POINTERS IN C++

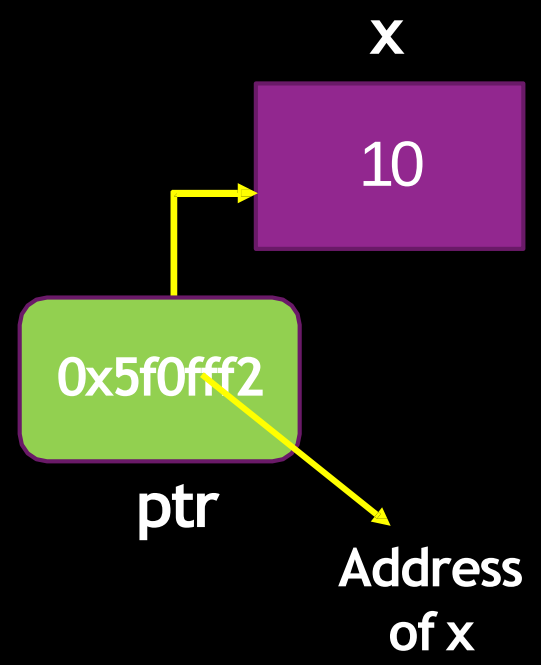
WHAT IS A POINTER?

- 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.

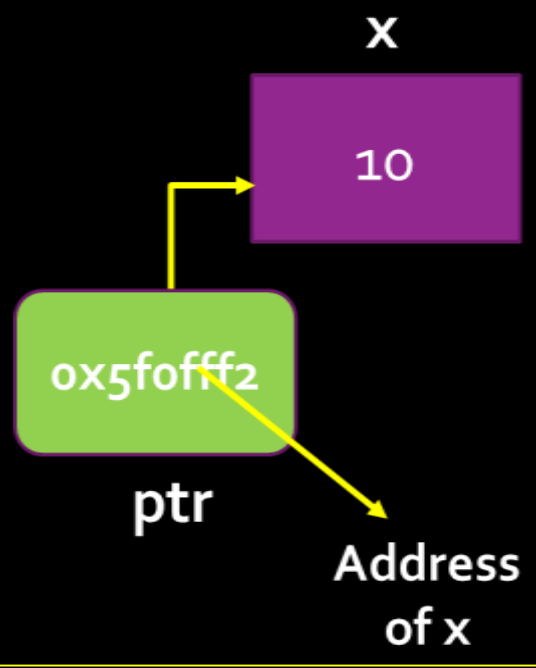
DEREFERENCE OPERATOR (*)

- 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

0x5f0fff2
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 Operators(Summary)

*	dereference operator, indirection operator	This is used to declare a variable as a pointer. It is also used when you want to access the value pointed to by the pointer variable.
&	reference operator, address-of operator	Use before a variable to indicate that you mean the ***address of*** that variable. You'll often see this in a function header where the parameter list is given.
->	member selection operator	This is used to refer to ***members of structures***

Arrays

Introduction

Array is a collection of variables that can hold value of same type and reference by common name .Its is a derived data Structure

Array are always stored in a continuous memory locations.

An Array either be a integer ,character, or float base type (Data type of array) .

Array indexing is always start from zero and highest address corresponds to last element

Int num [6]

Base type of arrray

Name of array

Size of array

Continuous memory allocation of array

Need of Arrray

To store processed large number of variables of same data type and reference/name

Easy understanding of program

Example:

Types of Array

* One dimensional
* Two dimensional
* Multi dimensional

1-D Array

A one dimensional array is one in which one subscript /indices specification is needed to specify a particular element of array

Declaration :

Data_type array_name [size of array];

Eg:

Int num[10];

Memory representation:

Total memory in bytes that an array is occupied :

Size of array=size of array*size of(base type)

