Ruby master - Feature #13166

Feature Request: Byte Arrays for Ruby 3

01/29/2017 09:43 PM - jzakiya (Jabari Zakiya)

Status:	Feedback	
Priority:	Normal	
Assignee:		
Target version:		

Description

I do a lot of numerically intensive applications.
In many instances I use arrays that contain boolean data (true|false or 1|0) values.

When I create such an array like:

data = Array.new(size, value) or just data = Array.new(size)

is it correct that the default memory unit size is that of the cpu, i.e. (32|64)-bit?

Since almost all modern cpus are byte addressable, I want to optimally use their system memory
by being able to explicitly create arrays of byte addressable elements.

For these use cases, this wlll allow my apps to extend their memory use capacity, instead
of wasting 31|63 bit of memory on 32|64 bit cpus systems just to store a boolean value.

To be clear, I am not talking about storing "strings" or "chars" but addessable 8-bit number elements.

I have not seen this capability documented in Ruby, thus I request this feature be added to
Ruby 3, and propose the following syntax that will be backwards compatible (non conflicting).

```
data = Array8.new(size, value)
```

Having explicit addressable byte arrays not only will increase memory use compactness of many
applications, this compactness will directly contribute to the Ruby 3x3 goal for performance
by allowing more data to be held entirely in cache memory when possible.

Thanks in advance for its consideratoin.

History

#1 - 01/30/2017 05:25 AM - shevegen (Robert A. Heiler)

I don't like the syntax (Array8) but I am not against it per se - I just want to
add that you actually made a good point nonetheless, simply by pointing out that
ruby 3 wants to be a lot faster. So this argument is pretty cool to see. :D

#2 - 01/31/2017 09:06 PM - jzakiya (Jabari Zakiya)

Whatever naming/syntax is used will be totally acceptable to me.

FYI, for what its worth, I translated a method from a rubygem I wrote
to Crystal using Int32 elements for 2 arrays of essentially boolean data,
just to get it to work easily. Then I changed those 2 arrays to arrays
of byte elements using Int8 types. These arrays can become very large for
this application. A mini benchmark run with the Crystal versions showed
a 12% performance increase with the Int8 (byte) arrays vs the Int32 arrays.

Since just about every modern cpu provides byte addressable elements, I suspect
a similar performance boost would occur for Ruby for any cpu OS if it created
native byte arrays to utilize the native instructions to handle byte addresses.

#3 - 02/01/2017 09:49 PM - funny_falcon (Yura Sokolov)

Why not write native extension? it is not hard if you know C.

#4 - 02/02/2017 06:56 AM - nobu (Nobuyoshi Nakada)

- Description updated

How about:

```
class Array8 < String
  def initialize(size, value = 0)
    [value].pack("C") * size
  end

  alias [] getbyte
  alias []= setbyte
end
```

#5 - 02/02/2017 08:04 AM - naruse (Yui NARUSE)

Usually on such use case I use String as a Int8Array.
I can access a Nth bit by str.getbyte(n/8)[n%8].

If you need further API, please share use cases.

#6 - 02/12/2017 08:19 PM - jzakiya (Jabari Zakiya)

I want to using an Array8 that has the same semantics, and inherits the same module methods (Enumerables, etc), as Array. I need its elements to be numberical values from 0..255 and/or -128..0..127.

Since 8-bit bytes are the minimal native addressable memory units on modern cpus (Int8 types in compiled languages), I not only can use theme to create fast and memory efficient boolean arrays, but also use them in a class of applications where I have numerical flags that can fit within an 8-bit byte.

To make them as fast/efficient as possible they should be core elements written in C, as is Array.
I can envision that this could be even useful for writing other core apps and gems to make them faster and/or greatly reduce their memory footprint, which makes them more cache and GC friendly too, which makes things faster.

Coming from my old Forth days of writing embedded systems, this will significantly aid writing for these types of apps where you want to be able to read/write/twiddle hardware bits easily/efficiently. It will make Ruby much more IoT friendly.

```
Boolen Examples:

bitmap = Array8.new(100,0)

bitmap[8] = 0  # or false
bitmap[9] = 1  # or true
bitmap.count(1)
lastbits = bitmap.last(10)
```

and also 8-bit numerical values

```
Numerical flags:

flags = Array8.new(10, 255)

is_refrig_light_on = (flags[7] | light_mask) == refrig_light_mask
```

#7 - 02/22/2017 09:18 AM - matz (Yukihiro Matsumoto)

Should we use narray/numarray instead? Maybe we can make either of them a bundled gem.

Matz.

#8 - 02/22/2017 09:41 AM - akr (Akira Tanaka)

I think String#getbit and String#setbit is useful.

#9 - 02/24/2017 11:23 PM - jzakiya (Jabari Zakiya)

Would your methods String#(get|set)bit be separate than an Array of bytes, or the name for them?

