Perl5 Memory Manglement

Your friendly introduction to a wastrel.

What it is

- From the inside it's how Perl utilizes the memory available to it on the machine.
  - Today I'll look at Scalars, Hashes, & Arrays.
- From the outside it's how you design the code to handle how Perl deals with things.
  - Controlling the lifespan of data.
  - Choosing the right structures.

Why you care

- Long lived, high-volume, or high-speed app's need to avoid swapping, heap fragmentation, or blowing ulimits.
- Examples are ETL, Bioinformatics, or long-lived servers.
- Non-examples are *JAPH* or WWW::Mechanize jobs that snag one page.

PerlGuts & Perl Data Intro

- The standard introduction to Perl's guts is “perlguts” (use perldoc or view it as HTML on CPAN).
- Perldoc also includes “perldebuguts” which shows how to use the debugger to look around for yourself.
- Simeon Cozens' Intro to Perl5 Internals is avaialable online at:

  `http://www.faqs.org/docs/perl5int/`

A view from perldebuguts

Perl is a profligate wastrel when it comes to memory use. There is a saying that to estimate memory usage of Perl, assume a reasonable algorithm for memory allocation, multiply that estimate by 10, and while you still may miss the mark, at least you won't be quite so astonished. This is not absolutely true, but may provide a good grasp of what happens.

Anecdotal estimates of source-to-compiled code bloat suggest an eightfold increase. This means that the compiled form of reasonable (normally commented, properly indented etc.) code will take about eight times more space in memory than the code took on disk.

First a bit of history...

- In the beginning there was assembly, and it was good – but it wasn't portable.
- Then there were A, B, and...
- C, which has been used for the guts of nearly every language since – including Perl.
- Perl uses C for its memory management.

C's Memory: Stack & Heap

- C memory is divided into Stack & Heap.
- Stack is automatically used and returnd for each function call.
- Heap is persistent and managed by the programmer.
    - malloc adds to the heap and returns a pointer to allocated memory.
    - realloc resizes an allocation in the heap.
    - free returns it to the O/S from the heap.

Perly Memory

- Perl's model is similar to C:
  - subrotine calls have a “scratchpad” (“PAD”)
  - variables are allocated in the heap.
- Perl uses malloc and realloc for its heap.
  - Notice I didn't mention free: *Perl only grows!*
- Perl variables are pointers in the PAD to memory on the heap.

Bugaboo of Heap: *Fragmentation*

- You can have half the heap free, but it may be in chunks too small for what you need.
- Often happens when allocations grow over time and need to be copied into larger chunks, leaving smaller ones behind.
- Perl's tendency to trade space for speed can fragment the existing heap and require extending it.

Perl5 Peep Show

- Devel::Peek displays the internals of an item.
- It is really handy when used with the Perl debugger: you can set things and eyeball the consequences easily.
- Devel::Size is also handy for showing just the memory footprint of an item.
- Both can be added to #! code or modules.

A View from the Perly Side

- Perl has – pretty much – scalars, arrays, and hashes.
- Scalars do all the work of simple data types in C and are managed in the Perl code via “SV” structures (leaving out Magic for the moment).
- Arrays are lists of scalars.
- Hashes are arrays of arrays.

Aside: NULL vs NUL

- NUL is an ACSII character with all zero bits used to terminate strings in C.
- NULL is a pointer value that will never compare true to a valid pointer.
    - It is used to check whether memory is allocated.
    - It may not be zero.
    - It is not used to terminate strings.

Scalars: “SV”

- Because SV's can handle text, numeric, and pointer values their perlguts are fairly detailed.
- For memory management the main issues are garbage collection (reference counting) and handling text.

Text

- Strings are allocated as C-style strings, with a list of characters followed by a NUL.
- They can grow by being extended or re-allocated.
- But they don't shrink the way you might expect.

Start Pointer & Length

- Amost anything in computing has a tradeoff between space and speed: small is slow, fast is big.
- Perl always trades space for speed.
- For example: Removing a character from the start of a string does't free up any memory, it just moves a 'start of string' pointer in the SV and reduces the length by one.

Example: filling memory

```
# trivial example of an off-by-one error
# that swallows memory.
# each of the strings on @text fills
# 1_000_001 char's, even if most of is
# unused!

my @text = ();
for( 1 .. 1_000_000 )
{
  my $a  = 'a' x 1_000_000;
  substr $a, 0, 1, '' for 1 .. 999_999;
  push @text, $a
}
```

```
$ perl -MDevel::Peek -Mdevel::Size -d -e 42;

DB<1> $a = ''
DB<2> Dump \$a

  SV = PV(0x8371fd8) at 0x82035d8
    REFCNT = 2                \$a upped the ref count
    PV = 0x834a550 ""\0       Notice trailing NUL
    CUR = 0                   0 offset to end
```

Using Devel::Size to look at the same variable, the empty string has a 36-byte memory footprint:

```
DB<3> p total_size $a;
36                           Even the empty string
                             takes up space.
```

Assigning a string to the variable allocates more space:

```
DB<4> $a = 'a' x 8
DB<5> Dump \$a

  SV = PV(0x8371fd8) at 0x82035d8

    PV = 0x834a550 "aaaaaaaa"\0
    CUR = 8     8 char's of data
```

