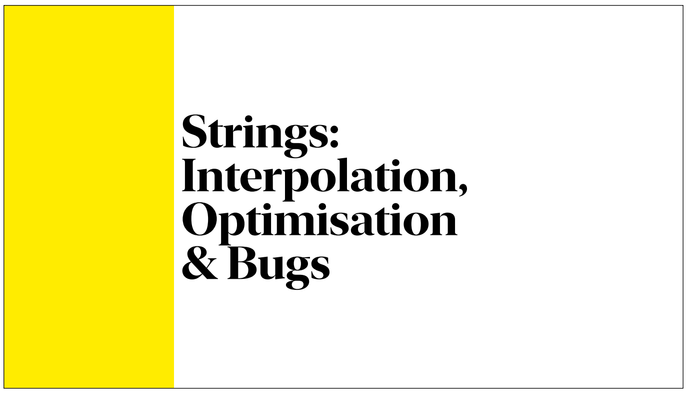


Strings: Interpolation, Optimisation & Bugs

みなさん、はいさい。 那覇はいいですね。
今日、本当は日本語で話たいですよ！最近勉強していますしね。
でも、残念ながら、まだそこまで上手ではないですよ。ということで、今日は英語でお話しします。よろしくお願いします！

Thanks for having me here to speak. It's a real pleasure to be in Okinawa amongst such wonderful company and I'm privileged to be able to talk about a fun little performance regression that we found in string interpolation, and fixed for Ruby 3.3

Matt Valentine-House

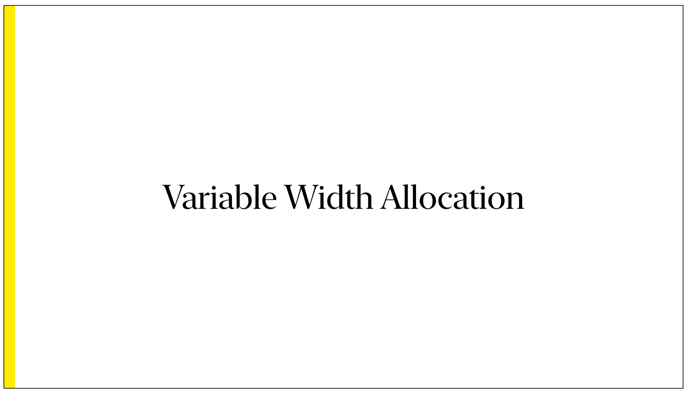
Senior Developer, Ruby Infrastructure

shopify

@eightbitraptor@ruby.social

@eightbitraptor

My name is Matt. I am known as eightbitraptor on the internet. I'm a full time Ruby Core committer and I work for Shopify in the Ruby Infrastructure team.

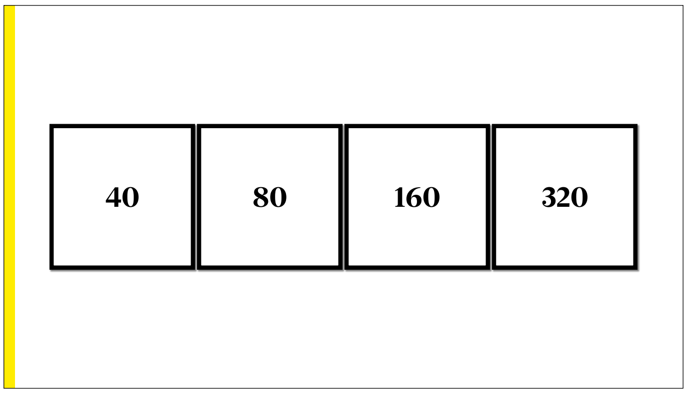

Variable Width Allocation

This talk is related to variable width allocation, A feature my team and I developed to improve the performance of Ruby by changing how memory was laid out. We shipped the first version for Ruby 3.2.

We continued to work on it during the development cycle of Ruby 3.3 to improve the performance, and clean up the implementation and we noticed a performance regression for certain types of string interpolation.

I set out to work out where that regression had come from, and whether it was related to our changes.

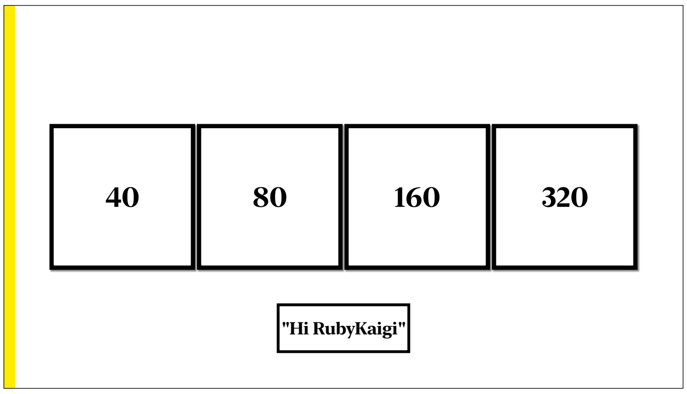
I'm not going to talk about VWA in depth in this talk, as we've already covered it at previous Ruby Kaigi's



It is important to understand how Ruby uses memory for this talk though, so I'm just going to give a quick summary of object allocation.

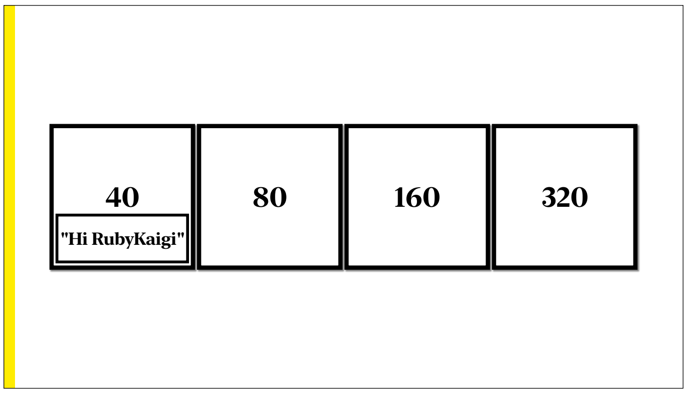
When you create a Ruby object, it gets allocated in a specific part of the heap, called a size pool, based on the size of the object.

the smallest size pool holds objects that are up to 40 bytes in size and the thresholds double from there. The largest size pool holds objects up to 640 bytes, and anything bigger than that is treated separately.

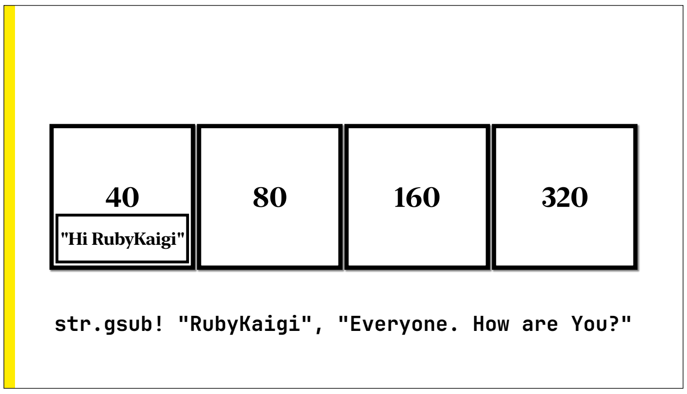


when we create an object, Ruby chooses an appropriate size pool depending on the total size of that object and allocates it.

An object that fits completely within the slots of its respective size pool it's called an "embedded" object.

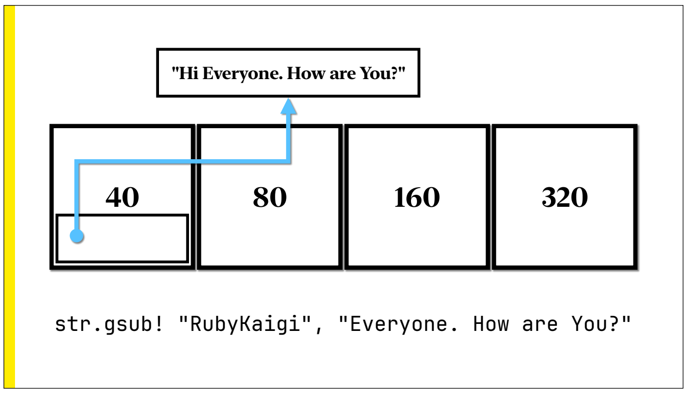


But, Objects are mutable, and can be resized after allocation.



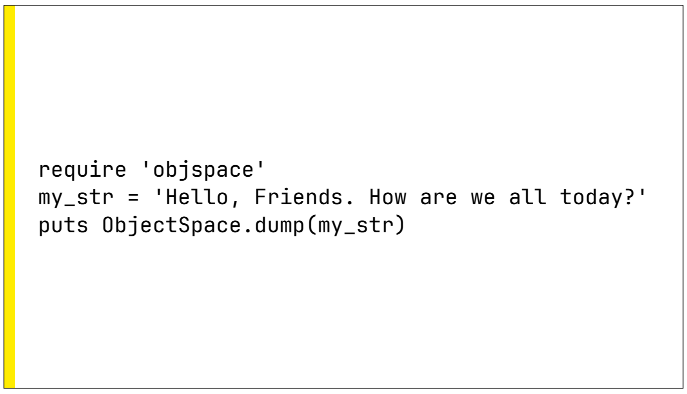
```
str.gsub! "RubyKaigi", "Everyone. How are You?"
```

For instance, if we substitute part of the string for something longer using gsub! The receiver object will be modified directly. Potentially causing it to no longer fit in the size pool to which it was allocated



```
str.gsub! "RubyKaigi", "Everyone. How are You?"
```

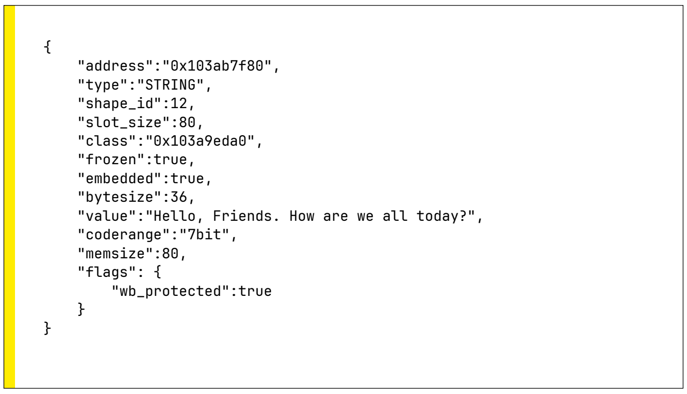
In this case, Ruby allocates a chunk of memory somewhere else outside of it's heap, its not really important where for now, and it creates a pointer from the original object in the size pool, out to the new memory. This object is now referred to as extended rather than embedded, because it doesn't fit fully in it's size pool anymore.



```
require 'objspace'
my_str = 'Hello, Friends. How are we all today?'
puts ObjectSpace.dump(my_str)
```

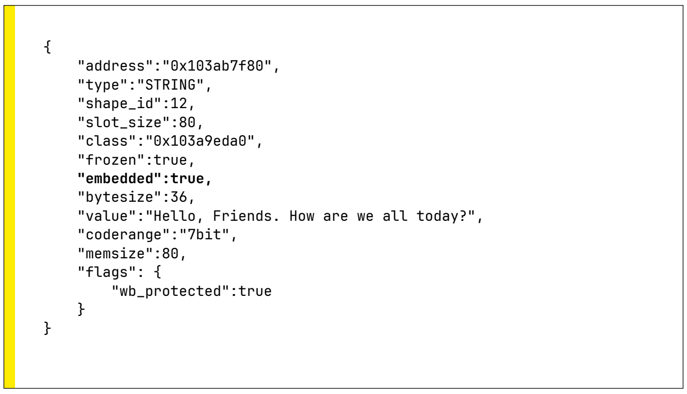
We can inspect vm specific information about objects using the Objectspace dump api, and now this will show you information now about the object size, which pool it's been allocated in and whether it's embedded or not.

So when we run code this to inspect a string.



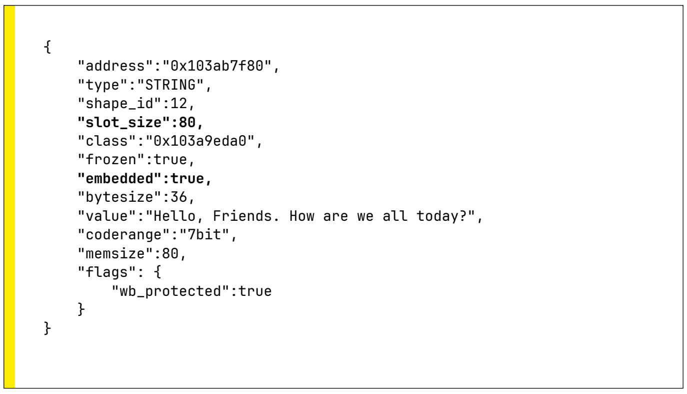
```
{
    "address":"0x103ab7f80",
    "type":"STRING",
    "shape_id":12,
    "slot_size":80,
    "class":"0x103a9eda0",
    "frozen":true,
    "embedded":true,
    "bytesize":36,
    "value":"Hello, Friends. How are we all today?",
    "coderange":"7bit",
    "memsize":80,
    "flags": {
        "wb_protected":true
    }
}
```

We get this. We can see lots of information about this string object - it's address, whether it's frozen or not, whether it's write barrier protected



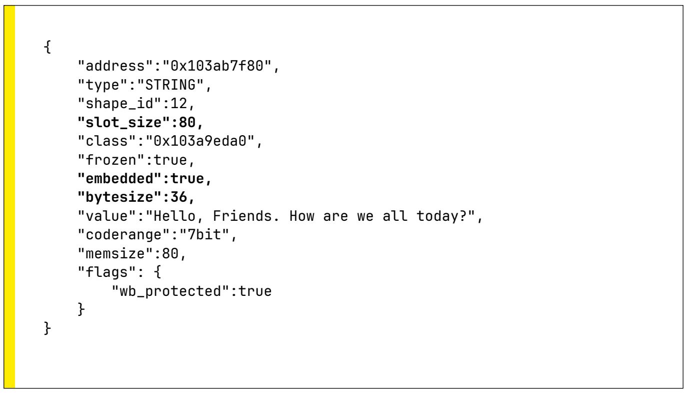
```
{
    "address":"0x103ab7f80",
    "type":"STRING",
    "shape_id":12,
    "slot_size":80,
    "class":"0x103a9eda0",
    "frozen":true,
    "embedded":true,
    "bytesize":36,
    "value":"Hello, Friends. How are we all today?",
    "coderange":"7bit",
    "memsize":80,
    "flags": {
        "wb_protected":true
    }
}
```

And also the embedded status, this string is clearly embedded



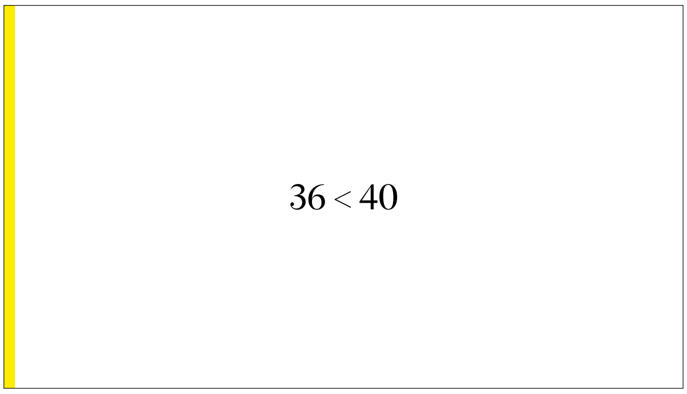
```
{
    "address":"0x103ab7f80",
    "type":"STRING",
    "shape_id":12,
    "slot_size":80,
    "class":"0x103a9eda0",
    "frozen":true,
    "embedded":true,
    "bytesize":36,
    "value":"Hello, Friends. How are we all today?",
    "coderange":"7bit",
    "memsize":80,
    "flags": {
        "wb_protected":true
    }
}
```

And it's in the 80 byte size pool



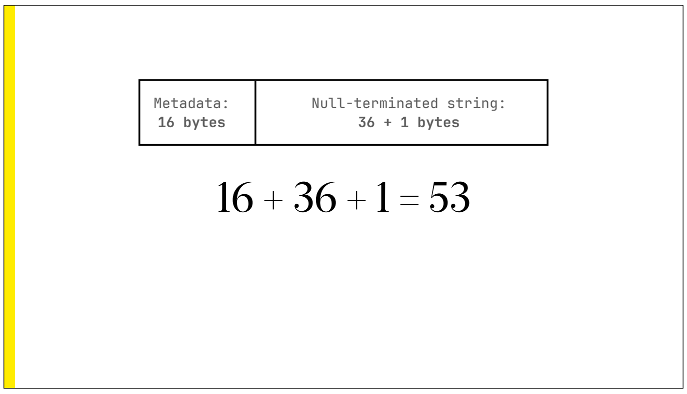
```
{
    "address":"0x103ab7f80",
    "type":"STRING",
    "shape_id":12,
    "slot_size":80,
    "class":"0x103a9eda0",
    "frozen":true,
    "embedded":true,
    "bytesize":36,
    "value":"Hello, Friends. How are we all today?",
    "coderange":"7bit",
    "memsize":80,
    "flags": {
        "wb_protected":true
    }
}
```

And the bitesize of the string, that is the amount of memory used by the actual string data, which is 36 bytes.



$$36 < 40$$

Now. Given that the string is only 36 bytes long, we need to explain why this has been allocated in the 80 byte size pool instead of the 40 byte one.



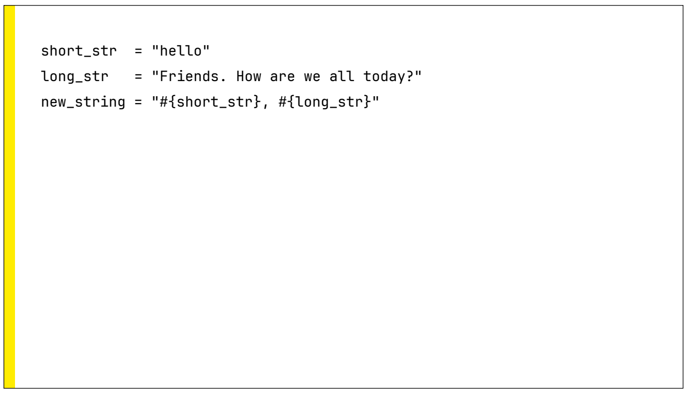
Metadata: 16 bytes	Null-terminated string: 36 + 1 bytes

$$16 + 36 + 1 = 53$$

Every Ruby object is prefixed with 16 bytes of metadata. This metadata contains some internal status flags for the object, as well as information about it's Klass.

And in Ruby strings are null terminated, so we have to add 1 for the terminator,

This gives us a total of 53 bytes, making it too big for the 40 byte bucket, so an 80 byte embedded allocation makes perfect sense.