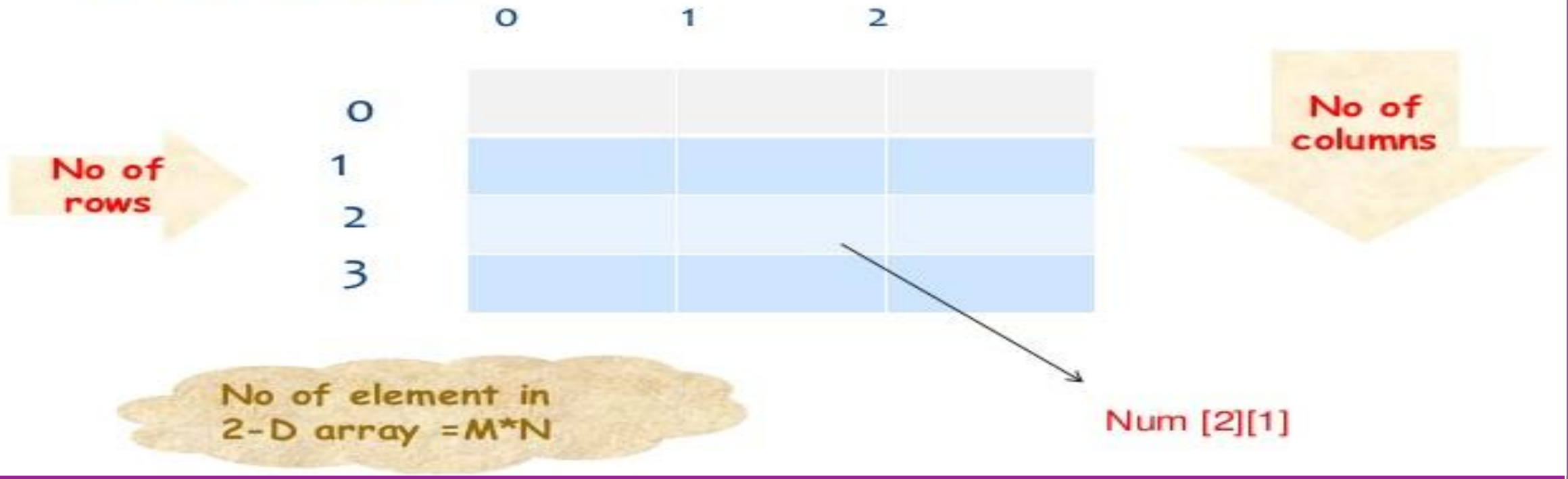
Hear,

10*2=20

2-D Array

A 2-d array is an array in which each element is itself an array

i.e int num[4][3]

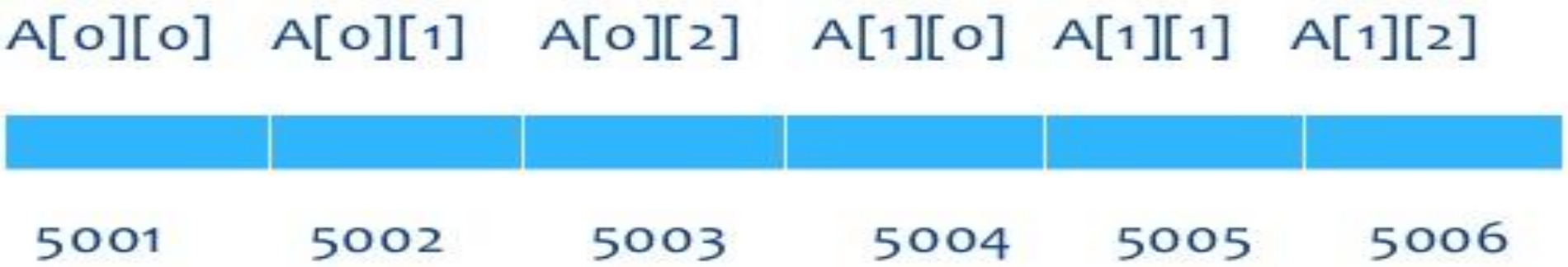


size of 2-D array

Total bytes= no of rows*no of columns*size of(base type)

Memory reprsentation in 2-D array:

char A [2][3]



Multi dimensional array

An array with dimensions more than two .The maximum limit of array is compiler dependent

Declration:

Data_type name [a][b][c][d][e][f].....[n];

