Automatic and dynamic allocation, malloc and free

The `malloc` function is used to allocate a certain amount of memory during the execution of a program. It will request a block of memory from the heap. If the request is granted, the operating system will reserve the requested amount of memory and `malloc` will return a pointer to the reserved space.

When the amount of memory is not needed anymore, you must return it to the operating system by calling the function `free`.

Automatic allocation

When you declare variables or when you use strings within double quotes, the program is taking care of all the memory allocation. You do not have to think about it.

```
julien@ubuntu:~/c/malloc$ head -n 14 cisfun.c
/**
 * cisfun - function used for concept introduction
 * @n1: number of projects
 * @n2: number of tasks
 *
 * Return: nothing.
 */
void cisfun(unsigned int n1, unsigned int n2)
{
    int n;
    char c;
    int *ptr;
    char array[3];
}
julien@ubuntu:~/c/malloc$
```

In the above example, the arguments and the local variables are stored automatically in memory when the function is called. The program reserves space and uses it without you having to think about it.

Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable	n1				n2				n				c							
Value	known value				known value				?				?	?	?	?	?	?	?	?

Address	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
Variable	ptr								array											
Value	?								?	?	?	?	?	?	?	?	?	?	?	?

By default, the memory used to store those variables can be read and written. When the program leaves the function, the memory used for all the above variables is released for future use.

Special case: string literals

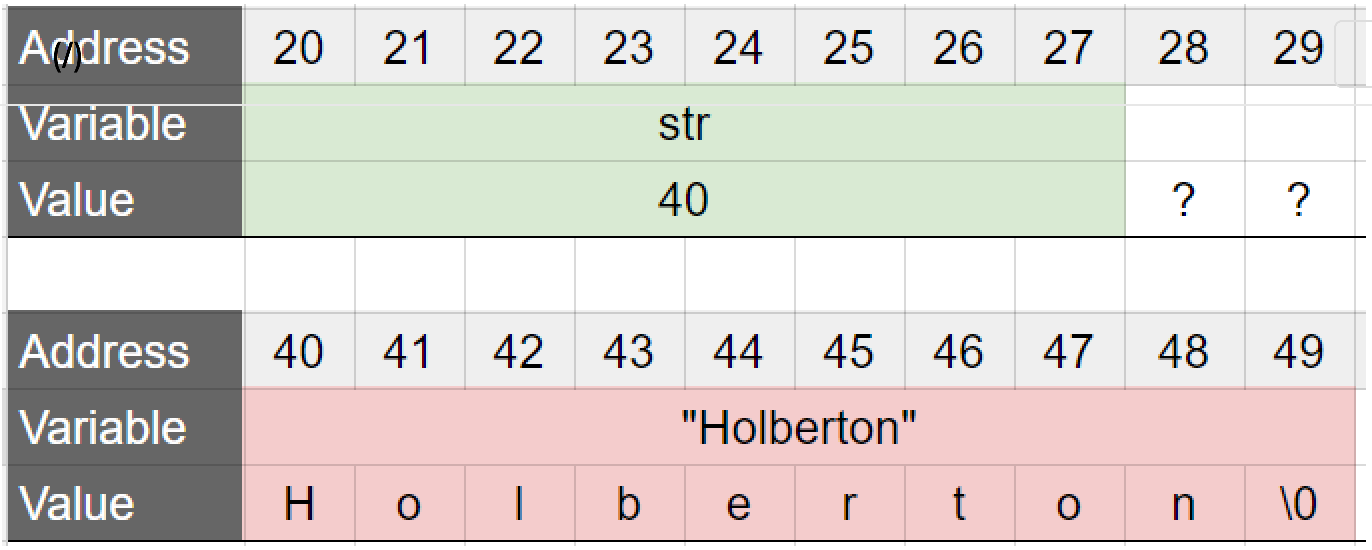
One special case we have seen so far is the memory used to store strings that we put within double quotes (`"`) in our programs. For instance:

```
char *str;

str = “Holberton”;
```

The string `"Holberton"` that was just declared is stored automatically in memory when the program is launched. But, the memory that stores the string is only readable. In fact, if you try to change a character using `str`, you will have a little surprise :)

```
julien@ubuntu:~/c/malloc$ cat segf.c
/**
 * segf - Let's segfault \o/
 *
 * Return: nothing.
 */
void segf(void)
{
    char *str;

    str = "Holberton";
    str[0] = 's';
}

/**
 *  main - concept introduction
 *
 * Return: 0.
 */
int main(void)
{
    segf();
    return (0);
}
julien@ubuntu:~/c/malloc$ gcc segf.c
julien@ubuntu:~/c/malloc$ ./a.out
Segmentation fault (core dumped)
julien@ubuntu:~/c/malloc$
```