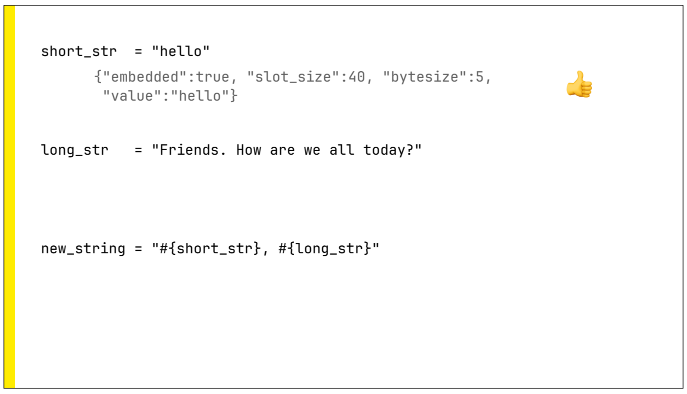


```ruby
short_str  = "hello"
long_str   = "Friends. How are we all today?"
new_string = "#{short_str}, #{long_str}"
```

So that's all for our background.

At some point while I was testing the Variable Width Allocation patches, I had some code that looked like this. It allocates a new string, which is built from two other strings combined together using string interpolation and inserts it into a local variable.

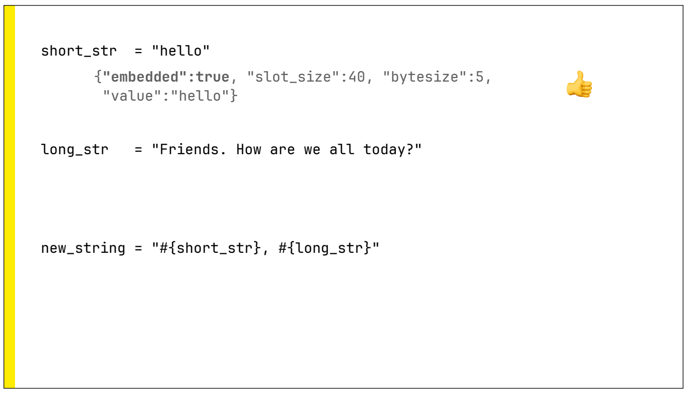
I wanted to make sure that all these strings are being allocated in the right place so I inspected them using the objectspace api.



```
short_str  = "hello"
      {"embedded":true, "slot_size":40, "bytesize":5,
       "value":"hello"}                                   👍

long_str   = "Friends. How are we all today?"



new_string = "#{short_str}, #{long_str}"
```

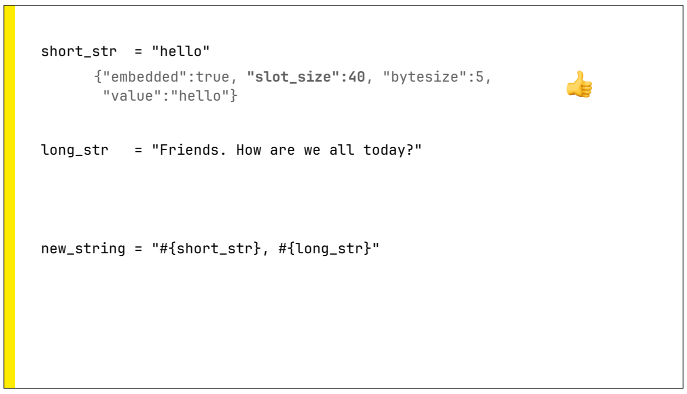
And this is what I got.

First string looks good



```
short_str  = "hello"
      {"embedded":true, "slot_size":40, "bytesize":5,
       "value":"hello"}                                  👍

long_str   = "Friends. How are we all today?"



new_string = "#{short_str}, #{long_str}"
```

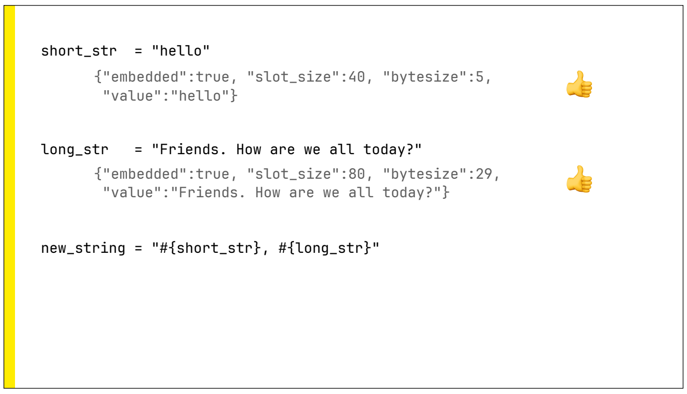
it's embedded



```
short_str  = "hello"
      {"embedded":true, "slot_size":40, "bytesize":5,
       "value":"hello"}                                  👍

long_str   = "Friends. How are we all today?"



new_string = "#{short_str}, #{long_str}"
```

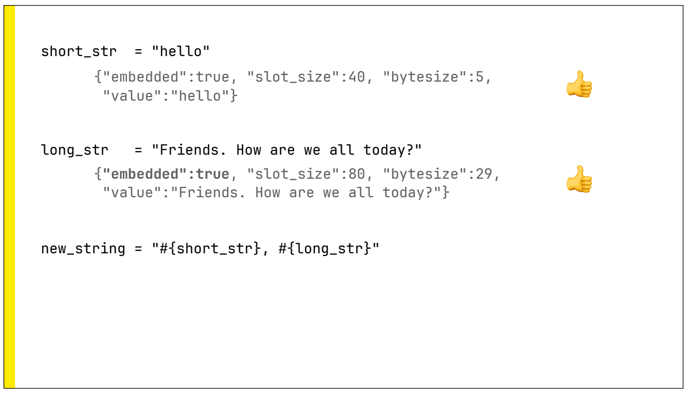
In the 40 byte bucket, which makes sense as it's bytesize is 5

remember 5 + 16 is 21, plus the null terminator makes the total size 22 bytes



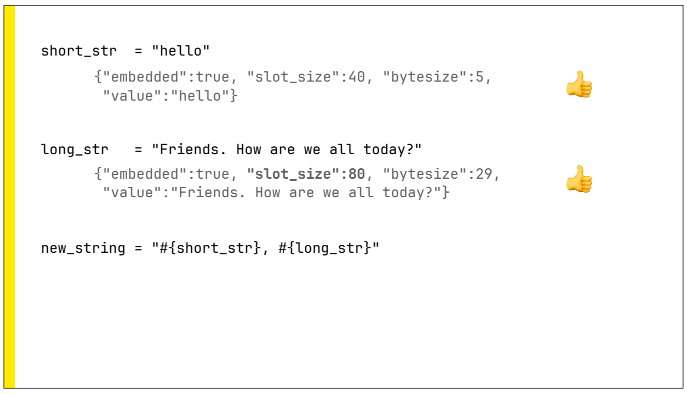
```
short_str  = "hello"
      {"embedded":true, "slot_size":40, "bytesize":5,
       "value":"hello"}                                   👍

long_str   = "Friends. How are we all today?"
      {"embedded":true, "slot_size":80, "bytesize":29,
       "value":"Friends. How are we all today?"}          👍

new_string = "#{short_str}, #{long_str}"
```

second string also looks good



```
short_str  = "hello"
      {"embedded":true, "slot_size":40, "bytesize":5,
       "value":"hello"}                                   👍

long_str   = "Friends. How are we all today?"
      {"embedded":true, "slot_size":80, "bytesize":29,
       "value":"Friends. How are we all today?"}          👍

new_string = "#{short_str}, #{long_str}"
```

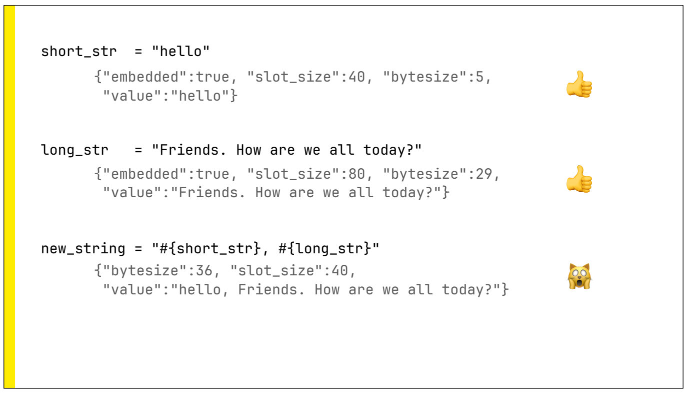
it's embedded



```
short_str  = "hello"
      {"embedded":true, "slot_size":40, "bytesize":5,
       "value":"hello"}                                   👍

long_str   = "Friends. How are we all today?"
      {"embedded":true, "slot_size":80, "bytesize":29,
       "value":"Friends. How are we all today?"}          👍

new_string = "#{short_str}, #{long_str}"
```

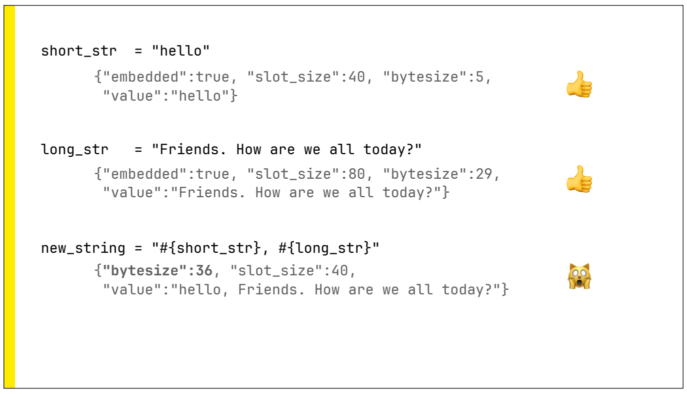
in the 80 byte bucket.

again bytesize of 29, all the extra stuff makes 46 bytes



```
short_str  = "hello"
      {"embedded":true, "slot_size":40, "bytesize":5,
       "value":"hello"}                                   👍

long_str   = "Friends. How are we all today?"
      {"embedded":true, "slot_size":80, "bytesize":29,
       "value":"Friends. How are we all today?"}          👍

new_string = "#{short_str}, #{long_str}"
      {"bytesize":36, "slot_size":40,
       "value":"hello, Friends. How are we all today?"}   🙀
```

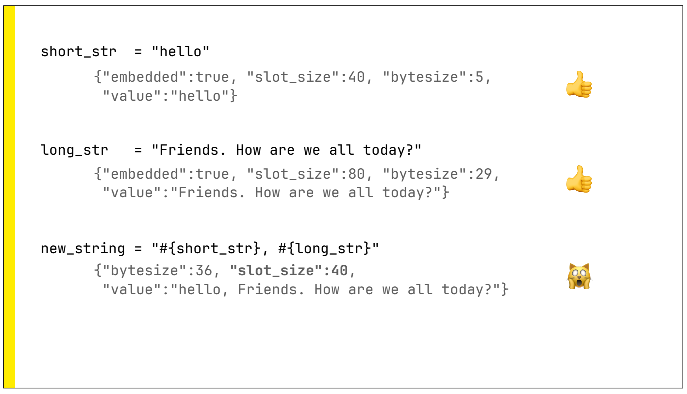
third string, is not what I expected to see.



```
short_str  = "hello"
      {"embedded":true, "slot_size":40, "bytesize":5,
       "value":"hello"}                                      👍

long_str   = "Friends. How are we all today?"
      {"embedded":true, "slot_size":80, "bytesize":29,
       "value":"Friends. How are we all today?"}             👍

new_string = "#{short_str}, #{long_str}"
      {"bytesize":36, "slot_size":40,
       "value":"hello, Friends. How are we all today?"}      🙀
```

This new string is the result of the interpolation of the first two strings into a new object.

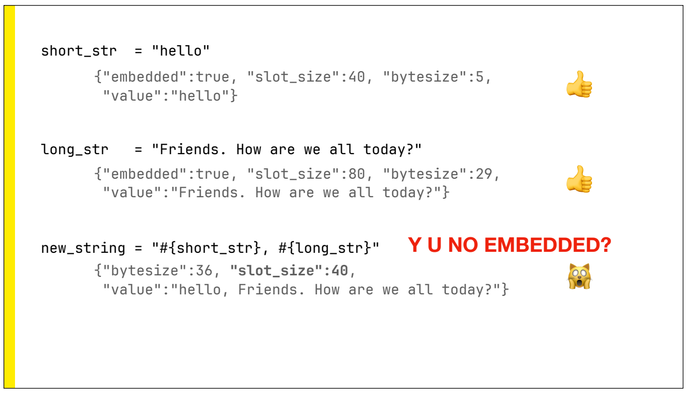
the bytesize of this new string is 36, so we add the extra header and terminator and we get a total object size of 53 bytes.



```
short_str  = "hello"
      {"embedded":true, "slot_size":40, "bytesize":5,
       "value":"hello"}                                      👍

long_str   = "Friends. How are we all today?"
      {"embedded":true, "slot_size":80, "bytesize":29,
       "value":"Friends. How are we all today?"}             👍

new_string = "#{short_str}, #{long_str}"
      {"bytesize":36, "slot_size":40,
       "value":"hello, Friends. How are we all today?"}      🙀
```

so why is it in the 40 byte bucket

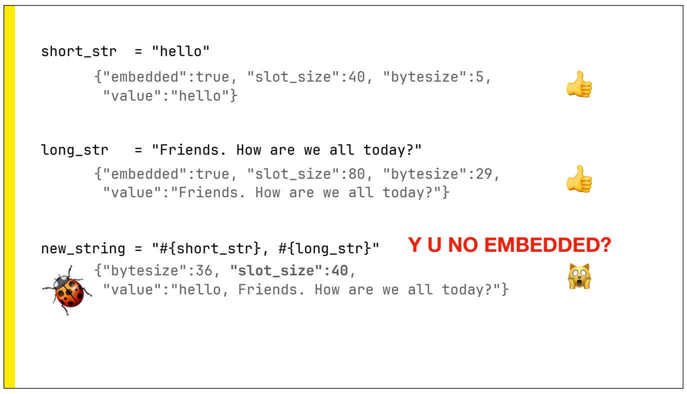
And the lack of an embedded flag means that this string has been created as an extended object.



```
short_str  = "hello"
      {"embedded":true, "slot_size":40, "bytesize":5,
       "value":"hello"}                                   👍

long_str   = "Friends. How are we all today?"
      {"embedded":true, "slot_size":80, "bytesize":29,
       "value":"Friends. How are we all today?"}          👍

new_string = "#{short_str}, #{long_str}"
      {"bytesize":36, "slot_size":40,
       "value":"hello, Friends. How are we all today?"}   🙀
```

Y U NO EMBEDDED?

Why was this not allocated as an embedded object in the 80 byte bucket?

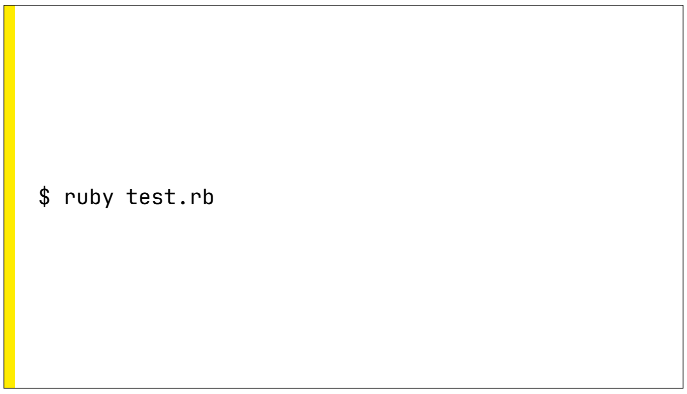


```
short_str  = "hello"
      {"embedded":true, "slot_size":40, "bytesize":5,
       "value":"hello"}                                   👍

long_str   = "Friends. How are we all today?"
      {"embedded":true, "slot_size":80, "bytesize":29,
       "value":"Friends. How are we all today?"}          👍

new_string = "#{short_str}, #{long_str}"
🐞    {"bytesize":36, "slot_size":40,
       "value":"hello, Friends. How are we all today?"}   🙀
```

Y U NO EMBEDDED?

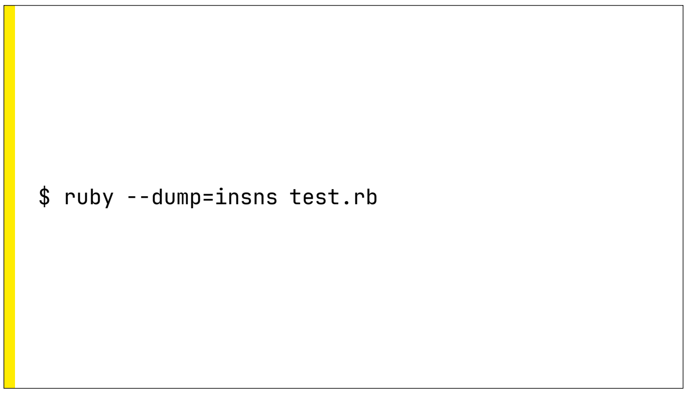
This was definitely a bug.

So I set out to try and find it.



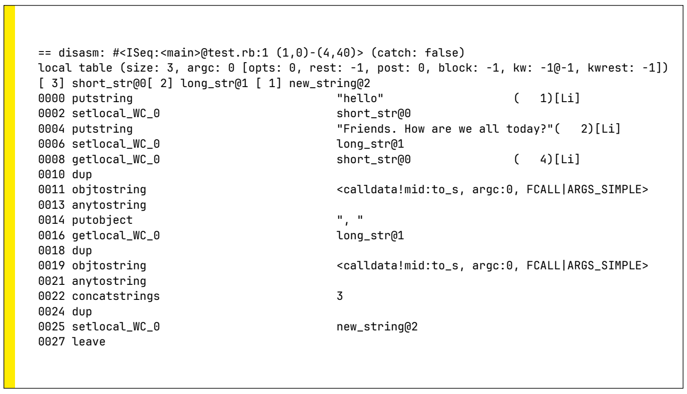
```
$ ruby test.rb
```

And the first thing I did was to look at exactly what the Ruby VM is doing when it executes that code.



