7. Dynamic Memory Allocation & Array

7.1 Introduction to Dynamic Memory

Up until now, we've used **static memory allocation** where array sizes must be known at compile time:

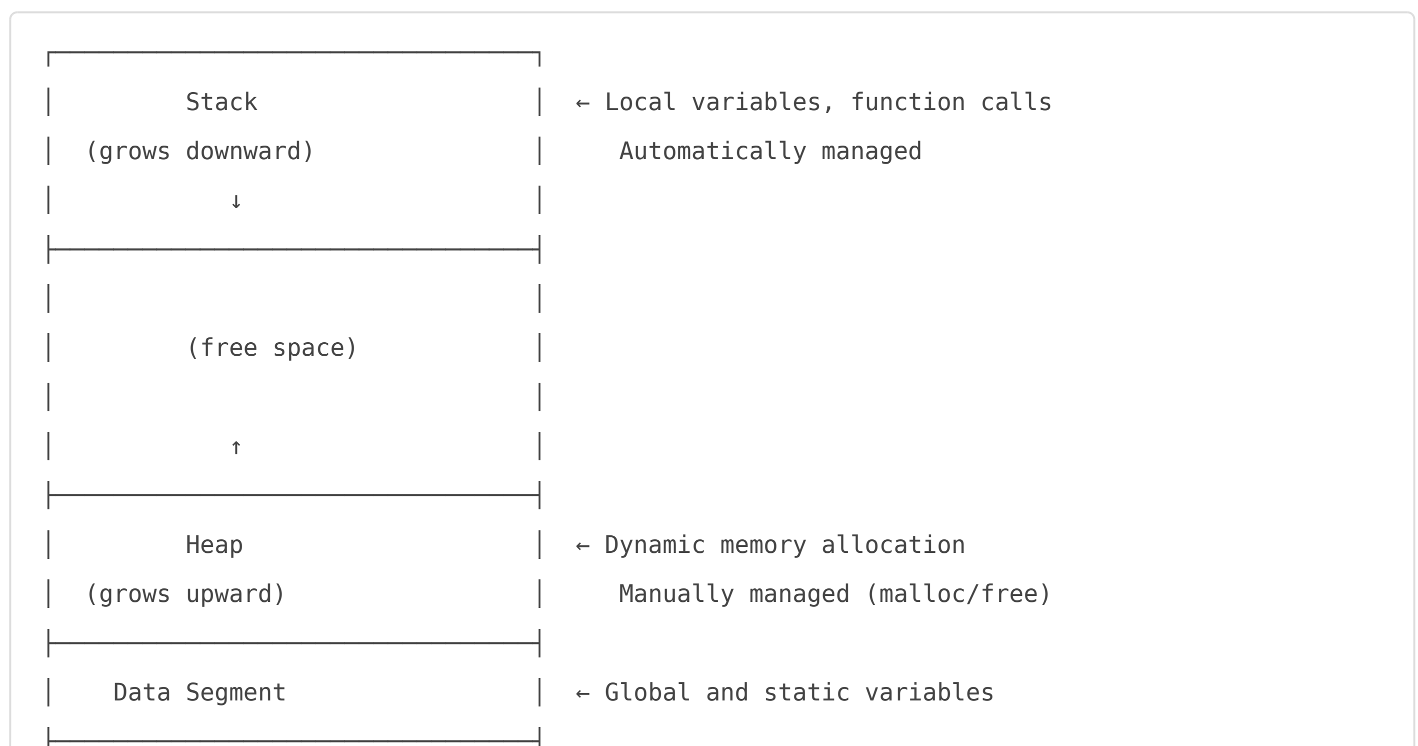
```
int arr[100];  // Size fixed at compile time
```

Problems with static allocation:

- Waste memory if you allocate too much
- Run out of space if you allocate too little
- Cannot adjust size during program execution

Dynamic memory allocation solves these problems by allowing you to allocate memory at runtime using special functions.

