MODULE 5: Pointers, Preprocessor Directives and Data Structures

1. What is pointer? Explain with an example program.

Solution:

Pointer is a variable which contains the address of another variable.

Two important operators used with pointers are

a. **Ampersand operator (&)** – it is used to access address of a variable

b. **Dereferencing or Indirection Operator (*)** – it is used to declare pointer and access value of a variable referenced by pointer. The process of accessing value of a variable indirectly by pointer is called as indirection.

Consider the following Program to illustrate pointers and their usage.

```
#include<stdio.h>
int main()
{
        int var;        //declares an integer variable var
        int  *ptr;      // declares a pointer to integer variable
        ptr = &var;     // initializes pointer ptr to address of variable var
        var = 10;       // stores 10 into variable var
 //prints value of variable i.e. 10
        printf("Value of Variable = %d\n" , var);
 // *ptr accesses value of variable var referenced by pointer ptr hence, it is again 10
        printf("value of variable using pointer = %d\n", *ptr);
//&var retrieves address of variable var
        printf("Address of var = %u\n" , &var);                 // it prints address of var
// ptr contains address of variable var
        printf("Address of var using pointer = %u\n" , ptr);  // it prints address of var
// modify the value of variable var using pointer
        *ptr = *ptr * 5;          // *ptr = 10 * 5 = 50, hence it stores 50 in variable var
        printf("New value of var = %d\n", *ptr);       // prints 50
        return 0;
}
```

2. Define pointer? With example, explain declaration and initialization of pointer variables.

Solution:

Pointer is a variable which contains the address of another variable.

Two important operators used with pointers are

a. Ampersand operator (&) – it is used to access address of a variable

b. Dereferencing or Indirection Operator (*) – it is used to declare pointer and access value of a variable referenced by pointer. The process of accessing value of a variable indirectly by pointer is called as indirection.

Declaration of pointer:

Every pointer variable before it is used, it must be declared. Declaration tells the compiler the name of pointer, to which type of variable it points.

General Syntax of Pointer Declaration:

Datatype *pointer_name;

Where, pointer_name should be framed with the help of rule of identifier.

Datatype can be any valid C data type to which the pointer will point.

Ex:

```
int   *ptr;      // ptr is a pointer to integer
char *c;         // c is pointer to character
float *fp;       // fp is a pointer to floating point variable
```

Initialization of Pointer variable:

The process of assigning address of a variable to the pointer variable is called as initialization.

General Syntax of Initialization:

pointer_name = &variable;

Here, the address of variable is assigned to pointer.

Example:

```
int   var;
```

After declaration of variable var, it is allocated in some memory location (address) say 1000.

```
int   *p;
```

here, it declares pointer p which can contain the address of any integer variable.

```
p  = &var;
```

here, the address of variable var i.e 1000 is stored into pointer variable p. Therefore p contains 1000. Now onwards p acts as a pointer to var.

r-value: if *pointer_name is used on the right side of an assignment operator(=) in an expression then, it refers to the value of a variable referred by pointer. It is called as read-value.

l-value: if *pointer_name is used on the left side of an assignment operator(=) in an expression then, it refers to the address of a variable referred by pointer. It is called as location-value.

3. Explain functions and pointers with example.

Like variables, pointer variables can also be used as parameters in the functions. This mechanism is known as call-by-address or call-by-reference mechanism.

In this call-by-address mechanism, the address of actual parameter are copied to the formal parameters. Any modification to the formal parameters in the function definition will affect actual parameters.

For Example:

```
Datatype  *function_name(datatype   *ptr1, datatype *ptr2, …)
{
        //set of statements
```

```
}
```

In this example, the function accepts address of actual parameters and returns address of result obtained.

Example Program: To swap two numbers using pointers and functions

```
// Pointer is a variable which contains address of another variable
#include<stdio.h>
void  swap(int *x, int *y);
int main()
{
      int   p, q;
      printf("Enter the value for p and q\n");
      scanf("%d%d", &p, &q);
      printf("In Main: Before Function Call\n");
      printf("p=%d, q=%d\n", p, q);
                                  swap(1000,2500)
      swap(&p,&q);
      printf("In Main: After Function Call\n");
      printf("p=%d, q=%d\n", p, q);
      return 0;
}
void   swap(int *x, int *y)
{
      int temp;
      printf("In Swap: Before Exchange\n");
      printf("x=%d, y=%d\n", *x, *y);
      temp=*x;
      *x=*y;
      *y=temp;
      printf("In Swap: After Exchange\n");
      printf("x=%d, y=%d\n", *x, *y);
}
```

