POINTERS

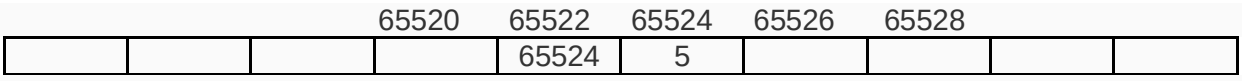
Pointer is a memory variable which can store address of an object of specified data type. For example:

```
#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'
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:

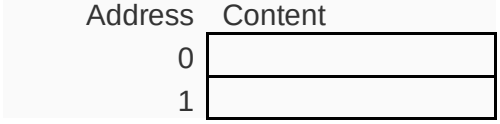
```
Value of x is : 5
Address of x is : 65524
Address stored in a is : 65524
Value stored at memory location pointed by a is : 5
Address of a is : 65522
```

In the above example the **&x** gives address of x which in this case happen to be 65524 and **dereference operator *** gives the value stored in the memory location stored in pointer variable 'a' as shown in the figure below.



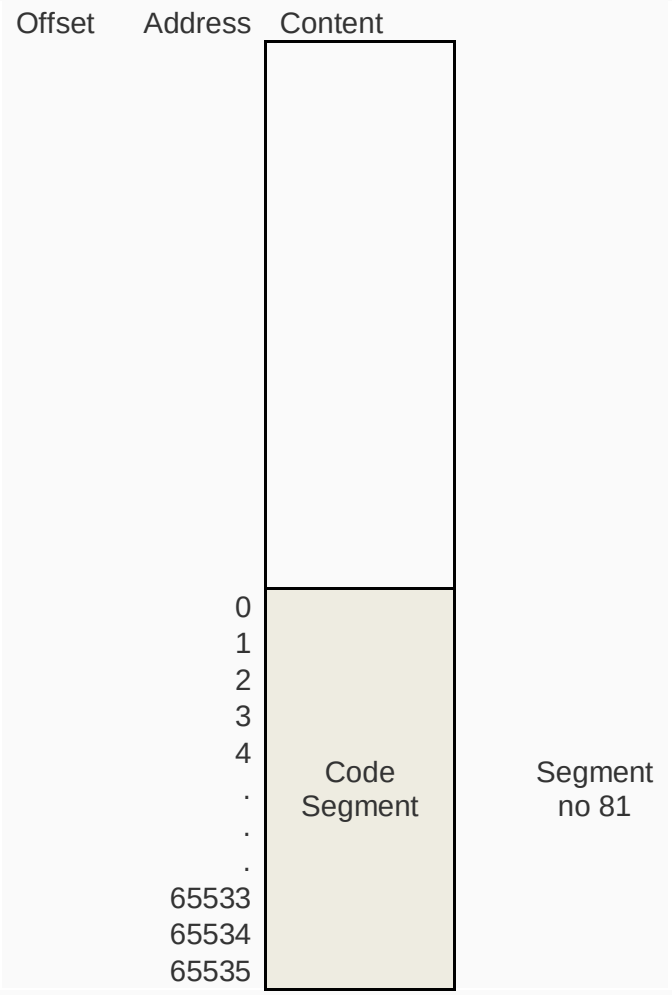
Some properties of pointer variables

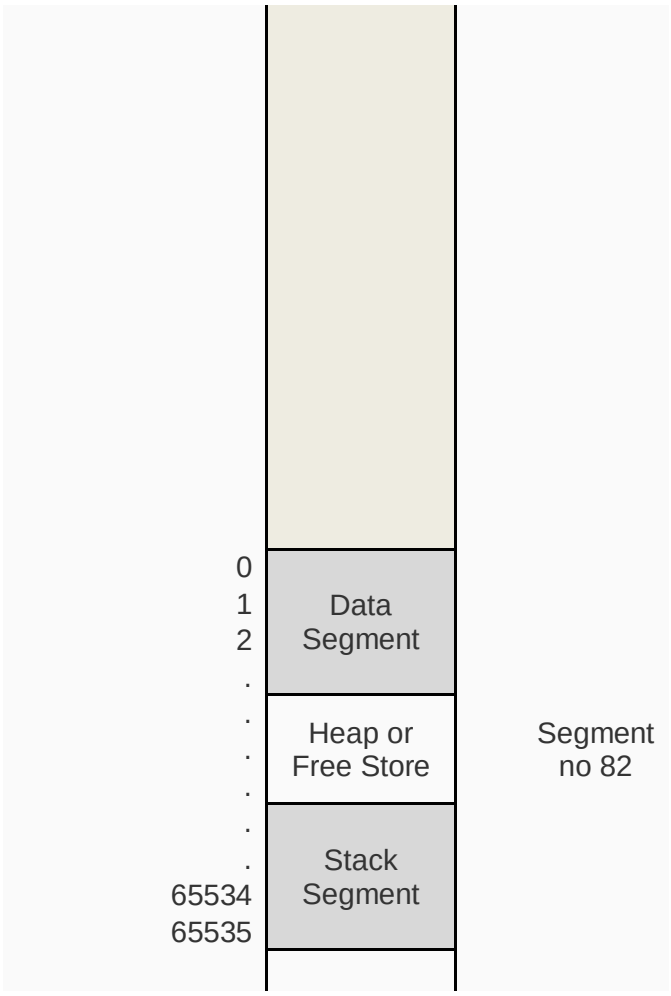
Since a pointer variable stores address of another variable in memory, we must understand the way C++ organizes memory for its program. We can view memory as linear sequence of bytes where every byte has a unique address. For example, if you have 64 Kbytes memory i.e. 64x1024=65536 Bytes, then the address number can be any value between 0 and 65535 as illustrated in the figure given below.



2	
3	
4	
.	.
.	.
.	.
65533	
65534	
65535	

An executable program generated after by C++ compiler is divided in three different segments. These segments are Code Segment (holds instruction of the program), Data Segment (Hold global and static variables of the program) and Stack Segment (holds local variables and return addresses used in the program). Code Segment may have one block of 64KB or multiple blocks of 64KB each depending on memory model used in the program compilation. Data Segment and stack segment shares common block of one block of 64KB or multiple blocks of 64KB each depending on memory model used in program compilation. The free space between Data Segment and Code Segment is known as **Memory Heap** or **Free Store** which is used in Dynamic(Run Time) memory allocation.





Pointers and Arrays in C++

The concept of array is very much bound to the one of pointer. In fact, the identifier of an array is equivalent to the address of its first element, as a pointer is equivalent to the address of the first element that it points to, so in fact they are the same concept. For example, supposing these two declarations:

int b[20];
int * p;

The following assignment operation would be valid:

p = b;

After that, p and b would be equivalent and would have the same properties. The only difference is that we could change the value of pointer p by another one, whereas b will always point to the first of the 20 elements of type int with which it was defined. Therefore, unlike p, which is an ordinary pointer, b is an array, and an array can be considered a constant pointer. Therefore, the following assignment statement would not be valid:

b= p;

Because b is an array, so it operates as a constant pointer, and we cannot assign values to constants. Due to the characteristics of variables, all expressions that include pointers in the following example are perfectly valid:

