2. Pointers

2.1 INTRODUCTION

- Pointers are used in C programming as it provides many benefits to programmers.
- Pointers are more efficient in handling arrays and data tables.
- Pointers are used to return multiple values from a function through function parameters
- Using pointer array to character strings is useful in saving storage space in memory.
- Pointers allow C to support Dynamic Memory Management.
- Pointers provide tools for manipulating data structures such as structures, linked lists, queues, stacks and trees.
- Pointers reduce length and complexity of programs.
- Pointers increase execution speed and reduce the program execution time.
- A pointer is derived data type in C.
- It is built from fundamental data types in C.
- It contains memory addresses as their value.
- It is used to access and manipulate data stored in memory.
- It is a distinct feature in C language.

2.2 CONCEPT

- Computers memory is sequential collection of storage cells, each cell is known as a byte and has an address associated with it.
- Address are numbers and numbered consecutively.

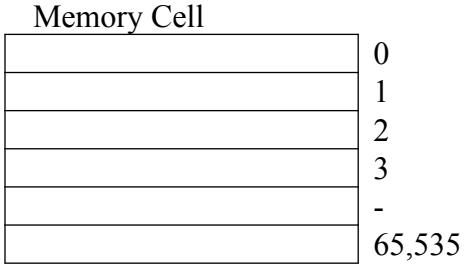

Fig. 2.1: Memory Organisation

Example: int rollnumber=20;

- Once variable is declared, system allocates memory cells on some memory address to hold value of variable.\
- Lets consider system has allocated the address 2000 for variable rollnumber which is first byte occupied by variable rollnumber.

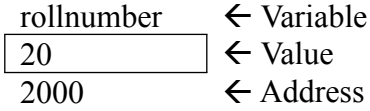

Fig. 2.2.: Variable representation

- During execution of program, system associates variable name rollnumber with address 2000. We can access the stored value 20 either by using variable name, rollnumber or as memory address 2000.
- As memory address are numbers they can be assigned to some variables that can stored in memory like other variable.
- Variables that holds memory addresses are called pointer variables.
- A pointer variable is a variable that contains an address, which is a location of another variable in memory.
- Note: As pointer is a variable its value (address of another variable) is also stored in memory on another memory address.

Example:

```
int rollnumbner = 20;
int *ptr;
ptr = &rollnumber;
```

Variable	Value	Address
rollnumber	20	2000
ptr	2000	4050

Fig. 2.3

- As the value of ptr is address of variable rollnumber i.e. 2000, we can access the value of rollnumber by using the value of ptr. Hence, we may say that the variable ptr points to variable rollnumber.
- Thus ptr is called as pointer.

2.2.1 Reference and Dereference

Reference "&" Operator:

- Pointer points to address of variable where variables value is stored in memory.
- Using & ampersand operator called as reference operator"&".
- The reference operator "&" is a unary operator.
- It is used to assign the address of the variable.
- It returns the pointer address of the variable.
- So it's called as referencing operator.

Example: int x = 10;

```
int *ptr;
ptr = &x;
```

where, ptr gets the address of variable X using referencing operator "&" then pointer ptr points to address of variable X where value 10 is stored.

Deference "*" Operator:

- To fetch the value stored on the address which is pointed by pointer dereferencing operator is used.
- "*" asterisk is dereferencing operator.
- It is also called as indirection operator.
- Its a unary operator that is used for pointer variable.
- It operates on a pointer variable and returns 1 value equivalent to the value at the pointer address.

  Example:
  ```
  int x = 10; y;
  int *ptr;
  ptr = &x;
  y = *ptr;
  ```

- With *ptr value will be fetched from the memory where pointer is pointing to variable address.
- In above example, pointer ptr points to address of variable x where its value 10 is stored.
- Using "*" variable y the value 10 is fetched. "*" which is a dereferencing operator.

Program 2.1: Program to understand reference operator "&" and deference operator"*"

```
#include<stdio.h>
void main()
{
int x = 10, y;
int *ptr;
ptr = &x;
printf("Address of x is %u \n", &x);
printf("Value of x is %d \n",x);
printf("Address of ptr is %u \n", ptr);
printf("Content of ptr is %d", *ptr);
```

Output:

```
Address of x is: 668
Value of x is: 10
Address of ptr is: 668
Value of ptr is: 10
```

2.2.2 Declaration

- In C programming each variable must be declared for its data type.
- As pointer variables contain addresses that belongs to separate data type they must be declared as pointer variable using same data type of variable to whom it will point.
- The pointer variable is declared using following format:

data type *pointer_name;

- This declaration tells three things to compiler about pointer variable:
  1. The asterisk (*) called as dereferencing operator, that variable pointer_name is a pointer variable.
  2. pointer_name needs a memory location...
  3. pointer name points to a variable of type data type (both variable and pointer vartiable needs to be of same data type).

