

UNIT- III

Pointers and Dynamic Memory Management

Syllabus

- Declaring and initializing pointers,
- accessing data through pointers,
- pointer arithmetic,
- memory allocation (static and dynamic), dynamic memory management using *new* and *delete* operators,
- pointer to an object,
- *this* pointer,
- pointer related problems - dangling/wild pointers,
- null pointer assignment,
- memory leak and allocation failures.
- **Constructors and Destructors:**
 - Need for constructors and destructors,
 - copy constructor,
 - dynamic constructors,
 - explicit constructors,
 - destructors,
 - constructors and destructors with static members,
 - initializer lists.

INTRODUCTION

Pointers is one of the key aspects of C++ language similar to C. Pointers offer a unique approach to handle data in C and C++. Pointer is a derived data type that refers to another data variable by storing the variables memory address rather than data.

Advantages of pointers

- i. A pointer allows a function or a program to access a variable outside the preview function or a program ,using pointer program can access any memory location in the computer's memory.
- ii. since using return statement a function can only pass back a single value to the calling function, pointers allows a function to pass back more than one value by writing them into memory locations that are accessible to calling function.
- iii. Use of pointer increases makes the program execution faster
- iv. using pointers, arrays and structures can be handled in more efficient way.
- v. without pointers it will be impossible to create complex data structures such as linked list , trees, and graphs.

Disadvantages of pointers

- i. We can access the restricted memory area.
- ii. Pointers require one additional dereference, meaning that the final code must read the variable's pointer from memory, then read the variable from the pointed-to memory. This is slower than reading the value directly from memory.
- iii. If sufficient memory is not available during run time for the storage of pointers, the program may crash.

DECLARING AND INITIALIZING POINTERS

The declaration of a pointer variable takes the following form:

data-type *pointer-variable;

Here, pointer-variable--> is name of the pointer

data-type--> refers to one of the valid c++ data types, such as int, char, float and so on.

The data-type is followed by an asterisk(*) symbol, which distinguishes a pointer variable from other variables to the compiler.

Note: we can locate asterisk (*) immediately before the pointer variable, or between the data type and the pointer variable, or immediately after the data type. It does not cause any effect in the execution process.

As we know, a pointer variable can point to any type of data available in C++. However, it is necessary to understand that a pointer is able to point to only one data type at the specific time. Let us declare a pointer variable, which points to an integer variable, as follows:

```
int *ptr;
```

Here,

ptr--> is a pointer variable and points to an integer data type. The pointer variable, ptr, should contain the memory location of any integer variable. In the same manner, we can declare pointer variables for other data types also.

We can initialize a pointer variable as follows:

```
int *ptr, a; //declaration
```

```
ptr=&a; //initialization
```

Pointer variable ptr, contains the address of the variable a. The second statement assigns the address of the variable a to the pointer ptr.

Program 1: Example of using pointers

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

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

```
int main()
```

```
{
```

```
    int a, *ptr1, **ptr2;
```

```
    clrscr();
```

```
    ptr1=&a;
```

```
    ptr2=&ptr1;
```

```
    cout<<"the address of a:"<<ptr1<<"\n";
```

```
    cout<<"the address of ptr1:"<<ptr2<<"\n";
```

```
    cout<<"After increment the address values:\n\n";
```

```
    ptr1=ptr1+2;
```

```
    cout<<"the address of a:"<<ptr1<<"\n";
```

```
    ptr2=ptr2+2;
```

```
    cout<<"the address of ptr1:"<<ptr2<<"\n";
```

```
    getch();
```

```
    return 0;
```

```
}
```

Output:

```
the address of a:0x8f7cfff4
the address of ptr1:0x8f7cfff2
After incrementing the address values:

the address of a:0x8f7cfff8
the address of ptr1:0x8f7cfff6
```

ACCESSING DATA THROUGH POINTERS

We can manipulate a pointer with the **indirection operator** i.e. ‘*’, which is also known as **dereference operator**. With this operator, we can indirectly access the data variable content. It takes the following form:

```
*pointer_variable;
```

As we know, dereferencing a pointer allows us to get the content of the memory location that the pointer points to. After assigning address of the variable to a pointer, we may want to change the content of the variable. Using the dereference operator, we can change the contents of the memory location.

