`malloc_allocated_size`: Querying What the Machine Knows

Document Number: N3869
Author: Fred Veldmeijer, Fontys University of Applied Sciences
Date: 2026–04–10
Audience: WG14
Proposal Category: New Features
Target Audience: Developers

Abstract

C provides access to what the machine has to offer. Yet the current standard forces programmers to manually track allocation sizes the machine already knows – creating redundant state that is a source of bugs. Worse, allocators routinely provide more memory than requested, but this excess memory remains inaccessible – causing unnecessary reallocations despite adequate storage, degrading performance and increasing heap fragmentation.

To address these issues, we propose `malloc_allocated_size`: a function that exposes the size the allocator already tracks internally, standardizing existing practice found on BSD, Linux, Windows, and macOS/iOS.

1. Problem and Rationale

1.1 Mandatory Redundant State

C normally exposes what the system already knows, e.g. it provides `sizeof` for types because the compiler knows their size, and `strlen` for strings because traversal reveals their length. Yet, for heap allocations, the current standard offers no querying mechanism, leading to redundancy:

```
struct buffer {
    char *data;
    size_t length;
    size_t allocated_size;  // The allocator already knows this
};
```

The allocator tracks the allocated size to implement the functions `free` and `realloc`. Without querying, programmers must manually duplicate this tracking, introducing the risk that tracked size diverges from reality after `realloc` failures or programming mistakes.

1.2 Cost of Inaccessible Allocated Memory

C should grant full access to the memory the machine provides. Heap allocation does not:

```
char *p = malloc(65);
// Allocator may have actually given 128 bytes due to alignment
// But programmer cannot safely use bytes [65..128)
// Those bytes are allocated but inaccessible
```

Most allocators work with fixed capacity sizes (powers of two, fibonacci sequences, etc.) for efficiency. A 65-byte request might receive a 128-byte block. The additional 63 bytes belong to this allocation, they consume memory, they will be freed when the pointer is freed – yet the current standard provides no way to discover or use them.

1.3 Cost of Unnecessary Reallocations

Without querying, programmers cannot detect when capacity exists, leading to unnecessary reallocations:

```
char *p = malloc(65);   // Actually got 128 bytes
p = realloc(p, 100);    // Must reallocate despite having capacity
                        // Copies 65 bytes unnecessarily
                        // Fragments memory
                        // May return null despite adequate storage
```

If the allocator grants 128 bytes, the programmer should be able to discover and use all 128 bytes.

1.4 Cost of Manual Tracking

When size is tracked, on 64-bit systems this costs 8 bytes per allocation. For a typical `struct buffer`, this is 24 bytes total (pointer + length + size) versus 16 bytes with querying (pointer + length).

The savings are larger in the common case of sentinel-terminated data. For example, a null-terminated string needs no metadata at all: the pointer provides the address, `strlen` provides the length, and `malloc_allocated_size` provides the size. Separate tracking is eliminated entirely.

2. Solution: Querying

2.1 Proposal

We propose adding one function to `<stdlib.h>`:

```
size_t malloc_allocated_size(const void *ptr);
```

This function returns the number of bytes actually available, not just the number requested.

2.2 Restoring C Principles

Eliminate mandatory redundancy: Query the allocator instead of tracking capacity separately.

```
// Before: Required redundancy
struct buffer {
    char *data;
    size_t allocated_size;  // Must track manually
};
struct buffer b;
b.data = malloc(100);
b.allocated_size = 100;

// After: Query when needed
char *data = malloc(100);
size_t actual = malloc_allocated_size(data);  // O(1) lookup
```

Reduce unnecessary work: Avoid reallocation when capacity exists.

```
bool append(char **buf, const char *str) {
    size_t current = strlen(*buf);
    size_t needed = current + strlen(str) + 1;
    size_t actual = malloc_allocated_size(*buf);

    if (needed > actual) {
        size_t new_capacity = actual * 2;
        if (needed > new_capacity) new_capacity = needed;

        char *tmp = realloc(*buf, new_capacity);
        if (!tmp) return false;
        *buf = tmp;
    }
    strcpy(*buf + current, str);
    return true;
}
```