The size goes from 36 -> 44 bytes with the addition of 8 chars:

```
DB<6> p total_size $a
44
```

Removing a leading character does not free up any space.
The offset is increased by one, ignoringo one character (“x” .):

```
DB<7> substr $a, 0, 1, ''
DB<8> Dump \$a
DB<9>

  SV = PVIV(0x804e240) at 0x82035d8

    IV = 1  (OFFSET)             offset to start
    PV = 0x834a551 ( "x" . ) "aaaaaaa"\0
    CUR = 7                      length $a


<10> p total_size $a
48                               No change in size!
```

Try this a few more times and you end up with an offset of 8: 7 unused chars, and a single character returned in the string. The SV still contains the full 8 char's, but only one of them is accessable from perl in $a.

```
  IV = 8  (OFFSET)
  PV = 0x834a558 ( "xxxxxxxx" . ) "x"\0
  CUR = 1
```

And the variable's size has not changed:

```
DB<12> p total_size $a
48
```

Splicing off the end of a string does not re-allocate the memory: the NUL is moved down, but the allocated size does not change:

```
DB<40> $a = 'a' x 8

DB<41> Dump $a
SV = PV(0x8372b70) at 0x82028b0
  PV = 0x834b198 "aaaaaaaa"\0
  CUR = 8

DB<42> substr $a, 1, length $a, '';
DB<43> Dump $a

SV = PV(0x8372b70) at 0x82028b0
  PV = 0x834b198 "a"\0
  CUR = 1

DB<43> p total_size $a
48
```

In fact, there isn't anything you can do to make the SV release its space:

```
DB<58> $f = ''
DB<59> p total_size $f
36
DB<60> $f = 'x' x 1024
DB<61> p total_size $f
1060
DB<62> substr $f, 512, 512, '' half of it is empty
DB<63> p total_size $f
1060
DB<64> $f = ''                   all of it is empty
DB<65> p total_size $f
1060
```

This can be useful in cases where you have to re-allocate a large string: just allocate a single varible once and use it as a buffer. Just don't expect to chew through parts of a string from either end and recover the space.

The way to get your memory back is to assign the string to a new SV:

```
DB<66> $i = 'x' x 1024
DB<67> substr $i, 512, 512, ''
DB<68> $j = $i
DB<69> p total_size $i
1060
DB<70> p total_size $j
548
```

For example, read into a fixed buffer then assign the

```
my $buffer = '';
while( $buffer = <$in> )
{
   # mangle the buffer as necessary, then assign it:

   my $post = $buffer;
}
```

Takeaway

- Strings don't shrink.
- Use lexicals to return memory.
- Clean variables up *before* assigning them:

```
my $buffer    = <$fh>;
$buffer =~ s{ $rx }{}gx;
$data{ $key } = $buffer;
```

Arrays are similar to text

- They are reduced by adding a skip count to the array and reducing the length.
- This can lead to all sorts of confusion if you try to 'free up space' by shifting data off of an array.

Empty arrays don't take up any extra space:

```
DB<14> @a = ()
DB<15> Dump \@a

  SV = PVAV(0x8386a08) at 0x8325d68

    ARRAY = 0x0
    FILL = -1
    MAX = -1
    ARYLEN = 0x0                 OK, it's empty...


DB<17> p total_size \@a
100                              But it still
                                 takes up space
```

Adding to the array puts more SV's onto the arrays list:

```
DB<16> @a = ('a'.. 'h' );
DB<17> Dump \@a

    ARRAY = 0x838ae00
    FILL = 7
    MAX = 7
    ARYLEN = 0x0
    SV = PV(0x8371fe8) at 0x8388fa8     Stringy SV...
      REFCNT = 1
      FLAGS = (POK,pPOK)
      PV = 0x8379340 "a"\0             With an 'x' and NUL...
      CUR = 1
      LEN = 4
    Elt No. 1
    SV = PV(0x8371f28) at 0x8389258    And another SV

DB<18> p total_size \@z
420                                    about 50 bytes/entry
```

Splicing off the front of the list does not free any memory:

```
DB<18> splice @a, 0, 7, ();
DB<19> Dump \@a

    ARRAY = 0x838ae1c (offset=7)  offset, just like strings
    ALLOC = 0x838ae00
    FILL = 0
    MAX = 0
    ARYLEN = 0x0
    FLAGS = (REAL)
    Elt No. 0
    SV = PV(0x80ab908) at 0x838e2e0
      REFCNT = 1
      FLAGS = (POK,pPOK)
      PV = 0x837b5d0 "a"\0
```

You recover space from SV's stored in an array, not the SA itself:

```
DB<82> @z = ()
DB<84> p total_size \@z
132                                 empty array
DB<85> @z = ( 'z' ) x 1024
DB<86> p total_size \@z
45140                               1024 text SV's
DB<87> splice @z, 512, 512, ()
DB<88> p total_size \@z
26708                               only 512 text SV's
DB<89> splice @z, 0, 512, ()
DB<90> p total_size \@z
8276                                no SV's, just the SA
DB<90> @y = @z
DB<91> p total_size \@y
100                                 new var is empty
```

What about about queues?

