CHAPTER 08

POINTERS

What is Pointer?

Pointer is a variable that holds a memory address, usually location of another variable.

The Pointers are one of the C++’s most useful and powerful features.

How Pointers are one of the C++'s most useful and powerful features?

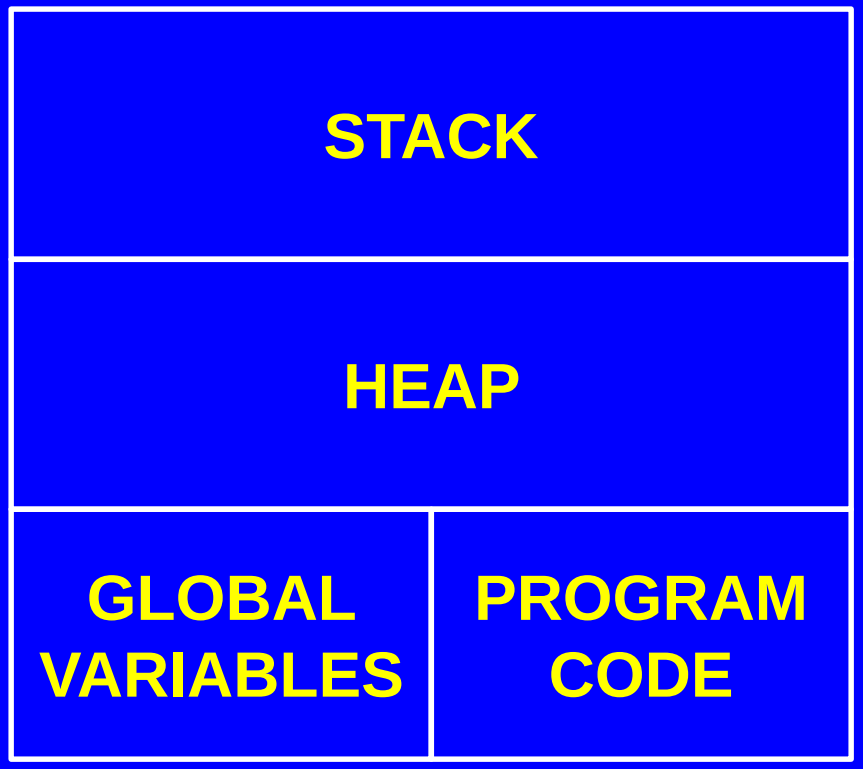
First : Pointer provide the means through which the memory location of variable can be directly accessed and hence it can be manipulated in the way as required.

How Pointers are one of the C++'s most useful and powerful features?

Second: Pointer supports C++ dynamic allocation routines.

Third: Pointer can improve the efficiency of certain routines.

C++ MEMORY MAP



STACK – This area is used for function calls return address, argument and local variables

HEAP – This area is used for Dynamic Memory Allocation

GLOBAL VARIABLES – This area is used to store global variables where they exists as long as the program continues

DYNAMIC AND STATIC MEMORY ALLOCATION

The Golden Rule of computer states that anything or everything that needs to be processed must be loaded into internal memory before its processing takes place. Therefore the main memory is allocated in two ways,

- ✓**STATIC MEMORY ALLOCATION**
- ✓**DYNAMIC MEMORY ALLOCATION**

STATIC MEMORY ALLOCATION

In this technique the demanded memory is allocated in advance that is at the compilation time is called static memory allocation.

For example,

```
int  s;
```

The compiler allocates the 2 bytes of memory before its execution.

DYNAMIC MEMORY ALLOCATION

In this technique the memory is allocated as and when required during run time (program execution) is called Dynamic memory allocation.

C++ offers two types of operators called **new** and **delete** to allocate and de-allocate the memory at runtime.

FREE STORE

FREE STORE is a pool of unallocated heap memory given to a program that is used by the program for dynamic allocation during execution.

DECLARATION AND INITIALIZATION OF POINTERS

Pointer variables are declared like normal variables except for the addition of unary operator * (character)

The General Format of Pointer variable declaration is ,

type * var_name;

where ,

type refers to any C++ valid data type and var_name is any valid C++ variable

DECLARATION AND INITIALIZATION OF POINTERS

For example,

int *iptr; *// integer pointer variable points to //another integer variable's location.*

float *fptr; *// float type pointer variable points to //another float type variable's location.*

char *cptr; *// character type pointer variable //points to another char type variable's location.*

Two Special Operators

Two special operators are used * and & are with pointers.

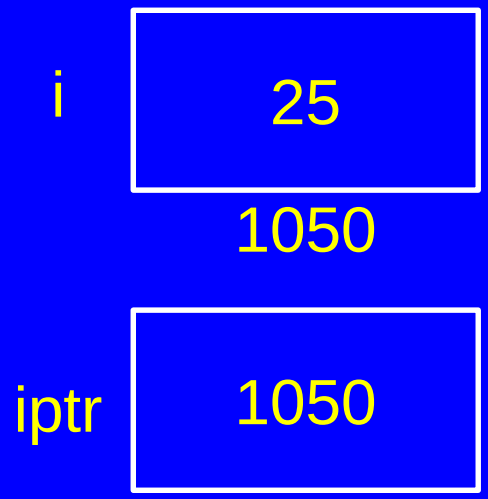
& operator is called as **address of** operator which provides the address of operand.

***** Operator is called as **at address** and also called as **differencing** which provides the value being pointed by pointer.