7.2 Memory Layout in C



```
|    Code Segment                  |  ← Program instructions
└──────────────────────────────────┘
```

7.3 Dynamic Memory Functions

C provides four main functions for dynamic memory management (defined in `<stdlib.h>`):

Function	Purpose	Syntax
`malloc()`	Allocates memory	`void* malloc(size_t size)`
`calloc()`	Allocates and initializes to zero	`void* calloc(size_t n, size_t size)`
`realloc()`	Resizes allocated memory	`void* realloc(void* ptr, size_t size)`
`free()`	Releases allocated memory	`void free(void* ptr)`

7.4 malloc() - Memory Allocation

Purpose: Allocates a block of memory of specified size (in bytes)

Syntax:

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

Returns:

- Pointer to allocated memory on success
- `NULL` if allocation fails

Example:

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

int main() {
    int *ptr;
    int n = 5;

    // Allocate memory for 5 integers
    ptr = (int*)malloc(n * sizeof(int));

    // Check if allocation was successful
```

```c
    if (ptr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Use the allocated memory
    for (int i = 0; i < n; i++) {
        ptr[i] = i * 10;
    }

    // Print values
    printf("Values: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }
    printf("\n");

    // Free the allocated memory
    free(ptr);
    ptr = NULL;  // Good practice

    return 0;
}
```

Important Notes:

- Always multiply by `sizeof(data_type)` to get correct byte size
- Always check if `malloc()` returns `NULL`
- Always cast the return value: `(int*)malloc(...)`
- Memory allocated by `malloc()` contains **garbage values**

7.5 calloc() - Contiguous Allocation

Purpose: Allocates memory and initializes all bytes to zero

Syntax:

```c
void* calloc(size_t n, size_t size);
```

Parameters:

- `n`: Number of elements

- `size`: Size of each element

Example:

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

int main() {
    int *ptr;
    int n = 5;

    // Allocate and initialize memory for 5 integers
    ptr = (int*)calloc(n, sizeof(int));

    if (ptr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Print values (all will be 0)
    printf("Initial values: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }
    printf("\n");

    free(ptr);
    ptr = NULL;

    return 0;
}

/* Output:
Initial values: 0 0 0 0 0
*/
```

malloc() vs calloc():

Feature	malloc()	calloc()
Parameters	1 (total bytes)	2 (number, size)

Feature	malloc()	calloc()
Initialization	Garbage values	All zeros
Speed	Faster	Slightly slower
Use case	When you'll initialize values	When you need zero-initialized memory

7.6 realloc() - Resize Memory

Purpose: Changes the size of previously allocated memory

Syntax:

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

Example:

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

int main() {
    int *ptr;
    int n = 5;

    // Initial allocation
    ptr = (int*)malloc(n * sizeof(int));
    if (ptr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Fill initial array
    for (int i = 0; i < n; i++) {
        ptr[i] = i + 1;
    }

    printf("Initial array (size %d): ", n);
    for (int i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }
    printf("\n");
```

```c
    // Resize to 10 elements
    n = 10;
    ptr = (int*)realloc(ptr, n * sizeof(int));

    if (ptr == NULL) {
        printf("Memory reallocation failed!\n");
        return 1;
    }

    // Fill new elements
    for (int i = 5; i < n; i++) {
        ptr[i] = i + 1;
    }

    printf("Resized array (size %d): ", n);
    for (int i = 0; i < n; i++) {
        printf("%d ", ptr[i]);
    }
    printf("\n");

    free(ptr);
    ptr = NULL;

    return 0;
}

/* Output:
Initial array (size 5): 1 2 3 4 5
Resized array (size 10): 1 2 3 4 5 6 7 8 9 10
*/
```

Important Notes about realloc():

- If `new_size` is larger, existing data is preserved, new space is uninitialized
- If `new_size` is smaller, data is truncated
- May move the block to a new location (address may change)
- If realloc fails, original pointer remains valid
- If `ptr` is `NULL`, behaves like `malloc()`

7.7 free() - Deallocate Memory

Purpose: Releases memory back to the system

Syntax:

```
void free(void* ptr);
```

Example:

```
int *ptr = (int*)malloc(100 * sizeof(int));


// Use the memory...


