POINTERS

INTRODUCTION

The concept of pointer is another important concept in C. Although it is bit confusing for the beginners, it is very powerful and become a powerful tool and handy to use in programming once the concept is clearly understood.
The use of pointer makes the code more efficient and compact. Some of them are categorized below:

- Accessing array elements
- Returning more than one value from a function
- Accessing dynamically allocated memory. Hence execution speed increases.
- Due to the dynamic concept memory wastage can be avoided.
- Implementing of data structures like linked list, tree, graph can be done using pointer concept

THE & AND * OPERATOR

In C we use the **&** operator to access the memory (address of variable) of a variable and * operator to access value stored at a particular address. **&** is called **address of** or **reference** operator and ***** is called **value at address** or **dereference** operator.

Consider the following program:

```
#include <stdio.h>
void main( ){
    int i ;
    i=3;
    printf ( "\nValue of i = %u", i ) ;
    printf ( "\nAddress of i = %u", &i ) ;
}
```

The output of the above program would be:

```
Value of i = 3
Address of i = 6485
```

Here, &i gives the address of i. So &i is 6485. So & is known as address operator. Another operator i.e. ***** is used when to get the value stored at a particular address. So it is known as ‘value at address’ operator.
Consider the following program:

```
#include <stdio.h>
void main( ){
        int i = 3 ;
        printf ( "\nAddress of i = %u", &i ) ;
        printf ( "\nValue of i = %d", i ) ;
        printf ( "\nValue of i = %d", *( &i ) ) ;
}
```

The output of the above program would be:

```
Address of i = 6485
Value of i = 3
Value of i = 3
```

Here *(&i) gives the value stored at location given by &i which is the value of i. So here printing *(&i) is same as i.

THE POINTER DEFINITION

A pointer is a memory variable that contains the address of another variable. A pointer of one data type can store the address of same data type variable. i.e. an integer pointer can only store the address of integer variable. It can't store the address of any other data type variable like float, char, etc.

The **declaration syntax** of pointer is :

$$\boxed{\textbf{data\_type * pointer\_name ;}}$$

Where,

- The data type can be any type identifying the type of variable
- The asterisk (*) tells that the variable **pointer_name** is a pointer variable
- **pointer_name** is any valid identifier name defining the pointer variable of specified data type

Example: int *p;

The **syntax of pointer initialization** is:

$$\boxed{\textbf{pointer\_name = \& variable;}}$$

e.g. p=&a;

Here pointer p is assigned with the address of the variable a. As we already know * operator gives the value stored at a particular location, ***p** gives the value stored at address represented by **p** i.e. **&a** . At the address **&a** the value of **a** has been stored. So, ***P** gives the value of a.

Consider the following example:

```
#include <stdio.h>
void main( ){
    int i;
    i= 3 ;
    int *p ;
    p = &i ;
    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nAddress of i = %u", p ) ;
    printf ( "\nAddress of p= %u", &p ) ;
    printf ( "\nValue of p = %d", p ) ;
    printf ( "\nValue of i = %d", i  ) ;
    printf ( "\nValue of i = %d", *( &i ) ) ;
    printf ( "\nValue of i = %d", *p) ;
}
```

The output of the above program would be:

```
Address of i = 6485
Address of i = 6485
Address of p = 3276
Value of p= 6485
Value of i = 3
Value of i = 3
Value of i = 3
```

Pointer to a Pointer

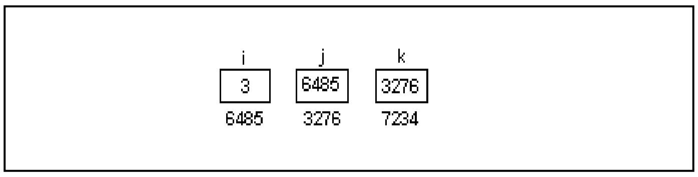
The concept of pointer can be further extended. We know that a pointer is a variable which stores the address of another variable of same data type. Now this variable itself could be another pointer. Thus, we now have a pointer which contains another pointer's address. The following example should make this point clear.

```
#include <stdio.h>
void main( ){
    int i = 3 ;
    int *j ;
    int **k ;
    j = &i ;
    k = &j ;
    printf ( "\nAddress of i = %u", &i ) ;
    printf ( "\nAddress of i = %u", j ) ;
    printf ( "\nAddress of i = %u", *k ) ;
    printf ( "\nAddress of j = %u", &j ) ;
    printf ( "\nAddress of j = %u", k ) ;
    printf ( "\nAddress of k = %u", &k ) ;

    printf ( "\n\nValue of j   = %d", j ) ;
    printf ( "\nValue of k   = %d", k ) ;
    printf ( "\nValue of i   = %d", i ) ;
    printf ( "\nValue of i   = %d", *( &i ) ) ;
    printf ( "\nValue of i   = %d", *j ) ;
    printf ( "\nValue of i   = %d", **k ) ;
}
```

The output of the above program would be:

```
Address of i = 6485
Address of i = 6485
Address of i = 6485
Address of j = 3276
Address of j = 3276
Address of k = 7234

Value of j = 6485
Value of k = 3276
Value of i = 3
Value of i = 3
Value of i = 3
Value of i = 3
```

The following memory map would help you in tracing out how the program prints the above output.



Observe how the variables i, j and k have been declared

```
int i;
int *j;
int **k;
```

Here, i is an ordinary **int**, **j** is a pointer to an **int**, whereas **k** is a **pointer to pointer**. In principle, there could be a pointer to a pointer's pointer and so on. There is no limit on how far can we go on extending this definition. This definition overall called as pointer to a pointer.

