Package 'mmap'

December 9, 2025

Type Package

Title Map Pages of Memory

Version 0.6-23

Date 2025-12-08

LazyLoad yes

Maintainer Jeffrey A. Ryan <jeff.a.ryan@gmail.com>

Description R interface to POSIX mmap and Window's MapViewOfFile.

VignetteBuilder utils

Classification/ACM-2012 500

License GPL-3

URL https://github.com/jaryan/mmap

BugReports https://github.com/jaryan/mmap/issues

NeedsCompilation yes

Author Jeffrey A. Ryan [aut, cre] (ORCID:
<https://orcid.org/0009-0003-3627-7896>)

Repository CRAN

Date/Publication 2025-12-09 06:10:19 UTC

Contents

C_types 2
make.fixedwidth 5
mmap 6
mmap.csv 9
mmapFlags 10
mprotect 12
msync 13
sizeof 13
struct 14

Index **19**

C_types *Virtual R Types On Disk.*

Description

These functions describe the types of raw binary data stored on disk.

Usage

```
char(length = 0, nul = TRUE)
uchar(length = 0)
logi8(length = 0)
logi32(length = 0)
int8(length = 0)
uint8(length = 0)
int16(length = 0)
uint16(length = 0)
int24(length = 0)
uint24(length = 0)
int32(length = 0)
int64(length = 0)
real32(length = 0)
real64(length = 0)
cplx(length = 0)
cstring(length = 0, na.strings = "NA")

as.Ctype(x)
is.Ctype(x)

cstring.MaxWidth()
sizeofCtypes()
```

Arguments

length	desired length. Not used when passed to mode= in mmap call.
x	R object to coerce or test
nul	are characters delimited by a nul byte?
na.strings	string to convert to R's NA. See Details for current implementation.

Details

R has very limited storage types. There is one type of integer and one type of float (double). Storage to disk often can be made more efficient by reducing the precision of the data. These functions provide for a sort of virtual mapping from disk to native R type, for use with mmap-ed files.

When a memory mapping is created, a conversion method if declared for both extracting values from disk, as well as replacing elements on disk. The preceeding functions are used in the internal compiled code to handle the conversion.

It is the user's responsibility to ensure that data fits within the prescribed types. All fixed-width types support extraction, replacement, and boolean Ops (e.g. ==). See below for note on `cstring` layout.

`cstring` reads nul-terminated strings from binary C-style arrays. To minimize memory allocation, two additional steps are carried out. First, when a memory map is initiated, the length (N) of the character array is calculated. The calculation of word offsets to facilitate access are deferred until the first request `[` or a `Ops` request. This offset calculation requires the creation of an internal index made up of short integers, representing the length of each character element. On most platforms, this is at least 65534 (sizeof(short) - 1 for nul byte), but can be found via `cstring.MaxWidth`. This index will consume sizeof(short) * N memory, allocated outside of R.

At present `na.strings="NA"` is ignored and all occurances of the (binary) string 'NA' are converted to `NA_character_` types in R. This is also used by the **mmap** `is.na` function.

Value

An R typed vector of length 'length' with a virtual type and class 'Ctype'. Additional information related to number of bytes and whether the vitrual type is signed is also contained.

Warning

The is no attempt to store or read metadata with respect to the extracted or replaced data. This is simply a low level interface to facilitate data reading and writing.

Note

R vectors may be used to create files on disk matching the specified type using the functions `writeBin` with the appropriate `size` argument. See also.

