Proposal for C2y

WG14 N3899
Title: Memory allocation with size feedback v4
Author, affiliation: Chris Kennelly, Google; Justin King, Google; Charles Munger, Google
Date: 2026-05-20
Proposal category: New features
Target audience: Developers working on on C codebases, allocator developers

Abstract: Provide a function that returns a size along with the allocated memory block, to standardize a number of existing extensions and permit more effective use of memory.

Prior art: Numerous allocator extensions

Memory allocation with size feedback

Summary of changes since N3880 (v3)

- Fixed incorrect function name in aligned_alloc_at_least
- Moved alloc_result_t struct description to General to address wording feedback
- New design discussion section on whether to permit additional members in the alloc_result_t struct, specified “shall contain exactly”

Summary of changes since N3849 (v2)

- Adapted suggested wording to apply to N3854 as a base
- NOTEs changed to recommended practice or normative description
- Added details of cost/benefit to Background & Motivation
- Clarified behavior when returning a non-NULL pointer from alloc_at_least(0)
- Changed specification to always specify 0 size when the pointer is not usable (NULL or zero allocated size)
- Adjusted wording based on committee feedback:
    - Moved struct/typedef description around to match other declared structs and their usage
    - Fixed pseudocode

Summary of changes since N3813 (initial revision)

- Added real-world examples of use and limitations of existing vendor extensions in widely used C libraries
- Added stdc_alloc_at_least/stdc_aligned_alloc_at_least as naming options
- Added discussion on strategy for returning size - struct vs in/out param
- Size is no longer set to 0 on allocation failure

Summary

Depending on the malloc implementation (referred to as the allocator), the requested allocation size may be less than the actual allocated size. This results in memory being unused which is equal to the difference between the requested allocation size and the actual allocated size. Some allocators, such as the one used by glibc, return mmap’ed memory directly when the requested size is over a certain threshold. In the worst case, this can result in close to a full page size of memory being unused.

To make use of this unused memory, various allocator extensions may be used. These extensions fall into two categories: querying size classes before allocating (`malloc_good_size`/`nallocx`) and querying the actual allocated size after allocating (`malloc_usable_size`/`malloc_size`/`_msize`). Some allocators support both, one, or rarely none. Both approaches have shortcomings for this use case which are explained in detail later. We propose standardizing on a third approach which avoids these shortcomings, by introducing sibling functions for `malloc` and `aligned_alloc` which return the actual size of the allocation. This is also known as size feedback.

Background & Motivation

Querying Size Classes

As mentioned in the summary, some allocators have exposed extensions for querying size classes before allocating. The most prominent extensions are: `nallocx` (jemalloc/tcmalloc) and `malloc_good_size` (Apple). No similar extension is exposed by glibc or Microsoft's CRT. This can be used as a basis for implementing size feedback, however we have avoided standardizing on this approach for a few reasons. This approach has a couple downsides as it requires two function calls to perform an allocation: `nallocx`/`malloc_good_size` followed by `malloc`. The first downside is that the logic performed by `nallocx`/`malloc_good_size` to determine the size class will have to be repeated by `malloc` as well, as it is unaware a size class has already been determined. The second downside is the potential mismatch between the size class picked by `nallocx`/`malloc_good_size` and `malloc`. For example, the allocator could choose to return a slightly larger memory block that is in its freelist which would result in unused memory.

Querying Allocation Size

As mentioned in the summary, some allocators have exposed extensions for querying the actual size after allocating. The most prominent extensions are: `malloc_usable_size` (glibc/jemalloc/tcmalloc/mimalloc), `malloc_size` (Apple), and `_msize` (Microsoft's CRT). This can be used as a basis for implementing size feedback, however we have avoided standardizing on this approach for a few reasons. This approach requires three function calls to perform an allocation: `malloc` followed by `malloc_usable_size`/`malloc_size`/`_msize` potentially followed by `realloc`.