Program 2: Use of dereference operator

```
#include<iostream>
using namespace std;
int main()
{
    int a = 10, *ptr;
    ptr = &a;
    cout<<"the value of a is:"<<a;
    cout<<"\n\n";
    *ptr=(*ptr)/2;
    cout<<"the value of a is:"<<a;
    cout<<"\n\n";
    return 0;
}
```

Output:

```
the value of a is:10
the value of a is:5
```

POINTERS ARITHMETIC

There are substantial number of arithmetic operations that can be performed with pointers. C++ allows pointers to perform the following arithmetic operations:

- A pointer can be incremented (++) or decremented (--)
- Any integer can be added to or subtracted from a pointer
- One pointer can be subtracted from another

Program 3: Program to illustrate the arithmetic operations that we can perform with pointers.

```
#include<iostream>
using namespace std;
int main()
{
    int num[]={56,75,22,18,90};
    int *ptr;
    int i;
    cout<<"the array values are:\n";
    for(i=0;i<5;i++)
        cout<<num[i]<<"\n";
    ptr=num; //initializing the base address of num to ptr
    cout<<"\n value of ptr      : "<<*ptr;
    cout<<"\n\n";
}
```

```

ptr++;
cout<<"the value of ptr++      : "<<*ptr;
cout<<"\n\n";
ptr--;
cout<<"the value of ptr--      : "<<*ptr;
cout<<"\n\n";
ptr=ptr+2;
cout<<"the value of ptr=ptr+2 : "<<*ptr;
cout<<"\n\n";
ptr=ptr-1;
cout<<"the value of ptr=ptr-1 : "<<*ptr;
cout<<"\n\n";
ptr=ptr+3;
cout<<"the value of ptr=ptr+3 : "<<*ptr;
cout<<"\n\n";
ptr=ptr-2;
cout<<"the value of ptr=ptr-2 : "<<*ptr;
cout<<"\n\n";
return 0;
}

```

Output:

```

the array values are:
56
75
22
18
90
value of ptr      :56
the value of ptr++ :75
the value of ptr-- :56
the value of ptr=ptr+2 :22
the value of ptr=ptr-1 :75
the value of ptr=ptr+3 :90
the value of ptr=ptr-2 :22

```

POINTERS WITH ARRAYS

Accessing an array using pointers is simpler than accessing the array index. Not only can pointers store address of a single variable, it can also store address of cells of an array.

One-dimensional array and pointers

Consider this example:

```

int *ptr;
int a[5];
ptr=&a[2]; &a[2] is the address of third element of a[5]

```

Program 4: Program to enter numbers from user and find sum of even numbers using pointer.

```

#include<iostream>
using namespace std;
int main()

```

```

{
    int num[50];
    int *ptr;
    int n, i;
    cout<<"Enter the count:\n";
    cin>>n;
    cout<<"Enter the numbers one by one\n";
    for(i=0;i<n;i++)
        cin>>num[i];
    ptr=num; //initializing the base address of num to ptr
    int sum=0;
    for(i=0;i<n;i++)
    {
        if(*ptr%2==0)
            sum=sum+*ptr;
        ptr++;
    }
    cout<<"\n sum of even numbers:"<<sum;
    cout<<"\n";
    return 0;
}

```

Output:

```

Enter the count:
5
Enter the numbers one by one
2 4 6 8 1
sum of even numbers:20

```

ARRAYS OF POINTERS

The array of pointers represents a collection of addresses. By declaring array of pointers, we can save a substantial amount of memory space. An array of pointers point to an array of data items. Each element of the pointer array points to an item of the data array. Data items can be accessed either directly or by dereferencing the elements of pointer array. we can reorganize the pointer elements without affecting the data items.

