Introduction to Data Structures

The main objective of this chapter is to explain the importance of data structures. Before discussing data structures, we will see some basic concepts.

1.1 Variable

In computer science, a variable is used to store or hold data. This data is of any type, for example integer, float, character, string etc...

EX: int a=20;

Where ‘a’ is a variable of type integer representing the data, holds value of 20. This data is available to compiler during execution of the program.

However, variables are not feasible to store or handle huge amount of data. Consider an example, if you want to store and access student information such as student name, roll number, section, subjects and marks obtained for each student with single variable it is not feasible task. So, we need an organised mechanism to handle any correlated information. Thus, the concept of data structure has been introduced.

1.2 Data Structure

Based on the above discussion, we need some mechanism for manipulating data efficiently to solve problems. ***Data Structure*** is a way of storing and organizing data (that are related to each other) in a computer so it can be used efficiently.

Depending on the organisation of data elements, data structures are classified into two types.

1. Linear data structure: Data elements of this data structure are accessed in a sequential order but it is not compulsory to store all elements sequentially (Example of this data structure is Linked Lists).
   Examples: Arrays, Linked Lists, Stacks and Queues.
2. Non- linear data structure: Elements of this type of data structure are stored/ accessed in a non-linear order.
   Examples: Trees and graphs

1.2 Recursion

Any function which calls itself is called *recursive function*. Recursion is useful technique which was borrowed from mathematics. Recursive code is generally shorter and easier to write. Generally, loops are turned into recursive functions when they are compiled. Recursion is most useful for tasks that can be defined in terms of similar subtasks. For example: sort, search, and tree traversal problems.

1.3 Abstract Data Types

The primitive (system) data types like, integers, character, float, double etc are used to access data. It supports basic operations like addition, subtraction, multiplication etc. Suppose an example, to access data from array then we need to define index with square bracket notation. For user defined data type also we need to define operations to access data. The implementation for these operations actually is useful when we want to use them.

To simplify the process of solving large problems, we generally combine the data structure along with their operations and are called *Abstract Data Types (ADTs)*. An ADT consists of two parts.

1. Declaration of data
2. Declaration of operations.

Examples for ADTs include: stack, linked lists, queue, tree etc. To perform push and pop operation on stack, we need to define operations like: push an element into stack, pop an element from stack and check whether stack is full or empty etc. These implementation procedures come into picture when we want to use them.

1.4 Dynamic Memory Allocation

In C language, the sizes of an array to be specified (to store elements) at compile time. This may cause failure of the program or wastage of memory space. The process of allocating memory at run time is called ***dynamic memory allocation***. Although 'C' doesn't support this facility, there are four library routines known as *memory management techniques* under **stdlib.h**, which can be used for allocating and freeing memory during program execution. Memory allocation functions listed in below table.

Function	**Task**
Malloc	Allocates requested size of bytes and returns a pointer to the first byte of the allocated space.
Calloc	Allocates space for an array of elements, initializes them to zero and then returns a pointer to the memory.
Free	Frees the previously allocated memory.
Realloc	Modifies the size of previously allocated space.

Table: Memory Allocation Functions

Memory Allocation Process: Before discuss about these functions, let us look at the memory allocation process associated with a C program. Figure shows the conceptual view of storage of a C program in memory.

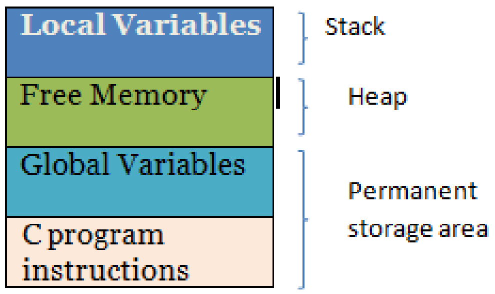


Figure: Storage of a C program

The program instructions, global variables and static variables are stored in region known as ***permanent storage area*** and local variables are stored in another area called ***stack***. The memory space between these two regions is available for dynamic allocation during execution of the program. The free memory region is called heap. The size of the heap keeps changing during execution, due to creation and deletion of variables that are local to functions and blocks. Therefore it is possible to encounter memory "*overflow*" during dynamic allocation process. In such situations memory allocation functions mentioned above return a NULL pointer (When they fail to locate enough memory requested).

Malloc (Allocating Block of Memory)

This name stands for *memory allocation*. The function malloc () reserves a block of memory of specified size and returns a pointer of type void(which can be casted into pointer of any type) to the first byte of the allocated space. Its contents can be accessedthrough pointer only.

Syntax:

ptr = (cast-type*) malloc(byte-size);

ptr is a pointer of type cast-type. The malloc returns a pointer of cast-type to an area of memory with size byte-size.

