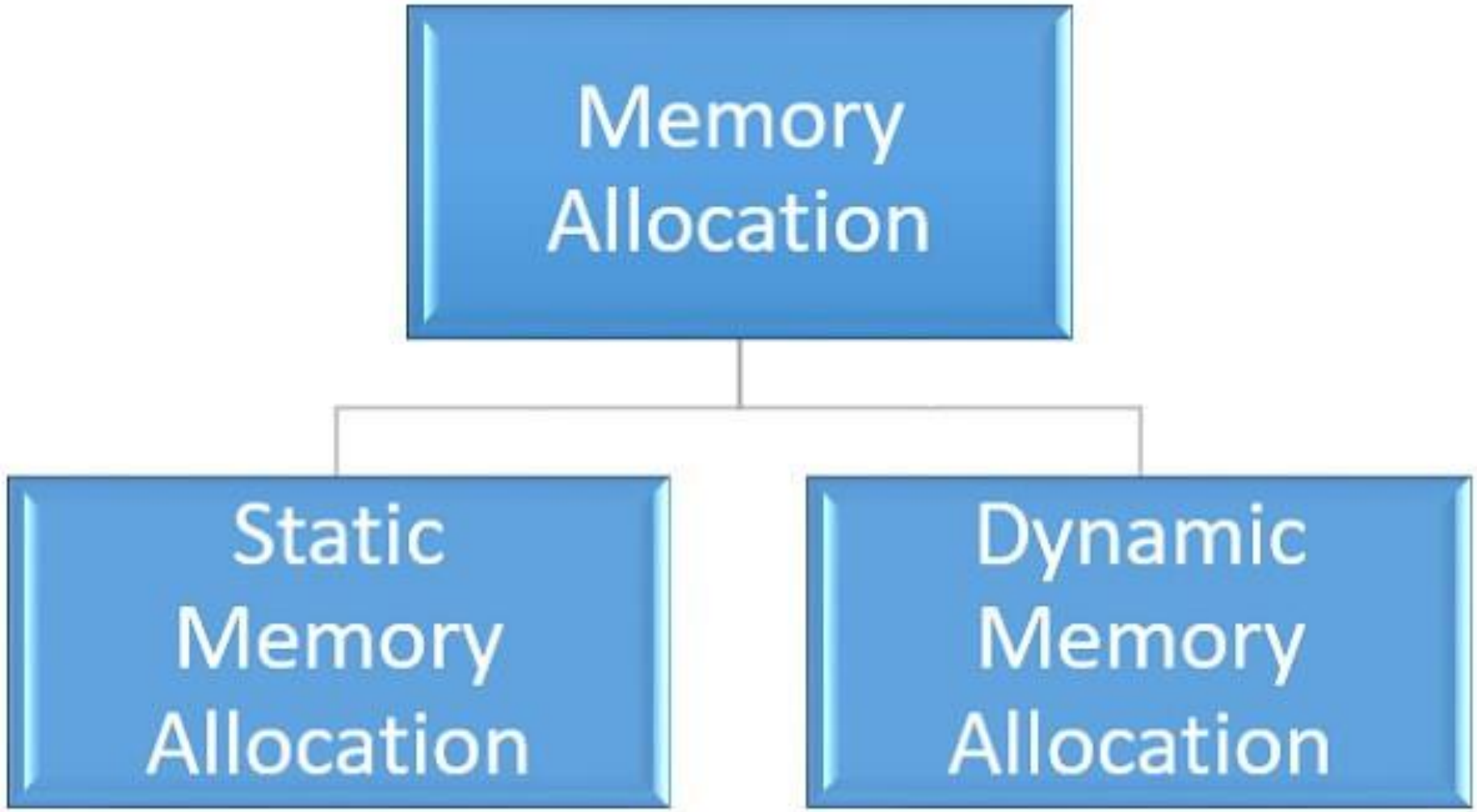
Computational Thinking for Structured Design

