Procedural Programming-II

UNIT-II

Memory Management

"Memory Management in C" refers to the process of controlling how a C program allocates and uses computer memory during execution, essentially deciding which parts of the memory are used for storing data related to your program, and when to reclaim that memory when it's no longer needed; this is primarily done using functions like malloc, calloc, realloc, and free from the <stdlib.h> header file, allowing for dynamic memory allocation at runtime, meaning you can allocate memory as needed during program execution rather than having to pre-define everything at compile time.

Static vs. Dynamic Memory:

Static Memory Allocation: Memory is allocated for variables when the program is compiled, with a fixed size throughout the program's execution.

Dynamic Memory Allocation: Memory is allocated at runtime using functions like malloc and calloc, etc. allowing you to adjust memory usage based on the program's needs.

Key Functions:

malloc(size): (memory allocation) Allocates a block of memory of size "size" bytes and returns a pointer to the first byte of that block.

syntax- ptr=(ptr-type*)malloc(size_in_bytes)

calloc(num, size): (**contiguous memory**)Similar to malloc, but initializes all allocated memory to zero.

syntax- ptr=(ptr-type*)calloc(n,size_in_bytes)

realloc(ptr, new_size): Resizes a previously allocated memory block pointed to by "ptr" to a new size "new_size".

free(ptr): Deallocates a previously allocated memory block pointed to by "ptr", returning it to the system for reuse.

Syntax: free(ptr)

Memory Leaks:

If you forget to free memory that is no longer needed, it can lead to a memory leak, where the program continues to use memory that it doesn't require, potentially causing performance issues.

Resizing and Releasing Memory

When your program comes out, operating system automatically release all the memory allocated

by your program but as a good practice when you are not in need of memory anymore then you

should release that memory by calling the function free.

Alternatively, you can increase or decrease the size of an allocated memory block by calling the

function realloc.

Dangling Pointer in C(*hanging, content which is deleted in memory location)*

A pointer pointing to a memory location that has been deleted (or freed) is called a dangling pointer. Such a situation can lead to unexpected behavior in the program and also serve as a source of bugs in C programs.

a pointer that points to an invalid memory address or one that is not in use anymore.

Example- int a

Int *Ptr=&a

When **a** is deleted then ptr is empty pointer i.e. dangling pointer.

Stack memory- Stack memory is a memory usage mechanism that allows the system memory to be used as temporary data storage that behaves as a first-in-last-out buffer.

Heap memory- The heap is a dynamic area of memory that grows towards the stack.
Heap memory is a region of computer memory used for dynamic memory allocation, allowing programs to request and release memory blocks during runtime, unlike stack memory which has a fixed size and follows a last-in, first-out (LIFO) order.

Pitfalls-

In C programming, common pitfalls include memory-related issues like buffer overflows, dangling pointers, and memory leaks, along with format string vulnerabilities and issues with string manipulatio

Pointer to an array:

Pointer to an array is also known as array pointer. We are using the pointer to access the components of the array.

```
int a[3] = {3, 4, 5 };

int *ptr = a;
```

We have a pointer ptr that focuses to the 0th component of the array. We can likewise declare a pointer that can point to whole array rather than just a single component of the array.

Syntax:

data type (*var name)[size of array];

Declaration of the pointer to an array:

// pointer to an array of five numbers

```
int (* ptr)[5] = NULL;
```

The above declaration is the pointer to an array of five integers. We use parenthesis to pronounce pointer to an array. Since subscript has higher priority than indirection, it is crucial to encase the indirection operator and pointer name inside brackets. Example:

```
// C++ program to demonstrate

// pointer to an array.

#include <iostream>

using namespace std;

int main()

{

    // Pointer to an array of five numbers

    int(*a)[5];

    int b[5] = { 1, 2, 3, 4, 5 };

    int i = 0;

    // Points to the whole array b
```

```
    a = &b;
    for (i = 0; i < 5; i++)
        printf("array is %d",*(*a+i));
    return 0;
}
```

Output:

1

2

3

4

5

Array of pointers:

"Array of pointers" is an array of the pointer variables. It is also known as pointer arrays.

Syntax:

```
int *var_name[array_size];
```

Declaration of an array of pointers:

```
int *ptr[3];
```

We can make separate pointer variables which can point to the different values or we can make one integer array of pointers that can point to all the values. Example:

```
// example of array of pointers.
#include <iostream>
using namespace std;
const int SIZE = 3;
int main()
{
    // creating an array
    int arr[] = { 1, 2, 3 };
    // we can make an integer pointer array to
    // storing the address of array elements
    int i, *ptr[SIZE];
    for (i = 0; i < SIZE; i++) {
        // assigning the address of integer.
        ptr[i] = &arr[i];
    }
    // printing values using pointer
```

```
    for (i = 0; i < SIZE; i++) {
        printf("value of %d ",arr[i]=*ptr[i]);
    }
}
```

Output:

Value of arr[0] = 1

Value of arr[1] = 2

Value of arr[2] = 3

Example: We can likewise make an array of pointers to the character to store a list of strings.