Author(s)

Jeffrey A. Ryan

References

https://en.wikipedia.org/wiki/C_variable_types_and_declarations https://cran.r-project.org/doc/manuals/R-exts.html

See Also

`writeBin`

Examples

```
tmp <- tempfile()

# write a 1 byte signed integer -128:127
writeBin(-127:127L, tmp, size=1L)
file.info(tmp)$size
one_byte <- mmap(tmp, int8())
one_byte[]
```

```
munmap(one_byte)

# write a 1 byte unsigned integer 0:255
writeBin(0:255L, tmp, size=1L)
file.info(tmp)$size
one_byte <- mmap(tmp, uint8())
one_byte[]
munmap(one_byte)

# write a 2 byte integer -32768:32767
writeBin(c(-32768L,32767L), tmp, size=2L)
file.info(tmp)$size
two_byte <- mmap(tmp, int16())
two_byte[]
munmap(two_byte)

# write a 2 byte unsigned integer 0:65535
writeBin(c(0L,65535L), tmp, size=2L)
two_byte <- mmap(tmp, uint16())
two_byte[]

# replacement methods automatically (watch precision!!)
two_byte[1] <- 50000
two_byte[]

# values outside of range (above 65535 for uint16 will be wrong)
two_byte[1] <- 65535 + 1
two_byte[]

munmap(two_byte)

# write a 4 byte integer standard R type
writeBin(1:10L, tmp, size=4L)
four_byte <- mmap(tmp, int32())
four_byte[]
munmap(four_byte)

# write 32 bit integers as 64 bit longs (where supported)
int64()  # note it is a double in R, but described as int64
writeBin(1:10L, tmp, size=8L)
eight_byte <- mmap(tmp, int64())
storage.mode(eight_byte[])  # using R doubles to preserve most long values
eight_byte[5] <- 2^40  # write as a long, a value in R that is double ~2^53 is representable
eight_byte[5]
munmap(eight_byte)

cstring()
cstring.MaxWidth()
writeBin(c("this","is","a","sentence"), tmp)
strings <- mmap(tmp, cstring())
strings[1:2]
strings[]
munmap(strings)
```

```
unlink(tmp)
```

`make.fixedwidth` *Convert Character Vectors From Variable To Constant Width*

Description

Utility function to convert a vector of character strings to one where each element has exactly 'width'-bytes.

Usage

```
make.fixedwidth(x, width = NA, justify = c("left", "right"))
```

Arguments

`x`	A character vector.
`width`	Maximum width of each element. width=NA (default) will expand each element to the width required to contain the largest element of x without loss of information.
`justify`	How should the results be padded? 'left' will add spacing to the right of shorter elements in the vector (left-justified), 'right' will do the opposite.

Details

The current implementation of mmap only handles fixed-width strings (nul-terminated). To simplify conversion of (potentially) variable-width strings in a character vector, all elements will be padded to the length of the longest string in the vector or set to length `width` if specified.

All new elements will be left or right justified based on the `justify` argument.

Value

A character vector where each element is of fixed-width.

Note

Future implementions will possibly support variable-width character vectors.

Author(s)

Jeffrey A. Ryan

Examples

```
month.name
make.fixedwidth(month.name)
```

mmap *Map And Unmap Pages of Memory*

Description

Wrapper to POSIX 'mmap' and Windows MapViewOfFile system calls.

Usage

```
mmap(file, mode = int32(),
     extractFUN=NULL, replaceFUN=NULL,
     prot=mmapFlags("PROT_READ","PROT_WRITE"),
     flags=mmapFlags("MAP_SHARED"),
     len, off=0L, endian=.Platform$endian,
     ...)

munmap(x)

as.mmap(x, mode, file, ...)
is.mmap(x)

extractFUN(x)
replaceFUN(x)

extractFUN(x) <- value
replaceFUN(x) <- value
```

Arguments

file	name of file holding data to be mapped into memory
mode	mode of data on disk. Use one of 'char()' (char <-> R raw), 'int8()' (char <-> R integer), 'uint8()' (unsigned char <-> R integer), 'int16()' (short <-> R integer), 'uint16()' (unsigned short <-> R integer), 'int24()' (3 byte integer <-> R integer), 'uint24()' (unsigned 3 byte integer <-> R integer), 'int32()' (R integer), 'real32()' (float <-> R double), 'real64()' (R double), 'cplx()' (R complex), 'cstring()' (R variable length character array), 'struct()' (Collection of Ctypes as defined by mmap). See the related functions for details.
extractFUN	A function to convert the raw/integer/double values returned by subsetting into a complex R class. If no change is needed, set to NULL (default).
replaceFUN	A function to convert the R classes to underlying C types for storage.
prot	access permission to data being mapped. Set via bitwise OR with mmapFlags to one or more of 'PROT_READ': Data can be read, 'PROT_WRITE': Data can be written, 'PROT_EXEC': Data can be executed, 'PROT_NONE': Data cannot be accessed. Not all will apply within the context of R objects. The default is PROT_READ | PROT_WRITE.