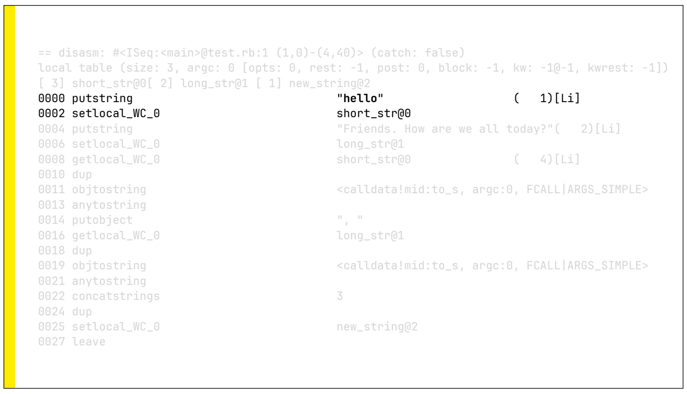
```
$ ruby --dump=insns test.rb
```

I ran the code again, using the dump insns option, to will parse and compile our code, but stop short of actually executing it



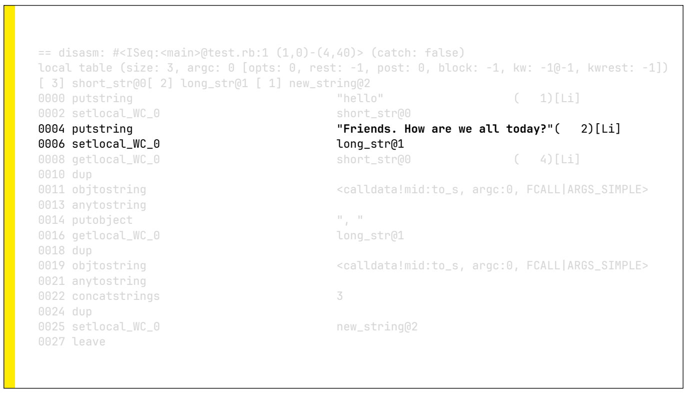
```
== disasm: #<ISeq:<main>@test.rb:1 (1,0)-(4,40)> (catch: false)
local table (size: 3, argc: 0 [opts: 0, rest: -1, post: 0, block: -1, kw: -1@-1, kwrest: -1])
[ 3] short_str@0[ 2] long_str@1 [ 1] new_string@2
0000 putstring                              "hello"                  (   1)[Li]
0002 setlocal_WC_0                          short_str@0
0004 putstring                              "Friends. How are we all today?"(   2)[Li]
0006 setlocal_WC_0                          long_str@1
0008 getlocal_WC_0                          short_str@0              (   4)[Li]
0010 dup
0011 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0013 anytostring
0014 putobject                              ", "
0016 getlocal_WC_0                          long_str@1
0018 dup
0019 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0021 anytostring
0022 concatstrings                          3
0024 dup
0025 setlocal_WC_0                          new_string@2
0027 leave
```

Instead it's going to print out the bytecode to the screen so we can see what's going on. This is a list of all the instructions the Ruby VM will execute, one after the other, to run the string interpolation code I showed on the previous slides



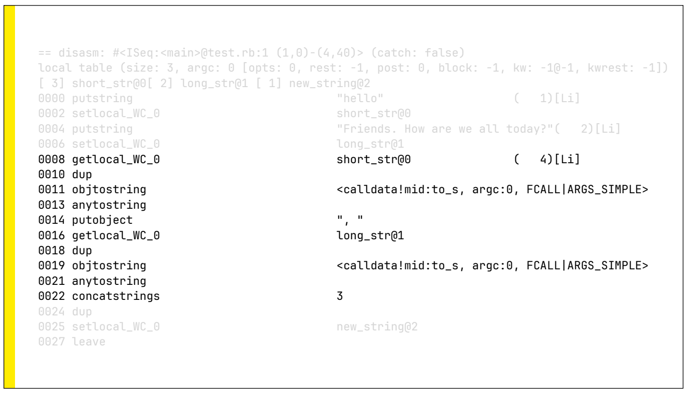
```
== disasm: #<ISeq:<main>@test.rb:1 (1,0)-(4,40)> (catch: false)
local table (size: 3, argc: 0 [opts: 0, rest: -1, post: 0, block: -1, kw: -1@-1, kwrest: -1])
[ 3] short_str@0[ 2] long_str@1 [ 1] new_string@2
0000 putstring                              "hello"                  (   1)[Li]
0002 setlocal_WC_0                          short_str@0
0004 putstring                              "Friends. How are we all today?"(   2)[Li]
0006 setlocal_WC_0                          long_str@1
0008 getlocal_WC_0                          short_str@0              (   4)[Li]
0010 dup
0011 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0013 anytostring
0014 putobject                              ", "
0016 getlocal_WC_0                          long_str@1
0018 dup
0019 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0021 anytostring
0022 concatstrings                          3
0024 dup
0025 setlocal_WC_0                          new_string@2
0027 leave
```

First Ruby takes our short string hello, which because it's static, is defined right inside the parser output, and pushes it onto the stack, and then uses the set local instruction. Which pops it straight off the stack and assign it to a local variable short_str



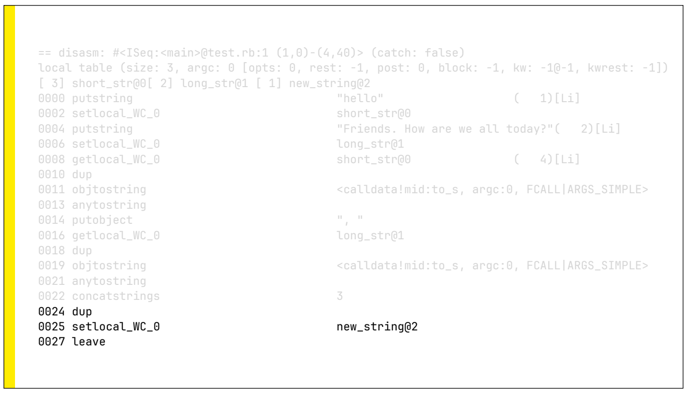
```
== disasm: #<ISeq:<main>@test.rb:1 (1,0)-(4,40)> (catch: false)
local table (size: 3, argc: 0 [opts: 0, rest: -1, post: 0, block: -1, kw: -1@-1, kwrest: -1])
[ 3] short_str@0[ 2] long_str@1 [ 1] new_string@2
0000 putstring                              "hello"                   (   1)[Li]
0002 setlocal_WC_0                          short_str@0
0004 putstring                              "Friends. How are we all today?"(   2)[Li]
0006 setlocal_WC_0                          long_str@1
0008 getlocal_WC_0                          short_str@0               (   4)[Li]
0010 dup
0011 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0013 anytostring
0014 putobject                              ", "
0016 getlocal_WC_0                          long_str@1
0018 dup
0019 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0021 anytostring
0022 concatstrings                          3
0024 dup
0025 setlocal_WC_0                          new_string@2
0027 leave
```

Then does the same thing with the long string, taking the string directly from the parser output and using set local to assign it directly to a local variable.



```
== disasm: #<ISeq:<main>@test.rb:1 (1,0)-(4,40)> (catch: false)
local table (size: 3, argc: 0 [opts: 0, rest: -1, post: 0, block: -1, kw: -1@-1, kwrest: -1])
[ 3] short_str@0[ 2] long_str@1 [ 1] new_string@2
0000 putstring                              "hello"                   (   1)[Li]
0002 setlocal_WC_0                          short_str@0
0004 putstring                              "Friends. How are we all today?"(   2)[Li]
0006 setlocal_WC_0                          long_str@1
0008 getlocal_WC_0                          short_str@0               (   4)[Li]
0010 dup
0011 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0013 anytostring
0014 putobject                              ", "
0016 getlocal_WC_0                          long_str@1
0018 dup
0019 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0021 anytostring
0022 concatstrings                          3
0024 dup
0025 setlocal_WC_0                          new_string@2
0027 leave
```

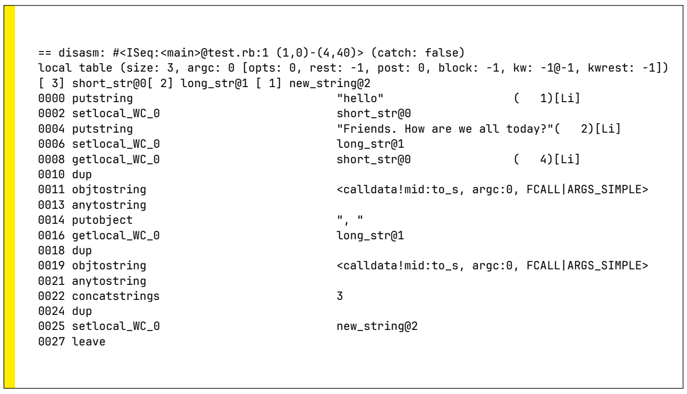
Next, This block is what actually performs the interpolation



```
== disasm: #<ISeq:<main>@test.rb:1 (1,0)-(4,40)> (catch: false)
local table (size: 3, argc: 0 [opts: 0, rest: -1, post: 0, block: -1, kw: -1@-1, kwrest: -1])
[ 3] short_str@0[ 2] long_str@1 [ 1] new_string@2
0000 putstring                              "hello"                   (   1)[Li]
0002 setlocal_WC_0                          short_str@0
0004 putstring                              "Friends. How are we all today?"(   2)[Li]
0006 setlocal_WC_0                          long_str@1
0008 getlocal_WC_0                          short_str@0               (   4)[Li]
0010 dup
0011 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0013 anytostring
0014 putobject                              ", "
0016 getlocal_WC_0                          long_str@1
0018 dup
0019 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0021 anytostring
0022 concatstrings                          3
0024 dup
0025 setlocal_WC_0                          new_string@2
0027 leave
```

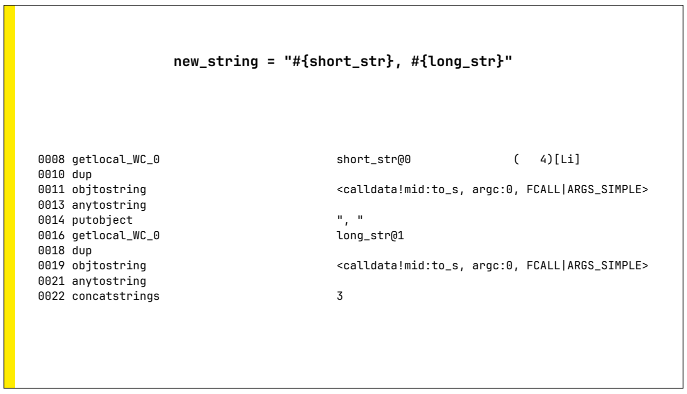
and finally, the value from the interpolation gets assigned to a local variable new_string, again using the set local instruction.

and then we're done. Our program exits, returning the newly concatenated string, which is the last thing left on the stack.



```
== disasm: #<ISeq:<main>@test.rb:1 (1,0)-(4,40)> (catch: false)
local table (size: 3, argc: 0 [opts: 0, rest: -1, post: 0, block: -1, kw: -1@-1, kwrest: -1])
[ 3] short_str@0[ 2] long_str@1 [ 1] new_string@2
0000 putstring                              "hello"                   (   1)[Li]
0002 setlocal_WC_0                          short_str@0
0004 putstring                              "Friends. How are we all today?"(   2)[Li]
0006 setlocal_WC_0                          long_str@1
0008 getlocal_WC_0                          short_str@0               (   4)[Li]
0010 dup
0011 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0013 anytostring
0014 putobject                              ", "
0016 getlocal_WC_0                          long_str@1
0018 dup
0019 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0021 anytostring
0022 concatstrings                          3
0024 dup
0025 setlocal_WC_0                          new_string@2
0027 leave
```

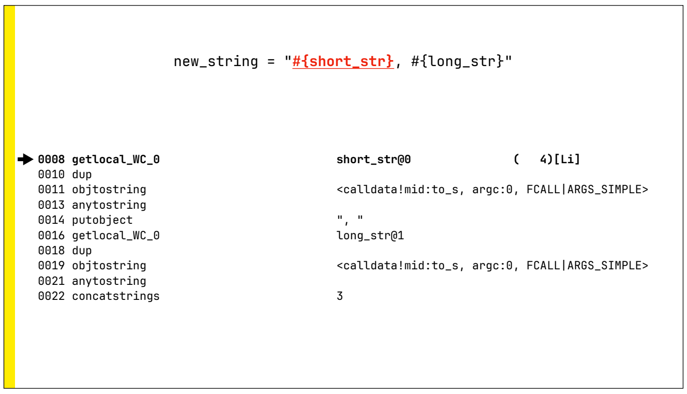
I don't think the actual local variable setting code is very relevant, so lets ignore that for now



```
new_string = "#{short_str}, #{long_str}"

0008 getlocal_WC_0                          short_str@0              (   4)[Li]
0010 dup
0011 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0013 anytostring
0014 putobject                              ", "
0016 getlocal_WC_0                          long_str@1
0018 dup
0019 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0021 anytostring
0022 concatstrings                          3
```

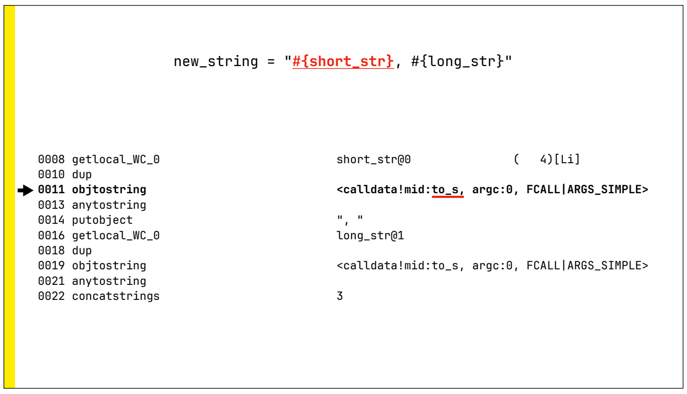
And zoom into the actual interpolation bytecode.

You can see the Ruby code that this byte code represents at the top of the screen here. We have two local variable substitutions, inside a string literal.



```
new_string = "#{short_str}, #{long_str}"

➔ 0008 getlocal_WC_0                        short_str@0              (   4)[Li]
0010 dup
0011 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0013 anytostring
0014 putobject                              ", "
0016 getlocal_WC_0                          long_str@1
0018 dup
0019 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0021 anytostring
0022 concatstrings                          3
```

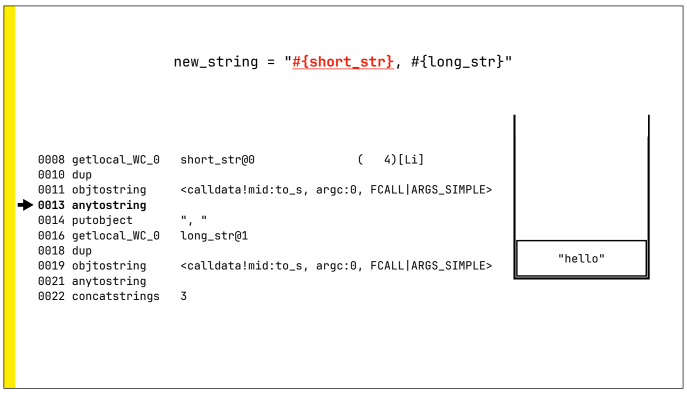
we start by computing the first substitution, substitutions can contain any syntactically valid ruby, but in this case it's been parsed as a local variable, so the get local instruction has been emitted, which will look up the local value and push it on to the stack



```
new_string = "#{short_str}, #{long_str}"
```

```
0008 getlocal_WC_0                          short_str@0              (   4)[Li]
0010 dup
0011 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0013 anytostring
0014 putobject                              ", "
0016 getlocal_WC_0                          long_str@1
0018 dup
0019 objtostring                            <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0021 anytostring
0022 concatstrings                          3
```

then we call the to_s method on it

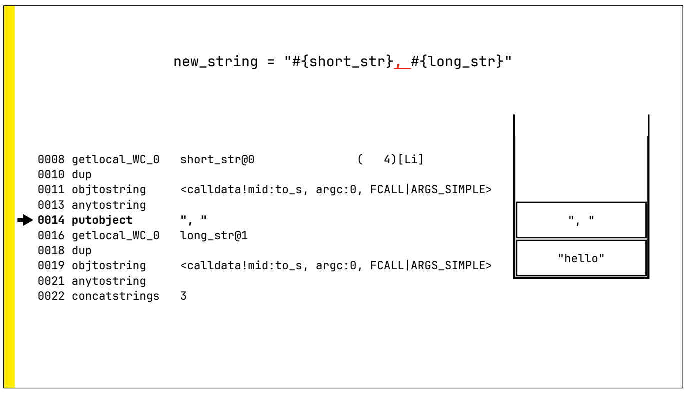


```
new_string = "#{short_str}, #{long_str}"
```

```
0008 getlocal_WC_0   short_str@0              (   4)[Li]
0010 dup
0011 objtostring     <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0013 anytostring
0014 putobject       ", "
0016 getlocal_WC_0   long_str@1
0018 dup
0019 objtostring     <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
0021 anytostring
0022 concatstrings   3
```

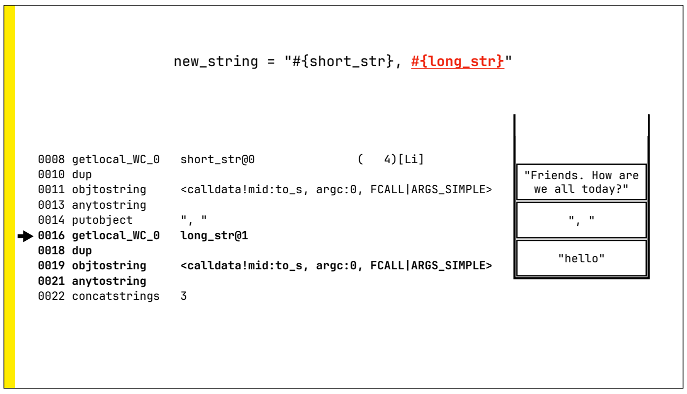
after that we use this anytostring instruction, this is just here to make sure that what we actually push on to the stack, is a string. for example, if to_s had been monkeypatched to return another type, then anytostring would brute force that into a string and push it on the stack

Here on the right, I've tried to visualise what the stack looks like at this point in the program.



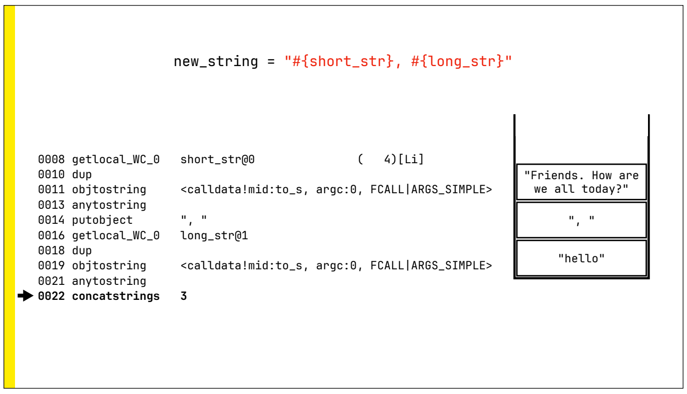
```
new_string = "#{short_str}, #{long_str}"

  0008 getlocal_WC_0   short_str@0              (   4)[Li]
  0010 dup
  0011 objtostring     <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
  0013 anytostring
→ 0014 putobject       ", "
  0016 getlocal_WC_0   long_str@1
  0018 dup
  0019 objtostring     <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
  0021 anytostring
  0022 concatstrings   3
```

Next we deal with the characters between the substitution blocks. This just means pulling out the comma and the space as a string literal and pushing it on the stack



```
new_string = "#{short_str}, #{long_str}"

  0008 getlocal_WC_0   short_str@0              (   4)[Li]
  0010 dup
  0011 objtostring     <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
  0013 anytostring
  0014 putobject       ", "
→ 0016 getlocal_WC_0   long_str@1
  0018 dup
  0019 objtostring     <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
  0021 anytostring
  0022 concatstrings   3
```

Then we do the same operations we did for the short string to the long string, grab the value for the local, coerce it into a string, first using to_s and then anytostring, and pushing the result onto the stack.

