BEFORE STARTING

Both versions correctly compute the sum of two integer

What can be the advantage / disadvantages of V2 ?

```c
int sum(int a, int b){
  int c = a + b;
  return c;
}

int main() {
  int result;
  result = sum(20, 25);

  return 0;
}
```

V1

```c
void sum(int a, int b, int* result) {
  *(result) = a + b;

}


int main() {
  int result;
  sum(20, 25, &result);

  return 0;
}
```

V2

ANSWER

Both versions correctly compute the sum of two integer

What can be the advantage / disadvantages of V2 ?

Advantages of V2:

- ✓ Can return multiple values using multiple pointer parameters.
- ✓ Avoids creating and copying temporary return values (potentially beneficial in large structures or frequent calls).

Disadvantages of V2:

- ✓ Increases the risk of bugs (e.g., null or uninitialized pointers).
- ✓ Harder to read and use correctly for beginners.
- ✓ Requires more care in memory management and testing.

ALGORITHM AND COMPUTATIONAL THINKING 2

WEEK 9 – Dynamic Memory Allocation

Session objectives

- ✓ Distinguish between **stack** and **heap memory** allocation.
- ✓ Allocate **memory dynamically** using **malloc**(), **calloc**(), and **realloc**().
- ✓ Release memory safely using **free**().
- ✓ Avoid common **memory management errors**

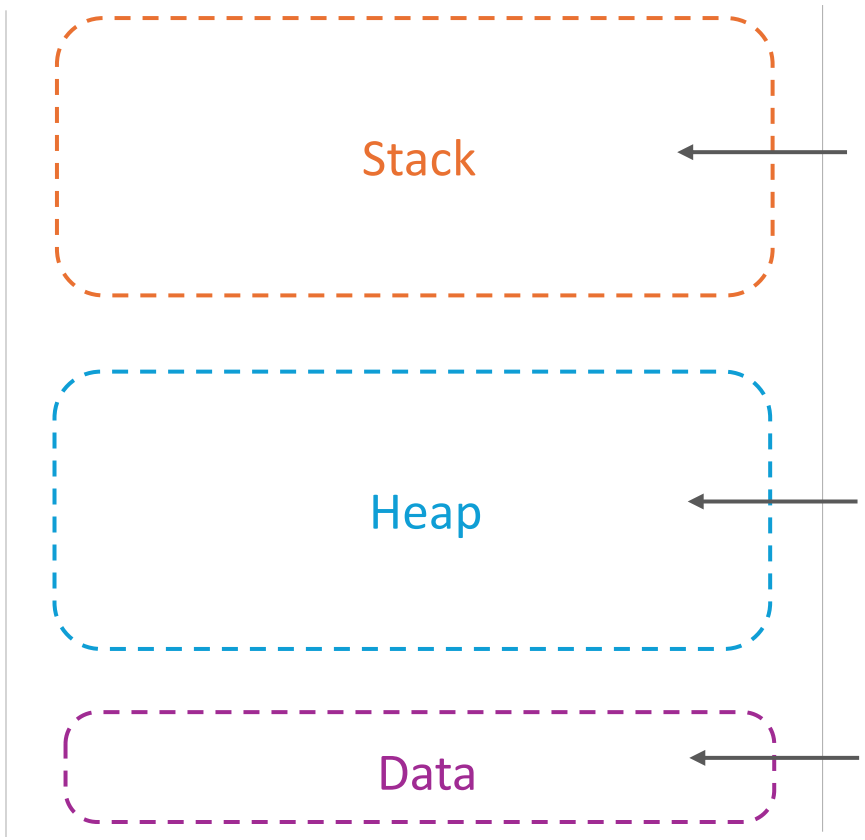
  (e.g., leaks, dangling pointers).

EXPLAIN

Overview of the **Memory Segments**

SEE MORE

A C program's memory is organized into specific regions (segments)



Stores **local variables** and **function** calls

Handle the **Dynamic Memory**
(what we will see today)

Stores **global variable**

Simplified vision of Segments in C Program's Memory

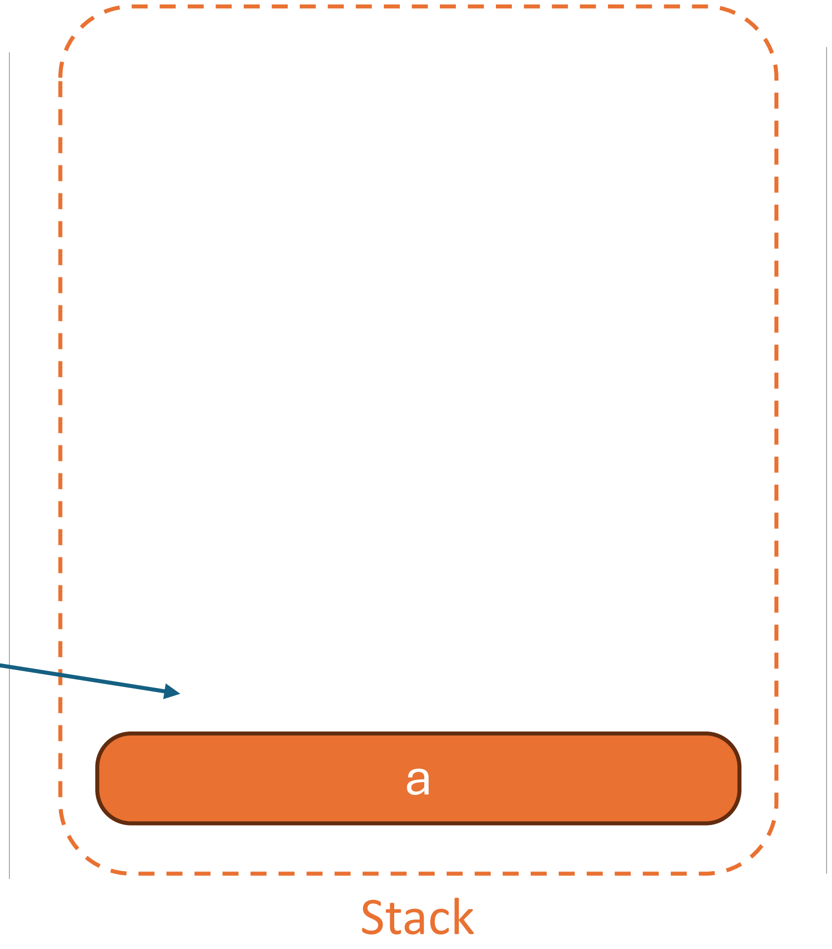
EXPLAIN

The **Stack** Segment

When we declare a variable inside a function…

```
int main() {
    int a =5;

    return 0;
}
```

.. The memory for that variable is statically allocated on the stack.

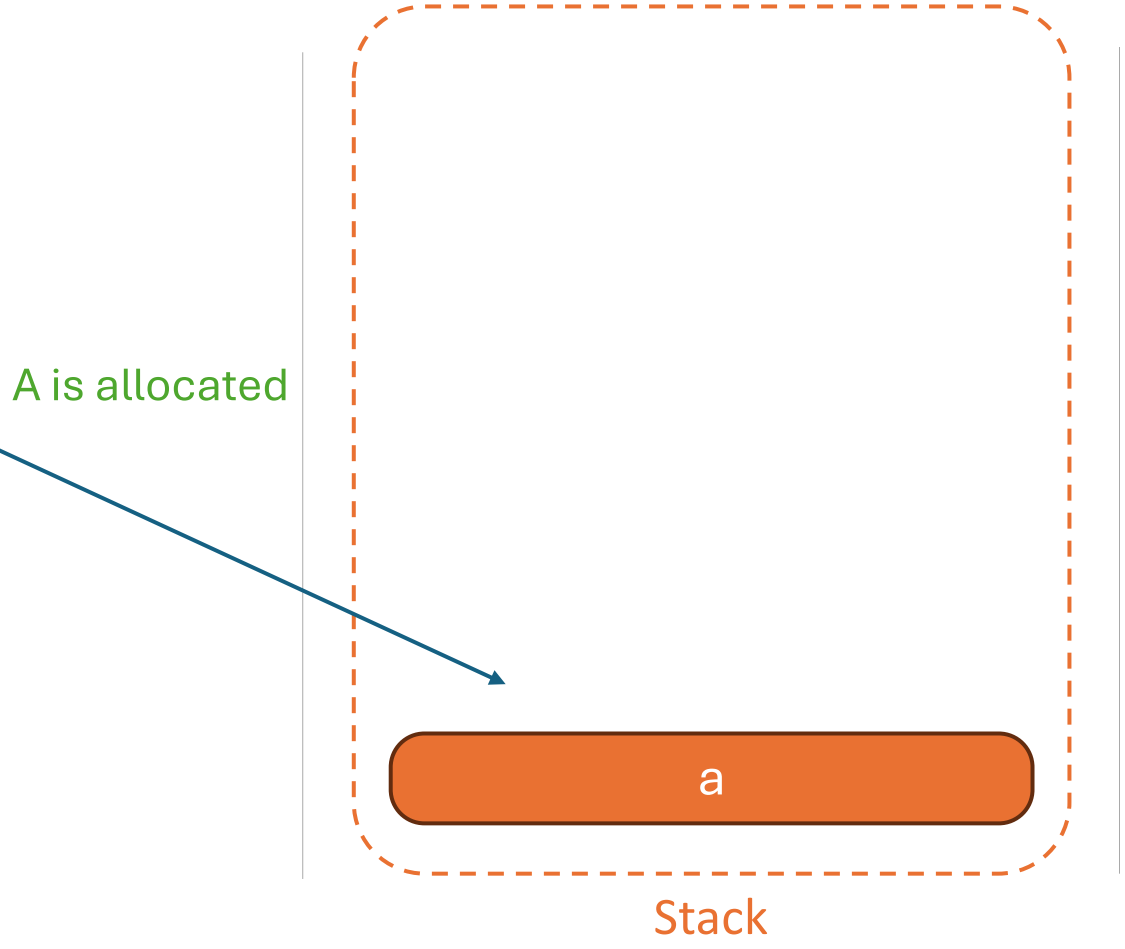


EXPLAIN

The **Stack** Segment

The stack segment stores local data with **automatic allocation** and **deallocation**

```c
int main() {
    int a = 10;

    if (a > 5) {
        int b = 20;
        printf("%d\n", b);
    }

    int c = 30;

   return 0;
}
```