Copies address of p (&p) assumed that it is at address 1000 to formal parameter *x and address of q (&q) assumed that it is at address 2500 to formal parameter *y and transfers control to function definition

Output of the above program is

Enter the value for p and q
10 20
In Main: Before Function Call
p=10, q=20
In Swap: Before Exchange
x=10, y=20
In Swap: After Exchange
x=20, y=10
In Main: After Function Call
p=20, q=10

From the above output and flow shown with arrow and text description, we can understand that the address of actual parameters p and q are copied to formal parameters *x and *y respectively. From the bold text of output we can understand that the modification that we have done for formal parameters in function definition have affected(modified) actual parameters in calling function hence, the p and q value before function call and after function are different.

4. What are preprocessor directives? Explain different types.

Solution:

Preprocessor Directives: Preprocessor directives are the statements of C programming language which begins with pound (#) symbol. They are used in the program to instruct or tell the compiler to perform the following

- including external file
- defining the constants or symbols or macros and
- controlling the execution of statements conditionally or unconditionally.

Examples : #define , #include, #if, #endif , #ifdef and #else

Three types of pre-processor directives are:

a. **Macro Substitution Directives**: It is a process where an identifier in a program is replaced by a predefined string composed of one or more tokens. This is done by using #define statement. This statement usually known as a macro definition (or simply a macro) and it takes the following general form :

#define identifier string

Where, identifier is a name of a macro, symbol or constant and string is one or more tokens.

Example :

```
#define   COUNT  100        // defines COUNT with value 100
#define   FALSE   0         // defines FALSE with value 0
#define   PI  3.142         // defines PI with value 3.142
```

b. **File Inclusion Directives**: An external file containing functions or macro definitions can be included as a part of a program so that we need not rewrite those functions or macro definitions. This is achieved by the preprocessor directive #include.

General form:

#include<filename> OR **#include** "**filepath**"

Where, filename and filepath refers to the file to included in the program.

Example:

```
#include<stdio.h>      // includes stdio.h header file
#include<stdlib.h>     // includes stdlib.h header file
#include "TEST.C"      // includes TEST.c program file
#include "SYNTAX.C"        // includes SYNTAX.C program file
```

c. **Compiler Control Directives**: C preprocessor offers a feature known as conditional compilation which can be used to control a particular line or group of lines in a program.

Various directives used to control compiler execution are

#if - can be used for two-way selection
#ifdef - if defined
#ifndef - if not defined
#elif - else if directive
#else - else
#endif - end of if directive

Example 1: To ensure that a constant is defined in the program

```
#include<define.h>
#ifndef TEST
    #define TEST 1
#end if
```

In this example, #ifndef directive checks whether a constant TEST is defined or not. If not defined then it defines TEST with value 1 otherwise it will does nothing.

Example 2: To ensure that selected part of program is executed

```
#include<define.h>
#if num < 10
    // execute part A section
#else
    // execute part B section
#end if
```

In this example, #if directive tests the condition num<10 is true or false. If true, then it allows the compiler to execute part A section code otherwise it executes part B section code.

5. What is static and dynamic memory allocation? Give the differences between the same. Explain dynamic memory allocation functions.

Solution:

Static memory allocation: the process of allocating memory during compile time is known as static memory allocation.

Dynamic memory allocation: the process of allocating memory during run-time or execution time as required is known as dynamic memory allocation.

Difference between static and dynamic memory allocation

Static memory allocation	Dynamic memory allocation
Memory is allocated during compile time	Memory is allocated during run-time
Amount of memory allocated is fixed or constant	Amount of memory allocated can be changed
Wastage of memory may occur	No wastage of memory
Memory is allocated from stack segment	Memory is allocated from heap segment

Dynamic memory allocation functions:

The following functions are used to allocate and deallocate the memory from heap segment dynamically. Here, the first three functions are used to allocate memory and last one is used to deallocate the memory. These functions are defined in header file alloc.h or stdlib.h.

a. malloc()
b. calloc()
c. realloc()
d. free()

a. **malloc() function**: It is used to allocate a single block of memory of specified size dynamically and returns a pointer to the allocated block. The initial values in all the memory locations will be garbage values.

The general form for memory allocation using malloc is:

```
datatype *ptr =(data type *)malloc (requiredAmountofmemory);
```

Example 1 :

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

In this example, it allocates 10 * sizeof(int) = 10 * 4 bytes = 40 bytes of memory and starting address of block is assigned to pointer ptr.