Function and pointers

In C, a function pointer is a type of pointer that stores the address of a function, allowing functions to be passed as arguments and invoked dynamically. It is useful in techniques such as callback functions, event-driven programs, and polymorphism (a concept where a function or operator behaves differently based on the context).

Let's take a look at an example:

```
#include <stdio.h>
int add(int a, int b) {
    return a + b;
}
int main() {
    // Declare a function pointer that matches
      // the signature of add() fuction
    int (*fptr)(int, int);
    // Assign to add()
    fptr = &add;
    // Call the function via ptr
    printf("%d", fptr(10, 5));
    return 0;
}
```

Output

15

Explanation: In this program, we define a function add(), assigns its address to a function pointer fptr, and invokes the function through the pointer to print the sum of two integers.

Passing Pointers to Functions

Passing the pointers to the function means the memory location of the variables is passed to the parameters in the function, and then the operations are performed. The function definition accepts these addresses using pointers, addresses are stored using pointers.

Arguments Passing without pointer

When we pass arguments without pointers the changes made by the function would be done to the local variables of the function.

Below is the C program to pass arguments to function without a pointer:

```
// C program to swap two values
//  without passing pointer to
// swap function.
#include <stdio.h>
void swap(int a, int b)
{
  int temp = a;
  a = b;
  b = temp;
}

// Driver code
int main()
{
  int a = 10, b = 20;
  swap(a, b);
  printf("Values after swap function are: %d, %d", a, b);
  return 0;
}
```

Output

Values after swap function are: 10, 20

Arguments Passing with pointers

A pointer to a function is passed in this example. As an argument, a pointer is passed instead of a variable and its address is passed instead of its value. As a result, any change made by the function using the pointer is permanently stored at the address of the passed variable. In C, this is referred to as call by reference.

Below is the C program to pass arguments to function with pointers:

```
// C program to swap two values
```

```
// without passing pointer to
// swap function.
#include <stdio.h>
void swap(int* a, int* b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

// Driver code
int main()
{
  int a = 10, b = 20;
  printf("Values before swap function are: %d, %d\n", a, b);
  swap(&a, &b);
  printf("Values after swap function are: %d, %d",  a, b);
  return 0;
}
```

Output

Values before swap function are: 10, 20

Values after swap function are: 20, 10

How to return a Pointer from a Function in C.

Pointers in C programming language is a variable which is used to store the memory address of another variable. We can pass pointers to the function as well as return pointer from a function. But it is not recommended to return the address of a local variable outside the function as it goes out of scope after function returns.

Program 1:

The below program will give segmentation fault since ‘A’ was local to the function:

```
// returning pointer from a function
#include <stdio.h>
int* fun()
{
    int A = 10;
```

```
    return (&A);
}
// Driver Code
int main()
{
    // Declare a pointer
    int* p;
    // Function call
    p = fun();
    printf("%p\n", p);
    printf("%d\n", *p);
    return 0;
}
```

Explanation:

The main reason behind this scenario is that compiler always make a stack for a function call. As soon as the function exits the function stack also gets removed which causes the local variables of functions goes out of scope.

Static Variables have a property of preserving their value even after they are out of their scope. So to execute the concept of returning a pointer from function in C you must define the local variable as a static variable.

Program 2:

```
// C program to illustrate the concept of
// returning pointer from a function
#include <stdio.h>
// Function that returns pointer
int* fun()
{
    // Declare a static integer
    static int A = 10;
    return (&A);
}
// Driver Code
int main()
{
    // Declare a pointer
```

```
    int* p;
    // Function call
    p = fun();
    // Print Address
    printf("%p\n", p);
    // Print value at the above address
    printf("%d\n", *p);
    return 0;
}
```

Output:

0x601038

10

Double Pointer:

double pointers are those pointers which stores the address of another pointer. The first pointer is used to store the address of the variable, and the second pointer is used to store the address of the first pointer. That is why they are also known as a pointer to pointer.

• Syntax of Double Pointer

The syntax to use double pointer can be divided into three parts:

• Declaration

A double pointer can be declared similar to a single pointer. The difference is we have to place an additional '*' before the name of the pointer.

type **name;

Above is the declaration of the double pointer with some name to the given type.

