

## Chapter -8

### POINTERS

#### Pointers :

- Pointer is a variable that holds a memory address of another variable.
- It supports dynamic allocation routines.
- It can improve the efficiency of certain routines.

#### C++ Memory Map :

- Program Code : It holds the compiled code of the program.
- Global Variables : They remain in the memory as long as program continues.
- Stack : It is used for holding return addresses at function calls, arguments passed to the functions, local variables for functions. It also stores the current state of the CPU.
- Heap : It is a region of free memory from which chunks of memory are allocated via DMA functions.

**Static Memory Allocation :** The amount of memory to be allocated is known in advance and it allocated during compilation, it is referred to as Static Memory Allocation.

Eg. `Int a;` // This will allocate 2 bytes for a during compilation.

**Dynamic Memory Allocation :** The amount of memory to be allocated is not known beforehand rather it is required to be allocated as and when required during runtime, it is referred to as dynamic memory allocation.

C++ offers two operators for DMA – new and delete.

**Free Store :** It is a pool of unallocated heap memory given to a program that is used by the program for dynamic memory allocation during execution.

### Declaration and Initialization of Pointers :

Datatype    *variable_name;
-----------------------------

Syntax : Datatype \*variable\_name;

Int *p;            float *p1;    char *c;
---

Eg. Int \*p;            float \*p1;    char \*c;

Two special unary operator \* and & are used with pointers. The & is a unary operator that returns the memory address of its operand.

Eg. Int a = 10; int \*p; p = &a;

### Pointer arithmetic:

Two arithmetic operations, addition and subtraction, may be performed on pointers.

When you add 1 to a pointer, you are actually adding the size of whatever the pointer is pointing at. That is, each time a pointer is incremented by 1, it points to the memory location of the next element of its base type.

Eg. Int \*p;            P++;

If current address of p is 1000, then p++ statement will increase p to 1002, not 1001.

If \*c is char pointer and \*p is integer pointer then

Char pointer	C	c+1	c+2	c+3	c+4	c+5	c+6	c+7
<b>Address</b>	<b>100</b>	<b>101</b>	<b>102</b>	<b>103</b>	<b>104</b>	<b>105</b>	<b>106</b>	<b>107</b>
Int pointer	p		p+1		p+2		p+3	

Adding 1 to a pointer actually adds the size of pointer's base type.

**Base address :** A pointer holds the address of the very first byte of the memory location where it is pointing to. The address of the first byte is known as BASE ADDRESS.

### Dynamic Allocation Operators :

C++ dynamic allocation routines obtain memory for allocation from the free store, the pool of unallocated heap memory provided to the program. C++ defines two unary operators new and delete that perform the task of allocating and freeing memory during runtime. The operators new and delete are also called as free store operators.

#### Creating Dynamic Array :

Syntax : pointer-variable = new data-type [size];

Eg. int \* array = new int[10];

Not array[0] will refer to the first element of array, array[1] will refer to the second element. No initializes can be specified for arrays.

All array sizes must be supplied when new is used for array creation.

#### Two dimension array :

Int \*arr, r, c;

R = 5; c = 5;

Arr = new int [r \* c];

Now to read the element of array, you can use the following loops :

For (int i = 0; i < r; i++)

{

    cout << "\n Enter element in row " << i + 1 << " : ";

    For (int j=0; j < c; j++)

        Cin >> arr [ i \* c + j];

}

**Memory released with delete as below :**

Syntax for simple variable :	For array :
Delete pointer-variable;	delete [size] pointer variable;
Eg. delete p;	Eg. delete [ ] arr;

### **Pointers and Arrays :**

C++ treats the name of an array as if it were a pointer i.e. memory address of some element. C++ interprets an array name as the address of its first element.

That is, if marks is an int array to hold 10 integers then marks stores the address of marks[0], the first element of the array i.e., the array name marks is a pointer to an integer which is the first element of array marks[10].

void main()

{

    int \*m;

    int marks[10];

    cout << "\n Enter marks :";

    for (int i = 0; i < 10; i++)

        cin >> marks[i];

    m = marks;

    cout << "\n m points to " << \*m;

    cout << "\n Marks points to " << \*marks;

}

The name of an array is actually a pointer pointing to the first element of the array.

Since the name of an array is a pointer to its first element, the array+1 gives the address of the second element, array+2 gives the address of the third element, and so on.

Thus, to print the fourth element of array marks, we can give either of the following :

Cout << marks [3];

OR

cout << \* (marks+3);

### **Array of Pointers :**

To declare an array holding 10 int pointers –

```
int * ip[10];
```

That would be allocated for 10 pointers that can point to integers.

