POINTERS

Pointer is a variable that holds a memory address of another variable of same data type.

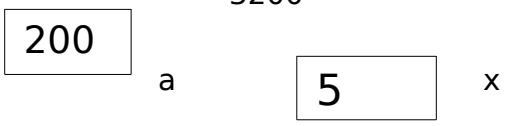
- It supports dynamic allocation routines.
- It can improve the efficiency of certain routines.

For example:

```
#include<iostream.h>
void main()
{
        int x=5;
        int *a;//here 'a' is a pointer to int which can store address of an object of
        type int
        a=&x;//address of x is assigned to pointer 'a', same as int *a=&x;
        cout<<"Value of x is : "<<x<<endl;
        cout<<"Address of x is : "<<&x<<endl;
        cout<<"Address stored in a is : "<<a<<endl;
        cout<<"Values stored at memory location pointed by a is : "<<*a<<endl;
        cout<<"Address of a is :"<<&a<<endl;
}
```

Output:

2000
Value of x is : 5
Address of x is : 2000
Address stored in a is : 2000
Value stored at memory location pointed by a is : 5
Address of a is : 3200

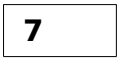


In the above example the **&x (&x means Address of, & is called Reference operator)** gives address of x which in this case happen to be 2000 (assumed value and always displayed as **HEXADECIMAL NUMBERS ie. For example as 0x867ffeda**) and **dereference operator *(contents at address)** gives the value stored in the memory location stored in pointer variable 'a'. Since a is also a memory location it also has an address, here it is assumed as 3200.

& (used in other than declaration statement) means *Address of*
and * means *At Address(to retrieve contents of a given address)*
& if used in a declaration statement means a *reference variable or alias*

Reference variable:

For example int p=7;



int &q=p; // here q is a reference variable of p, & in a declaration means **sharing/reference variable/alias** name(another name). Any change in q will affect p and vice-versa.

We can create any number of reference for a variable and any reference variable should be an initialized variable and a reference can be created for an original variable ie. Here int &r=q; is an error since we *cannot* create reference for a reference or pointer variable

Declaration and Initialization of Pointers :

Datatype *variable_name;

Syntax : Datatype *variable_name;

Eg. int *p; float *p1; char *c;

Some properties of pointer variables

- **Pointers can be of any data type.**
- **For EVERY pointer(any data type, char,int ,float, struct, class)** 2 bytes **is allocated in memory since it holds an address of a variable in memory and any address is an integer and every integer needs 2 bytes.**
- **Address of Variable of one datatype cannot be stored in pointer of another datatype.ie. the datatype of the pointer should be same as the address of the variable it is pointing to.**

```
int x=5;
float *q=&x; // ERROR
char * r=&x; //ERROR , DIFFERENT DATA TYPES
```

Pointer arithmetic:

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

e.g. int *p;

p++;

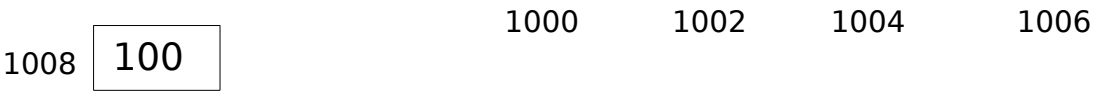
If current address of p is 1000, then p++ (++p or p=p+1) statement will increase p to 1002, not 1001.

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

Increment/decrement operation can be done on pointer variables. For example if p contains an address 1000, p+1 means 1000+1, ie 1000+ 1 location, 1000+2 bytes since an int variable is of 2 bytes, similarly if p+4 means 1000+4 which is 1000+ 4*2 ie 1008. Similarly if q is declared as float *q and if q contains the address 3000 , q+1 means 3000+4 bytes since float is of 4 bytes.