  Example: int *ptr; Integer pointer */
  Declares a variable ptr as a pointer variable that points to an integer data type variable. (Note: type int refers to the data type of variable being pointed to by ptr and not the type of value of pointer).
  Similarly.
  float *next;
  declares next as a pointer variable to a floating point variable.

- The pointer variable is declared using symbol. Symbol can appear any where between the type name and pointer variable name.

- Following styles can be used by programmers,

  int* next;
  int * next;
  int *next;

- However, third style int *next is popular.

2.2.3 Definition

- A pointer is a variable that stores address of another variable.

  Example: int rollnumber = 20;
  int *ptr; / pointer variable */
  ptr = &rollnumber;

  In above example ptr is a pointer variable as it stores the address of another variable i.e. rollnumber. And hence ptr is called a pointer to rollnumber.

2.2.4 Initialization

- The process of assigning address of a variable to a pointer variable is known as initialization.
- Once a pointer variable is declared, an assignment variable can be used to initialize the address of variable.

  Example:

```
 int rollnumber = 20;
int *ptr; / declaration */
ptr = &rollnumber; /* intialization */
```

- It is possible to combine initialization with declaration i.e.

```
int *ptr =&rollnumber.
```

- It is mandatory that variable rollnumber must be declared before the initialization takes place.
- Ensure that pointer variable points to the corresponding type of data.

Example: float per, x;
int rollnumber, *ptr;
ptr= &per; /* wrong intialization as datatype mismatch */

With result in wrong output as trying to assign the address of float variable to an integer pointer.

- When pointer is declared to be integer type. System assumes that any address that the pointer will hold will point to an integer variable.
- Compiler will not detect such errors so care should be taken while assigning wrong pointer.
- In one step it is possible to combine declaration of data variable, declaration of pointer variable and initialization of pointer variable.

Example: int rollnumber, *ptr =&rollnumber;

/* 3 in one in single statement */

2.2.5 Use of Pointer

- After a pointer has been assigned the address of variable, now let's see how to use a pointer, how to access the value of variable using the pointer.

- To use a pointer, we have to use a *(asterisk) operator known as indirection operator or dereferencing operator.

Example:

```
int rollnumbner, n;
int *ptr;
rollnumber=20;
ptr=&rollnumber;
n=*ptr;
```

- In first line we have declared two variables rollnumber and n as integer type.
- On next line ptr is declared as a pointer variable pointing to integer variable.
- Next line assigns value 20 to rollnumber.

- The next line assigns address of rollnumber to pointer variable ptr. Next line contains the indirection operator with pointer variable ptr.
- When operator is placed before a pointer variable in an expression ie. on right side of assignment operator. the pointer returns value of a variable, for which it has stored the address.
- Thus variable n contains value 20.

Program 2.2:Write a C program to show the use of pointer.

```
#include<stdio.h>
int main()
{
int rol number, x;
int *ptr;
rollnumber=20;
ptr=&rollnumber;
x=*ptr;

printf("value of rollnumber is %d \n", rollnumber):
printf("%d is stored at %u \n", rollnumber, &rollnumber);
printf("%d is stored at %u \n", *ptr, ptr);
printf("%d is stored at %u \n", ptr, &ptr);
printf("%d is stored at %u \n", x, &x);

*ptr =25;
printf("\n now rollnumber %d \n", rollnumber);
}
```

Output:

```
value of rollnumber is 20
20 is stored at 2010
20 is stored at 2010
2010 is stored at 2012
20 is stored at 2014
now rollnumber = 25
```

- %u gives memory address. It's a type specifier when we have to display memory address value.