Now each of the pointers, the elements of pointer array, may be initialized. To assign the address of an integer variable phy to the forth element of the pointer array, we have to write

```
ip[3] = & phy;
```

Now with \*ip[3], we can find the value of phy.

```
int *ip[5];
```

Index	0	1	2	3	4
address	1000	1002	1004	1006	1008

```
int a = 12, b = 23, c = 34, d = 45, e = 56;
```

Variable	a	b	c	d	e
Value	12	23	34	45	56
address	1050	1065	2001	2450	2725

```
ip[0] = &a; ip[1] = &b; ip[2] = &c; ip[3] = &d; ip[4] = &e;
```

Index	ip[0]	ip[1]	ip[2]	ip[3]	ip[4]
Array ip value	1050	1065	2001	2450	2725
address	1000	1002	1004	1006	1008

ip is now a pointer pointing to its first element of ip. Thus

ip is equal to address of ip[0], i.e. 1000

\*ip (the value of ip[0]) = 1050

\* (\* ip) = the value of \*ip = 12

\* \* (ip+3) = \* \* (1006) = \* (2450) = 45

Now see the program given below :

```
#include<iostream.h>
```

```
Void main()
```

```
{
```

```
int x[3][5] = { { 1,2,3,4,5}, { 6,7,8,9,10}, { 11,12,13,14,15} };
```

```
int *n = &x[0][0];
```

```
cout << "\n 1. * ( *(x+2)+1)          =      " << " << (*(x+2)+1);
```

```
cout << "\n 2. *(x+2)+5                =      " << " << (*(x+2)+5;
```

```
cout << "\n 3. (*(x+1))                 =      " << " << (*(x+1));
```

```
cout << "\n 4. *((x)+2)+1      =      " << *((x)+2)+1;
cout << "\n 5. *((x+1)+3)     =      " << *((x+1)+3);
cout << "\n 6. *n             =      " << *n;
cout << "\n 7. *(n+2)         =      " << *(n+2);
cout << "\n 8. *(n+3)+1       =      " << *(n+3)+1;
cout << "\n 9. *(n+5)+1       =      " << *(n+5)+1;
cout << "\n 10. ++*n          =      " << ++*n;
}
```

Output with Explanation :

1. * ( *(x+2)+1)	=	12	--- *((x+2)+1) = *(x[2]+1) = x[2][1] = 12
2. *((x+2)+5)	=	8	--- *((x+2)+5) = *(x[0]+2)+5 = x[0][2] + 5 = 3+5 = 8
3. *((x+1))	=	6	--- *((x+1)) = *(x[1]) = *(x[1]+0) = x[1][0] = 6
4. *((x)+2)+1	=	4	--- *((x)+2)+1 = *(x[0]+2)+1 = (x[0][2])+1 = 3+1 = 4
5. *((x+1)+3)	=	9	--- *((x+1)+3) = *(x[1] +3) = x[1][3] = 9
6. *n	=	1	--- *n = x[0][0] = 1 (first element)
7. *(n+2)	=	3	--- *(n+2) = x[0][2] = 3 (third element)
8. *(n+3)+1	=	5	--- *(n+3)+1 = x[0][3] (fourth element) + 1 = 4 + 1 = 5
9. *(n+5)+1	=	7	--- *(n+5)+1 = x[1][0] (sixth element) + 1 = 6+1 = 7
10. ++*n	=	2	--- ++ *n = ++ 1 = 2

## Pointers and Strings :

Pointer is very useful to handle the character array also.

Eg :

```
Char name[] = "computer";
```

```
Char *cp;
```

```
For (cp = name; *cp != '\0'; cp++)
```

```
    Cout << "--"<<*cp;
```

Output :

```
--c--o--m--p--u--t--e--r
```

An array of char pointers is very useful for storing strings in memory.

```
Char *subject[] = { "Chemistry", "Phycics", "Maths", "CS", "English" };
```

In the above given declaration subject[] is an array of char pointers whose element pointers contain base addresses of respective names. That is, the element pointer subject[0] stores the base address of string "Chemistry", the element pointer subject[1] stores the above address of string "Physics" and so forth.

An array of pointers makes more efficient use of available memory by consuming lesser number of bytes to store the string.

An array of pointers makes the manipulation of the strings much easier. One can easily exchange the positions of strings in the array using pointers without actually touching their memory locations.

## Pointers and CONST :

A constant pointer means that the pointer in consideration will always point to the same address. Its address can not be modified.

A pointer to a constant refers to a pointer which is pointing to a symbolic constant.

