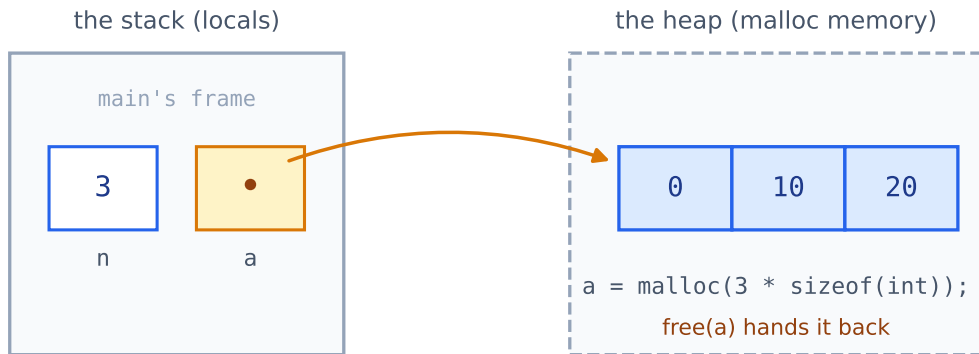


Dynamic memory

C Reference

malloc, free & realloc

`malloc` allocates memory on the heap at run time and returns a pointer to it. `realloc` resizes that block; `free` gives it back. Every `malloc` needs a matching `free`, or you get a memory leak. These live in `<stdlib.h>`.



09-stack-heap

Locals live on the stack; malloc memory lives on the heap until you free it

```

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

int main(void) {
    int n = 3;
    int *a = malloc(n * sizeof(int));    // room for 3 ints
    if (a == NULL) return EXIT_FAILURE;
    for (int i = 0; i < n; i++) a[i] = i * 10;

    int *grown = realloc(a, 4 * sizeof *a); // positive size
    if (grown == NULL) {
        free(a);
        return EXIT_FAILURE;
    }
    a = grown;
    a[3] = 30;

    for (int i = 0; i < 4; i++) printf("%d ", a[i]);
    printf("\n");                // 0 10 20 30

    free(a);                      // hand the memory back
    return 0;
}

```

- `calloc(n, size)` allocates an array like `malloc` AND fills it with zeros.

Common mistakes

- Every `malloc` needs a matching `free`; forgetting to free leaks memory.
- Using memory after `free` (a "use-after-free") is a serious bug.
- Check that `malloc` did not return `NULL` before you use the memory.

For positive sizes, a failed `realloc` returns `NULL` and leaves the old allocation intact: check a temporary result before replacing your only pointer. Success preserves the smaller of the old/new byte counts; added cells are uninitialised. This example avoids zero-size requests and size overflow. Ordinary non-static locals have automatic storage duration and are usually implemented on a stack; C does not require that layout.