We can declare an array of pointers as follows:

```
int *inarray[10];
```

This statement declares an array of 10 pointers, each of which points to an integer. The address of the first pointer is inarray[0], and the second pointer is inarray[1], and the final pointer points to inarray[9]. before initializing they point to some unknown values in the memory space.

Program 5: Program to implement arrays of pointers.

```

#include<iostream>
using namespace std;
int main()
{
    int (*ptr)[5],*p;
    int arr[5]={3,5,6,7,9};
    p=arr;
    ptr=&arr;

```

```

cout<<"p="<<p<<"\t";
cout<<"ptr="<<ptr<<endl;
p++;
ptr++;
cout<<"p="<<p<<"\t";
cout<<"ptr="<<ptr<<endl;
cout<<"p="<<*p<<"\t";
cout<<"ptr="<<*(ptr)<<endl;
return 0;
}

```

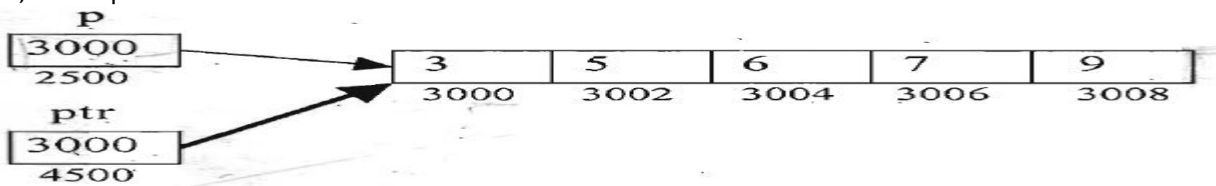
Output:

```

p = 3000. ptr = 3000
p = 3002, ptr = 3010

```

P=5, ptr= 3010

**Multi-Dimensional array and pointers****Creating a two dimensional array**

Here a two dimensional integer array num having 3 rows and 4 columns is create

```

int num[3][4] = {
    {1, 2, 3, 4},
    {5, 6, 7, 8},
    {9, 10, 11, 12}
};

```

The compiler will allocate the memory for the above two dimensional array **row-wise** meaning the first element of the second row will be placed after the last element of the first row. And if we assume that the first element of the array is at address 1000 and the size of type int is 2 bytes then the elements of the array will get the following allocated memory locations.

row-wise memory allocation

	← row 0 →				← row 1 →				← row 2 →			
value	1	2	3	4	5	6	7	8	9	10	11	12
address	1000	1002	1004	1006	1008	1010	1012	1014	1016	1018	1020	1022

↑
first element of the array num

Create pointer for the two dimensional array

We have created the two dimensional integer array num so, our pointer will also be of type int. We will assign the address of the first element of the array num to the pointer ptr using the address of & operator.

```
int *ptr=&num[0][0];
```

Accessing the elements of the two dimensional array via pointer

The two dimensional array num will be saved as a continuous block in the memory. So, if we increment the value of ptr by 1 we will move to the next block in the allocated memory.

In the following code we are printing the content of the num array using for loop and by incrementing the value of ptr.

Program 6: Program to implement multi-dimensional arrays of pointers.

```
#include <iostream>
using namespace std;
int main() {
    // 2d array
    int num[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
    // pointer ptr pointing at array num
    int *ptr = &num[0][0];
    // other variables
    int
        ROWS = 3,
        COLS = 4,
        TOTAL_CELLS = ROWS * COLS,
        i;
    // print the elements of the array num via pointer ptr
    for (i = 0; i < TOTAL_CELLS; i++) {
        cout<<*(ptr + i);
        cout<<"\t";
    }
    return 0;
}
```

Output:

1	2	3	4	5	6	7	8	9	10
11	12								

Address mapping

We can compute the address of an element of the array by using the rows and columns of the array. For this we use the following formula.

$$\text{arr}[i][j] = \text{baseAddress} + [(i * \text{no_of_cols} + j) * \text{size_of_data_type}]$$

Where, **arr**--> is a two dimensional array. i and j denotes the ith row and jth column of the array.

baseAddress--> denotes the address of the first element of the array.