flags	additional flags to mmap. Set via bitwise OR with mmapFlags to one or more of 'MAP_SHARED': Changes are shared (default), 'MAP_PRIVATE': Changes are private, 'MAP_FIXED': Interpret *addr* exactly (Not Applicable). Not all will apply within the context of R objects.
len	length in bytes of mapping from offset. (EXPERT USE ONLY)
off	offset in bytes to start mapping. This *must be* a multiple of the system pagesize. No checking is currently done, nor is there any mmap provision to find pagesize automatically. (EXPERT USE ONLY)
endian	endianess of data. At present this is *only* applied to int8,int16, int32,float,real32,double, and real64 types for both atomic and struct types. It is applied universally, and not at struct member elements.
...	unused
x	an object of class 'mmap'
value	a function to apply upon extraction or replacement.

Details

The general semantics of the R function map to the underlying operating system C function call. On unix-alikes this is 'mmap', on Windows similar functionality is provided by the system call 'MapViewOfFile'. The notable exception is the use of the R argument file in place of void *addr and int fildes. Additionally len and off arguments are made available to the R level call, though require special care based on the system's mmap and are advised for expert use only.

as.mmap allows for in-memory objects to be converted to mmapped version on-disk. The files are stored in the location specified by file. Passing an object that has an appropriate as.mmap method will allow R objects to be automatically created as memory-mapped object. This works for most atomic types in R, including numeric, complex, and character vectors. A special note on character vectors: the implementation supports both variable width character vectors (native R) as well as fixed width arrays requiring a constant number of bytes per element. The current default is to use fixed width, with variable width enabled by setting mode=cstring(). See as.mmap.character for more details.

Complex data types, such as 2 dimesioned vectors (matrix) and data.frames can be supported using appropriate extractFUN and replaceFUN functions to convert the raw data. Basic object conversion is made available in included as.mmap methods for boths types as of version 0.6-3.

All mode types are defined for single-column atomic data, with the exception of structs. Multiple column objects are supported by the use of setting dim. All data is column major. Row major orientation, as well as supporting multiple types in one object - imitating a data.frame, is supported via the struct mode.

Using struct as the mode will organize the binary data on-disk (or more correctly read data organized on disk) in a row-major orientation. This is similar to how a row database would be oriented, and will provide faster access to data that is typically viewed by row. See help(struct) for examples of semantics as well as performance comparisons.

Value

The mmap and as.mmap call returns an object of class mmap containing the fields:

data: pointer to the 'mmap'ped file.

bytes: size of file in bytes. This is not in resident memory.

filedesc: A names integer file descriptor, where the name is path to the file mapped.

storage.mode: R type of raw data on disk. See `types` for details.

pagesize: operating system pagesize.

extractFUN: conversion function on extraction (optional).

replaceFUN: conversion function for replacement (optional).

Author(s)

Jeffrey A. Ryan

References

mmap: http://www.opengroup.org/onlinepubs/000095399/functions/mmap.html

See Also

See Also as `mmapFlags`,

Examples

```
# create a binary file and map into 'ints' object
# Note that we are creating a file of 1 byte integers,
# and that the conversion is handled transparently
tmp <- tempfile()
ints <- as.mmap(1:100L, mode=int8(), file=tmp)

ints[1]
ints[]
ints[22]
ints[21:23] <- c(0,0,0)
ints[]  # changes are now on disk

# add dimension
dim(ints) <- c(10,10)
ints[]
ints[6,2] # 6th row of 2nd column
ints[ ,2] # entire 2nd column
munmap(ints)

# store Dates as natural-size 'int' on disk
writeBin(as.integer(Sys.Date()+1:10), tmp)
DATE <- mmap(tmp,extractFUN=function(x) structure(x,class="Date"))
DATE[]
munmap(DATE)

# store 2 decimal numeric as 'int' on disk, and convert on extraction
num <- mmap(tmp,extractFUN=function(x) x/100)
num[]
```

```
munmap(num)

unlink(tmp)

# convert via as.mmap munmap
int <- as.mmap(1:10L)
num <- as.mmap(rnorm(10))
```

`mmap.csv` *Memory Map Text File*

Description

Reads a file column by column and creates a memory mapped object.