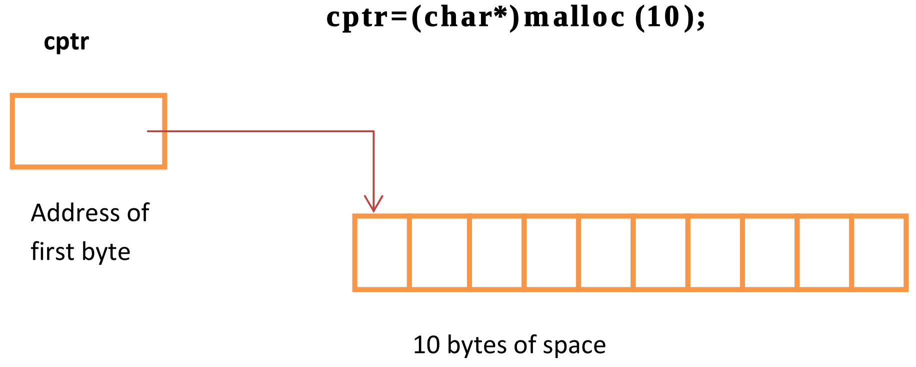
Example:

X= (int*)malloc (100*sizeof (int);

On successful execution of this statement, a memory space equivalent to "100 times the size of int" bytes is reserved and the address of the first byte of the memory allocated is assigned to the pointer **x** of type **int**.

Similarly, the statement allocates 10 bytes of space for the pointer **cptr**oftype **char.**

cptr=(char*)malloc (10);



We may also use malloc to allocate space for complex data types such as structures. Example:

st_var = (struct store*)malloc(sizeof(strcut store));

Where, **st_var**is a pointer of type**struct store.**

C Program:

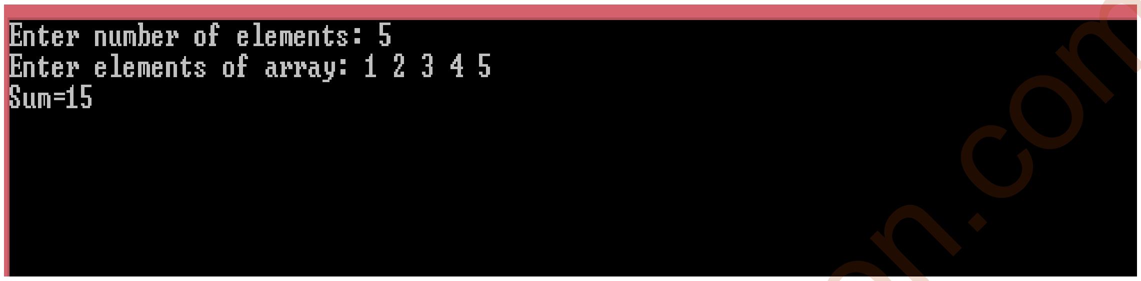
```
#include<stdio.h>
#include<stdlib.h>
#include<conio.h>

void main(){
intn,i,*ptr,sum=0;
printf("Enter number of elements: ");
scanf("%d",&n);
ptr=(int*)malloc(n*sizeof(int));  //memory allocated using malloc
if(ptr==NULL)
   {
printf("Error! memory not allocated.");
exit(0);
   }
printf("Enter elements of array: ");
for(i=0;i<n;++i)
   {
scanf("%d",ptr+i);
sum+=*(ptr+i);
```

```
   }
printf("Sum=%d",sum);
free(ptr);//deallocating the memory of an pointer

}
```

OutPut:



```
Enter number of elements: 5
Enter elements of array: 1 2 3 4 5
Sum=15
```

Calloc (Allocating Multiple Blocks of Memory)

Calloc is a another type of memory allocation function and is used for requesting multiple blocks of memory space at runtime for storing derived data types such as arrays and structures. While **malloc** allocates a single block of storage space, **calloc** allocates multiple blocks of storage, each of the same size, and sets all bytes to zero.

Syntax:

ptr = (cast-type*) calloc(n, elem-size);

The above statement allocates contiguous space for n blocks, each of size elem-size bytes. All bytes are initialized to zero. And a pointer returns to the first byte of the allocated region returned. If there is not enough space, a NULL pointer is returned.