No_of_cols--> is the total number of columns in the row.

Size_of_data_type--> is the size of the type of the pointer. If the type is int then size_of_data_type = 2 bytes and if the type is char then size_of_data_type = 1 bytes.

For example, in the case of num array

baseAddress = 1000,

no_of_cols = 4 and

size_of_data_type = 2.

So, we can compute the memory address location of the element num[2][3] as follows.

//address of element at cell 2,3

```
num[2][3] = baseAddress + [(i * no_of_cols + j)] * size_of_data_type]
           = 1000 + [(2*4+3)*2] = 1000 + 22 = 1022
```

Accessing the value of the two dimensional array via pointer using row and column of the array

If we want to get the value at any given row, column of the array then we can use the **value at the address of** * operator and the following formula.

```
arr[i][j] = *(ptr + (i * no_of_cols + j))
```

Where, **arr**--> is a two dimensional array and i and j denotes the ith row and jth column of the array.

ptr--> holds the address of the first element of the array.

no_of_cols--> denotes the total number of column in the row of the array.

In the following example we are finding the value at the location 2nd row and 3rd column of the array num.

//value at num[2][3] where, i=2 and j=3

```
num[2][3] = *(ptr + (i * no_of_cols + j))
           = *(ptr + (2 * 4 + 3))
           = *(ptr + 11)
```

Program 7: Program to implement multi-dimensional arrays of pointers.

```
#include <iostream>
using namespace std;
int main() {
    // 2d array
    int num[3][4] = {
        {1, 2, 3, 4},
        {5, 6, 7, 8},
        {9, 10, 11, 12}
    };
    int
    ROWS = 3,
    COLS = 4,
    i, j;
    // pointer
    int *ptr = &num[0][0];
    // print the element of the array via pointer ptr
    for (i = 0; i < ROWS; i++) {
        for (j = 0; j < COLS; j++) {
            cout << *(ptr + i * COLS + j);
            cout << "\t";
        }
        cout << "\n";
    }
    return 0;
}
```

Output:

1	2	3	4
5	6	7	8
9	10	11	12

POINTERS TO FUNCTIONS (FUNCTION POINTER)

The pointer to function is known as callback function. We can use these function pointers to refer to a function. Using function pointers we can allow a C++ program to select a function dynamically at run time. We can also pass a function as an argument to another function. Here, the function is passed as a pointer. The function pointers cannot be dereferenced. C++ also allows us to compare two function pointers.

C++ provides two types of function pointers

- i. Function pointers that point to static member functions
- ii. Function pointers that point to non-static member functions

These two function pointers are incompatible with each other. The function pointers that point to the non-static member function requires hidden argument.

Function point declaration syntax:

```
data_type (*function_name)();
```

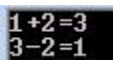
Example:

```
int (*num_function(int x));
```

Program 8: Program to declare and define function pointers.

```
#include<iostream>
using namespace std;
void add(int i, int j)
{
    cout<<i<<"+" <<j<<"="<<i+j;
}
void subtract(int i, int j)
{
    cout<<i<<"-" <<j<<"="<<i-j;
}

int main()
{
    void (*FunPtr)(int,int);
    FunPtr=add;
    FunPtr(1,2);
    cout<<endl;
    FunPtr=subtract;
    FunPtr(3,2);
    return 0;
}
```

Output:


```
1+2=3
3-2=1
```

Program 9: Program to calculate factorial of a given number function pointers.

```
#include <iostream>
using namespace std;
int fact(int n)
{
    int fact=1;
```

```

for(int i=1;i<=n;i++)
{
fact=fact*i;
}
cout<<"fact of "<<n<<"=";
return fact;
}
int main()
{
int (*ptr)(int);
ptr=&fact;
cout<<ptr(6);
return 0;
}