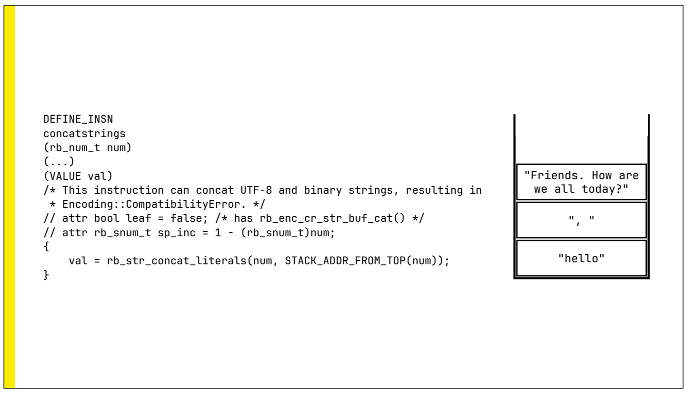


```
new_string = "#{short_str}, #{long_str}"
```

```
  0008 getlocal_WC_0   short_str@0              (   4)[Li]
  0010 dup
  0011 objtostring     <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
  0013 anytostring
  0014 putobject       ", "
  0016 getlocal_WC_0   long_str@1
  0018 dup
  0019 objtostring     <calldata!mid:to_s, argc:0, FCALL|ARGS_SIMPLE>
  0021 anytostring
➔ 0022 concatstrings   3
```

And finally we use the concatstrings instruction to pull the 3 strings off the stack and combine them together into our new string.

So, seeing as this concatstrings instruction seems to be what's doing the actual string building here lets investigate.

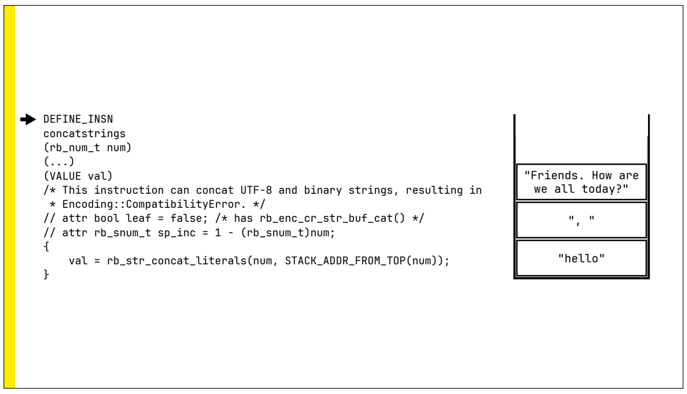


```
DEFINE_INSN
concatstrings
(rb_num_t num)
(...)
(VALUE val)
/* This instruction can concat UTF-8 and binary strings, resulting in
 * Encoding::CompatibilityError. */
// attr bool leaf = false; /* has rb_enc_cr_str_buf_cat() */
// attr rb_snum_t sp_inc = 1 - (rb_snum_t)num;
{
    val = rb_str_concat_literals(num, STACK_ADDR_FROM_TOP(num));
}
```

All valid VM instructions are defined in a file called insns.def inside the Ruby source code, using a C based DSL.

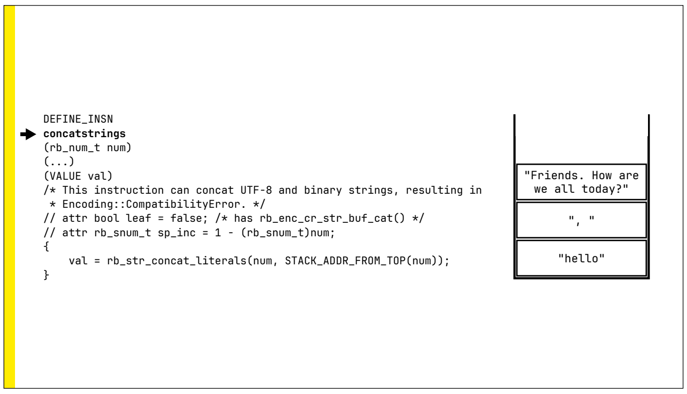
During the compilation stage of the Ruby interpreter, this DSL is processed, to generate C functions for each instruction that get linked into the VM Core.

Having them initially in this DSL form, gives them some structure that makes them easy to read and reason about. Every instruction starts with a call to the DEFINE_INSN macro.



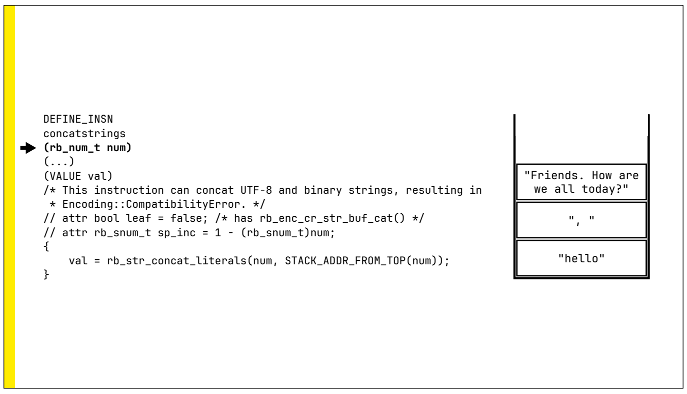
```
DEFINE_INSN
concatstrings
(rb_num_t num)
(...)
(VALUE val)
/* This instruction can concat UTF-8 and binary strings, resulting in
 * Encoding::CompatibilityError. */
// attr bool leaf = false; /* has rb_enc_cr_str_buf_cat() */
// attr rb_snum_t sp_inc = 1 - (rb_snum_t)num;
{
    val = rb_str_concat_literals(num, STACK_ADDR_FROM_TOP(num));
}
```

Every instruction starts with a call to the DEFINE_INSN macro.



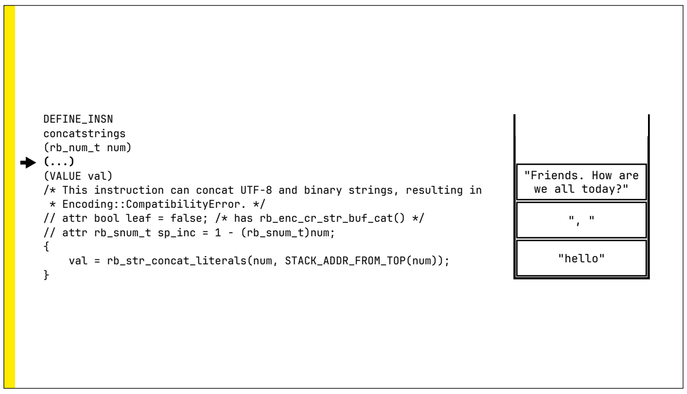
```
DEFINE_INSN
concatstrings
(rb_num_t num)
(...)
(VALUE val)
/* This instruction can concat UTF-8 and binary strings, resulting in
 * Encoding::CompatibilityError. */
// attr bool leaf = false; /* has rb_enc_cr_str_buf_cat() */
// attr rb_snum_t sp_inc = 1 - (rb_snum_t)num;
{
    val = rb_str_concat_literals(num, STACK_ADDR_FROM_TOP(num));
}
```

And that is then followed by the name of the instruction



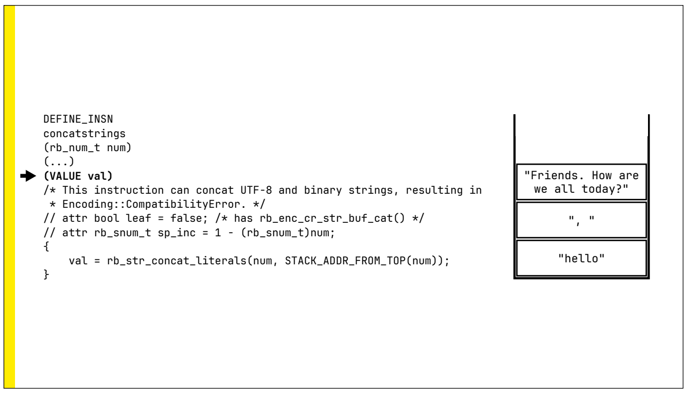
```
DEFINE_INSN
concatstrings
(rb_num_t num)
(...)
(VALUE val)
/* This instruction can concat UTF-8 and binary strings, resulting in
 * Encoding::CompatibilityError. */
// attr bool leaf = false; /* has rb_enc_cr_str_buf_cat() */
// attr rb_snum_t sp_inc = 1 - (rb_snum_t)num;
{
    val = rb_str_concat_literals(num, STACK_ADDR_FROM_TOP(num));
}
```

and a list of valid parameters and their types



```
DEFINE_INSN
concatstrings
(rb_num_t num)
(...)
(VALUE val)
/* This instruction can concat UTF-8 and binary strings, resulting in
 * Encoding::CompatibilityError. */
// attr bool leaf = false; /* has rb_enc_cr_str_buf_cat() */
// attr rb_snum_t sp_inc = 1 - (rb_snum_t)num;
{
    val = rb_str_concat_literals(num, STACK_ADDR_FROM_TOP(num));
}
```

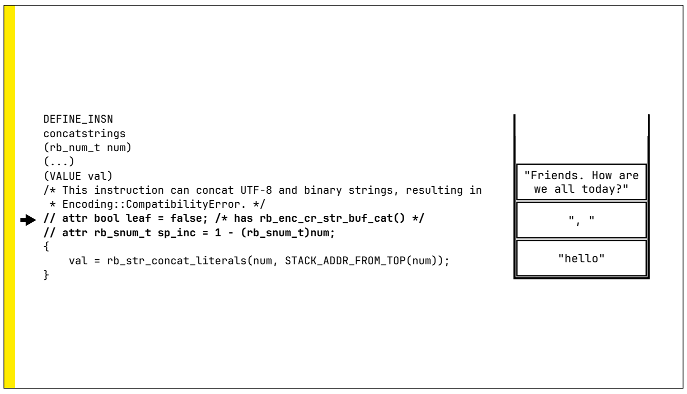
then the list of values this instruction will pop off the stack, in this case the ellipsis means this is variadic - it can be an arbitrary length list



```
  DEFINE_INSN
  concatstrings
  (rb_num_t num)
  (...)
➔ (VALUE val)
  /* This instruction can concat UTF-8 and binary strings, resulting in
   * Encoding::CompatibilityError. */
  // attr bool leaf = false; /* has rb_enc_cr_str_buf_cat() */
  // attr rb_snum_t sp_inc = 1 - (rb_snum_t)num;
  {
      val = rb_str_concat_literals(num, STACK_ADDR_FROM_TOP(num));
  }
```

"Friends. How are we all today?"
", "
"hello"

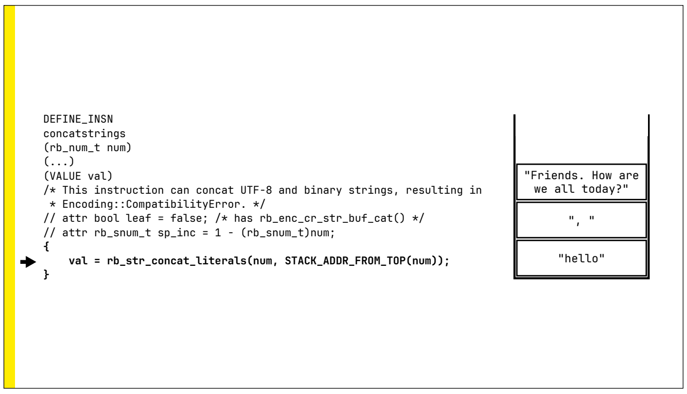
next we have the list of values that will be pushed back onto the stack



```
  DEFINE_INSN
  concatstrings
  (rb_num_t num)
  (...)
  (VALUE val)
  /* This instruction can concat UTF-8 and binary strings, resulting in
   * Encoding::CompatibilityError. */
➔ // attr bool leaf = false; /* has rb_enc_cr_str_buf_cat() */
  // attr rb_snum_t sp_inc = 1 - (rb_snum_t)num;
  {
      val = rb_str_concat_literals(num, STACK_ADDR_FROM_TOP(num));
  }
```

"Friends. How are we all today?"
", "
"hello"

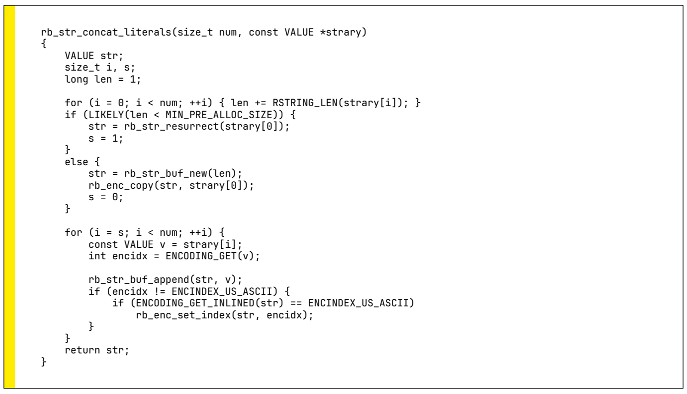
then some attributes, which I'm going to gloss over, as they're not important for our purposes in this talk.



```
DEFINE_INSN
concatstrings
(rb_num_t num)
(...)
(VALUE val)
/* This instruction can concat UTF-8 and binary strings, resulting in
 * Encoding::CompatibilityError. */
// attr bool leaf = false; /* has rb_enc_cr_str_buf_cat() */
// attr rb_snum_t sp_inc = 1 - (rb_snum_t)num;
{
    val = rb_str_concat_literals(num, STACK_ADDR_FROM_TOP(num));
}
```

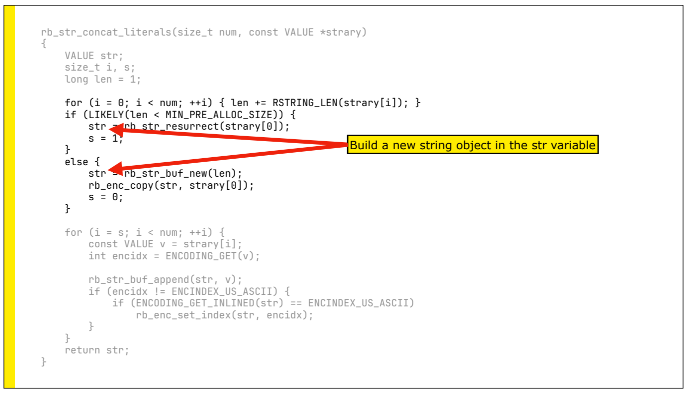
and finally the body of the instruction. This is the code that will be executed whenever the VM gets to a concatstrings instruction in a bytecode listing.

In this case, uses the num parameter passed in as an argument to the stack add from top macro to grab the top 3 elements from the stack and pass them as parameters to a c function rb_str_concat_literals, which is going to return a value



```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

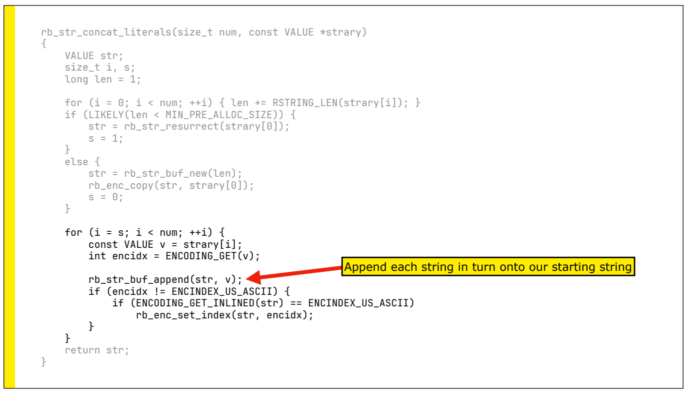
this is the rb_str_concat_literals function, and at first glance, it's sort of split into two halves



```c
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

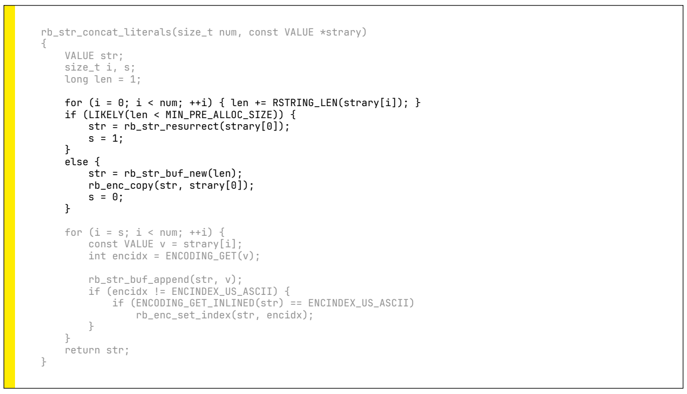
the first half, where we build a string object that will eventually be returned

---



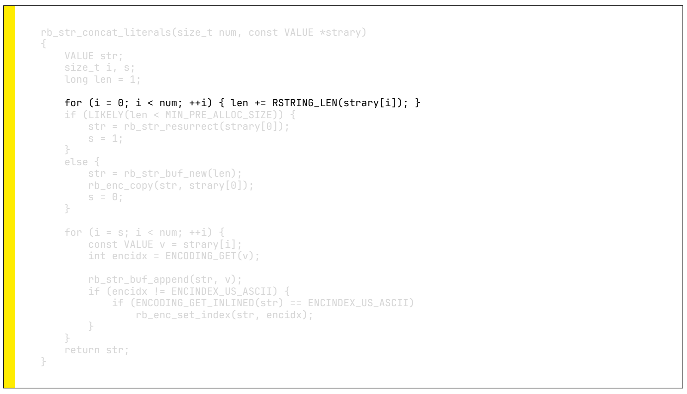
```c
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

and the second half, where we loops through the array passed in, appending those strings one by one onto the initial string we've defined in the first half



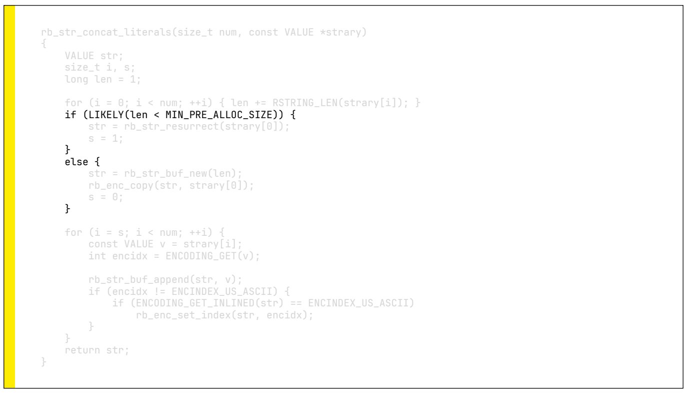
```c
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

but this first half is kind of weird



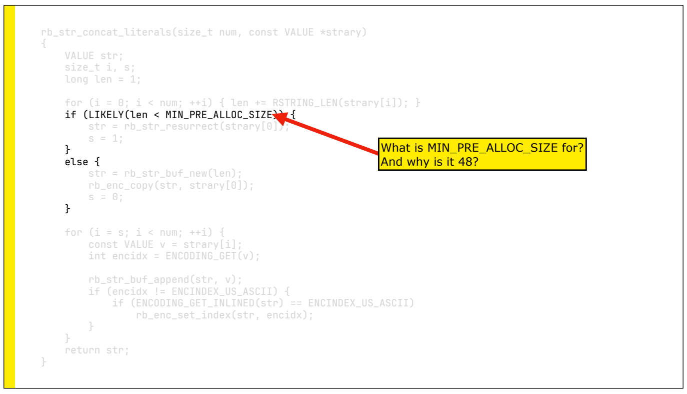
```c
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

it loops through the array of strings we've passed in to calculate the total length of the interpolated string