2.3 TYPES OF POINTERS

Following are the types of pointers in C:

1. NULL pointer

2. Dangling pointer
3. Generic pointer
4. Wild pointer
5. Complex pointer
6. Near pointer
7. Far pointer
8. Huge pointer

2.3.1 NULL Pointer

- NULL pointer is a pointer which points nothing.
- NULL pointer points the base address of segment.
- When you don't have address to be assigned to pointer they use NULL.
- Pointer initialized with NULL value is considered as NULL pointer.
- NULL is macro constant defined in header files like stdio.h, alloc.h, mem,h. stddef.h, stdlib.h.

defining NULL value

```
#define NULL 0
```

e.g.
```
float *ptr = (float *) 0;
char *ptr = (char *) 0;
int *ptr = NULL;
```

Program 2.3: Program to illustrate NULL pointer.

```
#include<stdio.h>
int main()
{
int *ptr = NULL;
printf("The value of ptr is %u", ptr);
}
```

Output:

The value of ptr is 0

2.3.2 Dangling Pointer

- Dangling pointers arise when an object is deleted or de-allocated, without modifying the value of pointer, so that pointer still points to memory location of de-allocated memory.
- -i.e. pointer pointing to a non-existing memory location is called dangling pointer.

Example:

```
#include<stdlib.h>
main()
{
char *ptr=malloc(constant_value);
```

```
-----
----
free(ptr );  /* ptr now becomes a dangling pointer */
}
```

2.3.3 Generic Pointers

- When variable is declared as pointer to type void, it is known as generic pointer.
- Since we cannot have a variable of type void the pointer will not point to any data and therefore cannot be dereferenced. Still it's a pointer and can be used to type cast to another type of pointer.
- This type of pointer is very useful when you want to pointer to point to data of different types of different times.

Program 2.4: Program to illustrate generic pointer.

```
#include<stdio.h>
int main()
{
int i;
char c;
void *the data;
i=6;
c='a';
the_data=&i;
printf("the_data points to integer value %d", *(int*) the_data);
the_data=&c;
printf("the_data points to the character %c", *(char*) the_data);
return 0;
}
```

Output:

the data points to the integer value 6
the data now points to the character a

2.3.4 Generic Pointers

Wild pointer:

- A pointer in C that has not been initialized till its first use is known as wild pointer.
- A wild pointer points to some random memory location.

Example:

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

```
main()
{
int *ptr;
/* ptr is a wild pointer, as it is not initialized yet */
printf("%d", *ptr);
}
```

2.3.5 Complex Pointer

- Operator precedence describes the order in which C reads expression.

Associativity:

- Order operators of equal precedence in an expression.
- Assign priority to pointer declaration considering precedence and associative according to following table.

Operator	Precedece
(),[]	1
*, identifier	2
Data type	3

where, (): this operator behaves as bracket operator or function operator.
[] : this operator behaves as array subscription.
* : this operator behaves as pointer operator, not as multiplication operator.

Identifier:

- Identifier is not an operator but it is name of pointer variable always priority is assigned to name of pointer.

Datatype:

- Datatype is not an operator. Data types also include modifier.

2.3.6 Near Pointer

- The pointer which can point only 64 KB data segment or segment number 8 is known as near pointer.
- This near pointer cannot access beyond the data segment like graphics video memory, text video memory etc.
- Size of near pointer is 2 byte.
- With keyword near we can make any pointer as near pointer.

Program 2.5: Program to illustrate near pointer.

```
#include<stdio.h>
int main()
{
int X =25;
int near *ptr;
ptr =&X;
printf("%d", sizeof(ptr));
}
```

Output:
2

2.3.7 far Pointer

- The pointer which can point or access whole residence memory of RAM i.e. which can access all 16 segment is known as far pointer.
- Size of far pointer is 4 bytes or 32 bit.

Program 2.6: Program to illustrate far pointer.

```
#include<stdio.h>
int main()
{
int X = 10;
int far *ptr;
ptr = & X;
printf("%d", sizeof(ptr));
}
```

Output: 4

2.3.8 Huge Pointer

- The pointer which can point or access whole the residence memory of RAM i.e. which can access all 16 segment is known as a huge pointer.
- Size of huge pointer is 4 byte or 32 bit.

Program 2.7: Program to illustrate huge pointer.

```
#include<stdio.h>
int main()
{
char huge far *p;
int far *ptr;
printf("%d %d", sizeof(p), sizeof(p), sizeof(**p);
}
```

Output:
4 4 1

where p is huge pointer, *p is far pointer and **p is char type data variable.

2.4 POINTER ARITHMETIC

- Arithmetic operators can be used with pointers.
- It's possible to use pointer variables in expressions if they are properly declared and initialized pointers.