POINTER ARITHMETICS

Like other variables, pointer variables can be used in expressions. For examples , if p1 and p2 are properly declared and initialized pointers, then the following are valid.

```
y=(*p1) * (*p2);
sum=sum+*p1;
z=*p2 + 10;
```

E.g. Program for all the arithmetic operations on two variables

```
#include <stdio.h>
void main( ){
    int a=5,b=2;
    int *pa,*pb;
    pa=&a;
    pb=&b;
    printf("\n Addition of %d and %d is : %d", *pa, *pb, *pa+ *pb);
    printf("\n Difference of %d and %d is : %d", *pa, *pb, *pa- *pb);
    printf("\n Multiplication of %d and %d is : %d", *pa, *pb, *pa*( *pb));
    printf("\n Division of %d by %d is : %f", *pa, *pb, *pa/float(*pb));
    printf("\n Modulus of %d by %d is : %d", *pa, *pb, *pa% (*pb));
}
```

The **output** of this program is:

```
    Addition of 5 and 2 is : 7
    Difference of 5 and 2 is : 3
    Multiplication of 5 and 2 is : 10
    Division of 5 by 2 is : 2.500000
    Modulus of 5 by 2 is : 1
```

FUNCTIONS RETURNING POINTERS

The way functions return an **int**, **float**, **double** or any other ata type, it can even return a pointer. However, to make a function returning pointer it has to be explicitly mentioned in the calling function as well as in the function declaration. The function prototype will look as below:

```
return_type *function_name (argument lists);
```

Consider the following example:

```
#include <stdio.h>
void main( ){
    int *p ;
    int *fun( ) ;
    p = fun( ) ;
    printf ( "\n%u", p ) ;
}
```

```
int *fun( ){
      int i = 20 ;
      return ( &i ) ;
}
```

POINTER TO FUNCTIONS

Like a variable, function also has an address location in the memory. Therefore, possible to declare a pointer to a function. A pointer to a function is declared as follows:

data_type(*fPtr)(arguments);

This tells compiler that **fPtr** is a pointer to a function which returns the specified data type and should have similar set of arguments. The parenthesis around ***fPtr** is necessary.
E.g. int (*p)( );
Here, p is pointer to a function which returns an integer and which does not take any argument.
Consider the following example:

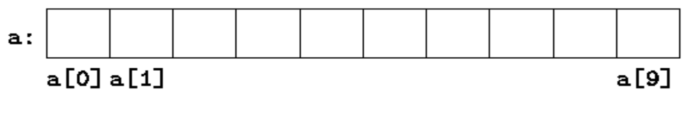
```
#include <stdio.h>
void main( ){
      int (*p)(int,int);
      int add(int,int);
      int a = 5, b = 4;
      p=add;
      printf("%d",(*p)(a,b));
}
int add(int x, int y){
      return (x+y);
}
```

Pointers and One-Dimensional Arrays

There is a strong relationship between pointers and arrays. Any operation that can be achieved by array subscripting can also be done with pointers. In general the pointer version operations are faster than array operations. Consider the condition of one dimensional array.

int a[10];

defines an array of size 10, that is, a block of 10 consecutive memory blocks named a[0], a[1], ...,a[9].



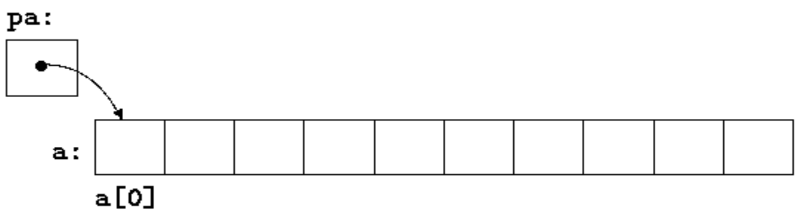
The notation a[i] refers to the i th element of the array. If pa is a pointer to an integer, declared as:

int *pa ;

then the assignment operation can be specified as:

pa = &a[0];

The above statement sets *pa to point to element zero of a ; that is, pa contains the address of a[0].*

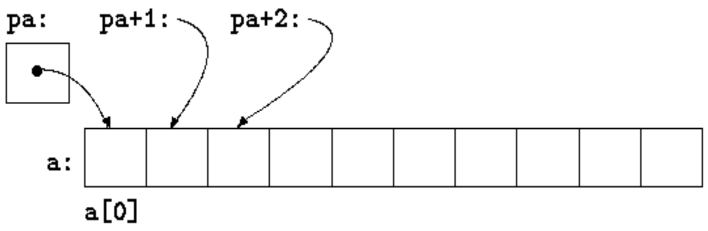


Now the assignment

*__x = *pa;__*

will copy the contents of a[0] into x.

If **pa** points to a particular element of an array, then by definition pa+1 points to the next element, pa+i points i elements after pa, and pa-i points i elements before. Thus, if pa points to a[0], *(pa+1)
refers to the contents of **a[1], pa+i** is the address of **a[i]**, and ***(pa+i)** is the contents of a[i].



These remarks are true regardless of the type or size of the variables in the array **a**. The meaning of *adding 1 to a pointer*, is that **pa+1** points to the next memory location.

The correspondence between indexing and pointer arithmetic is very close. Rather more surprising, at first sight, is the fact that a reference to **a[i]** can also be written as ***(a+i)**. In evaluating **a[i]**, C converts it to ***(a+i)** immediately; the two forms are equivalent. Applying the operator & to both parts of this equivalence, it follows that **&a[i]** and **a+i** are also identical. As the other side of this coin, if **pa** is a pointer, expressions might use it with a subscript; **pa[i]** is identical to ***(pa+i)**. In short, an array with index expression is equivalent to one written as a pointer with offset.

There is one *difference* between an array name and a pointer that must be kept in mind. A pointer is a variable. So **pa=a** and **pa++** are legal. But an array name is not a variable. So expressions like **pa=a** and **pa++** are illegal.

