Team: Code Wizard
Members:

- Phanthira Kositjaroenkul (6630003)
- Kaung Khant Lin (6540131)
- Thit Lwin Win Thant (6540122)

Task 1: Code Analysis

1. **Storage location**
    a. **counter -** Global variable with static storage duration; placed in the data segment (.data because it has an explicit non-zero initializer 10).
    b. **x -** Automatic (local) variable; stored on the stack frame of `foo()`.
    c. **p**: Automatic (local) variable (the pointer itself); stored on the stack frame of `foo()`.
    d. ***p (the int whose value becomes 20)**: Stored in a heap-allocated block returned by `malloc(sizeof(int))`.
2. **Allocation / deallocation timing**
    a. **counter**: Allocated (storage reserved) at program load time, before main(); deallocated when the process terminates.
    b. **x**: Space reserved when `foo()` is entered (stack frame creation); reclaimed automatically when `foo()` returns.
    c. **p** (the pointer variable): Same lifetime as x (allocated on entry to `foo()`, reclaimed on return).
    d. **Heap int (*p)**: Allocated at the moment `malloc()` succeeds; its lifetime ends only when `free(p)` is called (never in this code). If not freed, it remains allocated until process exit.
3. **If free(p) is not called before foo() ends**
    a. The pointer p goes out of scope; the program loses the only reference to that heap block → memory leak (4 bytes here).
    b. For this short-lived program the OS reclaims all process memory at exit, so no persistent system-level harm, but it is still a leak (detected by tools like Valgrind) and would accumulate in long-running code.
    c. No immediate undefined behavior occurs solely from failing to free; the risk is unbounded growth and possible fragmentation in larger / repeated allocations.

Task 2: Memory Layout Diagram

Static Area	counter
Stack	x, p
Heap	*p

1. **Static Area**:
   - counter resides here because it's a global variable (declared outside any function) with static storage duration.
2. **Stack**:
   - x is a local automatic variable inside foo(), so it's stored on the stack.
   - p is a local pointer variable (also automatic storage), so it's on the stack too. However, what p points to is in the heap.
3. **Heap**:
   - The memory allocated by malloc(sizeof(int)) comes from the heap. While te pointer p itself is on the stack, the actual integer it points to (*p) is in the heap.

Reasons:

- The static area stores variables that exist for the entire program duration.

- The stack stores function call information and local variables (automatic storage).

- The heap stores dynamically allocated memory that persists until explicitly freed.

The diagram shows the relationship between these memory regions and where each variable in the code resides during the execution of “foo()”.

Task 3: Discussion

1. Why do local variables disappear after a function ends?

Local variables disappear after their function ends because they're stored in a temporary workspace that's deleted as soon as the function is finished.

a temporary memory space called the call stack.

- When a function is called, a new memory block, known as a stack frame, is created on top of the stack to hold all of that function's local variables and parameters. This frame exists only for the lifetime of the function; as soon as the function finishes its job and returns, its entire stack frame is popped off, or removed, from the stack.

  This process automatically destroys all the local variables contained within that frame, reclaiming the memory they occupied and ensuring they cannot be accessed after the function is complete.

2. How is memory management in Python or ML different from C?

Memory management in Python and ML is fundamentally different from C because it is automatic rather than manual.

In C, the programmer is directly responsible for memory, using functions like `malloc()` to allocate it and `free()` to deallocate it, which offers high performance but carries the risk of errors like memory leaks.

Python, in contrast, automates this process using a system of reference counting where memory is freed once an object has no more references pointing to it and a backup garbage collector that cleans up more complex circular references.
For example, creating a variable in Python (`my_list = [1, 2, 3]`) requires no manual deallocation; the memory is reclaimed automatically when `my_list` is no longer in use.

In ML, libraries like TensorFlow and PyTorch build on this by implementing their own sophisticated memory managers that pre-allocate and cache large blocks of memory for tensors, especially on GPUs, to optimize performance, but the core principle for the developer remains the same: the memory is managed automatically, abstracting away the low-level complexity seen in C.

References

- https://medium.com/@lsltry404/memory-usage-in-c-programming-a-comprehensive-guide-b20038647992
- https://www.geeksforgeeks.org/c/function-call-stack-in-c/
- https://www.geeksforgeeks.org/python/memory-management-in-python/
- https://www.w3schools.com/c/c_memory_management.php
- https://www.quora.com/How-do-memory-management-techniques-differ-between-languages-like-C-and-Python
- https://www.gnu.org/software/libc/manual/html_node/Memory-Concepts.html
- https://learn.microsoft.com/en-us/cpp/cpp/memory-layout