Example:

- if P1 and P2 are properly declared and initialized pointers then following statements are valid.

```
X=*P1 * *P2; /* same as (*P1) (*P2)
sum=sum +*P1;
```

- C programming allows us to add integers or subtract integers from pointers. As well as subtract one pointer from another pointer.

  e .g.P1+4, P2-2 and P1-P2 are allowed.

We may also use shorthand operators with pointers.

```
P1 + +;
-P2;
sum-=*P2;
```

- In addition pointers can also be used with relational operators.
- Expressions such as P1> P2, P1 ==P2 and P1! = P2 are allowed.
- We may not use pointers in division or multiplication.

Example: P1/ P2 or P1* P 2 or P1/ 3 are not allowed.
Similarly, the pointers cannot be added if P1 + P2 is illegal.

2.4.1 Multiple Indirection

- C permits the pointer to point to another pointer. This creates many layers of point this situation is called as multiple indirection.
- A pointer to a pointer has similar declaration like normal pointer but have more indirections i.e. more asterisk sign.

Example:

- The pointer variable p2 contains address of pointer variable P1 which points location where actual value of variable is stored This is known as multiple indirections.

- A variable which is pointer to a pointer must be declared using additional indirection operator in front of the name of pointer e.g. int ** p2:
- This declaration, tells compiler that P is a pointer to pointer of int type variable. P2 not a pointer to an integer but it's a pointer to an integer pointer. To access the target value use indirection operator twice.

Program 2.9: Program 10 illustrate multiple indirections.

```
#include<stdio.h>
int main()
{
int m, *P1,**P2;
m = 200;
P1 = &m;  /* address of m */
P2 = &P1; /* address of P1 */
printf("%d",**P2);
}
```

Output:
200

2.5 PARAMETER PASSING

- Parameters can be passed to functions by using two methods:

1. Call by value
2. call by reference

2.5.1 Call by Value

- In call by value, value of variable is passed as parameter in function call.
- At function definition actual parameter is copied to formal parameters.
- In call by value method it's not possible to modify the value of actual parameter by formal parameter.
- Different memory is allocated for actual and formal parameters.
- In call by value actual parameter is copied into formal parameter.

Program 2.10: Program for call by value.

```
#include<stdio.h>
void change(int num)
{
printf("Before adding value inside function num = %d \n", num);
num=num+100;
printf("After adding value inside function num = %d \n", num);
}
int main()
```

```
{
int X = 100;
printf("Before function call x = %d \n", X);
change(X); /* call by value */
printf("After function call X = %d \n", X);
}
```

Output:
Before function call X = 100
Before adding value inside function num = 100
After adding value inside function num = 200
After function call X = 100

Program 2.11: Program for swapping values of two variables using call by value.

```
#include<stdio.h>
void swap(int, int); /* function prototype declaration */
int main()
{
int a = 10;
int b = 20;
printf("Before swapping the values in main a =\%d b =%d \n",a,b);
swap(a, b); /* function call with actual values */
}
void swap (int a, int b)
{
int temp;
temp = a;
a = b;
b = temp;
printf("After swapping values in function a=%d, b=%d \n",a,b) ;
}
```

Output:
Before swapping the values in main a = 10 b = 20
After swapping values in function a = 20 b = 10

2.5.2 Call by Reference

- In call by reference, address of variable is passed into function call as actual parameters.
- In call by reference the value of actual parameters can be modified by changing the formal parameters since the address of actual parameters is passed.
- In call by reference the memory allocation is similar for both formal parameters and actual parameters.
- All operations in function definition are performed on value stored at the address of actual parameters and the modified value gets stored at same address.

Program 2.12: Program for call by reference.

```
#include<stdio.h>
void change(int *num)
{
printf("Before adding value inside function num=%d", *num);
*num = *num+ 100;
printf("After adding value inside function num=%d", *num);
}
int main()
{
int x = 100;
printf("Before function call  x =%d “,x );
change(&x); /* passing reference in function */
printf("After function call x =%d”, x);
}
```

Output:

```
Before function call x = 100
Before adding value inside function num = 100
After adding value inside function num = 200
After function call x = 200
```

Program 2.13: Program for Swapping values of two variables using call by reference.

```
#include<stdio.h>
void swap(int *, int *);
int main()
{
int a = 10;
int b = 20;
printf("Before swapping the values in main a =\%d b = %d", a, b);
swap (&a, &b) ;
printf("After swapping values in main a %d, b =\%d “,a,b);
void swap (int *a, int *b)
{
int temp;
temp = *a;
*a = *b
* b = temp ;
printf("After swapping values in function a =%d b=%d", *a, *b);
}
```

Output:

```
Before swapping the values in main a = 10 b = 20
```

After swapping values in function a = 20 b = 10
After swapping values in main a = 20 b = 10

2.6 ARRAYS AND POINTERS

2.6.1 Pointer to Array

- When an array is declared, compiler allocates a base address and required amount of storage to store all elements of array in contiguous memory locations.
- The base address is location of first element i.e. index 0. Ex. a[0] of the array.
- Compiler defines array name as constant pointer to first element.