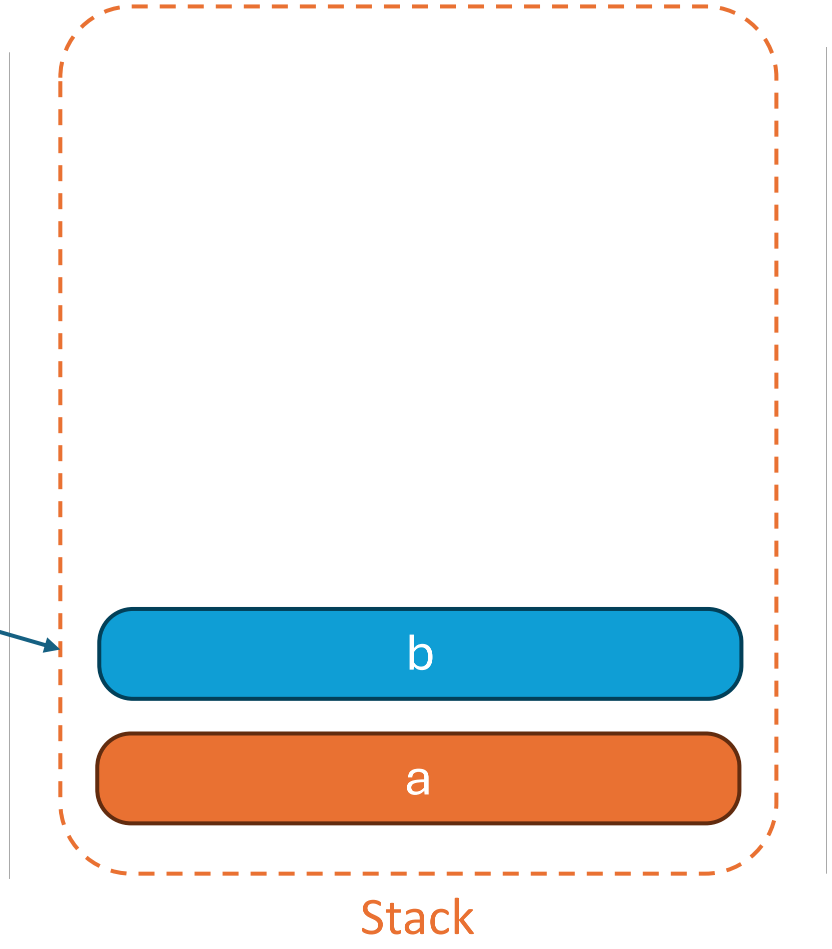
EXPLAIN

The **Stack** Segment

The stack segment stores local data with **automatic allocation** and **deallocation**

```
int main() {
    int a = 10;

    if (a > 5) {
        int b = 20;
        printf("%d\n", b);
    }

    int c = 30;

   return 0;
}
```

B is allocated



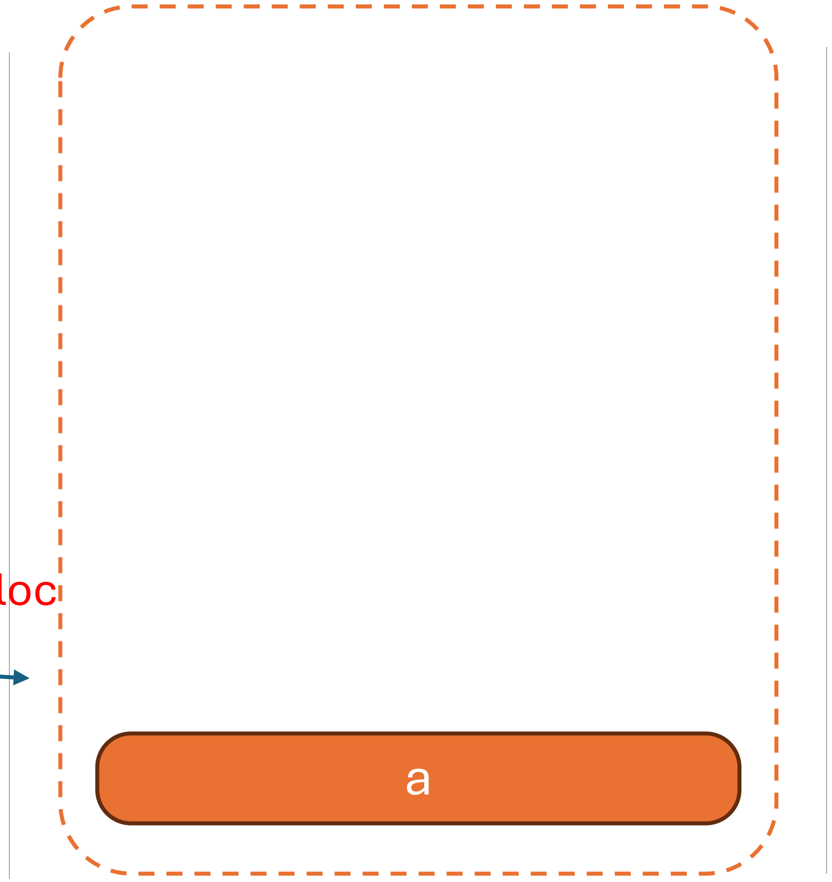
EXPLAIN

The **Stack** Segment

The stack segment stores local data with **automatic allocation** and **deallocation**

```c
int main() {
    int a = 10;

    if (a > 5) {
        int b = 20;
        printf("%d\n", b);
    }

    int c = 30;

  return 0;
}
```

B is deallocated
As we quit the bloc



Stack

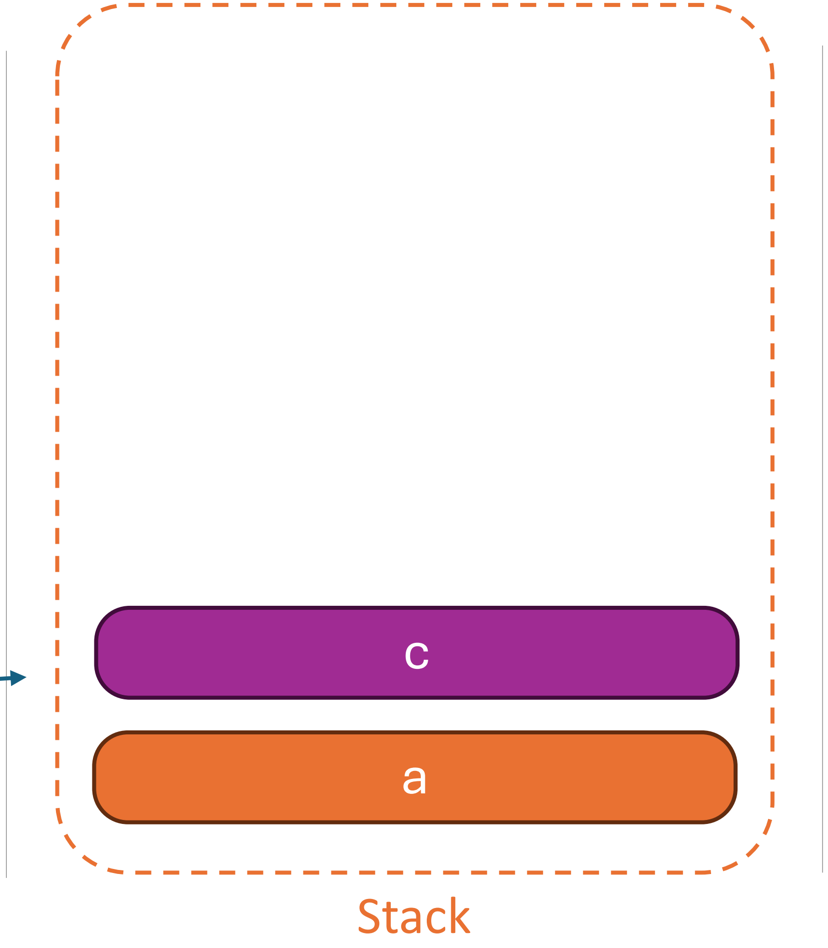
EXPLAIN

The **Stack** Segment

The stack segment stores local data with **automatic allocation** and **deallocation**

```c
int main() {
    int a = 10;

    if (a > 5) {
        int b = 20;
        printf("%d\n", b);
    }

    int c = 30;

   return 0;
}
```

C is allocated



EXPLAIN

The **Stack** Segment

The stack segment in memory is like the stack of plates at CADT cafeteria

Last plate added is the first taken off.