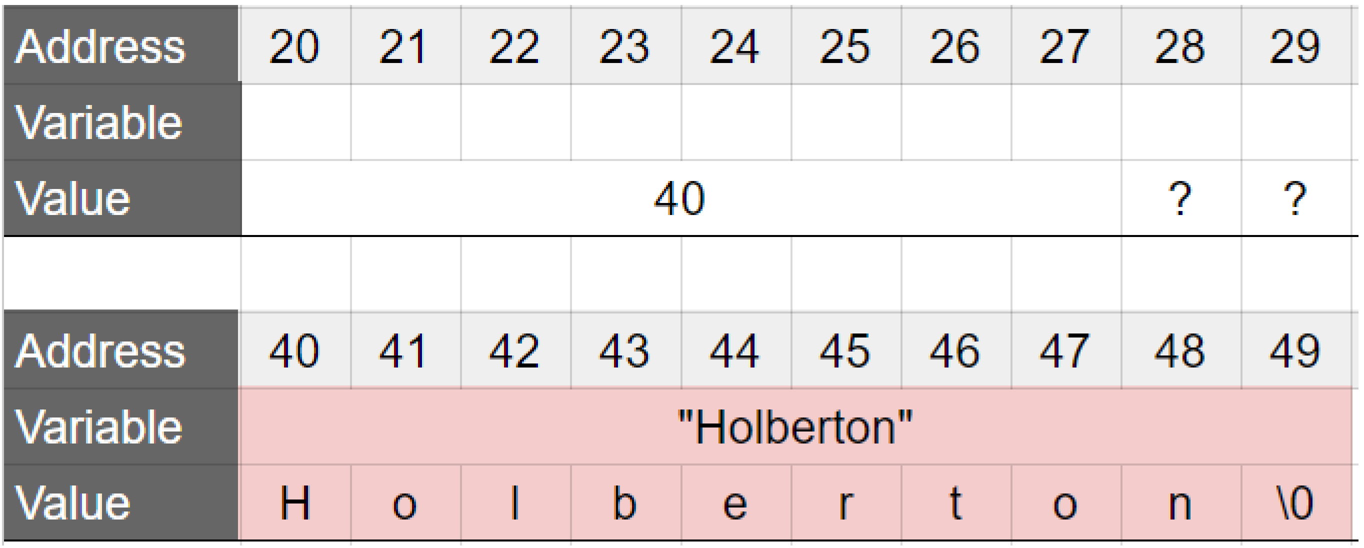
In the above example, the variable `str` is a pointer to a char, that is initialized to the address of the first character of the string “Holberton”. But the memory storing the string “Holberton” is read-only, and will also not be released when the function returns. This is the state of the memory after the line `str = "Holberton";` is executed (in red: read-only memory):



Address	20	21	22	23	24	25	26	27	28	29
Variable	str									
Value	40								?	?

Address	40	41	42	43	44	45	46	47	48	49
Variable	"Holberton"									
Value	H	o	l	b	e	r	t	o	n	\0

And this is the state of the memory after the function returns:



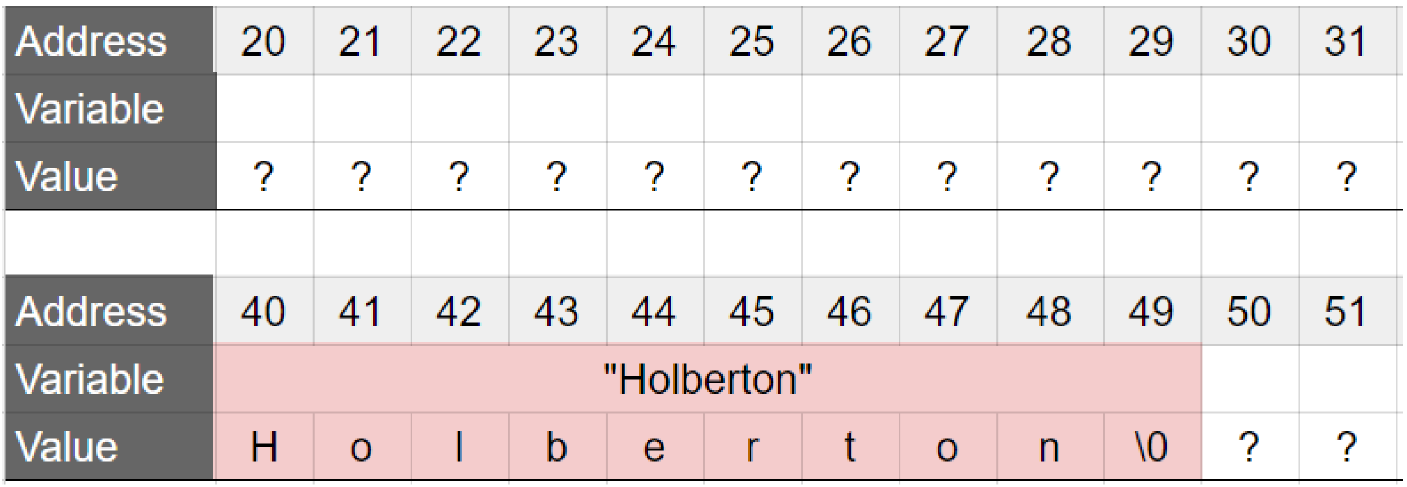
Address	20	21	22	23	24	25	26	27	28	29
Variable										
Value	40								?	?