Access all allocated memory: Use the full allocation the allocator actually provided.

```
char *p = malloc(100);
size_t actual = malloc_allocated_size(p);  // e.g., 128
memset(p, 0, actual);                      // Safely use all bytes in [p, p + actual)
```

3. Existing Practice

This functionality exists on every major platform. The C standard would not be introducing a new concept but codifying existing practice:

Platform	Function	Header
macOS/iOS	`malloc_size`	`<malloc/malloc.h>`
OpenBSD	`malloc_size`	`<stdlib.h>`
FreeBSD	`malloc_usable_size`	`<malloc_np.h>`
Linux (glibc)	`malloc_usable_size`	`<malloc.h>`
Windows (MSVC)	`_msize`	`<malloc.h>`
jemalloc	`malloc_usable_size`	`<jemalloc/jemalloc.h>`
mimalloc	`mi_usable_size`	`<mimalloc.h>`

Both jemalloc [5] and mimalloc [6] are high-performance drop-in allocators widely deployed in production. The concept is ubiquitous; only the names differ.

The proposed functionality has been implemented and is testable on the above platforms by including:

```
#if defined(__APPLE__)
    #include <malloc/malloc.h>
    #define malloc_allocated_size malloc_size
#elif defined(__OpenBSD__)
    #include <stdlib.h>
    #define malloc_allocated_size malloc_size
#elif defined(__FreeBSD__)
    #include <malloc_np.h>
    #define malloc_allocated_size malloc_usable_size
#elif defined(__linux__)
    #include <malloc.h>
    #define malloc_allocated_size malloc_usable_size
#elif defined(_MSC_VER) || defined(__MINGW32__)
    #include <malloc.h>
    #define malloc_allocated_size _msize
#else
    #error "malloc_allocated_size not available on this platform"
#endif
```

Standardization eliminates this conditional compilation.

4. Relationship to C23's free_sized

C23 introduced the `free_sized` function [1] for verification: the programmer manually tracks allocation sizes and the allocator validates them at deallocation:

```
// Verification (C23)
void *p = malloc(100);
size_t tracked_size = 100;    // Manual tracking
// ...
free_sized(p, tracked_size);  // Validated deallocation
```

The proposed function enables querying the allocator instead of tracking capacity separately:

```
// Querying (Proposed)
void *p = malloc(100);
// ...
size_t actual = malloc_allocated_size(p);  // Query when needed
free(p);
```

These target different needs:

Feature	`free_sized` (C23)	`malloc_allocated_size` (Proposed)
Strategy	Verification	Querying
Responsibility	Programmer tracks size	Allocator reports allocated size
Primary benefit	Detects corruption	Reduces overhead
Use case	Security-critical code	Dynamic containers

Verification suits security-critical code requiring redundant checks. Querying suits performance-sensitive code where tracking adds overhead and the allocator's internal bookkeeping suffices.

5. Design Decisions

Naming: Existing platforms expose this functionality under several names: `malloc_size` (macOS/iOS) [4], `malloc_usable_size` (glibc, FreeBSD, jemalloc) [3, 5], and `_msize` (Windows). We prioritize precision over elegance, and propose `malloc_allocated_size`, following the naming pattern established by C23's allocator family additions (`free_sized`, `free_aligned_sized`): the `malloc_` prefix identifies it as part of the malloc allocator family, and `allocated_size` unambiguously describes what the function returns – the size of what was actually allocated, as opposed to the size that was requested.

Undefined Behavior for Invalid Pointers: Like the `free` and `realloc` functions, this function requires the exact pointer returned by `malloc`, `calloc`, `realloc`, or `aligned_alloc`. Interior pointers (e.g., `ptr + 10`), freed pointers, and pointers to stack or static storage cause undefined behavior.

This aligns with C's philosophy: the standard specifies what works, not what fails. Implementations are permitted to detect invalid pointers and produce a diagnostic, as some already do in debug configurations, but the standard imposes no requirement to do so.

Thread Safety: The function performs read-only access to allocator metadata. Calling this function does not introduce data races, provided the memory block pointed to by `ptr` is not being concurrently deallocated or reallocated by another thread. Because the function does not access the contents of the allocated memory, it is perfectly safe to query the size even while the object's contents are being concurrently modified.