Look the following example :

```

Int m = 20;           // integer m declaration
Int *p = &m;          // pointer p to an integer m
++ (*p);              // ok : increments int pointer p
Int * const c = &n;    // a const pointer c to an integer n
++ (* c);              // ok : increments int pointer c i.e. its contents
++ c;                 // wrong : pointer c is const – address can't be modified
Const int cn = 10;     // a const integer cn
Const int *pc = &cn;   // a pointer to a const int
++ (* pc);             // wrong : int * pc is const – contents can't be modified
++ pc;                 // ok : increments pointer pc
Const int * const cc = *k; // a const pointer to a const integer
++ (* cc);             // wrong : int *cc is const
++ cc;                 // wrong : pointer cc is const

```

### Pointers and Functions :

A function may be invoked in one of two ways :

1. call by value
2. call by reference

The second method call by reference can be used in two ways :

1. by passing the references
2. by passing the pointers

Reference is an alias name for a variable. For ex :

```

Int m = 23;
Int &n = m;
Int *p;
P = &m;

```

Then the value of m i.e. 23 is printed in the following ways :

```

Cout << m;    // using variable name
Cout << n;    // using reference name
Cout << *p;   // using the pointer

```

### Invoking Function by Passing the References :

When parameters are passed to the functions by reference, then the formal parameters become references (or aliases) to the actual parameters to the calling function.

That means the called function does not create its own copy of original values, rather, it refers to the original values by different names i.e. their references.

For example the program of swapping two variables with reference method :

```

#include<iostream.h>
Void main()
{

```

```

void swap(int &, int &);
int a = 5, b = 6;
cout << "\n Value of a :" << a << " and b :" << b;
swap(a, b);
cout << "\n After swapping value of a :" << a << "and b :" << b;
}
Void swap(int &m, int &n)
{
    int temp;
    temp = m;
    m = n;
    n = temp;
}

```

output :

Value of a : 5 and b : 6

After swapping value of a : 6 and b : 5

### Invoking Function by Passing the Pointers :

When the pointers are passed to the function, the addresses of actual arguments in the calling function are copied into formal arguments of the called function.

That means using the formal arguments (the addresses of original values) in the called function, we can make changing the actual arguments of the calling function.

For example the program of swapping two variables with Pointers :

```
#include<iostream.h>
```

```
void main()
```

```

{
    void swap(int *m, int *n);
    int a = 5, b = 6;
    cout << "\n Value of a :" << a << " and b :" << b;
    swap(&a, &b);
    cout << "\n After swapping value of a :" << a << "and b :" << b;
}

```

```
void swap(int *m, int *n)
```

```

{
    int temp;
    temp = *m;
    *m = *n;
    *n = temp;
}

```

iutput :

Value of a : 5 and b : 6

After swapping value of a : 6 and b : 5

### Function returning Pointers :

The way a function can returns an int, an float, it also returns a pointer. The general form of prototype of a function returning a pointer would be

Type \* function-name (argument list);

```
#include <iostream.h>
```

```
int *min(int &, int &);
```

```

void main()
{
    int a, b, *c;
    cout << "\nEnter a :";      cin >> a;
    cout << "\nEnter b :";      cint >> b;
    c = min(a, b);
    cout << "\n The minimum no is :" << *c;
}

```

```

int *min(int &x, int &y)
{
    if (x < y )
        return (&x);
    else
        return (&y);
}

```

### **Dynamic structures :**

The new operator can be used to create dynamic structures also i.e. the structures for which the memory is dynamically allocated.

```
struct-pointer = new struct-type;
```

```

student *stu;
stu = new Student;

```

A dynamic structure can be released using the deallocation operator delete as shown below :

```
delete stu;
```

### **Objects as Function arguments :**

Objects are passed to functions in the same way as any other type of variable is passed.

When it is said that objects are passed through the call-by-value, it means that the called function creates a copy of the passed object.

A called function receiving an object as a parameter creates the copy of the object without invoking the constructor. However, when the function terminates, it destroys this copy of the object by invoking its destructor function.

If you want the called function to work with the original object so that there is no need to create and destroy the copy of it, you may pass the reference of the object. Then the called function refers to the original object using its reference or alias.

Also the object pointers are declared by placing in front of a object pointer's name.

```
Class-name * object-pointer;
```

Eg. Student \*stu;

The member of a class is accessed by the arrow operator (->) in object pointer method.