Array of 3 or more dimensional are not often use because of huge memory requirement and complexity involved

Array initialization

C++ provides the facility of array initialization at the time of declaration .General form of array initialization is as:

Type array_name[size 1].....[size N] ={vale list};

Eg:

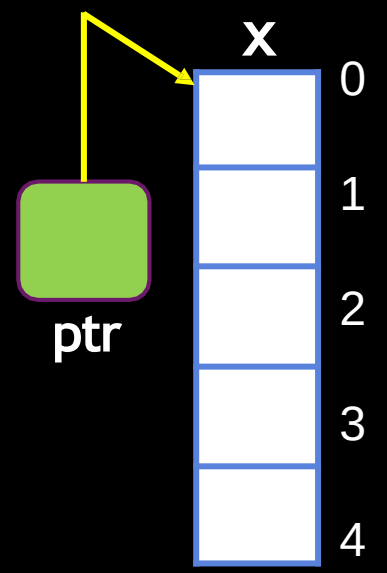
```
Int days_month[12]={31,25,29,03,31,19,20,31,18,20,31,29};

Char string[6]={'a','r','g','y','d','\0'};
```

2-D array are also initialized in same way as linear array

```
Int cube[4][2]={        1,3,
                        4,6,
                        9,37,
                        5,78
                };
```

POINTERS AND ARRAYS

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
```

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 Contd.....

NOTE: In the previousexample

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);

Introduction

- A string is a sequence of characters.
- Strings can contain small and capital letters, numbers and symbols.
- Each element of string occupies a byte in the memory.
- Every string is terminated by a null character('\o').
- Its ASCII and Hex values are zero.

Introduction

- The string is stored in the memory as follows:

```
char country [6] = "INDIA";
       I   N   D   I   A   '\0'
       73 78 68 73 65 00
```

- Each character occupies a single byte in memory as shown above.
- The various operations with strings such as copying, comparing, concatenation, or replacing requires a lot of effort in 'C' programming.
- These string is called as C-style string.

String library functions

Functions	Description
strlen()	Determines length of string.
strcpy()	Copies a string from source to destination
strcmp()	Compares characters of two strings.
stricmp()	Compares two strings.
strlwr()	Converts uppercase characters in strings to lowercase
strupr()	Converts lowercase characters in strings to uppercase
strdup()	Duplicates a string
strchr()	Determines first occurrence of given character in a string
strcat()	Appends source string to destination string
strrev()	Reversing all characters of a string
strspn()	finds up at what length two strings are identical

Program explains above functions

```
#include <iostream>
#include <string>
using namespace std;

int main()
{
        char name[10];
        cout << "Enter name: " << endl;
        cin >> name;
        cout << "Length :" << strlen (name)<< endl;
        cout << "Reverse :" << strrev (name)<< endl;
        return 0;
}
```

Output:

```
Enter name:
ABC
Length :3
Reverse :CBA
```

STRUCTURES

What a Structure is?

- "A structure is a collection of variables under a single name. These variables can be of different types, and each has a name that is used to select it from the structure"
- There is always a requirement in most of our data processing applications that the relevant data should be grouped and handled as a **group**
- In structure, we introduce a new data type

Declaration of a Structure

- **1) Declare Structure**
- **struct** keyword is used for creating a structure.

- **Structure Declaration Ways**
- By struct keyword
- By declaring variable at the time of defining structure.

What a Structure is?

Students

- Name
- Address
- Date of Birth
- CGPA
- Discipline

Steps to Create Structure

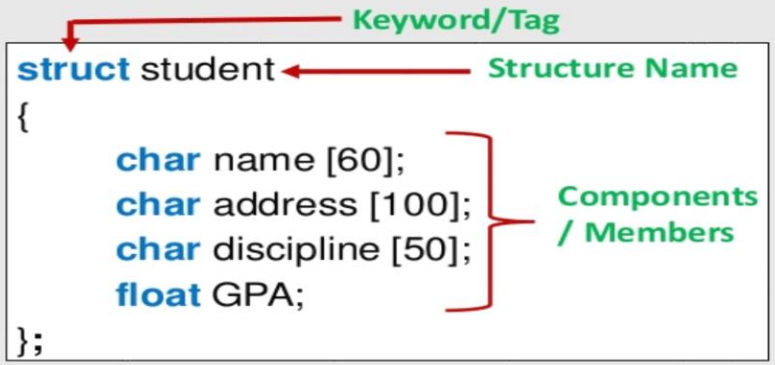
- Declare Structure
- Initialize Members of Structure
- Access Structure Elements

Declaration of a Structure

- 1) Declare Structure
- **struct** keyword is used for creating a structure.

- Structure Declaration Ways
- By struct keyword
- By declaring variable at the time of defining structure.

Approach 1(Declaration of a Structure)



Approach 1(Declaration of a Structure)