free(ptr);     // Release memory
ptr = NULL;    // Set to NULL to avoid dangling pointer
```

Important Rules:

1. Only free memory that was allocated with malloc/calloc/realloc
2. Free each block exactly once
3. Don't use memory after freeing it
4. Set pointer to NULL after freeing (good practice)

7.8 Dynamic Arrays - Complete Example

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


int main() {
    int n;
    int *arr;


    printf("Enter number of elements: ");
    scanf("%d", &n);


    // Allocate memory dynamically
    arr = (int*)malloc(n * sizeof(int));


    if (arr == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }
```

```
    // Input values
    printf("Enter %d integers:\n", n);
    for (int i = 0; i < n; i++) {
        scanf("%d", &arr[i]);
    }

    // Process: find sum and average
    int sum = 0;
    for (int i = 0; i < n; i++) {
        sum += arr[i];
    }
    double average = (double)sum / n;

    // Output results
    printf("\nArray elements: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\nSum: %d\n", sum);
    printf("Average: %.2f\n", average);

    // Free allocated memory
    free(arr);
    arr = NULL;

    return 0;
}
```

7.9 Dynamic 2D Arrays

Method 1: Array of Pointers (Rows can have different lengths)

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

int main() {
    int rows = 3, cols = 4;
    int **matrix;
```

```
    // Allocate array of row pointers
    matrix = (int**)malloc(rows * sizeof(int*));

    // Allocate each row
    for (int i = 0; i < rows; i++) {
        matrix[i] = (int*)malloc(cols * sizeof(int));
    }

    // Fill matrix
    int value = 1;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i][j] = value++;
        }
    }

    // Print matrix
    printf("Matrix:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%3d ", matrix[i][j]);
        }
        printf("\n");
    }

    // Free memory
    for (int i = 0; i < rows; i++) {
        free(matrix[i]);
    }
    free(matrix);
    matrix = NULL;

    return 0;
}
```

Method 2: Single Contiguous Block

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

```c
int main() {
    int rows = 3, cols = 4;
    int *matrix;

    // Allocate as single block
    matrix = (int*)malloc(rows * cols * sizeof(int));

    if (matrix == NULL) {
        printf("Memory allocation failed!\n");
        return 1;
    }

    // Fill matrix using formula: matrix[i][j] = matrix[i * cols + j]
    int value = 1;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            matrix[i * cols + j] = value++;
        }
    }

    // Print matrix
    printf("Matrix:\n");
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            printf("%3d ", matrix[i * cols + j]);
        }
        printf("\n");
    }

    // Free memory (single free needed)
    free(matrix);
    matrix = NULL;

    return 0;
}
```

7.10 Dynamic Memory Best Practices

1. Always Check for NULL

```
int *ptr = (int*)malloc(n * sizeof(int));
if (ptr == NULL) {
    fprintf(stderr, "Memory allocation failed!\n");
    exit(1);
}
```

2. Always Free Allocated Memory

```
// Allocate
int *data = (int*)malloc(100 * sizeof(int));

// Use data...

// Free when done
free(data);
data = NULL;
```

3. Don't Double Free

```
int *ptr = (int*)malloc(10 * sizeof(int));
free(ptr);
free(ptr);  // ERROR! Double free - undefined behavior
```

4. Don't Use After Free

```
int *ptr = (int*)malloc(10 * sizeof(int));
free(ptr);
ptr[0] = 5;  // ERROR! Using freed memory
```

5. Match Every malloc with free

```
void function() {
    int *ptr = (int*)malloc(100 * sizeof(int));
    // Use ptr...
    free(ptr);  // Don't forget!
}
```

7.11 Memory Leaks

A **memory leak** occurs when allocated memory is not freed:

Example of Memory Leak:

```
void badFunction() {
    int *ptr = (int*)malloc(1000 * sizeof(int));
    // Use ptr...
    return;  // BUG! Memory not freed - leaked!
}

int main() {
    for (int i = 0; i < 1000; i++) {
        badFunction();  // Leaks memory every iteration
    }
    return 0;
}
```

Fixed Version:

```
void goodFunction() {
    int *ptr = (int*)malloc(1000 * sizeof(int));
    // Use ptr...
    free(ptr);  // Properly freed
    return;
}
```

Another Common Leak:

```
int *ptr = (int*)malloc(100 * sizeof(int));
ptr = (int*)malloc(200 * sizeof(int));  // LEAK! Lost reference to first block
```

Fixed:

```
int *ptr = (int*)malloc(100 * sizeof(int));
free(ptr);  // Free first
ptr = (int*)malloc(200 * sizeof(int));  // Then allocate new
```

Revision #1
Created 2025-10-01 03:28:38 UTC by DS
Updated 2025-10-01 03:29:25 UTC by DS