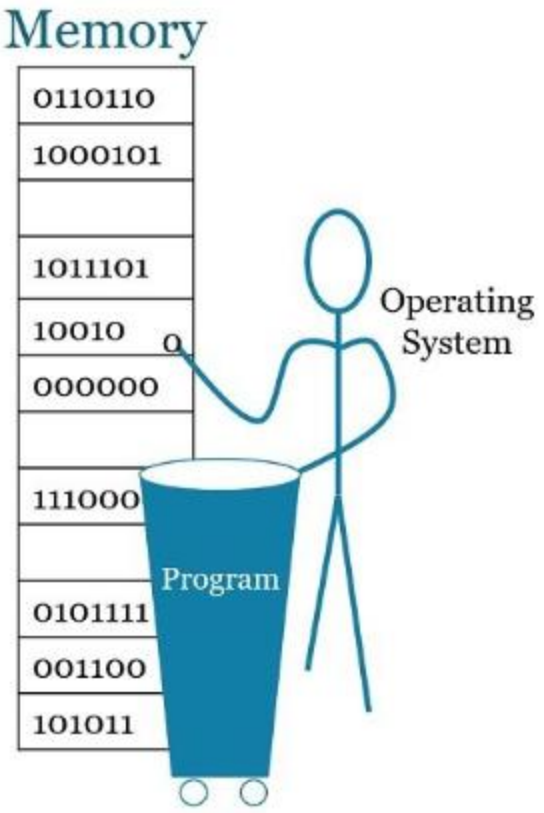
UNIT 1: Dynamic Memory Allocation:

- malloc()
- calloc ()
- realloc () and
- free ()
- Array of pointers
- Programing Applications
- Dangling Pointer


Memory Allocation
Static Memory Allocation
Dynamic Memory Allocation

Memory Allocation:

- Memory allocation is a process in computer programming where
- a program requests a certain amount of memory from
- the operating system to be used during its execution.
- There are two primary types of memory allocation:
  - **static** and **dynamic**.



Static Memory Allocation:

- **Definition:** Static memory allocation refers to the allocation of memory at compile-time. The size of the memory is determined before the program execution.
- **Variables:** Variables declared using keywords like int, float, or arrays with a specified size, fall under static memory allocation.
- **Storage Duration:** Memory is allocated during the program's compilation phase and remains constant throughout its execution.
- **Scope:** Variables usually have a local scope within the function or a global scope.

Dynamic Memory Allocation:

- **Definition:** Dynamic memory allocation involves allocating memory during the program's runtime.
- **Functions:** C provides functions like **`malloc()`**, **`calloc()`**, **`realloc()`**, and **`free()`** for dynamic memory management.
- **Storage Duration:** Memory is allocated on demand and can be resized during execution using appropriate functions.
- **Usage:** Particularly useful when the size of data structures is not known at compile-time or when memory needs to be managed efficiently during program execution.

  Example:

```
int *dynamicArray = (int *)malloc(5 * sizeof(int));
```

Importance of Dynamic Memory Allocation:

Variable Sizing: Dynamic memory allocation allows for flexibility in adjusting the size of data structures based on runtime conditions, unlike static memory allocation.

Efficient Memory Management: It enables efficient utilization of memory, as memory can be allocated or deallocated as needed during program execution.

Data Structures: Dynamic memory allocation is crucial when working with dynamic data structures such as linked lists, trees, and graphs, where the size may vary dynamically.

Reducing Memory Wastage: Memory is allocated only when required, reducing wastage and optimizing the usage of available resources.

Avoiding Stack Overflow: In cases where large amounts of memory are needed, dynamic allocation prevents potential stack overflow issues that may arise with limited stack space.

malloc Function:

1. **Definition:** malloc stands for "memory allocation" and is used in C to dynamically allocate a specified amount of memory during program execution.

Syntax:

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

The function returns a pointer of type `void*` which can be cast to any data type.

`size` represents the number of bytes to allocate.

malloc Function:

2. Example:

Allocating a Single Block of Memory:

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

int main() {
    int* dynamicInt;

    // Allocating memory for a single integer
    dynamicInt = (int*)malloc(sizeof(int));

    if (dynamicInt == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    // Assigning a value to the allocated memory
    *dynamicInt = 42;

    // Using the allocated memory
    printf("Value stored in dynamicInt: %d\n", *dynamicInt);

    // Deallocating the allocated memory
    free(dynamicInt);

    return 0;
}
```

Always check if the **memory allocation** was **successful** by verifying if the **returned pointer** is not **NULL.**

malloc Function:

2. Example:

Allocating memory for an array of integers:

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

int main() {
    int *dynamicArray;
    dynamicArray = (int *)malloc(5 * sizeof(int));

    if (dynamicArray != NULL) {
        // Memory allocation successful
        // Use dynamicArray for your operations
    } else {
        // Memory allocation failed
        printf("Memory allocation failed");
    }

    free(dynamicArray);  // Release allocated memory
    return 0;
}
```

Always check if the **memory allocation** was **successful** by verifying if the **returned pointer** is not **NULL.**

malloc Function:

3. Common Pitfalls and Best Practices:

Pitfalls:

- Forgetting to check if memory allocation was successful.
- Not releasing allocated memory using `free`.
- Accessing memory beyond the allocated size.