Eg :

```

#include<iostream.h>
class Point
{
private :
    int x, y
public :
    Point()
    {

```

```

        x = y = 0;
    }
    void getPoint(int x1, int y1)
    {
        x = x1; y = y1;
    }
    void putPoint()
    {
        cout << "\n Point : (" << x << ", " << y << ")";
    }
};

void main()
{
    Point p1, *p2;
    cout << "\n Set point at 3, 5 with object";
    p1.getPoint(3,5);
    cout << "\n The point is :";
    p1.putPoint();
    p2 = &p1;
    cout << "\n Print point using object pointer :";
    p2->putPoint();
    cout << "\n Set point at 6,7 with object pointer";
    p2->getPoint(6,7);
    cout << "\n The point is :";
    p2->putPoint();
    cout << "\n Print point using object :";
    p1.getPoint();
}

```

If you make an object pointer point to the first object in an array of objects, incrementing the pointer would make it point to the next object in sequence.

```
student stud[5], *sp;
```

```
---
```

```
sp = stud;           // sp points to the first element (stud[0]) of stud
```

```
sp++;                // sp points to the second element (stud[1]) of stud
```

```
sp += 2;             // sp points to the fourth element (stud[3]) of stud
```

```
sp--;                // sp points to the third element (stud[2]) of stud
```

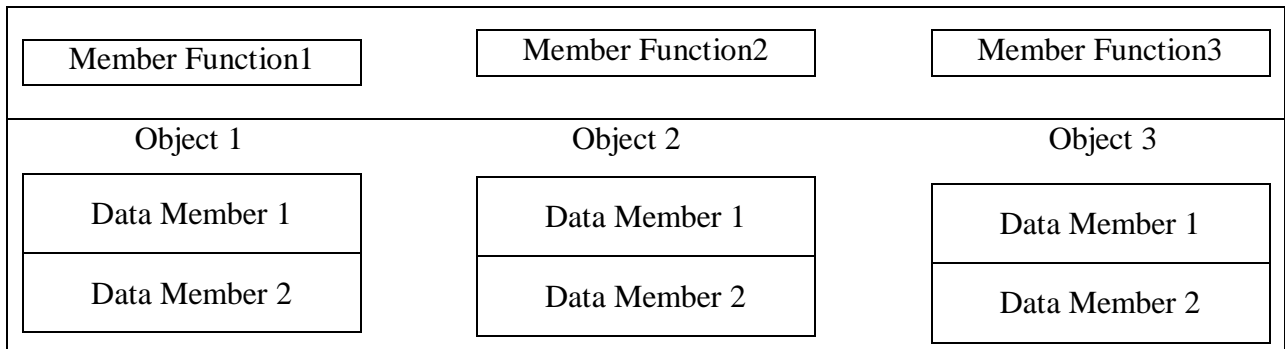
You can even make a pointer point to a data member of an object. Two points should be considered :

1. A Pointer can point to only public members of a class.
2. The data type of the pointer must be the same as that of the data member it points to.

### **this Pointer :**

In class, the member functions are created and placed in the memory space only once. That is only one copy of functions is used by all objects of the class.

Therefore if only one instance of a member function exists, how does it come to know which object's data member is to be manipulated ?



For the above figure, if Member Function2 is capable of changing the value of Data Member3 and we want to change the value of Data Member3 of Object3. How would the Member Function2 come to know which Object's Data Member3 is to be changed ?

To overcome this problem this pointer is used.

When a member function is called, it is automatically passed an implicit argument that is a pointer to the object that invoked the function. This pointer is called This.

That is if object3 is invoking member function2, then an implicit argument is passed to member function2 that points to object3 i.e. this pointer now points to object3.

The friend functions are not members of a class and, therefore, are not passed a this pointer.

The static member functions do not have a this pointer.

### Summary :

Pointers provide a powerful way to access data by indirection. Every variable has an address, which can be obtained using the address of operator (&). The address can be stored in a pointer.

Pointers are declared by writing the type of object that they point to, followed by the indirection operator (\*) and the name of the pointer. Pointers should be initialized to point to an object or to null (0).

You access the value at the address stored in a pointer by using the indirection operator (\*). You can declare const pointers, which can't be reassigned to point to other objects, and pointers to const objects, which can't be used to change the objects to which they point.

To create new objects on the free store, you use the new keyword and assign the address that is returned to a pointer. You free that memory by calling the delete keyword on the pointer. delete frees the memory, but it doesn't destroy the pointer. Therefore, you must reassign the pointer after its memory has been freed.

### Solved Questions

#### Q. 1 How is \*p different from \*\*p ?

Ans : \*p means, it is a pointer pointing to a memory location storing a value in it. But \*\*p means, it is a pointer pointing to another pointer which in turn points to a memory location storing a value in it.

#### Q. 2 How is &p different from \*p ?

Ans : &p gives us the address of variable p and \*p. dereferences p and gives us the value stored in memory location pointed to by p.

#### Q. 3 Find the error in following code segment :

```
Float **p1, p2;
P2 = &p1;
```

Ans : In code segment, p1 is pointer to pointer, it means it can store the address of another pointer variable, whereas p2 is a simple pointer that can store the address of a normal variable. So here the statement `p2 = &p1` has error.

