Dynamic Memory

Politecnico
di Torino

Dynamic Memory

Dynamic memory allocation allows programmers to write code that requires an amount of memory that is **NOT FIXED** "a priori."

The memory can be **dynamically allocated** or **freed** during the program execution.

Dynamic Memory

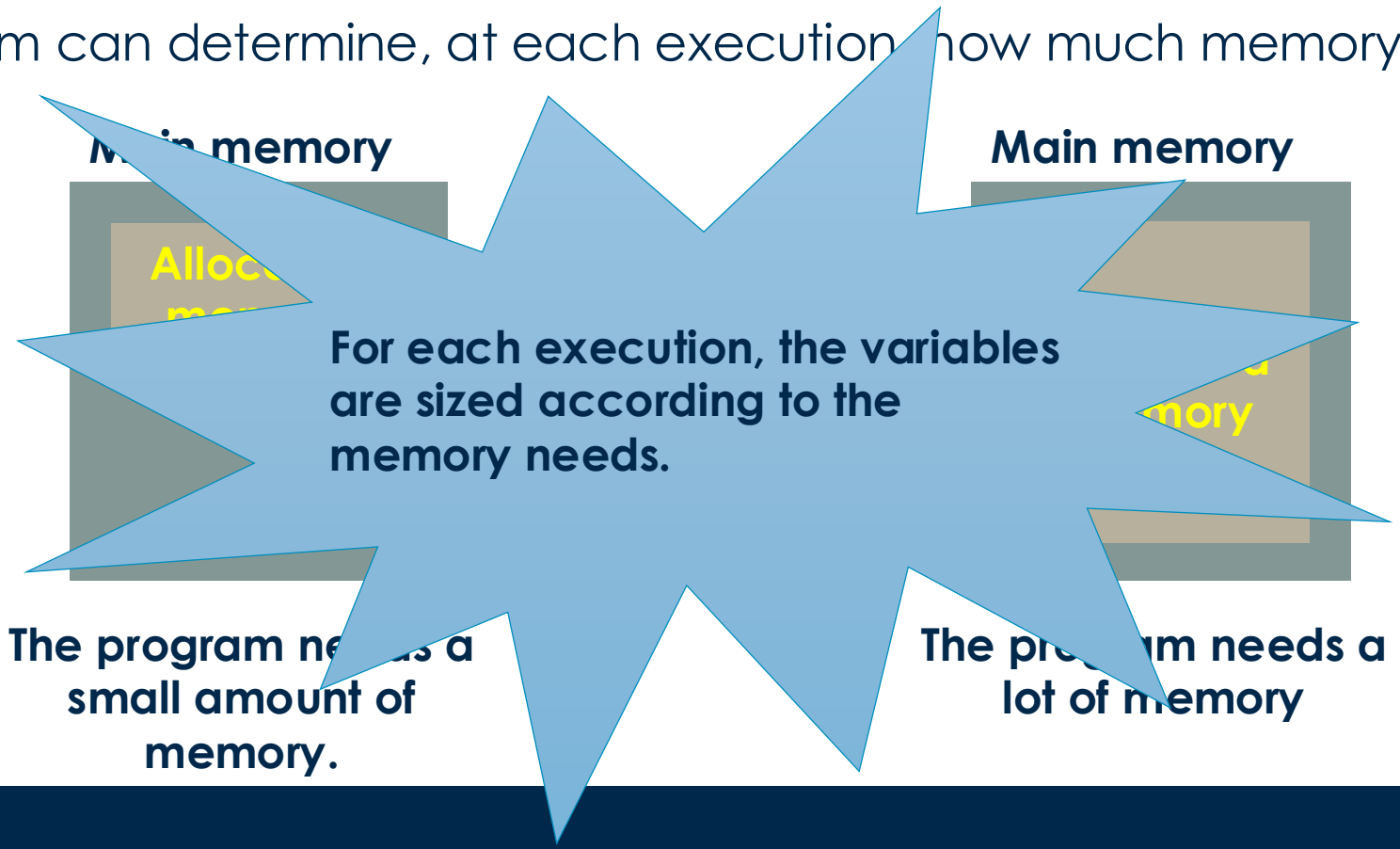
There are two scenarios in which *dynamic memory allocation* can be exploited:

1. The program can determine, at each execution, how much memory it needs

Dynamic Memory

There are two scenarios in which *dynamic memory allocation* can be exploited:

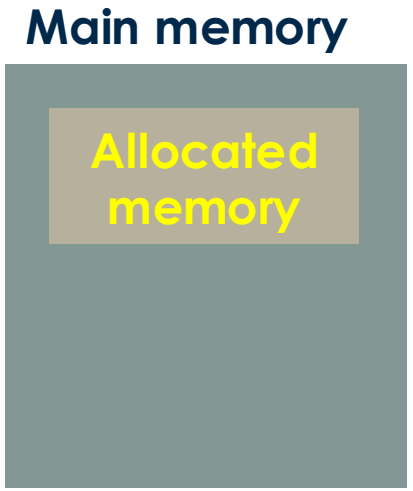
1. The program can determine, at each execution, how much memory it needs



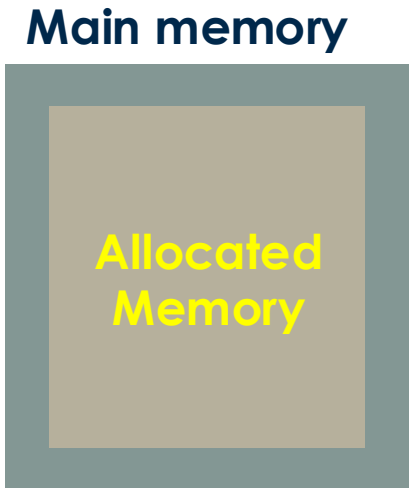
Dynamic Memory

There are two scenarios in which *dynamic memory allocation* can be exploited:

1. The program can determine, at each execution, how much memory it needs



The program needs a small amount of memory.



The program needs a lot of memory

Dynamic Memory

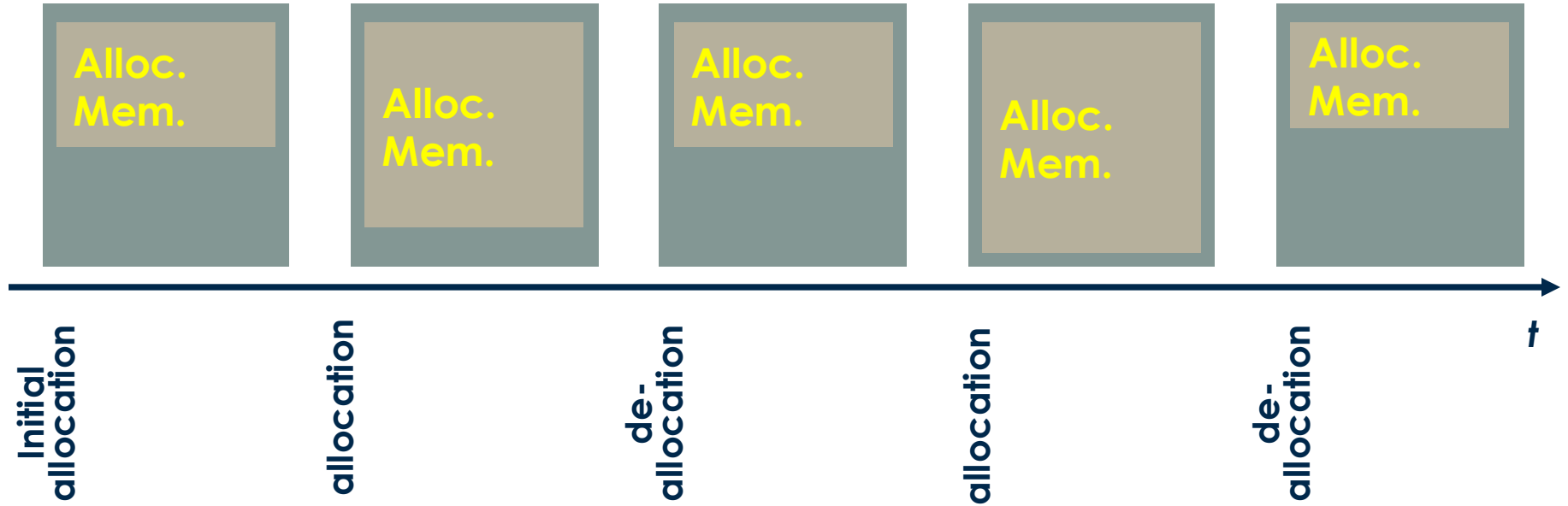
There are two scenarios in which *dynamic memory allocation* can be exploited:

1. The program can determine, at each execution, how much memory it needs
2. During execution, the program needs a variable amount of memory

Dynamic Memory

There are two scenarios in which *dynamic memory allocation* can be exploited:

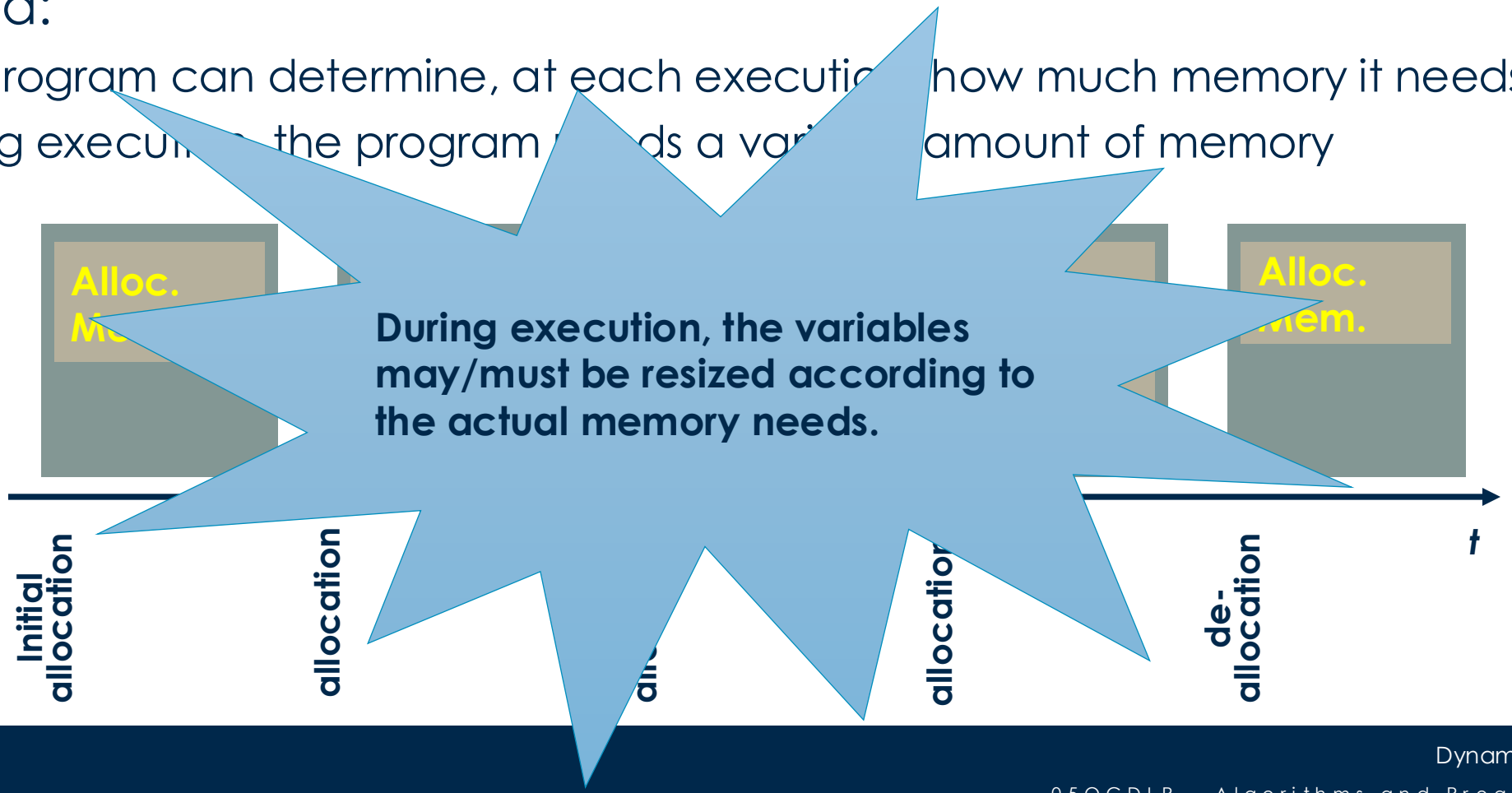
1. The program can determine, at each execution, how much memory it needs
2. During execution, the program needs a variable amount of memory



Dynamic Memory

There are two scenarios in which *dynamic memory allocation* can be exploited:

1. The program can determine, at each executi… how much memory it needs
2. During execu… the program …ds a va… amount of memory



How can we **dynamically** allocate and deallocate memory?

void * malloc (…)

The main function to allocate memory in C is:

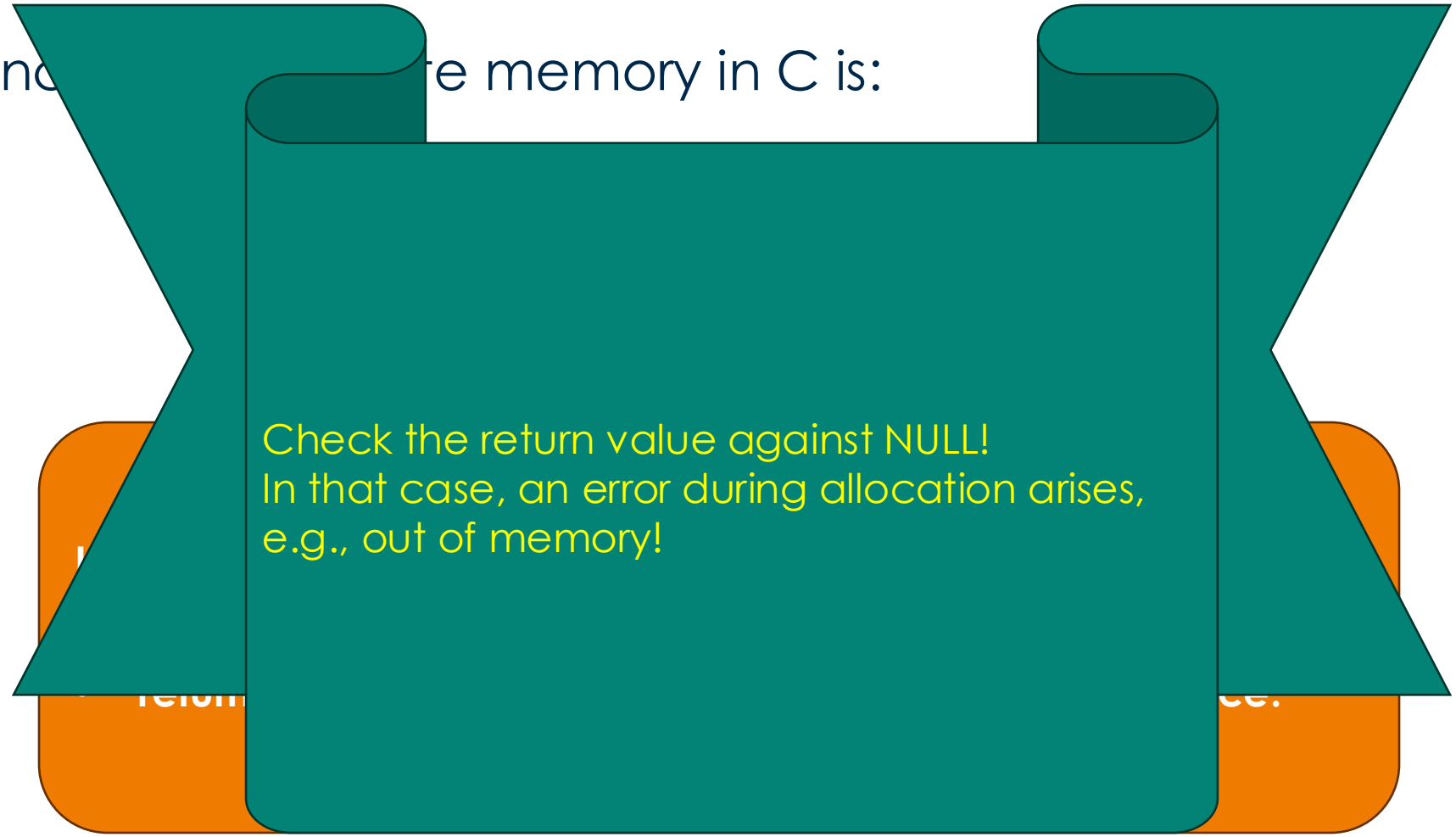
```
void * malloc ( <memory_size> );
```

It asks the Operating System to:

- **allocate a memory portion of bytes having size equal to <memory_size>**
- **return the pointer to the beginning of the allocated space.**

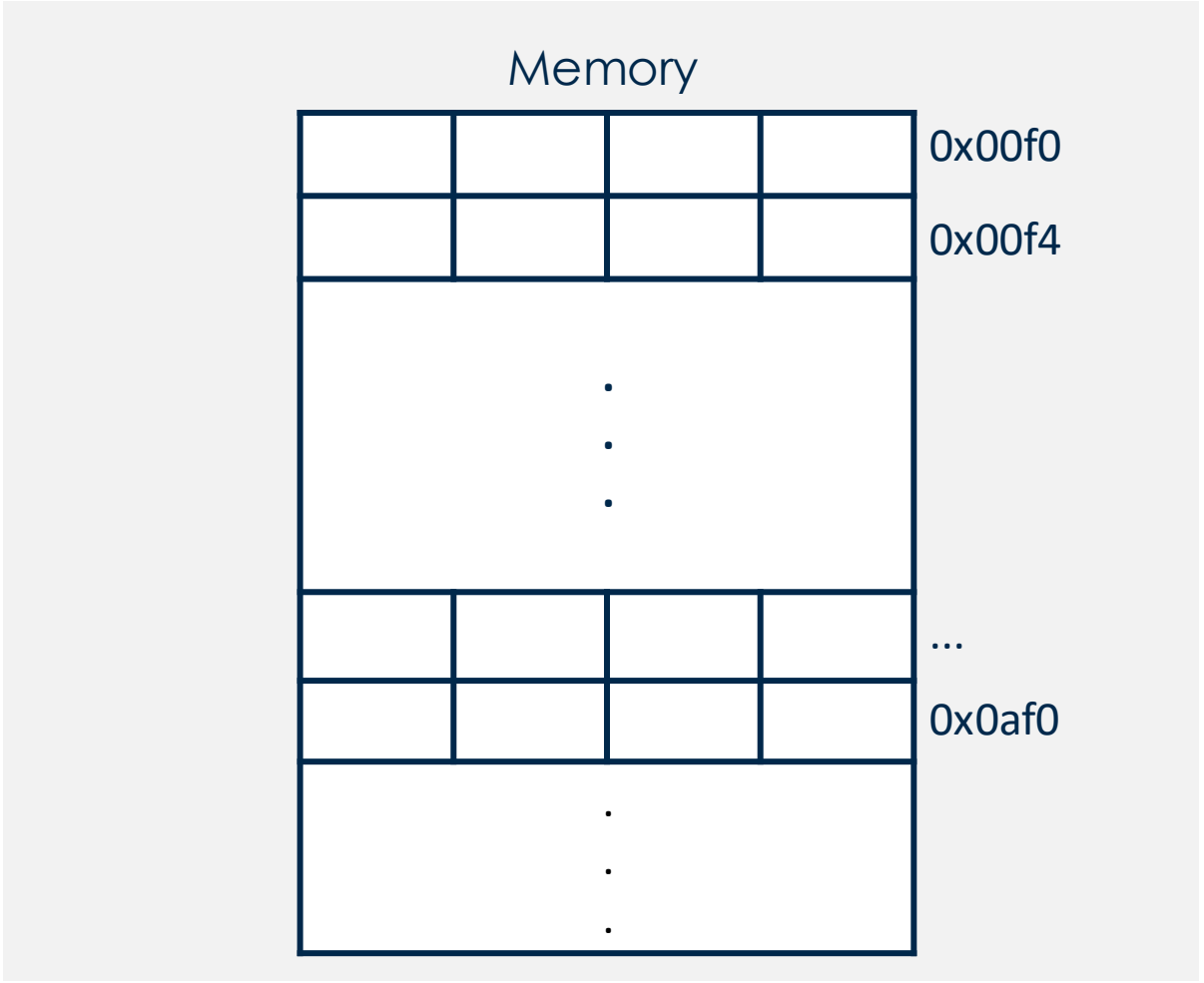
void * malloc (…)

The main func e memory in C is:



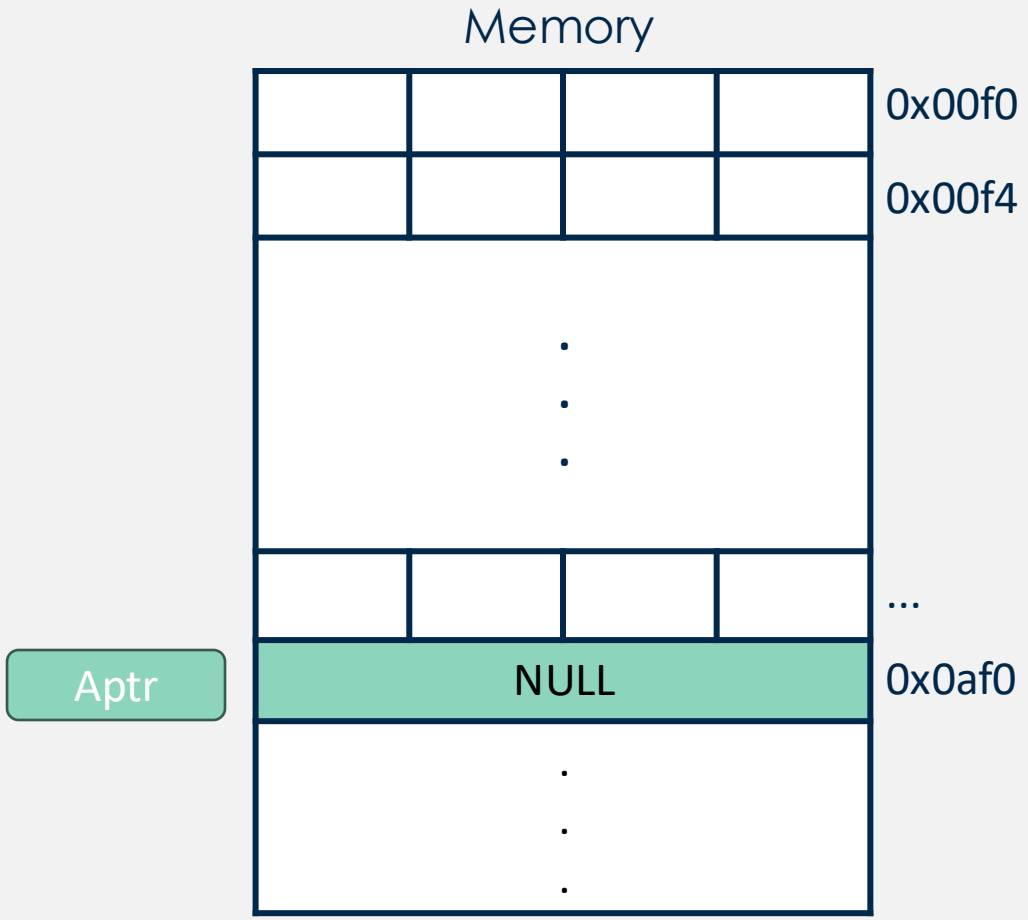
void * malloc (…)

```
char *Aptr = NULL;
Aptr = (char*) malloc(10 * sizeof (char));
```



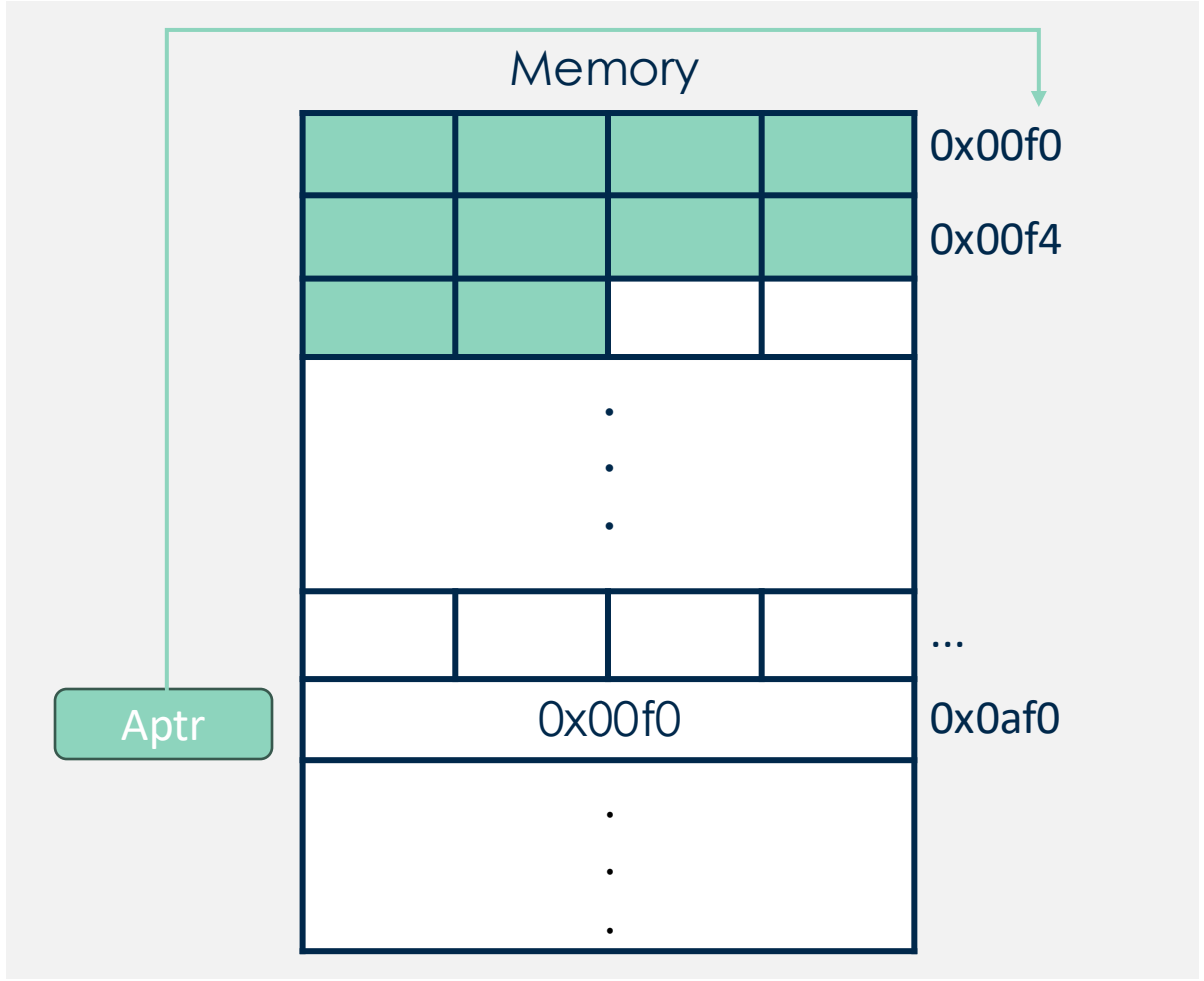
void * malloc (…)

```
char *Aptr = NULL;
Aptr = (char*) malloc(10 * sizeof (char));
```



