# Module 7: Dynamic Memory Allocation (Days 24-26)

## 1. Philosophy Focus: Knowledge Graph & Spaced Repetition

You know how to create arrays of a fixed size, like int my\_array[10];. But what if you don't know how many items you'll need when you write the program? This is the problem that **dynamic memory allocation** solves. This module connects your knowledge of pointers and structs to the **heap**, a pool of memory available for on-demand use. The review project will have you revisit and upgrade your to-do list, reinforcing these crucial memory management patterns.

### **DAY 24: malloc and free - The Basics**

**Goal:** Learn how to request memory from the operating system and how to return it when you're done.

**1. The Stack vs. The Heap**

* **The Stack:** This is where your local variables are stored. It's very fast and managed automatically. When a function is called, its variables are "pushed" onto the stack. When it returns, they are "popped" off. The size of everything on the stack must be known at compile time. int x; and char name[50]; live on the stack.
* **The Heap:** This is a large pool of memory that is available for your program to use "dynamically" (i.e., while it's running). You can request blocks of any size. It's more flexible but slightly slower. You, the programmer, are responsible for managing it.

**2. The Core Functions (#include <stdlib.h>)**

* void\* malloc(size\_t size): (Memory Allocate) Asks the OS for a block of memory of size bytes.
  + It returns a "generic" void\* pointer to the start of that block. You must "cast" this pointer to the type you want (e.g., int\*, struct Book\*).
  + If the OS cannot provide the memory, it returns NULL. **You must always check for NULL!**
* void free(void \*ptr): Takes a pointer that you got from malloc and returns the memory block back to the OS.
  + For every malloc, you MUST have a corresponding free. Failing to do so causes a **memory leak**.
  + After free(ptr), the pointer ptr is now "dangling." It still holds the address, but the memory is no longer yours to use. Accessing it is undefined behavior.

**3. Inductive Example: A Dynamic Array**

#include <stdio.h>  
#include <stdlib.h> // For malloc and free  
  
int main(void) {  
 int \*arr; // A pointer, which will hold the address of our array  
 int n = 5;  
  
 // Allocate memory for 5 integers on the heap.  
 // sizeof(int) gives the size of one int in bytes (usually 4).  
 arr = (int \*)malloc(n \* sizeof(int));  
  
 // ALWAYS check if malloc was successful.  
 if (arr == NULL) {  
 printf("Memory allocation failed!\n");  
 return 1; // Exit with an error code  
 }  
  
 printf("Memory allocated successfully. Address: %p\n", arr);  
  
 // We can now use 'arr' just like a regular array.  
 for (int i = 0; i < n; i++) {  
 arr[i] = i \* 10;  
 printf("arr[%d] = %d\n", i, arr[i]);  
 }  
   
 // CRUCIAL: When we are done with the memory, we must free it.  
 free(arr);  
 printf("Memory has been freed.\n");  
  
 return 0;  
}

**4. Day 24 Practice**

1. **Dynamic Struct:** Use malloc to allocate memory for a single struct User (from the previous module).
2. Check if the allocation was successful.
3. Assign values to its members using the arrow -> operator.
4. Print the members to verify.
5. free the memory.

### **DAY 25: realloc - Resizing an Allocation**

**Goal:** Learn how to change the size of an existing block of memory on the heap.

What if you allocate an array for 5 items, but suddenly you need space for 10? You could malloc a new, larger block, copy all the old data over, and then free the old block. This is so common that C provides a function to do it for you: realloc.

**1. void\* realloc(void \*ptr, size\_t new\_size)**

* Takes a pointer to an existing allocation (ptr) and a new\_size in bytes.
* It tries to expand the memory block. It might do this in place, or it might allocate a new, larger block, copy the old data, and free the old block automatically.
* It returns a pointer to the new, resized block. This new pointer **might be different** from the old one! You must always update your pointer with the return value.
* If it fails, it returns NULL, and the *original* block of memory is untouched.

**2. Inductive Example: A Growable Array**

#include <stdio.h>  
#include <stdlib.h>  
  
int main(void) {  
 int \*arr;  
 int size = 3;  
  
 // Start with space for 3 integers  
 arr = (int \*)malloc(size \* sizeof(int));  
 if (arr == NULL) { return 1; }  
  
 arr[0] = 10; arr[1] = 20; arr[2] = 30;  
 printf("Initial array at address %p\n", arr);  
  
 // Now we need more space. Let's resize to hold 5 integers.  
 int new\_size = 5;  
   
 // A temporary pointer is used as a safety measure.  
 int \*temp = realloc(arr, new\_size \* sizeof(int));  
  
 if (temp == NULL) {  
 printf("Failed to reallocate. Original data is safe.\n");  
 free(arr); // Clean up original memory  
 return 1;  
 }  
  
 // IMPORTANT: Update our main pointer only after success.  
 arr = temp;  
 printf("Resized array at address %p\n", arr); // Address may have changed!  
  
 // The old data is still there. We can add new data.  
 arr[3] = 40;  
 arr[4] = 50;  
  
 for (int i = 0; i < new\_size; i++) {  
 printf("%d ", arr[i]);  
 }  
 printf("\n");  
  
 free(arr);  
 return 0;  
}

**3. Day 25 Practice**

1. **Dynamic String Input:** Write a program that reads a single character at a time from the user until they press Enter (\n). Store the characters in a dynamically growing array (a string).
   * Start by malloc-ing a buffer of size 10.
   * Keep track of capacity and current length.
   * In a loop, if length == capacity, use realloc to double the capacity.
   * Add the character to the buffer.
   * When the loop finishes, don't forget to add a \0 at the end and free the memory.

### **DAY 26: Spaced Repetition - The Dynamic To-Do List**

**Goal:** Apply dynamic memory allocation to a real-world problem, reinforcing the patterns of malloc, realloc, and free.

1. Done-for-you Training Plan: Upgrade the To-Do List

Convert your to-do list from Day 22 (which used a fixed-size array) into one that can grow to hold any number of tasks.

**Project Requirements:**

1. In main, instead of struct Task task\_list[50];, you will start with a pointer: struct Task \*task\_list = NULL;.
2. You will also need int task\_count = 0; and int task\_capacity = 0;.
3. **The add\_task function:** This is the core of the project.
   * Its signature will need to change because it must be able to modify main's task\_list, task\_count, and task\_capacity variables. This is a perfect use case for passing pointers to pointers!  
     void add\_task(struct Task \*\*list, int \*count, int \*capacity, const char \*description)
   * **Inside add\_task:**
     + Check if the list needs to grow: if (\*count == \*capacity)
     + If it's full:
       - Calculate a new capacity. A good strategy is to start with a capacity of 4, and then double it each time you run out of space. int new\_capacity = (\*capacity == 0) ? 4 : \*capacity \* 2;
       - Use realloc to resize the list: struct Task \*temp = realloc(\*list, new\_capacity \* sizeof(struct Task));
       - Check for NULL.
       - Update the main list pointer and capacity: \*list = temp; and \*capacity = new\_capacity;
     + Now that you have space, add the new task to the list at the correct index: (\*list)[\*count] = ...
     + Increment the task count: (\*count)++;
4. In main, create a menu loop where the user can choose to add a task or list all tasks. When they add a task, call your add\_task function.
5. **At the very end of main, before exiting, you MUST free(task\_list);** to prevent memory leaks.