Example: int a[5] = {10, 20, 30, 40, 50}

The memory representation of above array is as follows:

a[0]	a[1]	a[2]	a[3]	a[4]	Array element
10	20	30	40	50	Values
2000	2002	2004	2006	2008	Address

- As shown base address of array a is 2000. Remaining elements stored on contiguous memory location as each int required 2 bytes.

- An array name without subscript is a pointer to first element in the one, two and multidimensional array.

  i.e. a = 2000
  &a[0] = 2000

- A specified earlier compiler defined a as a constant pointer pointing to first element So a and &a[0] is stored on location 2000.

- A pointer ptr can be used to point an array a

  ptr = a; is equivalent to ptr = &a[0];

- It's possible to access every value of a using ptr++ to move from one array element to another.

- Lets see the relation between ptr and a shown

  ptr = &a[0] = 2000
  ptr+1=&a[1] = 2002
  ptr+2= &a[2] = 2004
  ptr+3= &a[3] = 2006
  ptr+4 = &a[4] = 2008

- Address of element is calculated using its index and scale factor of data type.

Example:

address of a[3] = base address + (3 * scale factor of int)= 2000 + (3 * 2) = 2006

- Pointers can be used to access array elements instead of using array elements.
- Pointer accessing method is much faster than array indexing i.e. * (ptr + 3) gives value of a[3].

Program 2.14: Program to calculate sum of all elements in an array.

```
#include<stdio.h>
void main()
{
int *ptr, sum, 1;
int a[5] (10, 20, 30, 40, 50);
i = 0
ptr=a; // initialising pointer with base address of a
printf("Element value Address \n \n");

while(i < 5)
{
printf("a[%d] %d %u \n", i, *ptr, ptr);

sum = sum + *ptr;  //Accessing array element with *p ptr */

i++, ptr++;                /* incrementing pointer */
}
printf("\n sum=%d \n", sum);
printf("\n a[0] =%u \n",&a[0]);
printf("\np=%u \n",ptr);
}
```

- Pointers can be used to manipulate two dimensional arrays as well.
- In one dimension array a, elements a[i] can be represented using expression,

  *** (a + i) or *(ptr + i)**

- In two dimensional array elements can be represented using expression as,

  *** (* (a + i) + j) or *(* (ptr + i) + j)**

2.6.2 Arrays of Pointers

Array of pointers:

- We can have an array of pointer variables like an array of integer, float or char.
- As pointer variable contain an address, an array of pointers is collection of addresses.

- Pointer elements are stored in memory just like elements of other array.
  Syntax: data_type *array name[size];

  Example:
  int *ptr[5];
  ptr is an array of 5 integer pointers

Program 2.15: Program to illustrate array of pointers.

```
#include<stdio.h>
void  main()
{
        int arr [3] = {1, 2, 3};
        int i,* ptr[3] ;
        for( i = 0; i<3; i++)
                ptr[i] = arr + I            /* Assigning address of elements */
        for( i = 0;  i < 3 i++)
                printf("address=%u \t  value=%d \n", ptr[i], *ptr[i]);
}
```

output:
address = 3000 Value = 1
address 3802 Value = 2
address 3084 Value = 3

Initialization of array of pointer:

- An array of pointers can be initialized during declaration as follows:

Example:

```
static int a[] = {0, 1, 2, 3, 4};
static int *p[]= {a, a+1, a+2, a+3, a+4};
int **ptr = p;
```

Above lines can be represented as:

a[0]	a[1]	a[2]	a[3]	a[4]
2000	2002	2004	2006	2008

p[0]	p[1]	p[2]	p[3]	p[4]	ptr
2000	2002	2004	2006	2008	3000
3000	3002	3004	3006	3008	4000

Example of pointers, pointers to pointer.

Array of pointers to strings.