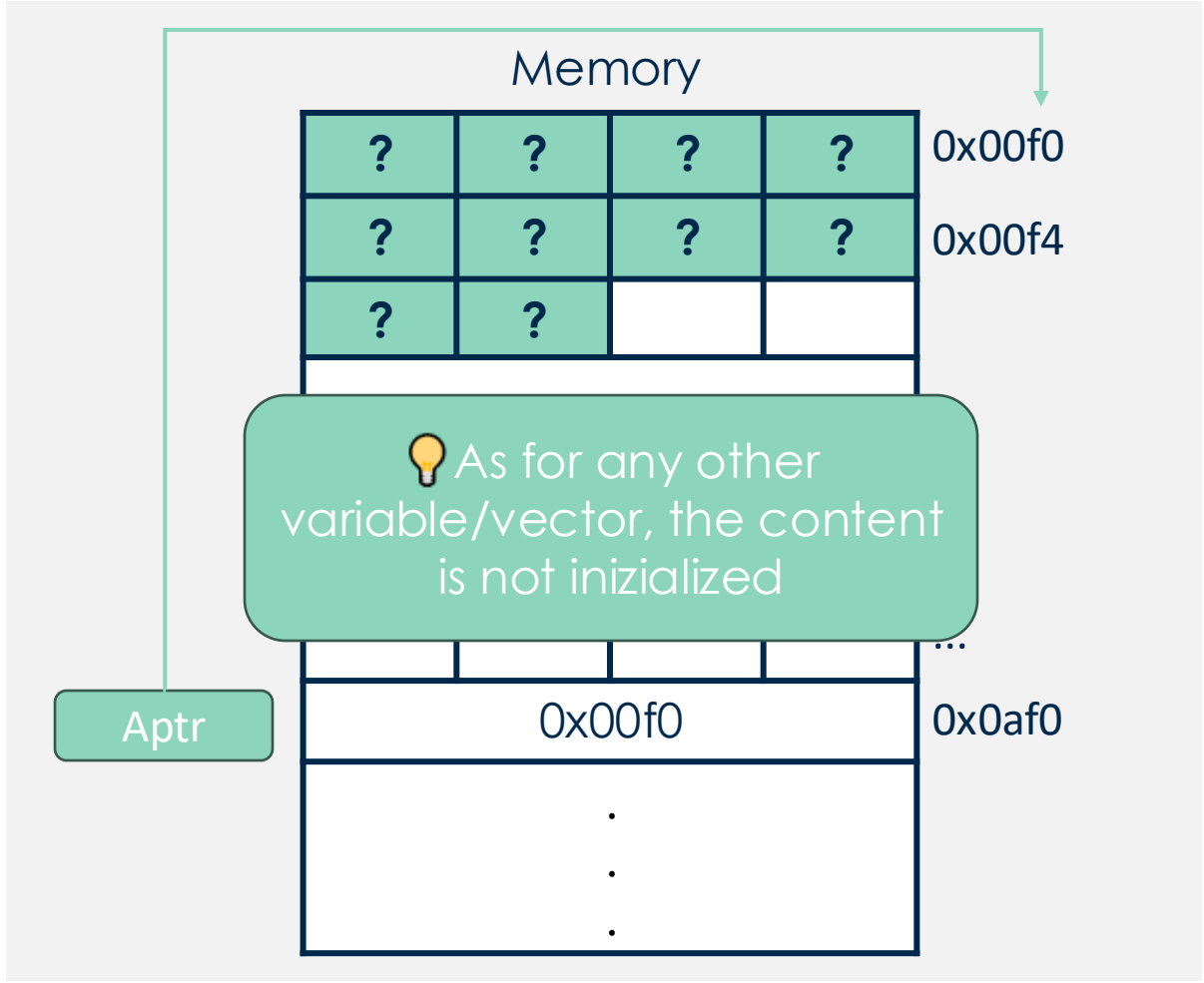
void * malloc (...)

```
char *Aptr = NULL;
Aptr = (char*) malloc(10 * sizeof (char));
```



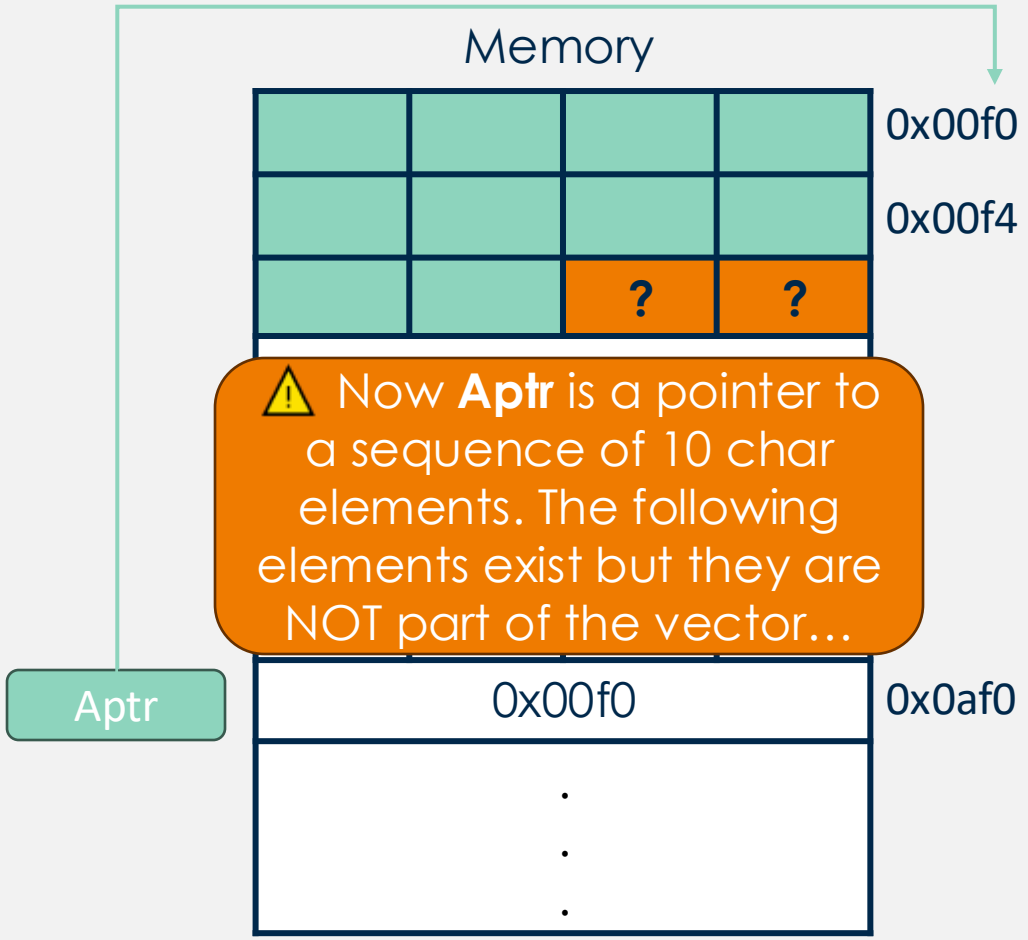
void * malloc (…)

```
char *Aptr = NULL;
Aptr = (char*) malloc(10 * sizeof (char));
```



void * malloc (…)

```
char *Aptr = NULL;
Aptr = (char*) malloc(10 * sizeof (char));
```



void * realloc (…)

In C, there are two main functions to deallocate memory dynamically:

- void* realloc (void* ptr, <memory_size>);

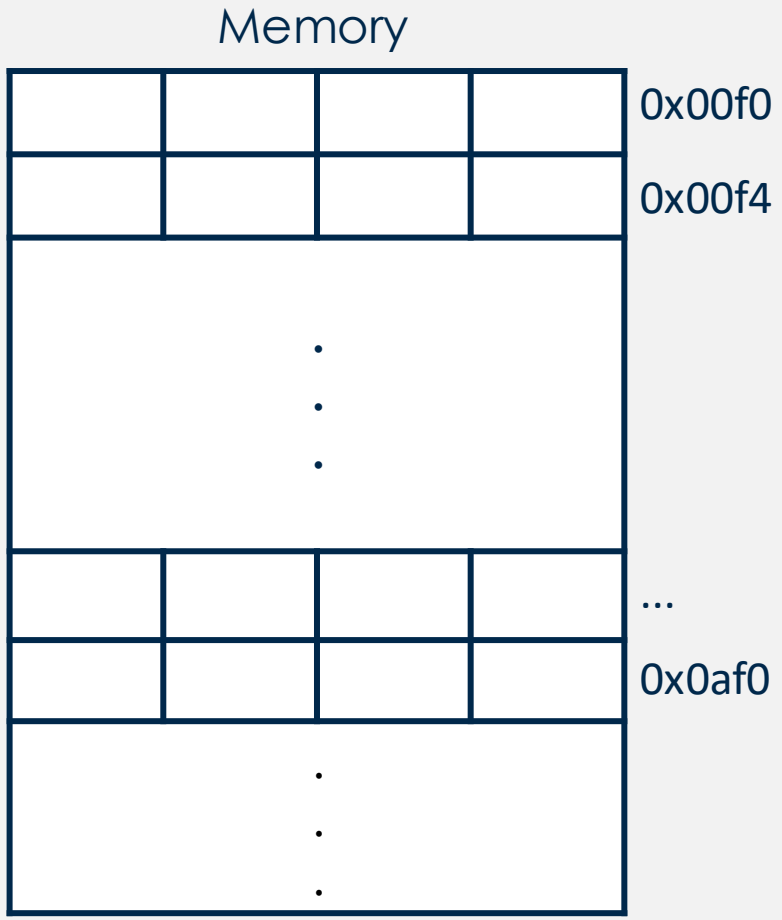
It asks the Operating System to resize, accordingly to `<memory_size>`, the memory portion pointed by `ptr`, and returns the pointer to the beginning of the reallocated space.

If ptr is NULL, the behavior is the same as calling malloc(new_size).

💡 due to a bug, on Windows, this behavior is not guaranteed

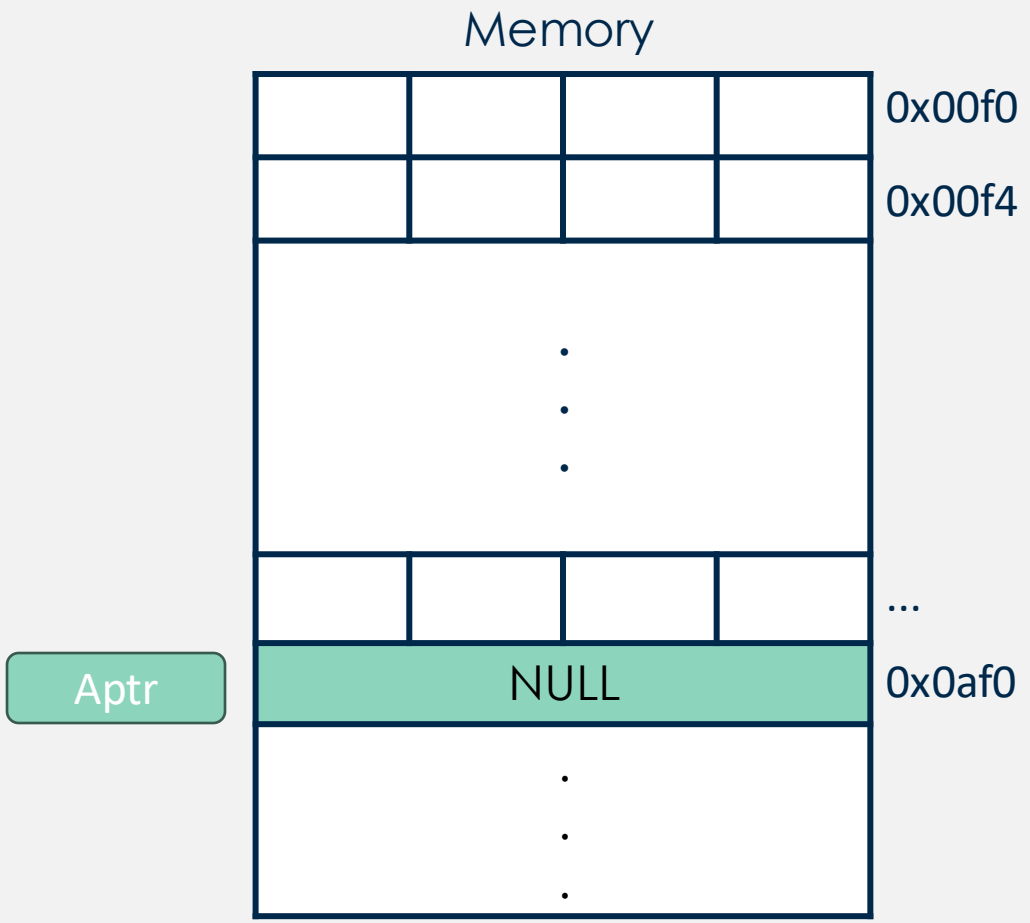
void * realloc (…)

```
char *Aptr = NULL;
Aptr = (char*) malloc(10 * sizeof (char));
Aptr = (char*) realloc(Aptr, 2* sizeof (char));
```