Example 2 :

```
char  *ptr ;
ptr = (char *) malloc(5*sizeof(char));
```

In this example, it allocates 5 * sizeof(char) = 5 * 1 bytes = 5 bytes of memory and starting address of block is assigned to pointer ptr.

b. **calloc() function**: It is used to allocate multiple blocks of memory of specified size dynamically and returns a pointer to the allocated block. The initial values in all the memory locations will be ZERO.

The general form for memory allocation using calloc is:

```
datatype *ptr =(data type *)calloc (NoOfBlocks , requiredAmountofmemory);
```

Example 1 :

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

In this example, it allocates 10blocks of memory. Each block will be of size 4bytes since size of int is 4bytes in 32bit machine. Therefore, total of 40 bytes of memory is allocated and starting address of first block is assigned to pointer ptr.

Example 2 :

```
char  *ptr ;
ptr = (char *) calloc(5, sizeof(char));
```

In this example, it allocates 5blocks of memory. Each block will be size 1byte since size of char is 1byte. Therefore, total of 5bytes of memory is allocated and starting address of first block is assigned to pointer ptr.

c. **realloc() function**: It is used to increase or decrease the size of already allocated memory using malloc or calloc function and returns a pointer to the newly allocated block.

The general form for memory allocation using realloc is:

```
datatype *ptr =(data type *)realloc (ptrname,  NewAmountofmemory);
```

Example 1 :

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

In this example, it allocates 10 * sizeof(int) = 10 * 4 bytes = 40 bytes of memory and starting address of block is assigned to pointer ptr.

```
ptr  = (int *)realloc(ptr, 20);
```

in this example, already allocated memory of 40bytes pointed by ptr is decreased to 20bytes.

Example 2 :

```
char  *ptr ;
ptr = (char *) malloc(5*sizeof(char));
```

In this example, it allocates 5 * sizeof(char) = 5 * 1 bytes = 5 bytes of memory and starting address of block is assigned to pointer ptr.

```
ptr  = (char *) realloc(ptr, 10*sizeof(char));
```

In this example, already allocated memory of 5 bytes pointed by ptr is increased to 10bytes using realloc.

d. **free() –** it is used to deallocate or release the momory which was already allocated by using malloc, calloc or realloc function.

The general form using free is:

```
free(pointer_name);
```

where, pointer_name must be pointing to block of memory allocated using malloc, calloc or realloc function.

Example 1:

```
char  *ptr ;
ptr = (char *) malloc(5*sizeof(char));
free(ptr);        // releases memory pointed by ptr
```

Example 2:

```
char  *p ;
p = (int *) malloc(5*sizeof(int));
free(p);          // releases memory pointed by p
```

6. What are primitive and non-primitive data types? Explain.

Solution:

Primitive and Non Primitive Data Types: Data is recorded facts and figures that can be retrieved and manipulated in order to produce useful information / results. A data type is a type of value of the variable that is stored in particular memory location.

Data Types can be broadly classified into Primitive and Non Primitive Data Types

Primitive Data Types: The primitive data types are basic or fundamental data types that can be manipulated or operated directly by machine instructions.

The C language provides the following primitive data types:

- char data type – it is used to define characters
- int data type – it is used to define integer values
- float data type – it is used to define single precision floating point numbers
- double – it is used to define double precision floating point numbers
- void – it is empty data type and used as generic data type.

Non Primitive Data Types: are those that are not defined by the programming language but are instead created by the programmer. These are also known as derived data types which are derived from the existing fundamental data types.

Non Primitive Data Types are classified into two types

a. **Linear Data Types**
b. **Non Linear Data types**

a. **Linear Data Types**: Here the data elements are arranged in linear fashion. Examples:

- **stack** – it is a linear data structure, where insertion and deletion are done from only one end i.e top end. It is also known as Last-In-First-Out(LIFO) data structure.
- **Queue** – it is a linear data structure, where insertion is done from rear end and deletion is done from front end. It is also known as First-In-First-Out(FIFO) data structure.
- **Linked list** – it is a linear data structure where insertion and deletion are done linearly.

b. **Non Linear Data Types**: Here the data elements are arranged in nonlinear fashion.

Examples:

- **Trees** – it is a non-linear data structure, where data are arranged in non-linear fashion.
- **Graphs** – where set of vertices and edges are arranged non-linearly.
- **Maps -** where different states, districts, cities, etc are represented in non-linear fashion.

7. What is abstract data type? Explain with examples.

Soltuion:

Abstract data type (ADT) is a process of defining data items and various operations which can be performed on data items in abstract form. It hides the implementation details of how the data items are stored in memory and how the operations are actually implemented.