**Q. 4 What will be the output of the following code segment ?**

```
char C1 = 'A';
char C2 = 'D';
char *i, *j;
i = &C1;
j = &C2;
*i = j;
cout << C1;
```

Ans : It will print A.

**Q. 5 How does C++ organize memory when a program is run ?**

Ans : Once a program is compiled, C++ creates four logically distinct regions of memory :

- (i) area to hold the compiled program code
- (ii) area to hold global variables
- (iii) the stack area to hold the return addresses of function calls, arguments passed to the functions, local variables for functions, and the current state of the CPU.
- (iv) The heap area from which the memory is dynamically allocated to the program.

**Q. 6 Identify and explain the error(s) in the following code segment :**

```
float a[] = { 11.02, 12.13, 19.11, 17.41 };
float *j, *k;
j = a;
k = a + 4;
j = j * 2;
k = k / 2;
cout << " *j = " << *j << ", *k = " << *k << "\n";
```

Ans : The erroneous statements in the code are :

```
j = j * 2;
k = k / 2;
```

Because multiplication and division operations cannot be performed on pointer and j and k are pointers.

**Q. 13 How does the functioning of a function differ when**

- (i) an object is passed by value ?
- (ii) an object is passed by reference ?

Ans : (i) When an object is passed by value, the called function creates its own copy of the object by just copying the contents of the passed object. It invokes the object's copy constructor to create its copy of the object. However, the called function destroys its copy of the object by calling the destructor function of the object upon its termination.

- (i) When an object is passed by reference, the called function does not create its own copy of the passed object. Rather it refers to the original object using its reference or alias name. Therefore, neither constructor nor destructor function of the object is invoked in such a case.

### UNSOLVED QUESTIONS

1. Differentiate between static and dynamic allocation of memory.
2. Identify and explain the error in the following program :

```
#include<iostream.h>
int main()
{
```

```

int x[] = { 1, 2, 3, 4, 5 };
for (int i = 0; i < 5; i++)
{
    cout << *x;
    x++;
}
return 0;
}

```

3. Give the output of the following :

```

char *s = "computer";
for (int x = strlen(s) - 1; x >= 0; x--)
{
    for(int y = 0; y <= x; y++)    cout << s[y];
    cout << endl;
}

```

4. Identify the syntax error(s), if any, in the following program. Also give reason for errors.

```

void main()
{
    const int i = 20;
    const int * const ptr = &i;
    (*ptr++);
    int j = 15;
    ptr = &j; }

```

5. What is 'this' pointer ? What is its significance ?

6. Are pointers really faster than array ? How much do function calls slow things down ? Is ++i faster than i = i + 1 ?

7. What will be the output of following program ?

```

#include<iostream.h>
void main()
{
    char name1[] = "ankur";
    char name2[] = "ankur";
    if (name1 != name2)
        cout << "\n both the strings are not equal";
    else
        cout << "\n the strings are equal"; }

```

8. Write a function that takes two string arguments and returns a string which is the larger of the two. The larger string has larger ASCII value. Also show how this function will be invoked.

9. Give and explain the output of the following code :

```

void junk (int, int *);
int main()
{
    int i = 6, j = -4;
    junk (i, &j);
    cout << "i = " << i << ", j = " << j << "\n";
    return 0;
}

void junk(int a, int *b)
{
    a = a * a;
    *b = *b * *b;
}

```

10. Give the output of the following program :

```
void main()
{
    int array[] = { 2, 3, 4, 5 };
    int *ap = array;
    int value = *ap;
    cout << value << "\n";
    value = *ap++;
    cout << value << "\n";
    value = * ap;
    cout << value << "\n";
    value = * ++ ap;
    cout << value << "\n";
}
```

### **High Order Thinking Skills (HOTS)**

- Q1. What is wrong with the following while loops ( ans how does the correct ones look like):**

(i) 

```
int counter =1;
While (counter<100)
{
    cout<<counter<<"\n";
    counter--;
}
```

(ii) 

```
int counter =1;
while (counter <100)
cout<<counter<< "\n";
counter + +;
```

**Ans** (i) In this loop the counter is decremented, so it will have values 1,0,-1,-2,-3..... so this loop is an infinite loop. To fix, we need to use counter ++ instead of counter - - to fix

```
while (counter<100)
{
    .....
    counter + +;
}
```

(ii) In this loop there are not brackets surrounding the code block of the while loop. Therefore, only the line immediately following the while statement repeats. To fix, we need to add grouping bracket around the indented lines after the while statement i.e. as :

```
while (counter<10)
{
    cout<<counter<<"\n";
    counter++;
}
```

- Q2. Write a c++ function that converts a 2-digit octal number into binary number and prints the binary equivalent.**