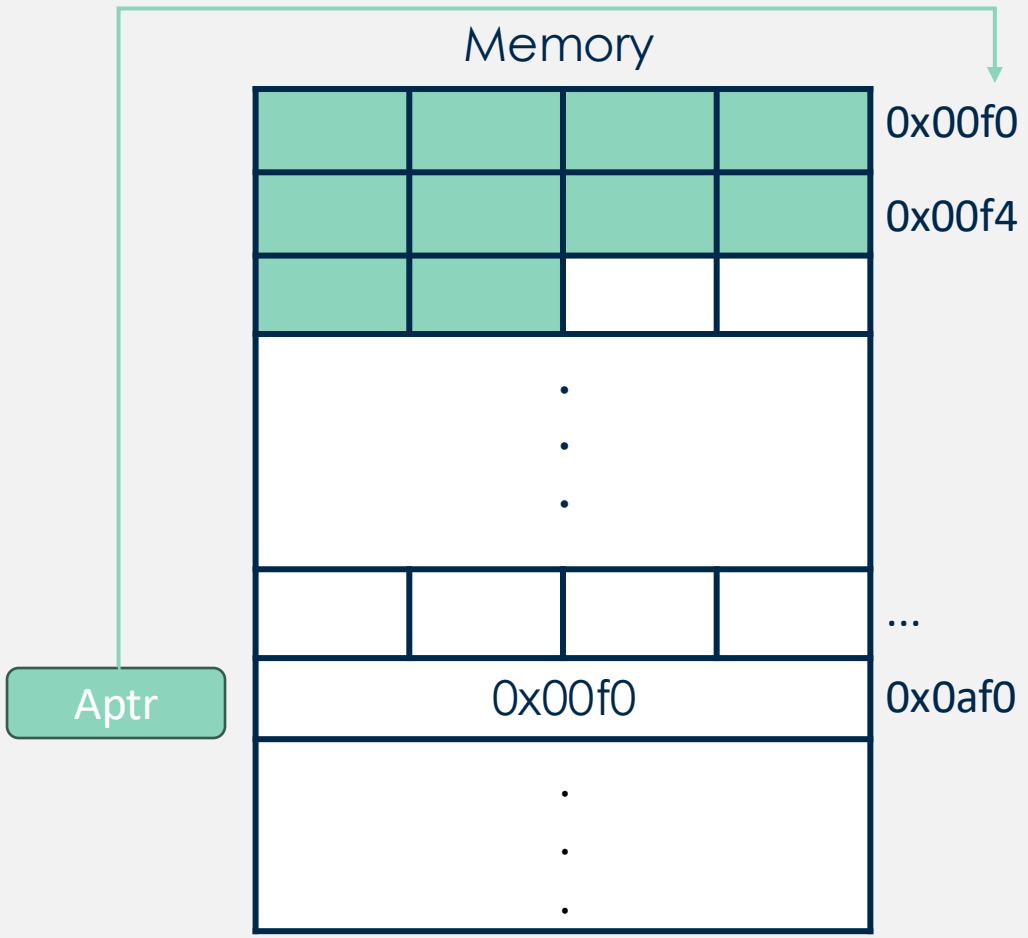
void * realloc (…)

```
char *Aptr = NULL;
Aptr = (char*) malloc(10 * sizeof (char));
Aptr = (char*) realloc(Aptr, 2* sizeof (char));
```



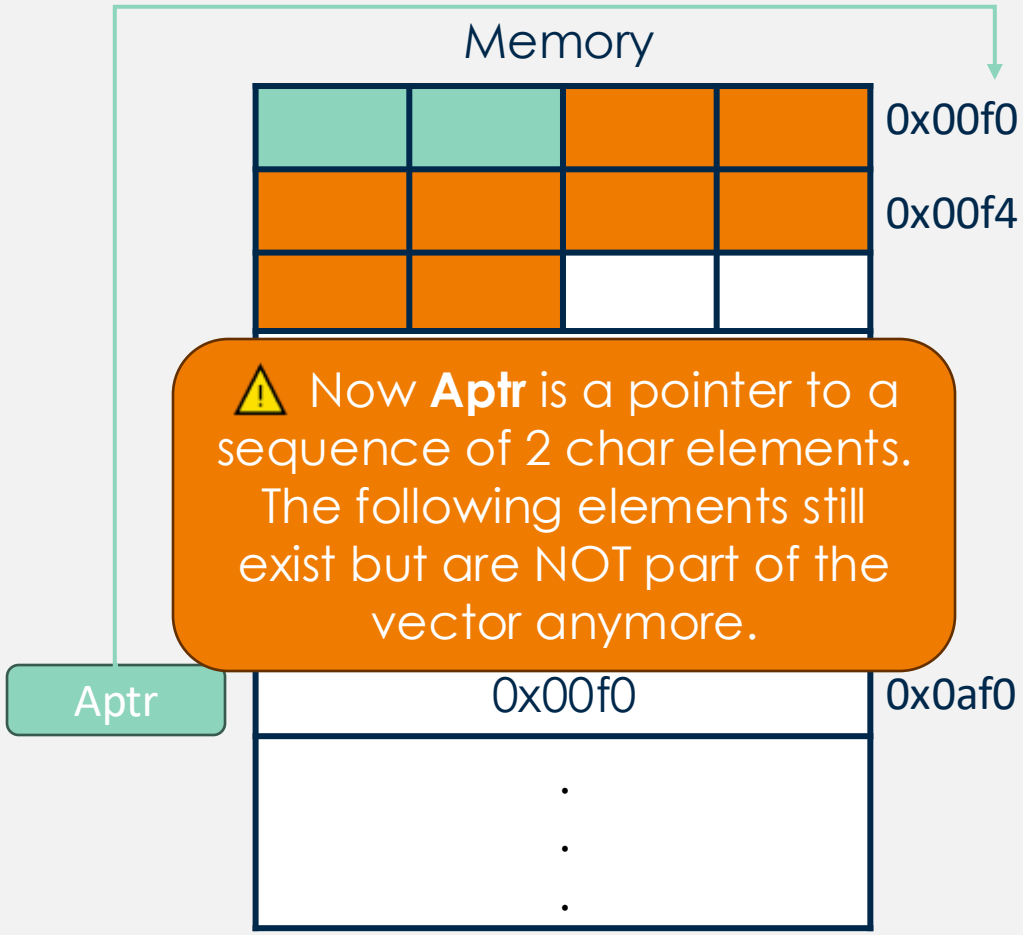
void * realloc (…)

```
char *Aptr = NULL;
Aptr = (char*) malloc(10 * sizeof (char));
Aptr = (char*) realloc(Aptr, 2* sizeof (char));
```



void * realloc (…)

```
char *Aptr = NULL;
Aptr = (char*) malloc(10 * sizeof (char));
Aptr = (char*) realloc(Aptr, 2* sizeof (char));
```



void * realloc (…)

In C, there are two main functions to deallocate memory dynamically:

- void* realloc (void* ptr, <memory_size>);

The reallocation is done by either:

1. Expanding or contracting the existing area pointed to by ptr.
    1. The area's contents remain unchanged up to the lesser of the new and old sizes. If the area is expanded, the contents of the new part of the array are undefined.
2. We are allocating a new memory block of size new_size bytes, copying the memory area with a size equal to the lesser of the new_size and the old dimensions, and freeing the old block.

⚠ If there is insufficient memory, the old memory block is not freed, and the NULL pointer is returned.

void free (…)

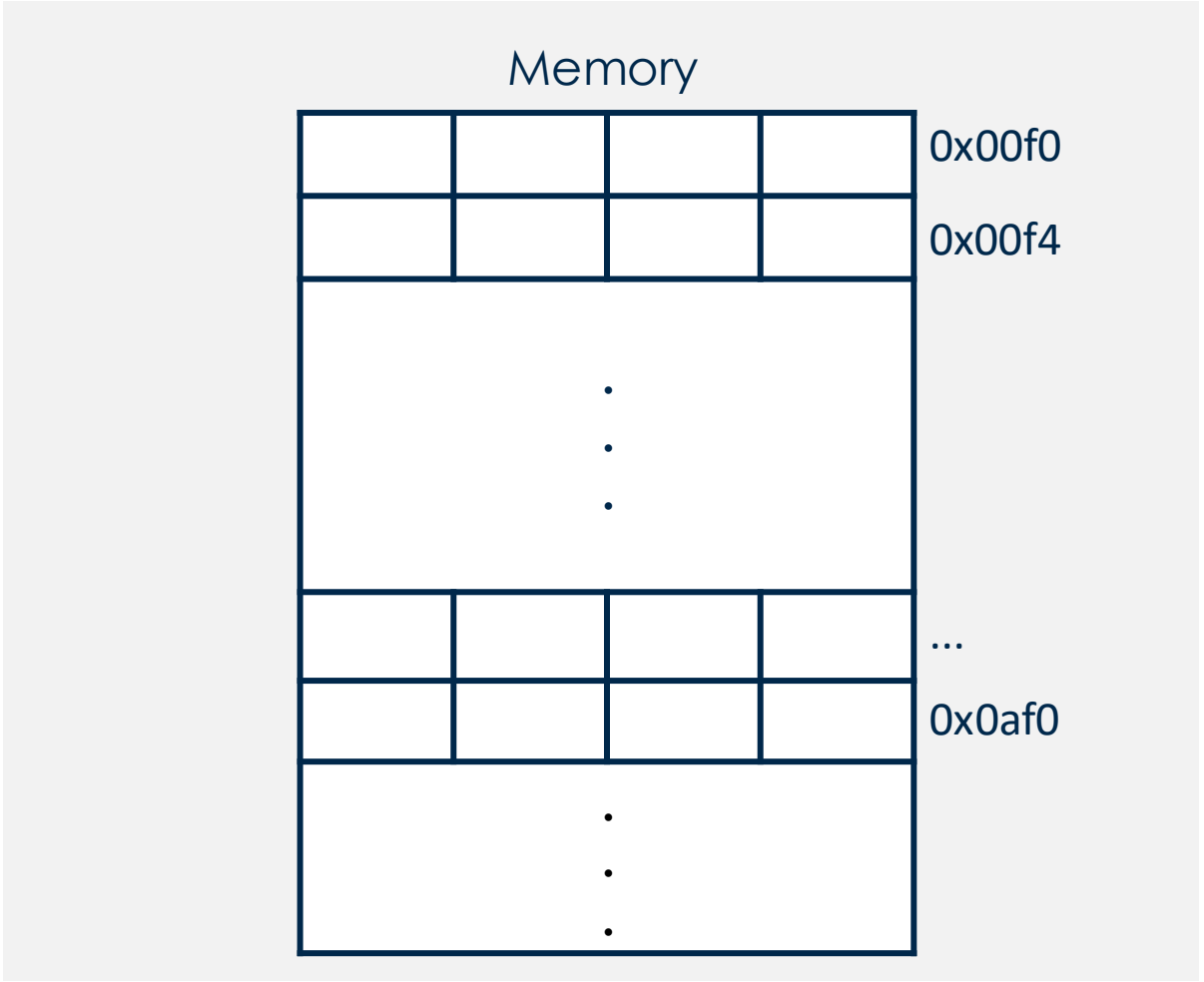
In C, there are two main functions to deallocate memory dynamically:

- void* realloc (void* ptr, <memory_size>);
- void free (void* ptr);

It asks the Operating System to deallocate the memory portion pointed by `ptr`.