When you're done, you **remove the top plate**.

→ variables are **automatically removed** when a function or block ends.

First in First out approach

EXPLAIN

The Stack Segment – *In Short*

The **size must be fixed** at compile time !!

✓ **Statically allocated**

Memory size is known at compile time.

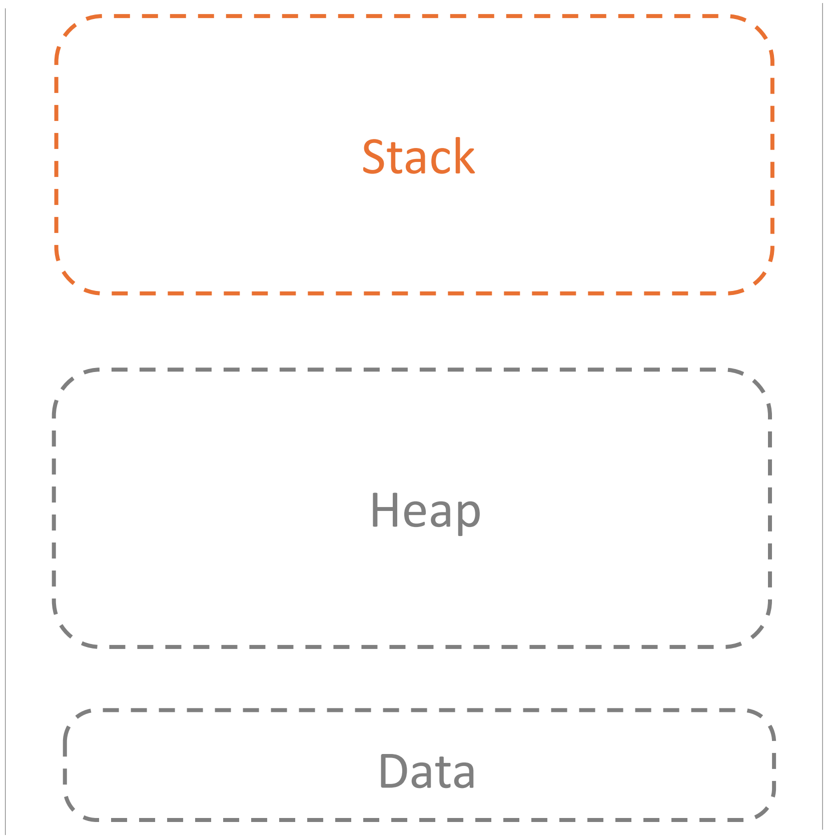
You **cannot change the size** of the variable afterwards !!

✓ **Automatically allocated / deallocated**

By the system when entering/exiting blocks or functions.

✓ **Variables are existing only within their scope**

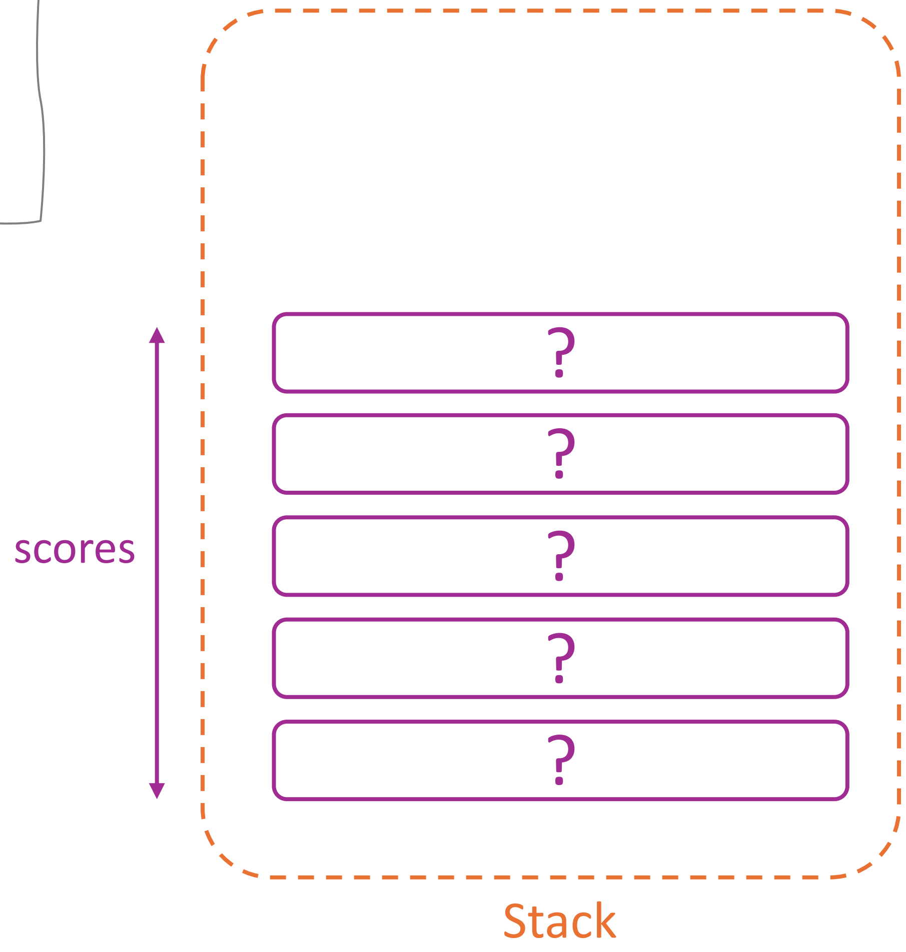
Scope = the block/function they were declared.



REFLECT

Suppose we want to store scores for 5 students

```
// We allocate space for 5 integer
int scores[5];
```

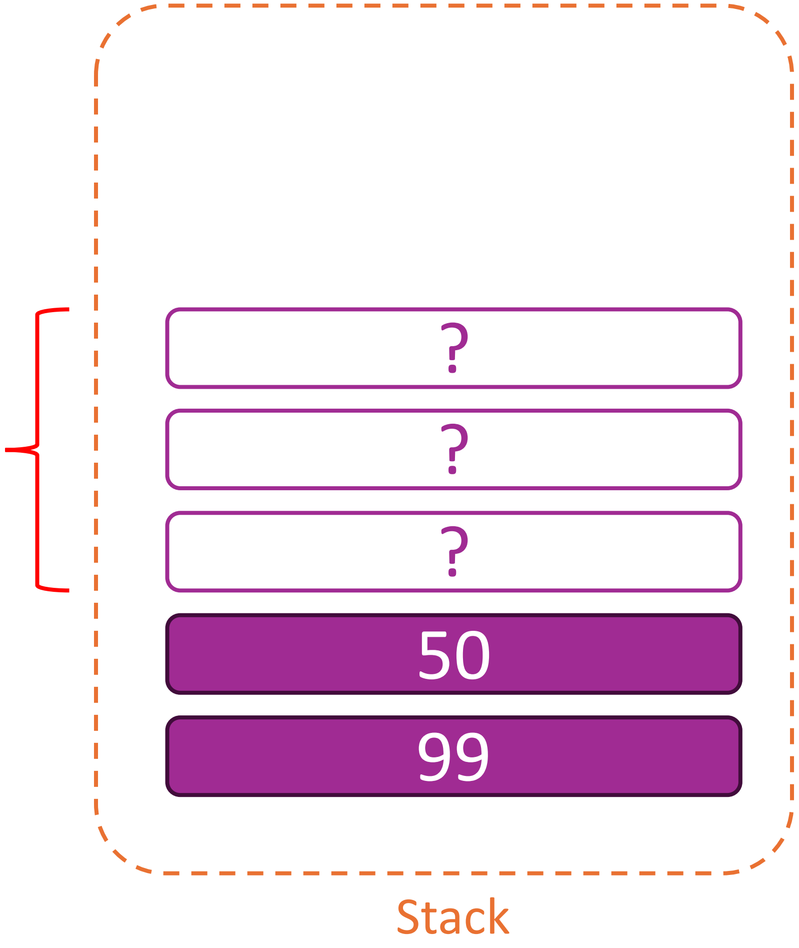


REFLECT

But we finally need only 2 scores….

```
// We allocate space for 5 integer
int scores[5];

// We set 2 scores
int scores[0] =99;
int scores[1] =50;
```

The space for 3 extra integers is wasted.



REFLECT

How to allocate exactly **what the user needs** in terms of memory ?

How would you declare an array if you don't know its size until the user inputs it?

EXPLAIN

The **Heap** Segment

If you need more flexibility, C allows you to **allocate memory at runtime** using the heap.

