Prog C — td6

version **#0.0.1-dirty**

Assistants C/Unix 2024 <assistants@tickets.forge.epita.fr>

Copyright

Contents

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

1 Dynamic Memory allocation

1.1 Use of memory allocation

You have already manipulated memory by declaring variables. In order for your program to run and access values of your variables, the compiler has to know how much memory to allocate for each of them.

```c
int main(void)
{
    int a = 42;             /* 4 bytes or 32 bits */
    int b[3] = {1, 2, 3};   /* 3 * 4 bytes or 3 * 32 bits */

    return 0;
}
```

The memory is `automatically` **allocated** by the compiler during the compilation phase. The compiler knows how much memory to allocate for each variable and array because you told it so by declaring them. When the program exits, the memory is `automatically` freed or **deallocated**.

Sometimes you want to use some memory without knowing ahead of time how much you might need. In order to solve this problem, you need to manage the allocation of your memory by hand. The memory is **manually allocated** by you, during the execution of your program using the `malloc(3)` function. Therefore you need to **manually deallocate** this memory when you don't need it anymore. To do so you will use the `free(3)` function. Allocating memory manually during run-time is known as `dynamic memory allocation`.

For instance, say you want the user to enter an unknown number of integers. You can create a huge array to store them all, but if the user enters only one integer, a lot of memory is wasted. The right solution here is to use dynamic memory allocation to adapt the memory you use according to the space you need to store the user's input.

1.2 Dynamic memory

In `Python`, the management of the allocated memory space is **automatic**: dynamic allocations and deallocations are **implicitly** managed by the interpreter. Memory allocated by a call to `list()` is **automatically** deallocated when the list is not used anymore.

In `C`, the management of the allocated memory space is **manual**: dynamic 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 or at the end of the process. You have to call `free(3)` to deallocate it.

Be careful!

Every memory allocated by calling `malloc(3)` has to be freed using `free(3)`.

1.3 Memory allocation

The `malloc(3)` function allocates a chunk of memory of the specified size (in bytes) and returns a pointer to the beginning of this chunk. The `free(3)` function frees the memory previously allocated by a call to `malloc(3)`.

The following block of code is an example of using `malloc(3)` and `free(3)` declared in `stdlib.h`.

```c
/**
** \file my_tiny_int_array.c
*/

#include <stdlib.h>
#include <stdio.h>

int *create_my_int_array(size_t size)
{
    int *array = malloc(size * 4); /* number of elements times size of one element */

    if (NULL == array) /* It is mandatory to check the return value of malloc */
    {
        puts("Error(create_my_int_array): malloc returned NULL\n");
        return NULL;
    }

    for (size_t i = 0; i < size; i++)
    {
        array[i] = i;
    }

    return array;
}

int main(void)
{
    size_t size = 10;
    int *ptr = create_my_int_array(size);

    if (NULL == ptr)
    {
        puts("Error(main): malloc returned NULL\n");
        return 1;
    }

    for (size_t i = 0; i < size; i++)
    {
        printf("%d\n", ptr[i]);
    }

    free(ptr);
    return 0;
}
```

```
$ gcc -Wall -Wextra -Werror -pedantic -std=c99 my_tiny_int_array.c -o my_tiny_int_array
$ ./my_tiny_int_array
```

(continues on next page)

(continued from previous page)

```
0
1
2
3
4
5
6
7
8
9
```

Now let's see a more complex example:

```c
#include <stdlib.h>
#include <stdio.h>

char *strupcase_dup(char *s, size_t size)
{
    char *new_s = malloc((size + 1) * sizeof(char));

    if (NULL == new_s)
    {
        puts("Error: malloc returned NULL\n");
        return NULL;
    }

    for (size_t i = 0; s[i] != '\0'; i++)
    {
        if (s[i] >= 'a' && s[i] <= 'z')
        {
            new_s[i] = s[i] - 'a' + 'A';
        }
        else
        {
            new_s[i] = s[i];
        }

    }
    new_s[size] = 0;
    return new_s;
}
```

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

Tips

The `sizeof` operator returns the size that a type occupies in the computer's memory. The type is passed as you would pass an argument to a function. For instance, on the PIE, `sizeof(int)` will be equal to four, since an `int` occupies 4 bytes in memory. Structures are types, thus `sizeof(struct vector)` will return the size of the whole structure, which depends on the combined size of all of its fields.

