

# **PROGRAMMAZIONE PROCEDURALE**

**A.A. 2025/2026**



# DYNAMIC MEMORY MANAGEMENT



# WHY DYNAMIC MEMORY

- ④ When you're writing a program, you often don't know how much data it will have to process.
- ④ Or you can anticipate that the amount of data to process will vary widely.
- ④ In these cases, efficient resource use demands that you allocate memory only as you actually need it at runtime, and release it again as soon as possible.
- ④ This is the principle of dynamic memory management, which also has the advantage that a program doesn't need to be rewritten in order to process larger amounts of data on a system with more available memory.

# WHY DYNAMIC MEMORY

- ④ Dynamic memory is a part of memory with different properties
  - ✓ It's the programmer who decides when to create an object and when to destroy it (i.e., remove it from memory)

# FUNCTIONS

- ④ The standard library provides the following four functions for dynamic memory management:
  - ④ Allocate a new block of memory.
    - ✓ **malloc( ), calloc( )**
  - ④ Resize an allocated memory block.
    - ✓ **realloc( )**
  - ④ Release allocated memory.
    - ✓ **free( )**
- ④ All of these functions are declared in the header file *stdlib.h*.
- ④ The size of an object in memory is specified as a number of bytes.
- ④ Various header files, including *stdlib.h*, define the type `size_t` specifically to hold information of this kind.
- ④ The `sizeof` operator, for example, yields a number of bytes with the type `size_t`.

# ALLOCATING MEMORY DYNAMICALLY



# MALLOC()

Ⓢ The **malloc()** function reserves a contiguous memory block whose size in bytes is at least *size*. When a program obtains a memory block through **malloc( )**, its contents are undetermined.

✓ `void *malloc( size_t size );`

# CALLOC()

- ④ The `calloc( )` function reserves a block of memory whose size in bytes is at least ***count* × *size***. In other words, the block is large enough to hold an array of *count* elements, each of which takes up *size* bytes.
- ④ Furthermore, **`calloc( )`** initializes every byte of the memory with the value 0.
  - ✓ `void *calloc( size_t count, size_t size );`



# MORE ON

- ④ Both functions return a pointer to void, also called a *typeless pointer*.
- ④ The pointer's value is the address of the first byte in the memory block allocated, or a **null pointer** if the memory requested is not available.
- ④ When a program assigns the void pointer to a pointer variable of a different type, the compiler implicitly performs the appropriate type conversion.
- ④ When you access locations in the allocated memory block, the type of the pointer you use determines how the contents of the location are interpreted.

# EXAMPLE

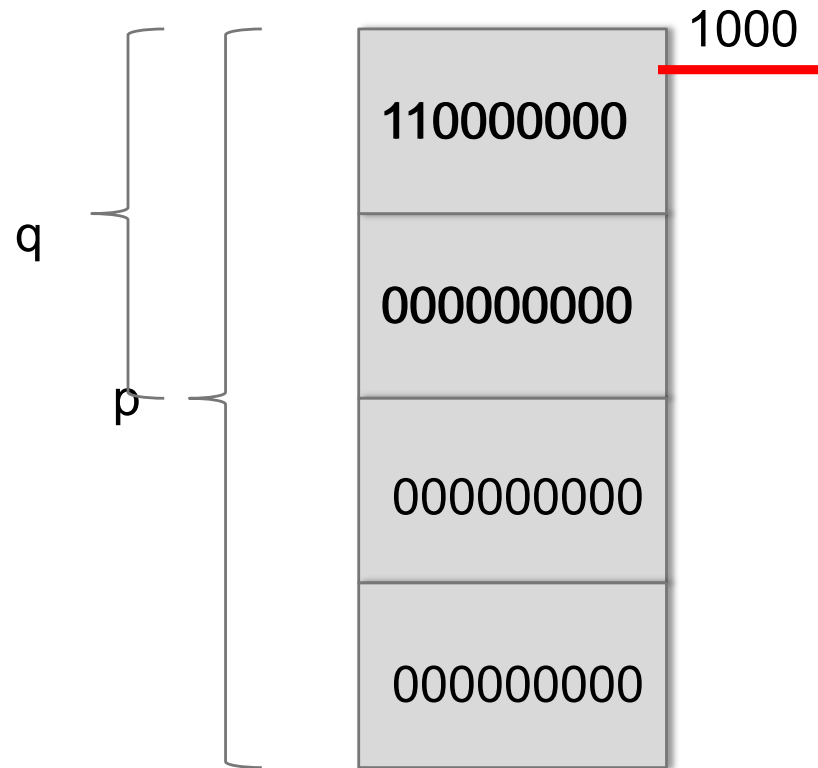
```
float *myFunc( size_t n )
{
    // Reserve storage for an object of type double.
    double *dPtr = malloc( sizeof(double) );
    if ( dPtr == NULL )                                // Insufficient memory.
    {
        /* ... Handle the error ... */
        return NULL;
    }
    else {                                             // Got the memory: use it.
        *dPtr = 0.07;
        /* ... */
    }
}
```