✓ **Not automatically managed**

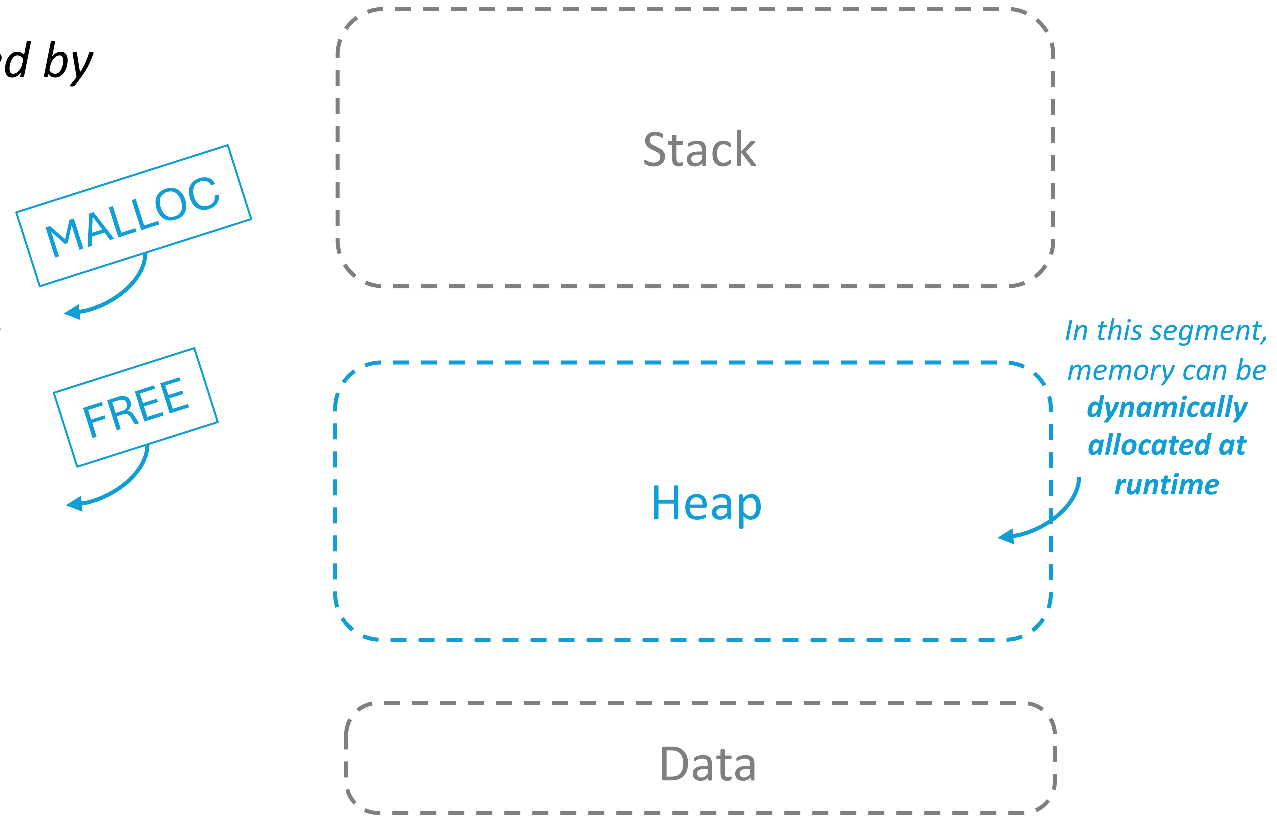
Unlike the stack, this memory space is not managed by the compiler.

✓ **Manually allocated**

Memory is manually allocated by the programmer.

✓ **Manually de-allocated**

Data in the heap stays there until the programmer free it.

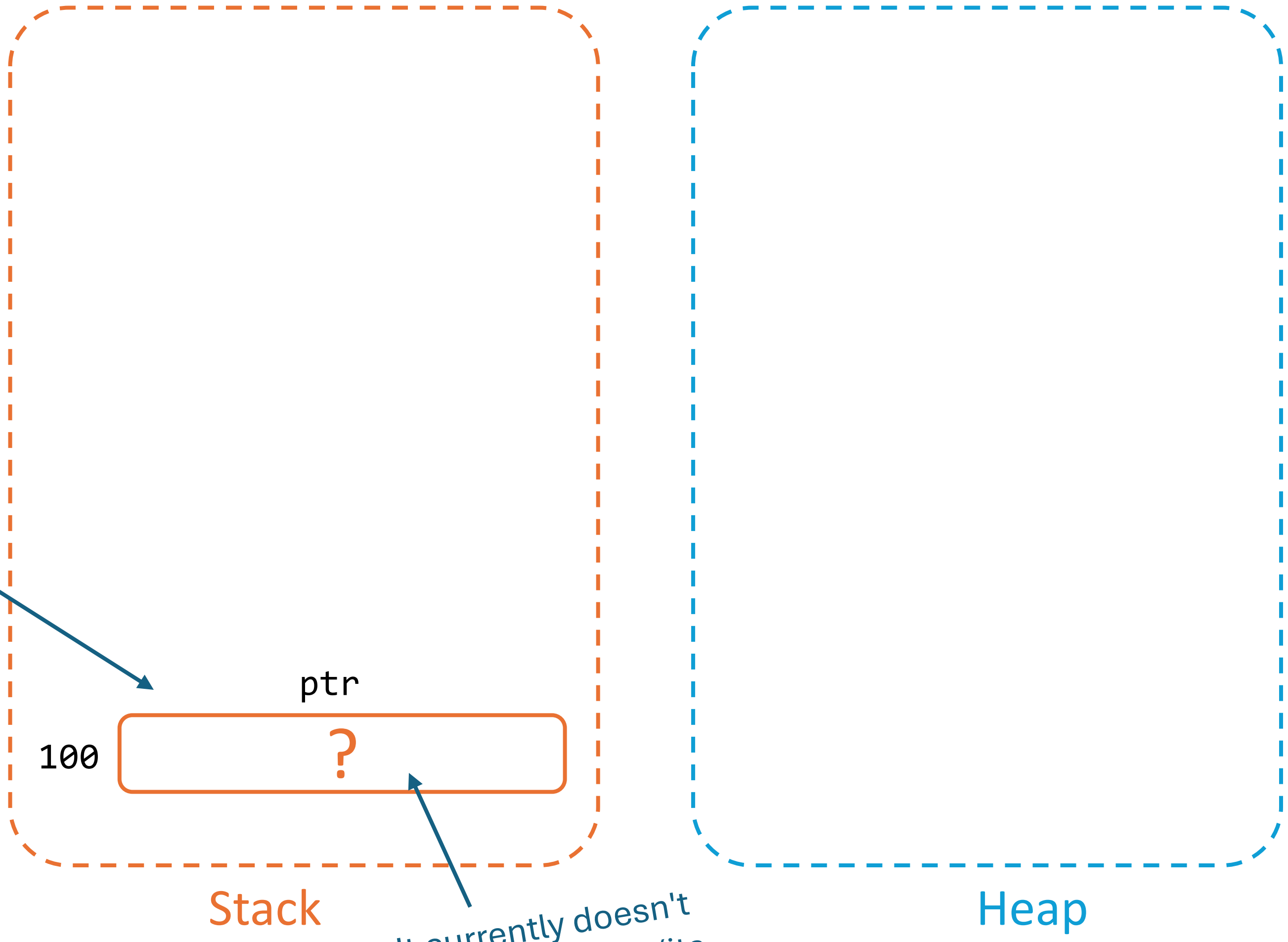


EXPLAIN

The **Heap** Segment - *Mechanism*

```
int* ptr;
```

A pointer to int is **declared** on the **stack segment**.



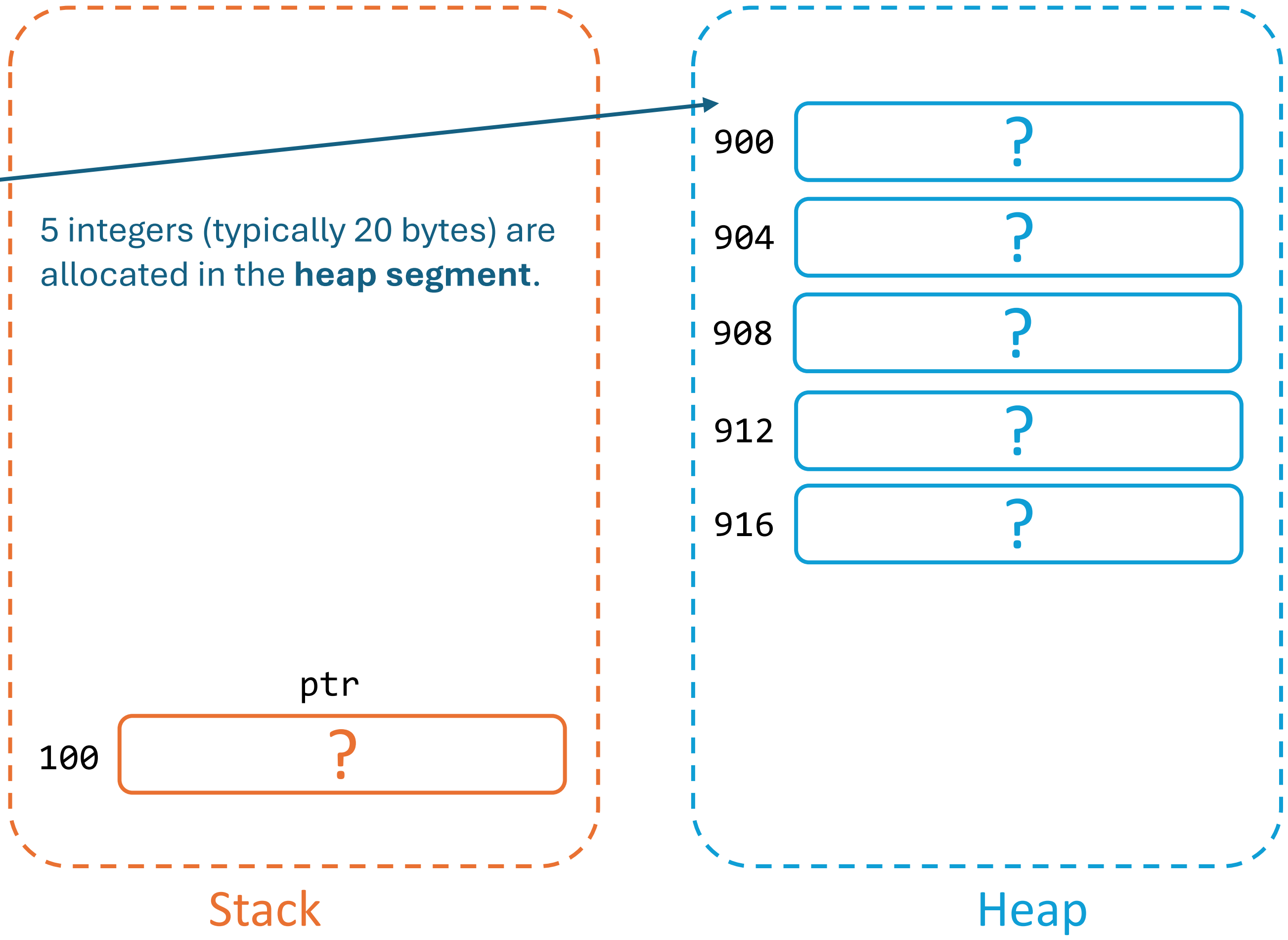
It currently doesn't **point anywhere** (its value is undefined).