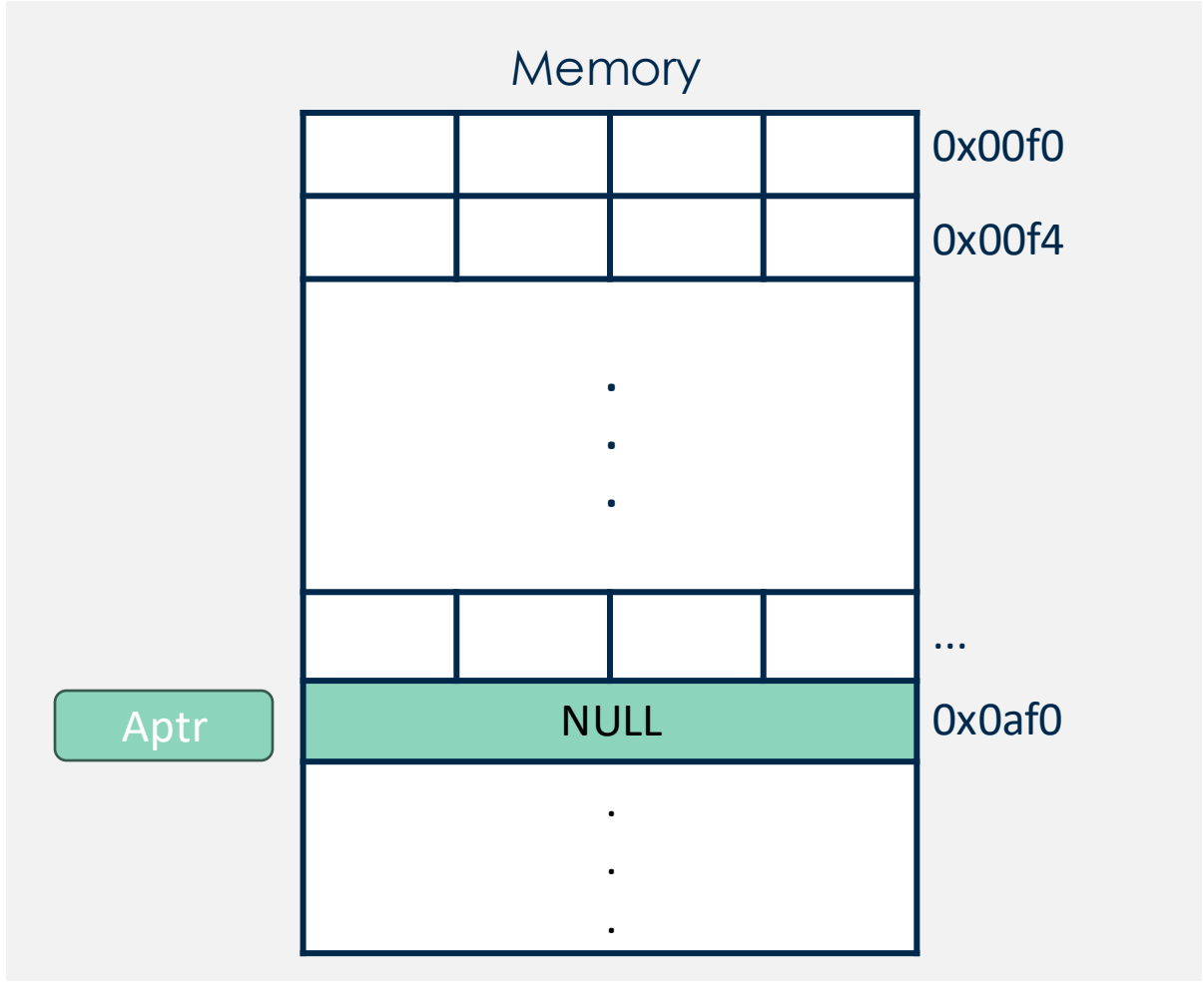
void free (…)

```
char *Aptr = NULL;
Aptr = (char*) malloc(10 * sizeof (char));
free(Aptr);
```



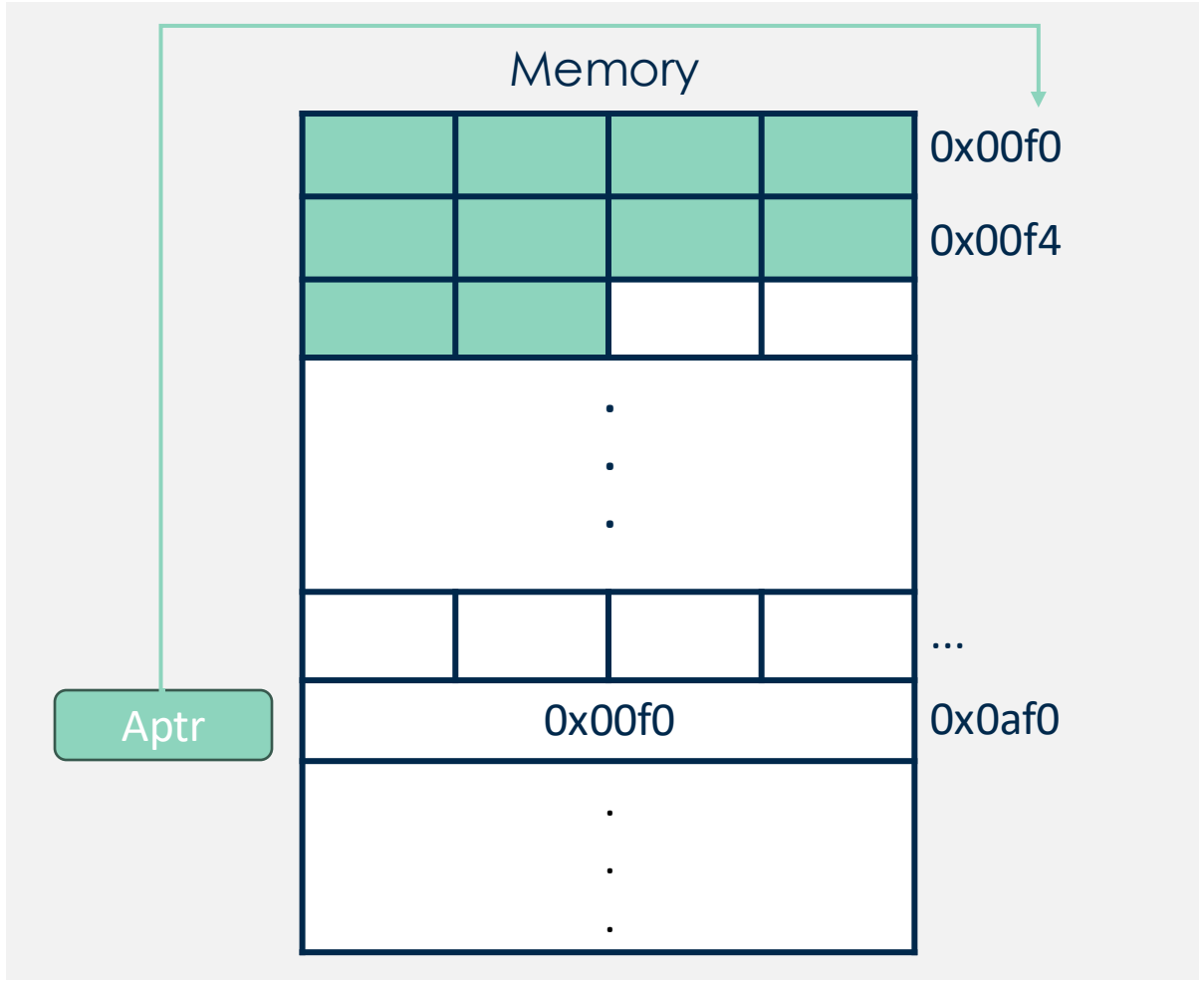
void free (…)

```
char *Aptr = NULL;
Aptr = (char*) malloc(10 * sizeof (char));
free(Aptr);
```



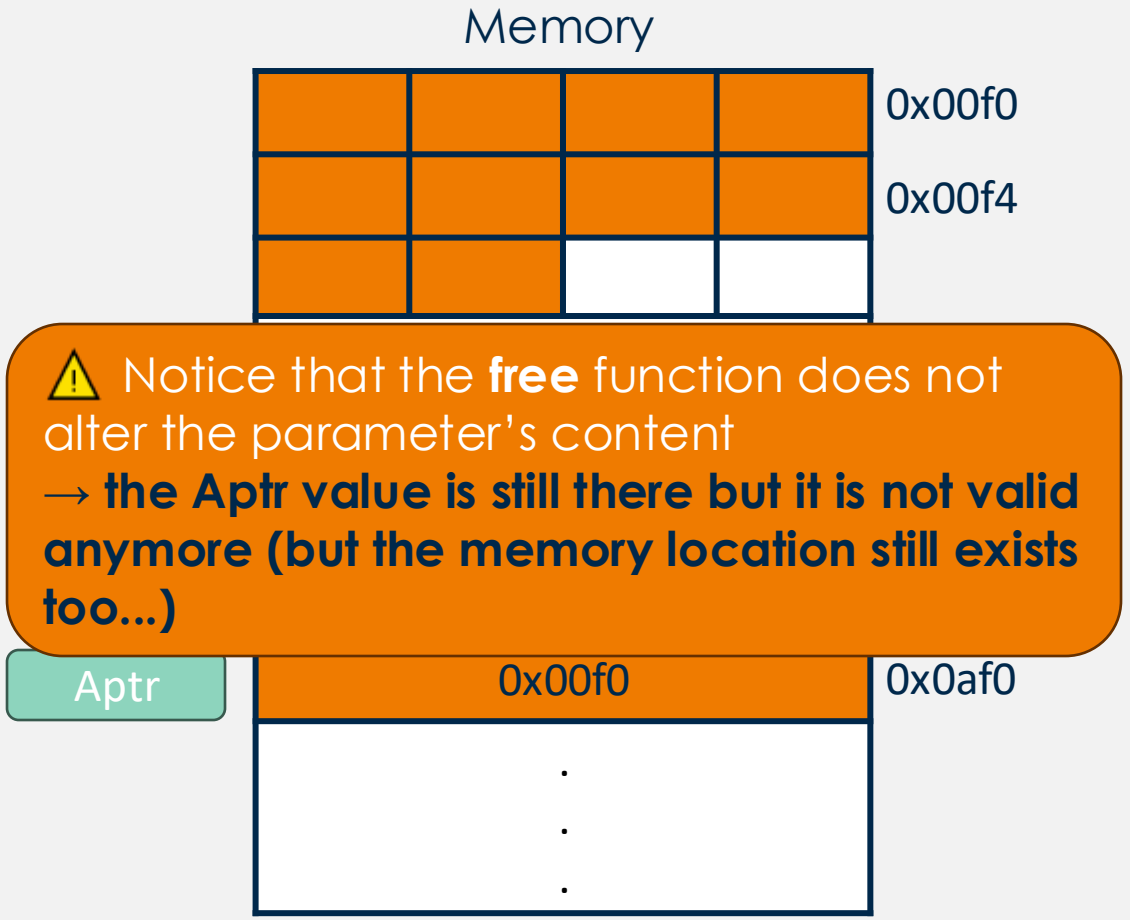
void free (…)

```
char *Aptr = NULL;
Aptr = (char*) malloc(10 * sizeof (char));
free (Aptr);
```



void free (…)

```
char *Aptr = NULL;
Aptr = (char*) malloc(10 * sizeof (char));
free (Aptr);
```



(Safe) Rules for Dynamic Memory

1. Everything that has been allocated must be freed:

```
int * v = (int*) malloc(10*sizeof(int));
...
free(v);
```

2. Everything that you want to free must be tested before:

```
if (v != NULL)
    free(v);
```

(Safe) Rules for Dynamic Memory

3. Everything that you would like to allocate should be initialized to NULL

```
int * v = NULL;
...
v = (int*) malloc(10*sizeof(int));
```

4. Everything that has been freed should be assigned to NULL.

```
if (v != NULL) {
        free(v);
        v = NULL;
}
```

Pointers and Arrays

Pointers and arrays are **almost** interchangeable in C

int A[10] ≅ int *APtr

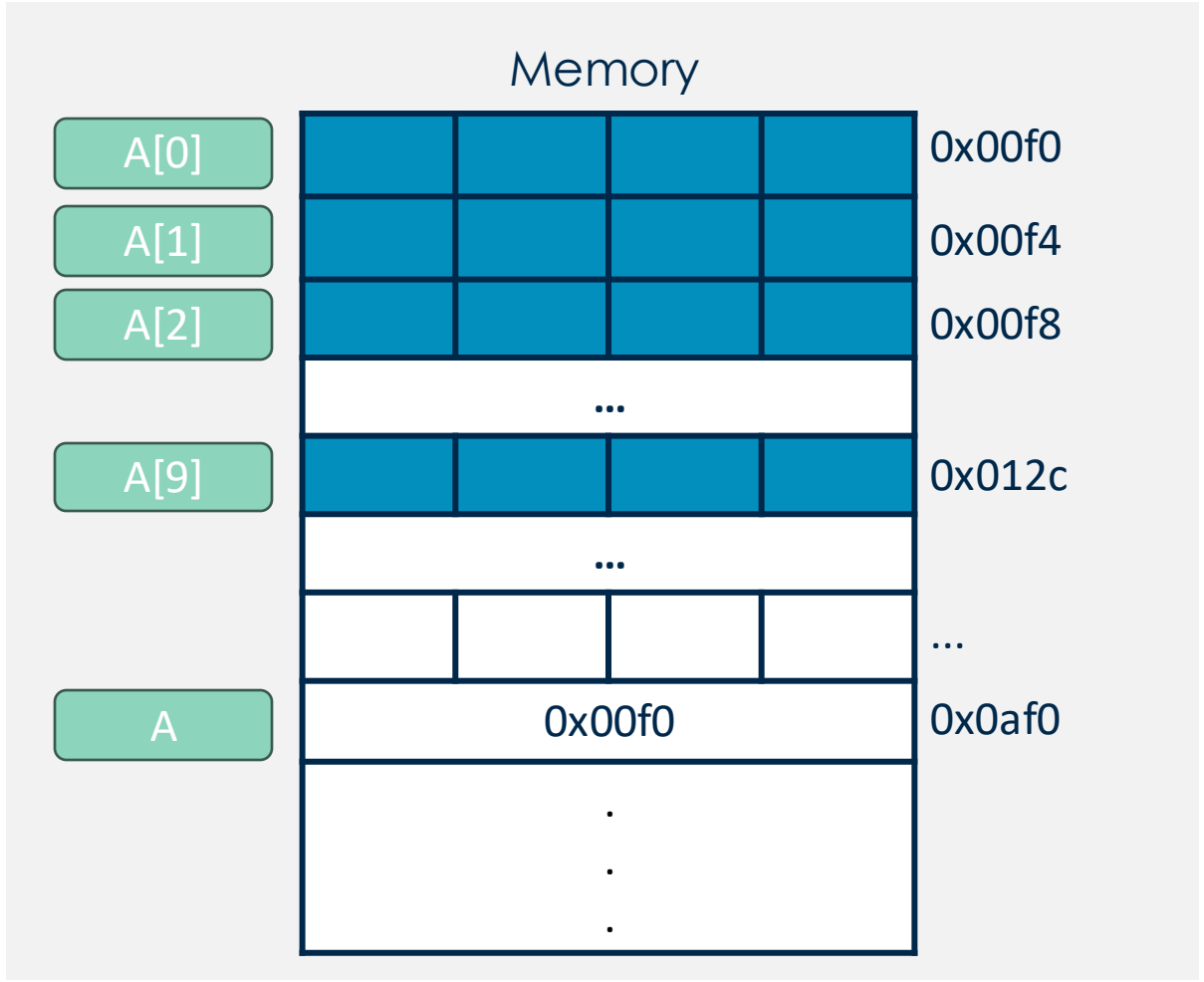
Why is it only **almost** interchangeable?