Example 1: Define ADT for stack

```
ADT stack
    Data items: top, array
    Operations: push(), pop(), display(), stackOverflow(), stackUnderflow()
End ADT
```

Example 2: Define ADT for Queue

```
ADT Queue
    Data items: rear, front, array
    Operations: insert_rear(), delete_front(), display(), queueOverflow(),
    queueUnderflow()
End ADT
```

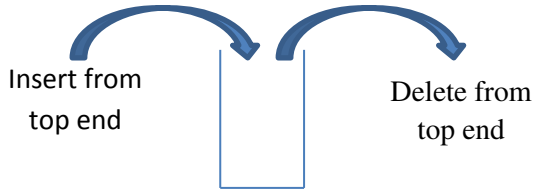
In the above example, we can observe that data items and various operations are listed but not detailed any information about how the data items are stored in memory, manipulated and operated. Similarly, how the operations are implemented is also not described. Hence, it hides the implementation details from the user.

8. Explain the following

a. Stack b. Queue c. Linked List d. Tree

a. Stack: It is a linear data structure, where insertion and deletion are done from only one end i.e., top end. It is also known as **Last-In-First-Out (LIFO)** data structure.

Pictorial View of Stack:



Various Applications of Stack:

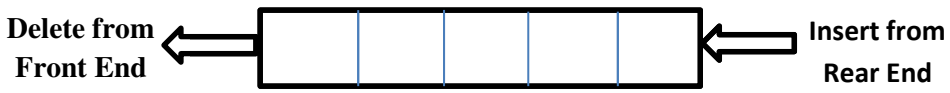
1. It is used to convert infix expression into postfix expression
2. It is used to evaluate postfix expression
3. It is used to store activation records during recursion in system
4. It is used to reverse string or numbers
5. It is used to store browsers data or history

Various Operations of Stack:

1. **Push():** this function is used to insert a data item from top of the stack.
2. **Pop():** this function is used to delete a data item from top of the stack.
3. **Display():** this function is used to list or print all the data items in the stack.
4. **stackOverflow():** this function is used to check whether stack is full or not.
5. **stackUnderflow():** this function is used to check whether stack is empty or not.

b. Queue: it is a linear data structure, where insertion is done from rear end and deletion is done from front end. It is also known as **First-In-First-Out(FIFO)** data structure.

Pictorial view of Queue

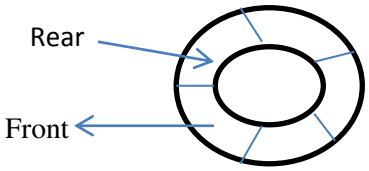


Types of Queue: There are three different types of queue based on how way they insert and delete from the queue.

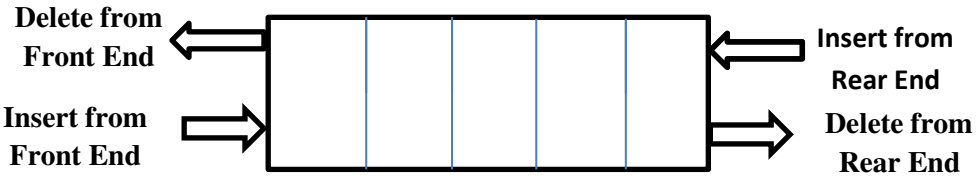
1. **Linear Queue:** where insertion and deletion is done linearly i.e. insertion is done from one end and deletion is done from other end.

Example:

2. **Circular Queue:** where insertion and deletion of data items is done in circular fashion.



3. **Double Ended Queue:** Where insertion and deletion is done from both the ends i.e., insertion can be done from both rear end and front end, similarly deletion can also be done from both ends.



Applications of Queue:

i. It is used in implementing job scheduling
ii. It is used in implementing CPU scheduling
iii. It is used in implementing Disk scheduling
iv. It is used in implementing Input and Output buffe
v. It is used in servicing printing requests of printer

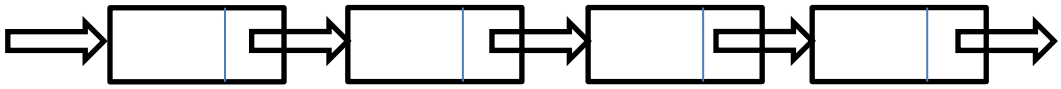
Various Operations of Queue:

1. **Insert_Rear()**: this function is used to insert a data item at rear end of the queue.
2. **Delete_Front()**: this function is used to delete a data item from front end of the queue.
3. **Display()**: this function is used to list or print all the data items in the queue.
4. **QueueOverflow()**: This function checks whether queue is full or not.
5. **QueueUnderflow()**: This function checks whether queue is empty or not.

c. Linked List:

It is a linear data structure, where insertion and deletion are done in linear fashion. It is a collection of data items, where each data item is linked to another. It is a chain of nodes.

