Guide to the **MemUse** Package

wrathematics

May 29, 2013

Contents

1 Introduction

1.1 History

Hi there. I am ∼testing this now∼.

This package was born out of a ≈ 10 line function I wrote to estimate the memory usage of (non-allocated) in-core, dense R objects of numeric (double precision) data. I need this in my life by a surprising amount, so it made sense to actually create this thing instead of constantly doing ad hoc multiplications of $nrows \times ncols \times 8$ then dividing by powers of 1024 (or 1000).

But then I got the great idea to make this application ∼enterprise ready∼ by adding a lot of unnecessary and convoluted OOP, and this stupid package was born. This is sort of a love letter to other needlessly complex programs, like the Enterprise Fizzbuzz[1].

1.2 License

Figure 1: The GNU GPL Explained

This package is free software licensed under the GNU General Public License, version ≥ 2 (see Figure 1). If you violate the terms of the GPL then Richard Stallman's beard will sue you in internet court.

[1] If you are unfamiliar with the fizzbuzz, see my posts "Honing your R skills for Job Interviews" and "The Fizzbuzz that Fortran Deserves".

1.3 Installation

The package consists entirely of R code, so everything should install fine no matter which platform you use. To install this from source on Windows, you will need to first install the Rtools package. The package should install on Mac or Linux[2] without problem.

The easiest way to install **MemUse** is via the **devtools** package. With this, you can effectively install packages from github just as you would from the CRAN. To install **MemUse** using **devtools**, simply issue the command:

```
library(devtools)
install_github(repo="memuse", username="wrathematics")
```

from R. Alternatively, you could download the sourcecode from github, unzip this archive, and issue the command:

```
R CMD INSTALL memuse-master
```

from your shell.

2 Using the MemUse Package

2.1 Size Matters, and How You Are Using It is Wrong

The core of the **MemUse** package is the `memuse` class object. You can construct a `memuse` object via the `memuse()` or `mu()` constructor. The constructor has several options. You can pass the size of the object, the unit, the unit prefix (IEC or SI), and the unit names (short or long). The size is the number of bytes, scaled by some factor depending on the unit. The unit is an abstract rescaling unit, like percent, used for the sake of simple comprehension at larger scales; for example, kilobyte and kibibyte are the typical storage units to represent "roughly a thousand" bytes (more on this later). Finally, the unit names are for printing, i.e., controlling whether the long version (e.g., kilobyte) or short version (kB) is used. Table 1 gives a complete list of the different units for the different prefices.

So for example, 1 kilobyte (kB) is equal to 1000 bytes, but 1 kibibyte (KiB) is equal to 1024 bytes. And so 1 kB is roughly 0.977 KiB.

The reason for this odd distinction is that there is general ambiguity in the public versus technical definition of these terms. People, even those who know the difference (myself included) almost overwhelmingly

[2] I'd just like to interject for a moment. What you're referring to as Linux, is in fact, GNU/Linux, or as I've recently taken to calling it, GNU plus Linux. Linux is not an operating system unto itself, but rather another free component of a fully functioning GNU system made useful by the GNU corelibs, shell utilities and vital system components comprising a full OS as defined by POSIX.

Many computer users run a modified version of the GNU system every day, without realizing it. Through a peculiar turn of events, the version of GNU which is widely used today is often called Linux, and many of its users are not aware that it is basically the GNU system, developed by the GNU Project.

There really is a Linux, and these people are using it, but it is just a part of the system they use. Linux is the kernel: the program in the system that allocates the machine's resources to the other programs that you run. The kernel is an essential part of an operating system, but useless by itself; it can only function in the context of a complete operating system. Linux is normally used in combination with the GNU operating system: the whole system is basically GNU with Linux added, or GNU/Linux. All the so-called Linux distributions are really distributions of GNU/Linux.