Pointers and Arrays - Differences

Main difference:

› When declaring an array, size is specified, and memory is allocated **statically**

```
int A[10];
```

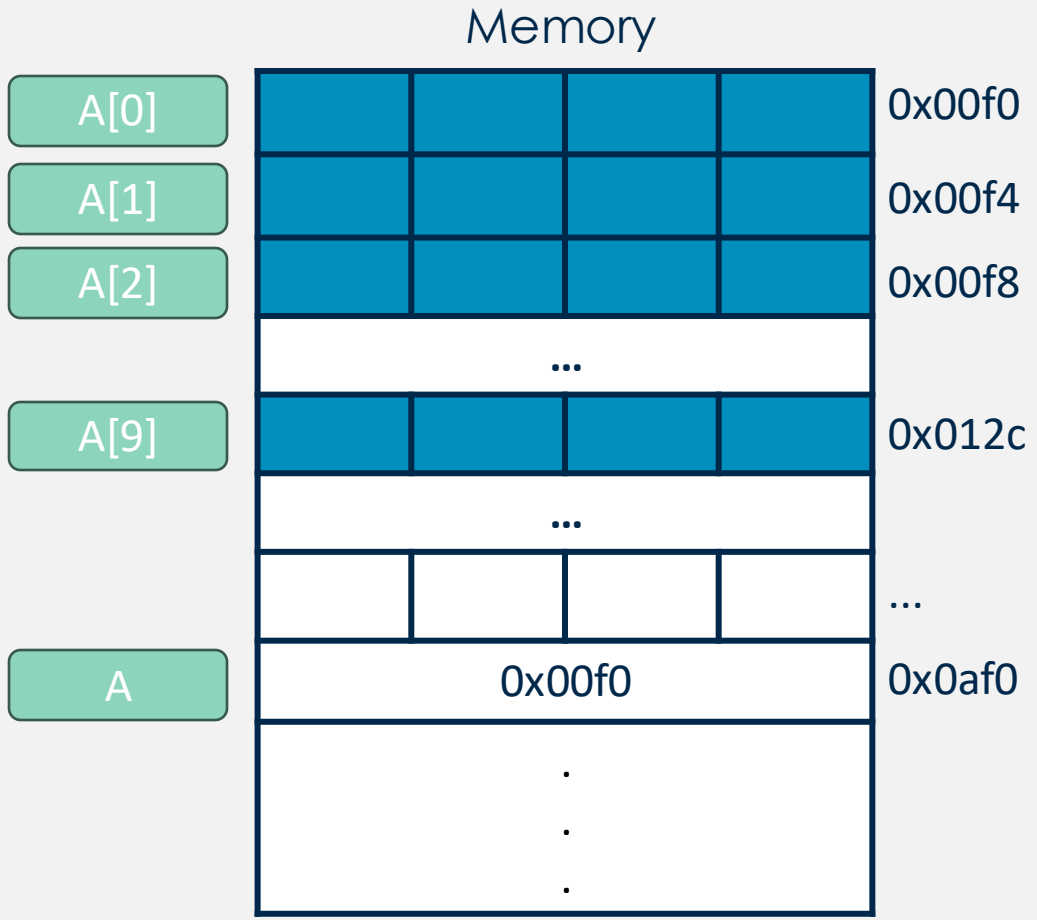


Pointers and Arrays - Differences

Main difference:

- When declaring an array, size is specified, and memory is allocated **statically**
- When declaring an array, the mathematics can be applied in read-only mode

```
int A[10];
int var = *(A+2); // OK
A = A+2 // NOT OK
```

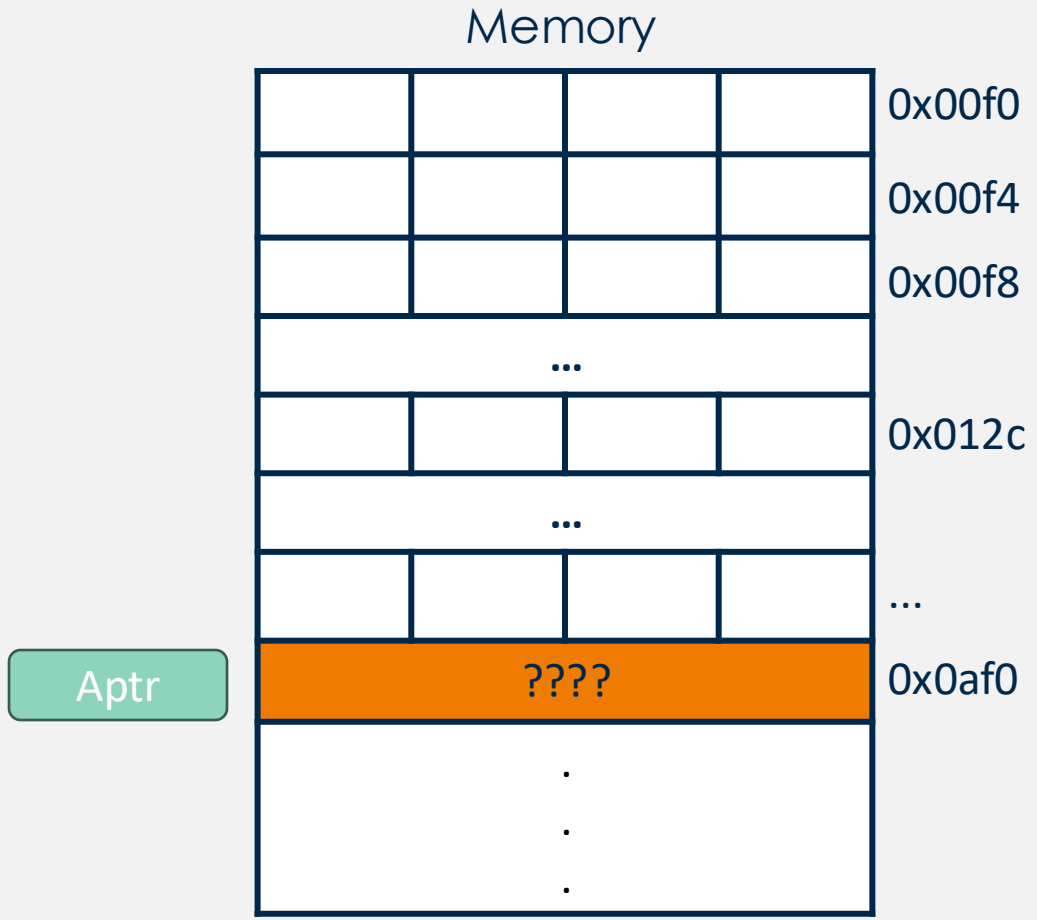


Pointers and Arrays - Differences

Main difference:

- When declaring an array, size is specified, and memory is allocated **statically**
- When declaring an array, the mathematics can be applied in read-only mode
- When declaring a pointer, **no additional** memory is initially allocated

```
int *APtr
```



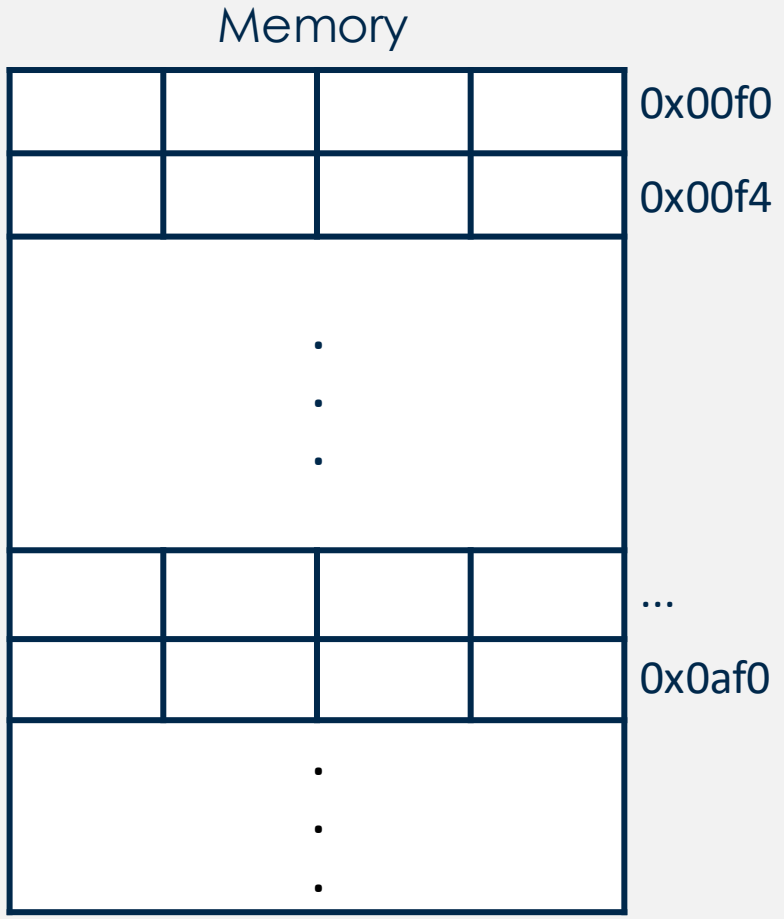
Pointers and Arrays

A pointer can be used to declare and scan an array exploiting:

- Dynamic memory allocation
- Pointers' arithmetic

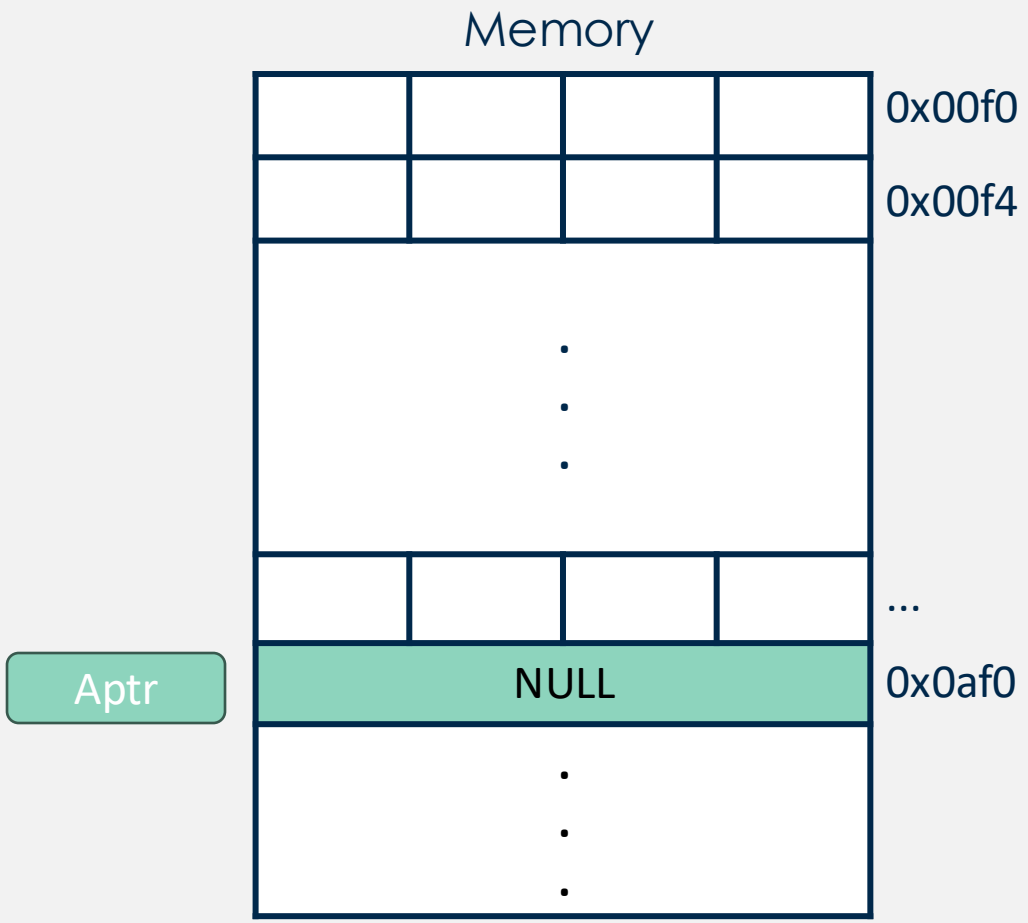
Scanning a Dynamic Array

```
char *Aptr = NULL;
 Aptr    = (char*) malloc(10 * sizeof(char));
 *Aptr   = 'A';
 Aptr   = Aptr + 5;
 *Aptr   = 'B';
 Aptr   = Aptr - 4;
 *Aptr   = 'C';
 *Aptr++ = 'D';
      .
      .
      .
```