Address	40	41	42	43	44	45	46	47	48	49
Variable	"Holberton"									
Value	H	o	l	b	e	r	t	o	n	\0

Note that when using the notation: `char s[] = "Holberton"`, the array `s` holds a **copy** of the string `"Holberton"`. So it is possible to modify this copy.

```
julien@ubuntu:~/c/malloc$ cat print_school.c
#include <stdio.h>

/**
 * print_school - prints "Holberton"
 *
 * Return: nothing.
 */
void print_school(void)
{
    char str[] = "Holberton";

    str[0] = 's';
    printf("%s\n", str);
}

/**
 *  main - concept introduction
 *
 * Return: 0.
 */
int main(void)
{
    print_school();
    return (0);
}
julien@ubuntu:~/c/malloc$ gcc print_school.c
julien@ubuntu:~/c/malloc$ ./a.out
Holberton
julien@ubuntu:~/c/malloc$
```

This is the memory before the call to `print_school` :

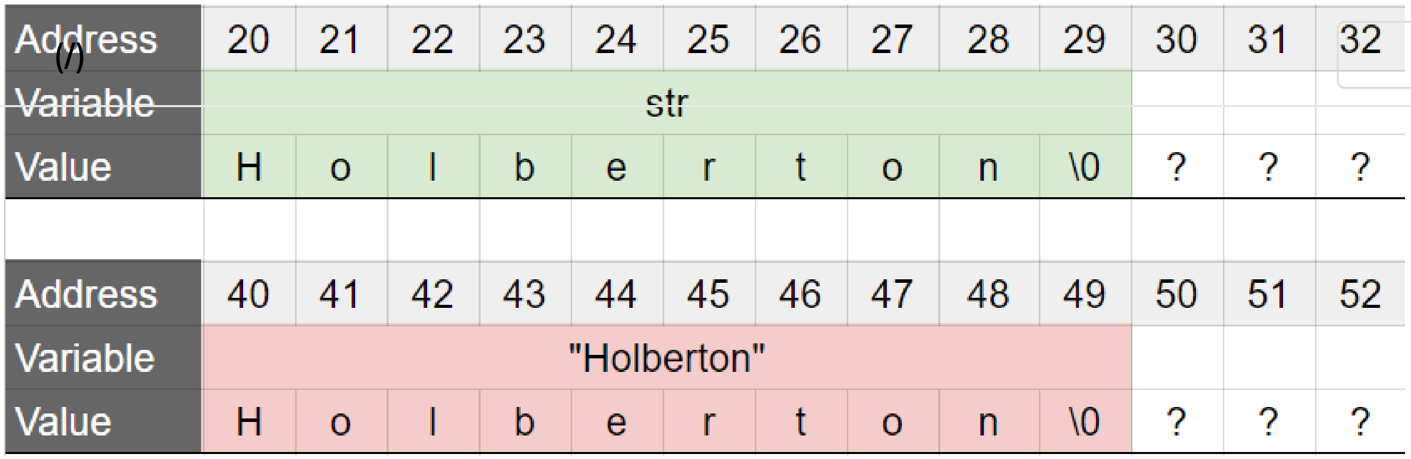


Address	20	21	22	23	24	25	26	27	28	29	30	31
Variable												
Value	?	?	?	?	?	?	?	?	?	?	?	?

Address	40	41	42	43	44	45	46	47	48	49	50	51
Variable	"Holberton"											
Value	H	o	l	b	e	r	t	o	n	\0	?	?

Note that the string `"Holberton"` is always present in the memory. We will see why later.

This is the memory right before the line `str[0] = 's';` is executed:



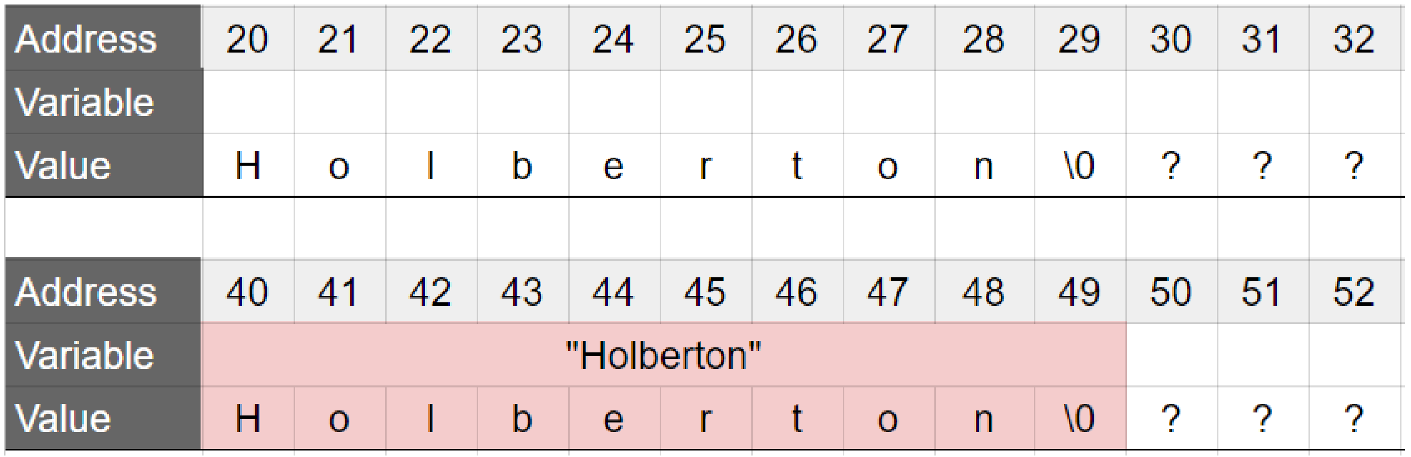
Address	20	21	22	23	24	25	26	27	28	29	30	31	32
Variable	str												
Value	H	o	l	b	e	r	t	o	n	\0	?	?	?