IEC Prefix			SI Prefix		
Short	Long	Factor	Short	Long	Factor
b	byte	1	b	byte	1
KiB	kibibyte	2^{10}	kB	kilobyte	10^3
MiB	mebibyte	2^{20}	MB	megabyte	10^6
GiB	gibibyte	2^{30}	GB	gigabyte	10^9
TiB	tebibyte	2^{40}	TB	terabyte	10^{12}
PiB	pebibyte	2^{50}	PB	petabyte	10^{15}
EiB	exbibyte	2^{60}	EB	exabyte	10^{18}
ZiB	zebibyte	2^{70}	ZB	zettabyte	10^{21}
YiB	yobibyte	2^{80}	YB	yottabyte	10^{24}

Table 1: Units, Unit Prefices, and Scaling Factors for Byte Storage

use, for example, gigabyte when they mean gibibyte. The reason for this is obvious; "gibibyte" sounds fucking stupid. This actually gets all the more confusing because in addition to conflating 1 MB with 1 MiB, ISP's advertise their speeds in terms of bits[3] *using the same goddamn symbol*, because they're huge assholes.

Another example is when people talk about ∼big data∼. Often I/O people will use the term "terabytes" or "exabytes" and mean it. Rescaling these units into the ones people are generally more familiar with is simple with the `MemUse` package:

```
> swap.prefix(mu(size=1, unit="tb", unit.prefix="SI"))
0.909 TiB
> swap.prefix(mu(size=1, unit="pb", unit.prefix="SI"))
0.888 PiB
```

These sizes represent an impressive amount of data, but this ambiguity in naming conventions allows people to lie a bit. For all of these reasons, since the package is meant to be useful for understanding R object size, the default behavior is somewhat complicated, but can be summarized as trying to provide what most people meant in the first place. We achieve this by offering several default string objects which the user can easily control. These units are `.UNIT`, `.PREFIX`, and `.NAMES`.

2.2 Default Parameters

The `.UNIT` object defaults to `best` and should probably just be left alone. Functions that need to know an input unit, such as the constructor, have default argument `unit=.UNIT`. Realistically, you are probably better off modifying that argument as necessary than changing `.UNIT`. For example, you want to construct a 100 KiB `memuse` object, you probably just want to call

```
mu(100, "KiB")
```

This is equivalent to calling

```
mu(102400)
```

[3] 1 byte is 8 bits

since the default `.UNIT=best` will make the choice to switch the units from b to KiB once you breach 1024 bytes. This sounds a lot more confusing than it really is.

More useful is the `.PREFIX` parameter. This must either be `SI` or `IEC`, with the latter being the package default.

```
> .PREFIX <- "SI"
> x <- mu(10, "kb")
> x
10.000 KB
> swap.prefix(x)
9.766 KiB
```

2.3 Methods

Aside from the constructor, you have already seen one very useful method: `swap.prefix()`. In addition to these, we have several other obvious methods, such as `swap.unit()`, `swap.names()`, `print()`, `show()`, etc. But we also have some simple arithmetic, namely `‘+‘` (addition), `‘*‘` (multiplication), and `‘^‘` (exponentiation). So for example:

```
> mu(100) + mu(200)
300.000 B
> mu(100) * mu(200) # 100*200/1024
19.531 KiB
```

It's not hard to implement other things like division, but I didn't because I thought it was stupid.

Finally, we have the methods that inspired the creation of this entire dumb thing in the first place: `howbig()` and `howmany()`. The former takes in the dimensions of a matrix (`nrow` rows and `ncol` columns) and returns the memory usage (as the package namesake would imply) of the object. So for example, if you wanted to perform a principal components decomposition on a 100,000 by 100,000 matrix via SVD (as we have), then you would need:

```
> howbig(100000, 100000)
74.506 GiB
```

Of ram just to store the data. Another interesting anecdote about this sized matrix is that we were able to generate it in just over a tenth of a second. Pretty cool, eh?

As mentioned before, there is also the `howmany()` method which does somewhat the reverse of `howbig()`. Here you pass a `memuse` object and get a matrix size out. You can pass (exactly) one argument `nrow` or `ncol` in addition to the `memuse` object; the method will determine the maximum possible size of the outlying dimension in the obvious way. If no additional argument is passed, then the largest square matrix dimensions will be returned.

2.4 Package Demos

In addition to all of the above, the **MemUse** package includes several demos. You can execute them via the command:

List of Demos

```
### (Use Rscript.exe for windows systems)

# Basic construction/use of memuse objects
Rscript -e "demo(demo, package='MemUse', ask=F, echo=F)"
# Arithmetic
Rscript -e "demo(demo2, package='MemUse', ask=F, echo=F)"
# howbig/howmany examples
Rscript -e "demo(demo3, package='MemUse', ask=F, echo=F)"
```

2.5 Comparison to `object.size()`

R contains a handy tool for telling you how big an already allocated object is, the `object.size()` method. This package is effectively an extension of that method for un-allocated objects, provided your objects are numeric (more on this later).

