POINTERS IN C++

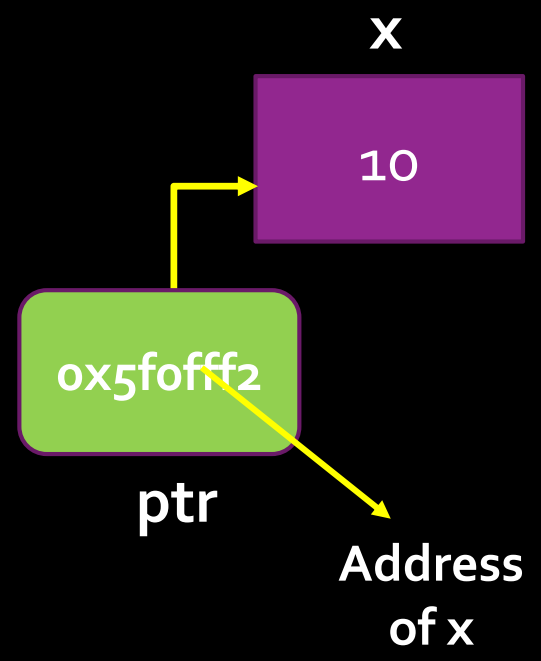
WHAT IS A POINTER?

- A pointer is a variable that holds the memory address of another variable of same type.
- This memory address is the location of another variable where it has been stored in the memory.
- It supports dynamic memory allocation routines.

DECLARATION AND INITIALIZATION OF POINTERS

Syntax : Datatype *variable_name;
eg. **int *x; float *y; char *z;**

Address of operator(&)- it is a unary operator that returns the memory address of its operand. Here the operand is a normal variable.



eg. **int x = 10;**
int *ptr = &x;
Now ptr will contain address where the variable x is stored in memory.

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.

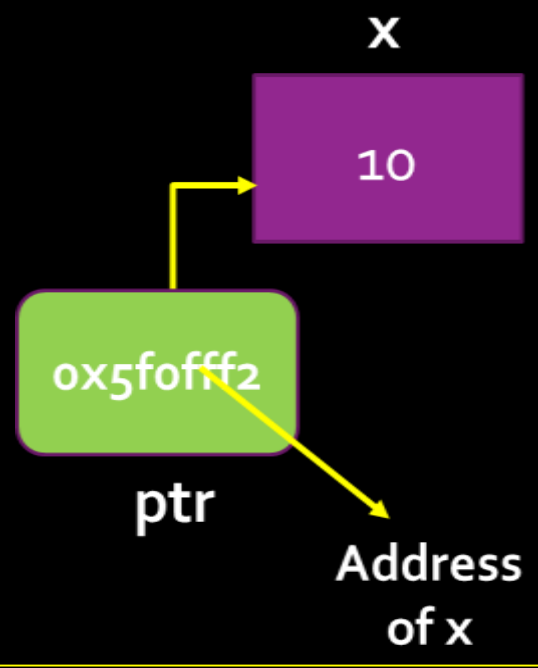
DEREFERENCE OPERATOR (*)

- It is a unary operator that returns the value stored at the address pointed to by the pointer.
- Here the operand is a pointer variable.

eg.

```
int x = 10;
int *ptr = &x;
cout<< ptr;// address stored at ptr will be displayed
cout<<*ptr;// value pointed to by ptr will be displayed
```

Now ptr can also be used to change/display the value of **x**.



OUTPUT

0x5fofff2
10

POINTER ARITHMETIC

Two arithmetic operations, addition and subtraction, may be performed on pointers. When we add 1 to a pointer, we are actually adding the size of data type in bytes, the pointer is pointing at.

For e.g. **int *x; x++;**

If current address of x is 1000, then x++ statement will increase x by 2(size of int data type) and makes it 1002, not 1001.

POINTER ARITHMETIC Contd..