Example:

```
char *message[6]= {"Pointers", "are", "Intresting", "but", "need", Practice");
for(i=0; i<6; 1++)
        printf("%s", message[i]);
```

Above lines can be represented as:

Message[0]	→	p	o	i	n	t	e	r	s	\0

Message[1]	→	a	r	e	\0

Message[2]	→	i	n	t	r	e	s	t	i	n	g	\0

Message[3]	→	b	u	t	\0

Message[4]	→	n	e	e	d	\0

Message[5]	→	p	r	a	c	t	i	c	e	\0

To access jth character in ith message *(message[i] + j)

2.7 FUNCTIONS AND POINTERS

2.7.1 Passing Pointer to Functions

- When an array is passed to a function as an argument, only the address of first element of an array is passed, but not the actual values of the array elements.
- Example: if X is an array, when a function is called like sort (X), the address of X[0] is passed to the function sort.
- Function then uses this address for manipulating the array elements.
- Similarly, it's possible to pass the address of variable as an argument to function.
- While passing addresses as argument to function, the receiving parameters receive address and the receiver should be a pointer.
- The process of calling a function using pointers to pass address of variables is known as call by reference.
- The function which is called by using reference can change the actual values of variable used in call.

Program 2.16: Program for passing pointer to function.

```
#include<stdio.h>
void modify(int *, int *)/ prototype declaration */
int main()
{
    int x1 = 2, x2 = 3;
    printf("modified values x1 = %d    x2 =%d",x1, x2 ) ;
}

void modify(int *ptr1, int *ptr2)
{
   *ptr1 = 100;
   *ptr2 = 200;
}
```

2.7.2 Returning Pointer from Function

- Function can return a single value by its name or return multiple values through pointer parameters
- As pointers are data types, we can also force a function to return a pointer to calling function.
- The format is as follows:

  Pointer_datatype *function_name(parameter list)

  Example: int *f1(int);

  where f1 is a function accepting an integer and returning pointer to an integer.

Program 2.17: Program for returning pointer from function.

```
int * larger (int *, int * ); / function declaration */
int main()
{
        int m = 20;
        int n = 30;
        int *ptr;
        ptr=larger (&m, &n);            /* function call/
        printf("After calling function larger the value is =%d", *ptr);
}

int *larger(int *a, int *b)
{
if (* a> * b)
        return(a);
else
        return(b);
}
```

Output:
after calling function larger the value is = 30

- The function larger receives address of m and n, decides which one is larger using pointers. a & b then returns address of location. Return value is assigned to pointer variable ptr in calling function.

2.7.3 Function Pointer

- C has a powerful feature i.e. function pointer.
- Even though a function is not a variable it still has a physical memory location in memory.
- A functions address is starting address of code of function in memory. Function address assigned to a pointer is entry point to function. The pointer can be used in place of function name.
- These features allow function to be passed as argument to other functions.
- Format of declaring pointer to a function:

returntype (*pointer variable) (functions_argument_list);

Example: int (*ptr) (int, int);

/* ptr is pointer to function accepting 2 integer values and returning an integer value */

Note: () around *ptr is necessary else

int * ptr (int, int); would declare ptr as a function returning a pointer to int type.

- We can make a function pointer to point to a specific function by assigning the name of function to pointer.

Example: double mul(int, int);
double (* p1)()
P1= mul :

Declare P1 as a pointer to function and mul as function. P1 point to function mul to call function.

(*P1) (x, y) is equal to mul (x, y)

Program 2.18: Program for pointer to function:

```
#include<stdio.h>
int fact(int n)
{
  if(n<=1)
    return(1);
  else
```

```
    return(n*fact(n-1));
}
int main()
{
    int(*ptr) (int); // pointer to function with prototype of fact function
    int ans;
    int n;
        ptr = fact; /* assigns address of fact to ptr */
        printf("Enter the number whose factorial is required");
        scanf("%d", &n);
        ans = ptr(n);   /* call to function fact using ptr */
        printf("\n the result is %d", ans);
}
```

2.7.4 Pointers and Const

2.7.4.1 Pointer to Constant Objects

- A pointer to constant object can be declared as

  ```
  const data_type *ptr_name.
  const int *ptr;
  ```

  Example:
  ```
  int i = 10;
  const int *ptr;
  ptr = &i;
  ```

  ptr is a pointer to i. It means the contents pointed to by ptr cannot be changed.

  ```
  *ptr = 20; is invalid
  ```

  However ptr itself can be changed

  ```
  ptr++ is valid
  ```

2.7.4.2 Constant Pointers

- A constant pointer cannot be modified however data item to which it points can be modified.