```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

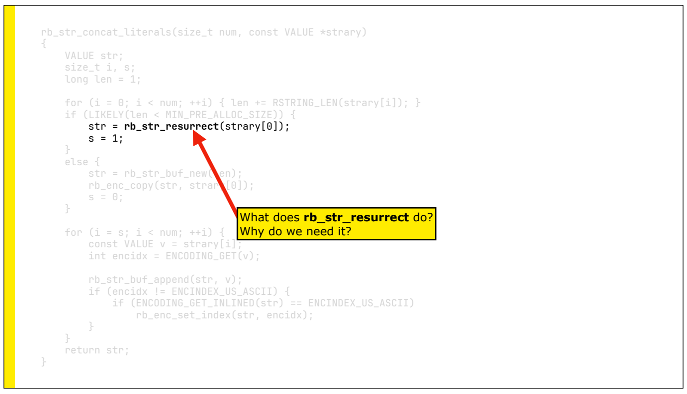
And then it branches



```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

What is MIN_PRE_ALLOC_SIZE for?
And why is it 48?

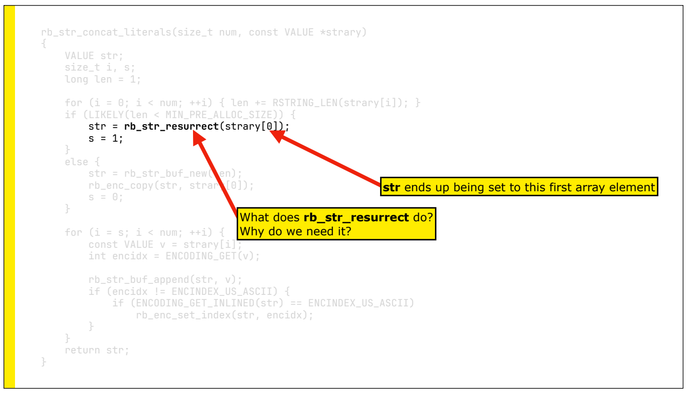
on this weird magic number, MIN_PRE_ALLOC_SIZE, which is 48



```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

What does **rb_str_resurrect** do?
Why do we need it?

if the length is less than 48, then it calls this rb_str_resurrect thing,

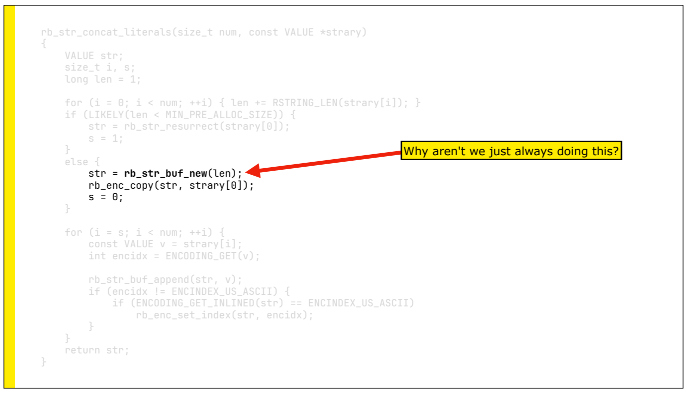


```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

str ends up being set to this first array element

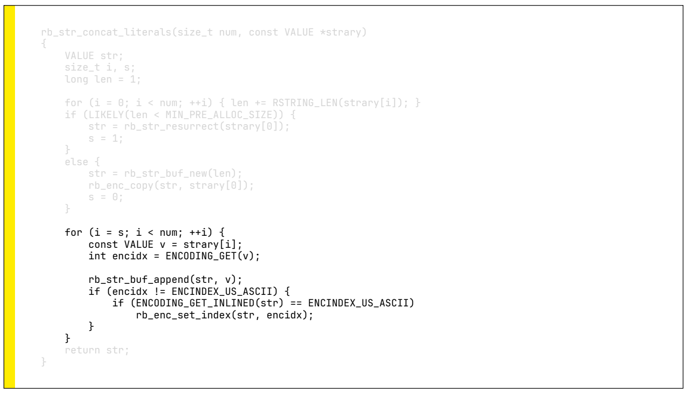
What does **rb_str_resurrect** do?
Why do we need it?

which eventually sets our base string to be the first string in the array



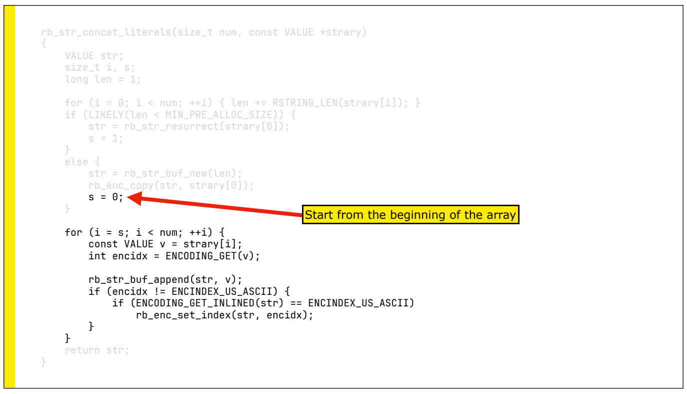
```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

however, if the length is longer than 48, it allocates a completely new string of the right size. I don't know why we're not just always doing this



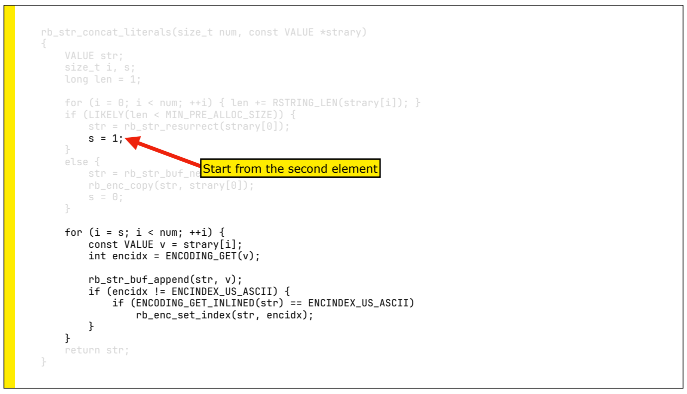
```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

when the second half of the function loops through the string array, it starts from either



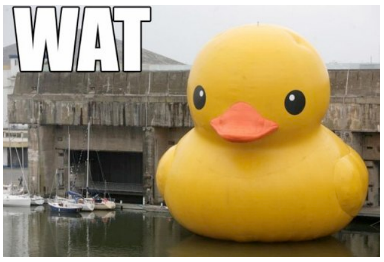
```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

the first, or



```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

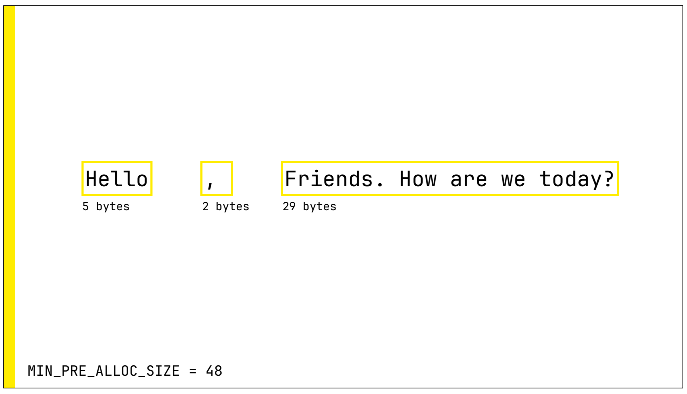
the second element, depending on which branch we took in the first half of the function



at this point I was pretty confused. I could not explain why we had to branch in this function, what the purpose of the string resurrection path was, what MIN_PRE_ALLOC_SIZE was supposed to represent or what the significance of the number 48 was.

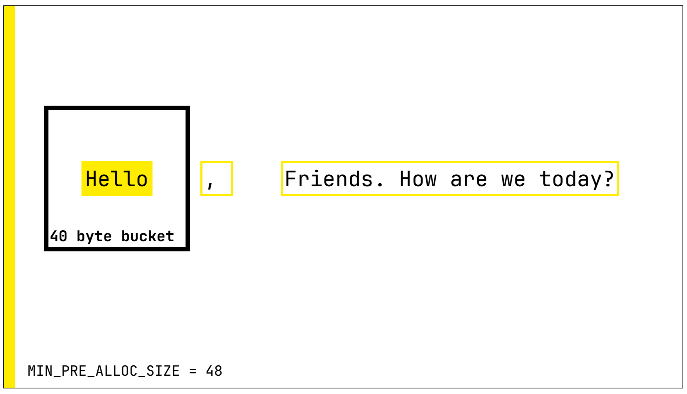
But this **does** explain our bug

However I had spotted the bug.

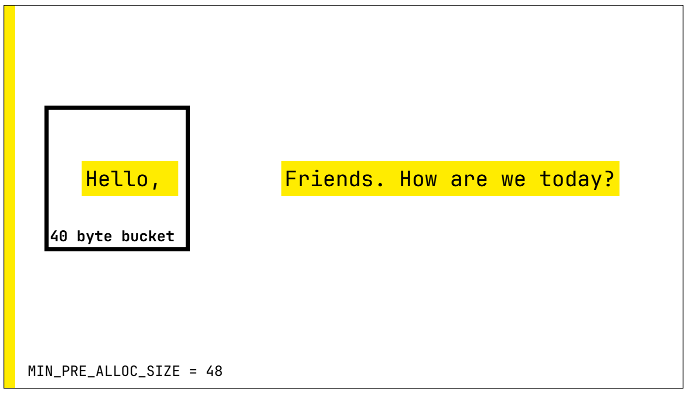


So we have these 3 strings. The total length of their data is 36 bytes.

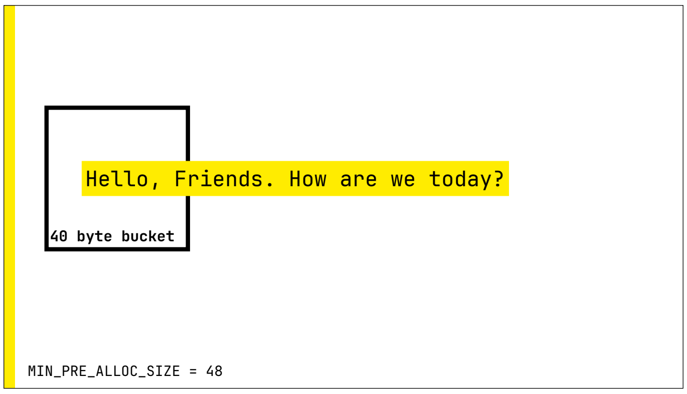
This is less than the MIN_PRE_ALLOC_SIZE of 48, so we'll go down the string resurrect path. Meaning we'll use our first string as a base



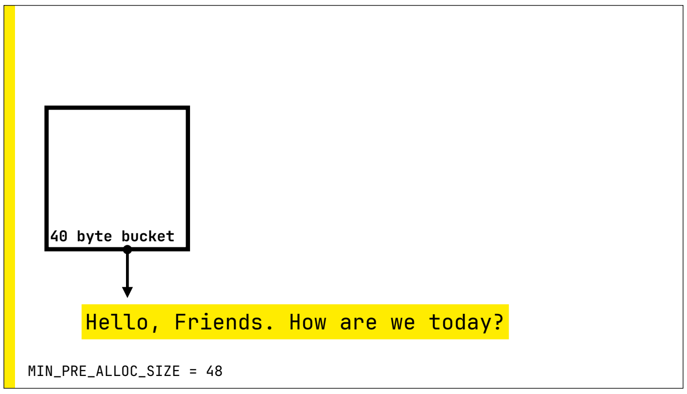
and as this string is in the 40 byte size pool



when we start appending things to it

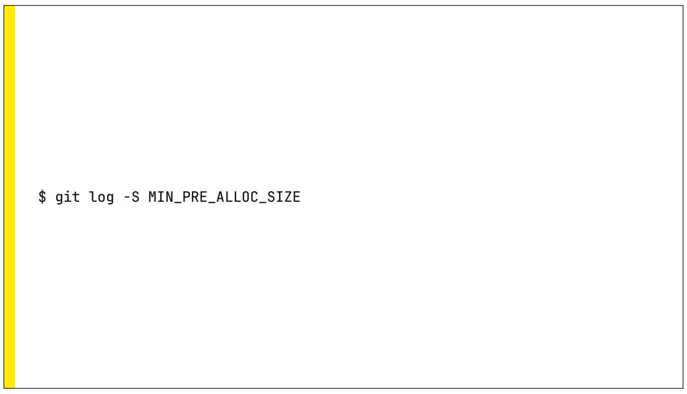


it'll quickly push past the boundary of the 40 byte size pool, causing Ruby to convert it to an extended object



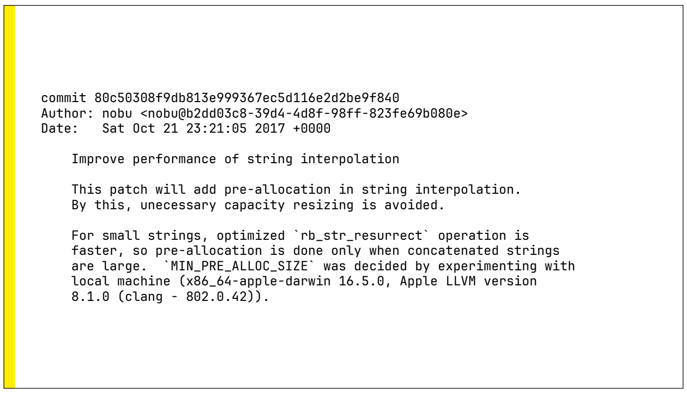
So we end up with an extended object in the 40 byte size pool, rather than the embedded 80 byte object we were expecting.

Cool. So we found our bug. Now I wanted to find out why that code even exists.



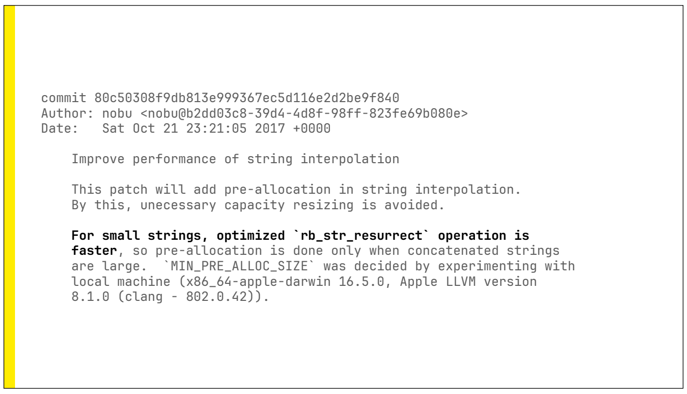
```
$ git log -S MIN_PRE_ALLOC_SIZE
```

I broke out the git pickaxe, that is git log -s, to find the commit that had first introduced this magic number



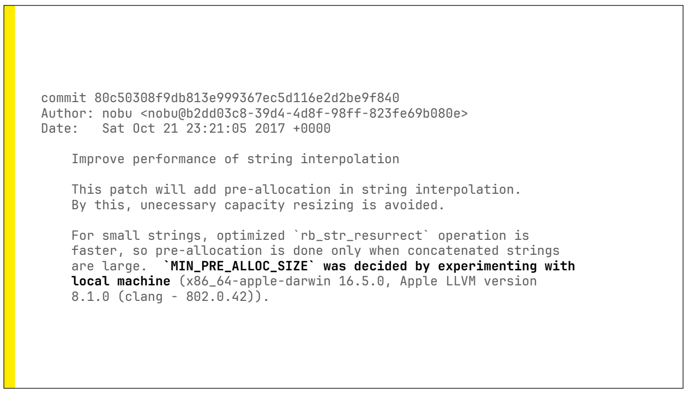
```
commit 80c50308f9db813e999367ec5d116e2d2be9f840
Author: nobu <nobu@b2dd03c8-39d4-4d8f-98ff-823fe69b080e>
Date:   Sat Oct 21 23:21:05 2017 +0000

    Improve performance of string interpolation

    This patch will add pre-allocation in string interpolation.
    By this, unecessary capacity resizing is avoided.

    For small strings, optimized `rb_str_resurrect` operation is
    faster, so pre-allocation is done only when concatenated strings
    are large.  `MIN_PRE_ALLOC_SIZE` was decided by experimenting with
    local machine (x86_64-apple-darwin 16.5.0, Apple LLVM version
    8.1.0 (clang - 802.0.42)).
```

and it led me to this optimisation, originally written by Minami Nao, and committed by nobu , in 2017



```
commit 80c50308f9db813e999367ec5d116e2d2be9f840
Author: nobu <nobu@b2dd03c8-39d4-4d8f-98ff-823fe69b080e>
Date:   Sat Oct 21 23:21:05 2017 +0000

    Improve performance of string interpolation

    This patch will add pre-allocation in string interpolation.
    By this, unecessary capacity resizing is avoided.

    For small strings, optimized `rb_str_resurrect` operation is
    faster, so pre-allocation is done only when concatenated strings
    are large.  `MIN_PRE_ALLOC_SIZE` was decided by experimenting with
    local machine (x86_64-apple-darwin 16.5.0, Apple LLVM version
    8.1.0 (clang - 802.0.42)).
```

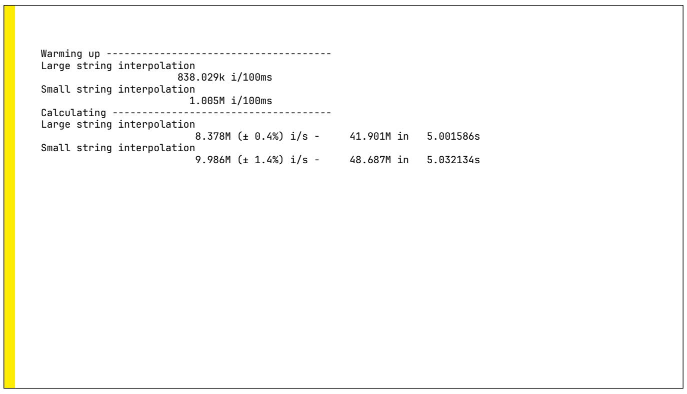
He talks about of rb_str_resurrect improving performance for small strings



```
commit 80c50308f9db813e999367ec5d116e2d2be9f840
Author: nobu <nobu@b2dd03c8-39d4-4d8f-98ff-823fe69b080e>
Date:   Sat Oct 21 23:21:05 2017 +0000

    Improve performance of string interpolation

    This patch will add pre-allocation in string interpolation.
    By this, unecessary capacity resizing is avoided.