Address	40	41	42	43	44	45	46	47	48	49	50	51	52
Variable	"Holberton"												
Value	H	o	l	b	e	r	t	o	n	\0	?	?	?

Note the differences:

- The variable `str` is not a pointer, it's an array. `str` does not hold the memory address of the string `"Holberton"`, but a copy of it
- The string "Holberton" is copied into this array

And this is the memory state when the program leaves the function `print_school`:



Address	20	21	22	23	24	25	26	27	28	29	30	31	32
Variable													
Value	H	o	l	b	e	r	t	o	n	\0	?	?	?

Address	40	41	42	43	44	45	46	47	48	49	50	51	52
Variable	"Holberton"												
Value	H	o	l	b	e	r	t	o	n	\0	?	?	?

Note, again, that the string `"Holberton"` is still present in the memory.

Why would I need dynamic allocation?

So far we only have seen functions and programs that had fixed inputs. What happens when we do not know in advance how much memory you need and we will only know this at runtime, after compilation?

For instance, imagine we have to create a program that will store all the words contained in a file in an array. That file is passed as an argument to our program. There is no way we could know in advance how many words the file will contain. We can not declare a big array like `char *words[1024];` and assume that there will never be more than 1024 words in the file. That's when `malloc` and friends come to the rescue, and will permit us to allocate dynamically the amount of memory we need.

Dynamic allocation

Malloc

The `malloc` function allocates a specific number of bytes in memory and returns a pointer to the allocated memory. This memory will have read and write permissions.

- Prototype: `void *malloc(size_t size);`
- (/)where `void *` means it is a pointer to the type of your choice
- and `size` is the number of bytes your need to allocate

```
julien@ubuntu:~/c/malloc$ cat malloc_example.c
#include <stdio.h>
#include <stdlib.h>

/**
 * main - introduction to malloc and free
 *
 * Return: 0.
 */
int main(void)
{
    char *str;

    str = malloc(sizeof(char) * 3);
    str[0] = 'O';
    str[1] = 'K';
    str[2] = '\0';
    printf("%s\n", str);
    return (0);
}
julien@ubuntu:~/c/malloc$ gcc malloc_example.c -o m
julien@ubuntu:~/c/malloc$ ./m
OK
julien@ubuntu:~/c/malloc$
```

In the above example we use `malloc` to create a 3 byte allocated space in memory, and we fill this space with characters. Note the use the operator `sizeof`. It is very important because as you know, the size of the different types will be different on different machines: we want 3 times the size of a `char` (which happens to be 3 times 1 byte on our 64-bit machine). Always use `sizeof` for a better portability.

Let's see another example, with integers.

```
julien@ubuntu:~/c/malloc$ cat malloc_example2.c
#include <stdio.h>
#include <stdlib.h>

/**
 * main - introduction to malloc and free
 *
 * Return: 0.
 */
int main(void)
{
    int *tab;

    tab = malloc(sizeof(*tab) * 3);
    tab[0] = 98;
    tab[1] = -1024;
    tab[2] = 402;
    printf("%d, %d, %d\n", tab[0], tab[1], tab[2]);
    return (0);
}
julien@ubuntu:~/c/malloc$ gcc malloc_example2.c -o m2
julien@ubuntu:~/c/malloc$ ./m2
98, -1024, 402
julien@ubuntu:~/c/malloc$
```

In the above example, we are using `malloc` to create a space in memory where we can store three integers. Again, by using `sizeof` instead of directly a number of bytes (`12` for instance in this case if using a 64 bits machine) we are sure to get the right amount of memory, no matter what system we are compiling and running on.

memory

Contrary to local variables and function parameters, the memory that is allocated with `malloc` is not automatically released when the function returns.