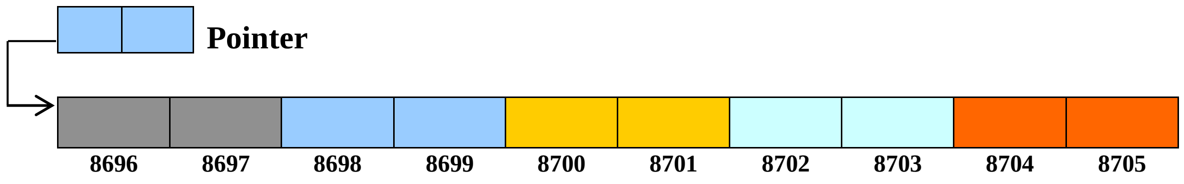
Consider the following example:

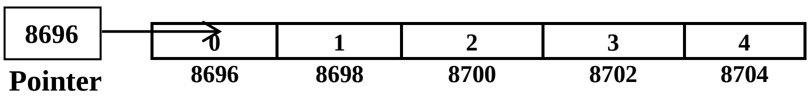
```
#include <stdio.h>
void main(){
        int a[5]={10,11,20,32,14};
        int *p,i;
        p=a;
        printf("Content\tAddress\n");
        for(i=0;i<5;i++){
            printf("%d\t",*(p+i)); /*Prints Contents*/
            printf("%d\n",(p+i));  /*Prints Address*/
        }
}
```

The output of the above program is:

```
Content  Address
10         8696
11         8698
20         8700
32         8702
14         8704
```

Have a look to the output. The addresses are increased by 2.We have repeatedly told you that the pointers always points to the memory block(combination of one or more memory cells).As the array contains integer type data, two bytes (two memory cells) forms one block. While pointing to the block it prints the address of first cell of the block. This can be well explained through the following figure:





POINTER TO STRING

String operation can be performed using pointer. It works asame as that of accessing array elements using pointer variable. A program representing such concept ios defined below:

```
#include <stdio.h>
#include <conio.h>
void main( ){
   char
day[7][15]={"Sunday","Monday","Tuesday","Wednesday","Thursday","Friday","Saturday"}
;
   int i=0, j;
   clrscr( );
   printf("\nThe Days are : ");
   for(i=0;i<7;i++){
      j=0;
      while(day[i][j]!='\0'){
         printf("%c",*(*(day+i)+j));
         j++;
      }
     printf("\n");
  }
  getch( );
}
```

The output will be:

```
Sunday
Monday
Tuesday
```

Wednesday
Thursday
Friday
Saturday
The output is based on the pointer to pointer concept and the string elements are accessed accordingly.

DYNAMIC MEMORY MANAGEMENT IN C

There are **two** ways of memory allocation in C programming.

1. Compile-time or Static allocation (Using array)
2. Run-time or Dynamic allocation (Using pointer)

When we declare an array in a program, the associated memory space is consumed till program ends regardless of being used or not. This creates a memory blockage. Such a situation can be avoided if we allocate necessary memory when needed and then free them when the use is over. We can do this at run time or execution time. *The process of allocating memory at run time is known as* ***dynamic memory allocation*** and *releasing the allocated memory at run time is known as* ***dynamic deallocation***. We can do this by using memory management functions. The memory management functions are:

Function	Task
malloc( )	Allocates request size of bytes and returns a pointer to the first byte of the allocated space.
calloc( )	Allocate space for an array of elements, initializes them to zero and then returns a pointer to the memory.
free( )	Frees previously allocated memory.
realloc( )	Modifies the size of previously allocated space.

malloc( )

1. The malloc( ) function allocates a block of memory in bytes. The user should explicitly give the block size it requires for the use. Here, numbers of contiguous memory locations are allocated.
2. malloc( ) function is like a request to the RAM of the system to allocate memory, if the request is granted, it returns a pointer to the first block of that memory.
3. The malloc( ) function is available in header file **alloc.h** or **stdlib.h** in Turbo C editor. <alloc.h> header file is used in UNIX operating system.
4. **Syntax:**
   malloc(number of elements * sizeof(data type)) ;
5. malloc( ) is allocated by a pointer variable. So, it must be defined before the allocation of memory by malloc( ) function.
   eg:
   ```
   int  *ptr;
   ptr = malloc( 10 * sizeof( int ) );
   ```
6. malloc( ) function returns a **void pointer**. So, a type cast operation is used to for memory allocation operation.
   i.e. ptr = (type_cast *)malloc(no of elements * sizeof(datatype));

   eg: int *ptr;
   ptr = (int *)malloc(10 * sizeof(int));
7. By default malloc( ) returns garbage value as output. i.e. if the values are not defined, it will return garbage value as output.

Program to find the sum and average of numbers of an array using dynamic memory allocation function.

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

void main( ){
    int i, n, *ptr, sum=0;
    float avg;
    printf("\nEnter the number of elements of the array");
    scanf("%d", &n);

    if(ptr==NULL){
      printf("\nNo elements are defined for an array");
      getch( );
      exit(0);                    // exit( ) undergoes <process.h> or <stdlib.h> header file
    }