Pointers and Arrays in C++

For example int a[5]={10,20,35};



a

10	20	35	0	0

a is of type int * a[0] a[1] a[2] a[3] a[4]

Whenever an array is declared compiled creates a constant pointer having same name that of the array to store the base address of the array. ie ***array name itself is a constant pointer created by the compiler to store the base address of the array***. 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.** Here a is type int * (since it holds address of a[0] which of type int) and it is a constant ie the address stored in a (1000) cannot be changed.)

cout<<a; //1000 cout<<*a; //10 (*(1000)= 10)
cout<<a[0]; //10 cout<<&a[0]; //1000 (Address of a[0])
cout<<*&a[0];// 10
int * p;
The following assignment operation would be valid: **p = a; // or int *p=a;**
After that, p and a would be equivalent and would have the same properties.
The only difference is a is an array, and an array can be considered as a constant pointer implicitly created by compiler whereas p is a pointer variable created explicitly by user.
cout<<p; //1000 cout<<*p; //10 (*(1000)= 10)
cout<<p[0]; //10 cout<<&p[0]; //1000 (Address of p[0])

***a is equal to * p is equal to a[0] is equal to p[0]**

a[i] is interpreted by the compiler during compilation as *(a+i) ,

a[i] is the **subscript method or index** method and
*(a+i) is the **pointer method**.

In the above example a++,--a, a=a+2 etc are invalid as a is a constant pointer whereas p++, --p, p=p+2 etc are valid. a[i] is equivalent to *(a+i) is equivalent to *(i+a)
for example if the base address of a is 1000 then address of a[3] is calculated as &a[3] =1000 + 3 * sizeof (int) = 1000 + 3 * 2=1000 + 6 = 1006 Note that a[3] and *&a[3] and *(a+3) will give the same value i.e. 0.

Arithmetic operation on pointers
We can increment or decrement a pointer variable for example

```
float a[5]={3.0,5.6,6.0,2.5,5.3};
float *p=a;
++p;
--p;
```
if the base address of a is 1000 then statement ++p will increment the address stored in pointer
variable by 4 because p is a pointer to float and size of a float object is 4 byte, since ++p is equivalent to p=p+1 and address of p+1 is calculated as Address stored in p + 1 * sizeof (float)= 1000 + 1 * 4 = 1004. We can add or subtract any integer number to a pointer variable because adding an integer value to an address makes another address e.g.

```
void main()
{
int p[10]={8,6,4,2,1};
 int *q;
q=p;//address of p[0] is assigned to q assuming p[0] is allocated at memory location 1000
q=q+4;//now q contains address of p[4] i.e. 1000+4*sizeof(int)=1000+4*2= 1008
cout<<*q<<endl;
q=q-2;//now q contains address of p[2] i.e. 1008-2*sizeof (int)=1008-2*2=1004
cout<<*q<<endl;
}
```
Output is
1
4

Addition of two pointer variables is meaningless.

Subtraction of two pointers is meaningful only if both pointers contain the address of different
elements of the same array

For example

```
void main()
{
int p[10]={1,3,5,6,8,9};
int *a=p+1,*b=p+4;
p=p+q; //error because sum of two addresses yields an illegal address
int x=b-a; //valid
cout<<x;
}
```
Output is
3

In the above example if base address of a is 1000 then address of a would be 1002 and address of b would be 1008 then value of b-a (p+4-(p+1)=3) is calculated as

(Address stored in b-address stored in a)/sizeof(data type in this case int)

(1008-1002)/2=3

Multiplication and division operation on a pointer variable is not allowed

```
void main()
{          clrscr();
       int b[]={11,22,33,44,55};
       int *q=b;
       int y=*q++;  //y=*q, q=q+1
       cout<<*q<<"  ";
       cout<<y;
}
output
22    11
void main()
{              clrscr();
          int b[]={11,22,33,44,55};
          int *q=b;
          int y=(*q)++;  //y=*q,*q=*q+1;
          cout<<*q<<"     ";
          cout<<y;
}
output
12    11
void main()
{                  clrscr();
           int b[]={11,22,33,44,55};
           int *q=b;
           int y=*++q;  //q=q+1,y=*q;
           cout<<*q<<"   ";
           cout<<y;
}
output
22   22

void main()
{              clrscr();
          int b[]={11,22,33,44,55};
          int *q=b;
```

```
        int y=++*q;  //*q=*q+1,y=*q;
        cout<<*q<<"   ";
        cout<<y;
}
output
12   12
```