```

Output:

```
fact of 6=720
```

POINTERS TO OBJECTS

A pointer can point to an object created by a class. When pointer variable store the address of object then it is called pointer to object. Consider a class item defined as follows:

```

class item
{
    int code;
    float price;
public:
    void getdata(int a, float b)
    {
        code=a;
        price=b;
    }

    void show(void)
    {
        cout<<"Code:"<<code<<"\n";
        cout<<"Price:"<<price<<"\n";
    }
};

```

Now we can define a pointer ptr of type item as follows:

```
item *ptr;
```

We can also use an object pointer to access the public members of an object.

```
item x;
```

```
ptr=&x; // the pointer ptr is initialized with the address of x.
```

We can refer to the member functions of item in two ways:

i. Using the dot operator and the object

```
x.getdata(100,75.50);
```

```
x.show();
```

ii. Using the arrow operator and the object pointer

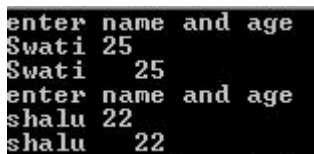
```
ptr->getdata(100,75.50);
ptr->show();
```

Since *ptr is an alias of x, we can also use (*ptr).show();

Program 10: Program to illustrate the use of pointers to objects.

```
#include<iostream>
using namespace std;
class person
{
    char name[20];
    int age;
public:
    void getdata()
    {
        cout<<"enter name and age";
        cin>>name>>age;
    }
    void display()
    {
        cout<<name<<"\t"<<age;
    }
};
int main()
{
    person x, *ptr;
    ptr=&x;
    ptr->getdata();
    ptr->display();
    cout<<endl;
    (*ptr).getdata();
    (*ptr).display();
    return 0;
}
```

Output:



```
enter name and age
Swati 25
Swati 25
enter name and age
shalu 22
shalu 22
```

THIS POINTER

The **this** pointer holds the address of current object, in simple words you can say that this **pointer** points to the current object of the class.

“This” pointer uses

- i. To know the current object address.

- ii. To distinguish class data member from local variables when both are declared with the same name. To identify data member “this” pointer is used.
- iii. Every no-static member is having local variable “this”.

Program 11: Program to illustrate the use of this pointers..

```
#include<iostream>
using namespace std;
class Test
{
    int a,b;
public:
    void show()
    {
        a=10;
        b=20;
        cout<<"obj address="<<this<<endl;
        cout<<"a = "<<this->a<<endl;
        cout<<"b = "<<this->b<<endl;
    }
};

int main()
{
    Test t;
    t.show();
    return 0;
}
```

Output:



```
obj address=0x22ff08
a = 10
b = 20
```

Program 12: Program to illustrate the use of this pointers..

```
#include<iostream>
using namespace std;
class Test
{
    int a,b;
public:
    void show(int a, int b)
    {
        this->a=a; //(*this).a=a
        this->b=b; //(*this).b=b
    }
    void display()
    {
        cout<<a<<endl<<b;
    }
};

int main()
{
    Test t;
```

```
t.show(10,20);
t.display();
return 0;
}
```

Output:

```
10
20
```

MEMORY ALLOCATION (STATIC AND DYNAMIC)

Often some situation arises in programming where data or input is dynamic in nature, i.e. the number of data item keeps changing during program execution. A live scenario where the program is developed to process lists of employees of an organization. The list grows as the names are added and shrink as the names get deleted. With the increase in name the memory allocate space to the list to accommodate additional data items. Such situations in programming require dynamic memory management techniques.

What is memory Allocation?

There are two ways via which memories can be allocated for storing data. The two ways are:

1. **Compile time allocation or static allocation of memory:** where the memory for named variables is allocated by the compiler. Exact size and storage must be known at compile time and for array declaration, the size has to be constant.
2. **Runtime allocation or dynamic allocation of memory:** where the memory is allocated at runtime and the allocation of memory space is done dynamically within the program run and the memory segment is known as a heap or the free store. In this case, the exact space or number of the item does not have to be known by the compiler in advance. Pointers play a major role in this case.