    else{
      printf("Enter the elements of the array\n");
      for(i=0 ; i<n ; i++)
          scanf("%d", ( ptr + i ));
      for(i=0 ; i<n ; i++)
          sum = sum + (*( ptr + i ));
      printf("\nSUM = %d", sum);
      avg = float(sum)/n;
      printf("\nAVERAGE = %f", avg);
    }
    getch( );
}
```

calloc( )

1. It works exactly similar to malloc( ), except that it uses two arguments in function declaration.
2. **syntax:** **calloc(number of elements , sizeof (datatype));**

   E.g. int *ptr;

   ptr = (int *) calloc(10 , sizeof(int));
3. Header file for calloc( ) is **<stdlib.h>** or **<alloc.h>**. <alloc.h> header file is used for UNIX operating system.
4. By default calloc( ) function gives **zero** output i.e. if the values are not defined it will return zero as output.

realloc( )

This function is used to modify the memory block, which is already allocated. It is used in **two** situations:

1. If the allocated memory block is much more than what is required by the current application.
2. If the allocated memory block is insufficient for current application.

Syntax: **ptr_variable = realloc(ptr_var , new_size);**

Where, the ***ptr_var*** is previously allocated using malloc( ) or calloc ( ).

free( )

1. It is used to deallocate the previously allocated memory using malloc( ) or calloc( ) function.
2. It is generally used after the completion of all the operations on the pointer variable used for memory allocation by malloc( ) or calloc( ).
3. **syntax:** free(ptr_variable);

Where, ***ptr_variable*** is the variable in which the address of the allocated memory block is assigned. The free( ) function is used to release the allocated memory to the RAM.

OUTPUT QUESTIONS AND ANSWERS

Considering the header file is assigned for each of the program.

1.
```
void main( ){
        int a=2, *p;
        p =  &a;
        printf("Enter a number : ");
        scanf("%d", p);
        printf("a = %d\n *p = %d", a, *p);
}
```
Output: Considering the entered number as a = 5, the output will be:
```
a = 5
*p = 5
```
Explanation: Both *p* and *a* points to same address. So, if the value is updated from the address, then it
is assigned on each variable.

2.
```
void main( ){
        int i, a[ ]={10, 20, 30, 40, 50}, *p;
        p=&a;
        for(i=0; i<5; i++)
            printf("%d\t", *(p+i));
}
```
Output:
```
10    20    30    40    50
```
Explanation: The pointer variable is assigned by the address of array. So, the array elements can be accessed using the the index values.

3.
```
void main( ){
        int i, a[5]={10, 20, 30, 40, 50};
        int b[5];
        b=a;
        for(i=0; i<5; i++)
            printf("%d\t", b[i]);
}
```
Output: Error. Lvalue required.

Explanation: Array is a static concept. By assignment of base address, other memory location values will not be copied. Hence, the assignment operation statement returns error.

4.
```
void main( ){
       int i, a[5]={10, 20, 30, 40, 50}, *p;
       p=&a[4];
       for(i=0; i<5; i++)
           printf("\n%d\t%d", *(p-i), p[-i]);
}
```
Output:
```
50   50
40   40
30   30
20   20
10   10
```
Explanation: The last element of array address is assigned on the pointer variable. So, by decrementing the address variable, the array elements can be accessed in the reverse sequence.

5. void main( ){

```
    int i, a[5]={10, 20, 30, 40, 50};
    for(i=0; i<5; i++){
        printf("\n%d", *a);
        a++;
    }
}
```

Output: Error. Lvalue required.

Explanation: The array name itself represents the base address. Array represents a static concept. The address of array can't be incremented. Hence, it returns error.

6. void main( ){

```
     int i, a[5]={10, 20, 30, 40, 50}, *p;
     p=&a;
     for(i=0; i<5; i++){
         (*p)++;
         printf("\t%d", *p);
         p++;
    }
}
```

```
Output: 11   21    31    41    51
```

Explanation: The pointer variable is assigned by the base address. So, "(*p)++" will returns the incremented value of array elements. The pointer variable is incremented to locate to the next element of array.

7. void main( ){

```
    int i, a[5]={10, 20, 30, 40, 50}, *p;
    p= &a;
    for(i=0; i<5; i++)
        printf("\t%d", *p++);
    for(i=0; i<5; i++)
        printf("\t%d", *--p);
}
```

```
Output: 10    20    30    40    50    50    40    30    20    10
```

Explanation: Initially, the pointer variable is assigned by the address of array. "*p++" represents print the array element first and then increment the address. So, the first *for* loop will return the array elements. After the execution of first *for* loop, the pointer variable *p* is present at the end of array. So, the second for loop will is defined by decrementing the value of pointer variable to access the array elements in reverse sequence.

8. void main( ){

```
    int x, a[7]={10, 20, 30, 40, 50, 60, 70};
    x=(a+2)[3];
    printf("%d", x);
}
```

Output: 60

Explanation: The array itself represents the base address. "a+2" represents the the third location of array and "(a+2)[3]" represents the third location from the third location of array i.e. it represents the the fifth index location of array element. Hence, output is 60.

9.
```
void main( ){
     int a[7]={10, 20, 30, 40, 50, 60, 70};
     int *p, *q;
     q=a/2;
     p=q*2;
     printf("%d\t%d", *p, *q);
}
```
Output: Error. Illegal use of pointer.

Explanation: Pointer address variables does not allow arithmetic operation.

10.
```
void main( ){
     int a=5, *p, **q;
     p=&a;
     q=&p;
     printf("\na=%d\tp=%u\tq=%u", a, p, q);
     a++;
     p++;
     q++;
     printf("\na=%d\tp=%u\tq=%u", a, p, q);
}
```
Output:
```
a=5    p=1000   q=2000
a=6    p=1002   q=2002
```
Explanation: The program is based on on pointer to pointer concept. Increment of integer variable will increase the value by one, whereas, increment of address variable will increase by two for each each increment of address.

11.
```
void main( ){
     int a[5]={1, 2, 3, 4, 5};
     int *p = a;
     printf("\nSize of p=%d\tSize of a=%d", sizeof(p), sizeof(a));
}
```
Output: Size of p=2   Size of a=10

Explanation: Pinter variable contains the address of another variable. Since address is integer type, so size of pointer variable *p* is 2, whereas, *a* represents the base address of array. So, size of array is 10.

12.
```
int *p;
void main( ){
     func( );
     printf("%d", *p);
}
void func( ){
     int n=10;
     p=&n;
}
```
Output: 10

Explanation: The operation is based on pass by address concept. So, operation performed in function definition will reflect in main( ).

