Piscine — Tutorial D2 PM

version #

Assistants C/Unix 2022 <assistants@tickets.assistants.epita.fr>

Copyright

Contents

1 Memory by allocation **3**
- 1.1 Introduction . . . 3
- 1.2 Types of memory . . . 3
- 1.3 Dynamic memory . . . 3
- 1.4 Memory allocation . . . 4
- 1.5 Memory deallocation . . . 5

2 Dynamic array **6**
- 2.1 Declaration . . . 6
- 2.2 Access . . . 7
- 2.3 Multidimensional array . . . 7
- 2.4 Strings . . . 8
- 2.5 Exercises . . . 8

3 Advanced memory functions **10**
- 3.1 Reminder . . . 10
- 3.2 `calloc(3)` . . . 10
- 3.3 `realloc(3)` . . . 11

*https://intra.assistants.epita.fr

1 Memory by allocation

1.1 Introduction

Up until now, you've been using a fixed amount of memory when writing `C` programs. You've been using variables that only existed during a function call, or a specific amount of memory that was used during the whole lifetime of your program.

There will be a lot of cases where you cannot plan ahead for the amount of memory your program will need. In order to solve this problem, you can manage some amounts of memory during the execution of your program using the `malloc(3)` and `free(3)` functions.

1.2 Types of memory

When a `C` program is executed, it can store data in different memory areas. These memory areas belong to different **types of storage**, which are defined by the language standard and have an associated **lifetime**.

There are three different types of storage:

- Static memory: where global variables are stored, or variables defined with the `static` keyword. The static memory persists for the lifetime of the program.
- Automatic memory: allocated on the **stack**, and its lifetime depends on specific scopes (function body, loops, ...).
- Dynamic memory: allocated on the **heap**, whose lifetime is that of the program itself unless explicitly released.

Be careful!

Do not confuse the *static* memory with a *static* array. In the case of the array, it is considered to be static when its size is known at compile time, not necessarily that it is stored in the static memory.

1.3 Dynamic memory

In `C`, management of the dynamically allocated memory space is **manual**: allocations and deallocations are **explicitly** managed by you, the developer. Memory allocated by a call to `malloc(3)` is **not** automatically deallocated at the end of the function.

Your allocator is a set of functions which manage all the memory chunks you allocate and free to fit them into a large memory region, owned by the allocator, and used to store all these blocks. You just need to request a memory region of a desired size to your allocator, and it will return a pointer to an area of such size if it is available.

1.4 Memory allocation

The following block of code is an example of using C's standard memory allocator: `malloc(3)` defined in `stdlib.h`.

```
#include <stdlib.h>

int *toto(void)
{
    return malloc(sizeof(int));
}
```

As you can see, `malloc(3)` returns a pointer. Here we have a function `toto` returning a pointer to an area large enough to hold an `int` value. The `sizeof` keyword is used to determine the memory size required for the `int` type. We give this size to `malloc(3)` and it will return a chunk of dedicated space in which you can store your `int` element.

According to the prototype of the `malloc(3)` function, the type returned is `void *`, this represents a generic pointer to unknown data. It's up to you to assign this pointer to a pointer of a specific type; which allows your compiler to ensure that you are correctly using your pointer. For example, if you specify a `char` pointer where an int pointer is expected, an error can be detected. This error would not have occurred had you been using a generic pointer. That's why the function `toto` returns a pointer to an integer (`int *`).

If the memory is full, the `malloc(3)` function will return `NULL`. This behavior is described in `malloc(3)`'s man page. `NULL` is a special value, it usually points to the first address in your program's memory. Why is that so? In the range of memory available for your program, some parts are not accessible. This is the case for the beginning of the virtual address space of your program. The first address is not accessible, and if you try to access it, a segmentation fault will occur. This behaviour will allow you to use the zero address represented by `NULL` as an invalid address for invalid pointers. Thus, if you try to access it, your program will crash. This address is guaranteed to be invalid (except if a kernel programmer makes a joke, but that's not funny), so it can be used by `malloc(3)` to notify you that the allocation cannot be done.

Be careful!