# EXAMPLE

```
malloc( sizeof(int) );  
  ↙  
void*
```

```
int*p = (int*) malloc(sizeof(int));  
*p = 3;
```

```
short int*q = (short int*) malloc(sizeof(int));  
*q = 3;
```



# CHARACTERISTICS OF ALLOCATED MEMORY

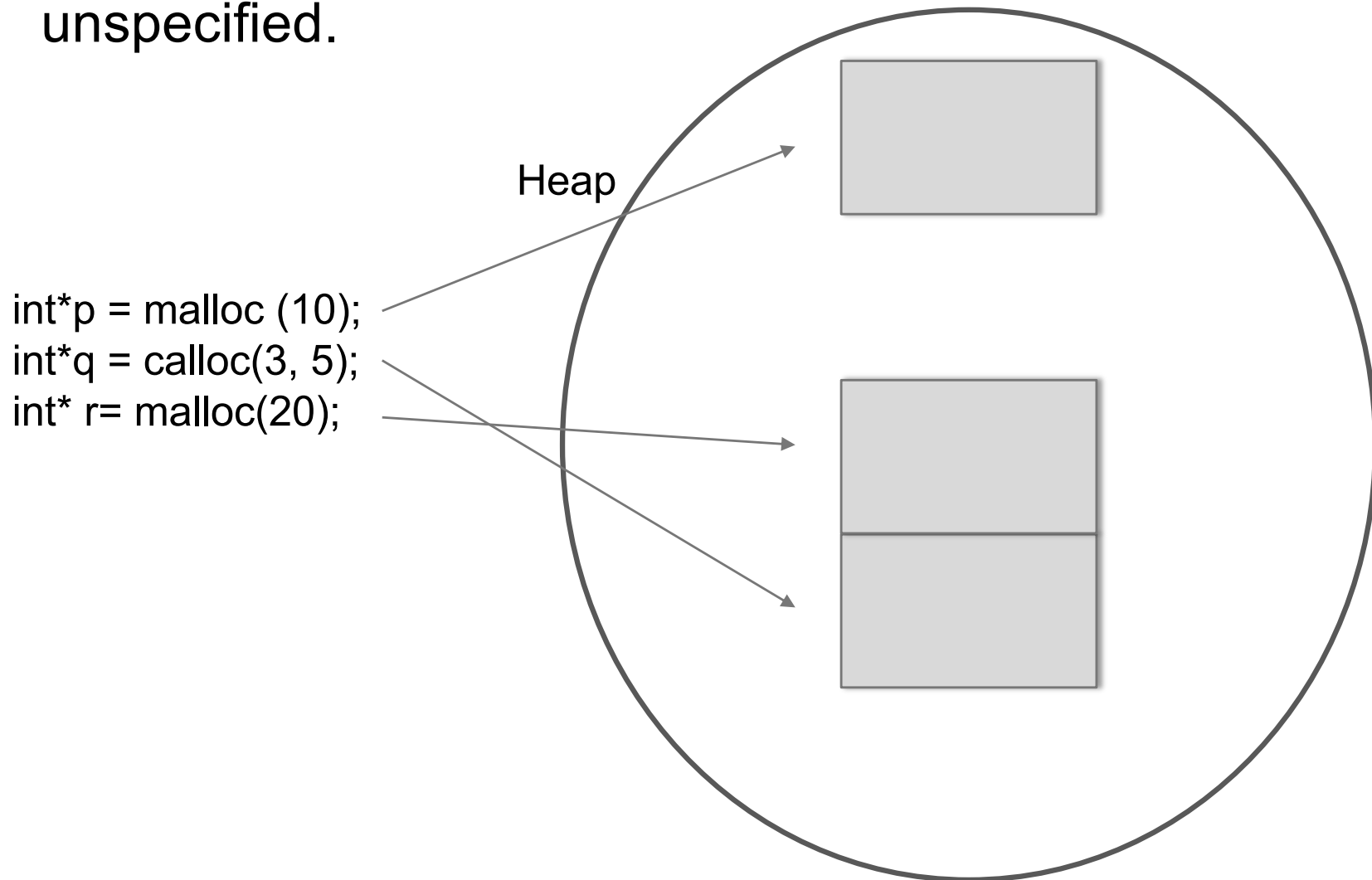


# CHARACTERISTICS 1

- ④ A successful memory allocation call yields a pointer to the beginning of a memory block. “The beginning” means that the pointer’s value is equal to the lowest byte address in the block.
- ④ An allocated memory block stays reserved for your program until you explicitly release it by calling `free( )` or `realloc( )`.
- ✓ In other words, the storage duration of the block extends from its allocation to its release, or to end of the program.

# CHARACTERISTICS 2

- ④ The arrangement of memory blocks allocated by successive calls to `malloc()`, `calloc()`, and/or `realloc()` is unspecified.



# RESIZING AND RELEASING MEMORY



# FREE() AND REALLOC()

- ④ When you no longer need a dynamically allocated memory block, you should give it back to the operating system. You can do this by calling the function **free( )**.
- ④ Alternatively, you can increase or decrease the size of an allocated memory block by calling the function **realloc()**.



# FREE()

- ④ The `free( )` function releases the dynamically allocated memory block that begins at the address in *ptr*. A **null pointer** value for the *ptr* argument is permitted, and such a call has no effect.
  - ✓ `void free( void *ptr );`

# REALLOC()

- ⌚ The `realloc( )` function releases the memory block addressed by *ptr* and allocates a new block of *size* bytes, returning its address. The new block may start at the same address as the old one.
  - ✓ `void *realloc( void *ptr, size_t size );`
- ⌚ `realloc( )` also preserves the contents of the original memory block—up to the size of whichever block is smaller.
- ⌚ If the new memory block is larger than the original, then the values of the additional bytes are unspecified.