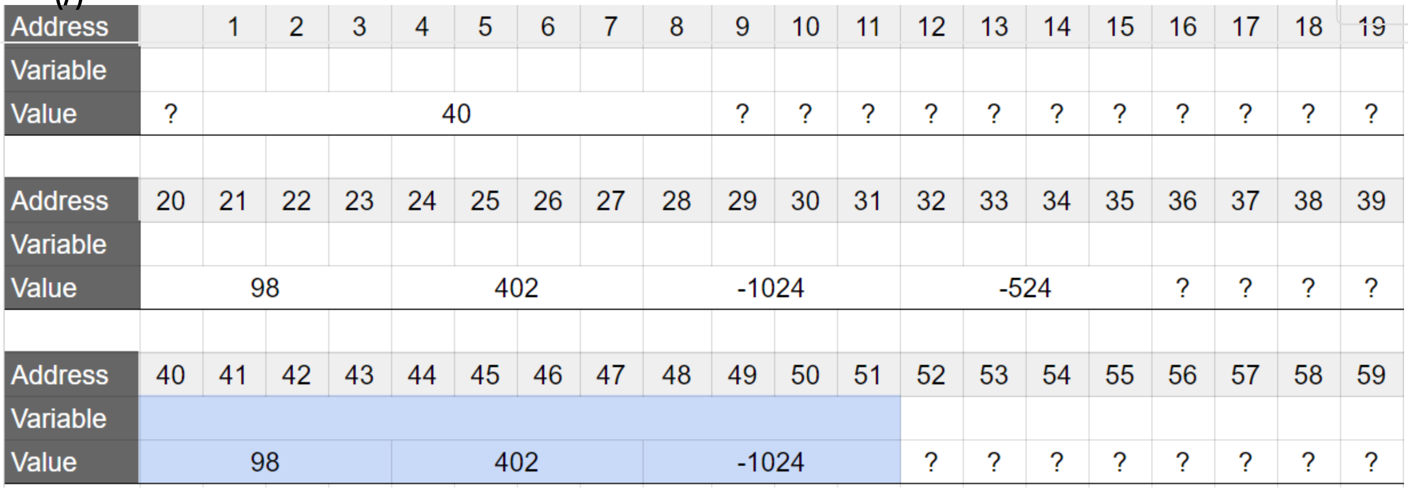
```
julien@ubuntu:~/c/malloc$ cat malloc_mem.c
#include <stdio.h>
#include <stdlib.h>

/**
 * m - stores 3 int in a new allocated space in memory and prints the sum
 * @n0: integer to store and print
 * @n1: integer to store and print
 * @n2: integer to store and print
 *
 * Return: nothing
 */
void m(int n0, int n1, int n2)
{
    int *t;
    int sum;

    t = malloc(sizeof(*t) * 3);
    t[0] = n0;
    t[1] = n1;
    t[2] = n2;
    sum = t[0] + t[1] + t[2];
    printf("%d + %d + %d = %d\n", t[0], t[1], t[2], sum);
}

/**
 * main - introduction to malloc and free
 *
 * Return: 0.
 */
int main(void)
{
    m(98, 402, -1024);
    return (0);
}
julien@ubuntu:~/c/malloc$
```

This is what the memory would look like before the function `m` returns:

Address		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Variable		t																		
Value	?	40								?	?	?	?	?	?	?	?	?	?	?
Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable	n0				n1				n2				sum							
Value	98				402				-1024				-524				?	?	?	?
Address	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
Variable																				
Value	98				402				-1024				?	?	?	?	?	?	?	?

And this will be the state of the memory after the function `m` returns:

(/)



Address		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
Variable																				
Value	?	40								?	?	?	?	?	?	?	?	?	?	?
Address	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39
Variable																				
Value	98				402				-1024				-524				?	?	?	?
Address	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59
Variable																				
Value	98				402				-1024				?	?	?	?	?	?	?	?

The memory is not initialized

Just like with automatic allocation, the memory allocated by `malloc` is not initialized.

Free

When you are using `malloc` , you have to handle the memory yourself. This means that:

- You need to keep track of all the addresses of the allocated memory (in a variable of type pointer)
- You have to deallocate every memory space you previously allocated yourself. If you do not do this, then your program can run out of memory. Your operating system could even kill your program while it is running

```
julien@ubuntu:~/c/malloc$ cat while_malloc.c
#include <stdio.h>
#include <stdlib.h>

/**
 * main - introduction to malloc and free
 *
 * Return: 0.
 */
int main(void)
{
    while (1)
    {
        malloc(sizeof(char) * 1024);
    }
    return (0);
}
julien@ubuntu:~/c/malloc$ gcc while_malloc.c -o killme
julien@ubuntu:~/c/malloc$ ./killme
Killed
julien@ubuntu:~/c/malloc$
```

The `free` function frees the memory space which have been allocated by a previous call to `malloc` (or `calloc` , or `realloc`).

- Prototype: `void free(void *ptr);`
- (/)where `ptr` is the address of the memory space previously allocated by and returned by a call to `malloc`

Example:

```
julien@ubuntu:~/c/malloc$ cat free_mem.c
#include <stdio.h>
#include <stdlib.h>

/**
 * m - stores 3 int in a new allocated space in memory and prints the sum
 * @n0: integer to store and print
 * @n1: integer to store and print
 * @n2: integer to store and print
 *
 * Return: nothing
 */
void m(int n0, int n1, int n2)
{
    int *t;
    int sum;

    t = malloc(sizeof(*t) * 3);
    t[0] = n0;
    t[1] = n1;
    t[2] = n2;
    sum = t[0] + t[1] + t[2];
    printf("%d + %d + %d = %d\n", t[0], t[1], t[2], sum);
    free(t);
}

/**
 * main - introduction to malloc and free
 *
 * Return: 0.
 */
int main(void)
{
    m(98, 402, -1024);
    return (0);
}
```

In the above example, the memory allocated by `malloc` is `free` ‘d with a call to the function `free` .

You should always `free` all `malloc` 'ed memory spaces.