Always[1] check `malloc(3)`'s return value! If the allocation fails and returns `NULL`, your program will crash when it will use the incorrectly allocated pointer (This may happen later in your code, making debugging harder ...). Of course, your ACUs may give your program a `malloc(3)` that will return `NULL` during tests...

[1] **Always**

We **strongly** advise you to always use `NULL` (defined in the header `stddef.h`, but included in `stdlib.h`) to initialize your pointers. Ideally, a pointer must contain either a valid address or `NULL`. Never leave an uninitialized pointer, because the address it holds can be anything, and that address may be `NULL` but may also be another invalid one, or worse, a valid address somewhere else in memory.

1.5 Memory deallocation

As said previously, memory areas allocated by `malloc(3)` are not destroyed (freed or unallocated) automatically. We need a function to deallocate the memory's areas at the addresses returned by `malloc(3)`. This function is named `free(3)` and takes a pointer to the memory that must be released as a parameter.

Whenever you don't need the memory allocated by `malloc(3)` anymore, you should free it using `free(3)`.

Forgetting to do so can cause what are called *memory leaks*. Those are some of the worst mistakes that can occur in a program. If a program with *memory leaks* runs for a long period of time (a server for example), it will completely fill the RAM and will slow the system down, or can even cause it to shutdown. *Memory leaks* are also some of the hardest bugs to find. You should **always**[2] keep in mind where you will free allocated memory.

Once you call `free(3)`, your pointer still holds the address it did before the call, which is not valid anymore. Dereferencing this address leads to an undefined behavior. If your pointer variable still exists after your `free(3)` (not right before function's end), you should assign it to `NULL` to avoid confusion.

For example:

```
int *i_ptr = NULL;

/* The memory space that i_ptr points to is allocated */
i_ptr = malloc(sizeof(int));

if (!i_ptr) /* malloc returned NULL, this pointer is not valid */
    return /* whatever */;

*i_ptr = 42; /* let's fill this memory with a value */

int b = *i_ptr; /* b's value is 42 */

free(i_ptr); /* memory chunk is given back */

i_ptr = NULL; /* mark this pointer as NULL to avoid keeping an invalid address */

/* Do some stuff and return */
```

Be careful not to mistake a pointer and the memory's area to which it points! In the previous example, the pointer (i.e. the variable `i_ptr`) is allocated automatically on the stack, and the area pointed by `i_ptr` is allocated manually with `malloc(3)`.

The man page of function `free(3)` specifies that it takes as parameter any pointer returned by `malloc(3)`, thus giving `NULL` pointer to `free(3)` is valid (but won't do anything).

Be careful!

For every `malloc(3)` call you make, you **must** have a corresponding `free(3)` call.

[2] **Always.**

2 Dynamic array

Sometimes, you do not know the size of an array you wish to create in advance (for example if the size of your array depends on the user input). For the moment you have only seen *static* arrays, whose sizes are known at compile time and directly specified in the source code. This section introduces *dynamic* arrays, whose size is determined at runtime.

2.1 Declaration

As we saw previously, the data in an array is stored contiguously in memory:

```
int arr[4] = { 12, 27, 42, 51 };
```

0x50	12	27
0x58	42	51

Fig. 1: The array is contiguous in memory, starting at 0x50

When allocating memory using `malloc(3)`, the returned address points to a contiguous block of memory. Therefore, you can use the allocated space to store all the data from an array.

For example, to store 4 integers:

```
int *arr = malloc(sizeof(int) * 4);
if (!arr)
    // error handling

// Unlike static arrays, we cannot use a brace-enclosed list of elements.
arr[0] = 12;
arr[1] = 27;
arr[2] = 42;
arr[3] = 51;

free(arr); // Do not forget to free the memory!
```

This can seem tedious, but we can now use variables to determine our array's size:

```
size_t nb_elements = /* Let us suppose it is initialized with user input */;

int *arr = malloc(sizeof(int) * nb_elements):
if (!arr)
    // error handling

// Fill, use, print, ... the dynamic array

free(arr);
```

2.2 Access

Static arrays and dynamic arrays use the same syntax to access elements:

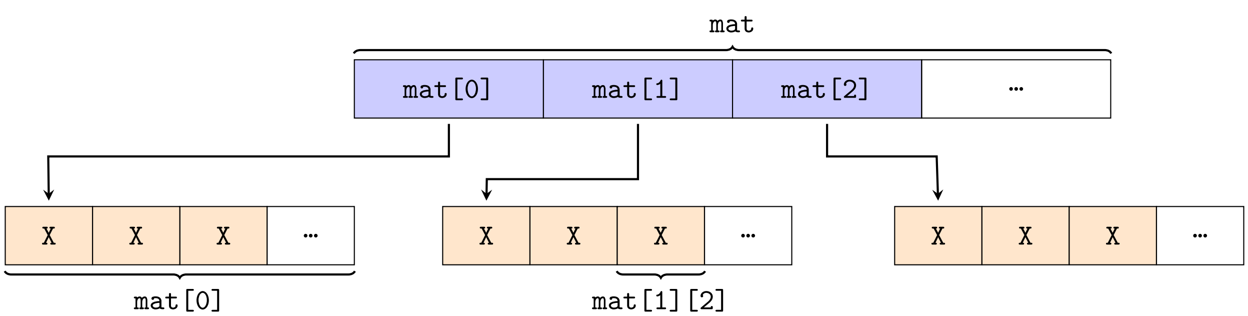
```
arr[1] = 12;          // Access to the second element
*(arr + 1) = 12;      // Same as above, remember pointer arithmetic?
```

2.3 Multidimensional array

A multidimensional array is an array of arrays. To allocate one you must first allocate the main array, and then all of its sub-arrays.

For example with a N*M matrix:

```
int **mat = malloc(sizeof(int *) * N); // Allocate the main array

for (size_t i = 0; i < N; i++)
    mat[i] = malloc(sizeof(int) * M); // Allocate all sub-arrays
```



Do not forget that you **MUST**[1] free **all** the allocated dynamic memory. The order is particularly important here since freeing the main array first means you cannot free the sub-arrays because you cannot access them anymore. It would result in memory leaks. The correct way to free a 2D dynamic array is the following:

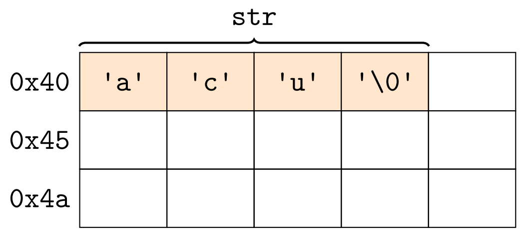
```
for (size_t i = 0; i < N; i++)
    free(mat[i]); // Free all sub-arrays first

free(mat); // Free main array
```

[1] **Must**.

2.4 Strings

As you now know, *C* strings are arrays of `char` ending with a '\0'. You **MUST** allocate one more character than your string size in order to set the last element to '\0'. For example, here is a dynamically allocated string of size 3:

```
char *str = malloc(sizeof(char) * (3 + 1));
str[0] = 'a';
str[1] = 'c';
str[2] = 'u';
str[3] = '\0';
```



Be careful!

Do not be confused between dynamically allocated string and statically allocated **read-only** string:

```
char *str = "acu";                          // Statically allocated and read-only
char *str = malloc(sizeof(char) * (3 + 1)); // Dynamically allocated
```

The statically created string automatically adds a '\0' to the end of the string, whereas the dynamically created string **does not**.

2.5 Exercises

2.5.1 addresses

Create variables on the stack, and display their address using `%p` of `printf(3)`.

Do the same using addresses returned by `malloc(3)`.

Explain the difference between two values.

2.5.2 create_array

You want to create arrays of `int`, but with a size only known at runtime.

```
int *create_array(size_t size);
```

Return a pointer to a memory region containing `size` integers (`int`). Return `NULL` if you cannot allocate the memory.

Tips

size_t is available in stddef.h, which is included by stdlib.h.

2.5.3 free_array

You do not need the previously allocated array anymore.

```
void free_array(int *arr);
```

Free the memory used by the given array. Do not do anything if arr is NULL.

2.5.4 read_and_inc

```
void read_and_inc(int *v);
```

Using printf, display the integer pointed by v and increment it by one.