# REALLOC()

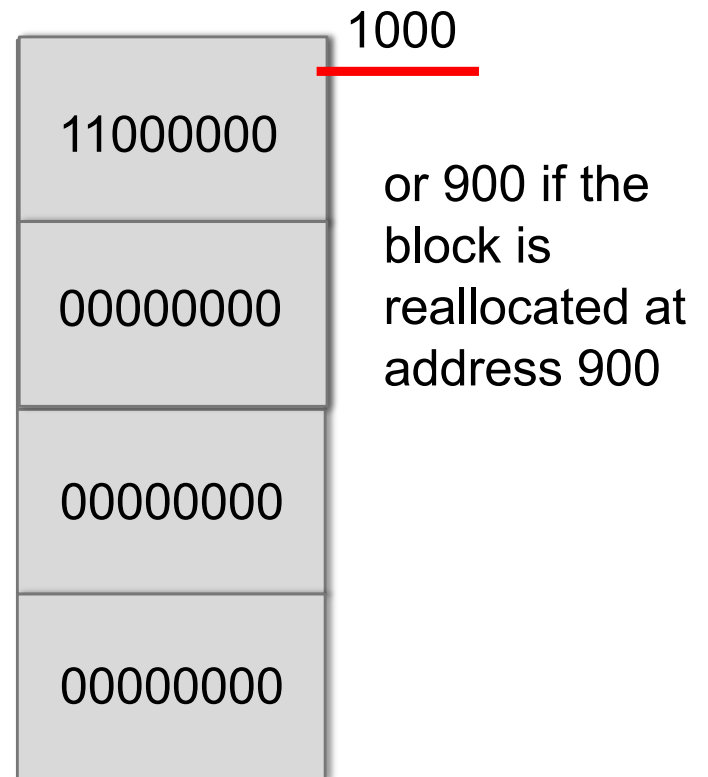
- ⊙ It is permissible to pass a null pointer to `realloc( )` as the argument *ptr*.
  - ✓ If you do, then `realloc()` behaves similarly to `malloc()`, and reserves a new memory block of the specified size.
- ⊙ The `realloc()` function returns a null pointer if it is unable to allocate a memory block of the size requested.
  - ✓ In this case, it does not release the original memory block or alter its contents.

# POINTERS IN FREE() & REALLOC()

- ④ You may pass these functions only a pointer value obtained from a prior call to `malloc( )`, `calloc( )`, or `realloc( )`.
- ④ If the pointer argument passed to `free( )` or `realloc( )` has any other value, or if you try to free a memory block that has already been freed, the program's behavior is undefined.

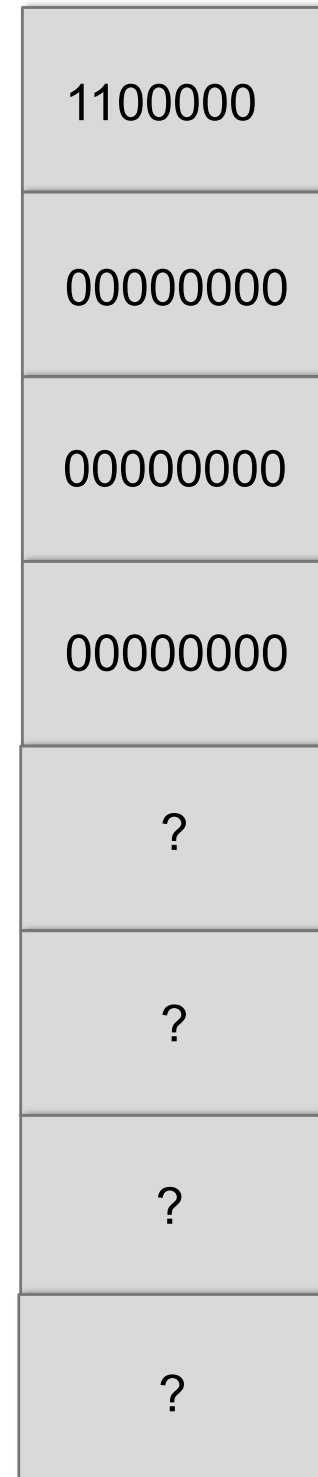
# REALLOC

```
int* p= (int*) malloc(sizeof(int));  
*p= 3;  
int* p= (int*) realloc(p, sizeof(short int));
```