Usage

```
mmap.csv(file,
         header = TRUE,
         sep = ",",
         quote = "\"",
         dec = ".",
         fill = TRUE,
         comment.char = "",
         row.names,
         ...)
```

Arguments

`file`	the name of the file containing the comma-separated values to be mapped.
`header`	does the file contain a header line?
`sep`	field separator character
`quote`	the set of quoting characters
`dec`	the character used for decimal points in the file
`fill`	unimplemented
`comment.char`	unimplemented
`row.names`	what it says
`...`	additional arguments

Details

`mmap.csv` is meant to be the analogue of read.csv in R, with the primary difference being that data is read, by column, into memory-mapped structs on disk. The intention is to allow for comma-separated files to be easily mapped into memory without having to load the entire object at once.

Value

An mmap object containing the data from the file. All types will be set to the equivelant type from mmap as would be in R from a call to read.csv.

Warning

At present the memory required to memory-map a csv file will be the memory required to load a single column from the file into R using the traditional read.table function. This may not be adequately efficient for extremely large data.

Note

This is currently a very simple implementation to facilitate exploration of the mmap package. While the interface will remain consistent with read.csv from **utils**, more additions to handle various out-of-core types available in mmap as well as performance optimization will be added.

Author(s)

Jeffrey A. Ryan

See Also

mmap, read.csv

Examples

```
data(cars)
tmp <- tempfile()
write.csv(cars, file=tmp, row.names=FALSE)

m <- mmap.csv(tmp)

colnames(m) <- colnames(cars)

m[]

extractFUN(m) <- as.data.frame  # coerce list to data frame upon subset

m[1:3,]

munmap(m)
```

---

mmapFlags *Create Bitwise Flags for mmap.*

Description

Allows for unquoted C constant names to be bitwise OR'd together for passing to mmap related calls.

Usage

```
mmapFlags(...)
```

Arguments

...	A comma or vertical bar 'l' seperated list of zero or more valid mmap constants. May be quoted or unquoted from the following: PROT_READ, PROT_WRITE, PROT_EXEC, PROT_NONE, MAP_SHARED, MAP_PRIVATE, MAP_FIXED, MS_ASYNC, MS_SYNC, MS_INVALIDATE. See details for more information.

Details

Argument list may contain quoted or unquoted constants as defined in <sys/mman.h>. See invididual functions for details on valid flags.

Multiple values passed in will be bitwise OR'd together at the C level, allowing for semantics close to that of native C calls.

Value

An integer vector of length 1.

Note

Read your system's 'mmap' man pages for use details.

Author(s)

Jeffrey A. Ryan

See Also

See Also as mmap, ~~~ See Also as mprotect, ~~~

Examples

```
mmapFlags(PROT_READ)
mmapFlags(PROT_READ | PROT_WRITE)
mmapFlags("PROT_READ" | "PROT_WRITE")
mmapFlags(PROT_READ , PROT_WRITE)
mmapFlags("PROT_READ" , "PROT_WRITE")
```

`mprotect` *Control Protection of Pages*

Description

Wrapper to `mprotect` system call. Not all implementations will guarantee protection.

Usage

```
mprotect(x, i, prot)
```

Arguments

`x`	`mmap` object.
`i`	location and length of pages to protect.
`prot`	protection flag set by `mmapFlags`. Must be one or more of: 'PROT_NONE', 'PROT_READ', 'PROT_WRITE', 'PROT_EXEC'.

