

Dynamic Allocation FAQ

Wednesday, July 19, 2023 10:28 PM



Is memory in heap which it is dynamically allocated automatically released when program execution done

Ans: I'm not definitely sure, but they say that it's not released automatically :(



If isn't released so if i run the program many times whiteout restart or shutdown pc and exceeded the memory size, is the os or pc will crash and i can't run any more programs

Ans: it depends on the previous answer if not auto-released after the program is finished the answer will be "Yes" otherwise, it will be "No"



How the `free()` function know the end of the allocated memory block in heap

Ans:



The `free()` function in C, which is used to deallocate memory previously allocated by `malloc()`, `calloc()`, or `realloc()`, does not inherently "know" the size of the allocated memory block. Instead, it relies on the memory management system to track metadata associated with each allocated block.

When you allocate memory dynamically using functions like `malloc()` or `calloc()`, the memory management system typically stores metadata alongside the allocated memory block. This metadata includes information such as the size of the allocated block, whether it's in use or freed, and any other necessary bookkeeping information.

When you call `free(ptr)` to deallocate a memory block pointed to by `ptr`, the memory management system uses the metadata associated with that block to perform the deallocation. It doesn't need to know the size of the block explicitly because it already has that information stored internally.

Here's a simplified overview of what happens when you call `free(ptr)`:

1. The memory management system retrieves the metadata associated with the memory block pointed to by `ptr`.
2. It marks the block as free in its internal data structures.
3. The memory management system may perform additional operations, such as merging adjacent free blocks or updating data structures for future allocations.
4. The memory block becomes available for subsequent allocations.

It's worth noting that this process is abstracted away from the programmer, who only needs to call `free()` with the pointer returned by `malloc()`, `calloc()`, or `realloc()`. The underlying memory management system handles the details of managing the allocated memory blocks and ensuring that memory is deallocated properly.

✓ What is the ``alloca()``, and ``cfree()``?

Ans:

1. ``alloca()``:

- allocates ``size`` bytes of space in the stack frame of the caller. This temporary space is automatically freed when the function that called ``alloca()`` returns to its caller. Do not return a pointer to this memory from the function, as it will become invalid.
- The ``alloca()`` function returns a pointer to the beginning of the allocated space. If the allocation causes stack overflow, program behavior is undefined.
- It's important to note that ``alloca()`` is not part of the C standard, and its use is generally discouraged in favor of standard dynamic memory allocation functions or variable-length arrays (VLAs) in C99. Always prefer portable and standard-compliant practices for memory management in C.

❖ **Look file:** [2 stack dynamic allocation.c](#)

2. ``cfree()``: is a non-standard function that was historically used to free memory allocated by ``calloc()``. In modern C programming, ``cfree()`` is obsolete, and you should use ``free()`` to deallocate memory allocated by any of the standard allocation functions (``malloc()``, ``calloc()``, or ``realloc()``).

✓ How was the dynamic allocation done in the old version of OS (Unix, etc)?

Ans:

1. **Old Systems:** Dynamic memory allocation in early Unix and similar operating systems was often done using system calls like ``brk()`` and ``sbrk()``.
2. **Modern Systems:** Modern operating systems and C libraries have more sophisticated memory allocators that use a combination of ``mmap()`` (for larger allocations) and ``sbrk()`` (for smaller ones).

✓ Can i Access the memory in heap after ``free()`` it, and what is the behavior if i do this?

Ans: Yes, and the behavior of this is undefined also the value in this location after freed is undefined

❖ **Look file:** [1 access freed location.c](#)

✓ Can I increment the pointer before ``free()`` it, double ``free()`` or use ``free()`` with pointer to a stack location?

Ans: No, and this will result in an error

✓ Can I dereference a ``void`` pointer ``int a; void *p = &a;`` or ``void *p = malloc(4 * sizeof(int));``


Ans: No, this will result in an error

❖ **Look file:** [3 dereference void pointer.c](#)



how do `malloc()`, `calloc()`, `realloc()`, or `free()` works internally, are they use other functions for help and what are the interfaces that they use?

Ans:

 **Copilot**

The functions `malloc()`, `calloc()`, `realloc()`, and `free()` are part of the C standard library and are used for dynamic memory allocation. Internally, these functions work by interacting with the operating system to manage the heap portion of a process's memory. Here's a high-level overview of how each function operates:

- `malloc()`: Allocates a specified number of bytes and returns a pointer to the allocated memory. The memory is not initialized.
- `calloc()`: Allocates memory for an array of elements, initializes them to zero, and then returns a pointer to the memory.
- `realloc()`: Changes the size of the memory block pointed to by a given pointer. It can enlarge or reduce the size of the block and may move it to a new location if necessary.
- `free()`: Deallocates the memory previously allocated by `malloc()`, `calloc()`, or `realloc()`.

