Chapter 14: Interlude: Memory API

This chapter provides a practical guide to memory allocation in UNIX/C programs. Understanding how to correctly allocate and manage memory is a critical skill for building robust and reliable software.

> **The Crux of the Problem: How to Allocate and Manage Memory?**
>
> In UNIX/C programs, understanding how to allocate and manage memory is critical. What interfaces are commonly used? What mistakes should be avoided?

1. Types of Memory

In a C program, memory is primarily allocated in two distinct regions: the stack and the heap.

1.1. Stack Memory (Automatic Memory)

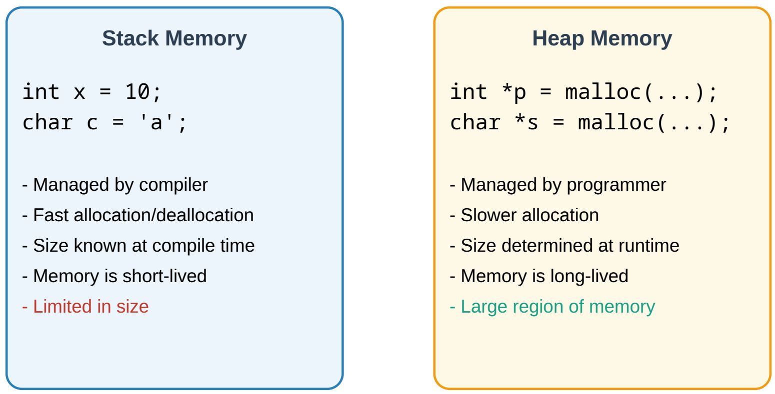
- **Management:** Allocation and deallocation are managed **implicitly** by the compiler. It is "automatic."
- **Usage:** Used for local variables within functions, function parameters, and return addresses.
- **Lifecycle:** Memory is allocated on the stack when a function is called and is automatically deallocated when the function returns.
- **Limitation:** Data on the stack is temporary. It cannot be used to store information that needs to persist beyond a single function call.

```c
void func() {
    int x; // 'x' is allocated on the stack.
           // It is automatically destroyed when func() returns.
}
```

1.2. Heap Memory

- **Management:** Allocation and deallocation are handled **explicitly** by the programmer.
- **Usage:** Used for long-lived data that must persist across function calls, such as complex data structures (linked lists, trees) or large arrays whose size is determined at runtime.
- **Lifecycle:** Memory allocated on the heap remains allocated until the programmer explicitly frees it.

Stack vs. Heap Memory



2. The `malloc()` and `free()` API

2.1. The `malloc()` Call

The `malloc()` function is the primary way to request heap memory.

- **Signature:** `void *malloc(size_t size);`
- **Action:** It takes a single argument, `size`, which is the number of bytes to allocate.
- **Return Value:**
  - On success, it returns a `void *` pointer to the newly allocated block of memory.
  - On failure (e.g., if the system is out of memory), it returns `NULL`. **Always check for `NULL`!**
- **Usage:**
  - Use the `sizeof()` operator to calculate the correct number of bytes needed for a data type (e.g., `malloc(sizeof(int))`). This is a compile-time operator, not a run-time function call.
  - The returned `void *` is a generic pointer. It should be **cast** to the appropriate pointer type (e.g., `(int *)`) to inform the compiler and other programmers of its intended use.

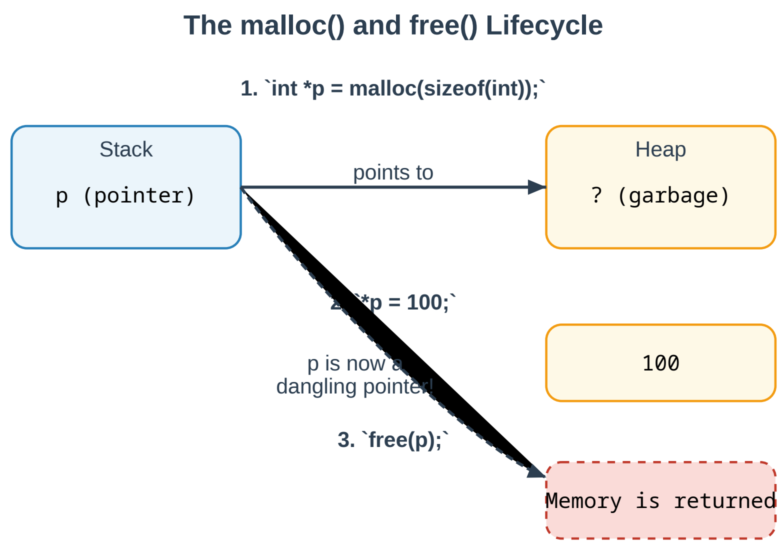
```
// Allocate space for 10 integers on the heap
int *data = (int *) malloc(10 * sizeof(int));
if (data == NULL) {
    // Handle allocation failure
    exit(1);
}
```

2.2. The `free()` Call

The `free()` function is used to return heap memory to the system.