    For small strings, optimized `rb_str_resurrect` operation is
    faster, so pre-allocation is done only when concatenated strings
    are large.  `MIN_PRE_ALLOC_SIZE` was decided by experimenting with
    local machine (x86_64-apple-darwin 16.5.0, Apple LLVM version
    8.1.0 (clang - 802.0.42)).
```

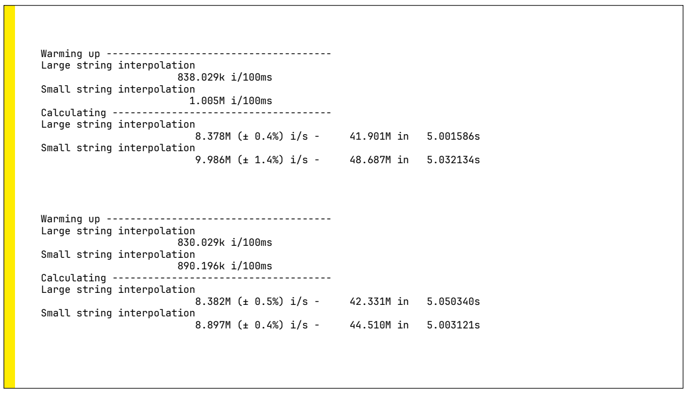
and confirms that yes, MIN_PRE_ALLOC_SIZE really is just a magic number

this was 6 years ago, maybe this optimisation isn't needed anymore.



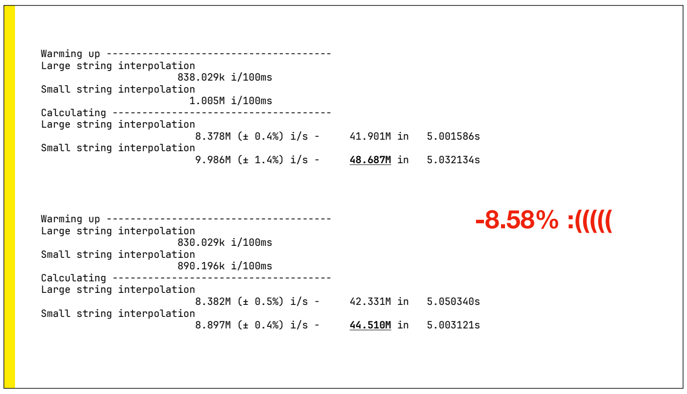
```
Warming up --------------------------------------
Large string interpolation
                       838.029k i/100ms
Small string interpolation
                         1.005M i/100ms
Calculating -------------------------------------
Large string interpolation
                          8.378M (± 0.4%) i/s -     41.901M in   5.001586s
Small string interpolation
                          9.986M (± 1.4%) i/s -     48.687M in   5.032134s
```

Thankfully Nobu included some benchmark code in the commit message, which I ran.



```
Warming up --------------------------------------
Large string interpolation
                       838.029k i/100ms
Small string interpolation
                         1.005M i/100ms
Calculating -------------------------------------
Large string interpolation
                          8.378M (± 0.4%) i/s -     41.901M in   5.001586s
Small string interpolation
                          9.986M (± 1.4%) i/s -     48.687M in   5.032134s


Warming up --------------------------------------
Large string interpolation
                       830.029k i/100ms
Small string interpolation
                       890.196k i/100ms
Calculating -------------------------------------
Large string interpolation
                          8.382M (± 0.5%) i/s -     42.331M in   5.050340s
Small string interpolation
                          8.897M (± 0.4%) i/s -     44.510M in   5.003121s
```

and then ripped out the entire optimisation, so we always pre-allocate in the correct bucket and ran it again.

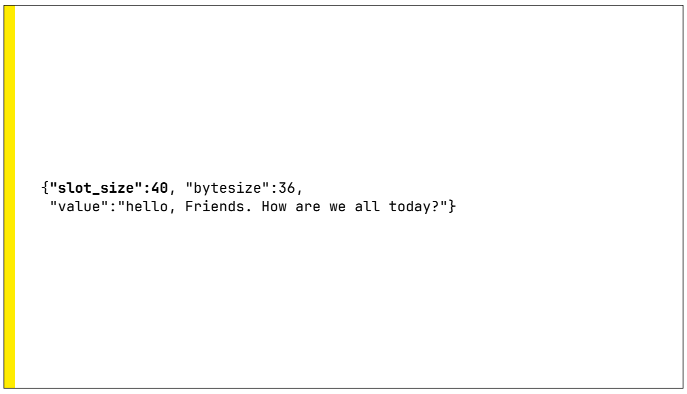


```
Warming up --------------------------------------
Large string interpolation
                       838.029k i/100ms
Small string interpolation
                         1.005M i/100ms
Calculating -------------------------------------
Large string interpolation
                          8.378M (± 0.4%) i/s -     41.901M in   5.001586s
Small string interpolation
                          9.986M (± 1.4%) i/s -     48.687M in   5.032134s


Warming up --------------------------------------
Large string interpolation
                       830.029k i/100ms
Small string interpolation
                       890.196k i/100ms
Calculating -------------------------------------
Large string interpolation
                          8.382M (± 0.5%) i/s -     42.331M in   5.050340s
Small string interpolation
                          8.897M (± 0.4%) i/s -     44.510M in   5.003121s
```

-8.58% :(((((

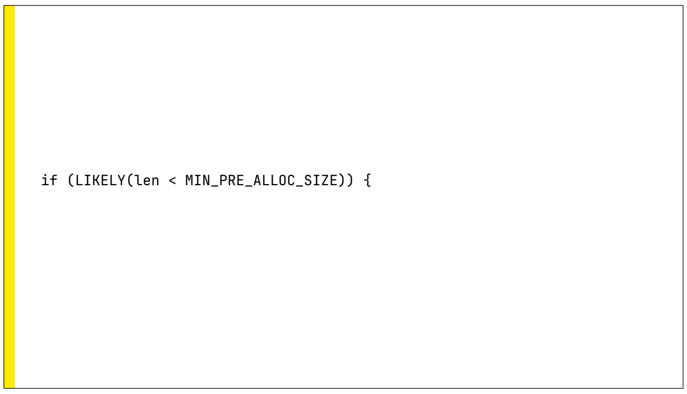
But removing the optimisation slowed down short string interpolations by nearly 9 percent.

So it looks like we still need this optimisation



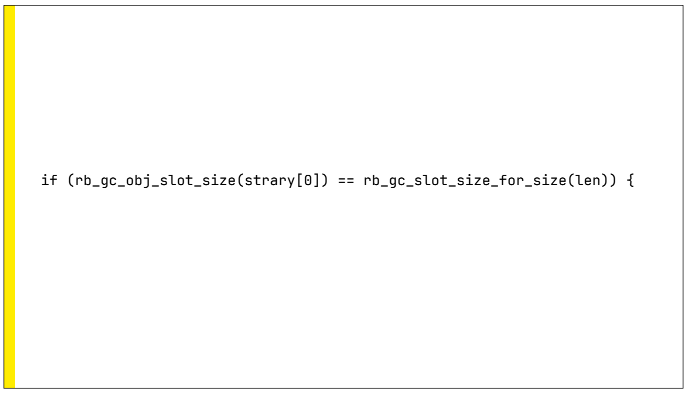
```
{"slot_size":40, "bytesize":36,
 "value":"hello, Friends. How are we all today?"}
```

It's just super frustrating that it's not creating the strings in the right place



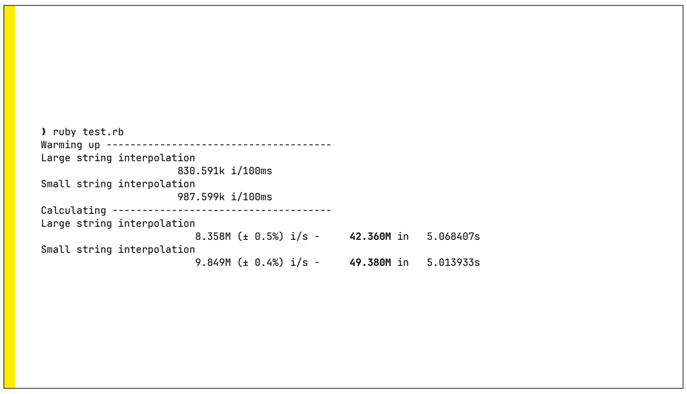
```
if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
```

So I tried to fix the bug within the scope of the optimisation



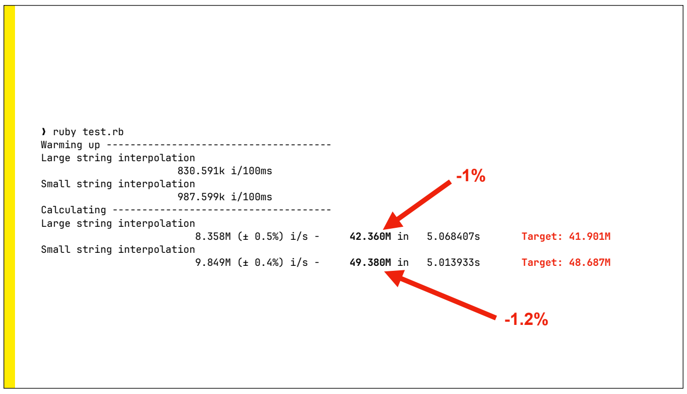
```
if (rb_gc_obj_slot_size(strary[0]) == rb_gc_slot_size_for_size(len)) {
```

I changed the conditional, so that instead of taking the optimisaed path only for strings that are shorter than the min prealloc size, we'd instead take the optimised path in the case where the final interpolated string still fits in the same size pool as the original string, to avoid the allocation penalty that comes from converting an embedded string to an extended one.



```
❯ ruby test.rb
Warming up --------------------------------------
Large string interpolation
                       830.591k i/100ms
Small string interpolation
                       987.599k i/100ms
Calculating -------------------------------------
Large string interpolation
                          8.358M (± 0.5%) i/s -     42.360M in   5.068407s
Small string interpolation
                          9.849M (± 0.4%) i/s -     49.380M in   5.013933s
```

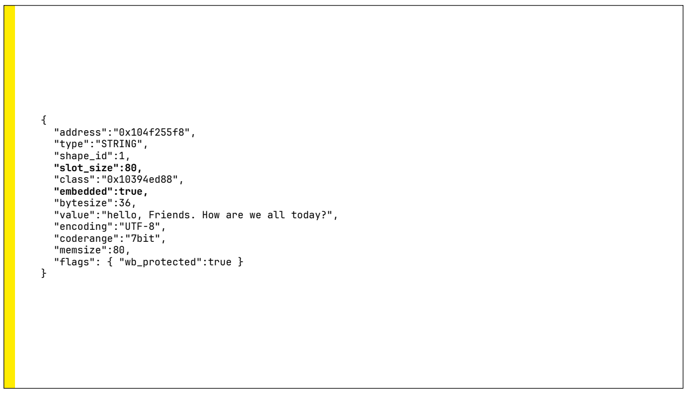
and I ran the benchmarks again



```
❯ ruby test.rb
Warming up --------------------------------------
Large string interpolation
                       830.591k i/100ms
Small string interpolation
                       987.599k i/100ms
Calculating --------------------------------------
Large string interpolation
                          8.358M (± 0.5%) i/s -     42.360M in   5.068407s       Target: 41.901M
Small string interpolation
                          9.849M (± 0.4%) i/s -     49.380M in   5.013933s       Target: 48.687M
```

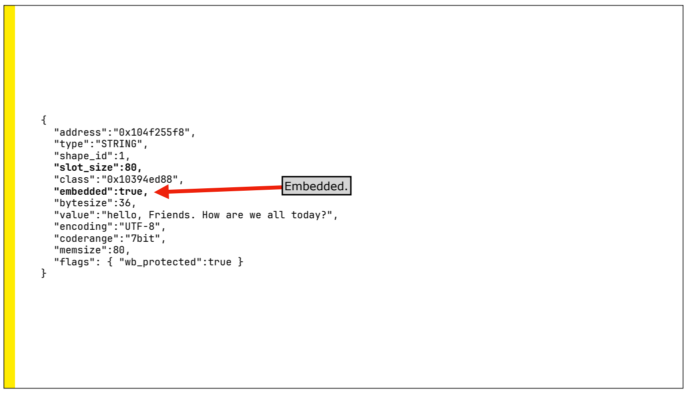
This time the performance numbers were much closer. There was still a small amount of variance, but these numbers are much closer. Nowhere near the large 9% performance drop we saw by removing the benchmark entirely

These smaller regressions could probably be explained away due benchmarking noise - I was working on my laptop at the time, as well as the subtle change we made to the optimisation condition



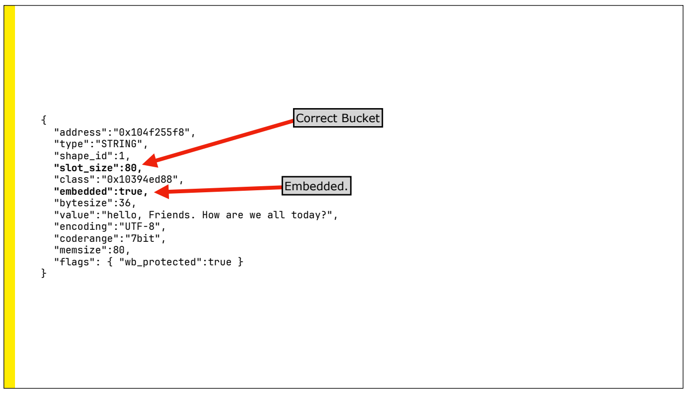
```
{
  "address":"0x104f255f8",
  "type":"STRING",
  "shape_id":1,
  "slot_size":80,
  "class":"0x10394ed88",
  "embedded":true,
  "bytesize":36,
  "value":"hello, Friends. How are we all today?",
  "encoding":"UTF-8",
  "coderange":"7bit",
  "memsize":80,
  "flags": { "wb_protected":true }
}
```

But at least our code is correct now,



```
{
  "address":"0x104f255f8",
  "type":"STRING",
  "shape_id":1,
  "slot_size":80,
  "class":"0x10394ed88",
  "embedded":true,
  "bytesize":36,
  "value":"hello, Friends. How are we all today?",
  "encoding":"UTF-8",
  "coderange":"7bit",
  "memsize":80,
  "flags": { "wb_protected":true }
}
```

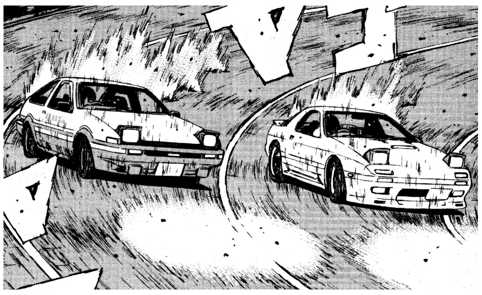
the string is embedded



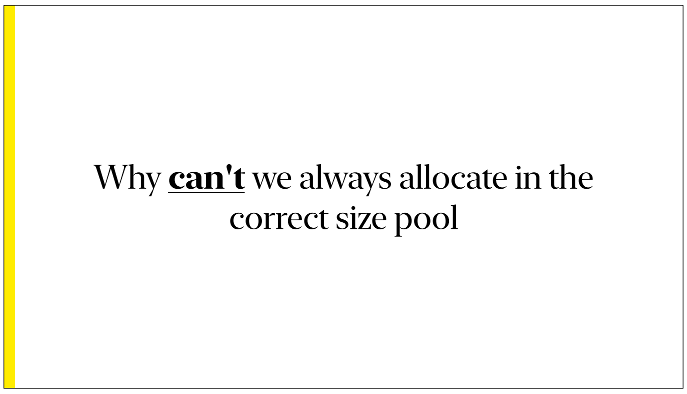
```
{
  "address":"0x104f255f8",
  "type":"STRING",
  "shape_id":1,
  "slot_size":80,
  "class":"0x10394ed88",
  "embedded":true,
  "bytesize":36,
  "value":"hello, Friends. How are we all today?",
  "encoding":"UTF-8",
  "coderange":"7bit",
  "memsize":80,
  "flags": { "wb_protected":true }
}
```

in the correct size pool, so our bug is fixed.

Time to ship right.



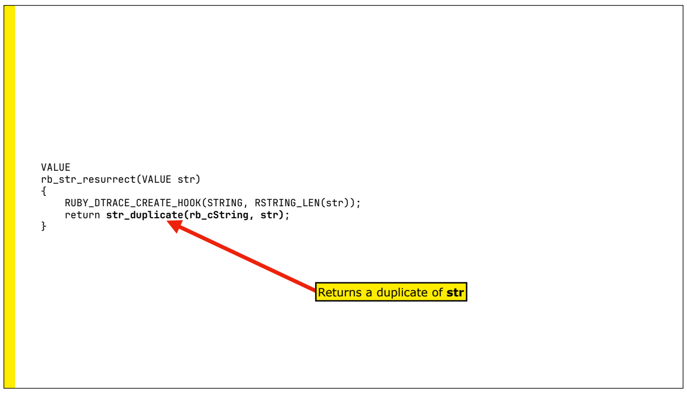
But let's not rush ahead.



Why **can't** we always allocate in the correct size pool

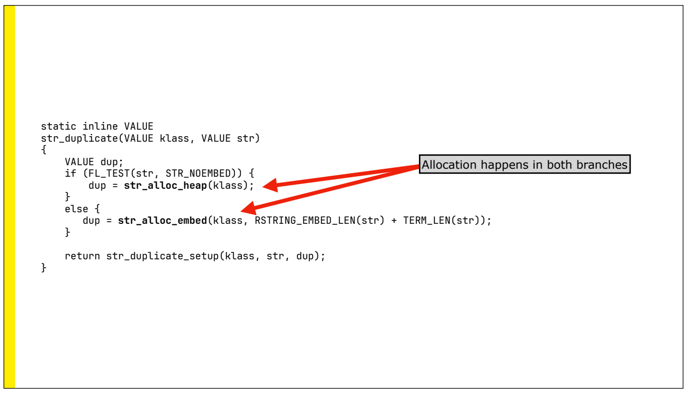
Why can't we just always allocate a new string in the correct size pool. Why does the use of rb_str_resurrect cause a 9% performance improvement, when used on arbitrarily small strings.

Despite fixing the bug, I still wasn't really satisfied with the conclusion.



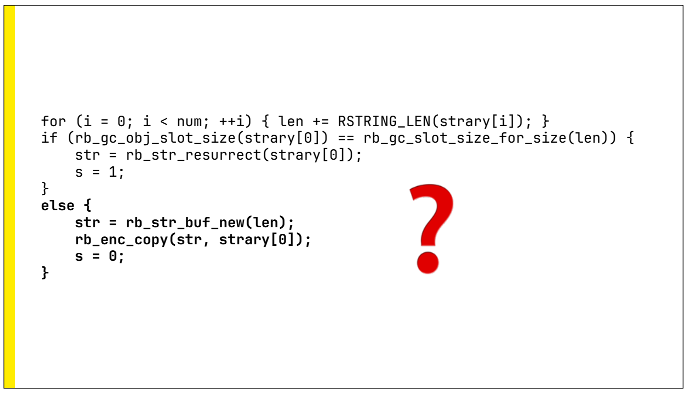
```
VALUE
rb_str_resurrect(VALUE str)
{
    RUBY_DTRACE_CREATE_HOOK(STRING, RSTRING_LEN(str));
    return str_duplicate(rb_cString, str);
}
```

especially as on closer inspection rb_str_resurrect duplicates the string



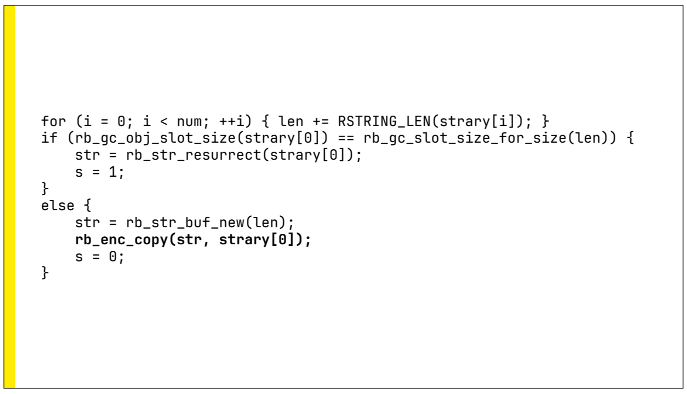
```
static inline VALUE
str_duplicate(VALUE klass, VALUE str)
{
    VALUE dup;
    if (FL_TEST(str, STR_NOEMBED)) {
        dup = str_alloc_heap(klass);
    }
    else {
       dup = str_alloc_embed(klass, RSTRING_EMBED_LEN(str) + TERM_LEN(str));
    }

    return str_duplicate_setup(klass, str, dup);
}
```

which does an allocation, so it can't be allocation that's slowing us down, which was my initial hypothesis for the slowdown.



