1.1. STATIC AND DYNAMIC IMPLEMENTATION

Objective

- Explain the static and dynamic implementation of Data structure
- Interpret its advantages and disadvantages

Static implementation(compile time) of Data Structures

- Structures whose memory allocated **at the start of the program.**
- Structure whose memory allocation **is fixed and determined by the compiler at compile time.**
- Structure which do not change its size while programming
  - Eg: Arrays
  - Once to decide the size, they cannot be changed

Eg- float a[5] ,

allocation of 20 bytes to the array, i.e. 5*4 bytes

Static implementation(compile time) of Data Structures advantages

- **Compiler can allocate space during compilation**
  - In static data structures, the size of the data structure is known/predicted at compile time.
- **Easy to program**
  - Easier to implement. Accessing and modifying the element is straightforward
- **Easy to check for overflow**
  - static data structures have a fixed size, making it easier to check for overflow conditions
- **Array allows random access**
  - Arrays allow direct access to any element using its index,

Static implementation(compile time) of Data Structures disadvantages

- **Inefficient Use of Memory-**
  - **Can cause under utilization of memory in case of over allocation**
    - That is if you store less number of elements than the number for elements which you have declared memory.
    - Rest of the memory is wasted, as it is not available to other applications.
  - **Can cause Overflow in case of under allocation**
    - For float a[5];
    - No bound checking in C for array boundaries, if you are entering more than fives values,
    - It will not give error but when these extra element will be accessed it leads to undefined behaviour.

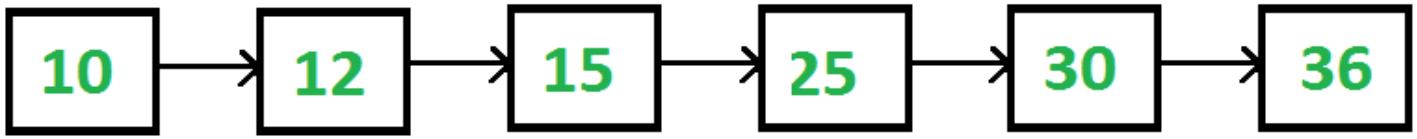
Static implementation(compile time) of Data Structures disadvantages

- **No reusability of allocated memory**
  - Once memory is allocated statically, it cannot be resized or reused, leading to potential wastage if the allocated size is too large

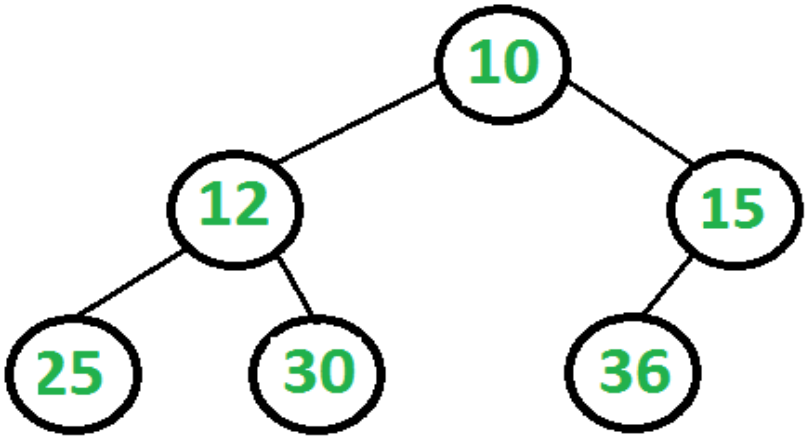
- **Difficult to guess the exact size of the data at the time of writing the program**
  - Estimating the exact size of the data at compile time is challenging, leading to risks of under-allocation (causing overflow) or over-allocation (wasting memory).

Dynamic implementation of Data structures

- Structures that can change size while programming Data structure
  - Linked lists, Binary tree



The above linked list represents following binary tree



Runtime or Dynamic Allocation

- **All Linked Data Structures are preferably implemented through dynamic memory allocation.**
- Dynamic data structures provide flexibility in **adding, deleting or rearranging data objects at run time.**
- Managed in C through a set of library functions:
  - malloc()
  - Calloc()
  - free()
  - Realloc()

Runtime or Dynamic Allocation

- Memory space required by Variables is **calculated and allocated during execution**
- Get Required chunk of memory at Run time or As the need arises
- Best Suited –
  - **When we do not know the memory requirement in advance**, which is the case in most of the real life problems.

