

PROGRAMMING AND DATA STRUCTURES

**ARRAYS & LINKED LIST**

CHAPTER

3

**1. POINTERS**

The pointer in C language is a variable which stores the address of another variable. This variable can be of type int, char, array, function, or any other pointer. The size of the pointer depends on the architecture.

Consider the following example to define a pointer which stores the address of an integer.

**int** n = 10;

**int**\* p = &n; // Variable p of type pointer is pointing to the address of the variable n of type integer.

**1.1. Declaring a pointer**

The pointer in c language can be declared using \* (asterisk symbol). It is also known as an indirection pointer used to dereference a pointer.

int \*a;  //pointer to int

char \*c;   //pointer to char

Diagram

Description automatically generated

To use pointers in C, we must understand below two operators.

• To access address of a variable to a pointer, we use the unary operator **&**(ampersand) that returns the address of that variable.

• One more operator is unary \* (Asterisk) which is used for two things :

To declare a pointer variable: When a pointer variable is declared in C/C++, there must a \* before its name.

To access the value stored in the address we use the unary operator (\*) that returns the value of the variable located at the address specified by its operand.

**How to Use Pointers**

There are a few important operations, which we will do with the help of pointers very frequently. **(a)** We define a pointer variable, **(b)** assign the address of a variable to a pointer and **(c)** finally access the value at the address available in the pointer variable. This is done by using unary operator **\*** that returns the value of the variable located at the address specified by its operand. The following example makes use of these operations −

#include <stdio.h>

int main () {

   int  var = 20;   /\* actual variable declaration \*/

   int  \*ip;        /\* pointer variable declaration \*/

   ip = &var;  /\* store address of var in pointer variable\*/

   printf("Address of var variable: %x\n", &var  );

   /\* address stored in pointer variable \*/

   printf("Address stored in ip variable: %x\n", ip );

   /\* access the value using the pointer \*/

   printf("Value of \*ip variable: %d\n", \*ip );

   return 0;

}

When the above code is compiled and executed, it produces the following result −

Address of var variable: bffd8b3c

Address stored in ip variable: bffd8b3c

Value of \*ip variable: 20

**NULL Pointers**

It is always a good practice to assign a NULL value to a pointer variable in case you do not have an exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries. Consider the following program

**1.**

#include <stdio.h>

int main () {

   int  \*ptr = NULL;

   printf("The value of ptr is : %x\n", ptr  );

   return 0;

}

When the above code is compiled and executed, it produces the following result −

The value of ptr is 0

In most of the operating systems, programs are not permitted to access memory at address 0 because that memory is reserved by the operating system. However, the memory address 0 has special significance; it signals that the pointer is not intended to point to an accessible memory location. But by convention, if a pointer contains the null (zero) value, it is assumed to point to nothing.

To check for a null pointer, you can use an 'if' statement as follows −

if(ptr)     /\* succeeds if p is not null \*/

if(!ptr)    /\* succeeds if p is null \*/

Example-

**int** number=50;

**int** \*p;

p=&number;    //stores the address of number variable

printf("Address of p variable is %x \n",p);   // p contains the address of the number therefore

printing p gives the address of number.

printf("Value of p variable is %d \n",\*p);     // As we know that \* is used to dereference a

pointer therefore if we print \*p, we will get the value stored at the address contained by p.

**Output**

Address of number variable is fff4

Address of p variable is fff4

Value of p variable is 50

**2.**

#include <stdio.h>

int main()

{

   int num = 10;

   printf("Value of variable num is: %d", num);

   /\* To print the address of a variable we use %p

    \* format specifier and ampersand (&) sign just

    \* before the variable name like &num.

    \*/

   printf("\nAddress of variable num is: %p", &num);

   return 0;

}

**Output:**

Value of variable num is: 10

Address of variable num is: 0x7fff5694dc58

A picture containing diagram

Description automatically generated

**1.1.1 Pointer to array**

**int** arr[10];

**int** \*p[10]=&arr;       // Variable p of type pointer is pointing to the address of an integer array arr.

double balance[50];

**balance** is a pointer to &balance[0], which is the address of the first element of the array balance. Thus, the following program fragment assigns **p** as the address of the first element of **balance** −

double \*p;

double balance[10];

p = balance;

It is legal to use array names as constant pointers, and vice versa. Therefore, \*(balance + 4) is a legitimate way of accessing the data at balance[4].

Once you store the address of the first element in 'p', you can access the array elements using \*p, \*(p+1), \*(p+2) and so on. Given below is the example to show all the concepts discussed above −

#include <stdio.h>

int main () {

   /\* an array with 5 elements \*/

   double balance[5] = {1000.0, 2.0, 3.4, 17.0, 50.0};

   double \*p;

   int i;

   p = balance;

   /\* output each array element's value \*/

   printf( "Array values using pointer\n");

   for ( i = 0; i < 5; i++ ) {

      printf("\*(p + %d) : %f\n",  i, \*(p + i) );

   }

   printf( "Array values using balance as address\n");

   for ( i = 0; i < 5; i++ ) {

      printf("\*(balance + %d) : %f\n",  i, \*(balance + i) );

   }

   return 0;

}

When the above code is compiled and executed, it produces the following result −

Array values using pointer

\*(p + 0) : 1000.000000

\*(p + 1) : 2.000000

\*(p + 2) : 3.400000

\*(p + 3) : 17.000000

\*(p + 4) : 50.000000

Array values using balance as address

\*(balance + 0) : 1000.000000

\*(balance + 1) : 2.000000

\*(balance + 2) : 3.400000

\*(balance + 3) : 17.000000

\*(balance + 4) : 50.000000

**1.1.2 Pointer to a function**

**void** show (**int**);

**void**(\*p)(**int**) = &display; // Pointer p is pointing to the address of a function

#include <stdio.h>

void swap(int \*n1, int \*n2);

int main()

{

    int num1 = 5, num2 = 10;

    // address of num1 and num2 is passed

    swap( &num1, &num2);

    printf("num1 = %d\n", num1);

    printf("num2 = %d", num2);

    return 0;

}

void swap(int\* n1, int\* n2)

{

    int temp;

    temp = \*n1;

    \*n1 = \*n2;

    \*n2 = temp;

}

**output**

num1 = 10

num2 = 5

**1.1.3 Pointer to structure**

**struct** st {

**int** i;

**float** f;

}ref;

1.**struct** st \*p = &ref;

2.Diagram

Description automatically generated

#include <stdio.h>

#include <string.h>

struct student

{

     int id;

     char name[30];

     float percentage;

};

int main()

{

     int i;

     struct student record1 = {1, "Raju", 90.5};

     struct student \*ptr;

     ptr = &record1;

         printf("Records of STUDENT1: \n");

         printf("  Id is: %d \n", ptr->id);

         printf("  Name is: %s \n", ptr->name);

         printf("  Percentage is: %f \n\n", ptr->percentage);

     return 0;

}

OUTPUT:

|  |
| --- |
| Records of STUDENT1: Id is: 1 Name is: Raju Percentage is: 90.500000 |

**1.1.4 NULL Pointer**

A pointer that is not assigned any value but NULL is known as the NULL pointer. If you don't have any address to be specified in the pointer at the time of declaration, you can assign a NULL value. It will provide a better approach.

int \*p=NULL;

In most libraries, the value of the pointer is 0 (zero).

**1.2. Advantage of pointer**

• Pointer **reduces the code** and **improves the performance**, it is used to retrieving strings, trees, etc. and used with arrays, structures, and functions.

• We can **return multiple values from a function** using the pointer.

• It makes you able to **access any memory location** in the computer's memory.

**1.3. Usage of pointer**

There are many applications of pointers in the C language.

• **Dynamic memory allocation**

In c language, we can dynamically allocate memory using malloc() and calloc() functions where the pointer is used.

• **Arrays, Functions, and Structures**

Pointers in c language are widely used in arrays, functions, and structures. It reduces the code and improves the performance.

**1.4. How to read the pointer: int (\*p)[10].**

To read the pointer, we must see that () and [] have the equal precedence. Therefore, their associativity must be considered here. The associativity is left to right, so the priority goes to ().

Inside the bracket (), pointer operator \* and pointer name (identifier) p have the same precedence. Therefore, their associativity must be considered here which is right to left, so the priority goes to p, and the second priority goes to \*.

The pointer will be read as p is a pointer to an array of integers of size 10.

**Example-** How to read the following pointer?

**int** (\*p)(**int** (\*)[2], **int** (\*)**void**))

Explanation-

This pointer will be read as p is a pointer to such function which accepts the first parameter as the pointer to a one-dimensional array of integers of size two and the second parameter as the pointer to a function whose parameter is void and return type is the integer.

**1.5**. **Pointer Expressions and Pointer Arithmetic**

A limited set of arithmetic operations can be performed on pointers. A pointer may be:

• incremented ( ++ )

• decremented ( — )

• an integer may be added to a pointer ( + or += )

• an integer may be subtracted from a pointer ( – or -= )

Pointer arithmetic is meaningless unless performed on an array.

**Note :** Pointers contain addresses. Adding two addresses makes no sense, because there is no idea what it would point to. Subtracting two addresses lets you compute the offset between these two addresses.

