Practical-3

Dynamic Memory Allocation

Pointers with Structures in C

C allows programmers to create their data type by grouping different types together into one using structures. For example, if we want to store information about our class, each student variable should hold information about the student's name, roll number, and grades. No pre-defined data type in C can alone store all this information. For such cases where we want to store information that no data type can hold, we create our data types using structure to hold the required information. Different components of a structure are called members for example, in the above case, student name and roll number are members of the structure. Like every other data type, structure variables are stored in memory, and we can use pointers to store their addresses.

Structure pointer points to the address of the structure variable in the memory block to which it points. This pointer can be used to access and change the value of structure members. This way, structures and pointers in C can be used to create and access user-defined data types conveniently.

Syntax to Define a Structure

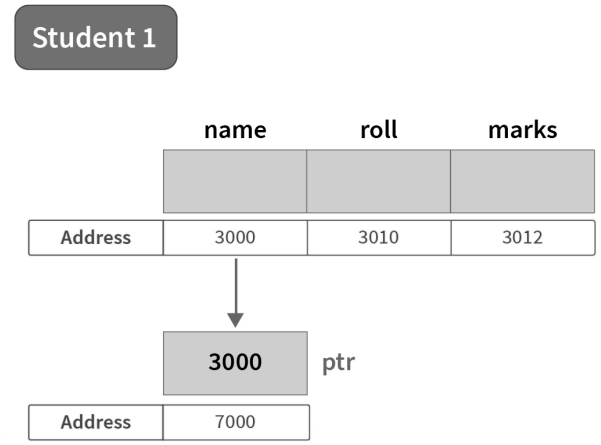
```
struct structure_name {
    data_type member_variable_1;
    data_type member_variable_2;
    // ...
    data_type member_variable_N;
};
…
struct structure_name structure_variable;
…
structure_variable.member_variable_1
```

Structure pointer in C is declared using the keyword struct followed by structure name to which the pointer will point to followed by pointer name. A structure pointer can only hold the address of a variable of the same structure type used in its declaration.

Declaration & Initialization of Structure Pointer

```
struct structure_name *structure_pointer;

struct structure_type *structure_pointer = &structure_variable;
```



Accessing Structure Member Using Pointer:

There are two ways to access the values of structure members using pointers -

- Using asterisk (*) and dot (.) operator with the structure pointer.
- Using membership or arrow (->) operator.

Example:

```
#include<stdio.h>
// create a structure Coordinate
struct Coordinate {
    // declare structure members
    int x,y;
};

int main() {
    struct Coordinate first_point;
    struct Coordinate *cp;
    cp = &first_point;
    (*cp).x = 5;
    (*cp).y = 10;
    printf("First coordinate (x, y) = (%d, %d)", (*cp).x,
(*cp).y);
    return 0;
}
```

Here, cp is a pointer that points to the structure variable first_point. This means dereferencing the pointer gives us the content of first_point. Hence, *cp and first_point are functionally identical. To access members of the structure dot operator can be used followed by the member name.

- (*cp).x refers to member x of first_point.
- (*cp).y refers to member y of first_point.

Note: Parentheses around the pointer is important because the precedence of dot operator is greater than indirection (*) operator.

```
#include<stdio.h>
struct Student {
    char name[30];
    int age;
    int roll_number;
};

int main() {
    struct Student student_1;
    struct Student *sp = &student_1;
     printf ("Enter the details of the Student (student_1)
\n");
    printf ("\tName: ");
    scanf ("%s", sp->name);
    printf ("\tAge: ");
    scanf ("%d", &sp->age);
    printf ("\tRoll Number: ");
    scanf ("%d", &sp->roll_number);
    printf ("\n Display the details of the student_1 using
Structure Pointer\n");
    printf ("\tName: %s\n", sp->name);
    printf ("\tAge: %d\n", sp->age);
    printf ("\tRoll Number: %d", sp->roll_number);
    return 0;
}
```

Another way to access structure members in C is using the (->) operator. Using this way, we don't need an asterisk and dot operator with the pointer. To access members of the structure using (->) operator we write pointer name with -> followed by the name of the member. Accessing members of the structure using the membership operator on the structure pointer makes code more readable when compared to the other approach.

Structure pointer in function arguments:

```
#include<stdio.h>
#include<math.h>
struct Coordinate {
    int x;
    int y;
};

float getDistance(struct Coordinate *X, struct Coordinate
*Y) {
    int x_diff = X->x - Y->x;
    int y_diff = X->y - Y->y;
     float distance = sqrt((x_diff * x_diff) + (y_diff *
y_diff));
    return distance;
```

```
}

int main() {
    struct Coordinate a,b;
    a.x = 5, a.y = 6;
    b.x = 4, b.y = 7;
    float distance = getDistance(&a, &b);
   printf("Distance between points (%d, %d) and (%d, %d) =
%.3f", a.x, a.y, b.x, b.y, distance);
    return 0;
}
```

Dynamic Memory Allocation:

There are two types of memory in our machine, one is Static Memory and another one is Dynamic Memory; both memories are managed by our Operating System. Our operating system helps us in the allocation and deallocation of memory blocks either during compile-time or during the run-time of our program.

