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 *ptr);
void *calloc(size_t nmemb, size_t size);
void *realloc(void *ptr, size_t size);
void *reallocarray(void *ptr, size_t nmemb, 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 v alue that can later be successfully passed to **free**(). (See "Nonportable behavior" for portability issues.)

free()

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

calloc()

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

If the multiplication of *nmemb* 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(nmemb * size);
```

realloc()

The **realloc**() function changes the size of the memory block pointed to by *ptr* 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 *ptr* is NULL, then the call is equivalent to *malloc(size)*, for all values of *size*.

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

Unless *ptr* is NULL, it must have been returned by an earlier call to **malloc** or related functions. If the area pointed to was moved, a *fr ee(ptr)* is done.

reallocarray()

The **reallocarray**() function changes the size of (and possibly moves) the memory block pointed to by *ptr* to be large enough for an array of *nmemb* elements, each of which is *size* bytes. It is equivalent to the call

```
realloc(ptr, nmemb * size);
```

However, unlike that **realloc**() call, **reallocarray**() fails safely in the case where the multiplication 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 allocated 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 *ptr* 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 *ptr* if the allocation was not moved (e.g., there was room to expand the allocation in-place), or different from *ptr* 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 described in **getrlimit**(2).

VERSIONS

reallocarray() was added in glibc 2.26.

malloc() and related functions rejected sizes greater than **PTRDIFF_MAX** starting in glibc 2.30.

free() preserved *errno* starting in glibc 2.33.

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**(): POSIX.1-2001, POSIX.1-2008, C99.

reallocarray() is a nonstandard extension that first appeared in OpenBSD 5.6 and FreeBSD 11.0.

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/overcommit−accounting.rst*.

Normally, **malloc**() allocates memory from the heap, and adjusts the size of the heap as required, 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 application in which threads simultaneously allocate and free memory, there could be contention for these mutexes. To scalably handle memory allocation in multithreaded applications, 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 replacement *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 overflowing an allocated chunk or freeing the same pointer twice.

The **malloc**() implementation is tunable via environment variables; see **mallopt**(3) for details.

Nonportable behavior

The behavior of these functions when the requested size is zero is glibc specific; other implementations may return NULL without setting *errno*, and portable POSIX programs should tolerate 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**().

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 nmemb, 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 nmemb, size_t size)
{
    return reallocarray(NULL, nmemb, 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⟩.