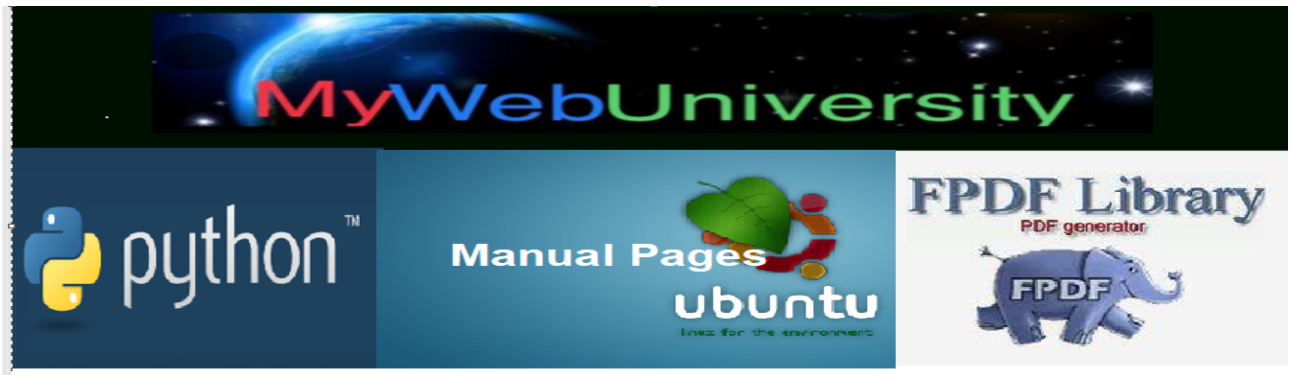

Full credit is given to the above companies including the OS that this PDF file was generated!

Linux Ubuntu 26.04 Manual Pages on command 'free.3'

$ man free.3

malloc(3) Library Functions Manual malloc(3)

NAME

malloc, free, calloc, realloc, reallocarray - allocate and free dynamic memory

LIBRARY

Standard C library (libc, -lc)

SYNOPSIS

```
#include <stdlib.h>

void *malloc(size_t size);

void free(void *_Nullable p);

void *calloc(size_t n, size_t size);

void *realloc(void *_Nullable p, size_t size);

void *reallocarray(void *_Nullable p, size_t n, size_t size);
```

Feature Test Macro Requirements for glibc (see feature_test_macros(7)):

reallocarray():

Since glibc 2.29:

_DEFAULT_SOURCE

glibc 2.28 and earlier:

_GNU_SOURCE

DESCRIPTION

malloc()

The malloc() function allocates size bytes and returns a pointer to the allocated memory.

The memory is not initialized. If size is 0, then malloc() returns a unique pointer value

that can later be successfully passed to free(). (See "Nonportable behavior" for portabil?

ity issues.)

free()

The free() function frees the memory space pointed to by p, which must have been returned by a previous call to malloc() or related functions. Otherwise, or if p has already been freed, undefined behavior occurs. If p is NULL, no operation is performed.

calloc()

The calloc() function allocates memory for an array of n elements of size bytes each and re? turns a pointer to the allocated memory. The memory is set to zero. If n or size is 0, then calloc() returns a unique pointer value that can later be successfully passed to free().

If the multiplication of n and size would result in integer overflow, then calloc() returns an error. By contrast, an integer overflow would not be detected in the following call to malloc(), with the result that an incorrectly sized block of memory would be allocated:

```
malloc(n * size);
```

realloc()

The realloc() function changes the size of the memory block pointed to by p to size bytes. The contents of the memory will be unchanged in the range from the start of the region up to the minimum of the old and new sizes. If the new size is larger than the old size, the added memory will not be initialized.

If p is NULL, then the call is equivalent to malloc(size), for all values of size.

If size is equal to zero, and p is not NULL, then the call is equivalent to free(p) (but see "Nonportable behavior" for portability issues).