***ptr++**	***(ptr++)**	**Increments pointer, and dereferences unincremented address i.e. This command first gives the value stored at the address contained in ptr and then it increments the address stored in ptr.**
***++ptr**	*(++ptr)	Increment pointer, and dereference incremented address i.e. This command increments the address stored in ptr and then displays the value stored at the incremented address.
++*ptr	++(*ptr)	Dereference pointer, and increment the value it points to i.e. This command first gives the value stored at the address contained in ptr and then it increments the value by 1.
(*ptr)++		Dereference pointer, and post-increment the value it points to i.e. This command first gives the value stored at the address contained in ptr and then it post increments the value by 1.

POINTERS AND ARRAYS

C++ treats the name of an array as constant pointer which contains base address i.e. address of first location of array.

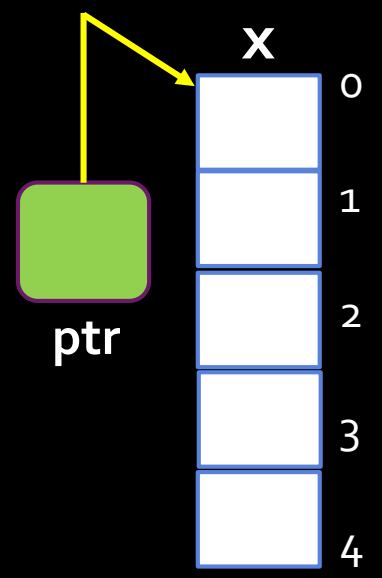
For eg.

```
int x[10];
```

Here x is a constant pointer which contains the base address of the array x.

POINTERS AND ARRAYS Contd.....

We can also store the base address of the array in a pointer variable. It can be used to access elements of array, because array is a continuous block of same memory locations.



For eg.

```
int x[5];
int * ptr=x; // ptr will contain the base address of x
```

we can also write

```
int * ptr= &x[0]; //ptr will contain the base address of x
```

POINTERS AND ARRAYS Contd.....

NOTE: In the previous example

ptr++; // valid

x++; //invalid

This happens because x is a constant pointer and the address stored in it can not be changed. But ptr is a not a constant pointer, thus the above statement will make ptr point to second element of the array.

The contents of array x can be displayed in the following ways:

for(int i=0;i<10;i++) cout<<*(ptr+i);	for(int i=0;i<10;i++) cout<<*ptr++;	for(int i=0;i<10;i++) cout<<*(x+i);

POINTERS AND STRINGS

We can also handle the character array using pointers. Consider the following program:

```
void main()
{ char str[] = "computer";
char *cp=str;
cout<<str; // using variable name
cout << cp; // using pointer variable
}
```

Here cp stores the address of str. The statement cout<<cp; will print computer as giving an address of a character with cout results in printing everything from that character to the first null character that follows it.

ARRAY OF POINTERS

- Like any other array, we can also have an array of pointers.
- A common use of array of pointers is an array of pointers to strings.

ARRAY OF CHARACTER POINTERS

An array of character pointers is used for storing several strings in memory. For e.g.

```
char * vehicle[ ]={"CAR","VAN","CYCLE",
      "TRUCK","BUS"};
 for(int i=0;i<5;i++)
cout<<vehicle[i]<<endl;
```

In the above example, the array vehicle[] is an array of char pointers. Thus vehicle[0] contains the base address of the string "Sunday", vehicle[1] contains the base address of the string "Monday" and so on.

POINTERS AND CONSTANTS

- **Constant Pointer-** It means that the address stored in the pointer is constant i.e. the address stored in it cannot be changed. It will always point to the same address. The value stored at this address can be changed.

- **Pointer to a Constant-** It means that the pointer is pointing to a constant i.e. the address stored in the pointer can be changed but the pointer will always point to a constant value.

POINTERS AND CONSTANTS Contd…

For e.g.