**Example 1:** The output of the following program is \_\_\_\_\_\_\_\_.

|  |
| --- |
| int main()  {      int a = 5;      int \*ptr ;      ptr = &a;      \*ptr = \*ptr \* 3;      printf("%d", a);        return 0;  } |
|  |

**Output:**  15

**Explanation:**  
ptr = &a; copies the address of a in ptr making \*ptr = a and the statement \*ptr = \*ptr \* 3; can be written as a = a \* 3; making **a** as 15.

**Example 2:** The output of the following program is \_\_\_\_\_\_\_\_.

int main()

{

    int x = 10;

    int \*y, \*\*z;

    y = &x;

    z = &y;

printf("x = %d, y = %d, z = %d\n", x, \*y, \*\*z);

    return 0;

}

Output:   x=10 y=10 z=10

**Explanation:**

\*y is a pointer variable whereas \*\*z is a pointer to a pointer variable. \*y gives the value at the address it holds and \*\*z searches twice i.e., it first takes the value at the address it holds and then gives the value at that address.

**2. STORAGE CLASSES IN C**

Storage classes in C are used to determine the lifetime, visibility, memory location, and initial value of a variable. There are four types of storage classes in C

* Automatic
* External
* Static
* Register

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Storage Classes | Storage Place | Default Value | Scope | Lifetime |
| auto | RAM | Garbage Value | Local | Within function |
| extern | RAM | Zero | Global | Till the end of the main program May Be declared anywhere in the program |
| static | RAM | Zero | Local | Till the end of the main program, Retains value between multiple functions call |
| register | Register | Garbage Value | Local | Within the function |

**2.1. Automatic**

* Automatic variables are allocated memory automatically at runtime.
* The visibility of the automatic variables is limited to the block in which they are defined.
* The scope of the automatic variables is limited to the block in which they are defined.
* The automatic variables are initialized to garbage by default.
* The memory assigned to automatic variables gets freed upon exiting from the block.
* The keyword used for defining automatic variables is auto.
* Every local variable is automatic in C by default.

**Example :**

#include <stdio.h>

**int** main()

{

**int** a = 10,i;

printf("%d ",++a);

{

**int** a = 20;

**for** (i=0;i<3;i++)

{

printf("%d ",a); // 20 will be printed 3 times since it is the local value of a

}

}

printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.

}

**Output:**

  11 20 20 20 11

**2.2. Static**

* The variables defined as static specify that it can hold their value between the multiple function calls.
* Static local variables are visible only to the function or the block in which they are defined.
* The same static variable can be declared many times but can be assigned at only one time.
* Default initial value of the static integral variable is 0 otherwise null.
* The visibility of the static global variable is limited to the file in which it has been declared.
* The keyword used to define a static variable is static.

**Example :**

#include<stdio.h>

**void** sum()

{

**static** **int** a = 10;

**static** **int** b = 24;

printf("%d %d \n",a,b);

a++;

b++;

}

**void** main()

{

**int** i;

**for**(i = 0; i< 3; i++)

{

sum(); // The static variables holds their value b/w multiple function calls.

}

}

**Output:**

10 24

11 25

12 26

**2.3. Register**

* The variables defined as the register are allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
* We cannot dereference the register variables, i.e., we can not use &operator for the register variable.
* The access time of the register variables is faster than the automatic variables.
* The initial default value of the register local variables is 0.
* The register keyword is used for the variable which should be stored in the CPU register. However, it is the compiler's choice whether or not; the variables can be stored in the register.
* We can store pointers into the register, i.e., a register can store the address of a variable.
* Static variables can not be stored into the register since we can not use more than one storage specifier for the same variable.

**2.4. External**

* The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
* The variables declared as extern are not allocated any memory. It is only a declaration and intended to specify that the variable is declared elsewhere in the program.
* The default initial value of the external integral type is 0 otherwise null.
* We can only initialize the external variable globally, i.e., we can not initialize the external variable within any block or method.
* An external variable can be declared many times but can be initialized at only once.
* If a variable is declared as external then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.

**Example-1**

#include <stdio.h>

**int** a = 20;

**int** main()

{

**extern** **int** a;

printf("%d",a);

}

**Output**

20

**3. ARRAYS**

• Array is a collection of similar elements having same data type, accessed using a common name.

• An array is a collection of items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array.

• Array elements occupy contiguous memory locations.

• If first element is “i” and last element is “j” then:

Number of elements before j = j - i

Number of elements including j = j – i + 1

type variable[num\_elements];

Example: int A[100];

• It creates an array A with 100 integer elements.

• The size of an array A can’t be changed.

•  We cannot declare an array without assigning size. If we declare an array without size, it will throw a compile time error.

• The number between the brackets must be a constant.

**Access Array Elements**

You can access elements of an array by indices.

Suppose you declared an array mark as above. The first element is mark[0], the second element is mark[1] and so on.

Table

Description automatically generated with medium confidence

**Few keynotes**:

* Arrays have 0 as the first index, not 1. In this example, mark[0] is the first element.
* If the size of an array is n, to access the last element, the n-1 index is used. In this example, mark[4]
* Suppose the starting address of mark[0] is **2120d**. Then, the address of the mark[1] will be **2124d**. Similarly, the address of mark[2] will be **2128d** and so on.  
  This is because the size of a float is 4 bytes.

**How to initialize an array**

It is possible to initialize an array during declaration. For example,

int mark[5] = {19, 10, 8, 17, 9};

You can also initialize an array like this.

int mark[] = {19, 10, 8, 17, 9};

Here, we haven't specified the size. However, the compiler knows its size is 5 as we are initializing it with 5 elements.

A picture containing table

Description automatically generated

Here,

mark[0] is equal to 19

mark[1] is equal to 10

mark[2] is equal to 8

mark[3] is equal to 17

mark[4] is equal to 9

**Change Value of Array elements**

int mark[5] = {19, 10, 8, 17, 9}

// make the value of the third element to -1

mark[2] = -1;

// make the value of the fifth element to 0

mark[4] = 0;

**Input and Output Array Elements**

Here's how you can take input from the user and store it in an array element.

// take input and store it in the 3rd element

​scanf("%d", &mark[2]);

// take input and store it in the ith element

scanf("%d", &mark[i-1]);

Here's how you can print an individual element of an array.

// print the first element of the array

printf("%d", mark[0]);

// print the third element of the array

printf("%d", mark[2]);

// print ith element of the array

printf("%d", mark[i-1]);

**Example 1: Array Input/Output**

// Program to take 5 values from the user and store them in an array

// Print the elements stored in the array

#include <stdio.h>

int main() {

  int values[5];

  printf("Enter 5 integers: ");

  // taking input and storing it in an array

  for(int i = 0; i < 5; ++i) {

     scanf("%d", &values[i]);

  }

  printf("Displaying integers: ");

  // printing elements of an array

  for(int i = 0; i < 5; ++i) {

     printf("%d\n", values[i]);

  }

  return 0;

}

**Output**

Enter 5 integers: 1

-3

34

0

3

**3.1. TYPES OF ARRAYS**

**3.1.1. One dimensional (1-D) arrays or Linear arrays:**

In it each element is represented by a single subscript. The elements are stored in consecutive memory locations. E.g. A [1], A [2], ….., A [N].

The [] is used for dimensional or the subscript of the array that is generally used for declaring the elements of the array. For Accessing the Element from the array we can use the Subscript of the Array like this

                                    a[3]=4

This will set the value of 4th element of array.

Table, calendar

Description automatically generated

**Example: Calculate Average**

// Program to find the average of n numbers using arrays

#include <stdio.h>

int main()

{

     int marks[10], i, n, sum = 0, average;

     printf("Enter number of elements: ");

     scanf("%d", &n);

     for(i=0; i<n; ++i)

     {

          printf("Enter number%d: ",i+1);

          scanf("%d", &marks[i]);

          // adding integers entered by the user to the sum variable

          sum += marks[i];

     }

     average = sum/n;

     printf("Average = %d", average);

     return 0;

}

**Output**

Enter n: 5

Enter number1: 45

Enter number2: 35

Enter number3: 38

Enter number4: 31

Enter number5: 49

Average = 3

**3.1.2. Two dimensional (2-D) arrays or Matrix arrays:**

In it each element is represented by two subscripts. Thus, a two dimensional m x n array A has m rows and n columns and contains m\*n elements. It is also called matrix array because in it the elements form a matrix. E.g. A [3] [4] has 3 rows and 4 columns and 3\*4 = 12 elements.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Column1** | **Column2** | **Column3** | **Column4** |
| Row1 | a[0] [0] | a[0] [1] | a[0] [2] | a[0] [3] |
| Row2 | a[1] [0] | a[1] [1] | a[1] [2] | a[1] [3] |
| Row3 | a[2] [0] | a[2] [1] | a[2] [1] | a[2] [3] |

**Example 1: Sum of two matrices**

// C program to find the sum of two matrices of order 2\*2

#include <stdio.h>

int main()

{

  float a[2][2], b[2][2], result[2][2];

  // Taking input using nested for loop

  printf("Enter elements of 1st matrix\n");

  for (int i = 0; i < 2; ++i)

    for (int j = 0; j < 2; ++j)