We have seen two examples where `malloc(3)` is used to allocate memory for a `char` pointer and an `int` pointer. Let's have a look to the prototype of `malloc(3)`:

```
void *malloc(size_t size);
```

According to the prototype of the `malloc(3)` function, its return type is `void*`. This represents a generic pointer of an unknown data type. It is up to you to assign this pointer to a pointer of the specific type. Your compiler tries 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.

Be careful!

You need to take special care when manipulating generic pointers. For instance, pointer arithmetic is not permitted by the C standard as the size of a *void* pointer is not defined.

`malloc(3)` will return `NULL` when it can't allocate memory, do not forget to check the return value of `malloc(3)`.

Be careful!

Always[1] check `malloc(3)`'s return value: if the allocation fails and returns `NULL`, your program will crash when it'll use the pointer not correctly allocated (This may happen later in your code, making debugging needlessly harder).

[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 it can point to anything, and this address may not be a valid one, nor `NULL`.

1.4 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 as parameter the pointer that must be released.

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 (for example a server), it will completely fill the RAM and will slow the system, or can even cause it to shutdown. *Memory leaks* are also some of the hardest bugs to find. You should **always**[1] keep in mind where you will free allocated memory.

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

[1] **Always**

For example:

```c
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 */
{
    /* handle the error case */
}

*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 */
// b's value is still 42
// 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 variable (i.e. `i_ptr`) and the area pointed by `i_ptr` allocated manually with `malloc(3)` are not in the same place in memory.

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).

1.5 Exercises: memory

1.5.1 Create an array

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

```c
int *create_array(unsigned n);
```

Return a pointer to a memory region containing `n` integers. Write a message if you cannot allocate the memory.

1.5.2 Free an array

You do not need the previously allocated array anymore.

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

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

1.5.3 Custom array

Implement the function `array_create`:

```c
struct my_array *array_create(size_t nb_elements);
```

This function allocates a new `my_array` of size `nb_elements`. Here is the given header for this exercise:

```c
/**
** \file my_array.h
*/

#ifndef MY_ARRAY_H
#define MY_ARRAY_H

#include <stddef.h>

struct my_array
{
    int *data;
    size_t size;
};

struct my_array *array_create(size_t nb_elements);

#endif /* ! MY_ARRAY_H */
```

And you will have to fix the function `create`, following this example :

```c
/**
** \file my_array.c
*/

#include "my_array.h"

struct my_array *array_create(size_t nb_elements)
{
    // FIXME
}
```

Here is how it can be used:

```c
/**
** \file main.c
*/

#include <stdio.h>
#include <stdlib.h>

#include "my_array.h"

int main(void)
{
    struct my_array *my_array = array_create(42);

```

(continues on next page)

(continued from previous page)

```c
    void *data = my_array->data;

    if (NULL == my_array)
    {
        printf("No hero has been created!\n");
    }
    else if (NULL == data)
    {
        printf("No pointer data has been created!\n");
    }
    else
    {
        printf("my_array has a size of %zu and my_array->data is %p.\n",
                my_array->size, data);
    }

    free(my_array);
}
```

Tips

When executing `free(my_array)`, `my_array` can be `NULL`. This is not an issue since `free` does nothing when its argument is null. You can check `man 3 free`.

Be careful!

In order to display the address of a pointer, we need to convert it to a `void *`, hence the `data` variable. For more information, check the manual for `printf(3)`.

Note that you will rarely have to manipulate generic pointers like this and this is done juste for the sake of the example.

You have to allocate a `struct my_array`, set the field `size` using the function argument and allocate the field `data` using `malloc(3)`. If the structure cannot be allocated you have to return `NULL`.

Here is what you get when you compile and execute

```
42sh$ gcc -Wall -Wextra -Werror -pedantic -std=c99 my_array.c main.c -o my_array
42sh$ ./my_array
my_array has a size of 42 and my_array->data (should be) different of NULL.
```

I must not fear. Fear is the mind-killer.