Unless p is NULL, it must have been returned by an earlier call to malloc or related func? tions. If the area pointed to was moved, a free(p) is done.

reallocarray()

The reallocarray() function changes the size of (and possibly moves) the memory block pointed to by p to be large enough for an array of n elements, each of which is size bytes.

It is equivalent to the call

```
realloc(p, n * size);
```

However, unlike that realloc() call, reallocarray() fails safely in the case where the mul? tiplication would overflow. If such an overflow occurs, reallocarray() returns an error.

RETURN VALUE

The malloc(), calloc(), realloc(), and reallocarray() functions return a pointer to the al?

located memory, which is suitably aligned for any type that fits into the requested size or less. On error, these functions return NULL and set errno. Attempting to allocate more than PTRDIFF_MAX bytes is considered an error, as an object that large could cause later pointer subtraction to overflow.

The free() function returns no value, and preserves errno.

The realloc() and reallocarray() functions return NULL if p is not NULL and the requested size is zero; this is not considered an error. (See "Nonportable behavior" for portability issues.) Otherwise, the returned pointer may be the same as p if the allocation was not moved (e.g., there was room to expand the allocation in-place), or different from p if the allocation was moved to a new address. If these functions fail, the original block is left untouched; it is not freed or moved.

ERRORS

calloc(), malloc(), realloc(), and reallocarray() can fail with the following error:

ENOMEM Out of memory. Possibly, the application hit the RLIMIT_AS or RLIMIT_DATA limit de?
scribed in getrlimit(2). Another reason could be that the number of mappings created by the caller process exceeded the limit specified by /proc/sys/vm/max_map_count.

ATTRIBUTES

For an explanation of the terms used in this section, see attributes(7).

Interface	Attribute	Value
malloc(), free(), calloc(), realloc()	Thread safety	MT-Safe

STANDARDS

malloc()

free()

calloc()

realloc()

C23, POSIX.1-2024.

reallocarray()

POSIX.1-2024.

realloc(p, 0)

The behavior of realloc(p, 0) in glibc doesn't conform to any of C99, C11, POSIX.1-2001,

POSIX.1-2004, POSIX.1-2008, POSIX.1-2013, POSIX.1-2017, or POSIX.1-2024. The C17 specifica?
tion was changed to make it conforming, but that specification made it impossible to write
code that reliably determines if the input pointer is freed after realloc(p, 0), and C23
changed it again to make this undefined behavior, acknowledging that the C17 specification
was broad enough, so that undefined behavior wasn't worse than that.
reallocarray() suffers the same issues in glibc.
musl libc and the BSDs conform to all versions of ISO C and POSIX.1.
gnulib provides the realloc-posix module, which provides wrappers realloc() and reallocar?
ray() that conform to all versions of ISO C and POSIX.1.
There's a proposal to standardize the BSD behavior: https://www.open-std.org/jtc1/sc22/wg14/
www/docs/n3621.txt.

HISTORY

malloc()
free()
calloc()
realloc()
POSIX.1-2001, C89.
reallocarray()
glibc 2.26. OpenBSD 5.6, FreeBSD 11.0.

malloc() and related functions rejected sizes greater than PTRDIFF_MAX starting in glibc
2.30.
free() preserved errno starting in glibc 2.33.

realloc(p, 0)

C89 was ambiguous in its specification of realloc(p, 0). C99 partially fixed this.
The original implementation in glibc would have been conforming to C99. However, and ironi?
cally, trying to comply with C99 before the standard was released, glibc changed its behav?
ior in glibc 2.1.1 into something that ended up not conforming to the final C99 specifica?
tion (but this is debated, as the wording of the standard seems self-contradicting).

NOTES

By default, Linux follows an optimistic memory allocation strategy. This means that when
malloc() returns non-NULL there is no guarantee that the memory really is available. In
case it turns out that the system is out of memory, one or more processes will be killed by
the OOM killer. For more information, see the description of /proc/sys/vm/overcommit_memory

and /proc/sys/vm/oom_adj in proc(5), and the Linux kernel source file Documentation/vm/over?
commit-accounting.rst.