Valgrind

When writing big and complex programs, it is not always easy to keep track of all allocated and deallocated memory. We can use the program Valgrind (/rltoken/tpNL9-gv6250TM7Yqd7OlQ) in order to ensure we are freeing all allocated memory. It is a programming tool for memory debugging, memory leak detection, and profiling.

```
julien@ubuntu:~/c/malloc$ cat malloc_mem.c
#include <stdio.h>
#include <stdlib.h>

/**
 * m - stores 3 int in a new allocated space in memory and prints the sum
 * @n0: integer to store and print
 * @n1: integer to store and print
 * @n2: integer to store and print
 *
 * Return: nothing
 */
void m(int n0, int n1, int n2)
{
    int *t;
    int sum;

    t = malloc(sizeof(*t) * 3);
    t[0] = n0;
    t[1] = n1;
    t[2] = n2;
    sum = t[0] + t[1] + t[2];
    printf("%d + %d + %d = %d\n", t[0], t[1], t[2], sum);
}

/**
 * main - introduction to malloc and free
 *
 * Return: 0.
 */
int main(void)
{
    m(98, 402, -1024);
    return (0);
}
julien@ubuntu:~/c/malloc$ gcc malloc_mem.c -o m
julien@ubuntu:~/c/malloc$ valgrind ./m
==3749== Memcheck, a memory error detector
==3749== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==3749== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3749== Command: ./m
==3749==
98 + 402 + -1024 = -524
==3749==
==3749== HEAP SUMMARY:
==3749==     in use at exit: 12 bytes in 1 blocks
==3749==   total heap usage: 2 allocs, 1 frees, 1,036 bytes allocated
==3749==
==3749== LEAK SUMMARY:
==3749==    definitely lost: 12 bytes in 1 blocks
==3749==    indirectly lost: 0 bytes in 0 blocks
==3749==      possibly lost: 0 bytes in 0 blocks
```

```
==3749==    still reachable: 0 bytes in 0 blocks
==3749==         suppressed: 0 bytes in 0 blocks
==3749== Rerun with --leak-check=full to see details of leaked memory
==3749==
==3749== For counts of detected and suppressed errors, rerun with: -v
==3749== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
julien@ubuntu:~/c/malloc$
```

In the above example, we **definitely lost: 12 bytes in 1 blocks**.

```
julien@ubuntu:~/c/malloc$ cat free_mem.c
#include <stdio.h>
#include <stdlib.h>

/**
 * m - stores 3 int in a new allocated space in memory and prints the sum
 * @n0: integer to store and print
 * @n1: integer to store and print
 * @n2: integer to store and print
 *
 * Return: nothing
 */
void m(int n0, int n1, int n2)
{
    int *t;
    int sum;

    t = malloc(sizeof(*t) * 3);
    t[0] = n0;
    t[1] = n1;
    t[2] = n2;
    sum = t[0] + t[1] + t[2];
    printf("%d + %d + %d = %d\n", t[0], t[1], t[2], sum);
    free(t);
}

/**
 * main - introduction to malloc and free
 *
 * Return: 0.
 */
int main(void)
{
    m(98, 402, -1024);
    return (0);
}
julien@ubuntu:~/c/malloc$ gcc free_mem.c -o f
julien@ubuntu:~/c/malloc$ valgrind ./f
==3763== Memcheck, a memory error detector
==3763== Copyright (C) 2002-2015, and GNU GPL'd, by Julian Seward et al.
==3763== Using Valgrind-3.11.0 and LibVEX; rerun with -h for copyright info
==3763== Command: ./f
==3763==
98 + 402 + -1024 = -524
==3763==
==3763== HEAP SUMMARY:
==3763==     in use at exit: 0 bytes in 0 blocks
==3763==   total heap usage: 2 allocs, 2 frees, 1,036 bytes allocated
==3763==
==3763== All heap blocks were freed -- no leaks are possible
==3763==
==3763== For counts of detected and suppressed errors, rerun with: -v
```

```
==3763== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
julien@ubuntu:~/c/malloc$
```

In the above example, we get **All heap blocks were freed – no leaks are possible**. This is what you should always aim for.

Don’t trust anyone

On error, `malloc` returns `NULL` . As for any other C library function, you should always check the `malloc` return value before using it. If you don’t you will run into segfaults.

```
julien@ubuntu:~/c/malloc$ cat malloc_segf.c
#include <stdlib.h>
#include <limits.h>

/**
 * main - introduction to malloc and free
 *
 * Return: 0.
 */
int main(void)
{
    char *s;

    while (1)
    {
        s = malloc(INT_MAX);
        s[0] = 'H';
    }
    return (0);
}
julien@ubuntu:~/c/malloc$ gcc malloc_segf.c -o s
julien@ubuntu:~/c/malloc$ ./s
Segmentation fault (core dumped)
julien@ubuntu:~/c/malloc$
```

This is an example on how to check the return value of `malloc` :

```
julien@ubuntu:~/c/malloc$ cat malloc_check.c
#include <stdlib.h>
#include <limits.h>
#include <stdio.h>

/**
 * main - introduction to malloc and free
 *
 * Return: 0.
 */
int main(void)
{
    char *s;
    int i;

    i = 0;
    while (1)
    {
        s = malloc(INT_MAX);
        if (s == NULL)
        {
            printf("Can't allocate %d bytes (after %d calls)\n", INT_MAX, i);
            return (1);
        }
        s[0] = 'H';
        i++;
    }
    return (0);
}
julien@ubuntu:~/c/malloc$ gcc malloc_check.c -o c
julien@ubuntu:~/c/malloc$ ./c
Can't allocate 2147483647 bytes (after 0 calls)
julien@ubuntu:~/c/malloc$
```

String literals and .rodata (advanced)

The string literals are stored in your executable at compilation. The way it is stored is actually dependent on both the operating system you are using and the linker. This is what happens when you compile the program on most modern operating systems:

- The compiler puts the string into a read-only data-section, usually `.rodata`
- The linker collects all the data in such read-only sections and puts them into a single segment. This segment resides in the executable file and is flagged with a "read only"-attribute.