• Initialization

The double pointer stores the address of another pointer to the same type.

name=&single_ptr; // After declaration

type **name = &single_ptr; // With declaration

• Deferencing

To access the value pointed by double pointer, we have to use the deference operator * two times.

*name; // Gives you the address of the single pointer

**name; // Gives you the value of the variable it points to

Application of Double Pointers in C

Following are the main uses of pointer to pointers in C:

• They are used in the dynamic memory allocation of multidimensional arrays.

• They can be used to store multilevel data such as the text document paragraph, sentences, and word semantics.

• They are used in data structures to directly manipulate the address of the nodes without copying.

• They can be used as function arguments to manipulate the address stored in the local pointer.

Structure:

A structure in C is a derived or user-defined data type. We use the keyword struct to define a custom data type that groups together the elements of different types. The difference between an array and a structure is that an array is a homogenous collection of similar types, whereas a structure can have elements of different types stored adjacently and identified by a name.

We are often required to work with values of different data types having certain relationships among them. For example, a employee is described by its name (string), id (int), salary(float) etc. Instead of using four different variables, these values can be stored in a single struct variable.

Syntax declaration-

Struct <structure name/tag name> //struct is a keyword that use for user defined datatype.

{

Structure member 1;

Structure member 2;

Structure member 3;

……..

……..

……..

};

Example-

```
#include<stdio.h>
#include<conio.h>
struct  employee
{
int id_no;
char name[20];
char des[20];
float salary;
};
void main()
{
struct employee emp1;
printf("enter id");
scanf("%d",&emp1.id_no);
printf("enter name");
scanf("%s",&emp1.name);
printf("enter designation");
```

```
scanf("%s",&emp1.des);
printf("enter salary");
scanf("%f",&emp1.salary);
//
printf("id is %d",emp1.id_no);
printf("name is %s",emp1.name);
printf("des is %s",emp1.des);
printf("salary is %f",emp1.salary);
getch();
}
```

Note the following point while declaring a structure type:

The structure declaration statement must end with semicolon (';').

The structure declaration statement does not tell the compiler to reserve any space in memory.

Usually the structure should be declared above the main() function. It can also be declared in a separate header file.

To access the structure members a structure variable is required.

Accessing structure member:

In arrays we can access individual elements of an array using subscript .structure uses a different scheme .they use a dot (.) operator with the structure variable e.g. to refer to individual member of the structure we have to give emp1.name, emp1.salary, etc.

Array of structure-

An 'Array of Structures' in computer science refers to a data structure where multiple copies of a structure layout are stored in memory as elements of an array. This allows for efficient access to individual elements within each structure.

example-

```
#include<stdio.h>
#include<conio.h>
struct book{
char name[15];
int price;
int pages;
};
void main()
{
struct book b[100];
int i;
for(i=0;i<=99;i++)
{
printf("enter the name, price & pages");
scanf("%s %d %d",&b[i].name,&b[i].price,&b[i].pages);
}
for(i=0;i<=99;i++)
{
printf("%s %d %d",b[i].name,b[i].price,b[i].pages);
}
getch();
}
```

Pointer to structure-

A structure pointer is a pointer variable that stores the address of a structure. It allows the programmer to manipulate the structure and its members directly by referencing their memory location rather than passing the structure itself.

Syntax of Structure Pointer

The syntax of structure pointer is similar to any other pointer to variable:

struct struct_name *ptr_name;

Here, struct_name is the name of the structure, and ptr_name is the name of the pointer variable.

Example: Access members using Pointer

To access members of a structure using pointers, we use the -> operator.

```
#include <stdio.h>
struct person
{
  int age;
  float weight;
}
int main()
{
   struct person *personPtr, person1;
   personPtr = &person1;
   printf("Enter age: ");
   scanf("%d", &personPtr->age);
   printf("Enter weight: ");
   scanf("%f", &personPtr->weight);
   printf("Displaying:\n");
   printf("Age: %d\n", personPtr->age);
   printf("weight: %f", personPtr->weight);
   return 0;
}
```

In this example, the address of person1 is stored in the personPtr pointer using personPtr = &person1;.

Now, you can access the members of person1 using the personPtr pointer.

(.) Dot Operator

First method is to first dereference the structure pointer to get to the structure and then use the dot operator to access the member.

(->) Arrow Operator

C language provides an array operator (->) that can be used to directly access the structure member without using two separate operators.

Structure and function-

How to Pass a structure as an argument to the functions?