    {

      printf("Enter a%d%d: ", i + 1, j + 1);

      scanf("%f", &a[i][j]);

    }

  // Taking input using nested for loop

  printf("Enter elements of 2nd matrix\n");

  for (int i = 0; i < 2; ++i)

    for (int j = 0; j < 2; ++j)

    {

      printf("Enter b%d%d: ", i + 1, j + 1);

      scanf("%f", &b[i][j]);

    }

  // adding corresponding elements of two arrays

  for (int i = 0; i < 2; ++i)

    for (int j = 0; j < 2; ++j)

    {

      result[i][j] = a[i][j] + b[i][j];

    }

  // Displaying the sum

  printf("\nSum Of Matrix:");

  for (int i = 0; i < 2; ++i)

    for (int j = 0; j < 2; ++j)

    {

      printf("%.1f\t", result[i][j]);

      if (j == 1)

        printf("\n");

    }

  return 0;

}

**Output**

Enter elements of 1st matrix

Enter a11: 2;

Enter a12: 0.5;

Enter a21: -1.1;

Enter a22: 2;

Enter elements of 2nd matrix

Enter b11: 0.2;

Enter b12: 0;

Enter b21: 0.23;

Enter b22: 23;

Sum Of Matrix:

2.2     0.5

-0.9    25.0

**3.1.3 Three Dimensional (3D ) Array-**

An Array is a group of elements with the same (homogeneous) data type. It is also called a Derived data type. As already noticed, a 3D array increases the space exponentially, and an extra position is added to locate the element in the array. In this topic, we are going to learn about 3D Arrays in C.

For example, consider a 4 level building with many slots for bike parking. So, here for getting the perfect slot directions of the bike that is parked, we need to tell the level number with row and column number. When you just tell the array, row 7 and column 4, which level does it search for? This 3D array is just for storing more amounts of data and representing the positions.

How can we define and implement them? Going further, let’s understand those concepts.

Syntax:

In C, Dimensional arrays can be declared as follows:

syntax 1.1

So, in the same way, we can declare the 3-D array as:

syntax 1.2

The meaning of the above representation can be understood as:

1. The memory allocated to variable c is of data type int.
2. The total capacity that this array can hold is 2\*3\*4, which is equal to 24 elements.
3. The data is being represented in the form of 2 arrays with 3 rows and 4 columns each.

**Example: Three-dimensional array**

// C Program to store and print 12 values entered by the user

#include <stdio.h>

int main()

{

  int test[2][3][2];

  printf("Enter 12 values: \n");

  for (int i = 0; i < 2; ++i)

  {

    for (int j = 0; j < 3; ++j)

    {

      for (int k = 0; k < 2; ++k)

      {

        scanf("%d", &test[i][j][k]);

      }

    }

  }

  // Printing values with proper index.

  printf("\nDisplaying values:\n");

  for (int i = 0; i < 2; ++i)

  {

    for (int j = 0; j < 3; ++j)

    {

      for (int k = 0; k < 2; ++k)

      {

        printf("test[%d][%d][%d] = %d\n", i, j, k, test[i][j][k]);

      }

    }

  }

  return 0;

}

**Output**

Enter 12 values:

1

2

3

4

5

6

7

8

9

10

11

12

Displaying Values:

test[0][0][0] = 1

test[0][0][1] = 2

test[0][1][0] = 3

test[0][1][1] = 4

test[0][2][0] = 5

test[0][2][1] = 6

test[1][0][0] = 7

test[1][0][1] = 8

test[1][1][0] = 9

test[1][1][1] = 10

test[1][2][0] = 11

test[1][2][1] = 12

**3.2. INITIALIZATION OF AN ARRAY**

• int A[5]= {1,2,3,4,5}; /\*Array can be initialized during declaration\*/

• int A[5]={1,2,3}; /\* Remaining elements are automatically initialized to zero\*/

• int A[5]={1,[1]=2, 3,4,[4]=0};/\* Array element can be initialized by specifying its index location\*/

**3.3. POINTERS & ARRAYS**

**int a[10]** : It will be read as an array of 10 elements with integer data type.

• Every array contains a base address which is the address of the first element of the array.

• An array variable is just a pointer to the first element in the array.

• You can access array elements using array notation or pointers.

• print a will print the base address of the array.

• a[0] is the same as \*a

• a[1] is the same as \*(a + 1)

• a[2] is the same as \*(a + 2)

• a = a+0 = &a[0]

• a+1 = &a[1]

• a+i = &a[i]

• &(\*(a+i)) = &a[i] = a+i

• \*(&a[i]) = \*(a+i) = a[i]

• Address of an element i of array a = a + i \* sizeof(element)

Let b be the 2- Dimensional Array b[i][j]

• \*(\*(b + i) + j) is equivalent to b[i][j], here first \* is used to select the rows and the second \* selects the element.

• &b +1 means the whole 2-D array is skipped.

• &b gives the base address of the array.

• \*b + 1 skips one element.

• \*(b + i) + j is equivalent to &b[i][j]

• \*(b[i] + j) is equivalent to b[i][j]

• b[i] + j is equivalent to &b[i][j]

• (\*(b+i))[j] is equivalent to b[i][j]

**Example-**

#include <stdio.h>

int main() {

   int x[4];

   int i;

   for(i = 0; i < 4; ++i) {

      printf("&x[%d] = %p\n", i, &x[i]);

   }

   printf("Address of array x: %p", x);

   return 0;

}

**Output**

&x[0] = 1450734448

&x[1] = 1450734452

&x[2] = 1450734456

&x[3] = 1450734460

Address of array x: 1450734448

There is a difference of 4 bytes between two consecutive elements of array x. It is because the size of int is 4 bytes (on our compiler).

Notice that, the address of &x[0] and x is the same. It's because the variable name x points to the first element of the array.

A picture containing table

Description automatically generated

From the above example, it is clear that &x[0] is equivalent to x. And, x[0] is equivalent to \*x.

Similarly,

* &x[1] is equivalent to x+1 and x[1] is equivalent to \*(x+1).
* &x[2] is equivalent to x+2 and x[2] is equivalent to \*(x+2).
* ...
* Basically, &x[i] is equivalent to x+i and x[i] is equivalent to \*(x+i).

**Example: Pointers and Arrays**

#include <stdio.h>

int main() {

  int i, x[6], sum = 0;

  printf("Enter 6 numbers: ");

  for(i = 0; i < 6; ++i) {

  // Equivalent to scanf("%d", &x[i]);

      scanf("%d", x+i);

  // Equivalent to sum += x[i]

      sum += \*(x+i);

  }

  printf("Sum = %d", sum);

  return 0;

}

When you run the program, the output will be:

Enter 6 numbers:  2

 3

 4

 4

 12

 4

Sum = 29

**3.4. CALCULATING LOCATION OF AN ELEMENT IN AN ARRAY**

**3.4.1. 1-D Arrays**

Consider a single dimensional array as **A[lb------ub]**

The base address of array = BA

Size of each element in array = c

Total elements in array is given by (ub-lb+1)

Then address of any random element A[i] is given by : BA + (i-lb+1)\*c

**Example 1:** Let A [78……380] , Base address = 1000, size of element = 10

Then LOC (A [250])= 1000 + (250 – 78) \* 10

                                   = 1000 + 2150

                                   = 3150

**Example 2:** Let A[ -50……100] , Base Address = 1000, size of element = 10

Then LOC( A [60]) = 1000 + (60 – (-50)) \* 10

                                 = 1000 + 1100

                                 = 2100

**3.4.2. 2-D Arrays**

2-D arrays can be stored in the system in two ways: Row Major order and Column Major order.

Consider a 2-D array as **A[lb1-----ub1] [lb2------ub2]**

The base address of array = BA

Size of each element in array = c

Number of rows = ub1-lb1+1

Number of columns = ub2-lb2+1

While storing the elements of a 2-D array in memory, these are allocated contiguous memory locations. Therefore, a 2-D array must be linearized so as to enable their storage. There are two alternatives to achieve linearization: Row-Major and Column-Major.

A picture containing table

Description automatically generated

Diagram

Description automatically generated

Address of an element of an array say A[I][J] is Calculated in two forms as given:  
1. Row Major System         2. Column Major System

1. **Row Major order**, then address of any random element A[i][j] is given by :

Where, (i-lb1) gives the number of rows before i,

                             (ub2-lb2+1) gives the number of column and

                             (j-lb2) tells the number of columns before j

2. **Column Major order**, then address of any random element A[i][j] is given by :

              Where, (j-lb2) gives the number of column before j ,

                            (ub1-lb1+1) gives the number of rows and

                            (i-lb1) tells the number of rows before i

**Examples:** Let A [30…….80 , 3……150] , Base Address = 1000 , size of element = 10

Find LOC( A[73][120]).