```
13. void main( ){
        int i, *p;
        int a[3][4]={{1, 2, 3, 4},{5, 6, 7, 8},{9, 10, 11, 12}};
        p=&a;
        for(i=0; i<12; i++)
           printf("%d  ", p[i]);
    }
```

Output: 1 2 3 4 5 6 7 8 9 10 11 12

Explanation: Though two dimensional array is represented in rectangular matrix form (in rows and columns), the elements are represented in row matrix form in contiguous memory locations sequentially.

PREVIOUS YEARS QUESTIONS AND ANSWERS

1. Determine the output when the following C program is executed. Ignore the typographic mistakes if any. [2004-BPUT]

```
#include <stdio.h>
int x=10, y=20;
void main( ){
    incr(x, &x) ;
    printf("x=%d, y=%d", x, y) ;
}
void incr(int a, int *p){
    int x=0 ;
    a=a+1 ;
    *p=*p+1 ;
    printf("a=%d, x=%d", a, x) ;
}
```

```
Ans: a=11, x=0
     x=11, y=20
```

Explanation : The function is called first. The reference variable *&x* will locate to the same meory location as that of *x*. Hence, operation performed on formal argument will refelect on actual argument. This represents pass by address feature of function. *a* value is incremented by *1*. So, output is *11* and *x=0* due to local initialization. The resultant updated value of *x=11* will transfer to the main( ). Hence, output is *x=11* and *y=20.*

2.a. What is the uses of malloc( ) and calloc( ) functions in C? In which header file these two functions are defined?
b. What is the full syntax of the above two functions?
c. Give an example (in C code) of any one of the above functions to allocate memory for a 2D integer matrix with m rows and n columns. [2004-BPUT]

Ans:a.) There are **two** ways of memory allocation in C programming.

1. Compile-time or Static allocation (Using array)
2. Run-time or Dynamic allocation (Using pointer)

When we declare an array in a program, the associated memory space is consumed till program ends regardless of being used or not. This creates a memory blockage. Such a situation can be avoided if we allocate necessary memory when needed and then free them when the use is over. We can do this at run time or execution time. *The process of allocating memory at run time is known as* ***dynamic memory allocation*** and *releasing the allocated memory at run time is known as* ***dynamic deallocation***. We can do this by using memory management functions. These functions are defined under **alloc.h** or **stdlib.h** header file. <alloc.h> header file is used in UNIX operating system.

b.) malloc( )

1. The malloc( ) function allocates a block of memory in bytes. The user should explicitly give the block size it requires for the use. Here, numbers of contiguous memory locations are allocated.
2. malloc( ) function is like a request to the RAM of the system to allocate memory, if the request is granted, it returns a pointer to the first block of that memory.
3. The malloc( ) function is available in header file **alloc.h** or **stdlib.h** in Turbo C editor. <alloc.h> header file is used in UNIX operating system.
4. **Syntax:**

   malloc(number of elements * sizeof(data type)) ;
5. malloc( ) is allocated by a pointer variable. So, it must be defined before the allocation of memory by malloc( ) function.

```
eg:     int  *ptr;
        ptr = malloc( 10 * sizeof( int ) );
```
6. malloc( ) function returns a **void pointer**. So, a type cast operation is used to for memory allocation operation.

   i.e.   ptr = (type_cast *)malloc(no of elements * sizeof(datatype));

```
eg:   int  *ptr;
      ptr = (int *)malloc( 10 * sizeof( int ) );
```
7. By default malloc( ) returns garbage value as output. i.e. if the values are not defined, it will return garbage value as output.

calloc( )

1. It works exactly similar to malloc( ), except that it uses two arguments in function declaration.
2. **syntax:** **calloc(number of elements , sizeof (datatype));**

```
E.g.  int *ptr;
      ptr = ( int * ) calloc( 10 , sizeof(int)  );
```
3. Header file for calloc( ) is **<stdlib.h>** or **<alloc.h>**. <alloc.h> header file is used for UNIX operating system.
4. By default calloc( ) function gives **zero** output i.e. if the values are not defined it will return zero as output.

c)
```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
void main( ){
    int i, j, m, n, *p;
    clrscr( );
    printf("\nEnter the size of matrix  : ");
    scanf("%d%d", &m, &n);
    p=(int )malloc(m*n*sizeof(int));
    if(p= =NULL)
        printf("\nNull memory space allocated");
    else{
        printf("\nEnter the elements of matrix : ");
        for(i=0 ; i<m ; i++)
```

```
            for(j=0 ; j<n ; j++)
              scanf("%d", p+i+j);
          printf("\nThe elements of matrix are :\n ");
          for(i=0;i<m;i++){
            for(j=0;j<n;j++){
              printf("%d\t", *(p+i*j+j));
            }
            printf("\n\n");
          }
      }
      free(p);
      getch( );
  }
```

3. What will be the output of the following program?

```
    void main( ){
         int a =25, b=10, *p, *q;
         p = &a;
         q = &b;
         clrsc( );
         printf("\n The sum a+b = %d and a-b = %d", *p+b, *p-b);
         printf("\n a*b = %d and a/b = %d", p*q, *p/ *q);
         printf("\n a mod b = %d", *p % *q);
    }
```

[2005-BPUT]

Ans:
```
The sum a+b = 35 and a-b = 15
    a*b = 350 and a/b = 2
    a mod b = 5
```

Note: Be careful of "*p/ *q". If there is no space between '/' and '*', then compiler will consider the rest block of code as comment line and returns syntax error.

4. What do you mean by pointer variable? Explain in details how pointer variables can be accessed using example. Write a C program to read a string from keyboard and display it using character pointer. [2005-BPUT]

Ans: ***A pointer is a memory variable that stores the address of another variable***. A pointer of one data type can store the address of same data type variable. i.e. an integer pointer can only store the address of integer variable. It can't store the address of any other data type variable like float, char, etc.

The **declaration syntax** of pointer is :

data_type * pointer_name ;

Where,

- The data type can be any type identifying the type of variable
- The asterisk (*) tells that the variable **pointer_name** is a pointer variable
- **pointer_name** is any valid identifier name defining the pointer variable of specified data type

Example: int *p;

The **syntax of pointer initialization** is:

pointer_name = & variable;

e.g. p=&a;

Here, pointer variable ‘p’ is assigned with the address of the variable a. As we already know ‘*’ operator gives the value stored at a particular address, “***p**” gives the value stored at address **p** i.e. **&a** . At the address **&a** the value of **a** has been stored. So, ***p** gives the value of a.