We cannot assign any value other than 0 to a pointer variable, only address can be assigned. ie a pointer variable can be assigned/initialized using & operator or using pointer of same type .
For example
int *p;
p=5; //not allowed
p=0; //allowed because 0 is a value which can be assigned to a variable of any data type

Pointers and strings
Consider the declaration **char a[]=”Computer”;**
cout<<a;   //Computer
cout<<*a; //C
a=”Pen”;// error

If an address is of type char * and if the address is asked to be displayed instead of displaying the address the entire contents till \0 is displayed and *a means the particular char stored in the given address.

The above array can also be declared as **char *a=”Computer”;**
cout<<a;// Computer
cout<<*a;//C
a=”Pen”// valid

First address of C is stored in a and a=“Pen” means a new location is allocated in memory for Pen and the address of P is stored in a(CHANGE IN ADDRESS). If strcpy(a,”Pen”) is used the contents Computer is replaced as Pen(CHANGE IN CONTENTS).

= MEANS CHANGE IN ADDRESS
AND strcpy() MEANS CHANGE IN CONTENTS.

The difference in both the declarations is that in the first case a is a constant pointer created by the compiler whereas in the second case a is a pointer variable created by user. The contents can be changed using strcpy() in the first and second case whereas the address can be

changed in the second case using = but not in the first case as array name is a constant pointer.

```
void main()
{char *p="aeiou";
char q=++*p++;   // same output for q=++(*p++),
// contents of p is incremented and stored in q and then p is incremented
cout<<"\n"<<q;//b
cout<<"\n"<<p;//eiou
cout<<"\n"<<*p;//e
}
```

Output

b

eiou

e

```
void main()
{char *p="aeiou";
char q=(++*p)++;  // no change in address stored in p
// contents of p is incremented twice
cout<<"\n"<<q;   //b
cout<<"\n"<<p;   //ceiou
cout<<"\n"<<*p;  //c
}
```

Output

b

ceiou

c

Note: the operation ++(*p)++ is a syntax error

```
void fun(char a[])  // here a is a constant pointer which now contains the address stored in b
{   strcpy(a,"xyz");  // contents of  address stored in a is changed
    cout<<"\n in function a="<<a;
}// during 1st call contents of 1000 is replaced and 2nd call contents of address 2001.
void main()
{   clrscr();
    char b[20]="abc";  // Assuming address of abc as 1000
 // here an array is created and abc is stored and it base address is stored in a constant pointer b
```

```
    fun(b);                                                    // Address stored in b is
passed . ie 1000
    cout<<"\n in main after calling b="<<b;
    char c[20]="abc"; // here c is a constant pointer
    fun(c+1); // if c contains address 2000 then 2001 is passed to function
    cout<<"\n in main after calling c="<<c;
}
```

Output

in function a=xyz
in main after calling b=xyz
in function a=xyz
in main after calling c=axyz

```
void fun(char *a) // char a[]
{    strcpy(a,"xyz");
      cout<<"\n in function a="<<a;
}
void main()
{     clrscr();
     char *b="abcdefgh"; // here b is a pointer variable
      fun(b);
      cout<<"\n in main after calling b="<<b;
}
```

Output:

in function a=xyz
in main after calling b=xyz

```
void fun(char *a) // char a[] is an error, here a is pointer variable and a contains
address of string abc
{  a="uvw"; // now address of string uvw is stored in a
   cout<<"\n in function a="<<a;
}
void main()
{  clrscr();
   char *b="abc";  // now address of string abc is stored in b
    fun(b); // b is passed to a
    cout<<"\n in main after calling b="<<b;
}
```