**Sol:**  Number of columns = 150 – 3 + 1 = 152

        Number of rows = 80 – 30 +  1 = 51

        Using Row major:

LOC( A[73][120]) = 1000 + [ (73 – 30)\* 152 + (120 – 3)] \* 10

                                       = 1000 + 54920

                                       = 55920

        Using Column Major:

LOC( A[73][120]) = 1000 + [ (120 – 3)\* 51 + (73 – 30)]\*10

                                = 1000 + 59713

                                = 60713

**Example 4:** Let A [-25…….75, -35…….125] , Base address = 1000, Size of element = 10

Find LOC( A[50][40])

**Sol:** Number of columns = 75 – (-25) + 1 = 101

        Number of rows = 125 – (-35) +  1 = 161

        Using Row major:

LOC( A[50][40]) = 1000 + [ (50-(-25))\*101 + (40 – (-35)) \*10

                                     = 1000 + (75 \*101 + 75) \* 10

                                     = 77500

        Using Column Major:

LOC( A[50][40]) = 1000 + [ (40 – (-35)) \* 161 + ((50-(-25))] \*10

                                              = 1000 + 121500

                                              = 122500

**3.5. STRINGS**

• Strings are defined as an array of characters. The difference between a character array and a string is that the string is terminated with a special character ‘\0’.

• A string is a sequence of characters terminated with a null character \0

• Declaring a string is as simple as declaring a 1- Dimensional array.

• Below is the basic syntax for declaring a string in C programming language.

                             char str\_name[size];

|  |
| --- |
| String Declaration  1. char str [] = {‘A’, ‘B’, ‘C’, ‘D’, ‘\0’};  2. char st [] = “ABCD”;                                              ⇓  ‘\0’ would automatically  Inserted at the end in this type of declaration |

**Example 5:** The string "hello world" contains 12 characters including '\0'

• Before the string class, the abstract idea of a string was implemented with just an array of characters. For example, here is a string:

char label[] = "Single";

The array in the memory will be as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **S** | **i** | **n** | **g** | **l** | **e** | **\0** |

where the beginning of the array is at some location in computer memory.

• A character array can have more characters than the abstract string held in it, as below:

char label [10] = "Single";

The array will look like:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **h** | **i** | **n** | **g** | **l** | **e** | **\0** |

(where 3 array elements are currently unused).

**How to declare a string**

Here's how you can declare strings:

char s[5];

Box and whisker chart

Description automatically generated with low confidence

Here, we have declared a string of 5 charac

**How to initialize strings**

You can initialize strings in a number of ways.

char c[] = "abcd";

char c[50] = "abcd";

char c[] = {'a', 'b', 'c', 'd', '\0'};

char c[5] = {'a', 'b', 'c', 'd', '\0'};

A picture containing text, clock

Description automatically generated

Let's take another example:

char c[5] = "abcde";

Here, we are trying to assign 6 characters (the last character is '\0') to a char array having 5 characters. This is bad and you should never do this.

**Assigning Values to Strings**

Arrays and strings are second-class citizens in C; they do not support the assignment operator once it is declared. For example,

char c[100];

c = "C programming";  // Error! array type is not assignable.

**Note:**Use the strcpy() function to copy the string instead.

**Read String from the user**

You can use the scanf() function to read a string.

The scanf() function reads the sequence of characters until it encounters whitespace (space, newline, tab, etc.).

**Example: scanf() to read a string**

#include <stdio.h>

int main()

{

    char name[20];

    printf("Enter name: ");

    scanf("%s", name);

    printf("Your name is %s.", name);

    return 0;

}

**Output**

Enter name: Gradeup

Your name is Gradeup

Even though Dennis Ritchie was entered in the above program, only "Dennis" was stored in the name string. It's because there was a space after Dennis.

**How to read a line of text**

You can use the fgets() function to read a line of string. And, you can use puts() to display the string

**Example: fgets() and puts()**

#include <stdio.h>

int main()

{

    char name[30];

    printf("Enter name: ");

    fgets(name, sizeof(name), stdin);  // read string

    printf("Name: ");

    puts(name);    // display string

    return 0;

}

**Output**

Enter name: Gradeup

Name: Gradeup

Here, we have used fgets() function to read a string from the user.

fgets(name, sizeof(name), stdlin); // read string

The sizeof(name) results to 30. Hence, we can take a maximum of 30 characters as input which is the size of the name string.

To print the string, we have used puts(name);.

**Note:** The gets() function can also be to take input from the user. However, it is removed.  
It's because gets() allows you to input any length of characters. Hence, there might be a buffer overflow.

**4. STRUCTURE**

Structure is a user-defined datatype in C language which allows us to combine data of different types together. Structure helps to construct a complex data type which is more meaningful. It is somewhat similar to an Array, but an array holds data of similar type only. But structure on the other hand, can store data of any type, which is practical and more useful.

**4.1. DEFINING A STRUCTURE**

struct keyword is used to define a structure. struct defines a new data type which is a collection of primary and derived data types.

**Syntax:**

struct [structure\_tag]

{

    //member variable 1

    //member variable 2

    //member variable 3

    ...

}[structure\_variables];

As you can see in the syntax above, we start with the struct keyword, then it's optional to provide your structure a name, we suggest you to give it a name, then inside the curly braces, we have to mention all the member variables, which are nothing but normal C language variables of different types like int, float, array etc.

After the closing curly brace, we can specify one or more structure variables, again this is optional.

**Note:** The closing curly brace in the structure type declaration must be followed by a semicolon (;).

**Example of Structure**

struct Student

{

    char name[25];

    int age;

    char branch[10];

    // F for female and M for male

    char gender;

};

Here struct Student declares a structure to hold the details of a student which consists of 4 data fields, namely name, age, branch and gender. These fields are called **structure elements or members**.

Each member can have different data type, like in this case, name is an array of char type and age is of int type etc. **Student** is the name of the structure and is called as the **structure tag**.

**4.2. DECLARING STRUCTURE VARIABLES**

It is possible to declare variables of a **structure**, either along with structure definition or after the structure is defined. **Structure** variable declaration is similar to the declaration of any normal variable of any other datatype. Structure variables can be declared in following two ways:

**4.2.1  Declaring Structure variables separately**

struct Student

{

    char name[25];

    int age;

    char branch[10];

    //F for female and M for male

    char gender;

};

struct Student S1, S2;      //declaring variables of struct Student

**4.2.2  Declaring Structure variables with structure definition**

struct Student

{

    char name[25];

    int age;

    char branch[10];

    //F for female and M for male

    char gender;

}S1, S2;

Here S1 and S2 are variables of structure Student. However this approach is not much recommended.

**4.3. ACCESSING STRUCTURE MEMBERS**

Structure members can be accessed and assigned values in a number of ways. Structure members have no meaning individually without the structure. In order to assign a value to any structure member, the member name must be linked with the **structure** variable using a dot (.) operator also called **period** or **member access** operator.

**For example:**

#include<stdio.h>

#include<string.h>

struct Student

{

    char name[25];

    int age;

    char branch[10];      //F for female and M for male

    char gender;

};

int main()

{

    struct Student s1;   /\*s1 is a variable of Student type and  age is a member of Student\*/

    s1.age = 18;   /\* using string function to add name   \*/

strcpy(s1.name, "Camy");         /\*displaying the stored values\*/

printf("Name of Student 1: %s\n", s1.name);

printf("Age of Student 1: %d\n", s1.age);

    return 0;

}

Name of Student 1: Camy

Age of Student 1: 18

We can also use scanf() to give values to structure members through terminal.

scanf(" %s ", s1.name);

scanf(" %d ", &s1.age);

**4.4. STRUCTURE INITIALIZATION**

Like a variable of any other datatype, the structure variable can also be initialized at compile time.

struct Patient

{

    float height;

    int weight;

    int age;

};

struct Patient p1 = { 180.75 , 73, 23 };    //initialization

or,

struct Patient p1;

p1.height = 180.75;     //initialization of each member separately

p1.weight = 73;

p1.age = 23;

**4.5. ARRAY OF STRUCTURE**

We can also declare an array of **structure** variables. in which each element of the array will represent a **structure** variable. **Example :** struct employee emp[5];

The below program defines an array emp of size 5. Each element of the array emp is of type Employee.

#include<stdio.h>

struct Employee

{

    char ename[10];

    int sal;

};

struct Employee emp[5];

int i, j;

void ask()

{

for(i = 0; i < 3; i++)

    {

printf("\nEnter %dst Employee record:\n", i+1);

printf("\nEmployee name:\t");

scanf("%s", emp[i].ename);

printf("\nEnter Salary:\t");

scanf("%d", &emp[i].sal);

    }

printf("\nDisplaying Employee record:\n");

for(i = 0; i < 3; i++)

    {

printf("\nEmployee name is %s", emp[i].ename);

printf("\nSalary is %d", emp[i].sal);

    }

}

void main()

{

ask();

}

**4.6. NESTED STRUCTURES**

Nesting of structures is also permitted in C language. Nested structures means that one structure has another structure as member variable.

**Example:**

struct Student

{

char[30] name;

    int age;

    /\* here Address is a structure \*/

    struct Address

    {

char[50] locality;

char[50] city;

        int pincode;

}addr;

};

**4.7. STRUCTURE AS FUNCTION ARGUMENTS**

We can pass a structure as a function argument just like we pass any other variable or an array as a function argument.