So say we have the vector `x <- 1.0`. This should be using 8 bytes to store that `1.0` as a double, right? Well...

```
> object.size(1.0)
48 bytes
```

So where is all that extra space coming from? Simply put, R objects are more than just their data. They contain a great deal of very useful metadata, which is where all the nice abstraction comes from. Whenever you create a vector, R keeps track, for example, its length. If you do not appreciate this convenience, go learn C and then get back to me.

For vectors, this overhead is 40 bytes, regardless of the type of data. Matrices, unsurprisingly cost more, clocking in at 200 bytes overhead. It is worth noting that this overhead does not scale; it is on a per-object basis. So we don't need 40 bytes for each element of a vector when just 8 would do (in the case of double precision values). We need 40 plus 8 per element:

```
> # 2 elements
> 40+8*2
[1] 56
> object.size(rnorm(2))
56 bytes
> # 100.000 elements
> 40+1e5*8
[1] 800040
‘> object.size(rnorm(1e5))
800040 bytes
```

The story is slightly more complicated for integer data (and a lot more complicated for strings; see the following section). On my machine (and probably yours, but not necessarily), ints costs 4 bytes. However, R does some aggressive allocation, most likely for reasons of efficiency:

```
> object.size(1L:3L)
56 bytes
> object.size(1L:4L)
56 bytes
```

Here we see R allocating more bytes than it needs for integer vectors sometimes, choosing to allocate in 16 byte chunks rather than 8 byte chunks.

The **MemUse** package does not adjust for this overhead, because it honestly just doesn't matter. This overhead is really not worth worrying about, and when you think about all the abstraction it buys you, it's a hell of a bargain. If you have a million R objects stored, you're wasting less than a mebibyte (1024^2 bytes); so you would need a billion objects to use just about a gibibyte (1024^3 bytes) on overhead. And if you're doing that kind of silly nonsense, my advice would be to learn how to properly use data structures.

2.6 Strings

String objects have been avoided up until this point because they are much more difficult to describe in general, unless they have a great deal of regularity imposed on them. In R, strings by default are allocated to use 56 bytes (not counting overhead), unless they need more. I'm not sure why this value was chosen, but 56 byte strings will allow for the storage of 7 chars (like `a` but not `aa`). Each char costs 1 byte, so there's some fat overhead for the strings here, and almost certainly an additional byte held out for the null terminator. So for example, recall that a vector allocates 40 bytes of overhead, so the vector string `letters` should use $56 \times 26 + 40$ bytes. We can easily verify that this is the case:

```
> 56*26+40
[1] 1496
> object.size(letters)
1496 bytes
```

If you have a string with more than 7 chars, R will allocate extra space in 8-16 byte blocks. After the initial 8 byte allocation (7 chars + null terminator), if you need more you get an additional 8 bytes (in reality this is probably a contiguous 16 byte allocation; I have not bothered to check). Beyond that, storage is allocate in 16 byte blocks for each string. For example:

```
>  object.size(c(paste(rep("a", 7), collapse=""), "a"))
152 bytes
>  object.size(c(paste(rep("a", 7+1), collapse=""), "a"))
160 bytes
>  object.size(c(paste(rep("a", 7+8+1), collapse=""), "a"))
176 bytes
>  object.size(c(paste(rep("a", 7+8+16+1), collapse=""), "a"))
192 bytes
```

If you have a vector of strings with them of varying lengths, the allocation of individual elements is handled on a case-by-case basis. Consider the following:

```
> object.size(c(paste(rep("a", 7+8+16+1), collapse=""), "a"))
192 bytes
```

This object (the vector of 2 elements with first element "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa" and second element "a") is using 40 bytes for the vector, $56 + 8 + 16 + 16$ bytes for the first element, and 56 bytes for the second.

For all of these reasons, and given the fact that I almost never (ever) deal with character data, I have not bothered to make any attempt to extend, for example, `howmany()` or `howbig()`, to incorporate strings. Deal with it, nerd.