Details

This functionality is very experimental, and likely to be of limited use with R, as the result of a page access that is protected is a SIG that isn't likely to be caught by R. This may be of use for other programs sharing resource with R.

Value

0 upon success, otherwise -1.

Author(s)

Jeffrey A. Ryan

References

'mprotect' man page.

msync *Synchronize Memory With Physical Storage*

Description

msync calls the underlying system call of the same name. This writes modified whole pages back to the filesystem and updates the file modification time.

Usage

```
msync(x, flags=mmapFlags("MS_ASYNC"))
```

Arguments

x	An mmap object.
flags	One of the following flags: 'MS_ASYNC': return immediately (default). 'MS_SYNC': perform synchronous writes. 'MS_INVALIDATE': invalidate all cached data. Per the man page, 'MS_ASYNC' is not permitted to be combined with the other flags.

Details

See the appropriate OS man page.

Value

0 on success, otherwise -1.

Author(s)

Jeffrey A. Ryan

sizeof *Calculate the Size of Datatypes*

Description

Calculate the number of bytes in an R data type used by **mmap**.

Usage

```
sizeof(type)
```

Arguments

type	A type constructor (function), R atomic, or **mmap** Ctype.

Details

A constructor for the purposes of `sizeof` is a function object used to create an atomic type for R or mmap. These include the base atomic type functions such as `integer`, `character`, `double`, `numeric`, `single`, `complex` and similar. In addition, the Ctype constructors in **mmap** such as `int8`, `uint8`, `real32`, etc may be passed in.

More typically a representative object of the above types can be passed in to determine the appropriate data size.

The purpose of this function is for use to help construct a proper `offset` argument value for mmap and `mprotect`, though neither use is common or encouraged since alignment to pagesize is required from the system call.

Value

Numeric bytes used.

Author(s)

Jeffrey A. Ryan

See Also

`pagesize` `as.Ctype`

Examples

```
# all are equal

sizeof(int32)
sizeof(int32())
sizeof(integer)
sizeof(integer())
sizeof(1L)
```

---

struct *Construct a Ctype struct*

Description

Construct arbitarily complex 'struct'ures in R for use with on-disk C struct's.

Usage

```
struct(..., bytes, offset)

is.struct(x)
```

Arguments

...	Field types contained in struct.
bytes	The total number of bytes in the struct. See details.
offset	The byte offset of members of the struct. See details.
x	object to test

Details

struct provides a high level R based description of a C based struct data type on disk.

The types of data that can be contained within a structure (byte array) on disk can be any permutation of the following: int8, uint8, int16 uint16, int32, real32, and real64. 'struct's are not recursive, that is all struct's contained within a struct must be logically flattened (core elements extracted).

All C types are converted to the appropriate R type internally.

It is best to consider a struct a simple byte array, where at specified offsets, a valid C variable type exists. Describing the struct using the R function struct allows mmap extraction to proceed as if the entire structure was one block, (a single 'i' value), and each block of bytes can thus be read into R with one operation.

One important distinction between the R struct (and the examples that follow) and a C struct is related to byte-alignment. Note that the R version is effectively serializing the data, without padding to word boundaries. See the following section on ANSI C for more details for reading data generated by an external process such as C/C++.

Value

A list of values, one element for each type of R data.

ANSI_C

ANSI C struct's will typically have padding in cases where required by the language details and/or C programs. In general, if the struct on disk has padding, the use of bytes and offset are required to maintain alignment with the extraction and replacement code in mmap for R.

A simple example of this is where you have an 8-byte double (real64) and a 4-byte integer (int32). Created by a C/C++ program, the result will be a 16-byte struct - where the final 4-bytes will be padding.

To accomodate this from mmap, it is required to specify the corrected bytes (e.g. bytes=16 in this example). For cases where padding is not at the end of the struct (e.g. if an additional 8-byte double was added as the final member of the previous struct), it would also be necessary to correct the offset to reflect the internal padding. Here, the correct setting would be offset=c(0,8,16) - since the 4-byte integer will be padded to 8-bytes to allow for the final double to begin on a word boundary (on a 64 bit platform).