Static Memory Allocation Vs. Dynamic Memory Allocation

Attribute	Static Memory Allocation	Dynamic Memory Allocation
Definition	Static memory allocation is a method of allocating memory, and once the memory is allocated, it is fixed.	Dynamic memory allocation is a method of allocating memory, and once the memory is allocated, it can be changed.
Modification	In static memory allocation, it is not possible to resize after initial allocation.	In dynamic memory allocation, the memory can be minimized or maximize accordingly
Implementation	Static memory allocation is easy to implement.	Dynamic memory allocation is complex to implement.
Speed	In static memory, allocation execution is faster than dynamic memory allocation.	In dynamic memory, allocation execution is slower than static memory allocation.
Memory Utilization	In static memory allocation, cannot reuse the unused memory.	Dynamic memory allocation allows reusing the memory. The programmer can allocate more memory when required . He can release the memory when necessary.

What is Dynamic memory allocation?

Dynamic memory Allocation refers to performing memory management for dynamic memory allocation manually.

Programmers can dynamically allocate storage space while the program is running, but programmers cannot create new variable names "on the fly", and for this reason, dynamic allocation requires two criteria:

- Creating the dynamic space in memory
- Storing its address in a pointer (so that space can be accessed)

Memory in your C++ program is divided into two parts:

- a. **stack:** All variables declared inside any function takes up memory from the stack.
- b. **heap:** It is the unused memory of the program and can be used to dynamically allocate the memory at runtime.

DYNAMIC MEMORY MNAAGEMENT USING NEW AND DELETE OPERATORS

There are two memory management operators used for dynamic memory management

- i. new
- ii. delete

Allocating space using new

To allocate space dynamically, use the unary operator **new**, followed by the type being allocated.

```
int * p; // declares a pointer p
p = new int; // dynamically allocate an int for loading the address in p
```

```
double * d; // declares a pointer d
d = new double; // dynamically allocate a double and loading the address in d
```

Dynamic Memory Allocation for Arrays

If you as a programmer; wants to allocate memory for an integer n. Using that same syntax, programmers can allocate memory dynamically as shown below.

```
int n;
int *p= new int[n];    // Request memory for the variable
```

De-allocating space using delete

When programmers think that the dynamically allocated variable is not required anymore, they can use the delete operator to free up memory space. The syntax of using this is:

```
delete var_name;
```

Here is a simple program showing the concept of dynamic memory allocation:

Example:

```
#include <iostream>
using namespace std;
int main(){
int *p;
```

```

float *q;
char *r;
p = new int;
q = new float;
r = new char;
cout<<"enter one integer, one float and one character value";
cin>> *p>>*q>>*r;
cout << "Value are : " << *p << "\t"<<*q << "\t"<<*r << "\t";
delete p;
delete q;
delete r;
}

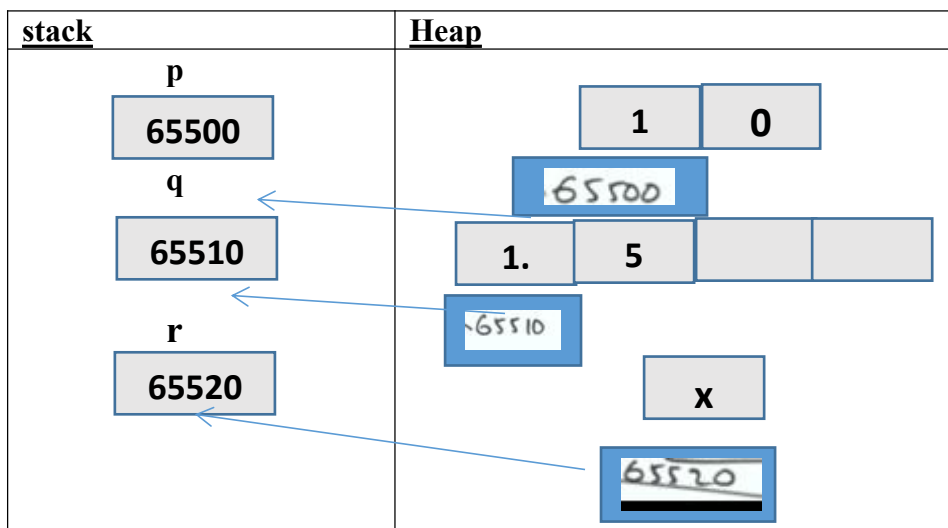
```