Runtime or Dynamic Allocation

- **Efficient use of Memory**
  - Additional Space can be allocated at run time.
  - Unwanted Space can be released at run time.

- **Reusability of Memory space**

The malloc() fn

- **Allocates a block of memory in bytes**
- The user should **explicitly give the block size** needed.

The malloc() fn

- **Request to the RAM of the system to allocate memory**,
  - If request is granted returns a pointer to the first block of the memory
  - If it fails, it returns NULL

The malloc() fn

- **The type of pointer is Void, i.e. we can assign it any type of pointer.**
- Available in header file alloc.h or stdlib.h

The malloc() fn

- **Syntax-**

ptr_var=(type_cast*)malloc(size)

The malloc() fn

- **Syntax-**

ptr_var=(type_cast*)malloc(size)

- ptr_var = name of pointer that holds the starting address of allocated memory block
- type_cast* = is the data type into which the returned pointer is to be converted
- Size = size of allocated memory block in bytes

The malloc() fn

Eg-

```
int *ptr;
ptr=(int *)malloc(10*sizeof(int));
```

The malloc() fn

Eg-

```
int *ptr;
ptr=(int *)malloc(10*sizeof(int));
```

- Size of int=2bytes
- So 20 bytes are allocated,
- Void pointer is casted into int and assigned to ptr

The malloc() fn

Eg-

```
char *ptr;
ptr=(char *)malloc(10*sizeof(char));
```

The malloc() fn- Usage in Linked List

```c
// Define the node structure
typedef struct Node {
    int data;
    struct Node* next;
} Node;

// Function that create a new node
Node* createNode(int data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    if (newNode == NULL) {
        printf("Memory allocation failed\n");
        exit(1);
    }
    newNode->data = data;
    newNode->next = NULL;
    return newNode;
}
```

The calloc() fn

- Similar to malloc
  - It neds two arguments as against one argument in malloc() fn

Eg-

```
int *ptr;
ptr=(int *)calloc(10,2));
```

1st argument=no of elements

2nd argument=size of data type in bytes

The calloc() fn

- Available in header file alloc.h or stdlib.h

Malloc() vs calloc() fn

Initialization:

- **malloc() doesn’t initialize the allocated memory.**
  - If we try to access the content of memory block(before initializing) then we’ll get segmentation fault error(or garbage values).

- **calloc() allocates the memory and also initializes the allocated memory block to zero.**
  - If we try to access the content of these blocks then we’ll get 0.

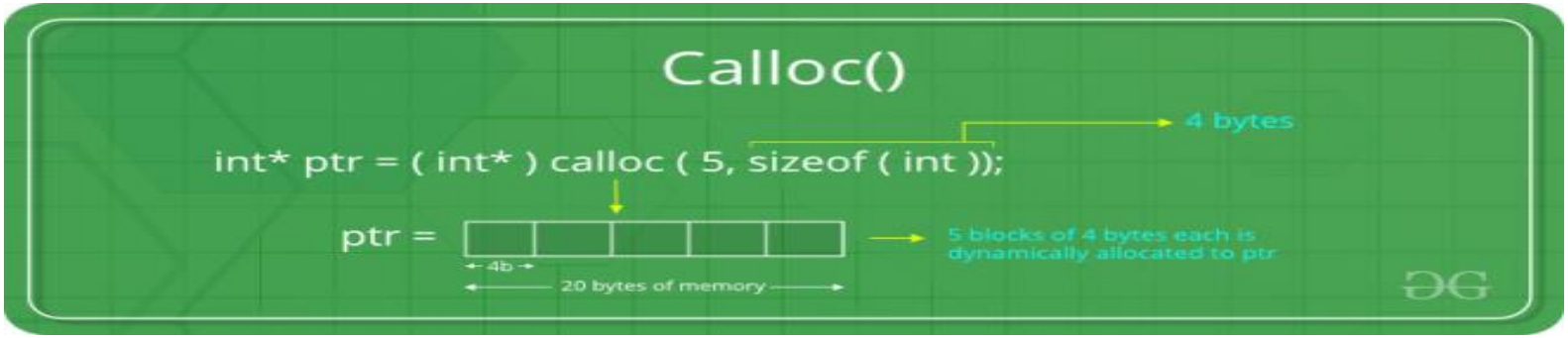
Malloc() vs callloc() fn

Malloc

- allocate a single large block of memory with the specified size.