Output

in function a=uvw

in main after calling b=abc

```
void fun(char **a)
{ char *b="pqrst";
  *a=b;
  cout<<"\n in function a="<<*a<<" b="<<b;
}
void main()
 {char *c="abcdef";
  fun(&c); // here address of c is passed
  cout<<"\n in main after calling c="<<c;
}
```

Output:

in function a=pqrst b=pqrst

in main after calling c=pqrst

Null Pointer

A null pointer is a regular pointer of any pointer type which has a special value that indicates that it is not pointing to any valid reference or memory address. This value is the result of type-casting the
integer value zero to any pointer type.

```
int *q=0; // q has a NULL pointer value
float * p;
p=NULL; // p has a NULL (NULL is defined as a constant whose value is 0) pointer value
```

Note: 0 memory address is a memory location which is not allocated to any variable of the
program.

Pointer to structure (->) as dereferencing instead of *

```
struct emp
{
int empno;
char name[20];
float sal;
};
void main()
{
emp eob={567,”Ajay K”, 30000};
emp *p=&eob;
```

```
cout<<p->empno<<"\t"<<p->name<<"\t"<<p->sal<<endl;
p->sal=p->sal +1000;
cout<<(*p).empno<<"\t"<<p->name<<"\t"<<(*p).sal;
}
```

Output is

```
567  Ajay K  30000
567  Ajay K  31000
```

Structure variable name . data member is same as Pointer -> data member. Here **->(arrow operator)** is the **dereferencing operator** in the case of structures and classes.

Pointer to class

```
class emp
{   int empno;
  char name[20];
  float sal;
  public:
             void getemp()
             {   cout<<"Enter employeenumber:";                cin>>empno;
                cout<<"Enter name:";                gets(name);
                cout<<"Enter salary:";                cin>>salary;
            }
             void putemp()
              {     cout<<empno<<"   "<<name<<"     "<<salary;
               }
};
void main()
{    emp ob;
     emp *p=&ob;
     ob.getemp();   // same as p->getemp();
      ob.putemp();  // same as p->putemp();
}
```

this pointer

When an object of a class invokes its member function it passes its own address implicitly through a special pointer called ***this*** pointer. For example

```
#include<iostream.h>
class A
{        int x;
public:  void input()
        {     cout<<"enter a number :";
              cin>>x;
        }
        void output()
```

```
        {       cout<<"address of the object is :"<<this<<endl;
                cout<<this->x<<"\t"<<x<<endl ; // both this->x and x will give the
        same value
        }
};
void main()
{A a1, a2;
a1.input();
a2.input();
a1.output();
a2.output();
}
```

Output is

enter a number : 5

enter a number : 8

address of the object is : 1000 //assumed address of a1

5 5

address of the object is : 2000 //assumed address of a2

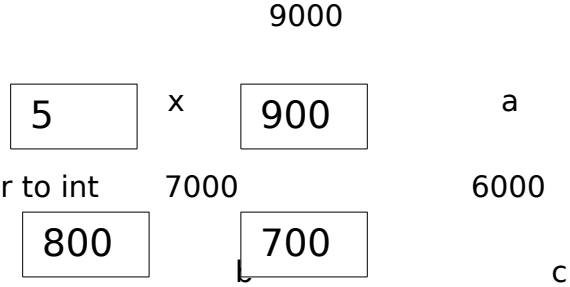
8 8

Note: the ***this*** pointer does not exist outside the member function, i.e. we cannot define or access this pointer in main function because this pointer comes into existence only when a member function of a class is invoked by an object of the class.