Case 1:

```
int x=10, y=20;
 int * p1=&x;  //non-const pointer to non-const int
 *p1=20;//valid i.e. value can be changed
 p1=&y;  //valid i.e. address in p1 can be changed. Now it will point to y.
```

Case 2:

```
 const int x=10;
 int y=20;
 const  int * p2=&x; // non-const pointer to const int
*p2=50; //invalid i.e. value can not be changed
  p2=&y; // valid i.e. address stored can be changed
 *p2=100;// invalid  as p2 is pointing to a constant integer
```

POINTERS AND CONSTANTS Contd…

Case 3:

```
 int x=10,y=20;
int * const  p3= &x; //const pointer to non-const int
*p3=60; //valid i.e. value can be changed
p3=&y; //invalid as it is a constant pointer, thus address can not be changed
```

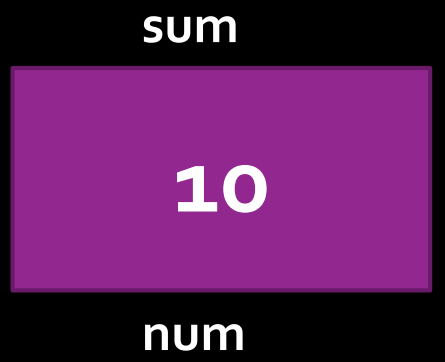
Case 4:

```
int x=10,y=20;
const int * const  p4=&x; // const pointer to const int
 p4=&y;// invalid
 *p4=90;// invalid
```

REFERENCE VARIABLE

A reference variable is a name that acts as an alias or an alternative name, for an already existing variable.

SYNTAX:
Data type & variable name = already existing variable;



EXAMPLE:

```
int num=10;
int & sum = num; // sum is a reference variable or alias name for num
```

NOTE: Both num and sum refer to the same memory location. Any changes made to the sum will also be reflected in num.

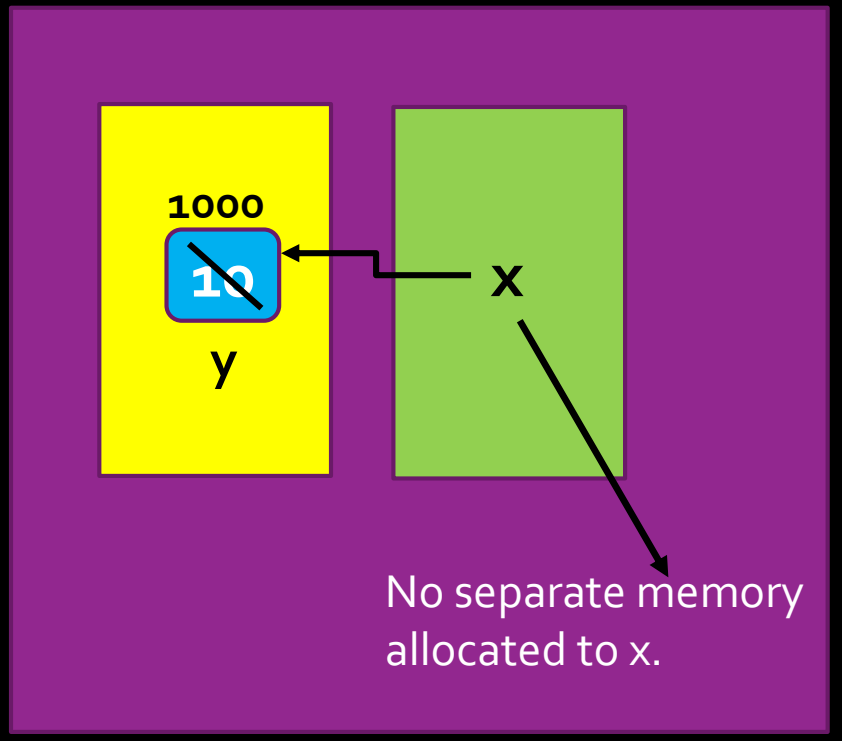
USAGE OF REFERENCE VARIABLES

- This method helps in returning more than one value from the function back to the calling program.
- When dealing with large objects reference arguments speed up a program because instead of passing an entire large object, only reference needs to be passed.