Two Special Operators

For example,

```
int  i = 25;
int  *iptr;
iptr= &i;
```



NULL Pointer (ZERO POINTER)

A pointer variable must not remain uninitilized since uninitialized pointer cause system crash. Even if you do not have legal pointer value to initialize a pointer, you can initialize it with NULL pointer (ZERO POINTER).

```
int *iptr=NULL;
```

Note: In C++ version 11 nullptr keyword is introduced to assign null. nullptr is legal empty null pointer.

POINTER AIRTHMETIC

Only two arithmetic operators, addition and subtraction may be performed on pointers.
When you add 1 to a pointer, you are actually adding the size of what ever the pointer pointing at.

For example,
iptr++;
iptr--;

POINTER AIRTHMETIC

Only two arithmetic operators, addition and subtraction may be performed on pointers.
When you add 1 to a pointer, you are actually adding the size of what ever the pointer pointing at.

For example,

```
iptr++;
iptr--;
```

Note: In pointer arithmetic all pointers increase and decrease by the length of the data type they point to.

DYNAMIC ALLOCATION OPERATORS - new OPERATOR

C++ offers two types of operators called **new** and **delete** to allocate and de-allocate the memory at runtime. Since these two operators operate upon free store memory, they are also called as **free store operators**

Syntax :

pointer_variable = **new** data type;

where pointer_variable pointer variable and datatype is valid C++ datatype and new is operator which allocates the memory (size of data type) at run time.

DYNAMIC ALLOCATION OPERATORS - new OPERATOR

For example,

```
int *iptr=new int;
char *cptr=new char;
float *fptr=new float;
```

Once a pointer points to newly allocated memory, data values can be stored there using **at address** operator

```
*cptr= 'a' ;
*fptr=1.34;
*iptr=89;
```

DYNAMIC ALLOCATION OPERATORS - new OPERATOR

Initializing values at the time of declaration one can rewrite like,

```
int *iptr=new int(89);
char *cptr=new char( ‘a’ );
float *fptr=new float(1.34);
```

DYNAMIC ALLOCATION OPERATORS

CREATING DYNMIC ARRAYS 1D ARRAYS:

Syntax :

pointer_variable = **new** data type[size];

```
int *value=new int[10];
```

It will create memory space from the free store to store 10 integers. Initilization while declaring dynamic array is not possible but it is included in C++ compiler ver 11.

DYNAMIC ALLOCATION OPERATORS - new OPERATOR

CREATING DYNMIC ARRAYS (2D ARRAYS):

One needs to be tricky while defining 2D array as,

```
int *val,r,c,i,j;
cout<<"Enter dimensions";
cin>>r>>c;
val=new int [ r * c ] ;
To read the elements of the 2D array
for(i=0;i<r;i++)
{
for(j=0;j<c;j++)
cin>> val [ i *c + j ] ;
}
```

DYNAMIC ALLOCATION OPERATORS - delete OPERATOR

The life time of the variable or object created by new is not restricted to the scope in which it is created. It lives in the memory until explicitly deleted using the delete operator.

Syntax:

delete pointer_variable;

For example, delete iptr;

FOR ARRAYS :

Syntax :

delete [ ] arraypointer_variable;

delete [ ] val;

DYNAMIC ALLOCATION OPERATORS – MEMORY LEAK

Improper use of new and delete may lead to memory leaks. Make sure that the memory allocated through new must be deleted through delete.

Otherwise this may lead to adverse effect on the system

ARRAYS

It is faster to use an element pointer than an index when scanning arrays,

For 1D Array,

S[0]= *(S + 0) or *S or *(S)
S[1] = *(S + 1)
S[2] = *(S + 2)

FOR 2D Arrays,

S[0][0] = *(S[0] + 0) or *(*(S+0))
S[0][1] = *(S[0] + 1) or *(*(S+1))

S[1][2] = *(S[1] + 2) or *(*(S+1)+2)

POINTERS AND STRINGS

A string is a one dimensional array of characters terminated by a null '¥0'. You have learnt in 11 std a string can be defined and processed as,

```
char   name [ ] = "POINTER";
for ( I = 0 ;  name [ i ] ! = ' ¥ 0 ' ;    i + + )
cout<<name[i];
```

Alternatively the same can be achieved by writing,

```
char   name [ ] = "POINTER";
char  *cp;
for ( cp = name ;  * cp ! = ' ¥ 0 ' ;    cp + + )
cout<< *cp ;
```

POINTERS AND STRINGS

Another Example,

```
char  *names [  ] = { “Sachin”, “Kapil”, “Ajay”, “Sunil”,  “Anil”
};
char  *t;
     t=name[1] ; //pointing to string “Kapil”
     cout<<*t;  // will produce out put “Kapil”
```

Similarly

T=name[3]; will point to?

POINTERS AND CONST

Constant pointer mean that the pointer in consideration will always point to the same address . Its address (to which it is pointing to) can not be modified.
For example,

```
int n=44;

int *const ptr = &n;

++(*ptr); // allowed since it modifies the content.

++ptr; // illegal because pointer ptr is constant pointer
```

CBSE QUESTION PAPER
QNO 1 (d) – 3 Marks
QNO 1 (e) – 3 Marks

1(d) What will be the output of the following program : 3

Delhi 2004

```
#include<iostream.h>
#include<ctype.h>
#include<conio.h>
#include<string.h>
void ChangeString(char Text[], int &Counter)
{
char *Ptr = Text;
int Length = strlen (Text);
for ( ;Counter<Length-2; Counter+=2, Ptr++)
{
* (Ptr + Counter) = toupper( * (Ptr + Counter) );
}
}
void main()
{
clrscr();
int Position = 0;
char Messaget[] = “Pointers Fun”;
ChangeString (Message, Position);
cout<<Message<<“ @ “<<Position;
}
```