When the memory is allocated during compile-time it is stored in the Static Memory and it is known as Static Memory Allocation, and when the memory is allocated during run-time it is stored in the Dynamic Memory and it is known as Dynamic Memory Allocation.

Static Memory Allocation

- Constant (Invariable) memory is reserved at compile-time of our program that can't be modified.
- It is used at compile-time of our program and is also known as compile-time memory allocation.
- We can't allocate or deallocate a memory block during run-time.
- Stack space is used in Static Memory Allocation.
- It doesn't provide reusability of memory while the program is running. So, it is less efficient.

Dynamic Memory Allocation

- Dynamic (Variable) memory is reserved at run-time of our program that can be modified.
- It is used at run-time of our program and is also known as run-time memory allocation.
- We can allocate and deallocate a memory block during run-time.
- Heap space is used in Dynamic Memory Allocation.
- It provides reusability of memory while the program is running. So, it is more efficient.

When we execute a C Program, it requires to reserve some memory in the machine to store its variables, functions, instructions and the program file itself. However, we also have a memory segment that we can dynamically use as much memory as the system has and that too during the run-time of a program.

- Stack Segment (Static Memory)
- Global Variables Segment (Static Memory)
- Instructions / Text Segment (Static Memory)
- Heap Segment (Dynamic Memory)

The amount of memory allocated for Stack, Global Variables, and Instructions / Text during compile-time is invariable and cannot be reused until the program execution finishes. However, the Heap segment of the memory can be used at run-time and can be expanded until the system's memory exhausts.

Instructions / Text

- Text outside the main() function is stored in the Static Memory.
- These instructions are stored during compile-time of a program.

Global Variables

- Global variables also known as static variables and can be declared by two methods,
- Using static keyword, ex. static int i = 0;
- Declaring variable outside main() or any other function.
- These variables are stored in the Static Memory during the compile-time of our program.

Stack

- Stack is a constant space (memory) allocated by our operating system to store local variables, function calls and local statements that are present in the function definition. It is the major part of the Static Memory in our system.
- There are some drawbacks of stack space as follows:
- The memory allocated for stack can't grow during the run-time of an application.
- We can't allocate or deallocate memory during execution.

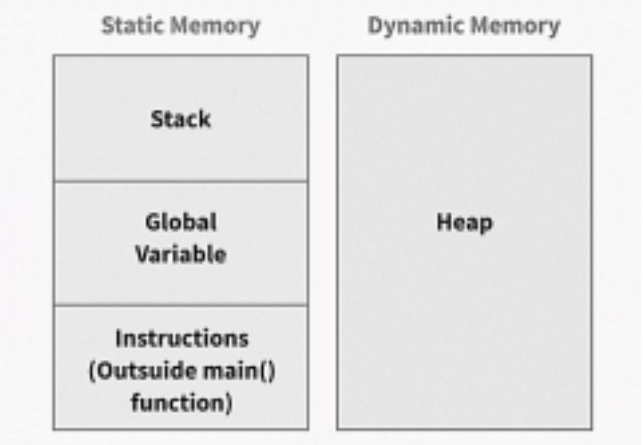
Heap

- Heap is the Dynamic Memory of our program, It can also be imagined as a big free pool of memory available to us. The space occupied by the heap section is not fixed, it can vary during the run-time of our program, and there are functions to perform allocation and deallocation of memory blocks during run-time.
- Heap space can grow as long as we do not run out of the system's memory itself, however it is not in the best interest of a programmer to exhaust the system's memory, so we need to be really careful while using the heap space in our program.

Dynamic Memory Allocation is a process in which we allocate or deallocate a block of memory during the run-time of a program. There are four functions malloc(), calloc(), realloc() and free() present in <stdlib.h> header file that are used for Dynamic Memory Allocation in our system. It can also be referred to as a procedure to use Heap Memory in which we can vary the size of a variable or Data Structure (such as an Array) during the lifetime of a program using the library functions.

The concept of **dynamic memory allocation in c language** *enables the C programmer to allocate memory at runtime*. Dynamic memory allocation in c language is possible by 4 functions of stdlib.h header file.

malloc()	allocates a single block of requested memory.
calloc()	allocates multiple blocks of requested memory.
realloc()	reallocates the memory occupied by malloc() or calloc() functions.
free()	frees the dynamically allocated memory.