**Example:**

#include<stdio.h>

struct Student

{

    char name[10];

    int roll;

};

void show(struct Student st);

void main()

{

    struct Student std;

printf("\nEnter Student record:\n");

printf("\nStudent name:\t");

scanf("%s", std.name);

printf("\nEnter Student rollno.:\t");

scanf("%d", &std.roll);

    show(std);

}

void show(struct Student st)

{

printf("\nstudent name is %s", st.name);

printf("\nroll is %d", st.roll);

}

**4.8. C STRUCTS AND POINTERS**

In this tutorial, you'll learn to use pointers to access members of structs in C programming. You will also learn to dynamically allocate memory of struct types.

C Pointers to struct

Here's how you can create pointers to structs

struct name {

member1;

member2;

   ---

    ---

};

int main()

{

    struct name \*ptr, Harry;

}

Here, **ptr** is a pointer to struct.

**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.

By the way,

* **personPtr->age** is equivalent to **(\*personPtr).age**
* **personPtr->weight** is equivalent to **(\*personPtr).weight**

**4.9. SELF REFERENTIAL STRUCTURES**

Self Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.

In other words, structures pointing to the same type of structures are self-referential in nature.

Example:

|  |
| --- |
| struct node {      int data1;      char data2;      struct node\* link;  };    int main()  {      struct node ob;      return 0;  } |

In the above example ‘link’ is a pointer to a structure of type ‘node’. Hence, the structure ‘node’ is a self-referential structure with ‘link’ as the referencing pointer.

An important point to consider is that the pointer should be initialized properly before accessing, as by default it contains garbage value.

**Types of Self Referential Structures**

1.Self Referential Structure with Single Link

2.Self Referential Structure with Multiple Links

**4.9.1. Self Referential Structure with Single Link:**These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members. The connection formed is shown in the following figure.

Diagram, arrow

Description automatically generated

|  |
| --- |
| #include <stdio.h>  struct node {      int data1;      char data2;      struct node\* link;  };    int main()  {      struct node ob1; // Node1        // Initialization      ob1.link = NULL;      ob1.data1 = 10;      ob1.data2 = 20;        struct node ob2; // Node2        // Initialization      ob2.link = NULL;      ob2.data1 = 30;      ob2.data2 = 40;        // Linking ob1 and ob2      ob1.link = &ob2;        // Accessing data members of  ob2 using ob1      printf("%d", ob1.link->data1);      printf("\n%d", ob1.link->data2);      return 0;  } |

**Output:**

30

40

**4.9.2. Self Referential Structure with Multiple Links:**Self referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one node at a time. The following example shows one such structure with more than one link.

The connections made in the above example can be understood using the following figure.

A picture containing diagram

Description automatically generated

**4.10. APPLICATIONS**

Self referential structures are very useful in creation of other complex data structures like:

* Linked Lists
* Stacks
* Queues
* Trees
* Graphs etc.

**5. UNIONS**

A union is a special data type available in C that allows to stores 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.

**5.1. Defining a Union**

To define a union, you must use the union statement in the same way as you did while defining a structure. The union statement defines a new data type with more than one member for your program. The format of the union statement is as follows −

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 but it is optional. Here is the way you would define a union type named Data having three members i, f, and str −

union Data {

   int i;

   float f;

   char str[20];

} data;

Now, a variable of Data type can store an integer, a floating-point number, or a string of characters. It means a single variable, i.e., same memory location, can be used to store multiple types of data. You can use any built-in or user defined data types inside a union based on your requirement.

The memory occupied by a union will be large enough to hold the largest member of the union. For example, in the above example, Data type will occupy 20 bytes of memory space because this is the maximum space which can be occupied by a character string. The following example displays the total memory size occupied by the above union

#include <stdio.h>

#include <string.h>

union Data {

   int i;

   float f;

   char str[20];

};

int main( ) {

   union Data data;

   printf( "Memory size occupied by data : %d\n", sizeof(data));

   return 0;

}

When the above code is compiled and executed, it produces the following result −

Memory size occupied by data : 20

**5.2. Accessing 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. The following example shows how to use unions in a program −

#include <stdio.h>

#include <string.h>

union Data {

   int i;

   float f;

   char str[20];

};

int main( ) {

   union Data data;

   data.i = 10;

   data.f = 220.5;

   strcpy( data.str, "Gradeup");

   printf( "data.i : %d\n", data.i);

   printf( "data.f : %f\n", data.f);

   printf( "data.str : %s\n", data.str);

   return 0;

}

When the above code is compiled and executed, it produces the following result −

data.i : 1917853763

data.f : 4122360580327794860452759994368.000000

data.str : Gradeup

Here, we can see that the values of i and f members of union got corrupted because the final value assigned to the variable has occupied the memory location and this is the reason that the value of str member is getting printed very well.

Now let's look into the same example once again where we will use one variable at a time which is the main purpose of having unions −

#include <stdio.h>

#include <string.h>

union Data {

   int i;

   float f;

   char str[20];

};

int main( ) {

   union Data data;

   data.i = 10;

   printf( "data.i : %d\n", data.i);

   data.f = 220.5;

   printf( "data.f : %f\n", data.f);

   strcpy( data.str, "C Programming");

   printf( "data.str : %s\n", data.str);

   return 0;

}

When the above code is compiled and executed, it produces the following result −

data.i : 10

data.f : 220.500000

data.str : C Programming

Here, all the members are getting printed very well because one member is being used at a time.

**5.3. Following are the important differences between Structure and Union.**

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Key** | **Structure** | **Union** |
| **1** | Definition | Structure is the container defined in C to store data variables of different types and also supports for the user defined variables storage. | On the other hand Union is also a similar kind of container in C which can also hold the different types of variables along with the user defined variables. |
| **2** | Internal implementation | Structure in C is internally implemented as that there is separate memory location is allotted to each input member | While in case Union memory is allocated only to one member having largest size among all other input variables and the same location is being shared among all of these. |
| **3** | Syntax | Syntax of declare a Structure in C is as follow :  struct struct\_name{     type element1;     type element2;     .     .  } variable1, variable2, ...; | On other syntax of declare a Union in C is as follow:  union u\_name{     type element1;     type element2;     .     .  } variable1, variable2, ...; |
| **4** | Size | As mentioned in the definition Structure does not have shared location for its members so size of Structure is equal or greater than the sum of size of all the data members. | On the other hand, Union does not have a separate location for each of its members so its size is equal to the size of largest member among all data members. |
| **5** | Value storage | As mentioned above, in the case of Structure there is a specific memory location for each input data member and hence it can store multiple values of the different members. | While in the case of Union there is only one shared memory allocation for all input data members so it stores a single value at a time for all members. |
| **6** | Initialization | In Structure multiple members can be initialized at same time. | On the other hand in the case of Union only the first member can get initialized at a time. |

**6. LINKED LIST**

• Linked List is a very commonly used linear data structure which consists of group of **nodes** in a sequence.

• Each node holds its own **data** and the **address of the next node** hence forming a chain like structure.

• Linked Lists are used to create trees and graphs.

**6.1. Representation of linked list:**

Diagram

Description automatically generated

A linked list is represented by a pointer to the first node of the linked list. The first node is called the head. If the linked list is empty, then the value of the head is NULL.

Each node in a list consists of at least two parts:

1) data

2) Pointer (Or Reference) to the next node

**6.2. Advantages of Linked Lists**

• They are dynamic in nature which allocates the memory when required.

• Insertion and deletion operations can be easily implemented.

• Stacks and queues can be easily executed.

• Linked List reduces the access time.

**6.3. Disadvantages of Linked Lists**

• Extra memory space for a pointer is required with each element of the list.

• No element can be accessed randomly; it must access each node sequentially.

• Reverse Traversing is difficult in the linked list.

• In Linked Lists we don't need to know the size in advance.

**6.4. Applications of Linked Lists**

• Linked lists are used to implement stacks, queues, graphs, etc.

• Linked Lists can also be used to implement Graphs. (Adjacency list representation of Graph).

• Implementing Hash Tables: Each Bucket of the hash table can itself be a linked list. (Open chain hashing).

• Undo functionality in Photoshop or Word. Linked list of states.

• A polynomial can be represented in an array or in a linked list by simply storing the coefficient and exponent of each term.

• However, for any polynomial operation , such as addition or multiplication of polynomials , linked list representation is easier to deal with.

• Linked lists are useful for dynamic memory allocation.

• The real-life application where the circular linked list is used is our Personal Computers, where multiple applications are running.

• All the running applications are kept in a circular linked list and the OS gives a fixed time slot to all for running. The Operating System keeps on iterating over the linked list until all the applications are completed.

**6.5. TYPES OF LINKED LIST**

**6.5.1. Singly Linked list:**

• Each node has a single link to another node, called a Singly Linked List.

• A Singly Linked List does not store any pointer or reference to the previous node.

• Each node stores the contents of the node and a reference to the next node in the list.

• In a singly linked list, the last node has a pointer which indicates that it is the last node. It requires a reference to the first node to store a single linked list.

• It has two successive nodes linked together in a linear way and contains the address of the next node to be followed.

• It has a successor and predecessor. First node does not have a predecessor while the last node does not have a successor. Last node has a successor reference as NULL.