REFERENCE AS FUNCTION ARGUMENTS

Consider a program:

```
void cube(int &x)
{ x= x*x*x; }
void main()
{int y=10;
cout<<y<<endl;
cube(y);
cout<<y<<endl;}
```



OUTPUT:
10
1000

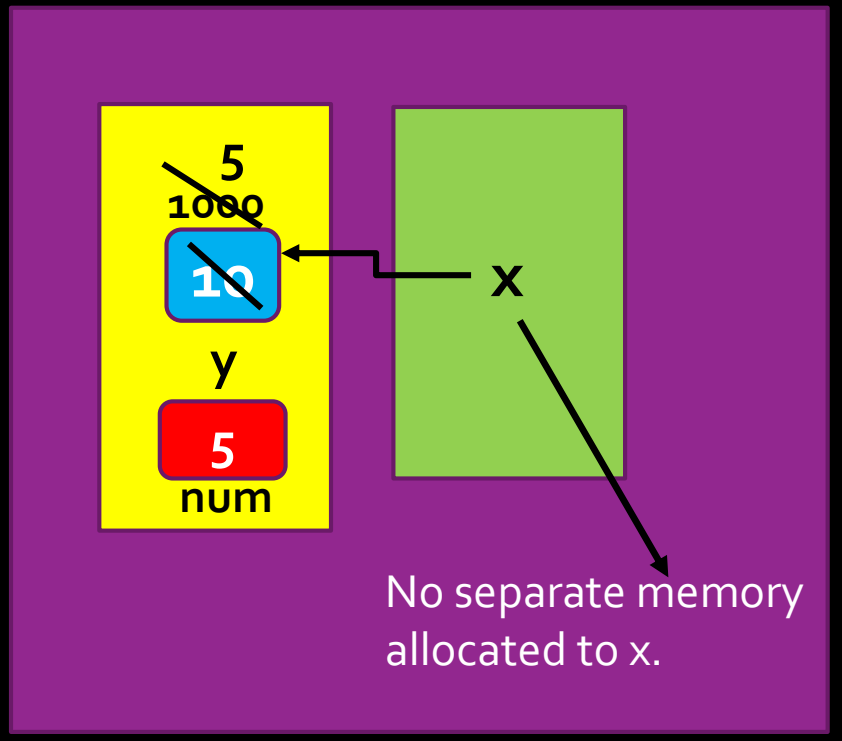
In the above program reference of y is passed. No separate memory is allocated to x and it will share the same memory as y. Thus changes made to x will also be reflected in y.

RETURN BY REFERENCE

A function can also return a reference. In this case function call can appear on the left hand side of an assignment statement. Consider a program:

```
void &cube(int &x)
{ x= x*x*x; return x;
}
void main()
{ int y=10, num=5;
cube(y) =num;
cout<<y<<endl;
cout<<num<<endl;}
```

OUTPUT:
5
5



In the function call statement firstly the function call is evaluated and value of x is changed to 1000. Then the function returns the reference of x which is nothing but y. Thus function call becomes equivalent to **y=num;** . As a result value of y becomes equal to 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, in the called function whatever changes we make in the formal arguments, the actual arguments also get changed.
- This is because formal arguments contain the address of actual arguments and point to the memory location where actual arguments are stored.

PASS BY POINTERS

Consider the program of swapping two variables with the help of pointers:

```
#include<iostream.h>
void swap(int *m, int *n)
{ int temp; temp = *m; *m = *n; *n = temp; }
void main()
{ int a = 5, b = 6;
cout << "\n Value of a :" << a << " and b :" << b;
swap(&a, &b); //we are passing address of a and b
cout << "\n After swapping value of a :" << a <<
"and b :" << b;
}
```

OUTPUT :

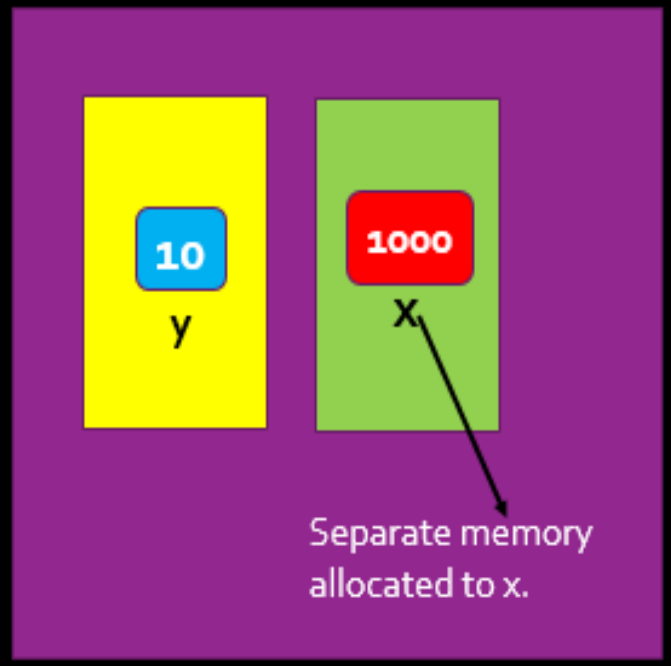
Value of a : 5 and b : 6
After swapping value of a : 6 and b : 5