Output:

```

enter one integer, one float and one character value
10 1.5 x
Value are : 10 1.5 x

```

Memory Representation of above program**Program 13: Dynamic 1-D arrays**

```

#include <iostream>
using namespace std;
int main()
{
    int n;
    cout<<"enter array size";
    cin>>n;
    int *p=new int[n];
    cout<<"enter the elements";
    for(int i=0;i<n;i++)
    {
        cin>>*(p+i);
    }
    cout<<"elements are";
    for(int j=0;j<n;j++)
    {
        cout<<*(p+j);
    }
}

```

```

    cout<<"\t";
}
cout<<"\n";
delete p;
return 0;
}

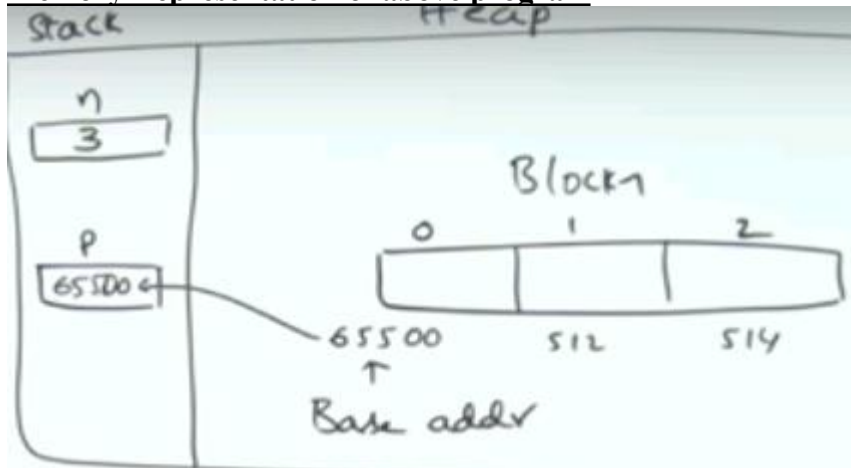
```

Output:

```

enter array size 3
enter the elements 1 2 3
elements are 1 2 3

```

Memory Representation of above program**Program 14: Dynamic 2-D arrays**

```

#include <iostream>
using namespace std;
int main()
{
    int **p, nr, nc, r, c;
    cout<<"enter number of rows and columns";
    cin>>nr>>nc;
    p=new int *[nr];
    for(r=0;r<nr;r++)
    {
        p[r]=new int[nc];
    }
    cout<<"enter    "<<nr*nc<<"    elements";
    for(r=0;r<nr;r++)
        for(c=0;c<nc;c++)
            cin>>p[r][c];
    cout<<"elements are"<<endl;
    for(r=0;r<nr;r++)
        for(c=0;c<nc;c++)
            cout<<p[r][c]<<"\t";
    cout<<"\n";
    delete p;
    return 0;
}

```

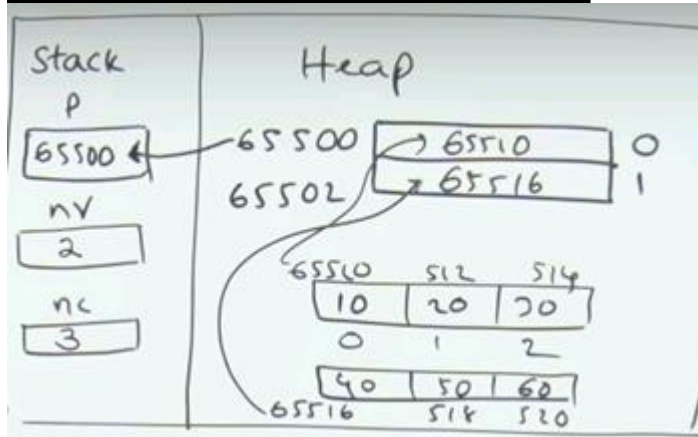
Output:


```

enter number of rows and columns 2 3
enter 6 elements
10 20 30 40 50 60
elements are
10      20      30      40      50      60