Internally, these functions may use system calls like `brk()` or `sbrk()` on Unix-like systems or functions like `HeapAlloc()` on Windows to allocate or deallocate memory. They maintain a data structure, often a form of linked list, to keep track of allocated and free blocks of memory.

When you call `malloc()` or `calloc()`, the allocation functions look for a suitable block in the free list. If a suitable block is found, it is marked as allocated and a pointer to the block is returned. If no suitable block is found, the allocator may request more memory from the operating system.

When you call `free()`, the deallocation function adds the block back to the free list so it can be reused in future allocations. If adjacent blocks are free, they may be coalesced into a larger block to reduce fragmentation.

`realloc()` is a bit more complex because it must handle the case where the block is being resized. If the block is being made smaller, it may simply adjust the size and leave it in place. If the block is being enlarged, it may need to move it to a new location if there isn't enough space to expand in place.

These functions are defined in the `<stdlib.h>` header file and are part of the C standard library, which provides a standard interface for programmers to use these functionalities ¹ ² ³.

It's important to note that the exact implementation details can vary between different C standard library implementations and operating systems, but the interface and expected behavior remain consistent across environments.



Is `malloc()` and `calloc()` reserve a contiguous block of needed space in memory

Ans: Yes



How garbage collection knows the memory leaks or unused memory spaces in heap

Ans: search on google about it (if you don't know where to start [click here](#))



try `int *a = realloc(a, 0)`, is `a = realloc(a, 0) == free(a)`, `a = realloc(NULL, 5 * sizeof(*a)) == a = malloc(5 * sizeof(*a))`?

Ans:

❖ **Look file:** [4_realloc_equal_malloc-free.c](#)



What is the all scenarios of use `realloc()` such as if I use `int *a = realloc(NULL, 5 * sizeof(*a));` or `int *a = malloc(5 * sizeof(*a));` then `a = realloc(a, 0);`, `int *a = realloc(a, 3 * sizeof(*a));` or `int *a = realloc(a, 10 * sizeof(*a));`?

Ans:

❖ **Look file:** [5_realloc_scenarios.c](#)

- ✓ try `int *a = malloc(n * sizeof(int));` and then `a = realloc(a, (n/2) * sizeof(int));` is this will free the 2nd half of `a`?

Ans: Yes

- ✓ What is the Datatype overflow, and what happens?

Ans: it's the exceeding of the datatype range where each datatype (such as `int`, `char`, `float`, etc.) has a defined range of values it can represent

EX:- when you want to store a value `256` in 1 Byte(8 bit) which the binary of value `256` is `00000000 00000000 00000001 00000000` where the maximum value that the 1 Byte(8 bit) can represent is `255` which the binary of value `255` is `00000000 00000000 00000000 11111111`

So when you try this `char c = 256;` the variable `c` will reset to value `0` because the `char` datatype represents only 1 Byte(8 bit)

❖ **Look file:** [0_data_overflow.c](#)

- ✓ What is the difference between `return()` and `exit()`

Ans:



General Explanation

The `return` and `exit` functions in C both cause a function to terminate, but they do so in different contexts and have different scopes of effect. Here's a detailed explanation of the differences between them:

Explanation of `return` and `exit`

`return`:

- **Context:** Used to exit from a function.
- **Scope:** Terminates the execution of the current function and returns control to the calling function.
- **Usage in `main`:** When used in the `main` function, `return` also ends the program and returns an exit status to the operating system.
- **Value:** Can return a value to the caller, which can be used for further processing.
- **Stack Unwinding:** Causes local variables to be destroyed and the stack frame for the current function to be popped.

`exit`:

- **Context:** Used to terminate the entire program.
- **Scope:** Ends the program immediately, regardless of which function calls it.
- **Usage:** Can be called from anywhere in the code.
- **Value:** Takes an integer argument which is returned to the operating system as the program's exit status.
- **Stack Unwinding:** Does not return to the calling function, so local variables and stack frames of all functions in the call stack are not cleaned up as they would be with `return`.
- **Additional Behavior:** Executes any registered cleanup functions (e.g., those registered with `atexit`), closes open files, flushes output buffers, etc.

Summary

- ``return``:
 - Used to return from a function.
 - In ``main``, it exits the program and returns a status code to the OS.
 - Causes stack unwinding and local variable destruction.
 - Can return values to the calling function.
- ``exit``:
 - Used to terminate the program from anywhere.
 - Returns an exit status to the OS.
 - Does not cause stack unwinding for the functions in the call stack.
 - Executes registered cleanup functions, closes files, and flushes buffers.

Use ``return`` to exit a function and optionally pass a value back to the caller, and use ``exit`` to terminate the program immediately and perform cleanup tasks.

❖ **look file:** [6_exit_not_like_return.c](#)