COMPARING PASS BY VALUE, PASS BY REFERENCE AND PASS BY POINTER

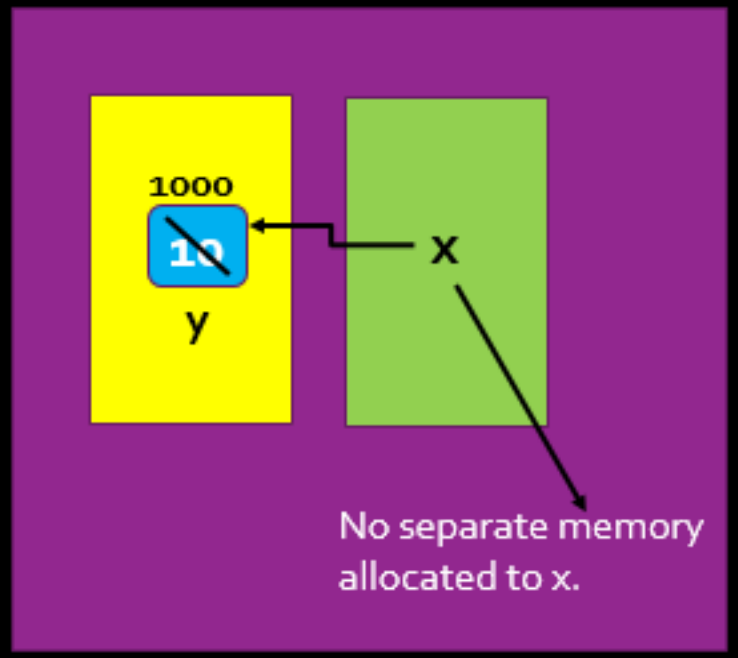
PASS BY VALUE	PASS BY REFERENCE	PASS BY POINTER
Separate memory is allocated to formal parameters.	Formal and actual parameters share the same memory space.	Formal parameters contain the address of actual parameters.
Changes done in formal parameters are not reflected in actual parameters.	Changes done in formal parameters are reflected in actual parameters.	Changes done in formal parameters are reflected in actual parameters.
For eg. **void cube(int x)** **{ x= x*x*x; }** **void main()** **{int y=10;** **cout<<y<<endl;** **cube(y); cout<<y<<endl;}** output: 1010	For eg. **void cube(int &x)** **{ x= x*x*x; }** **void main()** **{int y=10;** **cout<<y<<endl;** **cube(y); cout<<y<<endl;}** output: 101000	For eg. **void cube(int *x)** **{ *x= (*x)*(*x)*(*x); }** **void main()** **{int y=10;** **cout<<y<<endl;** **cube(&y); cout<<y<<endl;}** output: 101000