```
for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
if (rb_gc_obj_slot_size(strary[0]) == rb_gc_slot_size_for_size(len)) {
    str = rb_str_resurrect(strary[0]);
    s = 1;
}
else {
    str = rb_str_buf_new(len);
    rb_enc_copy(str, strary[0]);
    s = 0;
}
```

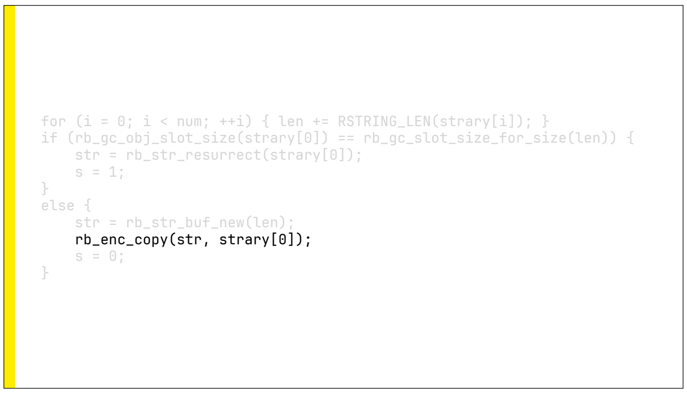
So I took a closer look at the allocation path



```
for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
if (rb_gc_obj_slot_size(strary[0]) == rb_gc_slot_size_for_size(len)) {
    str = rb_str_resurrect(strary[0]);
    s = 1;
}
else {
    str = rb_str_buf_new(len);
    rb_enc_copy(str, strary[0]);
    s = 0;
}
```

And noticed that it seems to be doing some extra work compared to the string resurrect path.

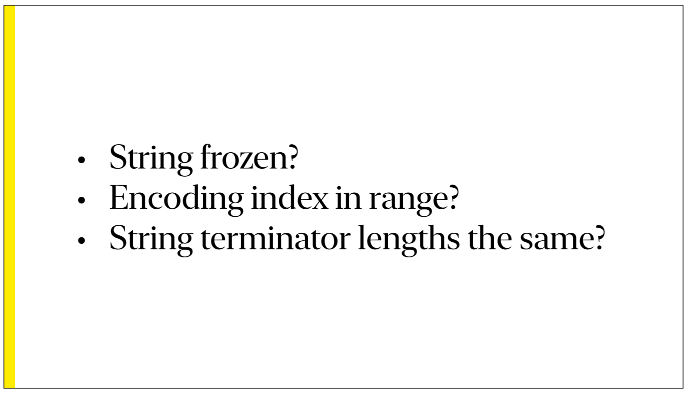
Once the new string is allocated, we copy the encoding from the string at the head of the array into the new string we've just created.



```c
for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
if (rb_gc_obj_slot_size(strary[0]) == rb_gc_slot_size_for_size(len)) {
    str = rb_str_resurrect(strary[0]);
    s = 1;
}
else {
    str = rb_str_buf_new(len);
    rb_enc_copy(str, strary[0]);
    s = 0;
}
```

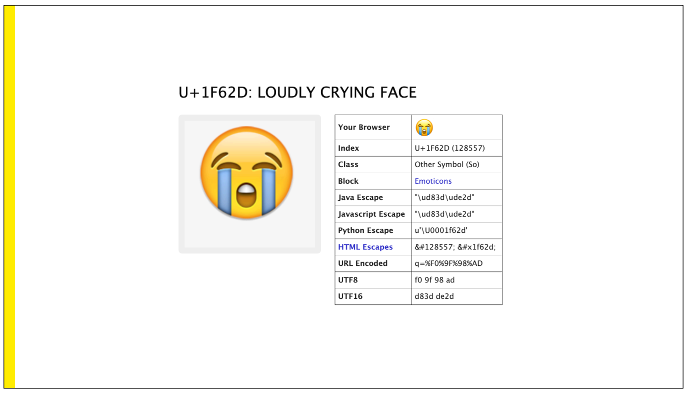
But to do that, it uses a public function, called rb_enc_copy, from Ruby's C API.
Because this is public, it can be used from C extensions. Which means that effectively, this function has to deal with user input, it cannot make assumptions about the validity of it's parameters.

And to that end it does a lot of safety checks on the String that's being passed in.



- String frozen?
- Encoding index in range?
- String terminator lengths the same?

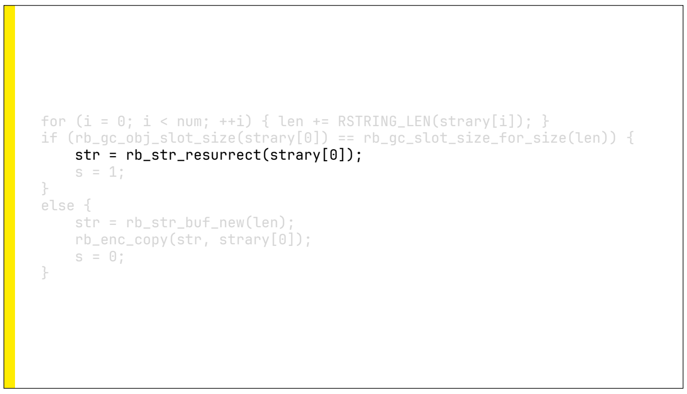
Things like checking whether the string is frozen, the encoding index is in the correct range, or that the terminator lengths are the same.



U+1F62D: LOUDLY CRYING FACE

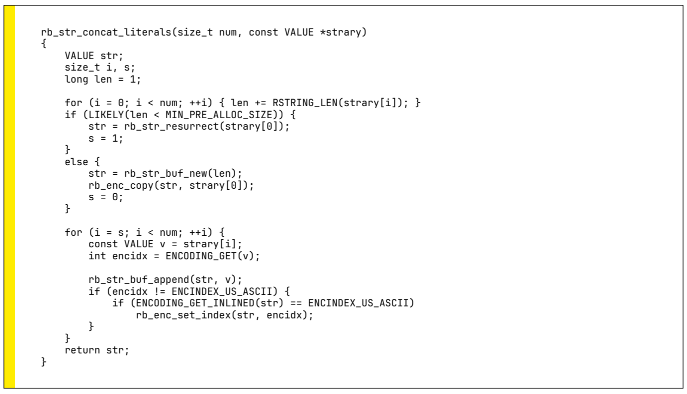
Your Browser	😭
Index	U+1F62D (128557)
Class	Other Symbol (So)
Block	Emoticons
Java Escape	"\ud83d\ude2d"
Javascript Escape	"\ud83d\ude2d"
Python Escape	u'\U0001f62d'
HTML Escapes	&#128557; &#x1f62d;
URL Encoded	q=%F0%9F%98%AD
UTF8	f0 9f 98 ad
UTF16	d83d de2d

I benchmarked the code again, and sure enough encoding, or rather the method of encoding, was exactly our problem.



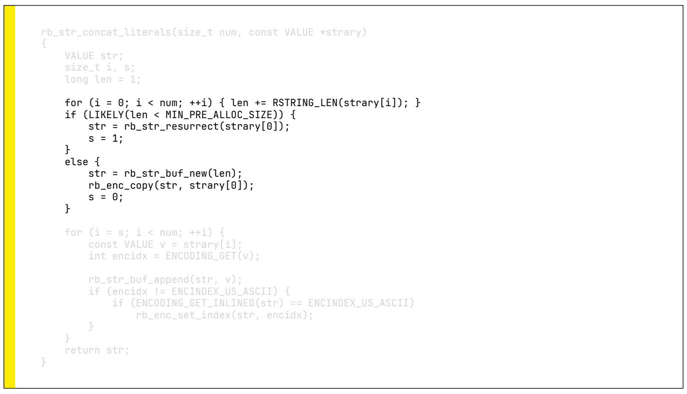
```
for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
if (rb_gc_obj_slot_size(strary[0]) == rb_gc_slot_size_for_size(len)) {
    str = rb_str_resurrect(strary[0]);
    s = 1;
}
else {
    str = rb_str_buf_new(len);
    rb_enc_copy(str, strary[0]);
    s = 0;
}
```

the string resurrection code skips safety checks when it sets up the encoding of the duplicated string because it knows exactly what it's duplicating from and into.



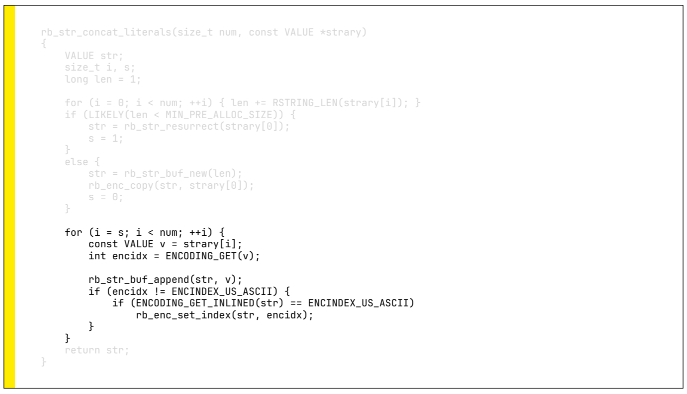
```c
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

But we don't need these safety checks in the string concat literals function



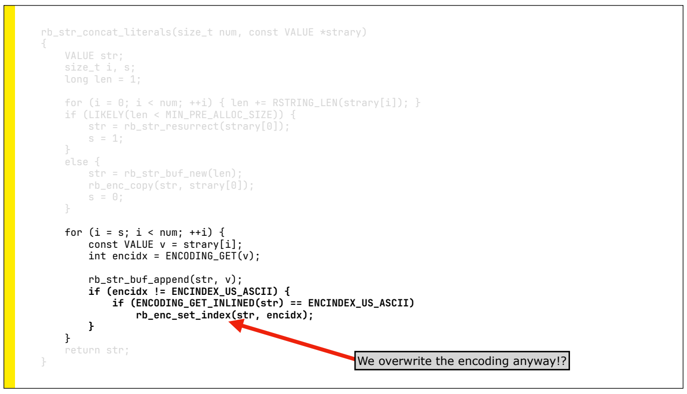
```c
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

Because after we've set up our target string,



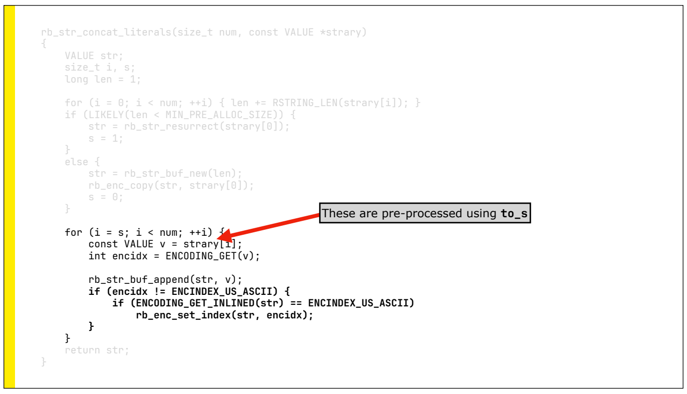
```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

and moved into our loop appending into this new string



```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

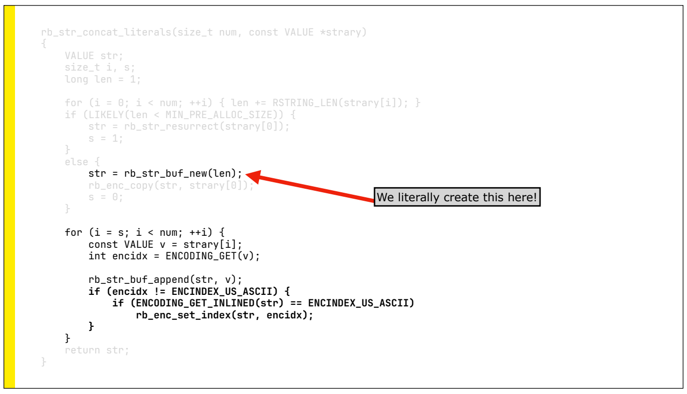
In most cases we just overwrite the encoding of the final string anyway.



```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

These are pre-processed using **to_s**

and as the strings we're looping over have already been pre-processed (by calling to_s on them), we can be confident that they have passed these checks.

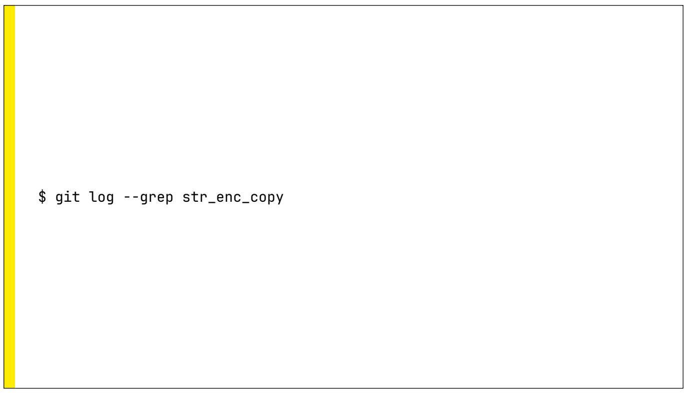


```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

We literally create this here!

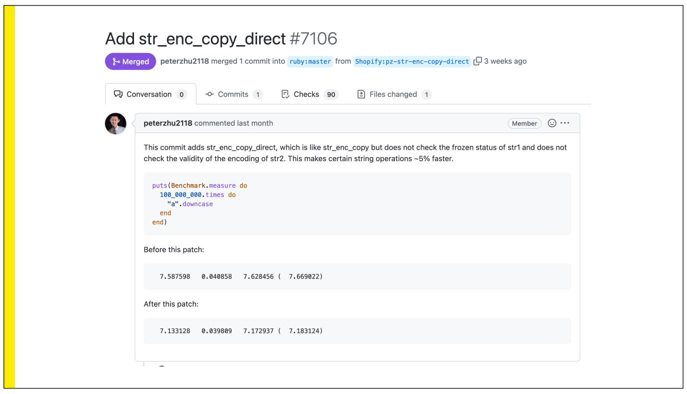
Additionally our destination string is a also a completely known quantity, we literally just created it in this function

we can afford to be less strict with our safety check



```
$ git log --grep str_enc_copy
```

When I went to research about the history of this function



Add str_enc_copy_direct #7106

peterzhu2118 merged 1 commit into `ruby:master` from `Shopify:pz-str-enc-copy-direct` 3 weeks ago

peterzhu2118 commented last month

This commit adds str_enc_copy_direct, which is like str_enc_copy but does not check the frozen status of str1 and does not check the validity of the encoding of str2. This makes certain string operations ~5% faster.