6. Implementation Feasibility

Implementation cost is negligible: All allocators store the allocated size to support `free` and `realloc` – this function exposes existing metadata. Common implementation strategies include:

- Reading a header word immediately preceding the returned pointer
- Mapping the address to page or slab metadata
- Looking up the allocated size in a hash table (for segregated-fit allocators)

Runtime cost is minimal: Retrieving the allocated size is an O(1) operation, typically a single memory read or hash lookup. No additional bookkeeping or locking is required beyond what `free` and `realloc` already perform.

Sanitizer Compatibility is Straightforward: Memory debugging tools (e.g., AddressSanitizer) currently use the requested size to detect out-of-bounds access. Implementations of these tools shall intercept `malloc_allocated_size` to dynamically unpoison the excess bytes up to the returned size, bringing the valid range in sync with the allocator's reality. This follows established sanitizer practice, as tools already intercept `malloc`, `calloc`, `realloc`, and `free` [7], and intercepting one additional function is a minimal extension.

7. Specification

Proposed addition to C2y §7.24.3 Memory management functions [2].

Synopsis

```
#include <stdlib.h>

size_t malloc_allocated_size(const void *ptr);
```

Description

Returns the number of bytes of storage available at the address pointed to by `ptr`. If `ptr` is a null pointer, the function returns zero. Otherwise, `ptr` shall be a pointer to space previously allocated by `aligned_alloc`, `calloc`, `malloc`, or `realloc` that has not yet been deallocated by a call to `free` or `realloc`. If this requirement is not met, the behavior is undefined.

The returned size is at least as large as the most recent request, and may be larger. The application may safely access the space up to the returned size.

The values of any bytes beyond the most recent request are indeterminate. This applies even if the space was allocated using `calloc`, which only guarantees initialization of the requested size to all bits zero.

Example

```
#include <stdlib.h>
#include <string.h>  // For memset

void func(void) {
    // Allocation failures are ignored for brevity in this example

    size_t actual = malloc_allocated_size(NULL); // actual == 0

    char *buf = malloc(100);
    actual = malloc_allocated_size(buf);         // actual >= 100

    memset(buf, 0, actual);                      // Initialize all allocated memory

    buf = realloc(buf, 200);
    actual = malloc_allocated_size(buf);         // actual >= 200

    actual = malloc_allocated_size(buf + 50);    // UNDEFINED BEHAVIOR (interior pointer)

    int n;
    actual = malloc_allocated_size(&n);          // UNDEFINED BEHAVIOR (automatic storage)

    free(buf);
    actual = malloc_allocated_size(buf);         // UNDEFINED BEHAVIOR (freed pointer)
}
```

8. Conclusion

The case for `malloc_allocated_size` rests on two points. First, C exposes what the machine knows: `sizeof` for types, `strlen` for strings – but not allocated size, despite the allocator tracking it for every allocation. This is not just redundant; it is a source of bugs when the programmer's record diverges from reality. Second, invisible capacity forces unnecessary work and storage: allocators routinely provide more memory than requested, but without querying this excess memory remains inaccessible.

Every major platform exposes allocated size: BSD, Linux, Windows, and macOS/iOS all provide this functionality. The C standard would not be introducing a new concept but codifying existing practice.

We recommend adding `malloc_allocated_size` to C. The machine knows. The programmer should too.

References

[1] ISO/IEC JTC1/SC22/WG14, N2699: Sized Memory Deallocation, 2021,
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n2699.htm

[2] ISO/IEC 9899:2026, Programming languages – C (working draft N3783),
https://www.open-std.org/jtc1/sc22/wg14/www/docs/n3783.pdf

[3] Free Software Foundation, GNU C Library Reference Manual, `malloc_usable_size`,
https://www.gnu.org/software/libc/manual/html_node/Heap-Consistency-Checking.html

[4] Apple Inc., Developer Documentation: `malloc_size`,
https://developer.apple.com/documentation/malloc/

[5] Jason Evans, jemalloc documentation,
https://nxmnpg.lemoda.net/3/jemalloc

[6] Microsoft Research, mimalloc documentation,
https://microsoft.github.io/mimalloc/overrides.html

[7] Microsoft, AddressSanitizer runtime reference,
https://learn.microsoft.com/en-us/cpp/sanitizers/asan-runtime