```
struct student
{
        char name [60];
        char address [100];
        char discipline [50];
        float GPA;
};
```

Approach 1(Declaration of a Structure)

```
struct student
{
        char name [60];
        char address [100];
        char discipline [50];
        float GPA;
};
```

Declaring Variables of Type struct

```
. . .
struct STUDENT_TYPE
{
    string name, street, city, state, zipcode;
    int age;
    double ID_num;
    double grade;
};

int main()
{
    // declare two variables of the new type
    STUDENT_TYPE student1, student2;

        . . .
}
```


Approach 2(Declaring Variables of Type struct)

```
struct STUDENT_TYPE
{
   string name, street, city, state, zipcode;
   int age;
   double ID_num;
   double grade;
}

student1, student2;
```


Accessing Structure Members

- To access any member of a structure, we use the **member access operator (.)**.
- The member access operator is coded as a period between the structure variable name and the structure member that we wish to access.
- Remember we would use **struct** keyword to define variables of structure type.

Accessing Structure Members

- Suppose, you want to access age of structure variable **student1** and assign it 50 to it. we can perform this task by using following code below:

```
student1.age = 50;
```

- Tasking input as:

```
cin >> student1.age;
```

Example 1

C++ Program to assign data to members of a structure variable and display it.

Solution-1/3

```
#include <iostream>
using namespace std;

struct Person
{
   char name[50];
   int age;
   float salary;
};
```

Declare a Structure of Person

Solution-2/3

```
int main()
{
   Person p1;
   cout << "Enter Full name: ";
   cin.get(p1.name, 50);
   cout << "Enter age: ";
   cin >> p1.age;
   cout << "Enter salary: ";
   cin >> p1.salary;
```

Structure Variable Creating an object p1 of Person type like other Data type

Use Member Access Operator

Solution-3/3

```
//Displaying user Enter information
  cout << "\nDisplaying Information." << endl;
  cout << "Name: " << p1.name << endl;
  cout <<"Age: " << p1.age << endl;
  cout << "Salary: " << p1.salary;
  return 0;
}
```

Example(Initializing Structures)

```
#include<iostream>
using namespace std;
struct Student {
        string Name;
        string Address;
        string Discipline;
        float GPA;
};
int main() {
        Student S = {"Adil Aslam", "PAK", "BSCS", 3.5};
        cout << "\nStudent Name : " << S.Name;
        cout << "\nStudent Address : " << S.Address;
        cout << "\nStudent Discipline : " << S. Discipline;
        cout << "\nStudent GPA : " << S. GPA;

}
```

```
Student Name : Adil Aslam
Student Address : PAK
Student Discipline : BSCS
Student GPA : 3.5
```

Pointers to Structures

- A pointer variable can be created not only for native types like (int, float, double etc.) but they can also be created for user defined types like structure.
- We can define a pointer to a structure in the same way as any pointer to any type. For example:

```
struct employee *ptr
```


Pointers to Structures(Summary)

```
struct Student *sptr;
```

How can we access the data with sptr?