When you run the program, the operation system executable loader loads the executable (or maps it into memory to be more exact). Once this is done, the loader walks the sections and sets access-permissions for each segment. For a read-only data segment it will most likely disable code-execute and write access. Code (for example, your functions) gets execute rights but no write access. Ordinary data like static variables gets read and write access and so on…

One very easy way to check that the string literal is actually stored in your executable is to use the command `strings` (`man strings`).

```
julien@ubuntu:~/c/malloc$ cat segf.c
/**
 *  segf - Let's segfault \o/
 *
 * Return: nothing.
 */
void segf(void)
{
    char *str;

    str = "Holberton";
    str[0] = 's';
}

/**
 *  main - concept introduction
 *
 * Return: 0.
 */
int main(void)
{
    segf();
    return (0);
}
julien@ubuntu:~/c/malloc$ gcc segf.c -o segf
julien@ubuntu:~/c/malloc$ strings segf
/lib64/ld-linux-x86-64.so.2
,o8E0
libc.so.6
__libc_start_main
__gmon_start__
GLIBC_2.2.5
UH-0
AWAVA
AUATL
[]A\A]A^A_
Holberton
[...]
julien@ubuntu:~/c/malloc$
```

It is also possible to use `objdump` to print the `.rodata` section contents.

```
julien@ubuntu:~/c/malloc$ objdump -s -j .rodata segf

segf:     file format elf64-x86-64

Contents of section .rodata:
 400580 01000200 486f6c62 6572746f 6e00      ....Holberton.
julien@ubuntu:~/c/malloc$
```

String literals are not always read-only

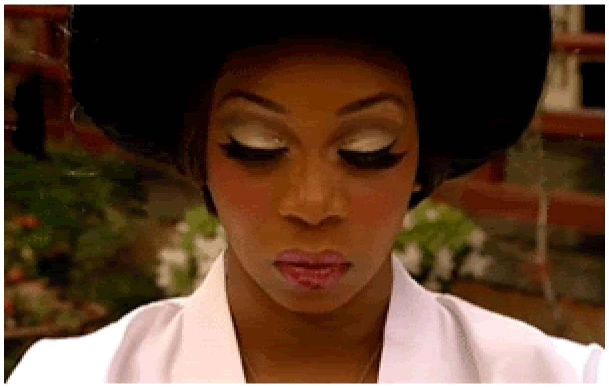
On most modern operating system, string literals will be read-only. But this is (and certainly was) not always the case. It actually depends which operating system and linker you are using. For instance, if you were to compile `segf.c` for DOS, the program would not segfault. It would actually run because the memory where the string "Holberton" would be stored would have write access. This would happen because the DOS loader does not know about read-only sections.

String literals are not always stored as strings in the executable

Note that we are using a 64-bit machine for the following example

Wait… WAT?

```
julien@ubuntu:~/c/strings$ cat print_school.c
#include <stdio.h>

/**
 * print_school - prints "school"
 *
 * Return: nothing.
 */
void print_school(void)
{
    char str[] = "Holberton";

    str[0] = 's';
    printf("%s\n", str);
}

/**
 *  main - concept introduction
 *
 * Return: 0.
 */
int main(void)
{
    print_school();
    return (0);
}
julien@ubuntu:~/c/strings$ gcc print_school.c -o ps
julien@ubuntu:~/c/strings$ strings ps | grep Holberton
julien@ubuntu:~/c/strings$ objdump -s -j .rodata ps

ps:     file format elf64-x86-64

Contents of section .rodata:
 2000 01000200                             ....
julien@ubuntu:~/c/strings$ ./ps
Holberton
julien@ubuntu:~/c/strings$
```



So how does the program know what to print, without actually using the string? Well, sometimes (actually more often than you think), the compiler will optimize your code and modify it, without telling you. Let's analyse what happens with `objdump` :