Calloc

- allocate multiple blocks of memory
- dynamically allocate the specified number of blocks of memory of the specified type.



The free() fn

- Used to deallocate the previously allocated memory using malloc or calloc() fn
- **Syntax-**

  free(ptr_var);
- ptr_var is the pointer in which the address of the allocated memory block is assigned.
- Returns the allocated memory to the system RAM..

What happens when you don’t free memory after using malloc()

What happens when you don’t free memory after using malloc()

- The memory allocated using malloc() is not de-allocated on its own.
- So, “free()” method is used to de-allocate the memory.
- But the free() method is not compulsory to use.

What happens when you don’t free memory after using malloc()

- If free() is not used in a program
  - the memory allocated using malloc() will be de-allocated after completion of the execution of the program (included program execution time is relatively small and the program ends normally).

What happens when you don't free memory after using malloc()

- Still, there are some important reasons to free() after using malloc():

1) Use of free after malloc is a good practice of programming.
2) Efficient memory usage.
3) Prevents memory leaks

(A memory leak occurs when a program loses the reference to allocated memory without freeing it. This results in the memory being unusable for the duration of the program)

The realloc() fn

- To resize the size of memory block, which is already allocated using malloc.

- Used in two situations:
  - **If the allocated memory is insufficient for current application**
  - **If the allocated memory is much more than what is required by the current application**

The realloc() fn

- Syntax-

  ptr_var=realloc(ptr_var,new_size);

- ptr_var is the pointer holding the starting address of already allocated memory block.

- Available inn header file<stdlib.h>

- **Can resize memory allocated previously through malloc/calloc only.**

Eg of malloc() fn

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

int main()
{
   char *mem_allocation;
   mem_allocation = (char *) malloc( 20 * sizeof(char) );

   if( mem_allocation == NULL )
   {
      printf("Couldn't able to allocate requested memory\n");
   }
   else
   {
     strcpy( mem_allocation,“ABCD");
   }
```

Eg of malloc() fn