EXPLAIN

The **Heap** Segment - *Mechanism*

```
int* ptr;

ptr = malloc(5*sizeof(int));
```

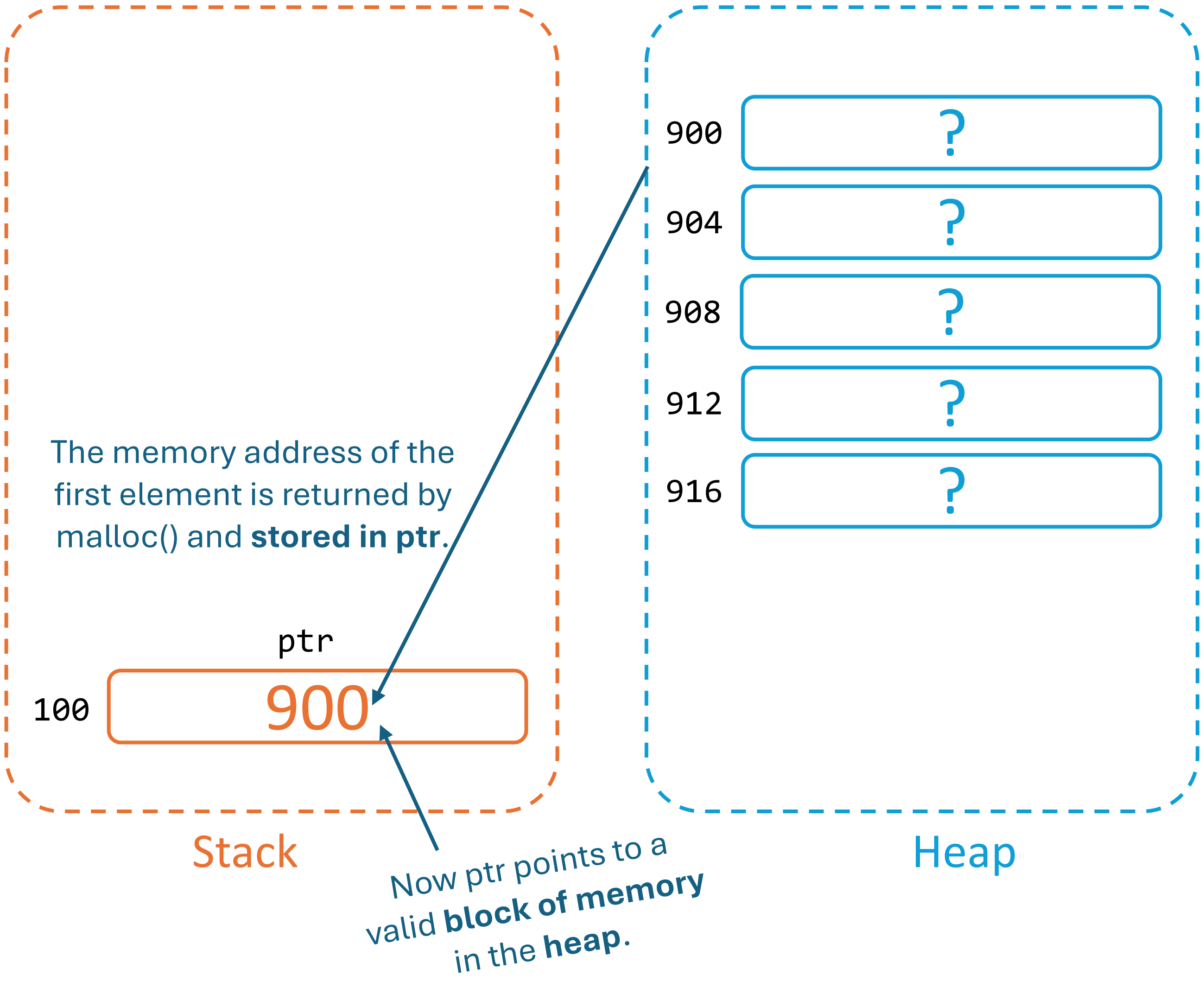
malloc() asks the OS for **dynamic memory** at runtime.



EXPLAIN

The **Heap** Segment - *Mechanism*

```
int* ptr;

ptr = malloc(5*sizeof(int));
```

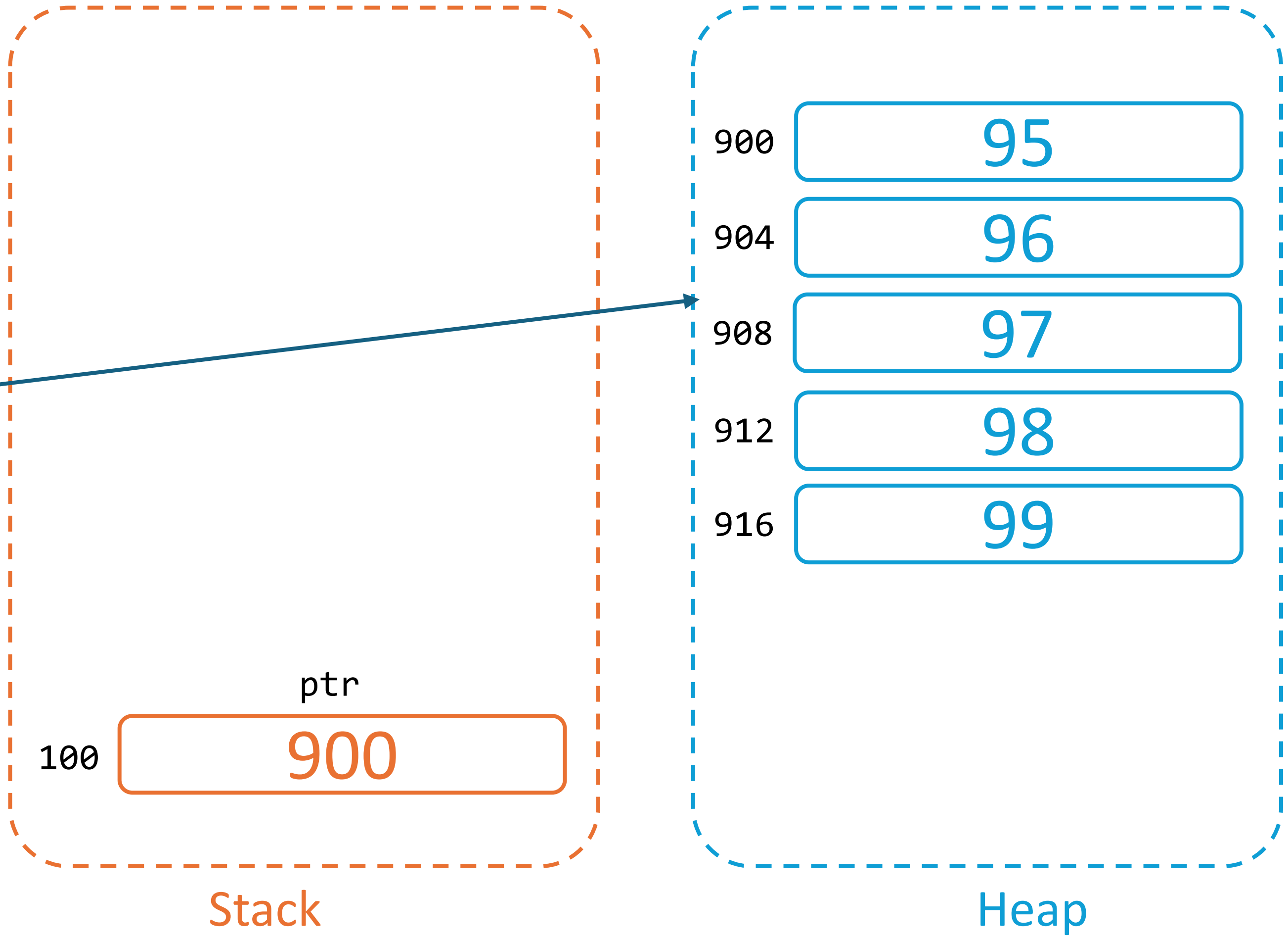


EXPLAIN

The Heap Segment - *Mechanism*

```c
int* ptr;

ptr = malloc(5*sizeof(int));

ptr[0] = 95;
ptr[1] = 96;
ptr[2] = 97;
ptr[3] = 98;
ptr[4] = 99;
```

Store the 5 values in the blocks of the heap memory pointed to by ptr.