**Ans** Assume that a header file and main() is including in a program

**The function is as follows:**

```
Void octobin(int oct)
{
    long binn=0;
```

```

int a[6];                                /*Each octal digit is converted into 3 bits thus for 2 octal digits
                                         -- space for 6 bits has been reserved here*/

int d1,d2,q,r;;
d1=oct%10;
d2= oct/10;
    for int (i=0; i<6; i++)
    {
        a[i] =0;
    }
for (i=0; i<3; i++)
{
    q=d1/2;
    r=d1%2;
    a[i]=r;
    d1=q;
}

for ( ; i<6;i++)
{
    q=d2/2;
    r=d2%2;
    a[i]=r;
    d2=q;
}
for (i=i-1; i>=0;i - -)
{
    binn * =10;
    binn += a[i];
}
cout<<endl<<binn<<endl;
}

```

### Q3. How we can use arrays as arguments? Explain with example

**Ans** Array can be used as other data types, as arguments to functions. Here is an example of it

#### // Array as arguments

```

#include <iostream.h>
const int district=4;
const int months=6;
void display (int [districts][months]);
void main()
{
    int d,m;
    int sales [districts] [months];
    cout<<endl;
    for (d=0;d<districts; d++)
    for (m=0;m<months; m++)
    {
        cout<<"enter sales:"<d++;
        cout<<" , months:"<<m+1;
        cin>>sales[d][m];
    }
    display (sales);
}

```

```

void display (int funcsales [districts] [months])
{
    int d,m;
    for (d=0; d<districts; d++)
    {
        cout <<"in district"<< d+1;
        for (m=0; m<months; m++)
            cout<<funcsales[d][m];
        cout<<endl;
    }
}

```

**Q4. Write a program that the** roll numbers, marks in English, Computers, Maths out of 100 for 50 students (i.e. need no read them)

Write a function in c++, using structures, to calculate the following:-

- (i) No. of students passed with distinction
- (ii) Details of top two students
- (iii) Number of students failed

For distinction, a student needs to score atleast 75% and minimum marks are 40%

**Ans** `#include<iostream.h>`  
`#include<conio.h>`  
`const int size=50;`  
`struct kvstudent`  
`{`  
`int kvrollno;`  
`float kveng;`  
`float kvcomp;`  
`float kvmaths;`  
`}`  
`kvstudent sarr[size],t1,t2;`  
`float total,avg,top1=0,top2=0;`  
`int ndist=0, nfail=0;`  
`void kvresult()`  
`{`  
`clrscr();`  
`for (int= 0;int<size;i++)`  
`{`  
`total= sarr[i].kveng + sarr[i].kvcomp+sarr[i].kvmaths;`  
`avg= total/3;`  
`if (avg>=75)`  
`ndist++;`  
`else if(avg<40)`  
`nfail++;`  
`if(top1<avg)`  
`{`  
`top1=avg;`  
`t1=sarr[i];`  
`}`  
`}`

```

else if (top2<avg && avg <=top1)
{
top2=avg;
t2= sarr[i];          }      }
cout<< "\n total number of distinction holders are : "<<ndist<< endl;
cout<< "\n toal number of failed students are:"<<nfail<<endl;

cout<< "\n Ist Topper is (Top Scorer) : \n";
cout<< " Roll Number : "<<t1.kvrollno<< "\t English:" <<t1.kveng
    << "\t Computers:" <<t1.kvcomp<<<< "\t Maths:" <<t1.kvmaths
    << "\n Aggregate % : " <<top1<<endl;

cout<< "\n IInd Topper is : \n";
cout<< " Roll Number : "<<t2.kvrollno<< "\t English:" <<t2.kveng
    << "\t Computers:" <<t2.kvcomp<<<< "\t Maths:" <<t2.kvmaths
    << "\n Aggregate % : " <<top2<<endl;
}

```

**Q5. What do you think about polymorphism and how you can explain for effective coding as a part of Object Oriented Language?**

**Ans** Polymorphism in an object oriented programming language works under main() and with statements like decision statements and looping statements and works under the pointers and structures. A polymorphism is also be understandable as a function overloading. This is the ability of an object to behave differently in different circumstances can effectively be implemented in programming through function overloading.

It helps in coding to represent the same function in different modes of program. The programmer is relieved from the burden of choosing the right function for a given set of values. This important responsibility is carried out by the compiler when a function is overloaded.

**Q6. How we can overload binary operator?. Expalin with example.**

**Ans** **Binary Operator** can be overloaded in the same manner as unary operator. We can take an example of overloading 'equal to' (=) operator. We will use this operator to compare the strings, returning values 'true' if strings are same and false otherwise.