1. The logic for determining the size of the allocation by `malloc_usable_size`/`malloc_size`/`_msize` is the same logic used by `free` and *avoided by* `free_sized`/`free_aligned_sized`. For allocators that store the size anywhere other than immediately preceding the allocated pointer, this can be expensive to obtain.

2. The program cannot actually make use of the potential extra memory reported by `malloc_usable_size`/`malloc_size`/`_msize` without also calling `realloc` (or an allocator extension equivalent) to expand the allocation. Some allocators do not explicitly forbid it, but glibc's allocator and Address Sanitizer's allocator explicitly do. The call to `realloc` will also perform the same size lookup required by `malloc_usable_size`/`malloc_size`/`_msize`. Android's version of the Scudo allocator intentionally reports the requested size rather than the actual size from `malloc_usable_size`, so that it doesn't have to copy the full set of bytes into a new allocation when `realloc` is used. There is also no guarantee that a call to `realloc(ptr, malloc_usable_size(ptr))` will not perform a costly operation to actually reallocate the backing storage rather than merely updating some metadata.

Allocators that store metadata adjacent to the allocated space (such as immediately before the pointer they returned) can cheaply satisfy queries for the size of an allocation; but in allocators that store metadata in other data structures for security hardening or to reduce overhead will incur greater expense. Making matters worse, allocators that don't store the requested allocation size at all (instead tracking the size rounded to a size class or internal alignment) are forced to reveal implementation details in order to implement this API, even if they would prefer not to.

Size Feedback

Returning the size at the time of allocation is implementable at very low performance cost regardless of the implementation details of the allocator; allocators that will never return a value other than the requested size can do so when the requested size is readily available. Allocators using size classes without inline metadata can return the available space directly without a pointer-to-size data structure lookup. Allocators that may return larger blocks from a freelist can return the size when the block is obtained.

Use Cases

Dynamic Array

A common data structure is a resizable array that grows on insertions by reallocating. To achieve amortized linear time, implementations allocate larger blocks of memory than they actually need to hold the current set of items - if they could use additional space at low/no cost, fewer resizes would be needed, minimizing copying and allocator churn.

Arena/Bump allocators

Repeated small allocations with the same lifetime can be grouped together by using `malloc` to obtain a large block of memory, and then adding the size of each small allocation to the returned pointer until it is full. If needed, allocate another block and repeat. However, chosen block sizes

can waste substantial memory if the requested sizes are a poor fit for underlying size classes used by `malloc`.

I/O Buffers

When streaming data to/from a file, the network, or some other system, we often don't know how much data we'll move in advance. Larger buffers use more memory but often increase throughput and decrease syscall and device overhead; if we can obtain a larger buffer than required at no additional cost, we want to do so.

Examples

- In SQLite (a database engine written in C), multiple different internal allocation strategies are used, that start by obtaining larger blocks of memory from the system allocator. They can be divided into smaller chunks as a per-connection pool allocator, or used as scratch buffers for other operations. Both of these benefit from making use of extra available memory. Prior to version 3.5.0 of SQLite, it used `malloc_usable_size` to take advantage of this space; however, this behavior was changed because (1) `malloc_usable_size` was slow as described earlier in this paper and (2) it's not safe to use the extra space indicated by `malloc_usable_size` without a `realloc` which invalidates existing pointers and also may incur a large performance cost.
- In upb (a encoder/decoder for the Protocol Buffers serialization format written in C), memory is managed by arenas, which are collections of chunks of memory obtained from the system allocator with a shared lifetime. When decoding, small chunks of memory needed for nodes in the parsed representation are allocated out of the current block. The sizes of the allocations performed are dependent on what information is being parsed, and are not predictable in advance. When no room is available, a new, larger block is obtained. Various heuristics are used to choose the size of these blocks; larger ones increase overhead but improve locality and amortize system allocator calls among more small allocations. Importantly on devices like phones, watches, or TVs, making use of the "extra" memory between the requested size and actual size results in substantial reductions in retained memory for applications.