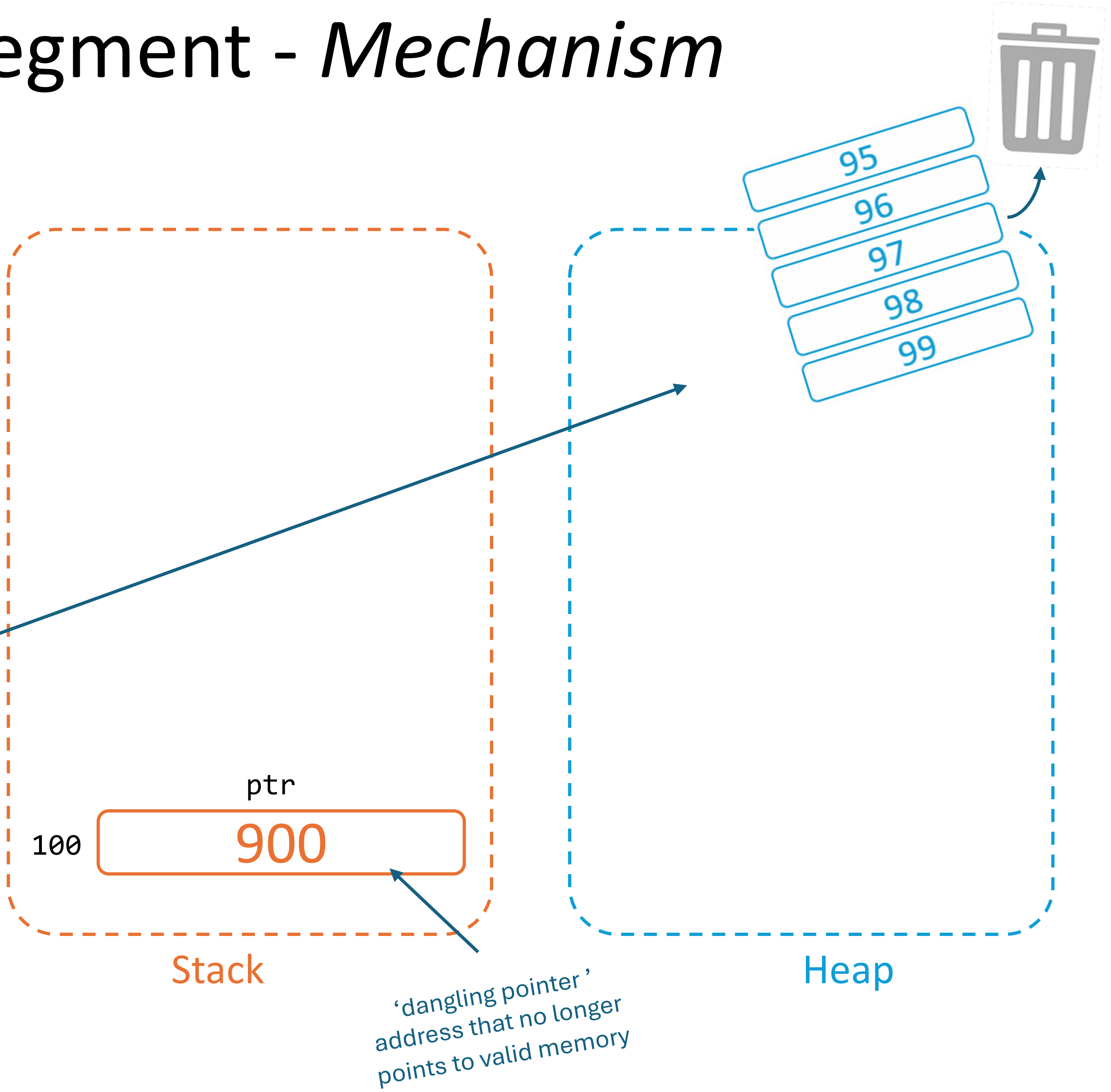


EXPLAIN

The Heap Segment - *Mechanism*

```
int* ptr;

ptr = malloc(5*sizeof(int));

ptr[0] = 95;
ptr[1] = 96;
ptr[2] = 97;
ptr[3] = 98;
ptr[4] = 99;


free(ptr)
```

De-allocate the memory in the **heap segment**



EXPLAIN

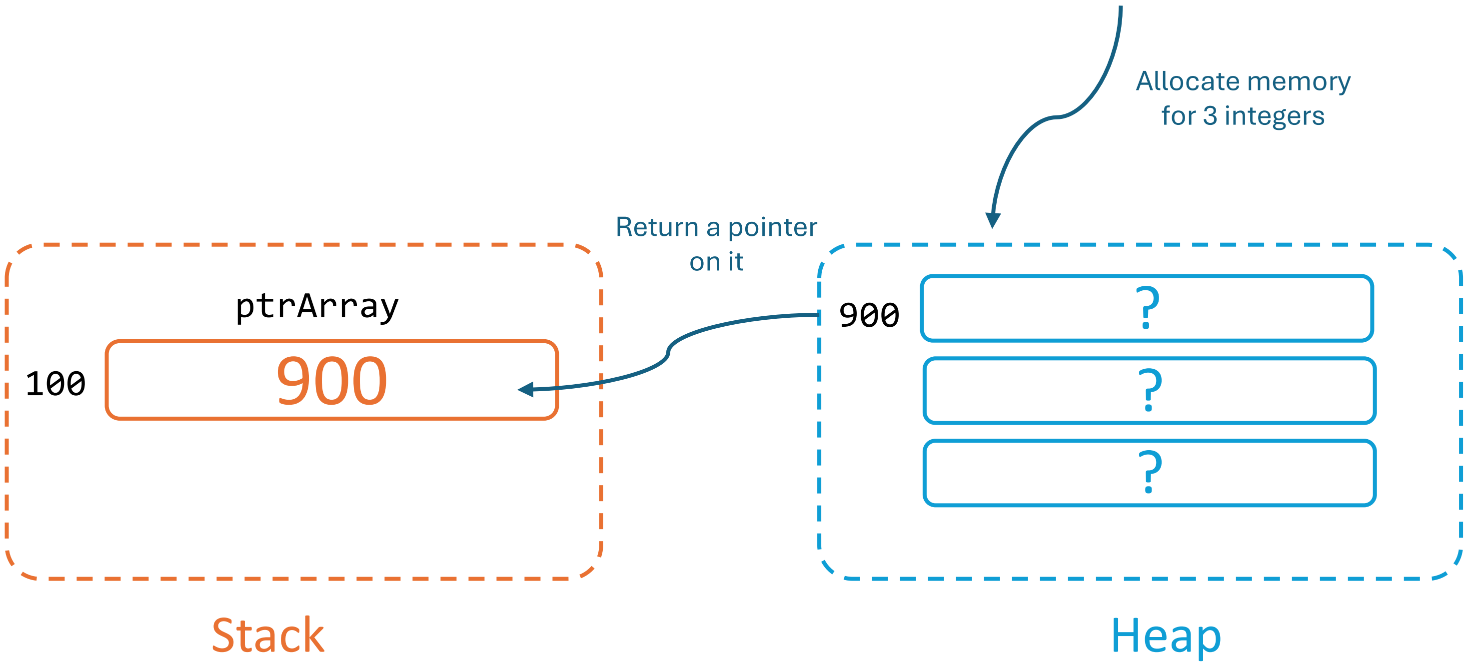
Malloc - Memory ALLOCation

The malloc() function **allocates memory** in the heap segment

A pointer to the newly allocated block of memory

The number of bytes of memory to allocate on heap.

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



EXPLAIN

Malloc - The returned void*

Dynamic memory **does not have its own data type** - It is just a sequence of bytes.

✓ The malloc function returns a void* pointer which is a **generic pointer**

```
// We allocate 4 bytes
void* ptr1 = malloc(4);
```

✓ We need then to cast the void* to the appropriate type of pointer, such as int*, char*, or float*

```
// We allocate 4 bytes, and use/hold it for 1 integer
int* ptr_on_integer = malloc(4);

// We allocate 4 bytes, and use/hold it for 4 chars
char* ptr_on_chars  = malloc(4);
```

EXPLAIN

Malloc - Important notes !

If the malloc function is **unable to allocate the memory** buffer, it returns **NULL**.

```
int *scores = malloc(n * sizeof(int));

if (scores == NULL) {
   printf("Memory allocation failed!\n");
}
```

malloc() **does not initialize** memory. Contents are random values.

?	?	?	?