  Example:
  ```
  int i = 20;
  int const ptr;  /* declares to constant pointer ptr */
  ptr =&i;
  *ptr = 30;        /* valid */
  ptr++;            /* invalid
  ```

- Const int *const ptr; will not allow any modification to be made to ptr or the integer to which it points to

2.8 DYNAMIC MEMORY MANAGEMENT

- Once variables and arrays are declared, explicitly memory is allocated to variables as per type and size of array.

- Such memory allocation is called static memory allocation. In other way programmer has to specify how much memory is required.
- For example, when we declare an array, we have to specify its size. Sometimes, we do not know how many elements are used by users, memory may be wasted or more memory is required to put more elements. This problem is solved in dynamic memory allocation method.
- We can allocate and deallocate memory at runtime is known as dynamic memory allocation.

- In C there are four library routines known as memory management functions.
- These functions help us built complex application programs that use available memory intelligently.
- The header file alloc.h contains prototypes of dynamic memory allocation functions.

Memory allocation functions:

Function	Task
malloc()	malloc function allocates requested size of bytes and returns a pointer to first by of allocated space.
calloc()	calloc function allocates space for an array of elements. It initialized all array elements to zero and then returns a pointer to memory.
free()	free function frees previously allocated space.
realloc()	realloc function modifies size of previously allocated space.

Memory allocation process:

Following is the process to allocate memory in C program.

<table>
<tr><td>Local Variables</td><td>Stack</td></tr>
<tr><td>Free Memory</td><td>Heap</td></tr>
<tr><td>Global Variables</td><td rowspan="2">Permanent Storage Area</td></tr>
<tr><td>C Program Instructions</td></tr>
</table>

- As shown above in conceptual view of storage of program in memory.
- Local variables are stored in area called stack.
- The program instructions and global and static variables are stored in an area called as permanent storage area.
- The memory space between two regions is available for dynamic memory allocation during program execution known a heap.
- The size of heap keeps changing during program execution due to creation and freeing of variables that are local to functions and blocks.
- Because of this it’s possible to encounter memory 'overflow' during dynamic allocation process.
- In such situation above mentioned memory allocation functions return a NULL pointer.

2.8.1 Memory Allocation

Memory can be allocated in two ways:

1. Allocating a block of memory using malloc().

2 Allocating multiple blocks of memory using calloc().

1. Allocation of Block of memory using malloc

- A block of memory can be allocated using function malloc.
- It reserves a block of memory of specified size and returns pointer of void type.
- Following is its format:

Ptr= (cost_type*) malloc(byte_size)

where ptr is a pointer of cast type. The malloc returns pointer to cast type to memory area of specified byte_size.

- Example: ptr = (int *)malloc(sizeof(int));

After execution of this statement a memory space of 2 bytes is reserved and address of first byte to allocated memory block is assigned to pointer ptr.

- ptr = (char *) malloc(20);

During execution of this statement memory of 20 bytes will be allocated and reserves address of first byte of allocated memory is assigned to pointer (ptr of type character).

Program 2.19: Program for memory allocation using malloc().

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void main()
{
        char name[100];
        char *description;
        strcpy(name, "FYBCA");
/* allocate memory dynamically */
        description = malloc( 200*sizeof(char));

        if( description==NULL)
         fprintf(stderr, "Error unable to allocate required memory\n");
        else
          strcpy(description, "I am Student of FYBCA ");
        printf("\nName = %s\n", name);
        printf("Description: %s\n", description);
```

}

2. Allocating multiple blocks of memory calloc

- Calloc is another memory allocation function.
- It is used for allocating memory for storing derived data type such as arrays and structures.
- malloc allocates single block of storage while calloc allocates multiple block of storage each of same size.
- It sets of byte to zero.
- Format: **ptr (cost_type*) calloc(n, elem_size);**
  ptr is a pointer of cast_type.
  calloc allocates contiguous space for n blocks each of size elem_size bytes.
- All bytes initialized to zero and a pointer to first byte of allocated region is returned.
- If enough space is not available a NULL pointer is returned.

Example:

```
struct employee
        {
                char name[20];
                float age;
                int emp_id;
        };
        typedef struct employee el;
                el *ptr;
                int n = 50;
                ptr=(el*) calloc(n, sizeof(el));
```

- In above example e1 is of type struct employee having three members name, age and emp_id.
- calloc allocates memory to hold data for 50 such records.

2.8.2 Releasing Memory

Releasing allocated space using free function:

- After successful dynamic memory allocation, its need to free the allocated memory when it's not required any more.
- Releasing memory after use is helpful in proper memory utilization.
- This way block of memory which is freed can be used in future.
- Memory can be released using free function.
  free(ptr);
- ptr is a pointer to a memory block which has already been created by malloc or calloc.

2.8.3 Resizing Memory

Reallocating size of block: realloc

- It is possible that we can change the allocated memory size with realloc function.
- Sometimes allocated memory is less or required more memory than allocated memory.

- In both situations we can use the function realloc.
  ptr=realloc(ptr, newsize):
- The above format is used if earlier allocation is done by statement
  ptr=malloc(size);
- The realloc function allocates a new memory space of size newsize to pointer variable ptr and returns a pointer to first type of new memory block. Newsize may be larger or smaller than the size.

Program 2.20: Program for resizing and releasing memory.

```
#include<stdio.h>
#include<stdlib.h>
#include<string.h>
void main()
{
        char name[100];
        char *description;
        strcpy(name, "FYBCA");
/* allocate memory dynamically */
        description = malloc( 30*sizeof(char));

        if( description==NULL)
         fprintf(stderr, "Error unable to allocate required memory\n");
        else
          strcpy( description, "I am Student of FYBCA ");
        printf("\nName = %s\n", name);
        printf("Description: %s\n", description);
  //realloc
  description=realloc(description, 100*sizeof(char));
  if( description==NULL)
          fprintf(stderr, "Error unable to allocate required memory\n");
   else
          strcpy( description, "I am Student of FYBCA, We study in MIT ACS college, alandi ");
        printf("\nName = %s\n", name);
        printf("Description: %s\n", description);
        free(description);
}
```

2.9 DANGLING POINTERS / MEMORY LEAK

- If a pointer still references the original memory after it has been freed, it is called dangling pointer.
- The pointer does not point to a valid object. This is sometimes referred to a premature free.
- A dangling pointer occurs when you are pointing to an object or other area of memory that is not valid anymore. In C, this may happen for several reasons, such as accessing a pointer that was freed.

- A memory leak is a point of a program where some memory becomes inaccessible even though it was not released by the system. Memory leaks are possible in C, only changing the mechanism by which memory is released.

- A dangling pointer is a pointer to storage that is being used in for another purpose. Typically, the storage has been deallocated. Operation dispose(p) leaves 'p' dangling.
- Storage that is allocated but in inaccessible is called garbage. Programs that create garbage are said to have memory leaks. Assignment to pointers can lead to memory leak.
- For example: Suppose a and b are pointers to cells. The pointer assignment, a: = b
  Leaves a and b pointing to same cell.
  The cell that b was pointing to, as shown in Fig

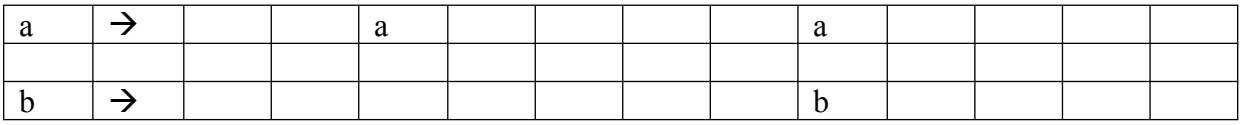

Fig. 2.5. Pointer Assignment can Result in the Memory Leak and Dispose can Result in Dangling Pointers.

The cell that ‘a’ pointed to is still in memory, but it is inaccessible, so we have a memory leak. Now, dispose(a) deallocates the cell that ‘a' points to. It leaves 'a' and 'b' both dangling, since they both point to same cell.

Summary

- Pointer variable is nothing but a variable that contains an address of another variable in memory.
- Types of pointers specifies details of how pointer can be used in different ways to make program execution easy and fast.
- Pointer arithmetic gives details of how pointer variables can be used in expression. 10. Multiple indirection pointer variable contains address of another pointer variable which points to location that contains desired value.
- Call by value, call by reference gives details about parameter passing ways.
- Functions and pointers given details of how pointer can be passed to function, how we can return pointer from function etc.
- Arrays and pointers give details of pointer to array and how array of pointer can be used.
- Dynamic memory allocation functions can be used to allocate single block, multiple block, reallocating and freeing memory.