Best Practices:

- Always check the return value for `NULL` after calling `malloc`.
- Initialize the allocated memory to a known state using `calloc` or by setting the values explicitly.
- Release memory using `free` when it is no longer needed to avoid memory leaks.

malloc Function:

3. Common Pitfalls and Best Practices:

Pitfalls:

- Forgetting to check if memory allocation was successful.
- Not releasing allocated memory using `free`.
- Accessing memory beyond the allocated size.

Best Practices:

- Always check the return value for `NULL` after calling `malloc`.
- Initialize the allocated memory to a known state using `calloc` or by setting the values explicitly.
- Release memory using `free` when it is no longer needed to avoid memory leaks.

calloc Function:

1. **Definition:**
   - Allocates a block of memory for an array of elements, each of the specified size.
   - Initializes the memory to zero.
   - Returns a pointer to the beginning of the allocated block.

   Syntax:

   ```
   void* calloc(size_t num_elements, size_t element_size);
   ```

   The function returns a pointer of type `void*` which can be cast to any data type.
   `num_element` represents the number of elements.
   `element_size` represents the number of bytes to allocate for each element.

calloc Function:

2. Example:

Allocating memory for an array of integers:

Always check if the **memory allocation** was **successful** by verifying if the **returned pointer** is not **NULL.**

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

int main() {
    int* dynamicIntArray;

    // Allocating memory for an array of 5 integers
    dynamicIntArray = (int*)calloc(5, sizeof(int));

    if (dynamicIntArray == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    // Using the allocated memory
    for (int i = 0; i < 5; ++i) {
        dynamicIntArray[i] = i * 10;
    }

    // Displaying the values in the allocated array
    for (int i = 0; i < 5; ++i) {
        printf("%d ", dynamicIntArray[i]);
    }

    // Deallocating the allocated memory
    free(dynamicIntArray);

    return 0;
}
```

calloc Function:

3. Common Pitfalls and Best Practices:

Pitfalls:

Not Checking for Allocation Failure:

- Like malloc, calloc may return NULL if it fails to allocate memory.
- Always check the returned pointer for NULL to handle allocation failures.

Casting the Result:

- While it's common to cast the result of malloc, it's not necessary in C.
- However, when using calloc, the cast is usually retained for compatibility and clarity.

calloc Function:

3. **Common Pitfalls and Best Practices:**

Best Practices:

Initialize Memory to Zero: calloc initializes the allocated memory to zero. This is helpful when working with arrays or structures where zero-initialized memory is a common requirement.

Use sizeof for Element Size: When calculating the total size of memory to be allocated, use sizeof for the element size. This ensures that the allocation is correct regardless of the size of the data type.

Check for Allocation Failure: Always check if the allocated memory is NULL. If it is, handle the failure gracefully (e.g., by reporting an error or terminating the program).

Deallocate Memory Properly: As with malloc, memory allocated with calloc must be released using free to prevent memory leaks.

Avoid Redundant Casting: While casting the result of malloc or calloc is common, it is not strictly necessary in C. Modern C compilers implicitly perform the conversion.

How is `malloc` different from `calloc`?

Difference:

- **malloc** allocates a specified number of bytes of uninitialized memory.
- **calloc** allocates a specified number of blocks of memory, each of a specified size, and initializes all bits to zero.

Example:

```
int *arrMalloc = (int *)malloc(5 * sizeof(int)); // Uninitialized memory
int *arrCalloc = (int *)calloc(5, sizeof(int));   // Initialized to zero
```

realloc Function:

1. **Definition:**

   realloc stands for "reallocate" and is used in C to dynamically change the size of the memory block that was previously allocated by malloc, calloc, or realloc.

   Syntax:

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

   - `ptr:` Pointer to the previously allocated memory.
   - `size:` New size of the memory block.
   - Returns a pointer to the newly allocated memory.

realloc Function:

2. Example:

Reallocating memory for an array of integers:

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

int main() {
    // Allocate an initial block of memory
    int *originalArray = (int*)malloc(5 * sizeof(int));

    if (originalArray == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    // Use the allocated memory
    for (int i = 0; i < 5; ++i) {
        originalArray[i] = i * 2;
    }

    // Reallocate the memory to a larger size
    int newSize = 10;
    int *newArray = (int*)realloc(originalArray, newSize * sizeof(int));

    if (newArray == NULL) {
        fprintf(stderr, "Memory reallocation failed\n");
        // Free the original memory if reallocation fails
        free(originalArray);
        return 1;
    }

    // Use the reallocated memory
    for (int i = 5; i < newSize; ++i) {
        newArray[i] = i * 2;
    }


    // Free the memory when done
    free(newArray);

    return 0;
}
```

Scenarios for Dynamic Memory Reallocation:

Array Resizing:

When the size of an array needs to be changed dynamically based on runtime conditions.

Appending Elements:

Dynamically growing data structures like lists, where elements are added over time.

Optimizing Memory Usage:

Adjusting memory allocation to match the actual needs of the program, avoiding excessive or insufficient memory usage.

Can realloc be used to allocate memory initially?

- `realloc` can be used for initial memory allocation, but it's typically used to resize an existing allocation.
- When used for initial allocation, it behaves like `malloc`.
- However, using `malloc` directly is more idiomatic and clear for the purpose of initial allocation.

Free Function:

1. **Definition:**

   `free` is used in C to deallocate memory that was previously allocated by `malloc`, `calloc`, or `realloc`.

   Syntax:

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

   - `ptr`: Pointer to the memory block to be deallocated.

2. **Importance of Memory Deallocation:**

   Proper memory deallocation is essential for preventing memory leaks, where allocated memory is not released and becomes unavailable for future use.

Free Function:

3. Proper Memory Deallocation:

Freeing dynamically allocated memory:

```c
#include <stdlib.h>

int main() {
    int *dynamicArray = (int *)malloc(5 * sizeof(int));

    // ... (use dynamicArray)

    free(dynamicArray);  // Release allocated memory
    return 0;
}
```

Free Function:

4. **What happens if you forget to free allocated memory?**
   - Forgetting to free allocated memory can lead to memory leaks.
   - Memory leaks occur when a program continues to hold references to memory that is no longer needed without releasing it.
   - Over time, this can result in the depletion of available memory, causing the program to consume more resources than necessary and potentially leading to performance issues or crashes.

Array of Pointers:

1. **Definition:** An array of pointers in C is an array where each element is a pointer. Each pointer in the array can point to a specific value or address in memory.

Syntax:

```
data_type *array_of_pointers[size];
```

`data_type:` The type of data that the pointers in the array will point to.
`size:` The size of the array..

Array of Pointers:

2. Example:

Creating an array of pointers to integers:

```c
#include <stdio.h>

int main() {
    int num1 = 10, num2 = 20, num3 = 30;
    int *ptr_array[3];  // Array of pointers to integers

    ptr_array[0] = &num1;
    ptr_array[1] = &num2;
    ptr_array[2] = &num3;

    // Accessing values using pointers in the array
    printf("Value at index 0: %d\n", *ptr_array[0]);
    printf("Value at index 1: %d\n", *ptr_array[1]);
    printf("Value at index 2: %d\n", *ptr_array[2]);

    return 0;
}
```

Array of Pointers:

3. Applications in Data Structures:

Dynamic Data Structures:

Useful for creating dynamic data structures where the size of the structure can vary during runtime.

String Arrays:

An array of pointers to strings is commonly used to store and manipulate an array of strings.

Function Pointers:

An array of function pointers allows dynamic selection and invocation of functions.

Dangling pointer

- A dangling pointer is a pointer that continues to point to a memory location after the memory it points to has been deallocated or freed.
- Accessing the memory through a dangling pointer can lead to undefined behavior and various issues, as the memory might have been reused for other purposes.

Dangling pointer

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

int main() {
    // Allocate memory for an integer
    int *ptr = (int*)malloc(sizeof(int));

    if (ptr == NULL) {
        fprintf(stderr, "Memory allocation failed\n");
        return 1;
    }

    // Assign a value to the allocated memory
    *ptr = 42;

    // Deallocate the memory, making the pointer dangling
    free(ptr);

    // Accessing the memory through the dangling pointer
    //can lead to undefined behavior
    printf("Dangling Pointer Value: %d\n", *ptr);

    return 0;
}
```