• It has only a single link for the next node.

Diagram

Description automatically generated

**Singly Linked List**

• In the above figure, the address of the first node is always stored in a reference node known as Head or Front. Reference part of the last node must be null.

**Operations on Single Linked List**

The following operations are performed on a Single Linked List

* **Insertion**
* **Deletion**
* **Display**

Before we implement actual operations, first we need to set up an empty list. First, perform the following steps before implementing actual operations.

* **Step 1 -**Include all the **header files** which are used in the program.
* **Step 2 -**Declare all the **user defined functions**.
* **Step 3 -**Define a **Node** structure with two members **data** and **next**
* **Step 4 -**Define a Node pointer **'head'** and set it to **NULL**.
* **Step 5 -**Implement the main method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

**Insertion**

In a single linked list, the insertion operation can be performed in three ways. They are as follows.

1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

**Inserting At Beginning of the list**

We can use the following steps to insert a new node at the beginning of the single linked list...

* **Step 1 -**Create a **newNode** with given value.
* **Step 2 -**Check whether list is **Empty** (**head** == **NULL**)
* **Step 3 -**If it is **Empty** then, set **newNode→next** = **NULL** and **head** = **newNode**.
* **Step 4 -**If it is **Not Empty** then,

set **newNode→next** = **head** and **head** = **newNode**.

**Inserting At End of the list**

We can use the following steps to insert a new node at the end of the single linked list...

* **Step 1 -**Create a **newNode** with given value and **newNode → next** as **NULL**.
* **Step 2 -**Check whether list is **Empty** (**head** == **NULL**).
* **Step 3 -**If it is **Empty** then, set **head** = **newNode**.
* **Step 4 -**If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
* **Step 5 -**Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp → next** is equal to **NULL**).
* **Step 6 -**Set **temp → next** = **newNode**.

**Inserting At Specific location in the list (After a Node)**

We can use the following steps to insert a new node after a node in the single linked list.

* **Step 1 -**Create a **newNode** with given value.
* **Step 2 -**Check whether list is **Empty** (**head** == **NULL**)
* **Step 3 -**If it is **Empty** then, set **newNode → next** = **NULL** and **head** = **newNode**.
* **Step 4 -**If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
* **Step 5 -**Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1 → data** is equal to **location**, here location is the node value after which we want to insert the newNode).
* **Step 6 -**Every time check whether **temp** is reached to last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion is not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
* **Step 7 -**Finally, Set **newNode → next** = **temp → next**' and '**temp → next** = **newNode**'

**Deletion**

In a single linked list, the deletion operation can be performed in three ways. They are as follows.

1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

**Deleting from Beginning of the list**

We can use the following steps to delete a node from the beginning of the single linked list.

* **Step 1 -**Check whether list is **Empty** (**head** == **NULL**)
* **Step 2 -**If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
* **Step 3 -**If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
* **Step 4 -**Check whether list is having only one node (**temp → next** == **NULL**)
* **Step 5 -**If it is **TRUE** then set **head** = **NULL** and delete **temp** (Setting **Empty** list conditions)
* **Step 6 -**If it is **FALSE** then set **head** = **temp → next**, and delete **temp**.

**Deleting from End of the list**

We can use the following steps to delete a node from the end of the single linked list...

* **Step 1 -**Check whether list is **Empty** (**head** == **NULL**)
* **Step 2 -**If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
* **Step 3 -** If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2'** and initialize '**temp1**' with **head**.
* **Step 4 -**Check whether list has only one Node (**temp1 → next** == **NULL**)
* **Step 5 -**If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate the function. (Setting **Empty** list condition)
* **Step 6 -**If it is **FALSE**. Then, set '**temp2 = temp1**' and move **temp1** to its next node. Repeat the same until it reaches the last node in the list. (until **temp1 → next** == **NULL**)
* **Step 7 -**Finally, Set **temp2 → next**= **NULL** and delete **temp1**.

**Deleting a Specific Node from the list**

We can use the following steps to delete a specific node from the single linked list...

* **Step 1 -**Check whether list is **Empty** (**head** == **NULL**)
* **Step 2 -**If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
* **Step 3 -**If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
* **Step 4 -**Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.
* **Step 5 -**If it is reached to the last node then display **'Given node not found in the list! Deletion is not possible!!!'**. And terminate the function.
* **Step 6 -**If it is reached to the exact node which we want to delete, then check whether list is having only one node or not
* **Step 7 -**If list has only one node and that is the node to be deleted, then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
* **Step 8 -**If list contains multiple nodes, then check whether **temp1** is the first node in the list (**temp1 == head**).
* **Step 9 -**If **temp1** is the first node then move the **head** to the next node (**head = head → next**) and delete **temp1**.
* **Step 10 -**If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == NULL**).
* **Step 11 -**If **temp1** is last node then set **temp2 → next** = **NULL** and delete **temp1** (**free(temp1)**).
* **Step 12 -**If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).

**Displaying a Single Linked List**

We can use the following steps to display the elements of a single linked list...

* **Step 1 -**Check whether list is **Empty** (**head** == **NULL**)
* **Step 2 -**If it is **Empty** then, display **'List is Empty!!!'** and terminate the function.
* **Step 3 -**If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
* **Step 4 -**Keep displaying **temp → data** with an arrow (**--->**) until **temp** reaches to the last node
* **Step 5 -**Finally display **temp → data** with arrow pointing to **NULL** (**temp → data ---> NULL**).

**Advantage**

1) Insertions and Deletions can be done easily.

2) It does not need movement of elements for insertion and deletion.

3) Space is not wasted as we can get space according to our requirements.

5) It can be extended or reduced according to requirements as size is not fixed.

6) Elements may or may not be stored in consecutive memory available, even then we can store the data in the computer.

7) It is less expensive.

**Disadvantage**  
1) It requires more space as pointers  are also stored  with the data.  
2) Different amounts of time is required to access each element.  
3) If we have to go to a particular element then we have to go through all those elements that come before that element.  
4) We cannot traverse it from last & only from the beginning.  
5) It is not easy to sort the elements stored in the linear linked list.

|  |
| --- |
| // A simple C program for traversal of a linked list  #include <stdio.h>  #include <stdlib.h>    struct Node {      int data;      struct Node\* next;  };    // This function prints contents of linked list starting from  // the given node  void printList(struct Node\* n)  {      while (n != NULL) {          printf(" %d ", n->data);          n = n->next;      }  }    int main()  {      struct Node\* head = NULL;      struct Node\* second = NULL;      struct Node\* third = NULL;        // allocate 3 nodes in the heap      head = (struct Node\*)malloc(sizeof(struct Node));      second = (struct Node\*)malloc(sizeof(struct Node));      third = (struct Node\*)malloc(sizeof(struct Node));        head->data = 1; // assign data in first node      head->next = second; // Link first node with second        second->data = 2; // assign data to second node      second->next = third;        third->data = 3; // assign data to third node      third->next = NULL;        printList(head);        return 0;  } |

**Output:**

 1   2   3

**6.5.2. Doubly linked list**

• Doubly linked list is a sequence of elements in which every node has a link to its previous node and next node.

• Traversing can be done in both directions and displays the contents in the whole list.

Diagram

Description automatically generated

**Doubly Linked List**

• In the above figure, Link1 field stores the address of the previous node and Link2 field stores the address of the next node. The Data Item field stores the actual value of that node. If we insert a data into the linked list, it will be look like as follows:

A picture containing diagram

Description automatically generated

**Doubly Linked List**

• First node is always pointed by the head.

• Previous field of the first node is always NULL, and the next field of the last must be NULL.

• Doubly linked list contains three fields. Links of two nodes allow traversal of the list in either direction. There is no need to traverse the list to find the previous node. We can traverse from head to tail as well as tail to head.

**Advantages of Doubly Linked List**

• Doubly linked list can be traversed in both forward and backward directions.

• To delete a node in a singly linked list, the previous node is required, while in doubly linked list, we can get the previous node using the previous pointer.

• It is more convenient than a singly linked list. Doubly linked list maintains the links for bidirectional traversing.

**Disadvantages of Doubly Linked List**

• In a doubly linked list, each node requires extra space for the previous pointer.

• All operations such as Insert, Delete, Traverse etc. require extra previous pointers to be maintained.

**6.5.3. Circular Linked List:**

• Circular linked list is similar to a singly linked list. The only difference is that in a circular linked list, the last node points to the first node in the list.

• It is a sequence of elements in which every element has a link to its next element in the sequence and has a link to the first element in the sequence.

Diagram

Description automatically generated

**Circular Singly Linked List**

Diagram

Description automatically generated

**Doubly Linked Circular list**

• In the above figure we see that each node points to its next node in the sequence but the last node points to the first node in the list. The previous element stores the address of the next element and the last element stores the address of the starting element. It forms a circular chain because the element points to each other in a circular way.

• In a circular linked list, the memory can be allocated when it is required because it has a dynamic size.

• Circular linked lists are used in personal computers, where multiple applications are running. The operating system provides a fixed time slot for all running applications and the running applications are kept in a circular linked list until all the applications are completed.

• Elements can be inserted anywhere in a circular linked list, but in the array, elements cannot be inserted anywhere in the list because it is in the contiguous memory.