When passing structures to or from functions in C, it is important to keep in mind that the entire structure will be copied. This can be expensive in terms of both time and memory, especially for large structures

```
struct student {
    char name[50];
    int roll;
    float marks;
};


void display(struct student* student_obj)
{
  printf("Name: %s\n", student_obj->name);
    printf("Roll: %d\n", student_obj->roll);
    printf("Marks: %f\n", student_obj->marks);
}
int main()
{
    struct student st1 = { "Aman", 19, 8.5 };
    display(&st1);
    return 0;
}
```

Output

Name: Aman

Roll: 19

Marks: 8.500000

Nested structure-

One structure can be defined within another structure. This is known as nesting of structure, using this facility complex data type can be created.

Outer Structure: This is the main structure that contains other data types, including the nested structure.

Inner Structure (Nested Structure): This is a structure defined within the outer structure as one of its members.

Syntax:

```
    struct OuterStructure {
```

```
    // Other members of the outer structure
    data_type member1;
    struct InnerStructure {
        // Members of the inner structure
        data_type inner_member1;
        data_type inner_member2;
    } inner_struct_member;
    // More members of the outer structure
    data_type member2;
};
```

Typedef and structure:

The typedef is a keyword that is used to provide existing data types with a new name. The C typedef keyword is used to redefine the name of already existing data types. When names of datatypes become difficult to use in programs, typedef is used with user-defined datatypes, which behave similarly to defining an alias for commands.

Let's take a look at an example:

```
#include <stdio.h>
typedef int Integer;
int main() {
    // n is of type int, but we are using
     // alias Integer
    integer n = 10;
    printf("%d", n);
    return 0;
}
```

Output

10

Define an Alias for a Structure

```
#include <stdio.h>
#include <string.h>
// Using typedef to define an alias for structure
typedef struct Students {
    char name[50];
    char branch[50];
```

```
    int ID_no;
} stu;
int main() {
     // Using alias to define structure
    stu s;
    strcpy(s.name, "Geeks");
    strcpy(s.branch, "CSE");
    s.ID_no = 108;
    printf("%s\n", s.name);
    printf("%s\n", s.branch);
    printf("%d", s.ID_no);
    return 0;
}
```

Output

```
Geeks
CSE
108
```

Explanation: In this code, typedef is used to define an alias stu for the structure Students. The alias simplifies declaring variables of this structure type, such as st. The program then initializes and prints the values of the structure members name, branch, and ID_no. This approach enhances code readability by using the alias instead of the full struct students declaration.

Define an Alias for Pointer Type

```
#include <stdio.h>
// Creating alias for pointer
typedef int* ip;
int main() {
    int a = 10;
    ip ptr = &a;
    printf("%d", *ptr);
    return 0;
}
```

Output

```
10
```

Union-

union is a user-defined data type that can contain elements of the different data types just like structure. But unlike structures, all the members in the C union are stored in the same memory location. A union is a special data type available in C that allows to store different data types in the same memory location. You can define a union with many members, but only one member can contain a value at any given time. Unions provide an efficient way of using the same memory location for multiple purpose.

All the members of a union share the same memory location. Therefore, if we need to use the same memory location for two or more members, then union is the best data type for that. The largest union member defines the size of the union.

Defining a Union

Union variables are created in same manner as structure variables. The keyword union is used to define unions in C language.

Syntax

Here is the syntax to define a union in C language −

```
union [union tag]{
   member definition;
   member definition;
   ...
   member definition;
} [one or more union variables];
```

The "union tag" is optional and each member definition is a normal variable definition, such as "int i;" or "float f;" or any other valid variable definition.

At the end of the union's definition, before the final semicolon, you can specify one or more union variables.

Accessing the Union Members

To access any member of a union, we use the member access operator (.). The member access operator is coded as a period between the union variable name and the union member that we wish to access. You would use the keyword union to define variables of union type.

Syntax

Here is the syntax to access the members of a union in C language −

```
union_name.member_name;
```

Difference between Structure and Unions:

Parameter	Structure	Union
Definition	A structure is a user-defined data type that groups different data types into a single entity.	A union is a user-defined data type that allows storing different data types at the same memory location.
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union
Size	The size is the sum of the sizes of all members, with padding if necessary.	The size is equal to the size of the largest member, with possible padding.
Memory Allocation	Each member within a structure is allocated unique storage area of location.	Memory allocated is shared by individual members of union.
Data Overlap	No data overlap as members are independent.	Full data overlap as members shares the same memory.
Accessing Members	Individual member can be accessed at a time.	Only one member can be accessed at a time.

The main difference between struct and union is how they allocate and manage the memory struct allocate separated memory for each member interdentally. Union on the other hand allocate a shared memory space, meaning only one member can be used at a time making them useful for memory efficient program.