malloc() function in C

- The malloc() function allocates a single block of requested memory.
- It doesn't initialize memory at execution time, so it has garbage value initially.
- It returns NULL if memory is not sufficient.
- The syntax of malloc() function is given below:
    - ptr=(cast-type*)malloc(byte-size)
    - int *ptr = (int *)malloc(sizeof(int));

calloc() function in C

- The calloc() function allocates multiple block of requested memory.
- It initially initialize all bytes to zero.
- It returns NULL if memory is not sufficient.
- The syntax of calloc() function is given below:
    - ptr=(cast-type*)calloc(number, byte-size)
    - void* calloc(size_t num, size_t size);

realloc() function in C

- If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.
- syntax of realloc() function.
    - ptr=realloc(ptr, **new**-size)
    - (cast-data-type *)realloc(ptr, new-size-in-bytes)

free() function in C

- The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.
- syntax of free() function.
    - free(ptr)
    - void free(void* ptr);

Example

```
// Program to calculate the sum of n numbers
entered by the user #include <stdio.h>
```

```
#include <stdlib.h>

int main() {
 int n, i, *ptr, sum = 0;

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

 ptr = (int*) malloc(n * sizeof(int));

 // if memory cannot be allocated
 if(ptr == NULL) {
 printf("Error! memory not allocated.");
 exit(0);
 }

 printf("Enter elements: ");
 for(i = 0; i < n; ++i) {
 scanf("%d", ptr + i);
 sum += *(ptr + i);
 }

 printf("Sum = %d", sum);

 // deallocating the memory
 free(ptr);

 return 0;
}
```

Output:

```
Enter number of elements: 3
Enter elements: 100
20
36
Sum = 156
```

Example

//Use of calloc

```
#include <stdio.h>
#include <stdlib.h> //
int main()
{
 /*int *ptr;
 int n;
 printf("Enter the size of the array you want to
create\n");  scanf("%d", &n);

 ptr = (int *)calloc(n , sizeof(int));
```

```
 for (int i = 0; i < n; i++)
 {
 printf("Enter the value no %d of this array\n",i);
scanf("%d", &ptr[i]);
 }

 for (int i = 0; i < n; i++)
 {
 printf("The value at %d of this array is %d\n",i, ptr[i]);
}

 //Use of realloc
 printf("Enter the size of the new array you want to
create\n");  scanf("%d", &n);

 ptr = (int *)realloc(ptr , n*sizeof(int));
 for (int i = 0; i < n; i++)
 {
 printf("Enter the new value no %d of this array\n",i);
scanf("%d", &ptr[i]);
 }

 for (int i = 0; i < n; i++)
 {
 printf("The new value at %d of this array is %d\n",i, ptr[i]);
}
*/
 free(ptr);
 return 0;
}
```

malloc() vs calloc()

malloc() method

- It is generally used to allocate a single memory block of the given size (in bytes) during the run-time of a program.
- It doesn't initialize the allocated memory block and contains some garbage value.
- malloc() takes a single argument i.e. the size of the memory block that is to be allocated.
- syntax: (cast-data-type *)malloc(size-in-bytes)
- Example : float *ptr = (float *)malloc(sizeof(float));

calloc() method

- It is generally used to allocate contiguous (multiple) blocks of memory of given size (in bytes) during the run-time of a program.
- It initializes all the allocated memory blocks with 0 (zero) value.
- calloc() takes two arguments i.e. the number of elements and the size of one element that are to be allocated.
- syntax: (cast-data-type *)calloc(num, size-in-bytes)
- Example : float *ptr = (float *)calloc(10, sizeof(float));

Let's assume we want to dynamically allocate a person structure. The person is defined like this:

```
typedef struct {
    char * name;
    int age;
} person;
```

To allocate a new person in the myperson argument, we use the following syntax:

```
person * myperson = (person *) malloc(sizeof(person));
```

Exercise

1. Define a structure Book which has members that include book_name, author_name, price and pages. Create a structure pointer variable which collects book information from the user and prints the book information for book_name starting with 'D'. Also create a function which displays all book information using pointers.
2. Write a program in C to reallocate previously allocated memory space. Print the address and value of original array and modified array [Take integer array]
3. Write a C program to accept a number n from the user and create an array of size n using dynamic memory allocation. Accept and store values in the array. Now take another number m from the user and revise the array size using dynamic memory allocation. Accept and store values in the revised array. Find the sum of all elements in both cases.
4. Write a program to store empID, name, age in a structure using DMA. The employee details should be stored for as many records as required. Display all the employee details. Use dynamic memory allocation.

Practice Exercise

- Write a program in C to dynamically allocate memory using malloc function to store N integer numbers entered by the user and then print the sum of all elements. Also free memory at the end of the program.
- Write a program to Rearrange array such that even index elements are smaller and odd index elements are greater.
  Input : arr[] = {2, 3, 4, 5}
  Output : arr[] = {2, 4, 3, 5} or arr[] = {3, 4, 2, 5}
  – *A simple solution is to sort the array in decreasing order, then starting from the second element, swap the adjacent elements.*
  – *An efficient solution is to iterate over the array and swap the elements as per the given condition.*
  At any point of time if i is even and arr[i] > arr[i+1], then we swap arr[i] and arr[i+1].
  Similarly, if i is odd and arr[i] < arr[i+1], then we swap arr[i] and arr[i+1].