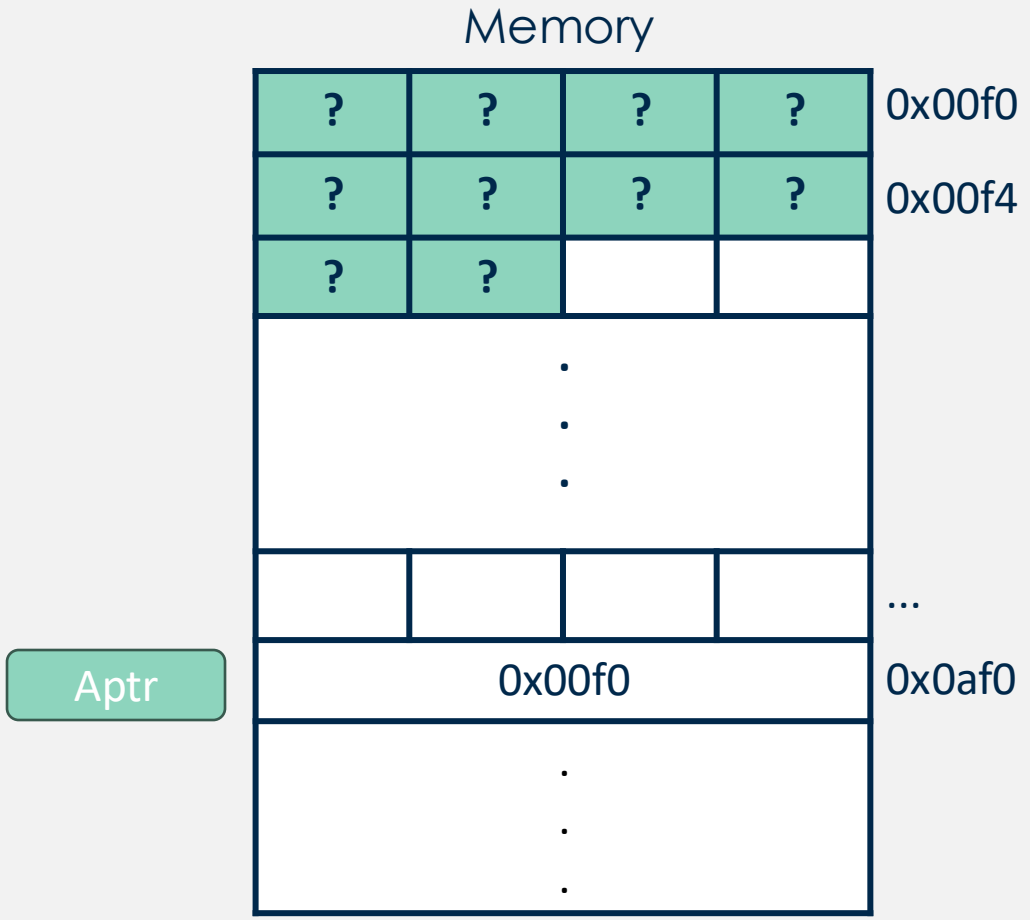
Scanning a Dynamic Array

```
char *Aptr = NULL;
Aptr    = (char*) malloc(10 * sizeof(char));
*Aptr   = 'A';
Aptr   = Aptr + 5;
*Aptr   = 'B';
Aptr   = Aptr - 4;
*Aptr   = 'C';
*Aptr++ = 'D';
      .
      .
      .
```



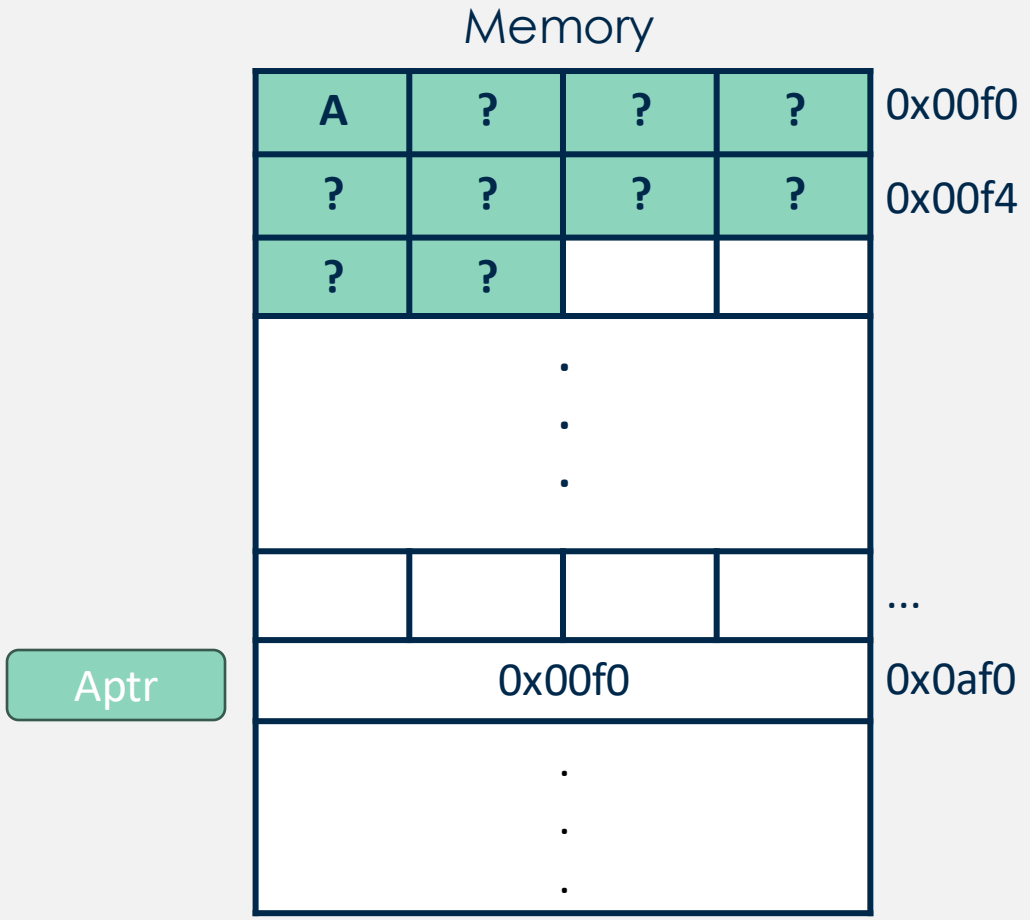
Scanning a Dynamic Array

```
char *Aptr = NULL;
Aptr    = (char*) malloc(10 * sizeof(char));
*Aptr   = 'A';
Aptr   = Aptr + 5;
*Aptr   = 'B';
Aptr   = Aptr - 4;
*Aptr   = 'C';
*Aptr++ = 'D';
      .
      .
      .
```



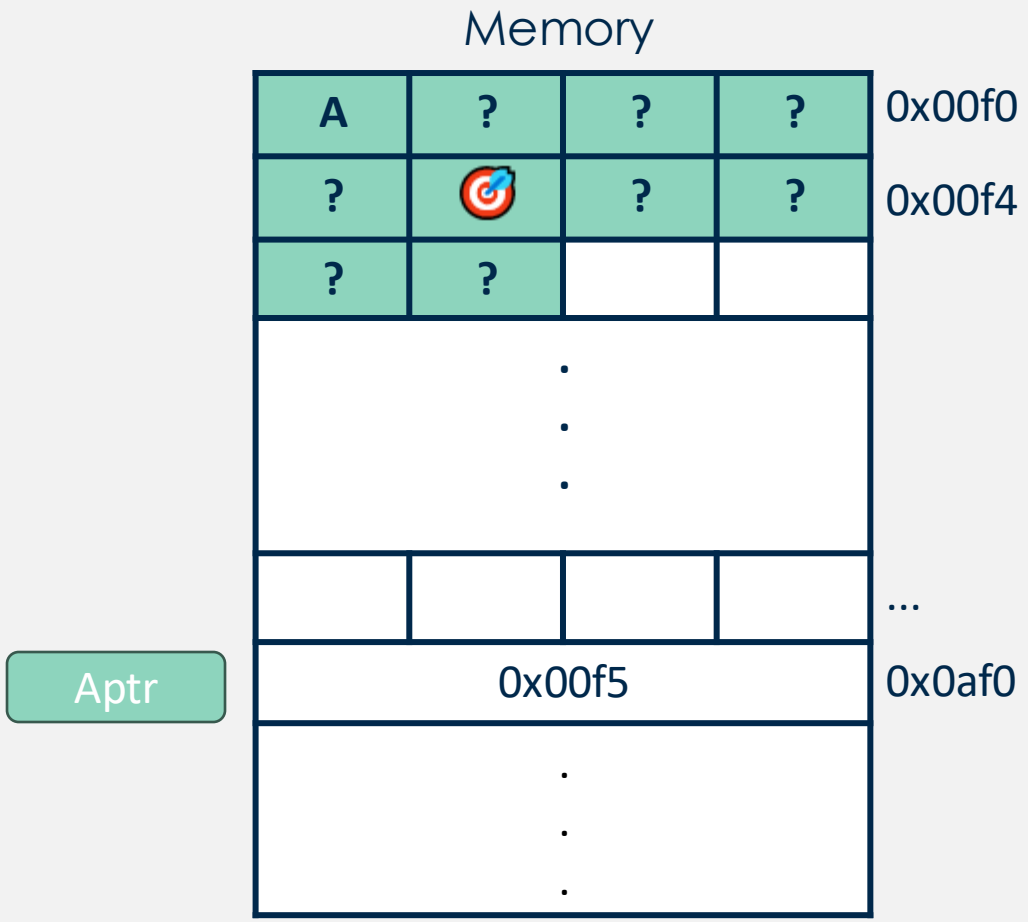
Scanning a Dynamic Array

```
char *Aptr = NULL;
 Aptr    = (char*) malloc(10 * sizeof(char));
*Aptr    = 'A';
 Aptr    = Aptr + 5;
 *Aptr   = 'B';
 Aptr    = Aptr - 4;
 *Aptr   = 'C';
 *Aptr++ = 'D';
      .
      .
      .
```



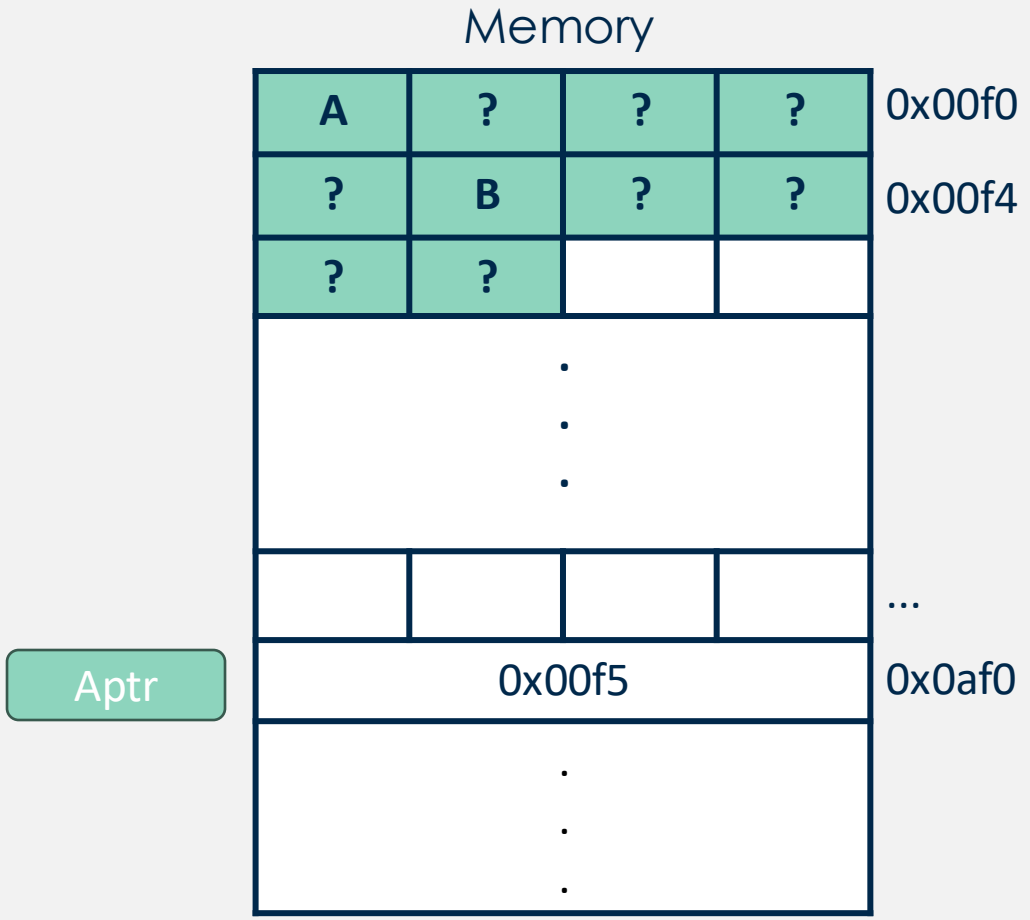
Scanning a Dynamic Array

```
char *Aptr = NULL;
 Aptr    = (char*) malloc(10 * sizeof(char));
 *Aptr   = 'A';
 Aptr    = Aptr + 5;
 *Aptr   = 'B';
 Aptr    = Aptr - 4;
 *Aptr   = 'C';
 *Aptr++ = 'D';
       .
       .
       .
```



