32;
}
```

If the value of c were "b", ord would be 98, and this code only works because 'a' and 'z' are converted to their ASCII values. If they were **String**s, they would both resolve to 0. This code would also be more explicit, and not need to rely on undocumented behavior, as

```
if ((ord < Asc('a')) or (ord > Asc('z')))
```

More examples

Arithmetic – addition

Asc(2) = 50, Asc (3) = 51, Asc (22) = 50 (**Asc only processes the first digit of its argument**)

quote 2 no quote 3: '2' + 3 = 53
quote2 quote3: '2' + '3' = 101
no quote2 quote3: 2 + '3' = 53
no quote2 no quote3: 2 + 3 = 5
quote22 no quote3: '22' + 3 = 25
no quote3 quote22: 3 + '22' = 25
quote3 quote22: '3' + '22' = 73
no quote3 no quote22: 3 + 22 = 25
quote3 quote2 to int before adding : 51 + 50 = 101
quote3 quote2 to str - no longer Character before adding : '3' + '2' = 5

Arithmetic – multiplication

no quote3 no quote2: 3 * 2 = 6
quote3 quote2: '3' * '2' = 2550
no quote3 quote2: 3 * '2' = 150
quote3 no quote2: '3' * 2 = 102
quote3 no quote2 cast to str - no longer Character before adding : '3' * '2' = 6

Workarounds for arithmetic

I had expected that casting values of numeric literals to integers (0 + var1) would fix these, but it does not.

- For quoted single-digit **numeric** literals, cast them to **String**s, and **they will resolve to their true numeric value.**
- For quoted single-digit non-**numeric** literals, cast them to **String**s, and **they will resolve to 0**. If you want to use the ASCII value of the characters, use **Asc**(<literal>.).

You can cast all literals to **String**s, and know that you will get a correct numeric value for numeric literals, and 0 for literals that do not start with a numeric character. If you want Ascii values, cast using **Asc()** and understand that it will process only the first character in the operand.

For one final weirdness, note that if you have a **string** that has **leading numeric characters followed by non-numeric characters**, these resolve to the **normal numeric value of the leading numeric digits**, ignoring the non-numeric characters and any trailing numeric digits:

("3abc" + 3) = 6
("44abc4" + 3) = 47

I don't know what to say about this, except try to avoid using these if possible unless you really know why you are doing this.

It is kind of amazing to me that more bugs like this have not shown up. These are really hard to debug.

The plugin **Is Type Character** was used to generate these examples, and is available on request.