```
*sptr.name;
```

Wrong

```
sptr->name;
(*sptr).name
```

OK

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 -> .

SELF REFERENTIAL STRUCTURES

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

For eg.

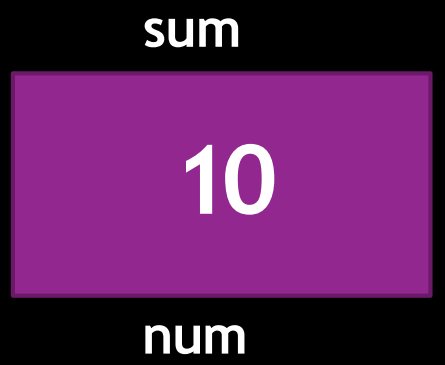
```
struct student
{int rollno;
char name[20];
student *  link}; //pointer of type structure itself
```

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

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.

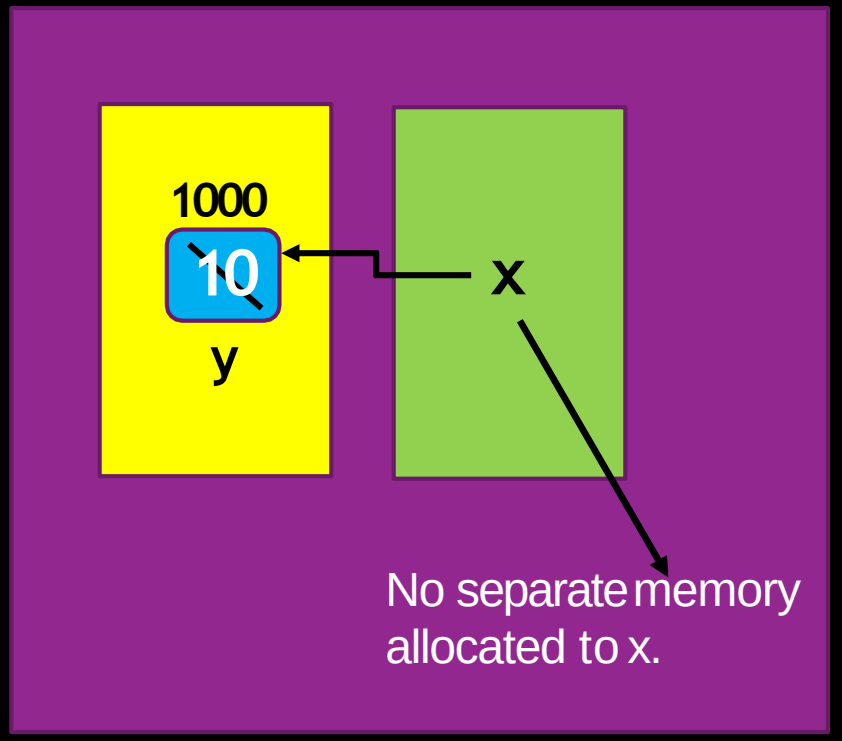
USAGE OF REFERENCE VARIABLES

- 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

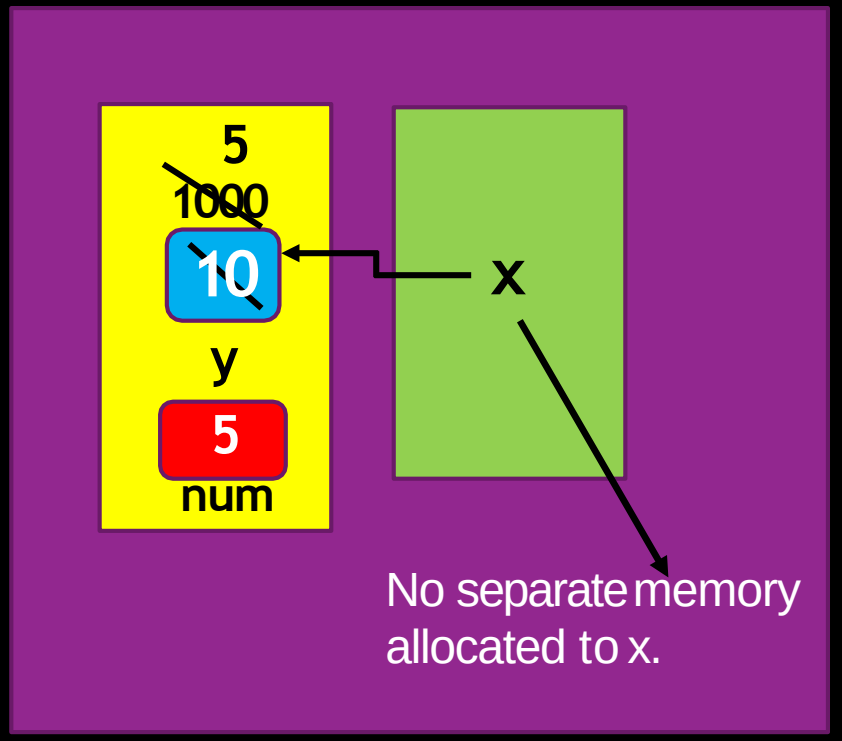