2.5.5 my_strdup

```
char *my_strdup(const char *str);
```

Allocate a char array large enough to hold the str string (null terminated) and copy the value of str into this memory area. Return the pointer to this memory. We must be able to pass the returned pointer to free(3). Return NULL if the allocation failed.

Use printf(3) to display the string returned by my_strdup to check consistency.

2.5.6 my_strndup

```
char *my_strndup(const char *str, size_t n);
```

Same as my_strdup but copies at most n bytes. If str is shorter than n, acts as my_strdup. n does not include the terminating null byte. The pointer returned should be freed using free(3). Returns NULL if the allocation failed.

Use printf(3) to display the string returned by my_strndup to check consistency.

3 Advanced memory functions

3.1 Reminder

You have already seen the basics of dynamic memory allocations previously. We will now explain the different types of memory, and present two other functions of the *C* standard library relating to memory allocation: `calloc(3)` and `realloc(3)`.

3.2 `calloc(3)`

When allocating memory for an array, you may need a memory chunk initialized to zero, thus setting all elements in your array to zeros. This is what `calloc(3)` does; like `malloc(3)`, it allocates a memory area, but it is filled with zeros. Like `malloc(3)`, the pointer returned will be `NULL` if the memory cannot be allocated, see `man 3 calloc` for more details.

```
// create an array of ten integers. All elements are set to zero.
int *array = calloc(10, sizeof(int));

if (!array)
    /* handle the error */

for (int i = 0; i < 10; ++i)
    printf("%d ", array[i]);

free(array);
```

The prototype of `calloc(3)` is a bit different from `malloc(3)`. Where `malloc(3)` takes only the size of the whole block as parameter; `calloc(3)`, takes the number of members of an array and the size of each member.

3.2.1 Application: my calloc

```
void *my_calloc(size_t n, size_t size);
```

Returns a pointer to an allocated memory area capable of holding `n` elements of `size` bytes. The whole memory must be set to zero. Returns `NULL` if the allocation failed. The returned pointer will be freed later using `free(3)`.

Going further...

Keep in mind that `calloc(3)` is way more than a simple function that calls `malloc(3)` and fills the area with *0s*[1].

[1] https://vorpus.org/blog/why-does-calloc-exist/

3.3 realloc(3)

Once you allocated a memory area, you may need to resize it later. For example if your memory is used to hold elements of a dynamic vector, you want to expand this memory when your vector is full, or reduce it when there is only a few elements in it. `realloc(3)` will give you the ability to resize a given memory zone. It takes two parameters: the pointer to an area you already allocated, and the new size. It returns a pointer to the new area.

Be careful!

If the new size is bigger than the previous one, `realloc(3)` may not be able to expand the current region, thus it will allocate another bigger chunk of memory, copy the previous elements in this new memory area, then free the old memory area. This is why `realloc(3)` returns a pointer to a new area, if it was not able to expand the current one.

Tips

`realloc(3)` may return `NULL` if the allocation of the new area failed. According to the man page, the previous area is left untouched in case of failure. Therefore, if you reassigned your pointer to the value returned by `realloc(3)`, you lost the previous pointer and won't be able to free it. The best way to avoid this leak is to use a temporary pointer and to check it before reassigning it to your initial pointer.

```
size_t size = 10;
int *array = calloc(size, sizeof(int)); // create an array of ten integer

if (!array)
    /* handle the error and leave */

/* do some stuff with your array */

size_t new_size = (size * sizeof(int)) * 2;
int *new = realloc(array, new_size);

if (!new)
    /* reallocation failed */
else
    array = new; /* reallocation succeeded, array has been freed */
/* if the allocation succeeded, old values are still there */

free(array);
```

3.3.1 my_free

Using `man 3 realloc`, implement an equivalent to `free(3)` function without calling `free(3)` directly.

3.3.2 my_malloc

Using `man 3 realloc`, implement an equivalent to `malloc(3)` function without calling `malloc(3)` directly.

3.3.3 realloc changes

Allocate a memory chunk of the size of your choice, then display the address using `printf(3)` with `%p`. Inside a loop, expand the memory area by one using `realloc(3)` and display the new address.

What is happening?

It is my job to make sure you do yours.