Workarounds

Various allocators provide facilities for querying the anticipated size of a future allocation so that a request can be made precisely for it, or querying the actual size (so the allocation can be realloc'd in place).

- Apple: `malloc_good_size`+`malloc`, `malloc_size`
- jemalloc/tcmalloc: `nallocx`+`malloc`
- dlmalloc: `realloc_in_place`
- mimalloc: `mi_expand`

- musl/glibc/many other linux libcs: `malloc_usable_size`
- Microsoft: `_msize`, `_expand`

Suggested Wording

7.25 General utilities `<stdlib.h>`

7.25.1 General

The types declared are size_t and wchar_t (both described in 7.22), once_flag (described in 7.30),

```
None
alloc_result_t
```

which is a structure type that is the type of the value returned by the `alloc_at_least` and `aligned_alloc_at_least` functions,

```
None
div_t
```

which is a structure type that is the type of the value returned by the `div` function,

```
None
ldiv_t
```

which is a structure type that is the type of the value returned by the `ldiv` function, and

```
None
lldiv_t
```

which is a structure type that is the type of the value returned by the `lldiv` function.

The `alloc_result_t` structure shall contain exactly the following members, in either order:

```
None
void* ptr;
size_t size;
```

The value of `size` is 0 if `ptr` is a null pointer.

7.25.4 Memory management functions

7.25.4.1 General

The order and contiguity of storage allocated by successive calls to the `aligned_alloc`, **`aligned_alloc_at_least`**, **`alloc_at_least`**, `calloc`, `malloc`, and `realloc` functions is unspecified. The pointer returned if the allocation succeeds is suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and size less than or equal to the size requested**, or in the case of `aligned_alloc_at_least` and `alloc_at_least`, the size returned.** It can then be used to access such an object or an array of such objects in the space allocated (until the space is explicitly deallocated). The lifetime of an allocated object extends from the allocation until the deallocation. Each such allocation shall yield a pointer to an object disjoint from any other object. The pointer returned points to the start (lowest byte address) of the allocated space. If the space cannot be allocated, a null pointer is returned. If the size of the space requested is zero, the behavior is implementation-defined: either a null pointer is returned to indicate an error, or the behavior is as if the size were some nonzero value, except that the returned pointer shall not be used to access an object.

7.25.4.6 The `alloc_at_least` function

Synopsis

```
C/C++
#include <stdlib.h>
alloc_result_t alloc_at_least(size_t min_size);
```

Description

The `alloc_at_least` function allocates space for an object whose minimum size is specified by `min_size`, whose actual size is specified by the `size` member of its return value, and whose bytes have unspecified values.

NOTE: A conforming implementation may simply call `malloc`, place the resulting address in the `ptr` member of its return value, and `min_size` or 0 in the `size` member of its return value depending on whether `malloc` returned a null pointer.

Recommended practice

An allocator should return an actual size larger than `min_size` if the additional memory could not be otherwise put to use until the returned pointer was freed.

`alloc_at_least(n).ptr` is usable everywhere that the result of `malloc(n)` is, however using `alloc_at_least(n)` and ignoring the resulting size should be avoided. If the resulting size is not used, prefer `malloc`.

Returns

The `alloc_at_least` function returns a structure of type `alloc_result_t` comprising both a pointer to the newly allocated space and the resulting size of that space. If `min_size` was 0 or the allocation failed, the resulting size is 0 and the pointer is a null pointer. Otherwise the resulting size is guaranteed to be greater than or equal to the `min_size`.