In the above program reference of y is passed. No separate memory is allocated to x and it will share the same memory as y. Thus changes made to x will also be reflected in y.

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 :

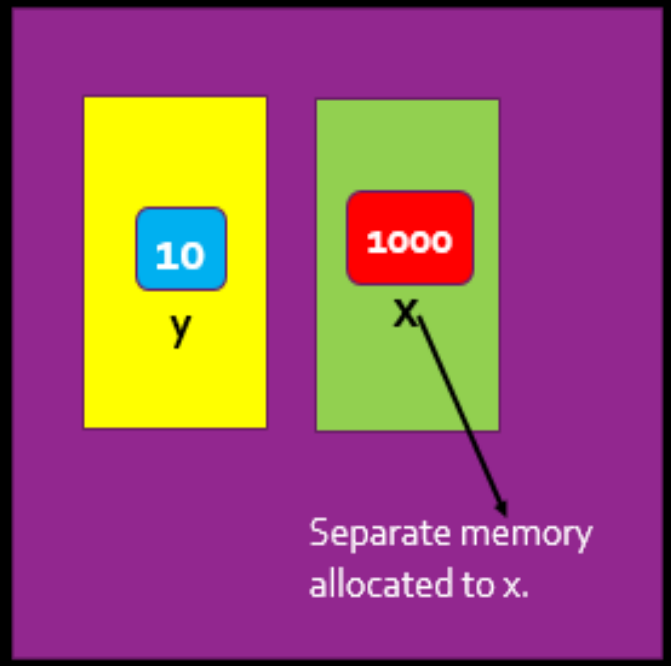
Value of a : 5 and b : 6
After swapping value of a : 6 and b : 5

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

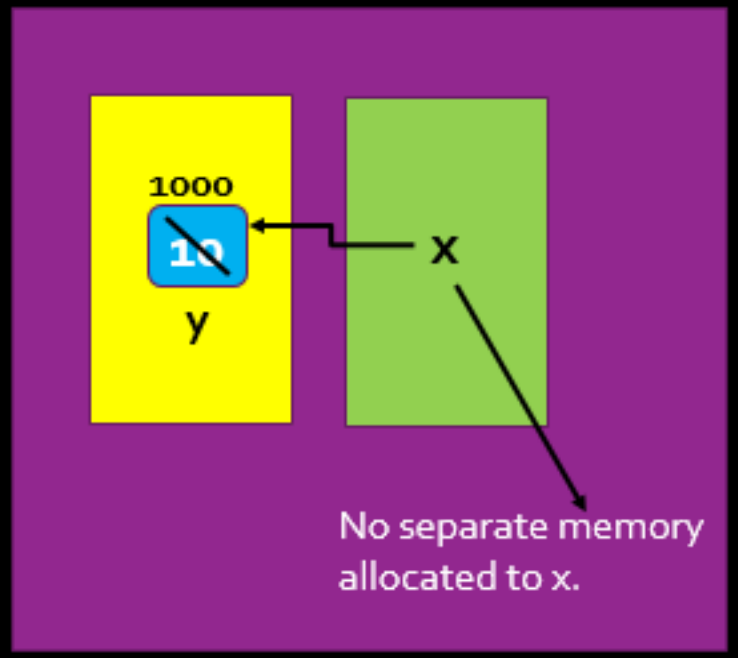
PASS BY VALUE	PASS BY REFERENCE	PASS BY POINTER
Separate memory is allocated to formal parameters.	Formal and actual parameters share the same memory space.	Formal parameters contain the address of actual parameters.