```
#include <iostream.h>
int main ()
{int b[5];
int * p;
p = b;
p[0] = 10; // p[0] and *p refers to the same value i.e. b[0]
p++;
*p = 20;
p = &b[2];
*p = 30;
p = b + 3;
*p = 40;
p = b;
*(p+4) = 50;     //*(p+4) and p[4] refers to the same value i.e. b[4]
p=b;
for (int n=0; n<5; n++)
        cout << p[n] << ", ";
cout<<endl;
```

```
for (int n=0; n<5; n++)
        cout << *(b+n) << ", ";
return 0;
}
```

Output is

10, 20, 30, 40, 50,
10, 20, 30, 40, 50,

Note: A pointer variable can be used as an array as in above example both *p and p[0] refers to the same variable similarly b[0] and *b refers to the same variable. Again p[1] and *(p+1) are same.

Address calculation method is same for both pointer and array. For example

```
int a[5]={2,4,6,7,9};
int *q=a;
```

The address of a[index] is calculated as

Base address of a (i.e. address of a[0]) + index * sizeof (data type of 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] both will give the same value i.e. 7

Similarly address of (q+3) is calculated as

Address stored in q + 3 * sizeof (data type of pointer belongs to in this case int)
(q+3) = 1000 + 3 * sizeof (int) = 1000 + 3 * 2=1006

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 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

We cannot assign any integer value other than 0 to a pointer variable.

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
```

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.

<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 can not 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)<<sizeof(b[0])<<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.

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.

6000	7000	8000	9000
7000	8000	9000	5

- c has type int*** and a value of 7000
- *c has type int ** and a value of 8000
- **c has type int * and a value of 9000
- ***c has type int and a value of 5

Pointer to constant and constant pointer

```
int x=5, z=7 ;
const int y=8;
const int *p; // here p is pointer to const int
int * const q=&x; // here q is constatnt pointer to int
int * const r; //error constant pointer must be initialized
p=&x; // valid since int * can be converted to const int *
*p=3; // error cannot modify const object
*q=3; //valid
q=&z; //error
int * const s=&y;// error  cannot convert const int * to int *
const int * const b= &x; //legal
const int * const c=&y; //legal
*b=2;//error cannot modify const object
 b=c; // error cannot modify const object
```

Static memory allocation and Dynamic 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.
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 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
```

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
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 function1()
{
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.

Pointer to structure

```
struct emp
{
int empno;
char name[20];
float sal;
};
void main()
{
emp e1={5,”Ankit Singh”, 30000.0};
emp *p=&e1;
cout<<p->empno<<”\t”<<p->name<<”\t”<<p->sal<<endl;
p->sal=p->sal + p->sal*3.0/100;
cout<<(*p).empno<<”\t”<<p->name<<”\t”<<(*p).sal;
}
```

Output is

```
5       Ankit Singh       30000.0
5       Ankit Singh       30900.0
```

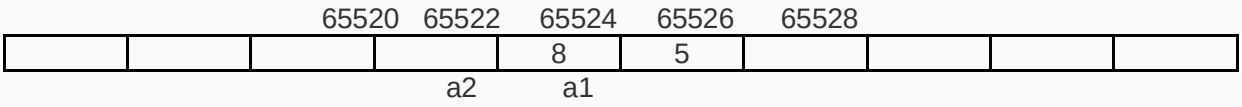
this pointer
When an object of a class invokes a member function then a special pointer is implicitly passed to the member function which contains the address of the object by which the function is being called, this special pointer is known as ***this*** pointer. For example

```
#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 i.e. value of
 }
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 : 65524 //assuming that a1 is allocated at memory location 1000
5 5
address of the object is : 65522 //assuming that a2 is allocated at memory location 998
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.

2 Marks Questions

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

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

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

```
void print (char * p )
{int i=0;
while(*p)
        *p=*p++ + i++;
cout<<"value is "<<p-i+2<<endl;
}
void main( )
{char * x = "Mumbai";
print(x);
cout<<"new value is "<<x<<endl;
}
```

6. . Identify the errors if any. Also give the reason for errors.

```
#include<iostream.h>
void main()
{int b=4;
const int i =20;
const int * ptr=&i;
(*ptr)++;
ptr =&j;
cout<<*ptr;
}
```

7. Identify errors on the following code segment

```
void main()
{float c[ ] ={ 1.2,2.2,3.2,56.2};
float *k,*g;
k=c;
g=k+4;
k=k*2;
```

```
g=g/2;
k=k+g;
c=k;
cout<<"*k="<<*k<<"*g="<<*g;
}
```

8. 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- -;
}
```

9. 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; // it is simply adding the addresses.
*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;
```