- **Signature:** `void free(void *ptr);`
- **Action:** It takes a single argument: a pointer that was previously returned by `malloc()` (or `calloc()`, `realloc()`).
- **Important:** The programmer does not need to specify the size of the memory block being freed; the memory allocation library tracks this internally.

  ```
  free(data); // 'data' is the pointer from the malloc() call above
  ```



3. Common Memory Errors

Manual memory management in C is powerful but error-prone. Many common bugs stem from incorrect usage of `malloc()` and `free()`.

> **Key Insight:** Just because a program compiles and runs does not mean it is correct. Memory errors can be subtle and may not cause a crash immediately, leading to unpredictable behavior.

- **Forgetting to Allocate Memory:** Trying to use a pointer before it has been initialized to point to valid, allocated memory. This often results in a **segmentation fault**.

  ```c
  // BUG: dst is an uninitialized pointer
  char *dst;
  ```

```c
strcpy(dst, "hello"); // Writes to a random, invalid memory location
```

- **Not Allocating Enough Memory (Buffer Overflow):** Allocating a buffer that is too small for the data being copied into it. A classic example is forgetting space for the null terminator (`\0`) in a string.

  ```c
  // BUG: Should be strlen(src) + 1
  char *dst = malloc(strlen(src));
  strcpy(dst, src); // Writes '\0' one byte past the end of the buffer
  ```

  This can corrupt adjacent memory or be exploited as a security vulnerability.

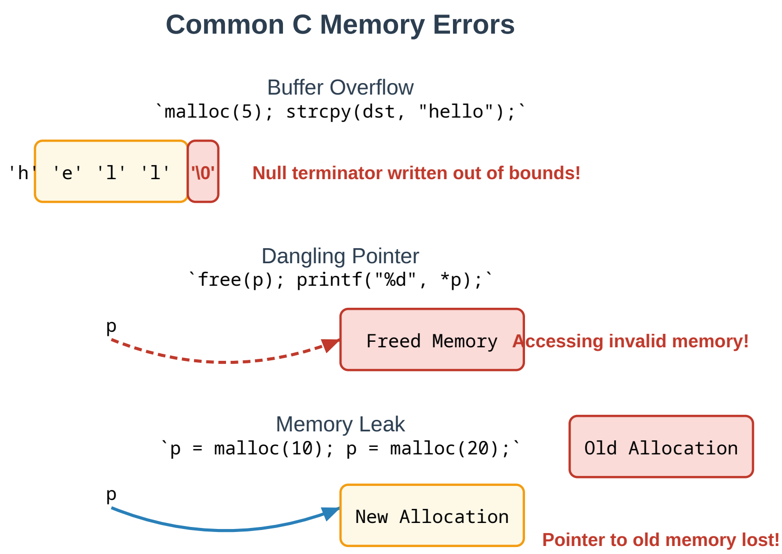
- **Forgetting to Initialize Allocated Memory:** `malloc()` returns a block of memory containing garbage values. Reading from this memory before writing to it first leads to an **uninitialized read**, causing unpredictable program behavior.

- **Forgetting to Free Memory (Memory Leak):** Failing to call `free()` on heap memory that is no longer needed. In long-running programs (like servers or the OS itself), this is a serious problem that can cause the application to eventually run out of memory and crash.

- **Freeing Memory Before You Are Done With It (Dangling Pointer):** After `free(p)` is called, the pointer `p` is now a "dangling pointer." It points to memory that is no longer valid. Using this pointer can lead to crashes or silent data corruption if that memory has been re-allocated for another purpose.

- **Freeing Memory Repeatedly (Double Free):** Calling `free()` on the same pointer more than once. This can corrupt the memory allocator's internal data structures, leading to undefined behavior.

- **Calling `free()` Incorrectly:** Passing an invalid pointer to `free()` (e.g., a pointer to the middle of an allocated block, or a pointer to a stack variable). This also leads to undefined behavior.



4. Underlying OS Support

`malloc()` and `free()` are **library calls**, not system calls. They manage memory within the process's virtual address space. To get more memory from the OS, the allocation library itself uses system calls.

- **`brk` and `sbrk`:** These system calls are used to change the location of the program's **break**, which marks the end of the heap segment. By moving the break, the heap can grow or shrink. These calls should not be used directly by application programmers.
- **`mmap()`:** An alternative way to get memory from the OS. It can create an **anonymous memory region** that is not backed by a file but by swap space. This memory can then be managed like a heap.

---

5. Other Useful Memory Calls

- **`calloc(num, size)`:** Allocates space for `num` items of `size` bytes each and **initializes the memory to zero**. This helps prevent uninitialized read errors.
- **`realloc(ptr, new_size)`:** Resizes a previously allocated memory block. It may allocate a new, larger region, copy the old data into it, free the old region, and return a pointer to the new one.