Changes done in formal parameters are not reflected in actual parameters.	Changes done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

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

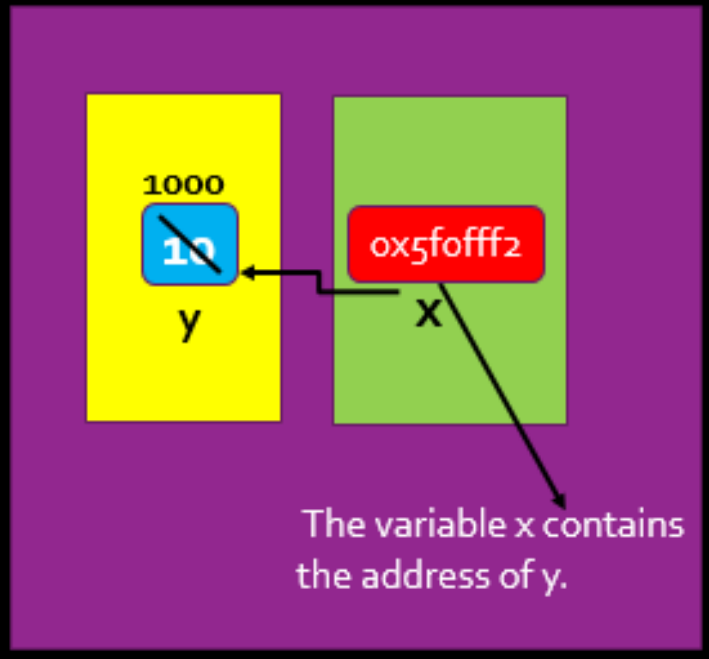
MEMORY DIAGRAM



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

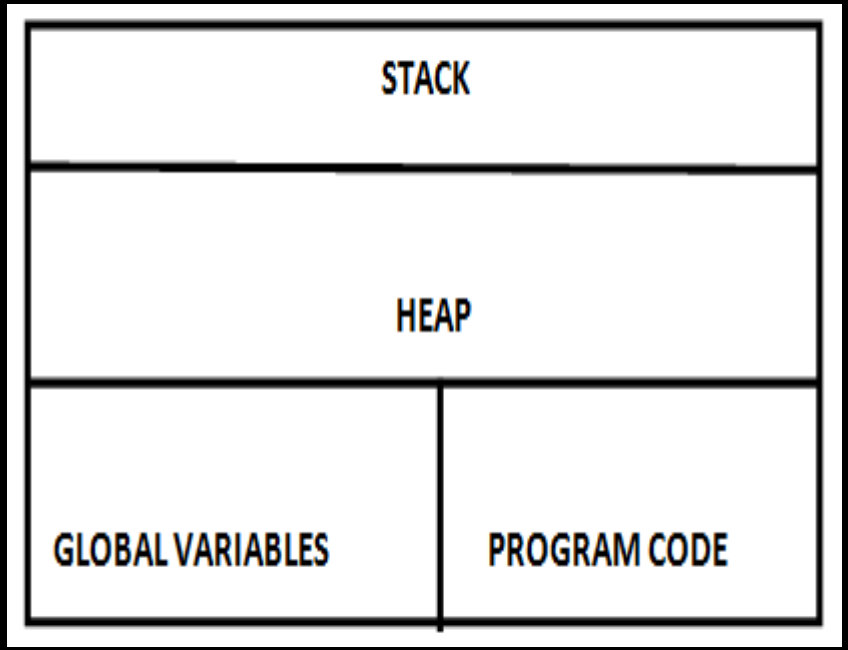
C++ dynamically allocates memory from the free store/heap/pool, the pool of unallocated heap memory provided to the program.

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;

DYNAMIC STRUCTURES

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

```
struct-pointer = newstruct-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];
```

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.