(/)

```
julien@ubuntu:~/c/strings$ objdump -d -Mintel ps

ps:     file format elf64-x86-64
[...]
0000000000400596 <print_school>:
  400596:   55                      push   rbp
  400597:   48 89 e5                mov    rbp,rsp
  40059a:   48 83 ec 20             sub    rsp,0x20
  40059e:   64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
  4005a5:   00 00
  4005a7:   48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
  4005ab:   31 c0                   xor    eax,eax
  4005ad:   48 b8 48 6f 6c 62 65    movabs rax,0x6f747265626c6f48
  4005b4:   72 74 6f
  4005b7:   48 89 45 e0             mov    QWORD PTR [rbp-0x20],rax
  4005bb:   66 c7 45 e8 6e 00       mov    WORD PTR [rbp-0x18],0x6e
  4005c1:   c6 45 e0 68             mov    BYTE PTR [rbp-0x20],0x68
  4005c5:   48 8d 45 e0             lea    rax,[rbp-0x20]
  4005c9:   48 89 c7                mov    rdi,rax
  4005cc:   e8 8f fe ff ff          call   400460 <puts@plt>
  4005d1:   90                      nop
  4005d2:   48 8b 45 f8             mov    rax,QWORD PTR [rbp-0x8]
  4005d6:   64 48 33 04 25 28 00    xor    rax,QWORD PTR fs:0x28
  4005dd:   00 00
  4005df:   74 05                   je     4005e6 <print_school+0x50>
  4005e1:   e8 8a fe ff ff          call   400470 <__stack_chk_fail@plt>
  4005e6:   c9                      leave
  4005e7:   c3                      ret
    START NEW OBJDUMP
    0000000000001169 <print_school>:
    1169:   f3 0f 1e fa             endbr64
    116d:   55                      push   rbp
    116e:   48 89 e5                mov    rbp,rsp
    1171:   48 83 ec 10             sub    rsp,0x10
    1175:   64 48 8b 04 25 28 00    mov    rax,QWORD PTR fs:0x28
    117c:   00 00
    117e:   48 89 45 f8             mov    QWORD PTR [rbp-0x8],rax
    1182:   31 c0                   xor    eax,eax
    1184:   c7 45 f1 53 63 68 6f    mov    DWORD PTR [rbp-0xf],0x6f686353
    118b:   66 c7 45 f5 6f 6c       mov    WORD PTR [rbp-0xb],0x6c6f
    1191:   c6 45 f7 00             mov    BYTE PTR [rbp-0x9],0x0
    1195:   c6 45 f1 73             mov    BYTE PTR [rbp-0xf],0x73
    1199:   48 8d 45 f1             lea    rax,[rbp-0xf]
    119d:   48 89 c7                mov    rdi,rax
    11a0:   e8 bb fe ff ff          call   1060 <puts@plt>
    11a5:   90                      nop
    11a6:   48 8b 45 f8             mov    rax,QWORD PTR [rbp-0x8]
    11aa:   64 48 33 04 25 28 00    xor    rax,QWORD PTR fs:0x28
    11b1:   00 00
    11b3:   74 05                   je     11ba <print_school+0x51>
    11b5:   e8 b6 fe ff ff          call   1070 <__stack_chk_fail@plt>
    11ba:   c9                      leave
    11bb:   c3                      ret
  (/)   END NEW OBJDUMP
[...]
```

`4005ad: 48 b8 48 6f 6c 62 65 movabs rax,0x6f747265626c6f48` is where the trick happens. The program moves the value `0x6f747265626c6f48` into the 64-bit register `rax`, and then copies this value on the stack, at the address where the array `str` is stored (`4005b7: 48 89 45 e0 mov QWORD PTR [rbp-0x20],rax`). Note that the value `0x6f747265626c6f48` takes 8 bytes in memory. As a result, this is what the memory looks like after the execution of `mov QWORD PTR [rbp-0x20],rax`:

Address	60	61	62	63	64	65	66	67	68	69	70	71	72
Variable	str												
Value	0x48	0x6f	0x6c	0x62	0x65	0x72	0x74	0x6f	?	?	?	?	?

Since `str` is an array of `char`, those bytes stored in `str` will be used as ASCII values of chars. Looking quickly at the ASCII table (`man ascii`), we can translate this into chars:

- 0x48 = 'H'
- 0x6f = 'o'
- 0x6c = 'l'
- 0x62 = 'b'
- 0x65 = 'e'
- 0x72 = 'r'
- 0x74 = 't'
- 0x6f = 'o'

This is the beginning of the string `Holberton`! We are just missing two chars: `n` and `\0`. This is what the line `4005bb: 66 c7 45 e8 6e 00 mov WORD PTR [rbp-0x18],0x6e` is taking care of:

- `0x6e` is the ASCII value of the char `n`
- `mov WORD` ensures that you actually moves `0x006e`, as a `WORD` is two-byte long*, and `0x00` is the ASCII code for the `\0` char, that marks the end of a C string

* *In this context, a `WORD` is 2-byte long, as we are referring to the Intel assembly language*

After the execution of the line `mov WORD PTR [rbp-0x18],0x6e`, the memory looks like this:

Address	60	61	62	63	64	65	66	67	68	69	70	71	72
Variable	str												
Value	0x48	0x6f	0x6c	0x62	0x65	0x72	0x74	0x6f	0x6e	0x00	?	?	?

And you have the complete string `Holberton` stored in the array. In other words, it is possible that a string is stored as “code” and not as “data” in your executable.

(/)

HolbertoH

Note that if you look at the strings of the executable with `strings` you will find the string `HolbertoH`.

```
julien@holberton:~/c/strings$ strings ph | grep HolbertoH
HolbertoH
julien@holberton:~/c/strings$
```

The way the program `strings` works: `strings` prints the printable character sequences that are at least 4 characters long and are followed by an unprintable character. It doesn't know about sections and tries to interpret every bytes of the executable as a char. Therefore, as our executable contains the bytes `48 b8 48 6f 6c 62 65 72 74 6f`, immediately followed by the bytes `48 89 45 e0`,

```
  4005ad:   48 b8 48 6f 6c 62 65    movabs rax,0x6f747265626c6f48
  4005b4:   72 74 6f
  4005b7:   48 89 45 e0             mov    QWORD PTR [rbp-0x20],rax
```

`strings` recognizes `Holberto` (`48 b8 48 6f 6c 62 65 72 74 6f`), followed by an `H` (`48`) and followed by a non-printable char (`89`), and prints `HolbertoH`