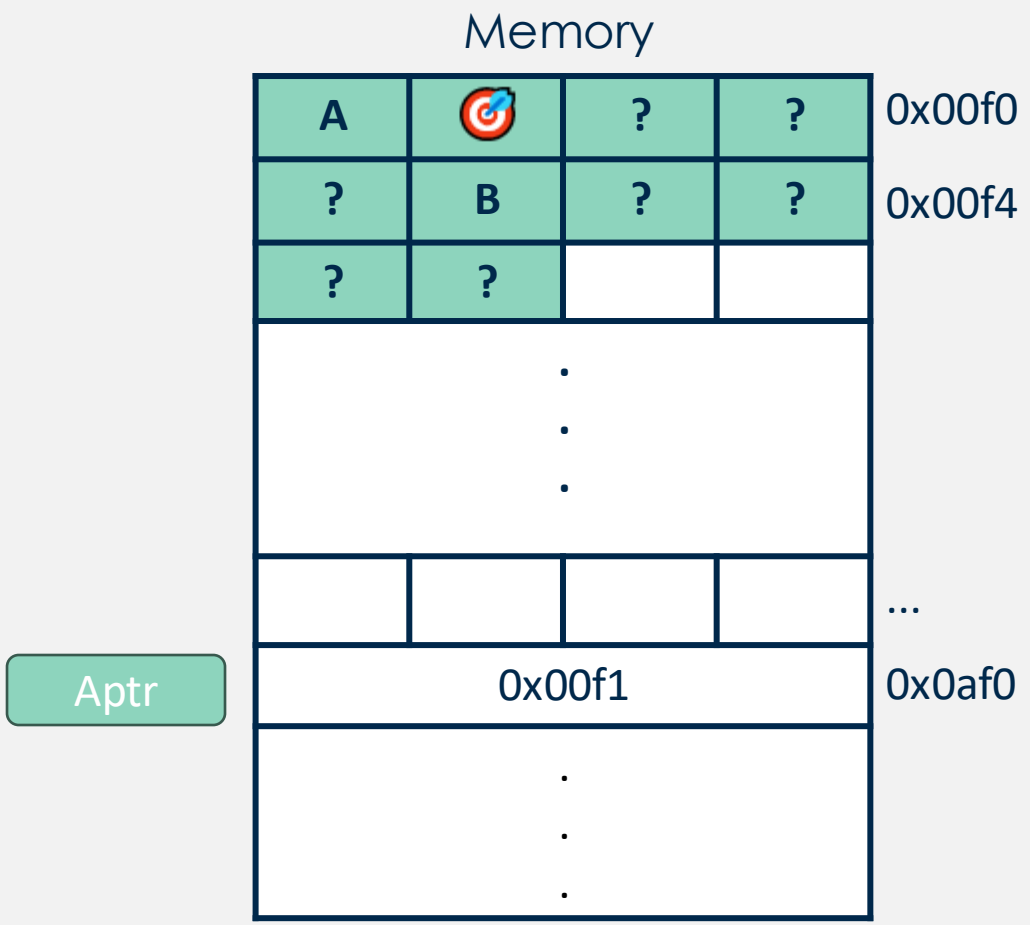
Scanning a Dynamic Array

```
char *Aptr = NULL;
 Aptr    = (char*) malloc(10 * sizeof(char));
 *Aptr   = 'A';
 Aptr   = Aptr + 5;
 *Aptr   = 'B';
 Aptr   = Aptr - 4;
 *Aptr   = 'C';
 *Aptr++ = 'D';
      .
      .
      .
```



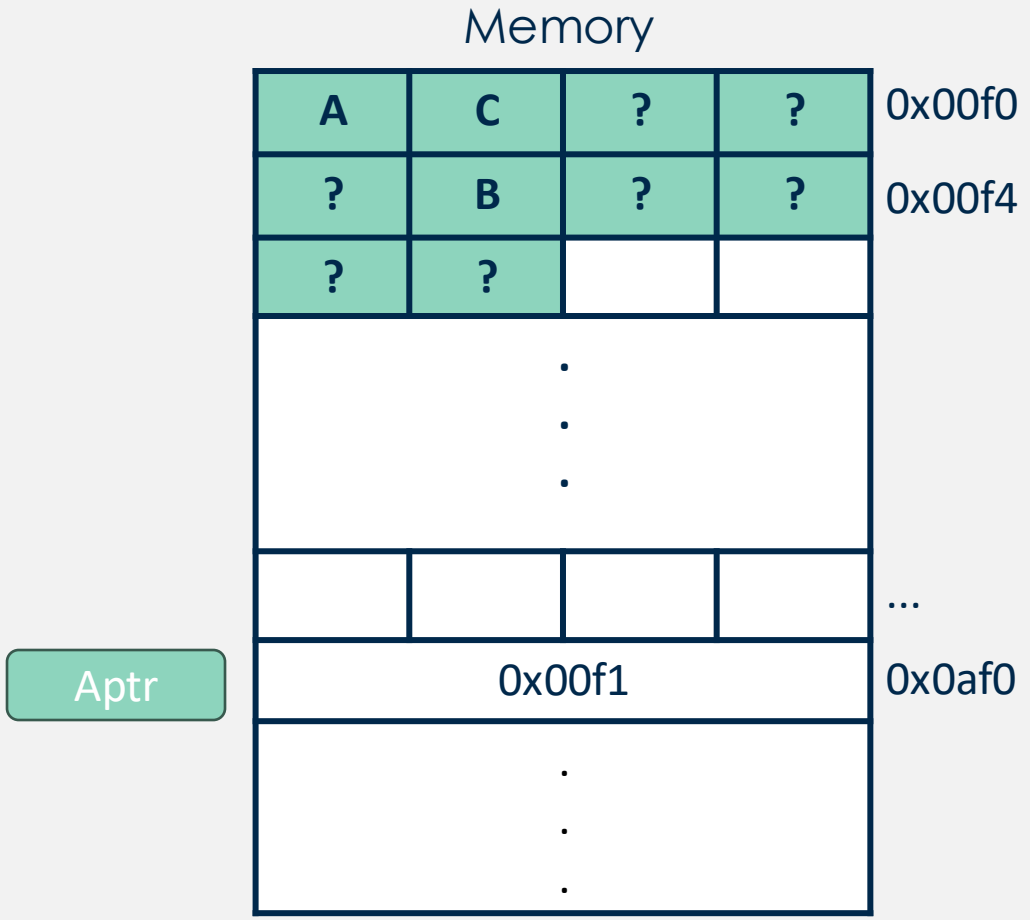
Scanning a Dynamic Array

```
char *Aptr = NULL;
 Aptr    = (char*) malloc(10 * sizeof(char));
 *Aptr   = 'A';
 Aptr   = Aptr + 5;
 *Aptr   = 'B';
 Aptr   = Aptr - 4;
 *Aptr   = 'C';
 *Aptr++ = 'D';
       .
       .
       .
```



Scanning a Dynamic Array

```
char *Aptr = NULL;
 Aptr    = (char*) malloc(10 * sizeof(char));
 *Aptr   = 'A';
 Aptr   = Aptr + 5;
 *Aptr   = 'B';
 Aptr   = Aptr - 4;
 *Aptr   = 'C';
 *Aptr++ = 'D';
       .
       .
       .
```

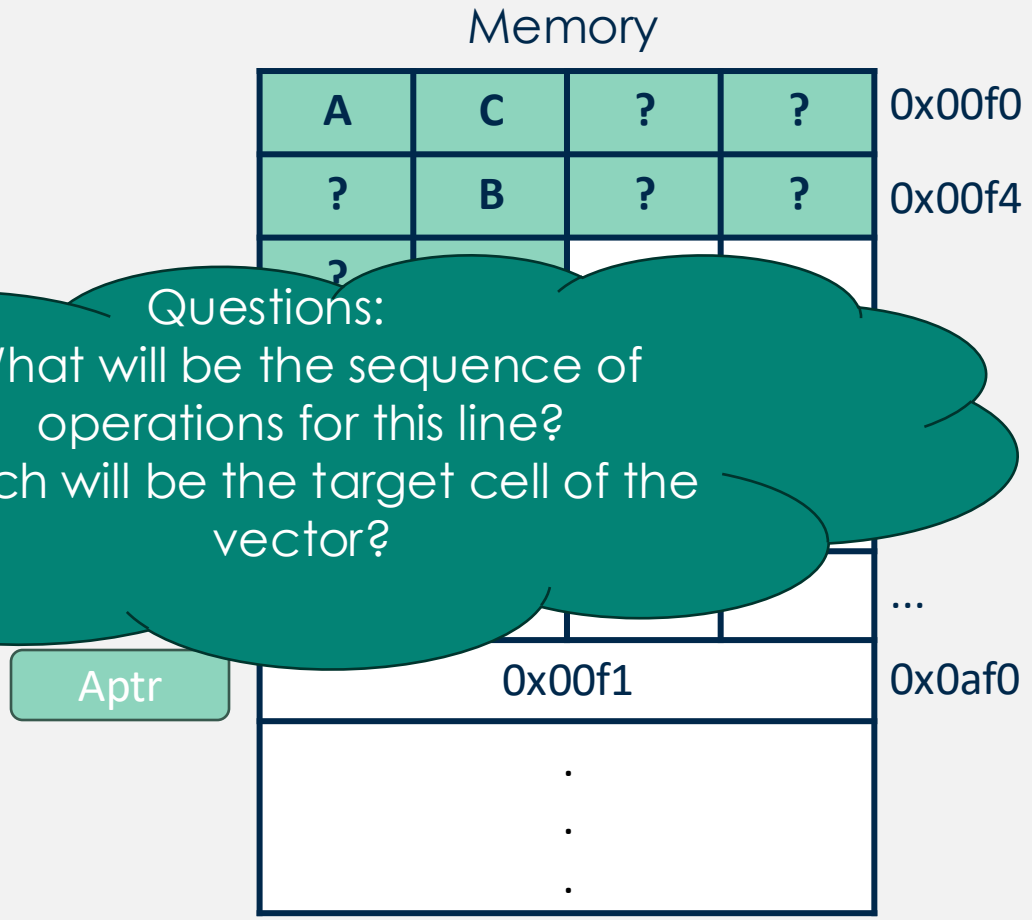


Scanning a Dynamic Array

```
char *Aptr = NULL;
 Aptr    = (char*) malloc(10 * sizeof(char));
 *Aptr   = 'A';
 Aptr    = Aptr + 5;
 *Aptr   = 'B';
 Aptr    = Aptr - 4;
 *Aptr   = 'C';
*Aptr++ = 'D';
      .
      .
      .
```

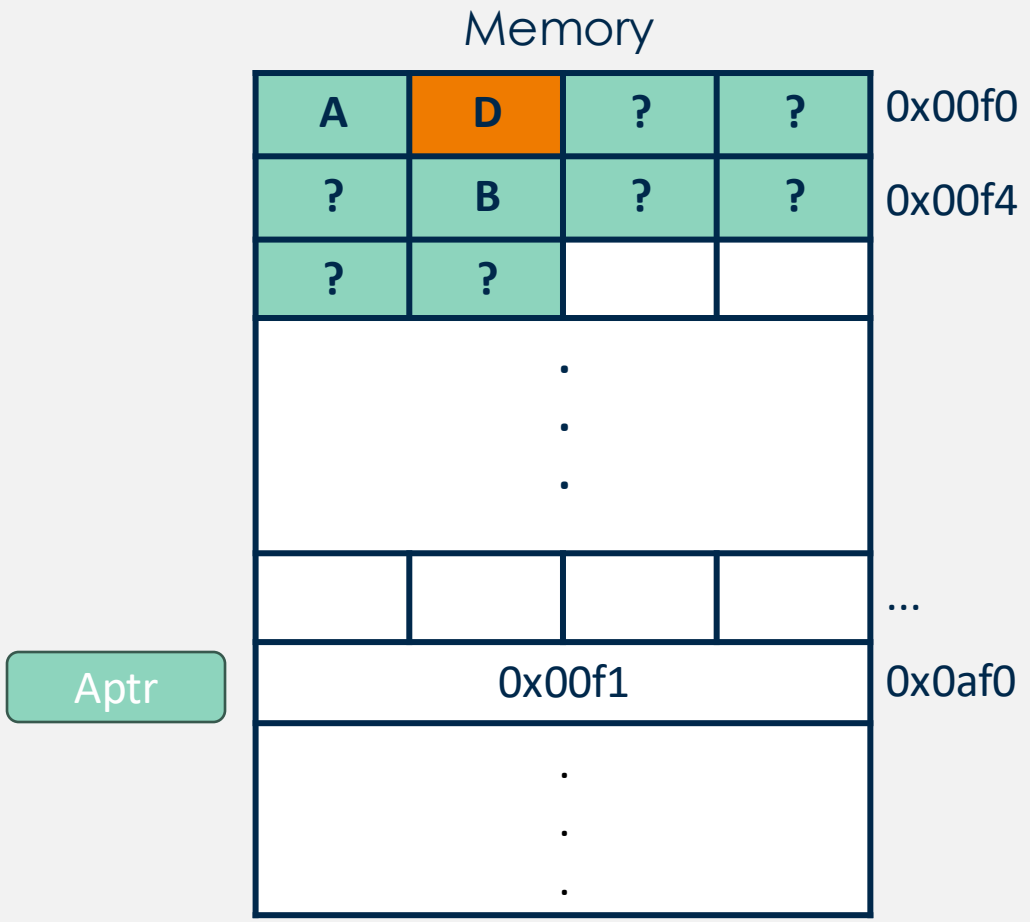
Questions:

1. What will be the sequence of operations for this line?
2. Which will be the target cell of the vector?



Scanning a Dynamic Array

```
char *Aptr = NULL;
 Aptr    = (char*) malloc(10 * sizeof(char));
 *Aptr   = 'A';
 Aptr   = Aptr + 5;
 *Aptr   = 'B';
 Aptr   = Aptr - 4;
 *Aptr   = 'C';
*Aptr++ = 'D';
      .
      .
      .
```



Scanning a Dynamic Array

```
char *Aptr = NULL;
 Aptr    = (char*) malloc(10 * sizeof(char));
 *Aptr   = 'A';
 Aptr   = Aptr + 5;
 *Aptr   = 'B';
 Aptr   = Aptr - 4;
 *Aptr   = 'C';
*Aptr++ = 'D';
      .
      .
      .
```