**Advantages:**

• If we are at a node, then we can go to any node. But in the linear linked list it is not possible to go to the previous node.

• It saves time when we must go to the first node from the last node. It can be done in a single step because there is no need to traverse the in between nodes. But in a double linked list, we will have to go through in between nodes.

**Disadvantages:**

• It is not easy to reverse the linked list.

• If proper care is not taken, then the problem of infinite loop can occur.

• Circular lists are complex as compared to singly linked lists.

• Like singly and doubly lists, circular linked lists also don’t support direct accessing of elements.

**6.6. CREATION OF A NODE**

**a. Singly Linked List:**

Creation of a Node:

**#include***<stdio.h>*

**#include***<stdlib.h>*

**struct** node

{

**int**data;

**struct** node \*next;

};

**int**main()

{

**struct** node \*prev,\*head,\*p;

p=malloc(**sizeof**(**struct** node));

scanf("%d",&p->data);

p->next=NULL;

**return**0;

}

Structure of a Node:

struct Node {

    int data;

    struct Node\* next;

};

p= malloc(sizeof(struct node)) : We are allocating the space required for a node by the malloc function. Now, ‘p’ points to a node (or space allocated for the node).

scanf("%d",&p->data) : We are giving a value to the ‘data’ of ‘p’ after taking the input from the user.

p->next=NULL :  We have given the value to ‘data’ in the previous line and a value of the pointer ‘next’ (NULL) in this line and thus making our node ‘p’ complete.

**b. Doubly Linked List:**

Creation of a Node:

void addNode(int data) {

//Create a new node

    struct node \*newNode = (struct node\*)malloc(sizeof(struct node));

    newNode->data = data;

Structure of aNode:

struct node

{

int data;

struct node \*next; // Pointer to next node

struct node \*prev; // Pointer to previous node

};

**c. Circular Linked List:**

Creation of a Node:

void create()

{

node \*newnode;

newnode=(node\*)malloc(sizeof(node));

printf("\nEnter the node value : ");

scanf("%d",&newnode->info);

newnode->next=NULL;

if(rear==NULL)

front=rear=newnode;

else

{

rear->next=newnode;

rear=newnode;

}

rear->next=front;

}

**6.7. OPERATIONS ON LINKED LIST**

**6.7.1. Single Linked List:**

**i. Insertion:**

**a.At Beginning-**

1. allocate node

    struct Node\* new\_node = (struct Node\*) malloc(sizeof(struct Node));

2. put in the data

new\_node->data  =new\_data;

3. Make next of new node as head \*/

new\_node->next = (\*head\_ref);

4. Move the head to point to the new node \*/

    (\*head\_ref)    = new\_node;

**b.At End-**

1. allocate node

    struct Node\* new\_node = (struct Node\*) malloc(sizeof(struct Node));

    struct Node \*last = \*head\_ref;

2. put in the data

    new\_node->data = new\_data;

3. This new node is going to be the last node, so make next

          of it as NULL

    new\_node->next = NULL;

4. If the Linked List is empty, then make the new node as head

    if (\*head\_ref == NULL)

    {

       \*head\_ref = new\_node;

       return;

    }

5. Else traverse till the last node

    while (last->next != NULL)

        last = last->next;

6. Change the next of last node

    last->next = new\_node;

    return;

**c.In the Middle:**

**Step 1 -** Allocate memory and store data for new node

struct node \*newNode;

newNode = malloc(sizeof(struct node));

newNode->data = 4;

**Step 2 –** Traverse to node just before the required position of new node

struct node \*temp = head;

    if(temp->next != NULL) {

        temp = temp->next;

    }

}

**Step 3 -** Change next pointers to include new node in between

newNode->next = temp->next;

temp->next = newNode;

**ii.Deletion:**

**a.At Beginning:**

**Step 1 -** The first node of the list is to be deleted, therefore, we just need to make the head points to the next of the head. This will be done by using the following statements.

ptr = head;

            head = ptr->next;

**Step 2 -** Now, free the pointer ptr which was pointing to the head node of the list. This will be done by using the following statement.

             free(ptr)

**b.At End-**

**Step 1** - If head → next = NULL , then the only node head of the list will be assigned to null.

ptr = head

        head = NULL

        free(ptr)

**Step 2 -** The condition head → next = NULL would fail and therefore, so traverse the node in order to reach the last node of the list.

For this purpose, declare a temporary pointer temp and assign it to the head of the list and also keep track of the second last node of the list. For this purpose, two pointers ptr and ptr1 will be used where ptr will point to the last node and ptr1 will point to the second last node of the list.

ptr = head;

                while(ptr->next != NULL)

                {

                    ptr1 = ptr;

ptr = ptr ->next;

                }

**Step 3 -** Now, make the pointer ptr1 point to the NULL and the last node of the list that is pointed by ptr will become free. It will be done by using the following statements.

                ptr1->next = NULL;

                free(ptr);

**c.In the Middle:**

**Step 1 –** If head is null.

if (head == NULL)

        return NULL;

    if (head->next == NULL)

    {

        delete head;

        return NULL;

    }

**Step 2 –** If had is not null then. Initialize slow and fast pointers to reach the middle of the linked list .

    struct Node \*slow\_ptr = head;

    struct Node \*fast\_ptr = head;

**Step 3 -** Find the middle and previous of the middle.

    struct Node \*prev; // To store previous of slow\_ptr

    while (fast\_ptr != NULL &&fast\_ptr->next != NULL)

    {

fast\_ptr = fast\_ptr->next->next;

prev = slow\_ptr;

slow\_ptr = slow\_ptr->next;

    }

**Step 4 -** Delete the middle node

prev->next = slow\_ptr->next;

    delete slow\_ptr;

    return head;

}

**6.7.2. Doubly Linked List**

**i. Insertion:**

**a.At end-**

**Step 1 -** Create a HEAD pointer which points to the first node of the linked list.

**Step 2 -** Create a new node TEMP.

Temp ->Data = New\_Value;

Temp->Prev = Null;

Temp->Next = Null;

**Step 3** –

if (HEAD ==NULL)

Then, move the address of the new node TEMP into HEAD

else,

Traverse pointer until reached the last node,

Assign HEAD to TEMP->prev and TEMP to Head->next.

**b.At beginning-**

**Step 1 -** Allocate the space for the new node in the memory. This will be done by using the following statement.

ptr = (struct node \*)malloc(sizeof(struct node));

**Step 2 -** Check whether the list is empty or not. The list is empty if the condition head == NULL holds. In that case, the node will be inserted as the only node of the list and therefore the prev and the next pointer of the node will point to NULL and the head pointer will point to this node.

ptr->next = NULL;

ptr->prev=NULL;

ptr->data=item;

        head=ptr;

**Step 3 -**  In the second scenario, the condition **head == NULL** become false and the node will be inserted in beginning. The next pointer of the node will point to the existing head pointer of the node. The prev pointer of the existing head will point to the new node being inserted.

ptr->next = head;

head→prev=ptr;

**c.In the middle-**

**Step 1 -** Allocate the memory for the new node. Use the following statements for this.

ptr = (struct node \*)malloc(sizeof(struct node));

**Step 2 -** Traverse the list by using the pointer temp to skip the required number of nodes in order to reach the specified node.

temp=head;

    for(i=0;i<loc;i++)

   {

       temp = temp->next;

if(temp == NULL) // the temp will be //null if the list doesn't last long //up to mentioned location

       {

return;

       }

   }

**Step 3 -** The temp would point to the specified node at the end of the for loop. The new node needs to be inserted after this node therefore we need to make a few pointer adjustments here. Make the next pointer of ptr point to the next node of temp.

ptr → next = temp → next;

**Step 4 -** Make the **prev** of the new node ptr point to temp.

ptr → prev = temp;

**Step 5 -** Make the **next** pointer of temp point to the new node ptr.

temp → next = ptr;

**Step 6 -** Make the **previous** pointer of the next node of temp point to the new node.

temp → next → prev = ptr;

**ii.Deletion:**

**a.At end-**

**Step 1 -** Traverse to second last element

struct node\* temp = head;

while(temp->next->next!=NULL){

  temp = temp->next;

}

**Step 2** - Change its next pointer to null

temp->next = NULL;

**b.At Beginning-**

**Step 1 -** Copy the head pointer to pointer ptr and shift the head pointer to its next.

Ptr = head;

    head = head → next;

**Step 2 -** Now make the prev of this new head node point to NULL. This will be done by using the following statements.

head → prev = NULL

**Step 3 -** Now free the pointer ptr by using the **free** function.

free(ptr)

**c.In the Middle-**

**Step 1 -** Traverse to element before the element to be deleted

for(int i=2; i< position; i++) {

    if(temp->next!=NULL) {

        temp = temp->next;

    }

}

**Step 2 -** Change next pointers to exclude the node from the chain

temp->next = temp->next->next;

**6.7.3. Circular Linked list**

**i.** **Insertion:**

**a.At beginning**

**Step 1 -** Allocate the memory space for the new node by using the malloc method of C language.

struct node \*ptr = (struct node \*)malloc(sizeof(struct node));