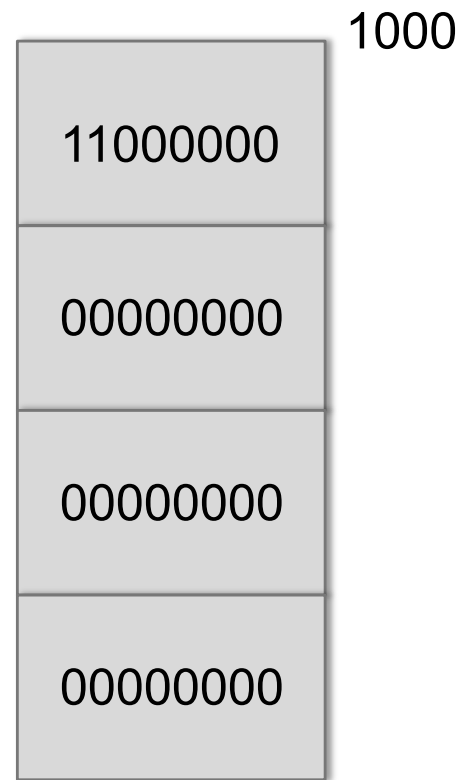
# REALLOC

```
int* p= (int*) malloc(sizeof(int));  
*p= 3;  
int* p= (int*) realloc(p, sizeof(long long int));
```



# FREE


```
int* p= (int*) malloc(sizeof(int));  
*p= 3;  
free(p);
```



# MORE INFO

- ⊙ The memory management functions keep internal records of the size of each allocated memory block.
  - ✓ This is why the functions `free( )` and `realloc( )` require only the starting address of the block to be released, and not its size.
- ⊙ There is no way to test whether a call to the `free( )` function is successful, because it has no return value.

```
int* p= (int*) malloc(4);  
int* q= (int*) calloc(3, 4);  
free(p);
```

	Starting address	Amount of bytes	
	1000	4	1000
	1020	12	1020



# EXAMPLE

```
int    *p_array;
double *d_array;

// Allocated in the heap
p_array = (int *) malloc(sizeof(int)*5);    // allocate 5 ints
d_array = (double *) malloc(sizeof(double)*6); // allocate 6 doubles

for(int i=0; i < 5; i++) {
    p_array[i] = 0;
}

for(int i=0; i < 6; i++) {
    d_array[i]= 0;
}

free (p_array);
p_array = (int *) malloc(sizeof(int)*100);    // allocate 100 ints
```

ERRORS



# ERRORS

- ⊗ Since in C is up to the programmer to manage dynamic memory (or heap) with allocation/deallocation and pointers to objects in the heap, this is more error-prone.
- ⊗ In other languages (e.g., Java) run-time support of the language (i.e., Java Virtual Machine) implements **garbage collection**.
- ⊗ The garbage collector, or just collector, automatically attempts to reclaim garbage, or memory occupied by objects that are no longer in use by the program.
- ⊗ Unreferenced objects are are automatically removed from the heap

```
1.Employee e=new Employee();  
2.e=null;
```

# GARBAGE COLLECTION

- ⌚ In C, `int p= malloc(sizeof(int)); p= NULL;` does not deallocate the memory once pointed by p
  - ✓ Free is always needed to deallocate memory
- ⌚ Typically, garbage collection has certain disadvantages, including consuming additional resources, performance impacts, possible stalls in program execution, and incompatibility with manual resource management.

# DEALLOCATION

- ⌚ Always remember to deallocate objects with `free()` when you do not need them anymore
- ⌚ Otherwise, the object may become unreachable: not referenced by any pointer
- ⌚ In this is repeated, the heap can be fully consumed: it is limited, as memory in general.
  - ✓ When it is fully consumed, a program may crash or however, you cannot allocate what you want
  - ✓ It is a problem if your program runs for a long period (or forever: servers)
- ⌚ These are defined as **memory leaks**

# EXAMPLE

```
#include<stdlib.h>

int main( ) {
    int* i_array= (int*) calloc(4, sizeof(int));
    i_array= (int*) malloc (20);

    // ..... Other instructions
}
```

The 16 bytes allocated starting from `i_array` are not deallocated from memory  
They just become unreachable: it's a memory leak

# FREQUENT ERRORS

- ④ Frequent errors are
  - ✓ memory usage after a call to **free** (**dangling pointer**) or
  - ✓ dereferencing a pointer assigned to NULL,
  - ✓ calling free twice ("double free"), etc.,
- ④ Usually they cause a **segmentation fault** and results in a crash of the program.
- ④ Your program is only allowed to touch memory that belongs to it -- the memory previously mentioned. Any access outside that area will cause a segmentation fault.