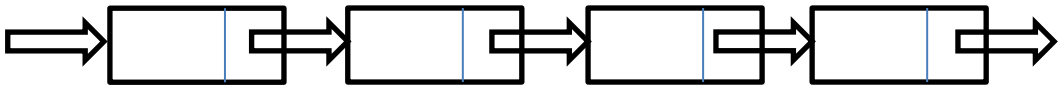
Pictorial view of linked list:



Types of Linked List:

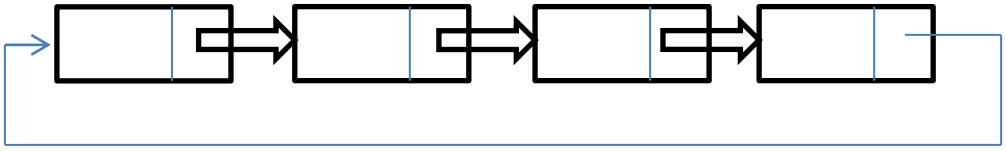
1. **Linear Linked List:** It is a linear singly linked list, where each node contains a pointer to the next node.

Example:



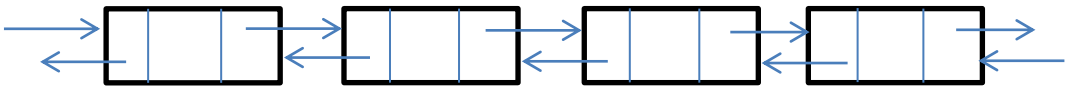
2. **Circular Linked list:** It is a linear data structure, where each node contains a pointer to the next node and last node contains a pointer to the first node.

Example:



3. **Doubly linked list:** It is a linear data structure, each node contains a pointer to the next node as well as to the previous node.

Example



Applications of Linked List:

1. It is used for implementing polynomials
2. It is used for implementing sparse matrix
3. It is used for implementing stack and queue

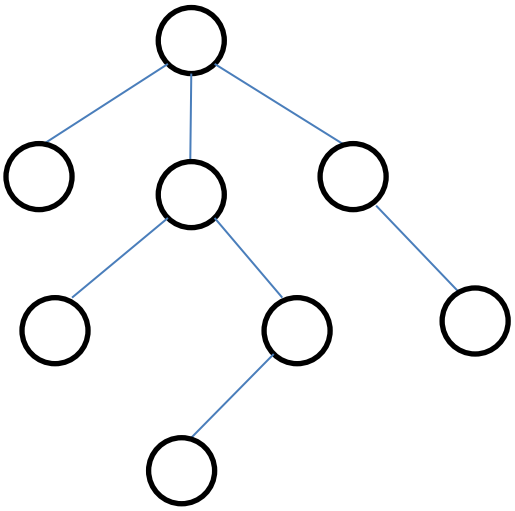
4. It is used for implementing SYMTAB and OPTAB in system software
5. It is used in implementing adjacency list representation of graphs.
6. It is used in implementing reverse of a number and string

Various operations of linked list:

1. **Insert(): ** this function can be used insert a data item at front of the list, at particular position of the list, or at end of the list.
2. **Delete(): ** this function can be used Delete a data item from front of the list, from a particular position of the list, or from end of the list.
3. **Search(): ** this function is used to search or find a particular data item in the list.
4. **Display(): ** this function is used to list or print all the data items in the list.

d. Tree: Tree is a non-linear data structure, where the data are arranged in hierarchical fashion.

Pictorial View of Tree:



Types of Tree:

1. Binary Tree
2. Binary Search Tree
3. RED-BLACK Tree
4. AVL Tree
5. B Tree
6. B+ Tree

Applications of Linked List:

1. It is used for implementing various data compression techniques
2. It is used for implementing routing table in networks
3. It is used for implementing graphs and maps

4. It is used for implementing social networks
5. It is used for implementing expression parser or tree
6. It is used for implementing evaluation of expressions
7. It is used in database such as indexing

Various operations of linked list:

1. **Insert():** This function is used insert a data item in the tree
2. **Delete():** This function is used Delete a data item from the tree
3. **Search():** This function is used to search or find a particular data item in the tree.
4. **Display():** This function is used to list or print all the data items in the tree.