7.25.4.7 The `aligned_alloc_at_least` function

Synopsis

```
C/C++
#include <stdlib.h>
alloc_result_t aligned_alloc_at_least(size_t alignment,
                                      size_t min_size);
```

Description

The `aligned_alloc_at_least` function allocates space for an object whose alignment is specified by `alignment`, whose minimum size is specified by `min_size`, whose actual size is specified by the `size` member of its return value, and whose bytes have unspecified values. If the value of alignment is not a valid alignment supported by the implementation the function shall fail by returning a null pointer as the `ptr` member of `alloc_result_t`.
NOTE: A conforming implementation may simply call `aligned_alloc`, place the resulting address in the `ptr` member of its return value, and `min_size` or 0 in the `size` member of its return value depending on whether `aligned_alloc` returned a null pointer.

Recommended practice

An allocator should return an actual size larger than `min_size` if the additional memory could not be otherwise put to use until the returned pointer was freed.

NOTE: `aligned_alloc_at_least(a, n).ptr` is usable everywhere that the result of `aligned_alloc(a, n)` is, however using `aligned_alloc_at_least(a, n)` and ignoring the resulting size should be avoided. If the resulting size is not used, prefer `aligned_alloc`.

Returns

The `aligned_alloc_at_least` function returns a structure of type `alloc_result_t` comprising both a pointer to the newly allocated space and the resulting size of that space. If `min_size` was 0 or the allocation failed, the resulting size is 0 and the pointer is a null pointer. Otherwise the resulting size is guaranteed to be greater than or equal to the `min_size`. If the pointer is not `NULL`, the pointer is aligned to at least the requested alignment.

7.25.4.~~6~~8 The `free_sized` function

Description

If `ptr` is a null pointer or the result obtained from a call to `malloc`, `realloc`, or `calloc`, where `size` ~~size~~ is equal to the requested allocation size, this function is equivalent to `free(ptr)`. **If `ptr` was the resulting pointer obtained from a call to `alloc_at_least(n)` and `m >= size && size >= n`, where `m` is the `size` returned in `alloc_result_t`, this function is equivalent to `free(ptr)`.** Otherwise, the behavior is undefined. The result of an `aligned_alloc` **or `aligned_alloc_at_least`** call may not be passed to `free_sized`.

7.25.4.~~7~~9 The `free_aligned_sized` function

Description

If `ptr` is a null pointer or the result obtained from a call to `aligned_alloc`, where `alignment` is equal to the requested allocation alignment and `size` is equal to the requested allocation size, this function is equivalent to `free(ptr)`. **If `ptr` was allocated by `aligned_alloc_at_least(alignment, n)` and `m >= size && size >= n`, where `m` is the `size` returned in `alloc_result_t`, this function is equivalent to `free(ptr)`.** Otherwise, the behavior is undefined. The result of a `malloc`, `calloc`, ~~**or**~~ `realloc`**, or `alloc_at_least`** call may not be passed to `free_aligned_sized`.

Design Discussion

Naming Bikeshed

The wording for this proposal uses `alloc_at_least`, but we have selected several alternatives for consideration:

- `alloc_at_least`/`aligned_alloc_at_least`
- `malloc_at_least`/`aligned_alloc_at_least`

- `sized_alloc`/`aligned_sized_alloc`
- `sized_malloc`/`aligned_sized_alloc`
- `size_returning_alloc`/`size_returning_aligned_alloc`
- `size_returning_malloc`/`sized_returning_aligned_alloc`
- `min_size_alloc`/`aligned_min_size_alloc`
- `stdc_alloc_at_least/stdc_aligned_alloc_at_least`

Choice of Return Type

The proposed API returns a struct, which is atypical although not unheard of in C's standard library, `div_t` being a prior example. The following alternative structures were considered:

```
C/C++
void* alloc_at_least(size_t* size);
```

The former is somewhat similar to atomic_compare_exchange_strong's expected parameter. Like that parameter, taking an address precludes passing a literal or expression for the size, and the developer has to pass an address, usually of a local variable, instead. If the original value is still needed later, it has to be stored somewhere else.

```
C/C++
void* alloc_at_least(size_t size, size_t* actual_size);
```