Pointers to pointers

C++ allows the use of pointers that point to pointers, that these, in its turn, point to data (or even to other pointers). In order to do that, we only need to add an asterisk (*) for each level of reference in their declarations:

```
void main()
{
int x=5;
int *a;       // a is pointer to int
int **b;      //b is pointer to pointer to int
int ***c;     //c is pointer to pointer to pointer to int
a=&x;
b=&a;
c=&b;
cout<<x<<"\t"<<*a<<"\t"<<**b<<"\t"<<***c<<endl;
}
```



Output is

5 5 5 5

Assuming variable x, a, b, c are allocated randomly at memory location 9000, 8000, 7000, and 6000
respectively. The value of each variable is written inside each cell; under the cells are their respective
addresses in memory.
c has type int*** and a value of 7000
*c has type int ** and a value of *(7000) ie contents of 7000 which is equal to 8000
**c has type int * and a value of 9000
***c has type int and a value of 5

Pointers and CONST :

A constant pointer means that the pointer in consideration will always point to the same
address. Its cannot be modified. A pointer to a constant refers to a pointer which is pointing to a symbolic constant.

```
int *p1 = &x;  // non-const pointer to non-const int
const int *p2 = &x;  // non-const pointer to const int
int * const p3 = &x;  // const pointer to non-const int
const int * const p4 = &x;  // const pointer to const int
```

Look the following example :

```
int a = 20; // integer a declaration
int *p = &a; // pointer p to an integer a
// Here a and p can be changed, p++,a++ etc are valid
int * const q = &a; // a const pointer q to an integer a
/*Here const with q indicates that q is a constant ie address stored in q should not change(ie always address of m), q++ ,q=q-2 are invalid, but a can be changed since a is of type int , *a=9 is valid since here q remains unchanged*/
const int  b = 10; // a const integer b
const int *r = &b; // a pointer to a const int
//Here r  is pointer variable to a constant integer, ie r can hold address of any integer constants
const  int c=20;
r=&c;  // now r contains address of c instead of b
//*r=8; is invalid since it points towards a constant and it cannot be modified but r++,r=r+2 are valid
//ie address can be changed
const int * const  s = &b; // a const pointer to a const integer
//Here neither   s   nor   b can be changed
```

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;
}
```

Input :
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);
Example:

```
#include <iostream.h>
 int*  min(int &, int &);  // function is returning address of an integer
 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);      // returns the address of x
else
return (&y)
}
```

Pointers to functions

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

int add(int a,int b)
{return a+b;
```

```
}
int sub(int a,int b)
{return a-b;
}
void main()
{clrscr();
int (*p)(int,int);   //here p is not a function but a pointer variable
/* The () indicates that it is a pointer to a function p can hold address of any function having return datatype int and having two arguements of type int. it is different from int *p(int,int); which means p is a function having return datatype int* and two arguements of type int. */
int x=23,y=5;
p=&add;      // or p=add;
cout<<(*p)(x,y)<<"\n";                    //or cout<<p(x,y)<<"\n";
p=sub;
cout<<p(x,y);
}
```

Output

28

18

C++MemoryMap : In c++ memory is logically divide into

- 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 Dynamic Memory Allocation.

Static memory allocation and Dynamic memory allocation

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. If memory allocation is done at the time of compilation of the program then memory allocation is known as static memory allocation. For example memory allocation for variables and arrays are done at compilation of the program as size of variable and arrays are already known at the time of compilation of the program.

e.g. int a; // This will allocate 2 bytes for a during compilation.

int b[5];

Dynamic Memory Allocation : The amount of memory to be allocated is not known beforehand rather it is required to allocated as and when required during

runtime, it is referred to as dynamic memory allocation. C++ offers two operator for DMA – **new and delete**.
e.g int x =new int; float y= new float; // dynamic allocation
delete x; delete y; //dynamic reallocation
Dynamic memory allocation is the memory allocation required during execution of the program. For
example if the amount of memory needed is determined by the value input by the user at run time.
Dynamic memory allocation using new operator
int *a=new int(5); /* new operator will create an int object somewhere in memory heap
and initialize the object with value 5 and returns address of the object to pointer variable a */
int n;
cin>>n;
int *b=new int[n]; /*here new operator will create an array of 5 int and return the base
address of the allocated array to pointer variable b */
If the new operator fails to allocate memory then it returns NULL value which indicates that the
required memory is not available for the requested instruction.
Note: We cannot create array of variable length statically for example
int n;
cin>>n;
int a[n]; // error because we can use only constant integral value in array declaration
const int p=3;
int a[p]; //legal