COMPARING PASS BY VALUE, PASS BY REFERENCE AND PASS BY POINTER Contd…

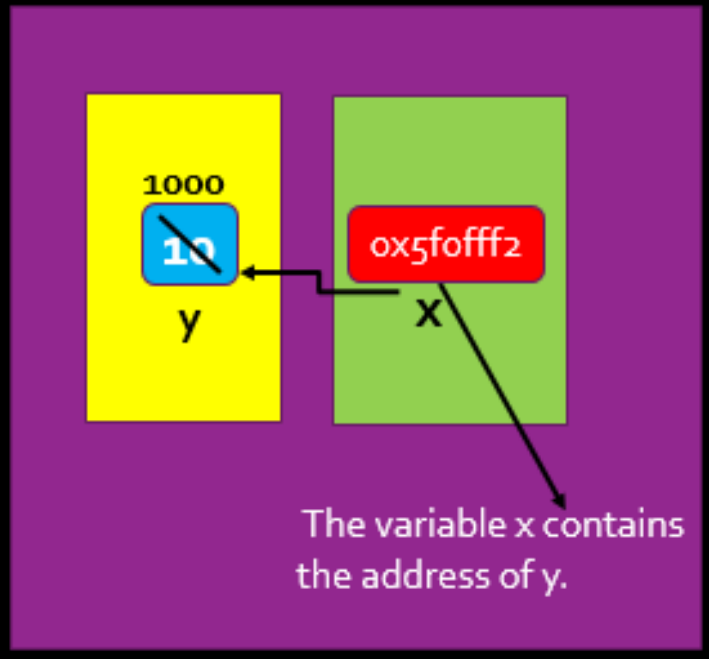
MEMORY DIAGRAM



PASS BY VALUE



PASS BY REFRENCE



PASS BY POINTER

FUNCTION RETURNING POINTERS

The general form of prototype of a function returning a pointer is:

Data Type * function-name (argument list);

Q WAP to find the minimum of two numbers using the pointers concept.

```
#include<iostream.h>
 int *min(int &, int &)
{ if (x < y ) return (&x); else return (&y); }
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; }
```

DYNAMIC MEMORY ALLOCATION OPERATORS

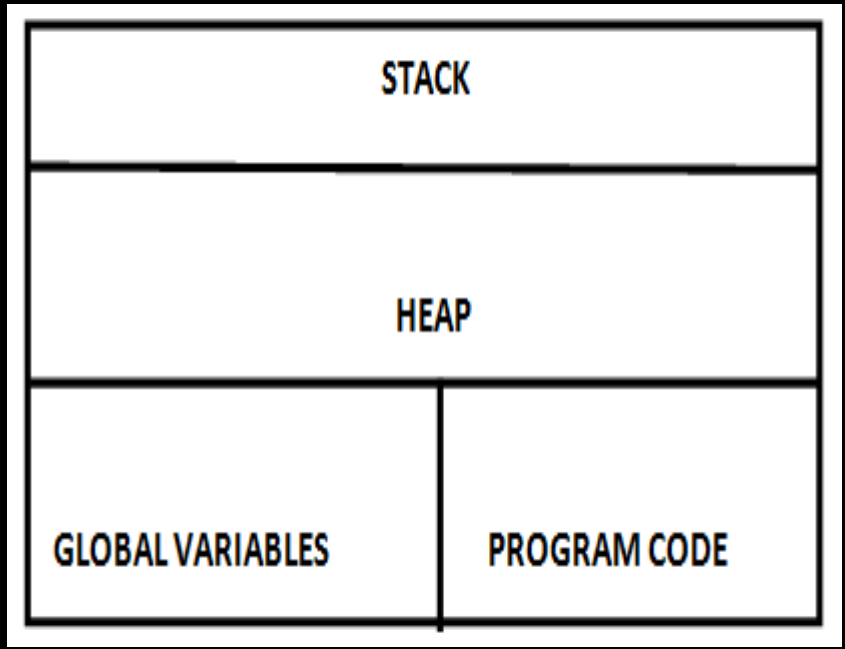
C++ dynamically allocates memory from the free store/heap/pool, the pool of unallocated heap memory provided to the program.

There are two unary operators new and delete that perform the task of allocating and deallocating memory during runtime.

C++ MEMORY MAP

C++ memory is divided into four parts which are listed as follows:

- 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 functions.



NEW OPERATOR

New operator is used to allocate memory of size equal to its operand and returns the address to the beginning of the new block of memory allocated. For eg.

```
int * I =new int;
```

In the above example new operator allocates memory of size two bytes(size of int) at run time and returns the address to the beginning of the new block of memory allocated. This address is stored in the pointer variable I.

The memory allocated using new operator can also be initialized at the time of allocation itself. This can be done as follows:

```
char * ptr=new char('A');
```

This statement will allocate one byte of memory from free store, stores the value 'A' in it and makes the pointer ptr points to it.

DELETE OPERATOR

Delete operator is used to deallocate memory which was allocated using new. For eg.

delete I;

The above command will deallocate the memory pointed to by the pointer variable I.