Normally, malloc() allocates memory from the heap, and adjusts the size of the heap as re?
quired, using sbrk(2). When allocating blocks of memory larger than MMAP_THRESHOLD bytes,
the glibc malloc() implementation allocates the memory as a private anonymous mapping using
mmap(2). MMAP_THRESHOLD is 128 kB by default, but is adjustable using mallopt(3). Prior to
Linux 4.7 allocations performed using mmap(2) were unaffected by the RLIMIT_DATA resource
limit; since Linux 4.7, this limit is also enforced for allocations performed using mmap(2).

To avoid corruption in multithreaded applications, mutexes are used internally to protect
the memory-management data structures employed by these functions. In a multithreaded ap?
plication in which threads simultaneously allocate and free memory, there could be con?
tention for these mutexes. To scalably handle memory allocation in multithreaded applica?
tions, glibc creates additional memory allocation arenas if mutex contention is detected.
Each arena is a large region of memory that is internally allocated by the system (using
brk(2) or mmap(2)), and managed with its own mutexes.

If your program uses a private memory allocator, it should do so by replacing malloc(),
free(), calloc(), and realloc(). The replacement functions must implement the documented
glibc behaviors, including errno handling, size-zero allocations, and overflow checking;
otherwise, other library routines may crash or operate incorrectly. For example, if the re?
placement free() does not preserve errno, then seemingly unrelated library routines may fail
without having a valid reason in errno. Private memory allocators may also need to replace
other glibc functions; see "Replacing malloc" in the glibc manual for details.

Crashes in memory allocators are almost always related to heap corruption, such as overflow?
ing an allocated chunk or freeing the same pointer twice.

The malloc() implementation is tunable via environment variables; see mallopt(3) for de?
tails.

Nonportable behavior

The behavior of these functions when the requested size is zero is glibc specific; other im?
plementations may return NULL without setting errno, and portable POSIX programs should tol?
erate such behavior. See realloc(3p).

POSIX requires memory allocators to set errno upon failure. However, the C standard does
not require this, and applications portable to non-POSIX platforms should not assume this.

Portable programs should not use private memory allocators, as POSIX and the C standard do

not allow replacement of malloc(), free(), calloc(), and realloc().

BUGS

Programmers would naturally expect by induction that realloc(p, size) is consistent with free(p) and malloc(size), as that is the behavior in the general case. This is not explic? itly required by POSIX.1-2024 or C11, but all conforming implementations are consistent with that.

The glibc implementation of realloc() is not consistent with that, and as a consequence, it is dangerous to call realloc(p, 0) in glibc.

A trivial workaround for glibc is calling it as realloc(p, size?size:1).

The workaround for reallocarray() in glibc ?which shares the same bug? would be reallocarray(p, n?n:1, size?size:1).

EXAMPLES

```
#include <err.h>
#include <stddef.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MALLOCARRAY(n, type)  ((type *) my_mallocarray(n, sizeof(type)))
#define MALLOC(type)          MALLOCARRAY(1, type)
static inline void *my_mallocarray(size_t n, size_t size);
int
main(void)
{
    char  *p;
    p = MALLOCARRAY(32, char);
    if (p == NULL)
        err(EXIT_FAILURE, "malloc");
    strlcpy(p, "foo", 32);
    puts(p);
}
static inline void *
my_mallocarray(size_t n, size_t size)
{
```

```
    return reallocarray(NULL, n, size);
}
```

SEE ALSO

valgrind(1), brk(2), mmap(2), alloca(3), malloc_get_state(3), malloc_info(3), malloc_trim(3), malloc_usable_size(3), mallopt(3), mcheck(3), mtrace(3), posix_memalign(3)

For details of the GNU C library implementation, see https://sourceware.org/glibc/wiki/MallocInternals.