```
printf("Dynamically allocated memory content  : "%s\n", mem_allocation );
    mem_allocation=realloc(mem_allocation,100*sizeof(char));
    if( mem_allocation == NULL )
    {
        printf("Couldn't able to allocate requested memory\n");
    }
    else
    {
        strcpy( mem_allocation,"space is extended upto 100 characters");
    }
    printf("Resized memory : %s\n", mem_allocation );
    free(mem_allocation);
}
```

Advantages of Dynamic Data structures

- **Efficient Memory Utilization**:
  - Dynamic data structures only use the space needed at any time. Memory is allocated when needed and deallocated when not in use, leading to efficient memory utilization.
- **Adaptability and Flexibility**:
  - They can grow and shrink at runtime, making them adaptable to varying data sizes, which is crucial for applications with unpredictable or frequently changing data requirements.
- **Reclaiming Unused Memory**:
  - Storage that is no longer used can be returned to the system for other uses, helping manage overall memory usage and improving the application's efficiency.

Dis advantages of Dynamic Data structures

- **Difficult to progam**
  - More complex implementation and management compared to static data structures.
- **Can be slow to implement searches**
  - Slower access times due to pointer referencing and dynamic memory management.
- A linked list allows only serial access

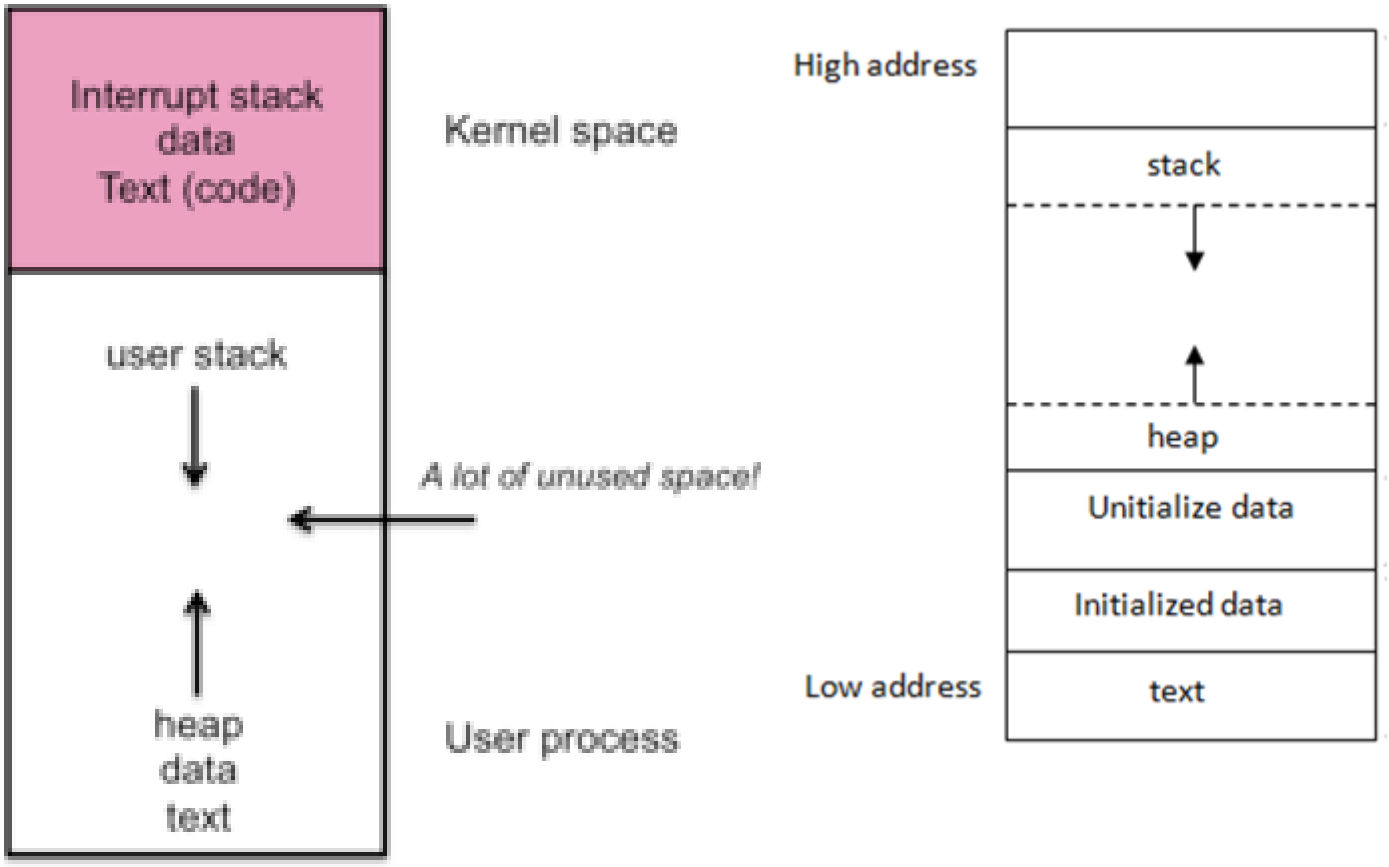
Which part of the memory is allocated when static memory allocation is used, for int,char,float,arrays,struct,unions…..?

Which part of the memory is allocated when malloc and calloc are called for any variable?

- ??

Stack and Heap

- **Stack and Heap both stored in the computer's RAM .**



Stack

- Basic type variables such as int, double, etc, and complex types such as arrays and structs. These variables are put on the **stack** in C.
- The stack is a region of memory that stores temporary variables created by each function. When a function is called, a stack frame is created which contains:
  - **Function parameters**: The values passed to the function.
  - **Local variables**: Variables declared within the function.
  - **Return address**: The address of the next instruction to execute after the function call.

- **There is a limit (varies with OS) on the size of variables that can be stored on the stack.**

Heap

- The heap is a region of memory used for dynamic memory allocation. Unlike the stack, the heap does not have a strict organization like LIFO. Memory can be allocated and freed at any time, in any order.

Stack vs Heap

Stack –

- **Size**: Generally small and fixed. The size is determined at the start of the program.
- **Speed**: Very fast access, as it is managed by the CPU.
- **Scope**: Variables on the stack are only accessible within the function they are declared in.
- **Lifetime**: Variables are destroyed when the function call completes.
- **Usage**: Used for function call management, local variables, and temporary data.

Heap

- **Size**: Can be much larger than the stack. It grows as needed (up to system limits).
- **Speed**: Slower access compared to the stack, because memory management on the heap is more complex.
- **Scope**: Variables on the heap are accessible from anywhere in the program as long as you have a pointer to them.
- **Lifetime**: Variables persist until explicitly deallocated using free() (or the program ends). They are not automatically destroyed when a function exits.
- **Usage**: Used for dynamic memory allocation where the size or lifetime of the data cannot be determined at compile time.