I think it would be confusing to associate a general byte array (an Array8 of numeric bytes) with Strings, especially since string characters can now be multiple bytes long.

An **Array8** is an array and a **String** a string.

#10 - 03/02/2017 08:06 AM - shyouhei (Shyouhei Urabe)

Jabari Zakiya wrote:

> An **Array8** is an array and a **String** a string.

I think I understand what you mean but in Ruby, classes tend not be split when they share same backends. For instance Array class can also be used as stacks (push/pop), queues (shift/unshift), association lists (assoc/rassoc), and sets (& / |). It is completely reasonable to separate those concepts into classes and other languages actually do so, but Ruby's design goes differently.

So if Array8 shares the same data structures with String, why not just let String do what you want?

#11 - 03/03/2017 01:32 AM - nirvdrum (Kevin Menard)

I'm in favor of a separate byte type as well. I think it conveys intent much more clearly, is easier to reason about, is easier to optimize, and is less error-prone.

While an ASCII-8BIT string can do the work, it leads to two use cases for Strings that may be at odds with each other (e.g., code ranges don't really mean anything for binary data). It also requires extra diligence to get the desired outcome. It's very easy to look like you're doing what you want, but get different outcomes. E.g., assuming a default encoding of UTF-8:

```
a = String.new
a << 0xff
a.encoding # => #<Encoding:ASCII-8BIT>
a.bytes # => [255]

b = ""
b << 0xff
b.encoding # => #<Encoding:UTF-8>
b.bytes # => [195, 191]

c = ""
c.encoding # => #<Encoding:UTF-8>
c << a
c.encoding # => #<Encoding:ASCII-8BIT>
c.bytes # => [255]
```

This may seem a bit contrived, but it's very easy to think you're doing one thing and actually be doing something else when working with Ruby strings if you're not paying careful attention to the encodings. String encoding negotiation might help fix common problems, but it could also silently change the encoding on you without your realization. Unfortunately, I've seen a fair bit of code that resorts to String#force_encoding to fix that problem. Now, if you're not careful you have a CR_BROKEN string and String operations aren't very well-defined on CR_BROKEN strings.

All of these issues are avoidable, but it requires a lot of forethought and, to some extent, familiarity with the way String is implemented to ensure you're maintaining the integrity of your binary data and achieving your performance objectives. I can appreciate that using String for two purposes like this looks attractive in Ruby because they're both backed by byte arrays in C. However, this is a situation where I think splitting the two use cases into different classes leads to more user-friendly code. As an added benefit, I believe a dedicated byte array type would be easier to optimize when it doesn't need to be concerned with adhering to the String API as well.

#12 - 03/03/2017 10:20 PM - jzakiya (Jabari Zakiya)

The points Kevin makes are exactly some of the reason I think, **from a users perspective**, its clearer to provide a separate name and API for this resource.

A normal user will have no knowledge of how the things works under the hood (and shouldn't be forced to), and it actually will allow Ruby to develop and improve these resources independently of each other (especially to potentially optimize their implementation for different hardware, Intel, Arm, AMD, whatever).

The key thing to document for users is that this is a generally purpose array of bytes, (not a special purpose use of Strings), that has the same API as an Array, but whose content is limited to byte data values.

#13 - 03/13/2017 08:21 AM - matz (Yukihiro Matsumoto)

- Status changed from Open to Feedback

It seems OP wanted BitVector, not Array8.

> In many instances I use arrays that contain boolean data (true|false or 1|0) values.

that means the intention can be achieved by the combination of strings and getbit/setbit methods.

Or it's possible that OP really wants packed arrays of (8bit) numbers. In that case, NumArray can be the solution.

Matz.

#14 - 03/14/2017 07:40 PM - jzakiya (Jabari Zakiya)

My original use case was for creating an array of data that essentially contained 1|0 data to represent true|false flags.

In the C version of my program I just created an array of bytes (char) because they are fast, and waste the least amount of memory. Creating a true bit_array is the most memory efficient, but is much slower than using an array of bytes (char), and was the best trade-off for that use case.

But beyond that original use case, having a true Array-of-Bytes not only can create a fast facsimile of a true bit_array/vector it can also be used to efficiently store and manipulate inherently byte valued data, as I've given examples of previously.

So given some semantics for an Array-of-Bytes I want to create arrays like:

```
arrybytes = Array8.new(n)
arrybytes = Array8.new(n,0)
arrybytes = Array8.new(n,255)
arrybytes = Array8.new
arrybytes = [1, 0 49, 30, 126, 200, 65, 17]
arrybytes << 48 << 0 << 8
arrybytes => [1, 0 49, 30, 126, 200, 65, 17, 48, 0, 8]
arrybytes[5] => 200
arrybttes[7] = 0x20
arrybytes => [1, 0 49, 30, 126, 200, 65, 32, 48, 0, 8]
arrybytes.size => 11
```

So I want to do everything I can with a regular Array, except that its content is limited to 8-bit byte values.