Example: The following segment of program allocates space for a structure variable.

```
struct student{

        char name[25];

        float age;

        longintid_num;

};

typedefstruct student record;
```

```
record* st_ptr;

int class _size=30;

st_ptr=(record*)calloc(class_size,sizeof(record));

.......

.......
```

record is of type **struct student** having three members: name, age and id_num. The **calloc** allocates memory to hold data for 30 seconds; we must sure that requested memory has been allocated successfully before using **st_ptr**. This may be done as follows:

```
if(st_ptr==NULL){

        printf("Insuffient memory");

        exit(1);

}
```

C program:

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

void main(){
intn,i,*ptr,sum=0;
printf("Enter number of elements: ");
scanf("%d",&n);
ptr=(int*)calloc(n,sizeof(int));//memory allocated using calloc
if(ptr==NULL)
   {
printf("Error! memory not allocated.");
exit(0);
   }
printf("Enter elements of array: ");
for(i=0;i<n;++i)
   {
scanf("%d",ptr+i);
sum+=*(ptr+i);
   }
printf("Sum=%d",sum);
free(ptr);// deallocates the allocated memory

}
```

Free (Releasing used Space)

Dynamically allocated memory using either malloc() or calloc() must be explicitly released using this function. The release of storage is very important when the storage is limited.

free(ptr);

Where, **ptr** is a pointer to a memory block which has already been created by malloc or calloc. Use of an invalid pointer in the call may create problems and cause system crash.

Realloc (Altering the size of block)

If the allocated memory is insufficient and an additional space is required for more elements or if in other case, the memory allocated is much larger than necessary and we want to reduce it, in both cases we can change the memory size already allocated with the help of the function **realloc**. This process is also called "***reallocation of memory***" . For example, if the original allocation done by the statement

ptr= malloc(size);

Then, the reallocation of space done by the statement,

ptr=realloc(ptr, new_size);

The function allocates a new memory space of size *new_size* to the pointer variable *ptr* and returns a pointer to the first byte of the new memory block. The new size may be larger or smaller than original size.

*****Note**: The new memory block may or may not begin at the same place as the old one. In case it fails to locate the additional space in the same region then it will create the same in an entirely new region and move the contents of the old block into the new block. The function guarantees that the old data will remain intact.

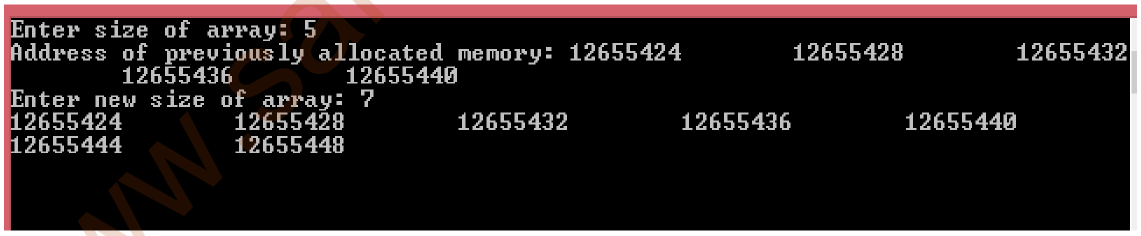
C Program:

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

void main(){
int *ptr,i,n1,n2;
printf("Enter size of array: ");
scanf("%d",&n1);
ptr=(int*)malloc(n1*sizeof(int));//initially allocated memory using malloc
printf("Address of previously allocated memory: ");
for(i=0;i<n1;++i)
printf("%u\t",ptr+i);
printf("\nEnter new size of array: ");
scanf("%d",&n2);
ptr=(int*)realloc(ptr,n2);//reallocate memory for previously initialized array.
for(i=0;i<n2;++i)
printf("%u\t",ptr+i);

}
```

Output:



```
Enter size of array: 5
Address of previously allocated memory: 12655424        12655428        12655432
        12655436        12655440
Enter new size of array: 7
12655424        12655428        12655432        12655436        12655440
12655444        12655448
```

****Difference between Static and Dynamic Memory Allocation

Sl. NO.	Static memory Allocation	Dynamic Memory Allocation
1.	In static memory allocation, user requested memory allocated at compile time.	In Dynamic memory allocation, user requested memory allocated while executing the program i.e. at run time.
2.	Memory size can't be modified while execution. Example: Array	Memory size can be modified while execution. Example: Linked Lists

***Difference between Malloc and Calloc:

Sl. No.	Malloc	Calloc
1.	This function used to allocate single block of memory of requested size.	This function used to allocate multiple blocks of memory of requested size..
2.	Malloc doesn't initialize the allocated memory (returns void pointer). Contains garbage values.	Calloc initializes the allocated memory with null values.
3.	Malloc take one argument which is the number of bytes to allocate.	Calloc takes two arguments, one being the number of elements and the other being the number of bytes allocated to each element.

4.	int *ptr;
	ptr = malloc(10 * sizeof(int));
	For the above, 10*4 bytes of memory only allocated in single block.
	Total = 40 bytes
		int *ptr;
		Ptr = calloc(10, 10 * sizeof(int));
		For the above, 10 blocks of memory will be created and each contains 10*4 bytes of memory.
		Total = 400 bytes