```
#include <stdio.h>
void main( ){
    char *p, str[10];
    printf(“\nEnter a string : ”);
    gets(str);
    p = str;
    printf(“\nThe string using character pointer : %s”, p);
}
```

5. Declare a variable x which takes integer values. Intializes it to 5, create a pointer (of integer type) which stores the address of x. [2005S-BPUT]

Ans:
```
        int x;
        x=5;      //initialization
        int *ptr;
        ptr=&x;          //pointer containing the address of x
        printf(“\nx=%d\t*ptr=%d”, x, *ptr) ;
```
Output :
```
        x=5     *ptr=5
```

6. Consider the following array of floats :

a[ ]={1.1, 2.2, 3.3, 4.4, 5.5}

Is ‘a’ pointer? If your answer is yes, then what does it dtore? What will be the output for each of the following print statements:

```
printf("%d\n", a);
printf("%d\n", *a);
printf("%d\n", (a+2));
printf("%d\n", *(a+2));
```
[2005S-BPUT]

Ans: Yes, *a* is a pointer. This is because, each array element can be represented using a pointer. The array name itself locates the base address. By specifying the index value, we can directly access the array elements. So, the output will be:

“***printf("%d\n", a);***” returns the base address (first memory location address). Since address is represented using “%u – unsigned int” access specifier, one can consider the output as a garbage value.

“***printf("%d\n", *a);***” returns the base address value.

“***printf("%d\n", (a+2));***” returns the third memory location address. Since address is represented using “%u – unsigned int” access specifier, one can consider the output as a garbage value.

“***printf("%d\n", *(a+2));***” returns the third memory location value.

Array index→	a[0]	a[1]	a[2]	a[3]	a[4]
Pointer representation →	*a	*(a+1)	*(a+2)	*(a+3)	*(a+4)

7.
```
#include <stdio.h>
void main( ){
    char *p = "hello!";
    p="Hi hello";
    *p='G';
    printf("%s",p);
}
```
Find the output/error if any. [2006-BPUT]

Ans: Gi hello

Explaination: The pointer variable is assigned with a string ("hello!") initially which is updated by another string ("Hi hello"). Then the base address is assigned with the character ('G') rest characters being unchanged. Hence the resultant output will be "*Gi hello*". The concepts can be explained using the following program:

```
#include <stdio.h>
void main( ){
    char *p = "hello!";
    printf("\n%s",p);
    p="Hi hello";
    printf("\n%s",p);
    *p='G';
    printf("\n%s",p);
}
```
```
Output: hello!
        Hi hello
        Gi hello
```

8. #include <stdio.h>
```
   void main( ){
        int a=5, *p=&a;
        printf("%d", ++(*p));
   }
```
Find the output/error if any. [2006-BPUT]

Ans: 6

Explaination: The pointer variable value is cremented by 1. The value of "*p" is 5 which is cremented by 1 which will return 6 as output.

9. #include <stdio.h>
```
   void main( ){
        int const *p=5;
        printf("%d", ++(*p));
   }
```
Find the output/error if any.
[2006-BPUT]

Ans: Error

Explaination: Constant variable can contain value only once. The value can't be modified in any way. So modification (increment) on the constant pointer will return error.

10. #include <stdio.h>
```
     void main( ){
         char s[ ];
         int i;
         for(i=0 ; s[i] ; i++)
             printf("\n%c  %c  %c  %c ", s[i], *(s+i), *(i+s), i[s] );
     }
```
Find the output/error if any. [2006-BPUT]

```
Ans:   m  m  m  m
       a   a   a   a
       n   n   n   n
```

Explaination : The different ways of accessing the individual characters of string are: s[i], *(s+i), *(i+s), i[s]. So, the string characters are printed 4 times until the end of string.

11. #include <stdio.h>
```
    void main( ){
        int c[ ]={2.8, 3.4, 4, 6.7, 5};
        int j, *p=c, *q=c;
        for(j=0 ; j<5 ; j++){
            printf("%d", *c);
            ++q;
        }
        for(j=0 ; j<5 ; j++){
            printf("%d ", *p);
            ++p;
        }
    }
```
Find the output/error if any. [2006-BPUT]