I am not familiar with NumArray. Can you give examples of how it can be used to provide this similar usage?

#15 - 03/29/2017 04:21 PM - jzakiya (Jabari Zakiya)

This is a comparison of real code I have in a gem that is optimized for CRuby and JRuby. JRuby allows you to use Java byte-arrays, which is both more memory efficient than the CRuby version (I can create bigger arrays), but its much, much faster in JRuby than using **Array** in JRuby, here as an array of boolean (1|0) values.

An equivalent construction in CRuby can have similar advantages in mem/speed advantages.

```
    case RUBY
    when "jruby"
      def array_check(n,v)  # catch out-of-memory errors on array creation
        Java::byte[n].new rescue return
      end
      .....
      .....
    else
      def array_check(n,v)  # catch out-of-memory errors on array creation
        Array.new(n,v) rescue return  # return an array or nil
      end
      .....
      .....
   end
```

#16 - 03/29/2017 04:35 PM - jzakiya (Jabari Zakiya)

Crystal allows you to create byte-arrays as below:

```
byte_array = [] of Int8
```

#17 - 03/31/2017 08:04 AM - jwmittag (Jörg W Mittag)

I agree that the OP probably is more interested in a BitVector/BitArray than a ByteArray, at least for the specific use case he is describing. Nonetheless, such a data type sounds useful for high-performance code; it may also make it easier to self-host larger portions of the stdlib and corelib.

I would suggest to take a good look at the structured data types that have been added to ECMAScript in the last few years, specifically TypedArrays, DataView, and ArrayBuffer, which are a generalization of what the OP is asking about: ArrayBuffer is an untyped contiguous portion of memory. It cannot be manipulated directly, it can only be manipulated through *views*. A buffer can have multiple views associated with it, and a view can be associated with only a subsection of the buffer. There are two kinds of views: DataViews offer heterogeneous access, with methods like set_int8, set_uint8, set_int16, set_float64 (and the corresponding get_* methods) and so on. TypedArrays offer homogeneous access, there are types like

Int8Array, UInt8Array, and so on. TypedArrays behave like Arrays, but only support a subset of Array methods.

Translating the ECMAScript API to Ruby could look something like this:

```
class ArrayBuffer
  def initialize(length) end

  attr_reader :byte_length

  def slice(begin_offset, end_offset = byte_length) end
end

class DataView
  def initialize(buffer, byte_offset = 0, byte_length = buffer.byte_length - byte_offset) end

  attr_reader :buffer, :byte_offset, :byte_length

  def get_int8(byte_offset) end
  def set_int8(byte_offset, value) end
  def get_uint8(byte_offset) end
  def set_uint8(byte_offset, value) end
  def get_uint8c(byte_offset) end
  def set_uint8c(byte_offset, value) end
  def get_int16(byte_offset, little_endian = false) end
  def set_int16(byte_offset, value, little_endian = false) end
  def get_uint16(byte_offset, little_endian = false) end
  def set_uint16(byte_offset, value, little_endian = false) end
  def get_int32(byte_offset, little_endian = false) end
  def set_int32(byte_offset, value, little_endian = false) end
  def get_uint32(byte_offset, little_endian = false) end
  def set_uint32(byte_offset, value, little_endian = false) end
  def get_int64(byte_offset, little_endian = false) end
  def set_int64(byte_offset, value, little_endian = false) end
  def get_uint64(byte_offset, little_endian = false) end
  def set_uint64(byte_offset, value, little_endian = false) end
  def get_float32(byte_offset, little_endian = false) end
  def set_float32(byte_offset, value, little_endian = false) end
  def get_float64(byte_offset, little_endian = false) end
  def set_float64(byte_offset, value, little_endian = false) end
end

class TypedArray
  private_class_method :new # TypedArray is abstract
  def initialize(length) end
  def initialize(typed_array) end
  def initialize(enum) end
  def initialize(buffer, byte_offset = 0, byte_length = buffer.byte_length - byte_offset) end

  attr_reader :buffer, :byte_offset, :byte_length, :length

  def set(array, offset = 0) end
  def subarray(begin_offset = 0, end_offset = byte_length)

  include Enumerable

  class Int8 < self
    BYTES_PER_ELEMENT = 1

    def each(&blk) end
    def [](…) end
    def []=(…) end

    # additional array methods …
  end

  class UInt8 < self
    BYTES_PER_ELEMENT = 1

    # …
  end

  class UInt8C < self
    BYTES_PER_ELEMENT = 1

    # …
```

```
  end

  class Int16 < self
    BYTES_PER_ELEMENT = 2

    # …
  end

  # and so on
end

class Array
  def to_typed_array(type) end
end
```