**Step 2** - In the first scenario, the condition head == NULL will be true. Since, the list in which we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only. We also need to make the head pointer point to this node.

if(head == NULL)

        {

            head = ptr;

ptr -> next = head;

        }

**Step 3 -** In the second scenario, the condition head == NULL will become false which means that the list contains at least one node. In this case, we need to traverse the list in order to reach the last node of the list. This will be done by using the following statement.

temp = head;

            while(temp->next != head)

                temp = temp->next;

**Step 4 -** At the end of the loop, the pointer temp would point to the last node of the list. Since, in a circular singly linked list, the last node of the list contains a pointer to the first node of the list. Therefore, we need to make the next pointer of the last node point to the head node of the list and the new node which is being inserted into the list will be the new head node of the list therefore the next pointer of temp will point to the new node ptr.

temp -> next = ptr;

**Step 5** - The next pointer of temp will point to the existing head node of the list.

ptr->next = head;

**Step 6 -** Now, make the new node ptr, the new head node of the circular singly linked list.

head = ptr;

**b.At End-**

**Step 1 -** Allocate the memory space for the new node by using the malloc method of C language.

struct node \*ptr = (struct node \*)malloc(sizeof(struct node));

**Step 2 -** In the first scenario, the condition head == NULL will be true. Since, the list in which we are inserting the node is a circular singly linked list, therefore the only node of the list (which is just inserted into the list) will point to itself only. We also need to make the head pointer point to this node. This will be done by using the following statements.

if(head == NULL)

        {

            head = ptr;

ptr -> next = head;

        }

**Step 3 -** In the second scenario, the condition head == NULL will become false which means that the list contains at least one node. In this case, traverse the list in order to reach the last node of the list.

temp = head;

            while(temp->next != head)

                temp = temp->next;

**Step 4 -** At the end of the loop, the pointer temp would point to the last node of the list. Since, the new node which is being inserted into the list will be the new last node of the list. Therefore, the existing last node i.e. temp must point to the new node ptr.

temp -> next = ptr;

**Step 5 -** The new last node of the list i.e. ptr will point to the head node of the list.

ptr -> next = head;

**c.In the Middle-**

**Step 1:** Declaring head and tail pointer as null.

struct node \*head = NULL;

struct node \*tail = NULL;

int size = 0;

**Step 2:** Creating a new node.

    struct node \*newNode = (struct node\*)malloc(sizeof(struct node));

newNode->data = data;

**Step 3:** Check if the list is empty. If the list is empty, both head and tail would point to a new node.

if(head == NULL){

head = newNode;

        tail = newNode;

newNode->next = head;

}else {

        tail->next = newNode;

        tail = newNode;

        tail->next = head;

    }

**Step 4:** Adding node in the middle.

if(head == NULL){

        //If the list is empty, both head and tail would point to a new node.

        head = newNode;

        tail = newNode;

newNode->next = head;

    }

else{

           struct node \*temp, \*current = NULL;

        //Store the mid-point of the list

        int count = (size % 2 == 0) ? (size/2) : ((size+1)/2);

        //temp will point to head

        temp = head;

for(i = 0; i < count; i++){

            //Current will point to node previous to temp.

            current = temp;

            //Traverse through the list till the middle of the list is reached

            temp = temp->next;

        }

        //current will point to new node

        current->next = newNode;

        //new node will point to temp

newNode->next = temp;

   }

**ii.Deletion:**

**a.At Beginning**

**Step 1 -** If the list is empty then the condition head == NULL will become true, in this case, print underflow on the screen and make exit.

if(head == NULL)

    {

printf("\nUNDERFLOW");

return;

    }

**Step 2 -** If the list contains a single node then, the condition head → next == head will become true. In this case, delete the entire list and make the head pointer free.

if(head->next == head)

{

    head = NULL;

    free(head);

}

**Step 3 -** If the list contains more than one node then, in that case, traverse the list by using the pointer ptr to reach the last node of the list.

ptr = head;

while(ptr -> next != head)

ptr = ptr ->next;

**Step 4 -** At the end of the loop, the pointer ptr points to the last node of the list. Since, the last node of the list points to the head node of the list. Therefore, this will be changed as the last node of the list will point to the next of the head node.

ptr->next = head->next;

**Step 5 -** Free the head pointer by using the free() method in C language.

free(head);

**Step 6 -** Make the node pointed by the next of the last node, the new head of the list.

head = ptr->next;

**b.At End:**

**Step 1 -** If the list is empty then the condition head == NULL will become true, in this case, print underflow on the screen and make exit.

if(head == NULL)

    {

printf("\nUNDERFLOW");

return;

    }

**Step 2 -** If the list contains a single node then, the condition head → next == head will become true. In this case, delete the entire list and make the head pointer free.

if(head->next == head)

{

    head = NULL;

    free(head);

}

**Step 3 -** If the list contains more than one element, then in order to delete the last element, we need to reach the last node. We also need to keep track of the second last node of the list. For this purpose, the two pointers ptr and preptr are defined.

ptr = head;

while(ptr ->next != head)

        {

preptr=ptr;

ptr = ptr->next;

        }

**Step 4 -** Make just one more pointer adjustment. Make the next pointer of preptr point to the next of ptr (i.e. head) and then make pointer ptr free.

preptr->next = ptr ->next;

        free(ptr);

**c.In the Middle:**

**Step 1:** Find length of list

    int len = Length(\*head);

    int count = 1;

    struct Node \*previous = \*head, \*next = \*head;

**Step 2:** Check if the list doesn't have any nodes.

    if (\*head == NULL) {

printf("\nDelete Last List is empty\n");

return;

    }

**Step 3:** Check whether given index is in list or not

    if (index >= len || index < 0) {

printf("\nIndex is not Found\n");

return;

    }

**Step 4:** Delete first node and traverse to the last node and  delete the required node.

    if (index == 0) {

DeleteFirst(head);

return;

    }

    while (len> 0) {

        if (index == count) {

            previous->next = next->next;

            free(next);

return;

        }

        previous = previous->next;

        next = previous->next;

len--;

        count++;

    }

return;

}

**6.8. TIME COMPLEXITY**

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Operations →** | **Insertion** | | | **Deletion** | | |
| **Types of Linked list**  **↓** | **At Starting** | **At Middle** | **At End** | **At Starting** | **At Middle** | **At End** |
| Singly Linked List | 0(1) | 0(n) | 0(1) | 0(n) | 0(n) | 0(n) |
| Doubly Linked List | 0(1) | 0(n) | 0(1) | 0(1) | 0(n) | 0(n) |
| Circular  Linked List | 0(1) | 0(n) | 0(1) | 0(n) | 0(n) | 0(n) |

**Example 1 : Consider the below code, which deletes a node from the beginning of a list:**

Void deletefront()

{

 If (head == Null)

return;

 else

{

 ……………….

 .………………

 ……………….

}

}

**Which lines will correctly implement else part of the above code?**

(a)if (head ->  next == Null )

      head = head ->next;

(b) if ( head == tail )

      head = tail = Null;

      else

      head = head ->next;

(c) if ( head == tail == null)

      Head = head ->next;

(d) head = head ->next;

**Example 2 : What does the following function do for a given linked list :**

void fun1(struct Node\* head)

{

  if(head == NULL)

    return;

  fun1(head->next);

  printf("%d  ", head->data);

}

**Sol:** fun1() prints the given Linked List in reverse manner. For Linked List 1->2->3->4->5, fun1() prints 5->4->3->2->1.

**Example 3 :** Consider the following function that takes reference to the head of a Doubly Linked List as a parameter. Assume that a node of doubly linked list has the previous pointer as prev and next pointer as next.

void fun(struct node \*\*head\_ref)

{

    struct node \*temp = NULL;

    struct node \*current = \*head\_ref;

     while (current !=  NULL)

    {

        temp = current->prev;

        current->prev = current->next;

        current->next = temp;

        current = current->prev;

    }

if(temp != NULL )

        \*head\_ref = temp->prev;

}

**Assume that the reference of the head of the following doubly linked list is passed to above function.**

**1 2 3 4 5 6.**

**What should be the modified linked list after the function call?**

(a) 2 1 4 3 6 5

(b) 5 4 3 2 1 6.

(c) 6 5 4 3 2 1.

(d) 6 5 4 3 1 2

**Sol.** The given function reverses the given doubly linked list.

Hence, option C is correct.

**Example 4 :** The following C function takes a simply-linked list as input argument. It modifies the list by moving the last element to the front of the list and returns the modified list. Some parts of the code are left blank. Choose the correct alternative to replace the blank line.

              typedef struct node

{

  int value;

  struct node \*next;

}Node;

Node \*move\_to\_front(Node \*head)

{

  Node \*p, \*q;

  if ((head == NULL: || (head->next == NULL))

    return head;

  q = NULL; p = head;

  while (p->next !=NULL)

  {

    q = p;

    p = p->next;

  }

  return head;

}

(a) q = NULL; p->next = head; head = p;

(b) q->next = NULL; head = p; p->next = head;

(c) head = p; p->next = q; q->next = NULL;

(d) q->next = NULL; p->next = head; head = p;

**Sol**: Option D is correct.

**\*\*\*\***