- A classic arrangement is to shift work off the front, pushing new tasks onto the array.
- Which works fine, but doesn't do anything to recover any space unless the array is emptied completely by the shifts.
- New items are pushed onto the end, regardless of unsed initial space.

Net result: shift+push on a non-empty array leaves you with a larger array.

```
  DB<11> p total_size \@a
340

 DB<12> shift @a;

 DB<13> p total_size \@a
304

 DB<14> push @a, 'z'

 DB<15> p total_size \@a
364
```

So, a queue will eventually reach a steady state, but it will be larger the initial:

```
my @a = ( 'a' .. 'f' );
print
"Start: " . ( total_size \@a );
for( 'g' .. 'z'  )
{
   shift @a;
   push @a, $_;
   print ( total_size \@a );
}
```

```
Start:   340
Add g:   364
Add h:   364
Add i:   364
Add j:   364
Add k:   364
Add l:   364
Add m:   364
Add n:   428
Add o:   428
Add p:   428
Add q:   428
Add r:   428
Add s:   428
Add t:   428
Add u:   428
Add v:   428
Add w:   428
Add x:   428
Add y:   428
Add z:   428
```

When the array needs to be re-allocated then only its active portion is copied to the new start address:

```
$a = [ 1 .. 1_000_000 ];
print '0', total_size $a;
for ( 1 .. 1_000_000 )
{
   shift @$a;
   push @$a, $_;
   print $_, total_size $a
}

0     20000100
1     24388692
...
```

Takeaway

- Like strings:
  - Arrays don't shrink.
  - Lexicals return their space.
- You can recover space from values stored *in* the array, but the SA structure itself only grows.
- Clean up the array and then assign it to a new variable.

Hashes

- Hashes are composed of a bucket list and collision chains of arrays (array of arrays.
- You get 8 buckets with the hash structure, even if it is empty.

```
DB<20> %a = ()
DB<21> Dump \%a

  SV = PVHV(0x82dda9c) at 0x8325340
    FLAGS = (SHAREKEYS)
    ARRAY = 0x0
    KEYS = 0
    FILL = 0
    MAX = 7
    RITER = -1
    EITER = 0x0

DB<22> p total_size \%a
76
```

```
DB<22> @a{ ('a' .. 'z' ) } = ()
DB<23> Dump \%a

    FLAGS = (OOK,SHAREKEYS)
    ARRAY = 0x838a8e8  (0:13, 1:12, 2:7)    bucket use: 13 w/0, 12 w/1, 7 w/ 2
    hash quality = 115.8%                   uneven distribution
    KEYS = 26                               keys in use
    FILL = 19                               buckets filled (12 + 7)
    MAX = 31
    RITER = -1
    EITER = 0x0
    Elt "w" HASH = 0x58a0b120               collision chain == array
    SV = NULL(0x0) at 0x838e5a0             remember that arrays do not give
      REFCNT = 1                            back all of their space!
      FLAGS = ()
    Elt "r" HASH = 0x26014be2
    SV = NULL(0x0) at 0x838e4d0
      REFCNT = 1
      FLAGS = ()
    Elt "a" HASH = 0xca2e9442
    SV = NULL(0x0) at 0x83890f8
      REFCNT = 1
      FLAGS = ()
DB<24> p total_size \%a
1186                                        Keys only! ~ 45bytes/letter
```

```
DB<99> delete @a{ 'a' .. 'm' }
DB<100> Dump \%a

      ARRAY = 0x840a638  (0:19, 1:13)
      hash quality = 175.0%
      KEYS = 13
      FILL = 13
      MAX = 31

      Elt "w" HASH = 0x58a0b120
      SV = NULL(0x0) at 0x83ee3d8
        REFCNT = 1
      Elt "r" HASH = 0x26014be2

DB<101> p total_size \%a
679
DB<102> %a = ()
DB<103> p total_size \%a
172
```

the SV's used for keys go away, but the final size is still larger than the original hash. incl. arrays for chains.

Takeaway

- Hashes are bigger than arrays.
- Hash chain is an array: it only grows.
- Even assigning @a = () or %a = () only gets back the internal SV space, not the scaffolding.
- Hashes grow in two dimensions over time when use for persistent structures.

What can you do about it?

- Buffer inputs, clean them up then assign:

  ```
  my $buffer = <$read>;
  # chomp, split, etc, and then...
  push @linz, $buffer;
  ```

  - Same with hashes and arrays: recycle static varables for input and assign them to lexicals for use.

- Generate structures before you fork: at least the read-only portion will be shared.

Use arrays instead of hashes

- Store a tree as $tree{ parent } => @children instead of $tree{ $child }{ $parent } = ();
- Regenerating the tree in pre-order is simpler via @{ $tree{ $parent } } and small arrays take up less space than hashes – this makes a big difference for handling binary trees!

Summary

- Perl does not give back space to the O/S.
- Scalar strings only grow.
- Array and hash structures only grow.
- Lexicals are a big help.
- Clean up buffers before assigning them.