```
Ans : 2    2    2    2   2   2   3   4   6   5
```
Explaination : In the first for loop,
```
    for(j=0 ; j<5 ; j++){
        printf("%d", *c);
        ++q;
    }
```
The "*c" value is printed which points to the base address. When the q value is incremented "*c" points to the same base address. Hence, the for loop prints 2 five times.

Where as, for the second for loop,
```
    for(j=0 ; j<5 ; j++){
        printf("%d ", *p);
        ++p;
    }
```
Here, the pointer address is incremented and that value is printed. Hence, it prints the integer equivalent values of the float values. So output will be: 2   3   4   6   5

12. How can a function return a pointer to its calling routine? [2007-BPUT]

Ans: The operation can be defined using a pointer to function concept.

E.g. #include <stdio.h>
```
    int *sum(int, int);
    void main( ){
        int x, y, *p;
        printf("Enter the value of x and y : ") ;
        scanf("%d%d", &x, &y);
        p=sum(x,y);
        printf("Sum output = %d", *p) ;
    }
    int *sum(int a, int b){
        int z;
        z=x+y;
        return(&z) ; //The function retruns a pointer to its calling routine
    }
```

13. Write the difference between const char *p, char const *p, const char *const p. [2007S-BPUT]
Ans: The keyword *const* can be used for pointers before the type, after the type or in both places. All the declaration has its identical meaning.
a. const char *a;
Here, *a* is a pointer to a constant character. The value that is pointed to can't be changed.
b. char *const b;
Here, *b* is a constant pointer to an character. The character can be changed, but *b* can't point to anything else.
c. const int * const c;
Here, *c* is a constant pointer to a constant character. The value that is pointed to can't be changed, and *c* can't be changed to point to anything else.

14. What is the difference between malloc( ) and calloc( )? [2007S-BPUT]
Ans: a. malloc( ) uses one argument, whereas calloc( ) uses two arguments.
a. malloc( ) by default returns garbage value whereas calloc( )by default returns zero as output.
b. Both are dynamic memory allocation functions and are defined under <stdlib.h> header file.

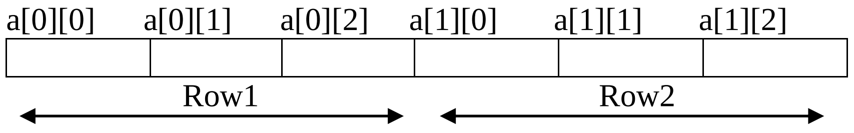
15. How much memory does a pointer variable allocates? [2007S-BPUT]
Ans: Since pointer is a memory variable used for *storing the address* of another variable of same type as that of itself, so memory allocated by a pointer variable will be 2-bytes. This is because address is integer type and size fo integer variable is 2-bytes.

16. What is the real difference between array and pointer? [2007S-BPUT]
Ans: Array and pointers are equal in power, but the difference is that:
   a. Array is a static implementation where as, pointer is a dynamic implementation.
   b. Most of the time array provides waistage or insufficiency of memory space where as pointer uses the allocated memory space completely.
   c. Memory allocated by pointer can be released after implementation using dynamic memory deallocation function (free( )), where as array doesn't allow this property.

17. How is a multidimensional array defined in terms of a pointer to a collection of contiguous array of lower dimensionality? Explain with an example. [2007S-BPUT]
Ans: Array is the homogenous collection of elements stored in contiguous memory locations. Irrespective of dimension of array, the elements are store in contiguous memory locations.
E.g. "int a[2][3];" represents an array defined in two rows and three columns.

a[0][0]	a[0][1]	a[0][2]
a[1][0]	a[1][1]	a[1][2]

So, the memory can be represented in form of two one dimensional array of size three. They are stored in contiguous locations i.e. the first row elements will be stored first and the second row elements will be stores at the end of first row elements.



The multi-dimensional array elements can be accessed using a pointer to pointer concept. Addresses of different memory locations can be generated by adding the subscript values with the base address. A 3D array *a[2][3][4]* represents two number of *3X4* array. A 3D array concept is represented in the following example given below.

```
#include <stdio.h>
#include <conio.h>
void main( ){
    int a[2][2][2]={ { {1,1},
                        {2,2} },
                      { {3,3},
                        {4,4} }
                   };
    int i, j, k;
    clrscr( );
    printf("\nThe 3D array using pointer is\n");
    for(i=0;i<2;i++){
        for(j=0;j<2;j++){
            for(k=0;k<2;k++){
                printf("%d\t", (*(*(*a+i)+j)+k));
            }
            printf("\n");
        }
        printf("\n");
    }
    getch( );
}
```

18. *p++ increments p, or what it points to? [2007S-BPUT]
Ans: *p++ denotes the value at the current memory location. After locating the current memory location, the pointer will increment and will locate to the next contiguious memory location.

19. What is "near" and "far" pointers? [2007S-BPUT]
Ans: Each memory address consists of two parts:

→ Segmnt address
→ Offset address

Far and near pointers are memory management pointers. A "far" pointer allocates 4 bytes for operation where as "near" pointer allocates 2 bytes of memory space for operation. Irrespective of data type the size of near and far pointer remains constant due to memory allocation function.

KEYWORD	DATA	CODE	POINTER ARITHMETIC
Near	16 bit address	16 bit address	16 bit
Far	32 bit address	32 bit address	16 bit

20. Define pointer in C language. How the declarations are made for pointer variables? What is the difference between the function pointer and pointer to a function? What is a far pointer? [2008-BPUT]
Ans: ***A pointer is a memory variable that contains the address of another variable***. A pointer of one data type can store the address of same data type variable. i.e. an integer

pointer can only store the address of integer variable. It can't store the address of any other data type variable like float, char, etc.

The **declaration syntax** of pointer is :

data_type * pointer_name ;

Where,

- The data type can be any type identifying the type of variable
- The asterisk (*) tells that the variable **pointer_name** is a pointer variable
- **pointer_name** is any valid identifier name defining the pointer variable of specified data type

Example: int *p;

Function pointer refers to the function returning pointer. Where as, pointer to function is a function which points to another function of same return type as that of itself. It is basically used for storing the address of another function for copy of function content operations.

Far pointer: Each memory address consists of two parts:

→ Segmnt address

→ Offset address

Far pointers are memory management pointers. A "far" pointer allocates 4 bytes for operation. It allocates 32-bit data address space, 32 bit code address space and 16-bit pointer arithmetic operations. Irrespective of data type, the size of far pointer remains constant due to memory allocation function.

21. int arr[3][4]={1,2,3,4,5,6,7,8,9,10,11,12};
Assume "arr" as an integer pointer. Write a C statement to print 10 from the list using "arr" as pointer. [2009-BPUT]
Ans: printf("\n%d",*(*(a+2)+1));

21. Write a program to find the addition, subtraction, multiplication, division value of any two real numbers. The above operations should be carried out with the help of individual user defined functions. User will be asked to enter a choice and based upon that choice the results will be shown by calling only one function at a time. No control statements such as "if…else", "while", "do…while", "switch…case" can be used. Use of ternary operator is also not allowed. [2009-BPUT]

Program:

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

float add(float,int);
float sub(float,int);
float mul(float,int);
float div(float,int);

void main( ){
    float a,r;
    int b,choice;
    clrscr();
    float (*fp[ ])(float,int)={add,sub,mul,div};
    printf("\nPress 1. Addition\n2. Subtraction\n3. Multiplication\n4. Division");
    printf("\nEnter your choice : ");
    scanf("%d", &choice);
    printf("\nEnter the value of a and b : ");
    scanf("%f%d", &a, &b);
    r=(*fp[choice-1])(a,b);
    printf("\nResult : %2.2f", r);
    getch( );
}
float add(float a, int b){
    return a+b;
}
float sub(float a, int b){
    return a-b;
}
float mul(float a, int b){
    return a*b;
}
float div(float a, int b){
    return (float)a/b;
}
```

