Gorgo C for C++ Programmers

April 11, 2026

8. Allocating Memory

Every variable in your program lives somewhere in memory, but not all memory is created equal. Understanding where variables live — and how long they last — is important for writing correct C programs. It's such a challenging task that new languages such as Java and Rust were designed to help developers manage variable lifetimes.

Global Variables

A **global variable** is declared outside of any function. It is created when the program starts and exists until the program exits:

```c
#include <stdio.h>

int high_score = 0;   // global — lives for the entire program

void update_score(int points) {
    if (points > high_score) {
        high_score = points;
    }
}

int main(void) {
    update_score(1000);
    update_score(500);
    printf("High score: %d\n", high_score);  // 1000
```

```
    return 0;
}
```

Global variables are visible to every function in the file. They are convenient but can make programs harder to reason about, since any function can change them.

Tip: Use global variables sparingly. When every function can read and write the same variable, bugs become harder to track down. Prefer passing data through function parameters.

Local Variables

A **local variable** is declared inside a function (or block). It is created when the function is called and destroyed when the function returns:

```c
void greet(void) {
    char message[] = "Hola, amigo";  // local — exists only during greet()
    printf("%s\n", message);
}
// message is gone once greet() returns
```

Local variables live on the **stack** — a region of memory that grows and shrinks automatically as functions are called and return. You do not need to free stack memory; it is reclaimed automatically.

Trap: Never return a pointer to a local variable. The memory is freed when the function returns, and the pointer becomes a **dangling pointer** — it points to memory that no longer belongs to you:

```c
int *bad(void) {
    int x = 42;
    return &x;   // BUG: x is destroyed when bad() returns
}
```

Static Local Variables

A **static local variable** has the scope of a local variable but the lifetime of a global. It is declared inside a function with the `static` keyword, created once when the program starts, and retains its value between calls:

```c
#include <stdio.h>

void count_calls(void) {
    static int count = 0;   // initialized once, persists between calls
    count++;
    printf("Called %d time(s)\n", count);
}

int main(void) {
    count_calls();   // Called 1 time(s)
    count_calls();   // Called 2 time(s)
    count_calls();   // Called 3 time(s)
    return 0;
}
```

Without `static`, `count` would be reset to 0 on every call. With `static`, it lives in the data segment (like a global) but is only accessible inside `count_calls`.

Dynamic Allocation: `malloc` and `free`

```
void *malloc(size_t size);
void free(void *ptr);
```

Sometimes you need memory that outlives the function that created it, or memory whose size you do not know at compile time. For this, C provides `malloc` and `free` from `<stdlib.h>`.

`malloc` allocates a block of memory on the **heap** and returns a pointer to it. The heap is a region of memory that persists until you explicitly release it:

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

int main(void) {
    int *nums = malloc(5 * sizeof(int));
    if (nums == NULL) {
        printf("Allocation failed!\n");
        return 1;
    }

    for (int i = 0; i < 5; i++) {
        nums[i] = (i + 1) * 10;
    }

    for (int i = 0; i < 5; i++) {
        printf("nums[%d] = %d\n", i, nums[i]);
    }

    free(nums);   // release the memory back to the system
    return 0;
}
```

`malloc` returns a `void *` — a generic pointer that can be assigned to any pointer type without a cast in C. It returns `NULL` if the allocation fails. Always check for `NULL` after calling `malloc`. Having said that, there is a school of thought held by some very good engineers that `NULL` checks just clutter the code. To handle `NULL` gracefully, there will be a check and logic every place where it could be `NULL`, so you end up with checks and logic that are rarely used and almost never tested. If you are out of memory, you'll probably need to shut down the program, so the reasoning goes that if you try to use a NULL pointer the CPU will do the check for you. For safety-critical systems, the above argument does not hold, although those systems often forbid the use of dynamically allocated memory entirely.

> **Tip:** There are no smart pointers in C. There is no RAII. There is no garbage collector. If you call `malloc`, you *must* call `free` when you are done. If you forget, you leak memory. If you call `free` twice on the same pointer, you get undefined behavior. If you use a pointer after freeing it, you get undefined behavior. Memory management in C is entirely your responsibility.

`calloc` is a variant that allocates memory and initializes it to zero:

```
void *calloc(size_t count, size_t size);

int *nums = calloc(5, sizeof(int));  // 5 ints, all initialized to 0
```

And `realloc` lets you resize a previously allocated block:

```
void *realloc(void *ptr, size_t size);

nums = realloc(nums, 10 * sizeof(int));  // grow to 10 ints
```

Trap: Never assign the result of `realloc` directly back to the same pointer. If `realloc` fails, it returns `NULL` and the original memory is not freed — so `nums = realloc(nums, ...)` loses your only pointer to the original block, causing a memory leak. Use a temporary pointer instead:

```c
int *tmp = realloc(nums, 10 * sizeof(int));
if (tmp == NULL) {
    // handle error — nums is still valid
} else {
    nums = tmp;
}
```

Working with Raw Memory: `memcpy` and `memset`

Two functions from `<string.h>` operate on raw bytes rather than strings. You will see them constantly in C code:

`memset` fills a block of memory with a byte value. It is commonly used to zero out a buffer:

```c
void *memset(void *s, int c, size_t n);
```

```c
int nums[10];
memset(nums, 0, sizeof(nums));   // set all bytes to 0
```

`memcpy` copies a block of bytes from one location to another. Unlike `strcpy`, it does not stop at a `'\0'` — you tell it exactly how many bytes to copy:

```c
void *memcpy(void *dest, const void *src, size_t n);
```

```c
int src[] = {10, 20, 30};
int dest[3];
memcpy(dest, src, sizeof(src));  // copy all 12 bytes (3 ints × 4 bytes)
```

Trap: `memcpy` requires that the source and destination do not overlap. If they might overlap (e.g., shifting elements within the same array), use `memmove` instead, which handles overlapping regions correctly.

```c
void *memmove(void *dest, const void *src, size_t n);
```

Where Variables Live: A Summary

Kind	Where	Lifetime	Example
Global	Data segment	Entire program	`int count = 0;` (outside functions)
Local	Stack	Until function returns	`int x = 5;` (inside a function)
Static local	Data segment	Entire program	`static int n = 0;` (inside a function)
Dynamic	Heap	Until you call `free`	`int *p = malloc(...)`

Try It: Memory Lifetimes

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

int total = 0;   // global — lives for the whole program

void add_to_total(int n) {
```

```c
    int local = n;       // local — gone when add_to_total returns
    total += local;
}

int main(void) {
    add_to_total(10);
    add_to_total(20);
    printf("Total: %d\n", total);    // 30

    // dynamic — lives until we free it
    int *data = malloc(3 * sizeof(int));
    if (data == NULL) return 1;

    data[0] = 1985;
    data[1] = 1986;
    data[2] = 1987;

    for (int i = 0; i < 3; i++) {
        printf("data[%d] = %d\n", i, data[i]);
    }

    free(data);
    return 0;
}
```

Key Points

- Global variables live for the entire program; local variables live only until the function returns.
- Static local variables have the scope of a local but the lifetime of a global.
- `malloc` allocates memory on the heap. You must call `free` when done.
- `calloc` allocates and zeroes memory. `realloc` resizes an allocation.
- `memcpy` copies bytes between non-overlapping regions. Use `memmove` for overlapping regions.
- `memset` fills a block of memory with a byte value.

Exercises

1. **Think about it:** Why would you choose `calloc` over `malloc` followed by `memset` to zero?
2. **What does this print?**

   ```c
   #include <stdio.h>

   void counter(void) {
       static int n = 0;
       n++;
       printf("%d ", n);
   }

   int main(void) {
       counter(); counter(); counter();
       return 0;
   }
   ```

3. **Calculation:** On a system where `int` is 32 bits, how many bytes does `malloc(5 * sizeof(int))` allocate?

4. **Where is the bug?**

```c
int *p = malloc(10 * sizeof(int));
for (int i = 0; i < 10; i++) {
    p[i] = i;
}
free(p);
printf("%d\n", p[0]);
```

5. **Where is the bug?**

```c
int *a = malloc(5 * sizeof(int));
int *b = a;
free(a);
free(b);
```

6. **Write a program** that uses `malloc` to allocate an array of `n` integers (where `n` is provided by the user via `scanf`), fills the array with squares (0, 1, 4, 9, ...), prints them, and frees the memory.