This is a general mechanism to adjust for offset - but requires knowledge of both the struct on disk as well as the generating process. At some point in the near future struct will attempt to properly adjust for offset if mmap is used on data created from outside of R.

It is important to note that this alignment is also dependent on the underlying hardware word size (size_t) and is more complicated than the above example.

Note

'struct"s can be thought of as 'rows' in a database. If many different types need always be returned together, it will be more efficient to store them together in a struct on disk. This reduces the number of page hits required to fetch all required data. Conversley, if individual columns are desired it will likely make sense to simply store vectors in seperate files on disk and read in with mmap individually as needed.

Note that not all behavior of struct extraction and replacement is defined for all virtual and real types yet. This is an ongoing development and will be completed in the near future.

Author(s)

Jeffrey A. Ryan

References

https://en.wikipedia.org/wiki/Struct_(C_programming_language) https://en.wikipedia.org/wiki/Data_structure_alignment

See Also

types

Examples

```
tmp <- tempfile()

f <- file(tmp, open="ab")
u_int_8 <- c(1L, 255L, 22L)  # 1 byte, valid range 0:255
int_8 <- c(1L, -127L, -22L)  # 1 byte, valid range -128:127
u_int_16 <- c(1L, 65000L, 1000L) # 2 byte, valid range 0:65+k
int_16 <- c(1L, 25000L, -1000L) # 2 byte, valid range -32k:32k
int_32 <- c(98743L, -9083299L, 0L) # 4 byte, standard R integer
float_32 <- c(9832.22, 3.14159, 0.00001)
cplx_64 <- c(1+0i, 0+8i, 2+2i)

# not yet supported in struct
char_ <- writeBin(as.raw(1:3), raw())
fixed_width_string <- c("ab","cd","ef")

for(i in 1:3) {
  writeBin(u_int_8[i],  f, size=1L)
  writeBin(int_8[i],    f, size=1L)
  writeBin(u_int_16[i], f, size=2L)
  writeBin(int_16[i],   f, size=2L)
  writeBin(int_32[i],   f, size=4L)
  writeBin(float_32[i], f, size=4L)  # store as 32bit - prec issues
  writeBin(float_32[i], f, size=8L)  # store as 64bit
  writeBin(cplx_64[i],  f)
  writeBin(char_[i], f)
  writeBin(fixed_width_string[i], f)
}
```

```
close(f)

m <- mmap(tmp, struct(uint8(),
                      int8(),
                      uint16(),
                      int16(),
                      int32(),
                      real32(),
                      real64(),
                      cplx(),
                      char(),  # also raw()
                      char(2)  # character array of n characters each
                     ))
length(m) # only 3 'struct' elements
str(m[])

m[1:2]

# add a post-processing function to convert some elements (rows) to a data.frame
extractFUN(m) <- function(x,i,...) {
                   x <- x[i]
                   data.frame(u_int_8=x[[1]],
                              int_8=x[[2]],
                              int_16=x[[3]],
                              int_32=x[[4]],
                              float_32=x[[5]],
                              real_64=x[[6]]
                             )
                 }
m[1:2]
munmap(m)

# grouping commonly fetched data by row reduces
# disk IO, as values reside together on a page
# in memory (which is paged in by mmap). Here
# we try 3 columns, or one row of 3 values.
# note that with structs we replicate a row-based
# structure.
#
# 13 byte struct
x <- c(writeBin(1L, raw(), size=1),
       writeBin(3.14, raw(), size=4),
       writeBin(100.1, raw(), size=8))
writeBin(rep(x,1e6), tmp)
length(x)
m <- mmap(tmp, struct(int8(),real32(),real64()))
length(m)
m[1]

# create the columns in seperate files (like a column
# store)
t1 <- tempfile()
t2 <- tempfile()
```

```
t3 <- tempfile()
writeBin(rep(x[1],1e6), t1)
writeBin(rep(x[2:5],1e6), t2)
writeBin(rep(x[6:13],1e6), t3)

m1 <- mmap(t1, int8())
m2 <- mmap(t2, real32())
m3 <- mmap(t3, real64())
list(m1[1],m2[1],m3[1])

i <- 5e5:6e5

# note that times are ~3x faster for the struct
# due to decreased disk IO and CPU cost to process
system.time(for(i in 1:100) m[i])
system.time(for(i in 1:100) m[i])
system.time(for(i in 1:100) list(m1[i],m2[i],m3[i]))
system.time(for(i in 1:100) list(m1[i],m2[i],m3[i]))
system.time(for(i in 1:100) {m1[i];m2[i];m3[i]}) # no cost to list()

# you can skip struct members by specifying offset and bytes
m <- mmap(tmp, struct(int8(),
                      #real32(),   here we are skipping the 4 byte float
                       real64(),
                       offset=c(0,5), bytes=13))
# alternatively you can add padding directly
n <- mmap(tmp, struct(int8(), pad(4), real64()))

pad(4)
pad(int32())

m[1]
n[1]

munmap(m)
munmap(n)
munmap(m1)
munmap(m2)
munmap(m3)
unlink(t1)
unlink(t2)
unlink(t3)
unlink(tmp)
```