22. Give the prototype declarations of "malloc", "calloc", "realloc" functions that are used for dynamic memory allocation in C. What is the difference between "malloc" and "calloc" function? What is the importance of void pointer in connection with dynamic memory allocation? [2009-BPUT]

Ans:

malloc( )

1. The malloc( ) function allocates a block of memory in bytes. The user should explicitly give the block size it requires for the use. Here, numbers of contiguous memory locations are allocated.
2. malloc( ) function is like a request to the RAM of the system to allocate memory, if the request is granted, it returns a pointer to the first block of that memory.
3. The malloc( ) function is available in header file **alloc.h** or **stdlib.h** in Turbo C editor. <alloc.h> header file is used in UNIX operating system.
4. **Syntax:**

   ```
   malloc( number of elements * sizeof(data type)) ;
   ```
5. malloc( ) is allocated by a pointer variable. So, it must be defined before the allocation of memory by malloc( ) function.

   ```
   eg:     int  *ptr;
           ptr = malloc( 10 * sizeof( int ) );
   ```
6. malloc( ) function returns a **void pointer**. So, a type cast operation is used to for memory allocation operation.

   ```
   i.e.    ptr = ( type_cast * )malloc( no of elements * sizeof(datatype) );

   eg:     int  *ptr;
           ptr = (int *)malloc( 10 * sizeof( int ) );
   ```
7. By default malloc( ) returns garbage value as output. i.e. if the values are not defined, it will return garbage value as output.

Program to find the sum and average of numbers of an array using dynamic memory allocation function.

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main( ){
    int i, n, *ptr, sum=0;
    float avg;
    printf("\nEnter the number of elements of the array");
    scanf("%d", &n);
    if(ptr==NULL){
      printf("\nNo elements are defined for an array");
      getch( );
      exit(0);                    // exit( ) undergoes <process.h> or <stdlib.h> header file
    }
    else{
      printf("Enter the elements of the array\n");
      for(i=0 ; i<n ; i++)
          scanf("%d", ( ptr + i ));
      for(i=0 ; i<n ; i++)
          sum = sum + (*( ptr + i ));
      printf("\nSUM = %d", sum);
      avg = float(sum)/n;
      printf("\nAVERAGE = %f", avg);
    }
    getch( );
}
```

calloc( )

1. It works exactly similar to malloc( ), except that it uses two arguments in function declaration.
2. **syntax: calloc(number of elements , sizeof (datatype));**

   E.g. int *ptr;

   ```
   ptr = ( int * ) calloc( 10 , sizeof(int)  );
   ```
3. Header file for calloc( ) is **<stdlib.h>** or **<alloc.h>**. <alloc.h> header file is used for UNIX operating system.
4. By default calloc( ) function gives **zero** output i.e. if the values are not defined it will return zero as output.

realloc( )

This function is used to modify the memory block, which is already allocated. It is used in **two** situations:

1. If the allocated memory block is much more than what is required by the current application.
2. If the allocated memory block is insufficient for current application.

Syntax: ptr_variable = realloc(ptr_var , new_size);

Where, the ***ptr_var*** is previously allocated using malloc( ) or calloc ( ).

23. Write a complete C program to create an integer array dynamically. The array dimension is not given during runtime. Create the array and initialize the elements and also display the elements. [2009-BPUT]

Ans:

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
void main( ){
    int i, n, *ptr;
    printf("\nEnter the number of elements of the array : ");
    scanf("%d", &n);
    if(ptr==NULL){
      printf("\nNo elements are defined for an array");
      getch( );
      exit(0);                    // exit( ) undergoes <process.h> or <stdlib.h> header file
    }

    else{
      printf("Enter the elements of the array\n");
      for(i=0 ; i<n ; i++)
          scanf("%d", ( ptr + i ));
     printf("\nThe array elements are : ");
     for(i=0 ; i<n ; i++)
          printf("%d\t", *( ptr + i ));
    }
    getch( );
}
```

HOME ASSIGNMENT

1. What is the function of a pointer variable? What are its uses?

2. What are address operator and dereferencing operator symbols in C? Why do they do?

3. Write the difference between the address of operator ‘&’ and bitwise operator ‘&’ using suitable example.

4. What do you mean by scope of pointer variable?

5. Write the declaration and initialization procedure of pointer variable?

6. What is dereferencing? How is a pointer variable dereferenced?

7. Do all pointer variables have same size? Justify your answer using suitable example.

8. Write a program to accept a string and print the address of string.

9. Write the difference between pass by value and pass by reference. Perform swapping of two numbers program to illustrate such concept.

10. How reference and dereference operator is related. Illustrate such concept using suitable example.

11. What is void pointer? Write the need of void pointer.

12. What do you mean by pointer to pointer concept? Illustrate the concept using suitable example.

13. Write a program using C to access the elements of two dimensional array using a pointer to pointer concept.

14. Write a program using C to find the greatest among array elements using pass by reference.

15. Explain the difference between array and pointer.

16. Distinguish between the following declarations: “const int *p;” and “int *const p;”. Illustrate the difference using suitable example. Which one of them is a constant pointer and which one is a pointer to constant.

17. What is the difference between the following declations:

- void fun(int a[ ]);
- void fun(int *a);

18. Write a program using C to perform string copy operation.