**Program of overloading binary operator(=), declaration of functions inside the class.**

```

#include <iostream.h>
#include<string.h>
enum boolean { true , false };          // use of enum
class string
{
private:
char s[100];
public:
string()
{
strcpy(s, "");
}

```

```

string (char a[])           // overloading of string function
{
    strcpy (s,a);
}
void display()
{
    cout<<s;
}
void gets()
{
    cin.get(s,100)
}
Boolean operator ==(string ss)
{
    return (strcmp(s,ss.s)==0)?true :false;
}
};

void main()
{
    string s1= "overlaod";
    string s2;
    cout<< "\n enter a word";
    s2.gets();           //gets strings from user
    if (s2== s1)
        cout<<" you typed a correct word \n";
    else
        cout<< " Invalid Match \n";
}

```

**Q7. How we can overload constructor?. explain with example.**

Ans The constructor is defined as class name. A constructor of a class may also be overloaded so that even with different number and types of initial values, an object may still be initialized.

```

#include<iostream.h>
#include<conio.h>
class Deposit
{
    long int principal;
    int time;
    float rate;
    float total_amt;
public:
    Deposit();
    Deposit(long p, int t, float r);
    Deposit(long p, int t);
    Deposit(long p, float r);
    void calc_amt(void);
    void display(void);
};

```

```

Deposit:: Deposit()
{
principal =time=rate=0.0;
}

Deposit:: Deposit(long p, int t, float r)
{
principal=p;
time=t;
rate =r;
}

Deposit:: Deposit(long p, int t)
{
principal=p;
time=t;
rate =0.07;
}

Deposit:: Deposit(long p, float r)
{
principal=p;
time=4;
rate =r;
}

void Deposit::calc_amt(void)
{
totat_amt= principal +(principal *time * rate)/100;
}

void Deposit::display (void)
{
cout<< "\n Principal Amount : Rs."<<principal;
cout<< "\n Period of Investment: "<< time << "years";
cout<< "\n Rate of intrest : "<<rate;
cout<< "\n Total amount is:Rs. :- > " <<total_amt;
}

void main()
{
clrscr();
Deposit D1;
Deposit D2(5000,2,0.05);
Deposit D3(6000,4);
Deposit D4(4000,0.08);

D1.calc_amt();
D2.clac_amt();
D3.calc_amt();
D4.clac_amt();
cout<< "\n display of object One is: ";
D1.display();
cout<< "\n display of object Two is: ";

```

```

D2.display();
cout<< "\n display of object Three is: ";
D3.display();
cout<< "\n display of object Four is: ";
D4.display();
}

```

**Q8. Find the errors in the following program. State reasons:**

```
#include<iostream.h>
```

```

class A
{
    int a1;
    public:
    int a2;
    protected:
    int a3;
};
class B: public A
{
    public:
    void func()
    {
        int b1,b2,b3;
        b1=a1;
        b2=a2;
        b3=a3;
    }
};
class C: A
{
    public:
    void f()
    {
        int c1,c2,c3;
        c1=a1;
        c2=a2;
        c3=a3;
    }
};

```

```

int main()
{
    int p,q,r,i,j,k;
    B 01;
    C 02;
    p=01.a1;
    q=01.a2;
    r=01.a3;
    i=01.a1;
    j=01.a2;
    k=01.a3;
}

```

```
return 0;    }
```

**Ans** The errors in the above given program are as described below:

1. **B::func()** cannot access **A::a1** as **a1** is a private member of **A**, Therefore, **b1=a1;** is illegal
2. **C=f()** cannot access **A::a1** as **a1** is a private number of **A**, Therefore, **c1=a1; is illegal**
3. In **main()**, the statement **p=01.a1;** is illegal because **a1** is not the public member of **B** and hence cannot be accessed directly by its objects.
4. In **main()**, the statement **r = 01.a3;** is illegal for the same reason as specified in point 3.
5. The statements  

```
i = 02.a1;
f = 02.a2;
k = 02.a3;
```

are illegal as neither of **a1,a2** and **a3** are public members of **C** (which is inheriting privately from **A**) and hence cannot be accessed directly by the objects of **C** class.