Two dimensional array : We cannot create a 2d dynamic array in C++
The only solution is to create a 1d array and process it as 2d.
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];
}
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.

Operators delete and delete[ ]

Since the necessity of dynamic memory is usually limited to specific moments within a program, once
it is no longer needed it should be freed so that the memory becomes available again for other
requests of dynamic memory. This is the purpose of the operator delete, whose format is:

```
delete pointer; //used to delete the memory allocated for single object
delete pointer[];//used to delete the memory allocated for array of objects
```

The value passed as argument to delete must be either a pointer to a memory block previously
allocated with new, or a null pointer (in the case of a null pointer, delete produces no effect).

```
void main()
{int n;
Int *q;
cout<<"enter no of integers required :";
cin>>n;
q=new int [n];
if(q==NULL)
cout<<"memory could not be allocated";
else
{for(int i=0;i<n;i++)
q[i]=i+1;
for(i=0;i<n;i++)
cout<<q[i]<<" ";
delete q[ ];
}
}
```

For n=3 and successful memory allocation the output of the program would be
1 2 3

Otherwise on failure of new operator the message "memory could not be allocated" would be the
output. It is always a good habit to check the pointer with NULL value to make sure that memory is allocated or not.

Memory leaks/Memory bleeding

If dynamically allocated memory has become unreachable (no pointer variable is pointing the
allocated memory) then such situation is known as memory leak because neither it can be accessed
nor it can be deleted from the memory heap by garbage collection. For example

```
void func()
```

```
{
int x=5;
int *p;
p=new int [100]; // dynamic memory allocation for an array of 100 integers
p=&x;
}
```

In the above example the statement **p=new int [100]** assign the address of the dynamically allocated memory to p and the next statement **p=&x** assign the address of x to p therefore, after that the dynamically allocated memory become unreachable which cause memory leak. So every location should be explicitly removed using delete after use before the scope of the pointer in which the address is stored.

Dynamic structures/class :

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

```
student *stu;
stu = new Student;
```

A dynamic structure/class can be released using the reallocation operator delete as shown below :

```
delete stu;
```

<u>void pointers</u>

void pointer is a special pointer which can store address of an object of any data type. Since void pointer do not contain the data type information of the object it is pointing to we cannot apply dereference operator * to a void pointer without using explicit type casting.

```
void main()
{       int x=5;
        float y=3.5;
int *p;
float *q;
void *r;
p=&y; //error cannot convert float * to int *
q=&x; //error cannot convert int * to float *
p=&x; //valid
cout<<*p <<endl; //valid
q=&y; //valid
cout<<*q<<endl ; //valid
r=&x; //valid
cout<<*r<<endl; //error pointer to cannot be dereference without explicit type cast
```

```
cout<<*(int *)r<<endl; // valid and display the values pointed by r as int
r=&y; //valid
cout<<*(float *)r<<endl; //valid and display the values pointed by r as float
}
```

Note: Pointer to one data type cannot be converted to pointer to another data type except
pointer to void

Array of pointers and pointer to array

An array of pointer is simply an array whose all elements are pointers to the same data type.