Index

∗ **IO**
  C_types, 2
  struct, 14
∗ **data**
  mmap.csv, 9
∗ **iteration**
  struct, 14
∗ **manip**
  mmap.csv, 9
∗ **programming**
  struct, 14
∗ **utilities**
  make.fixedwidth, 5
  mmap, 6
  mmapFlags, 10
  mprotect, 12
  msync, 13
  sizeof, 13

as.char *(C_types)*, 2
as.cplx *(C_types)*, 2
as.cstring *(C_types)*, 2
as.Ctype, *14*
as.Ctype *(C_types)*, 2
as.int16 *(C_types)*, 2
as.int24 *(C_types)*, 2
as.int32 *(C_types)*, 2
as.int8 *(C_types)*, 2
as.list.Ctype *(struct)*, 14
as.mmap *(mmap)*, 6
as.real32 *(C_types)*, 2
as.real64 *(C_types)*, 2
as.struct *(struct)*, 14
as.uchar *(C_types)*, 2
as.uint16 *(C_types)*, 2
as.uint24 *(C_types)*, 2
as.uint8 *(C_types)*, 2

bits *(C_types)*, 2

C_types, 2
char *(C_types)*, 2
cplx *(C_types)*, 2
cstring *(C_types)*, 2

dim.mmap *(mmap)*, 6
dim<-.mmap *(mmap)*, 6
dimnames.mmap *(mmap)*, 6
dimnames<-.mmap *(mmap)*, 6

extractFUN *(mmap)*, 6
extractFUN<- *(mmap)*, 6

int16 *(C_types)*, 2
int24 *(C_types)*, 2
int32 *(C_types)*, 2
int64 *(C_types)*, 2
int8 *(C_types)*, 2
is.array.mmap *(mmap)*, 6
is.cstring *(C_types)*, 2
is.Ctype *(C_types)*, 2
is.mmap *(mmap)*, 6
is.na.mmap *(mmap)*, 6
is.struct *(struct)*, 14

logi32 *(C_types)*, 2
logi8 *(C_types)*, 2

make.fixedwidth, 5
mmap, 6, *10, 11*
mmap.csv, 9
mmapFlags, *8*, 10
mprotect, *11*, 12
msync, 13
munmap *(mmap)*, 6

nbytes *(C_types)*, 2

pad *(struct)*, 14
pagesize, *14*
pagesize *(mmap)*, 6

read.csv, *10*
real32 *(*C_types*)*, 2
real64 *(*C_types*)*, 2
replaceFUN *(*mmap*)*, 6
replaceFUN<- *(*mmap*)*, 6

sizeof, 13
sizeofCtypes *(*C_types*)*, 2
struct, 14

tempmmap *(*mmap*)*, 6
types, *16*
types *(*C_types*)*, 2

uchar *(*C_types*)*, 2
uint16 *(*C_types*)*, 2
uint24 *(*C_types*)*, 2
uint8 *(*C_types*)*, 2

writeBin, *3*