**Q9. What will be the output of the following:**

```
#include <iostream.h>
void main()
{
    int v1=5, v2=10;
    for (int x=1; x<=2; x++)
    {
        cout<< ++v1 << "\t" << v2-- << endl;
        cout<< --v2 << "\t" << v1++ << endl;
    }
}
```

**Ans** The output will be

```
6      10
8      6

8      8
6      8
```

**Q10 Write a program that reads a string and counts the number of vowels, words and blank spaces present in the string.**

**Ans**

```
#include<iostream.h>
#include<stdio.h>
main()
```

**Q11 Identify the errors in the following code segment**

```
int main()
{
    cout<<" enter two numbers";
    cin>>num>>auto;
    float area=length * breadth;
}
```

**Ans**

1. Variable auto is invalid ( it is a reserved keyword)
2. Variables (num,auto (though invalid) are not defined before their usage.
3. Even though variable length and breadth are also not defined
4. Return statement is missisng

**Q12 Name the header file for using in built functions in the program**  
**(i) setw(), (ii) puts(), (iii) isdigit(), (iv) fabs()**

**Ans Header files are**  
**(i) iomanip.h, (ii) stdio.h (iii) ctype.h (iv) math.h**

**Q13. Write a program to generate a function with parameters and array in function**  
**e.g. show is function name**

**then show(int[ ],int);**

**Ans**

```
#include<iostream.h>
#include<conio.h>
main()
{
clrscr();
int a[5];
void show(int[],int);           // declaration of a function (prototype)
cout<<"enter the number=";
for(int i=0;i<5;i++)
{
cin>>a[i];
}
cout<<"ARRAYS"<<endl;
show(a,5);                     // calling of function
getch();                       // for freeze the monitor
}
void show(int s[ ],int n)       //defining of a function
{
for(int i=0;i<n;i++)
{
cout<<s[i]<<endl;
}
}
}
```

**Q. 14 What will be the output of following code fragment ?**

```
#include<iostream.h>
#include<conio.h>
main()
{
clrscr();
int a[] = {3, 5, 6, 7};
int *p, **q, ***r, *s, *t, ** ss;
p = a;
s = p + 1;
q = &s;
t = (*q + 1);
```

```

ss = &t;
r = &ss;
cout << *p << '\t' << **q << '\t' << ***r << end;

```

Ans : 3      5      6

**Q15** What is the relationship between an array and a pointer ? Given below a function to tranverse a character array using For-loop. Use a pointer in place of an index X and substitute for-loop with while-loop so that the output of the function stringlength() remains the same.

```

int stringlength(char s[])
{
    int count = 0;
    for (int x = 0; s[x]; x++)
        cout++;
    return (count);
}

```

Ans : The relationship between an array and a pointer is that the name of an array is actually a pointer pointing to the first element of the array.

```

int stringlength(char s[])
{
    int count = 0;
    while (*s)
    {
        count++;
        s++;
    }
    return (count);
}

```

**Q16** Give the output of the following program segment : (assuming all required header files are included in the program)

```

char *name = "KenDriYa";
for (int x = 0; x < strlen(name); x++)
    if (islower (name[x]) )
        name[x] = toupper (name[x]);
    else
        if ( isupper (name[x]) )
            if (x%2 != 0)
                name[x] = tolower (name[x-1])
            else
                name[x]--;
cout << name << endl;

```

Ans : jENnRiXA

**Q17** What do you under by memory leaks ? What are the possible reasons for it ? How can memory leaks be avoided ?

Ans : If the objects, that are allocated memory dynamically, are not deleted using delete, the memory block remains occupied even at the end of the program. Such memory blocks are known as orphaned memory blocks.

This orphaned memory blocks when increase in number, bring as adverse effect on the system. This situation is known as memory leak. The possible reasons for this are :

- (i) A dynamically allocated object not deleted using delete.
- (ii) Delete statement is not getting executed because of some logical error.
- (iii) Assigning the result of a new statement to an already occupied pointer.

The memory leaks can be avoided by

- (ii) making sure that a dynamically allocated object is deleted.
- (iii) A new statement stores its return values (a pointer) in a fresh pointer.

**Q18** Predict and explain the output of the following program :

Ans : `#include<iostream.h>`

`#include<conio.h>`

`int main()`

```
{
    clrscr();
    float x = 5.999;
    float *y, *z;
    y = &x;
    z = y;
    cout << x << " , " << "&x" << " , " << *y << " , " << *z << "\n";
    return 0;
}
```

**Ans :** The output of this program will be

5.999, 5.999, 5.999, 5.999

The reason for this is x gives the value stored in the variable x. \*(&x) gives the data value stored in the address &x i.e. address of x i.e. the data value of x. Since y points to x ( y = &x), \*y gives the value of x. And because z has the same address as that of y. \*z also gives the value of x i.e. 5.999.

**Q19** Give the output following program :

`#include<iostream.h>`

`Int a = 13;`

`Void main()`

```
{
    Void demo(int &, int , int *);
    Int a = 7, b = 4;
    Demo (::a, a, &b);
    Cout << ::a << " " << a << " " << b << endl;
}
Void demo(int &x, int y, int *z)
{
    A += x;
    Y * = a;
    *z = a + y;
    Cout << x << " " << y << " " << *z << endl;
}
```

**Ans :**

26	182	208
26	7	208