STATIC MEMORY ALLOCATION vs DYNAMIC MEMORY ALLOCATION

STATIC MEMORY ALLOCATION	DYNAMIC MEMORY ALLOCATION
The amount of memory to be allocated is known before hand.	The amount of memory to be allocated is not known before hand. It is allocated depending upon the requirements.
Memory allocation is done during compilation.	Memory allocation is done during run time.
For eg. int i; **This command will allocate two bytes of memory and name it 'I'.**	Dynamic memory is allocated using the new operator. For eg. int*k=new int; In the above command new will allocate two bytes of memory and return the beginning address of it which is stored in the pointer variable k.
The memory is deallocated automatically as soon as the variable goes out of scope.	To deallocate this type of memory delete operator is used. For eg. delete k;

POINTERS TO STRUCTURES

The general syntax for creating pointers to structures is:

struct-name * struct-pointer;

For eg.

```
struct student
{ int rollno;
char name[20];};
void main()
{students s1;
cin>> s1.rollno;gets(s1.name);
student *stu;
stu=&s1; //now stu points to s1 i.e. the address of
                    // s1 is stored in stu
 cout<<stu->rollno<<stu->name;}
```

NOTE:

- Here s1 is a variable to the structure student and stu is a pointer variable to the structure student.
- The data members of structure can be accessed by pointer to structure using the symbol -> .

POINTERS TO OBJECTS

The general syntax for creating pointers to objects is:

class-name * object-pointer;

For eg.

```
class student
{ int rollno;
char name[20];
void indata()  {cin>> s1.rollno;gets(s1.name); }
void showdata() {cout<<stu.rollno<<stu.name;} };
void main()
{ student s1 , *stu;
s1.indata();
stu=&s1; //now stu points to s1
s1.outdata(); stu->outdata();}
```

NOTE:

- Here s1 is an object of the class student and stu is a pointer variable to the class student.
- The pointer stu can access the members of class by using the symbol -> .

SELF REFERENTIAL STRUCTURES

A self referential structure is a structure which contains an element that points/refers to the structure itself.

For eg.

```
struct student
{ int rollno;
char name[20];
student * link}; //pointer of type structure itself
```

The self referential structures are used for creating a node in a linked list.

DYNAMIC STRUCTURES

The new operator can be used to create dynamic structures. The syntax is:

struct-pointer = new struct-type;

For eg.

```
struct student
{ int rollno;
char name[20];};
void main()
{ student *stu;
stu = new student;
stu->rollno=1;
strcpy(stu->name,"Ramesh");
cout<<stu->roll<<stu->name;}
```

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

delete stu;

DYNAMIC ARRAY

The new operator can be used to create dynamic arrays. The syntax is:

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

For e.g.

int * array = new int[10];

Now array[0] will refer to the first element of array, array[1] will refer to the second element.

Similarly we can create a one dimensional dynamic character array using pointers

For e.g.

char * name=new char[20];

TWO DIMENSIONAL ARRAY USING POINTERS

The new operator can be used to create two dimensional arrays. For e.g.

```
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]; }
```

For dynamic arrays memory can be released with delete as below:

delete [size] pointer variable;

Eg. **delete [ ] arr;**

STATIC ARRAY vs DYNAMIC ARRAY

Static Array	Dynamic Array
It is created in stack area of memory	It is created in heap area of memory
The size of the array is fixed.	The size of the array is decided during run time.
Memory allocation is done during compilation time.	Memory allocation is done during run time.
They remain in the memory as long as their scope is not over.	They need to be deallocated using delete operator.

MEMORY LEAK

- A memory leak occurs when a piece (or pieces) of memory that was previously allocated by a programmer is not properly deallocated by the programmer.
- Even though that memory is no longer in use by the program, it is still "reserved", and that piece of memory cannot be used by the program until it is properly deallocated by the programmer.
- That's why it's called a memory *leak*– because it's like a leaky faucet in which water is being wasted, only in this case it's computer memory.

MEMORY LEAK Contd…

For e.g.

```
void main()
{
int * ptr = new int;
 *ptr=100;
}
```

The above code resulted in memory leak because the memory reserved using new operator is not freed. This memory is never deallocated and is wasted.