An advantage of these styles is that they avoid standardizing the struct return type. Some examples of usage with a struct with a flexible array member follow:

```
C/C++
typedef struct {
  size_t count;
  uint64_t data[];
} T;
```

Returning struct:

```
C/C++

alloc_result_t r = alloc_at_least(offsetof(T, data[4]));
if (r.ptr) {
  T* result = r.ptr;
  result->count = (r.size - offsetof(T, data[0])) /
sizeof(uint64_t);
}
```

Taking pointer to the requested size and updating it with the allocated size:

```C/C++
size_t size = offsetof(T, data[4]);
T* result = alloc_at_least(&size);
if (result) {
  result->count = (size - offsetof(T, data[0])) / sizeof(uint64_t);
}
```

Taking the requested size by value, and a pointer to the allocated size:

```C/C++
size_t actual_size;
T* result = alloc_at_least(offsetof(T, data[4]), &actual_size);
if (result) {
  result->count = (actual_size - offsetof(T, data[0])) /
sizeof(uint64_t);
}
```

Each of these is basically the same amount of code, and offer similar opportunities for programmer error. However, there are some practical concerns. With the out-param style, an ABI is effectively required to load and store. When returning a struct by value, some ABIs will return the small struct on the stack at cost similar to putting a size_t on the stack and passing its address. However, implementers may pass the struct's members in registers, and some do - notably including aarch64, x86_64 using the System V ABI, and Risc-V. WebAssembly/Wasm can also take advantage of multiple return values.[1]

A key consideration here is the motivation of a library author intending to write portable code across multiple platforms. Some targets may never return memory in excess of what's requested; in those cases, any overhead compared to an ordinary malloc call is pure waste. Returning the struct by value permits an implementation greater freedom to use an optimal ABI, which in turn may improve developer confidence in using alloc_at_least liberally in place of malloc where the extra space could be used.

As far as developer ergonomics, there are costs and benefits to each approach. The functions that return a pointer (and use an out-param for the size) can assign the return value directly to where it's likely to be consumed, to a non-`void` pointer. However, they also generally have to declare a variable to pass the address for the actual size; often the allocated size will be stored inside the just-allocated block.

[1] https://godbolt.org/z/cG8eMs366

Permit additional members?

There was some discussion on the mailing list about whether the “The structure shall contain (in either order) the members” as `div_t` is, or whether “structure shall contain at least the following members, in any order” as `tm` says is more appropriate for `alloc_result_t`. For pure consumers of that struct, there’s not really a material difference. But there’s an important use case of interposing allocators, where some function wraps an underlying allocator.

Pro

- Passing back the size is a useful thing while we have it at allocation time; maybe there’s other useful information that some implementation wants to pass back, such as whether the memory is pre-zeroed.
- Timespec and other structs specify “at least” and don’t seem to have suffered for it - although they also aren’t returned by value

Con

- An interposing allocator implemented to the standard won’t know what to do with those extra members. Zero initialization? Copy from the backing allocator? What if the fields they modify are inconsistent with whatever extra information is passed back?
- Malloc doesn’t contain caveats about possibly returning additional info, we shouldn’t do it here either.
- If people want to add allocator extensions, they should create new functions rather than smuggling things through a struct in the standard.
- As a matter of practice, reserving this freedom to add members doesn't make an addition in a future version of the standard easier, as adding new stuff would be an ABI break.

Interaction with Sized Free

For allocations made with `alloc_at_least` and `aligned_alloc_at_least`, we need to relax `free_sized`’s and `free_aligned_sized`’s size argument (7.25.4.5). For allocations of `T`, the size quanta used by the allocator may not be a multiple of `sizeof(T)`, leading to both the original and returned sizes being unrecoverable at the time of deallocation.