# DEFERENCING A NULL POINTER

```
int main(void) {  
    int*p =NULL;  
    int c = 3 + *p;  
}
```

```
MacBook-Francesco:Programmi francescosantini$  
./test  
Segmentation fault: 11
```



# DANGLING POINTERS

- Ⓢ Dangling pointers arise during object destruction, when an object that has an incoming reference is deleted or deallocated, without modifying the value of the pointer, so that the pointer still points to the memory location of the deallocated memory.

```
if (message_type == value_1) {  
    /* Process message type 1 */  
    free(message);  
}
```

```
if (message_type == value_1) {  
    /* Process message type 1 */  
    free(message);  
    message = NULL;  
}
```

To be sure not to use that memory, assign null to pointer

# ERROR

```
int main() {  
    int* p1, p2;  
    int n = 30;  
    p1 = &n;  
    p2 = &n;  
}
```

p2 is not a pointer: warning, but an error in practice (how to fix?)

```
int *p1, *p2;
```

# ERROR

```
int main() {  
    int* p1;  
    int n = *p1;  
    printf("%d", n);  
    return 0;  
}
```

p1 is not assigned, it can be anything

# ERROR

```
int main() {  
    int* p1;  
    int m;  
    p1 = &m;  
    int n = *p1;  
    printf("%d", n);  
    return 0;  
}
```

\*p1 can be anything because m is not initialized

# ERROR

```
int main() {  
    int* p1;  
    int m = 100;  
    p1 = &m;  
    free(p1);  
    return 0;  
}
```

Free can be called only on memory allocated in the heap

# ERROR

```
int main() {
    int* p1;
    if ((p1 = (int*) malloc(sizeof(int))) == NULL)
        return 1;

    *p1 = 99;
    free(p1);
    *p1 = 100;
    return 0;
}
```

After a free, the pointer cannot be used to access to that memory

# ERROR

```
int main() {  
    char* str1 = (char*)malloc(strlen("Thunderbird") + 1);  
    strcpy(str1, "Thunderbird");  
    //...  
    free(str1);  
    free(str1);  
}
```

Two free on the same pointer

# ERROR

```
int main() {  
    int* p1;  
    int m = 100;  
    p1 = &m;  
    *(p1++);  
    printf("%d\n", *p1);  
    printf("%d\n", m);  
    return 0;  
}
```

Postfix ++ has higher precedence than \*



# C

- Ⓢ C assumes that **you** are a genius.
- Ⓢ It will do pretty much anything you tell it to.
- Ⓢ It doesn't make assumptions.
- Ⓢ It doesn't guess.
- Ⓢ It does exactly what you say.

# EXAMPLE

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

int main()
{
    int i= 0;
    while (1) {

        int* p= (int*) malloc(1000000000000000);
        i++;
        printf("Number of times: %d\n", i);

        if (p == NULL)
            break;
    }
}
```

# OUTPUT

```
MacBook-Francesco:ProgrammI francescosantini$ ./a.out
Number of times: 1
Number of times: 2
Number of times: 3
Number of times: 4
Number of times: 5
Number of times: 6
Number of times: 7
Number of times: 8
Number of times: 9
Number of times: 10
Number of times: 11
Number of times: 12
Number of times: 13
Number of times: 14
a.out(19788,0x7fff9fd113c0) malloc: ***
mach_vm_map(size=1000000000000000) failed (error code=3)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
Number of times: 15
```

# SU LIBRO

Sezione 12.3