Use sizeof(type) to avoid hardcoding byte sizes:

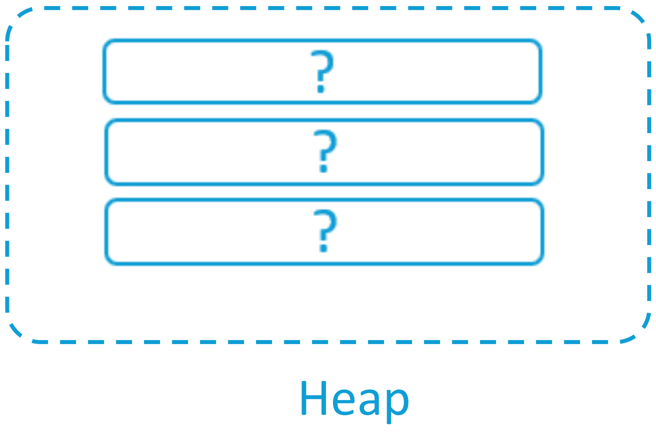
```
Point* points = malloc(10 * 24);               // unsafe
Point* points = malloc(10 * sizeof(Point) );   // safer and adaptable
```

EXPLAIN

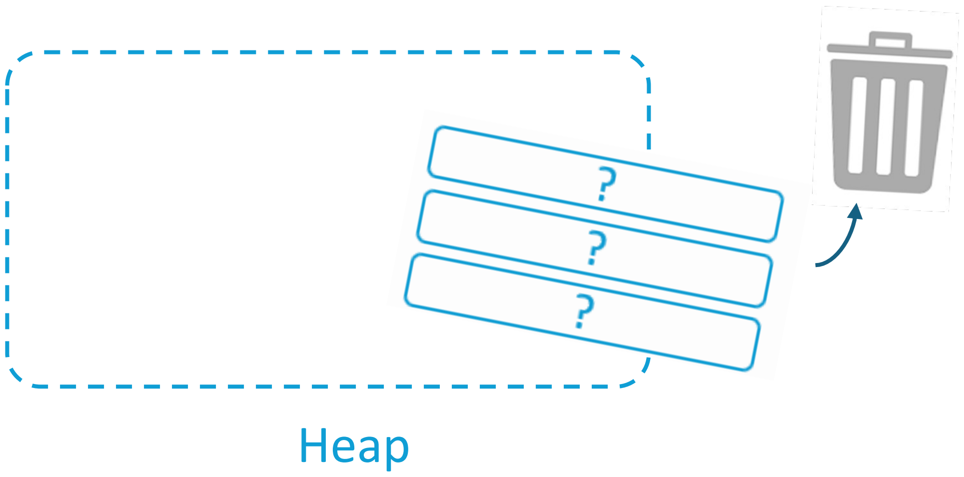
Free - Memory De-allocation

The memory allocated using malloc() is not de-allocated on their own. You need **free()**

```c
// Allocate memory
int* ptr = malloc(3*sizeof(int));
```




```c
// Releases memory allocated by malloc
free(ptr);

// Good practice, avoid dangling pointer
ptr = NULL;
```

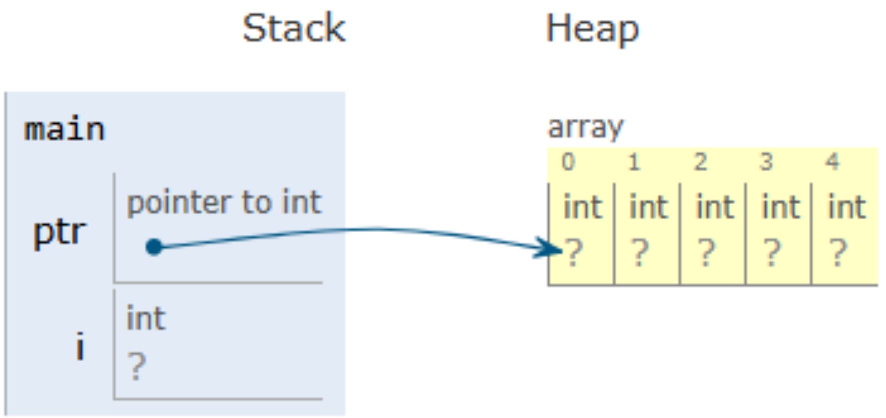


ACTIVITY

Let’s **Try** !

ONLINE DEBUGGER

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

int main() {

  // Allocate an array of 5 numbers on heap

  // Assign the 5 numbers to 1,2,3,4,5

  // De-allocate the array

  // Set the ptr on array to NULL

  return 0;
}
```



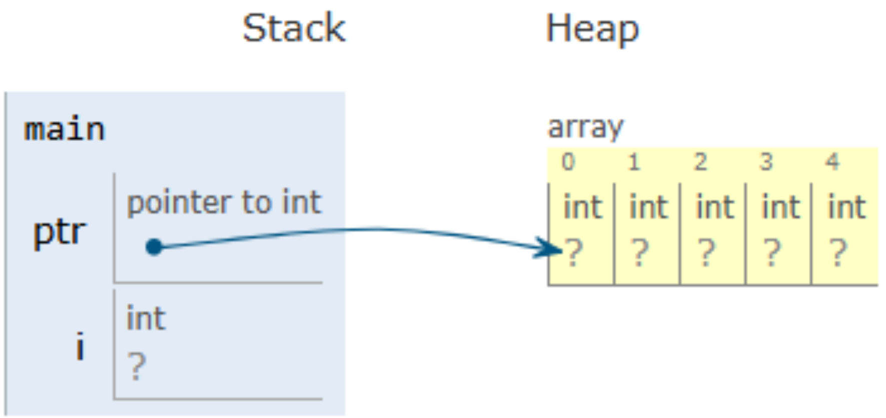
You need to observe step by step what’s happen on stack and on Heap

ANSWER

Let's Try !

ONLINE DEBUGGER

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

int main() {

  // Allocate an array of 5 numbers on heap
  int* ptr = malloc(5 * sizeof(int));

  // Assign the 5 numbers to 1,2,3,4,5
  for(int i=0; i<5; i++) {    ptr[i] = i + 1;   }

  // De-allocate the array
  free(ptr);

  // Set the ptr on array to NULL
  ptr = NULL;

  return 0;
}
```



You need to observe step by step what's happen on stack and on Heap

EXPLAIN

Wrap Up

Feature	Static (Stack)	Dynamic (Heap)
When allocated	Compile time	Runtime
Memory location	Stack	Heap
Automatically freed?	Yes (when block ends)	No (you must free() it manually)
Lifetime	Short (bloc-based)	Long (until free)
Size changeable?	❌ No	✅ Yes (by reallocating)
Syntax example	int arr[10];	int *arr = malloc(10 * sizeof(int));
Usage	Local variables	Dynamic arrays, large data...

ACTIVITY

Let's **Try** !

VS CODE

<u>Write a C program that:</u>

- Asks the user **how many integers they want to enter**.
- **Allocates dynamic memory** (on the heap) for that many integers using **malloc**.
- Asks the user to **input the integers**.
- Calculates and displays the sum of the entered values.
- **Frees** the allocated memory before exiting.

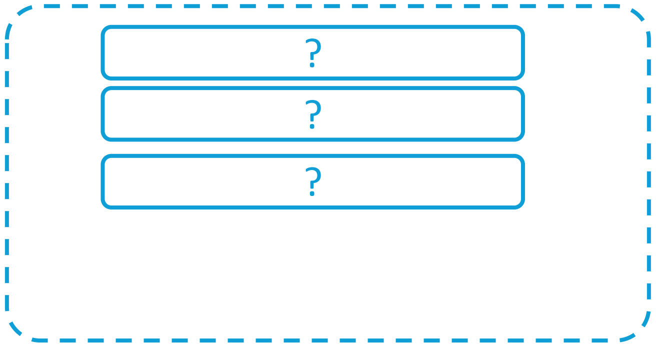
```
How many numbers? 4
Enter 4 integers: 10 20 30 40
Sum = 100
```

EXPLAIN

realloc - Resize Allocated Memory

Sometime the amount of data your program needs to store can change.

```
// Allocate memory
int* ptr = malloc(3*sizeof(int));
```

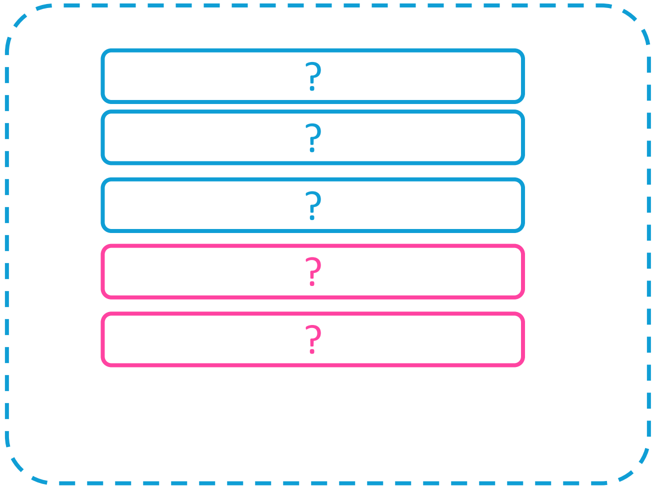


Heap

```
// Re-Allocate memory
int* newptr  = realloc (ptr , 5*sizeof(int));
```

A pointer to the resized memory block
(May be different from the original! May be also NULL !).

```
If (newptr != NULL) {
        ptr = newptr;
}
```



Heap

EXPLAIN

realloc - Important notes !