Consider the memory allocated by:

```C/C++
typedef struct {
  uint64_t data[2];
} T;
```

```
alloc_result_t r = alloc_at_least(sizeof(T) * 4);
T* p = r.ptr;
size_t s = r.size / sizeof(T);
```

or

```
C/C++
typedef struct {
  size_t count;
  uint64_t data[];
} T;

alloc_result_t r = alloc_at_least(offsetof(T, data[4]));
```

- The memory allocator may return a 72 byte object. Since there is no `k` such that `sizeof(T) * k == 72`, we can't provide that value to `free_sized`. The only option would be storing 72 explicitly, which would be wasteful.
- The memory allocator may instead return an 80 byte object (5 `T`'s): We now cannot represent the original request when deallocating without additional storage.

For allocations made with

```
C/C++
alloc_result_t r = alloc_at_least(n);
```

we permit `free_sized(r.ptr, s)` where `n <= s && s <= r.size`.

Where `s` must fall between the requested size `n` and the actual allocated size. This behavior is consistent with jemalloc's `sdallocx` and tcmalloc's size returning extensions. `free_sized` (N2801) recommends implementations accept sizes up to the actual size provided by extensions (i.e., `r.size` in this proposal) already.

Interaction with realloc

If a pointer obtained from `alloc_at_least` is passed to `realloc`, and `realloc` allocates a new block, the new block contains the contents of the previous block up to the returned size from `alloc_at_least`.

Why not realloc alone?

`realloc` must determine from the allocator's metadata the true size of the block. Even if paired with extensions like `malloc_usable_size` to resize to the precise, actual size, these pointer-to-size lookups are costly for allocators that do not store metadata inline. Avoiding this lookup was a motivation behind C++14's sized delete and C23's `free_sized` features. When a program can make use of the added space, the best time to determine it is at allocation time when the allocator has all of the relevant metadata available.

While an allocator may obtain memory via facilities like `mmap` under the hood and leave an opportunity to grow an allocation arbitrarily, this is not always possible in practice.

- Some allocators (like TCMalloc) cache deallocated objects rather than having a 1-to-1 correspondence between them and VMAs.
- VMA limitations (for example Linux's `/proc/sys/vm/max_map_count`) may preclude having too many independent regions for allocations, each of which is arbitrarily growable. This motivates coalescing allocations onto fewer address regions, so another allocation may be directly "after" the allocation we wish to grow.

Where is `realloc_at_least`?

Allocators have two options for implementing `realloc` when programs attempt to grow an allocation: extending in-place or performing a new allocation, copying the memory from the old allocation to the new allocation, deallocating the old allocation, and returning a pointer to the new allocation. When the allocator hands back OS memory pages directly for an allocation, the allocator may be able to use platform-specific optimizations to expand or shrink an allocation such as mremap(2) on Linux. This is employed by glibc on Linux.

With `alloc_at_least`, the available memory was already reported to the program so it is unlikely the allocator would be able to extend the allocation in-place, leaving the allocator to copy which the program could do itself with `alloc_at_least`+`memcpy`+`free_sized`. Thus we have chosen to not introduce `realloc_at_least` as it provides no clear benefit.

Alignment of `alloc_at_least`

The standard requires that allocated memory be "suitably aligned so that it may be assigned to a pointer to any type of object with a fundamental alignment requirement and size less than or equal to the size requested." To preserve the behavior that the result is usable anywhere the result of the same-sized malloc call is, we have to make the guaranteed alignment match the returned size, not the requested one.

Implementation Experience

TCMalloc has had a C++-style implementation of this since 2018. Since mid-2025, it now has an implementation of `alloc_at_least`.

Interaction with Bounds Checks

This proposal does not impede “hardened” allocators or sanitizers that wish to perform bounds checks, and can replace the use of `malloc_usable_size` and `_msize` which make bounds checking more difficult.

Avoiding Ossification (Hyrum’s Law)

Exposing the underlying size of allocations may result in code depending on the sizes returned in practice. GWP-ASan returns the requested size rather than the actual size in tcmalloc, to detect this possibly buggy code on a sampled basis, and other allocators wishing to be defensive against dependence on implementation details can apply the same strategy.