For example

```
void main()
{
float x=3.5,y=7.2,z=9.7;
float *b[5]; // here b is an array of 5 pointers to float
b[0]=&x;
b[1]=&y;
b[2]=&z;
cout<<*b[0]<<"\t"<<*b[1]<<"\t"<<*b[2]<<endl;
cout<<sizeof(b)<<"\t"<<sizeof(b[0])<<"\t"<<sizeof(*b[0]) ;
}
```

Output is

```
3.5    7.2   9.7
10      2       4
```

In the above example the expression sizeof(b) returns the number of bytes allocated to b i.e. 10
because array b contain 5 pointers to float and size of each pointer is 2 bytes. The expression
sizeof(b[0]) returns the size of first pointer of the array b i.e. 2bytes. The expression sizeof(*b[0])
returns the size of the data type the pointer b[0] points to which is float, so the expression returns 4 as output.

Pointer to array is a pointer which points to an array of for example

```
void main()
{
float a[10], b[5],c[10];
float (*q)[10]; //here q is a pointer to array of 10 floats
q=&a; //valid
q=&c; //valid
q=&b; //error cannot convert float [5]* to float [10]*
cout<<sizeof(q)<<"\t"<<sizeof(*q)
```

}
After removing the incorrect statement q=&b from the program the output of the program is
2 40
Because q is a pointer and pointer to any thing is and address which is of 2bytes. But sizeof(*q)
returns the size of an array of 10 floats i.e. 40 because *q points to an array of 10 floats.

Exercise

1. Give the output of the following program:

```
void main()
{char *p = “School”;
char c;
c = ++ *p ++;
cout<<c<<“, ”<<p<<endl;
cout<<p<<“, ”<<++*p- -<<“, ”<<++*p++;
}
```

2. Give the output of the following program:

```
void main()
{int x [ ] = {50, 40, 30, 20, 10}:
int *p, **q, *t;
p = x;
t = x + 1;
q = &t;
cout << *p <<“, ” << **q << “,” << *t++;
}
```

3. Give the output of the following program (Assume all necessary header files are included):

```
void main( )
{char * x = “TajMahal”;
char c;
x=x+3 ;
c = ++ *x ++;
cout<<c<<”, “;
cout<< *x<<“, ”<<- -*x++<<- -x ;
}
```

4. What will be the output of the program (Assume all necessary header files are included):

```
void main( )
{int a[ ]={4,8,2,5,7,9,6}
int x=a+5,y=a+3;
cout<<++*x- -<<”,”<<++*x- -<<”,”<<++*y++<<”,”<<++*y- -;
```

}

5. What will be the output of the program (Assume all necessary header files are included):

```
void main()
{int arr[ ] = {12, 23, 34, 45};
int *ptr = arr;
int val = *ptr ; cout << val << endl;
val = *ptr++; cout << val << endl;
val = *ptr; cout << val << endl;
val = *++ptr; cout << val << endl;
val = ++*ptr; cout << val << endl;
}
```

3 Marks Questions

1.Give output of following code fragment assuming all necessary header files are included:

```
void main()
{char *msg = “Computer Science”;
for (int i = 0; i < strlen (msg); i++)
if (islower(msg[i]))
msg[i] = toupper (msg[i]);
else
if (isupper(msg[i]))
if( i % 2 != 0)
msg[i] = tolower (msg[i-1]);
else
msg[i--];
cout << msg << endl;
}
```

2. What will be the output of the program (Assume all necessary header files are included):

```
void main( )
{clrscr( );
int a =32;
int *ptr = &a;
char ch = ‘A’;
char  &cho=ch;
cho+=a;
*ptr + = ch;
cout<< a << “” <<ch<<endl;
}
```

3. What will be the output of the program (Assume all necessary header files are included):

```
void main( )
```

```
{char *a[ ]={“DELHI”, “MUMBAI”,”VARANSAI”};
char **p;
p=a;
cout<<sizeof(a)<<”, “<<sizeof(a[0])<<”, “<<sizeof(p)<<endl;
cout<< p << “, ”<<++*p<<“, ”<<*++p<<endl;
cout<< **a << “, ” <<*(a+1)<<“, ” <<a[2]<<endl;
}
```

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 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.