If the realloc function is **unable to reallocate the memory** buffer, it returns **NULL**.

```
int *newPtr = realloc(ptr, n * sizeof(int));
if (scores == NULL) {
   printf("Memory reallocation failed!\n");
}
```

If ptr == NULL: Behaves like malloc(new_size).

```
realloc (NULL, ...);
```

If new size = 0 , It frees the memory (same as free(ptr))

```
realloc (ptr, 0);
```

EXPLAIN

Dynamic Allocation - *Common Dangers*

Danger	Impact
❌ Not using free	Memory leak
❌ Using memory after free	Dangling pointer
❌ Not checking for NULL	Crash if allocation fails
❌ Forgetting sizeof()	Incorrect allocation size

ACTIVITY

Let’s **Try** !

Copy a string into a newly allocated memory area using **dynamic memory allocation**

1. Prompt the user to enter a string.

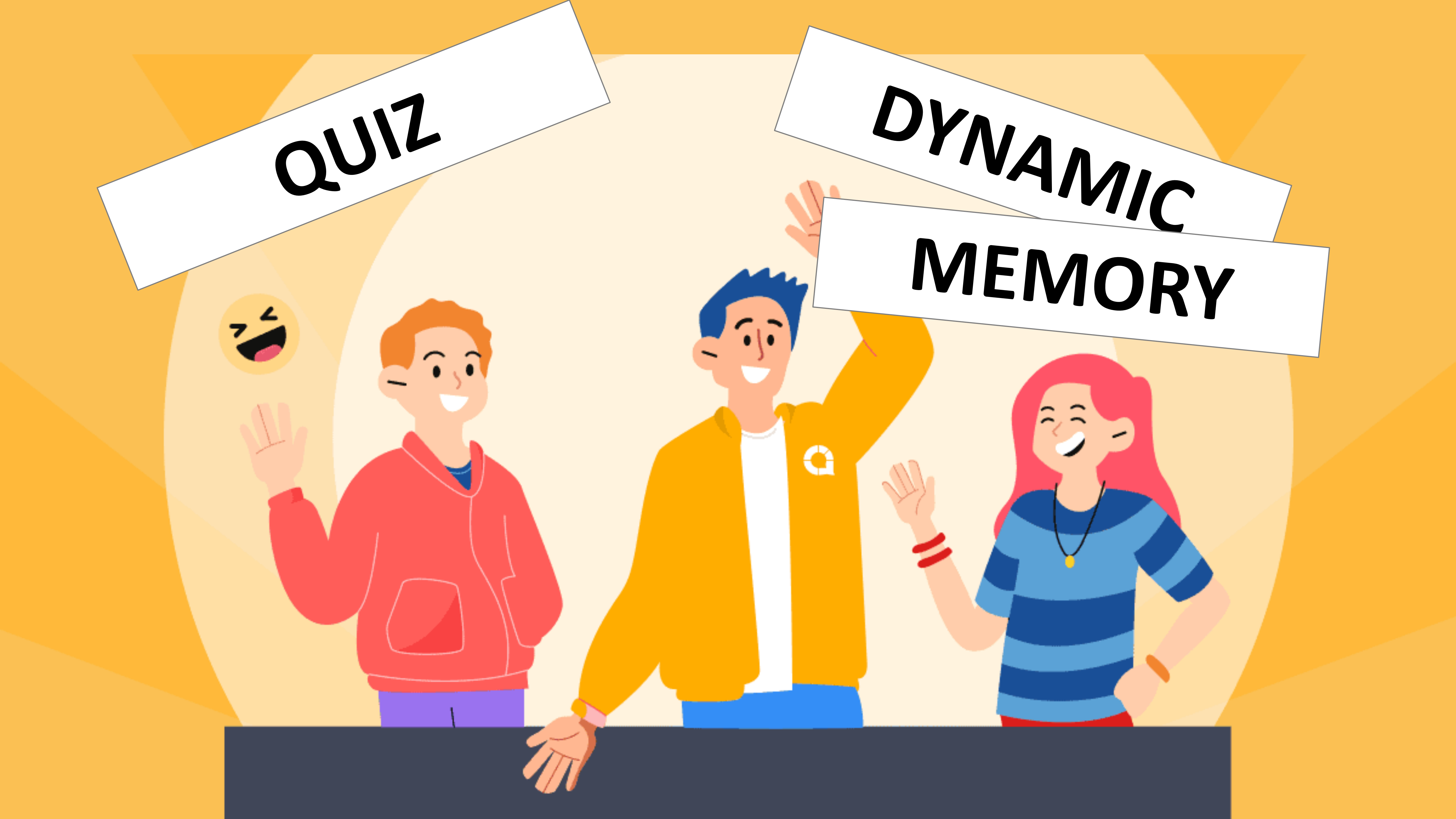
2. Determine the length of the string.
3. Dynamically allocate memory for the copy using malloc().

4. Copy the original string into the allocated memory (use strcpy()).
5. Display the copied string.

6. Free the dynamically allocated memory using free().

Don’t forget to allocate space for the null terminator '\0'.

```
Enter a string: Hello, world!
Copied string: Hello, world!
```


QUIZ
DYNAMIC
MEMORY

QUIZ

Fill in the blank to correctly allocate memory for n integers:

```
int* arr = ________;
```

A. malloc(n)
B. malloc(n * int)
C. malloc(n * sizeof(int))
D. malloc(sizeof(arr))

ANSWER

Fill in the blank to correctly allocate memory for n integers:

```
int* arr = ________;
```

A. malloc(n)
B. malloc(n * int)
C. malloc(n * sizeof(int))
D. malloc(sizeof(arr))

QUIZ

What is the size of memory allocated by:

```
int* a    = malloc(sizeof(int));
char* b   = malloc(5);
float* c  = malloc(2 * sizeof(float));
```

- **A.** 10 bytes
- **B.** 13 bytes
- **C.** 17 bytes
- **D.** 9 bytes

ANSWER

What is the size of memory allocated by:

```
int* a   = malloc(sizeof(int));
char* b  = malloc(5);
float* c = malloc(2 * sizeof(float));
```

A. 10 bytes

B. 13 bytes

C. 17 bytes

D. 9 bytes

QUIZ

The malloc() function returns a pointer of type ________

A. int*
B. char*
C. void*
D. size_t

ANSWER

The malloc() function returns a pointer of type ________

A. int*
B. char*
C. void*
D. size_t

QUIZ

What will happen when the following code is executed

```
int* p = malloc(3 * sizeof(int));
p[3] = 40;
printf("%d\n", p[3]);
```

A. The code prints 40 with no issue
B. The program crashes or shows undefined behavior
C. The compiler prevents access beyond allocated memory
D. The code prints a random value but is still safe

ANSWER

What will happen when the following code is executed

```
int* p = malloc(3 * sizeof(int));
p[3] = 40;
printf("%d\n", p[3]);
```

malloc(3 * sizeof(int)) allocates space for 3 integers

Accessing p[3] is out-of-bound

A. The code prints 40 with no issue
B. The program crashes or shows undefined behavior
C. The compiler prevents access beyond allocated memory
D. The code prints a random value but is still safe

QUIZ

Which line causes a **memory leak**?

```
int* p    = malloc(10 * sizeof(int));
p         = malloc(20 * sizeof(int));
free(p);
```

A. First malloc
B. Second malloc
C. free(p)
D. No memory leak

ANSWER

Which line causes a **memory leak**?

```
int* p    = malloc(10 * sizeof(int));
p         = malloc(20 * sizeof(int));
free(p);
```

The pointer to the first block is lost before it can be freed.

A. First malloc
B. Second malloc
C. free(p)
D. No memory leak

QUIZ

What is the purpose of this check?

```
int* p = malloc(100);
if (p == NULL) {
    printf("Allocation failed");
}
```

A. To check if memory is freed

B. To validate pointer type

C. To avoid using an invalid pointer if allocation fails

D. To initialize memory

ANSWER

What is the purpose of this check?

```
int* p = malloc(100);
if (p == NULL) {
    printf("Allocation failed");
}
```

A. To check if memory is freed

B. To validate pointer type

C. To avoid using an invalid pointer if allocation fails

D. To initialize memory

🥇 Let's sum up 🥇

🧠 **Static memory (stack)**

- Is allocated at compile time with fixed size
- Managed automatically,

⚠️ *cannot be resized during runtime.*

🧠 **Dynamic memory (heap)**

- Is allocated during program execution
- Shall be manually release with free() to avoid memory leaks.

🧱 **malloc()**

- Allocates uninitialized memory and returns a pointer

⚠️ *Always check for NULL !*

⚠️ *Use sizeof to calculate the correct size.*

🧹 **free()**

- Releases dynamically allocated memory

⚠️ *Using freed memory or forgetting to free leads to errors*

🔄 **realloc()**

- resizes a previously allocated block

⚠️ *Assign the result to a temporary pointer (if allocation fails)*

Go further after the class…

Memory allocation
https://www.w3schools.com/c/c_memory_access.php

https://www.programiz.com/c-programming/c-dynamic-memory-allocation

Exercises
https://c-pointers.com/malloc_ptr/exercises/exercises.html