```
puts(Benchmark.measure do
  100_000_000.times do
    "a".downcase
  end
end)
```

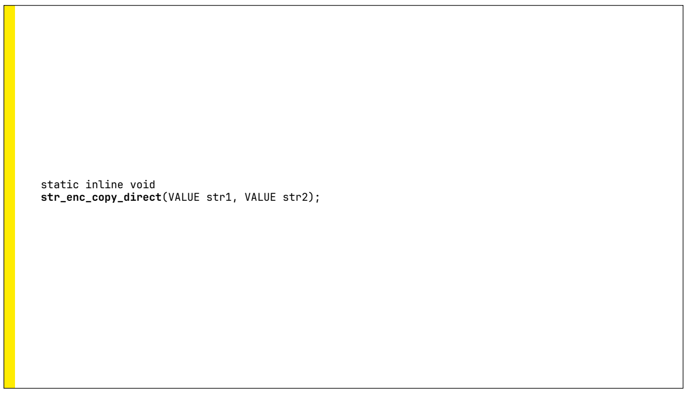
Before this patch:

```
  7.587598   0.040858   7.628456 (  7.669022)
```

After this patch:

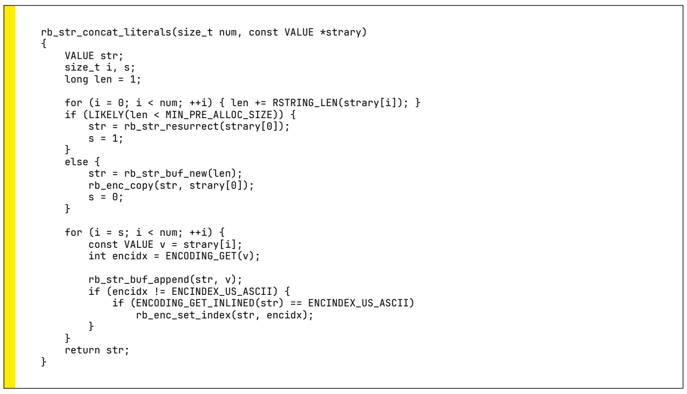
```
  7.133128   0.039809   7.172937 (  7.183124)
```

I found that, serendipitously, Peter had already run into a case where these safety checks had impacted performance of other string functions just a few days earlier.



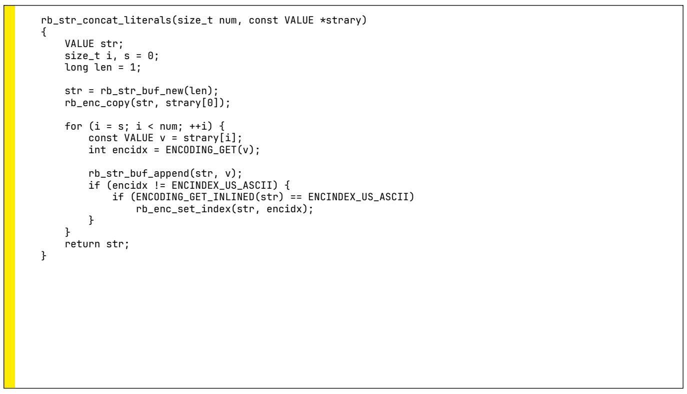
```
static inline void
str_enc_copy_direct(VALUE str1, VALUE str2);
```

And his solution was to introduce a private function str_enc_copy_direct, which sets up the encoding, but skips all the safety checks that are part of the public api, so it's much faster.



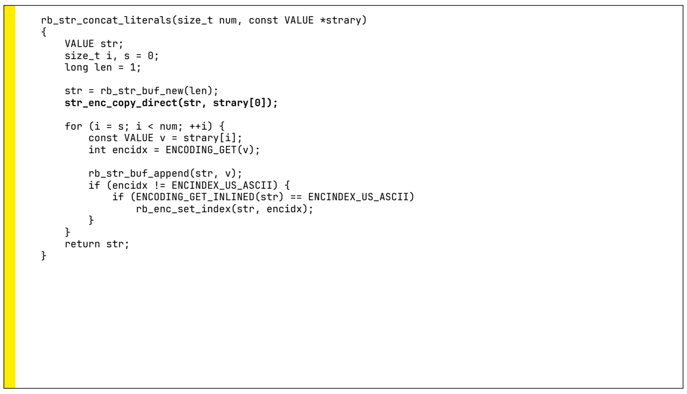
```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s;
    long len = 1;

    for (i = 0; i < num; ++i) { len += RSTRING_LEN(strary[i]); }
    if (LIKELY(len < MIN_PRE_ALLOC_SIZE)) {
        str = rb_str_resurrect(strary[0]);
        s = 1;
    }
    else {
        str = rb_str_buf_new(len);
        rb_enc_copy(str, strary[0]);
        s = 0;
    }

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

so let's try this refactor again.



```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s = 0;
    long len = 1;

    str = rb_str_buf_new(len);
    rb_enc_copy(str, strary[0]);

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

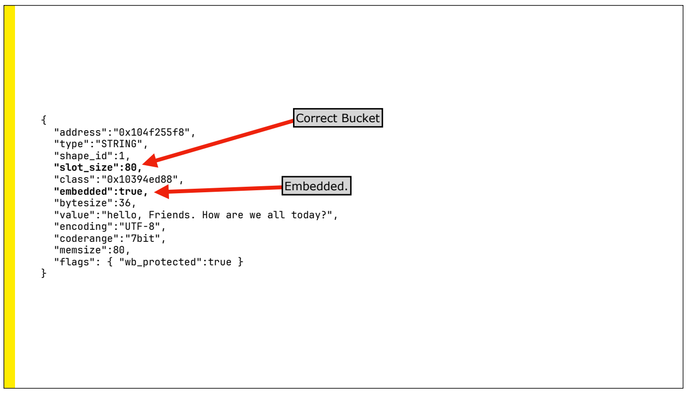
We'll rip the whole conditional out, so that we always pre-allocate, this gets rid of our magic number.



```
rb_str_concat_literals(size_t num, const VALUE *strary)
{
    VALUE str;
    size_t i, s = 0;
    long len = 1;

    str = rb_str_buf_new(len);
    str_enc_copy_direct(str, strary[0]);

    for (i = s; i < num; ++i) {
        const VALUE v = strary[i];
        int encidx = ENCODING_GET(v);

        rb_str_buf_append(str, v);
        if (encidx != ENCINDEX_US_ASCII) {
            if (ENCODING_GET_INLINED(str) == ENCINDEX_US_ASCII)
                rb_enc_set_index(str, encidx);
        }
    }
    return str;
}
```

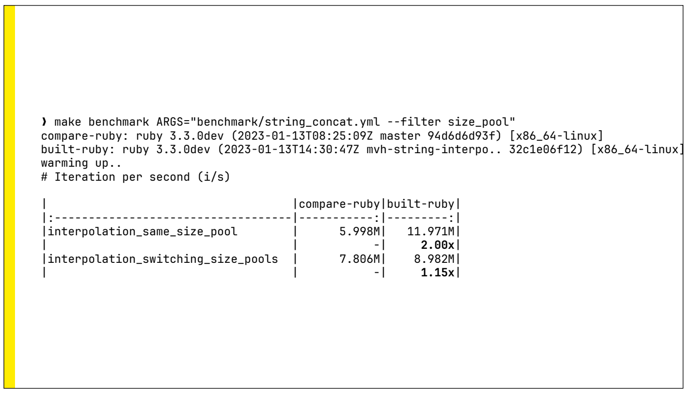
but this time we'll use the less safe but faster internal api for encoding

And I like this code a lot better now. The complexity is reduced, the magic number is gone, and in my opinion at least, it's easier to read and reason about.



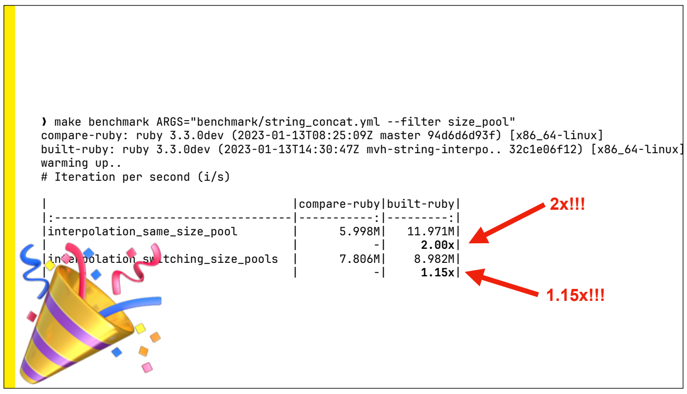
```
{
  "address":"0x104f255f8",
  "type":"STRING",
  "shape_id":1,
  "slot_size":80,
  "class":"0x10394ed88",
  "embedded":true,
  "bytesize":36,
  "value":"hello, Friends. How are we all today?",
  "encoding":"UTF-8",
  "coderange":"7bit",
  "memsize":80,
  "flags": { "wb_protected":true }
}
```

This fixes the original bug, so interpolated strings will always be embedded in the correct sized bucket.



```
❱ make benchmark ARGS="benchmark/string_concat.yml --filter size_pool"
compare-ruby: ruby 3.3.0dev (2023-01-13T08:25:09Z master 94d6d6d93f) [x86_64-linux]
built-ruby: ruby 3.3.0dev (2023-01-13T14:30:47Z mvh-string-interpo.. 32c1e06f12) [x86_64-linux]
warming up..
# Iteration per second (i/s)

|                                    |compare-ruby|built-ruby|
|:-----------------------------------|-----------:|---------:|
|interpolation_same_size_pool        |      5.998M|   11.971M|
|                                    |           -|     2.00x|
|interpolation_switching_size_pools  |      7.806M|    8.982M|
|                                    |           -|     1.15x|
```

We took Nobu's original benchmark and incorporated it into the CRuby benchmark suite, to make it more repeatable and consistent, And when we compared this new patch with the original code we saw that



```
❱ make benchmark ARGS="benchmark/string_concat.yml --filter size_pool"
compare-ruby: ruby 3.3.0dev (2023-01-13T08:25:09Z master 94d6d6d93f) [x86_64-linux]
built-ruby: ruby 3.3.0dev (2023-01-13T14:30:47Z mvh-string-interpo.. 32c1e06f12) [x86_64-linux]
warming up..
# Iteration per second (i/s)

|                                   |compare-ruby|built-ruby|
|:----------------------------------|-----------:|---------:|
|interpolation_same_size_pool       |      5.998M|   11.971M|
|                                   |           -|     2.00x|
|interpolation_switching_size_pools |      7.806M|    8.982M|
|                                   |           -|     1.15x|
```

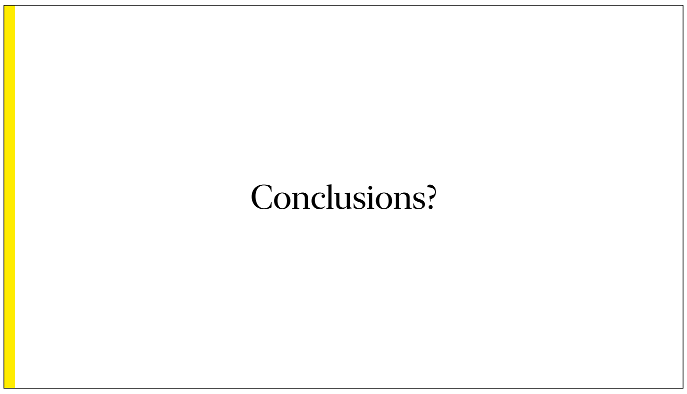
we've doubled the performance of interpolations within the same bucket, and improved the performance of interpolations that cross a bucket. This is a fantastic result.

We removed 8 lines of code

and all this by removing 8 lines of code

❤️ Ruby 3.3

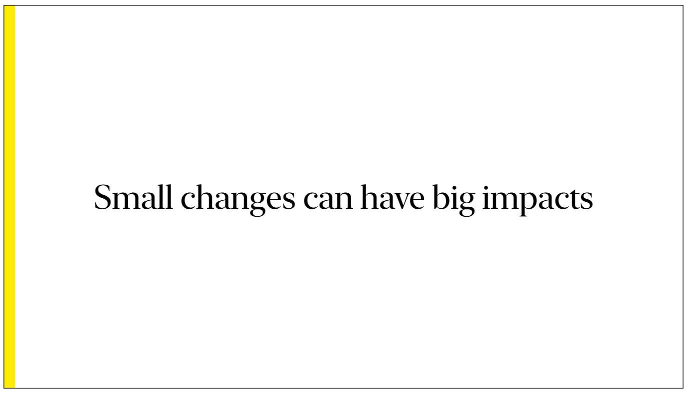
And these improvements made their way into Ruby 3.3, and now the string interpolation code is faster and cleaner than before.



Conclusions?

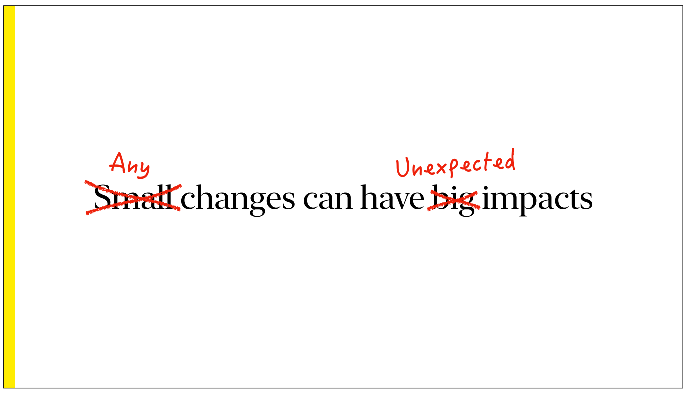
I hope you enjoyed this story. I'd love to tell you that I have some groundbreaking conclusions to share, but I don't really. This was just a debugging war story, that I had a load of fun with.

But, that being said, I'd still like to leave you with a few thoughts, followed by a Thank you.



Small changes can have big impacts

The first is that small changes can often have big impacts. This applies to our little patch certainly, I wasn't expecting the removal of 8 lines of code to result in a 2x speedup.



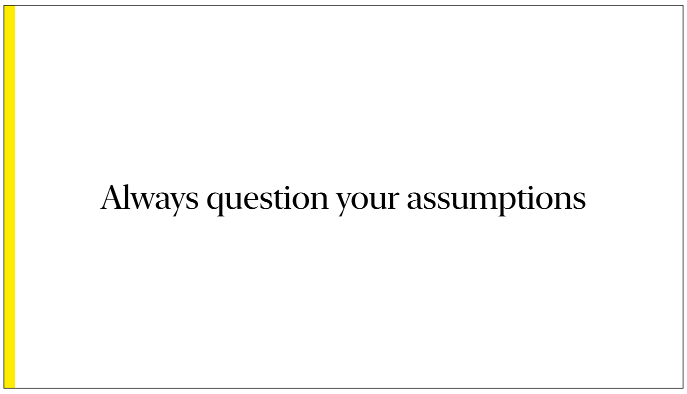
Any ~~Small~~ changes can have Unexpected ~~big~~ impacts

But the real message here is that any change can have unexpected impacts.

When we initially merged the Variable Width Allocation work originally, we hadn't anticipated that a 7 year old optimisation would be responsible for a bug in a very specific part of string interpolation and that removing the optimisation would have such huge impact.

Unexpected impacts like this, are not something that can necessarily be mitigated, but instead we should treat them as a learning opportunity.

So I'd like to give you some unsolicited advice.

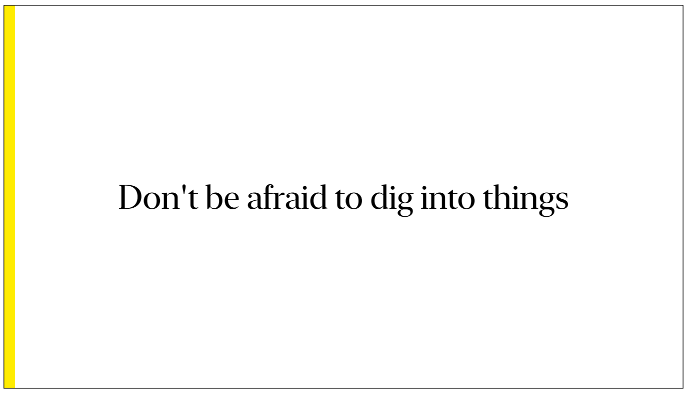


Always question your assumptions

Which is, always question your assumptions. It would have been very easy for me to have stopped at the first step here. Simply assuming, that because I'd seen the massive performance jump in the optimisation path, that it was absolutely still relevant and that all I needed to do was to make the smallest change in order to fix my specific bug.

But at this point I still didn't have an answer to my question: why couldn't we allocate the correct size string up front.

Which leads on to my next point



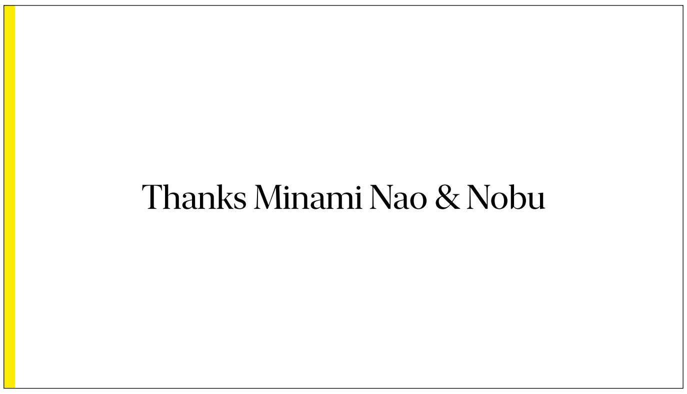
Don't be afraid to dig into things

Don't be afraid to dig into things you don't understand yet, or can't explain. Follow your curiosity and see where it leads.

I probably would have stopped at the first change if I didn't have a burning desire to answer my question, why couldn't we just allocate the correct size thing up front.

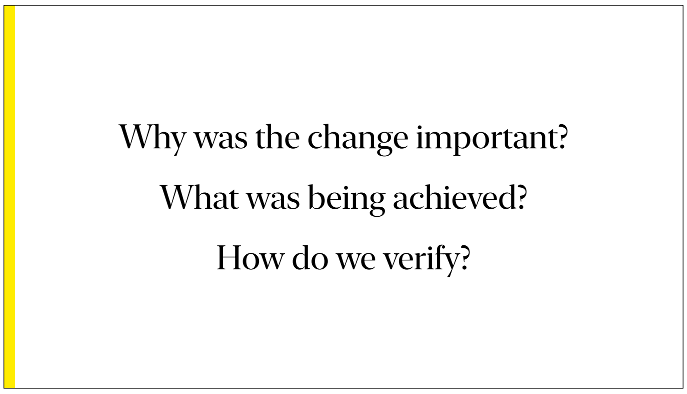
And it turns out that we can, we just needed to think about things in a slightly different way to the author of the original optimisation.

And this diversity of thought is valuable. The more people inquiring minds who are questioning assumptions and delving into things they don't understand will result in more opportunities to make positive changes.


Thanks Minami Nao & Nobu

Now I want to say a big Thank You. First to Minami Nao, the author of the original optimisation for writing such a detailed commit message.

And also to Nobu, for preserving it. For not squaring the commits down during the merge, and instead preserving that excellent documentation.



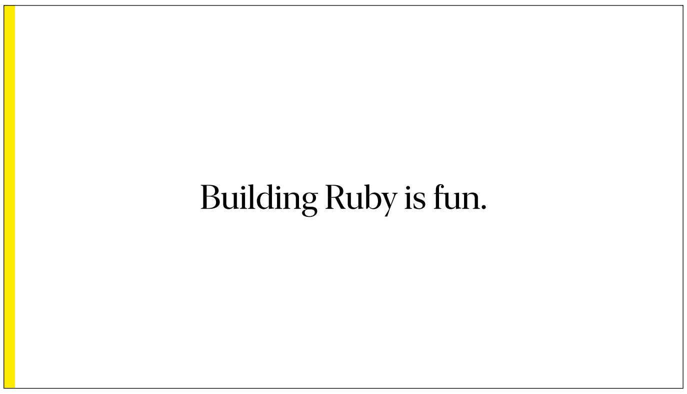
Why was the change important?

What was being achieved?

How do we verify?

Seriously, This change would have been a lot more arduous to make and test were it not for those detailed notes. Minami san summarised exactly why the change was important, what he was trying to achieve, and perhaps most importantly, how he verified that it worked.

As someone coming into this code for the first time, and trying to decipher why it was behaving oddly, this was worth it's weight in gold.



Building Ruby is fun.

So, what I'm trying to say really is - Working on Ruby is a lot of fun. It's rewarding and it can teach you a lot. And I would really encourage anyone in the audience who hasn't tried contributing to Ruby before to get out there, explore the codebase, experiment, and to learn.

And above all else have fun.

Thanks! ❤️

And that's all I've got, thanks for listening.