```

Memory Representation of above program



POINTER RELATED PROBLEMS-DANGLING/WILD POINTERS

Dangling pointer

A pointer pointing to a memory location that has been deleted (or freed) is called dangling pointer.

Example:

```

#include <iostream>
using namespace std;
int main(){
    int *ptr=NULL;
    ptr = new int;
    cout<<"enter a integer value";
    cin>> *ptr;
    cout << "Value is : " << *ptr << endl;
    delete ptr; //ptr becomes dangling pointer
    ptr=NULL; //no more dangling pointer
    return 0;
}

```

Wild pointer

A pointer which has not been initialized to anything (not even NULL) is known as wild pointer. The pointer may be initialized to a non-NULL garbage value that may not be a valid address.

Example:

```

#include<iostream>
using namespace std;
int main()
{
    int *p; //wild pointer
    int x=10; //p is not a wild pointer now
    p=&x;
    return 0;
}

```

}

Null Pointer

It is always a good practice to assign the NULL to a pointer variable in case you do not have exact address to be assigned. This is done at the time of variable declaration. A pointer that is assigned NULL is called a **null** pointer.

The NULL pointer is a constant with a value of zero defined in several standard libraries, including iostream.

Example:

```
#include <iostream>
using namespace std;
int main () {
    int *ptr = NULL;
    cout << "The value of ptr is " << ptr ;
    return 0;
}
```

Output:

The value of ptr is 0

Void Pointers

A void pointer also known as generic pointers, which refer to variable of any data type. Before using void pointers, we must type cast the variables to the specific data types that they point to.

```
int a=10;
char b='x';
void *p=&a; //void pointer holds address of int 'a'
p=&b; //void pointer holds address of char 'b'.
```

Example

```
#include<iostream>
using namespace std;
int main()
{
    int a = 10;
    void *ptr = &a;
    cout<<*(int *)ptr; //type casting
    return 0;
}
```

Output:

MEMORY LEAK AND ALLOCATION FAILURES

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 can not 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.

Problems can be caused by memory leaks

The problem caused by a memory leak is that it leaves chunk(s) of memory unavailable for use by the programmer. If a program has a lot of memory that hasn't been deallocated, then that could really slow down the performance of the program. If there's no memory left in the program because of memory leaks, then that could of course cause the program to crash.

Here is an example of a memory leak in C++:

```
void memLeak()
{
    int *data = new int;
    *data = 15;
}
```

So, the problem with the code above is that the “*data” pointer is never deleted – which means that the data it references is never deallocated, and memory is wasted.

NEED FOR CONSTRUCTORS

Constructor is a ‘special’ function whose task is to initialize the objects of its class. It is special because its name is the same as the class name. The constructor is invoked whenever an object of its associated class is created. It is called constructor because it constructs the values of data members of the class.

Constructor is used for

- i. **To initialize data member of class:** In the constructor member function (which will be declared by the programmer) we can initialize the default values to the data members and they can be used further for processing.
- ii. **To allocate memory for data member:** Constructor can also be used to declare run time memory (dynamic memory for the data members).

Constructor characteristics

- i. They should be declared in the public section.
- ii. They are invoked automatically as soon as Objects of Class are created.
- iii. They do not have return types, not even void.
- iv. They can not be Inherited, though derived class can call the base class constructor.
- v. Like other C++ functions, they can have default arguments.
- vi. Constructors cannot be virtual.
- vii. Constructor overloading is possible.
- viii. It can not be static.
- ix. It is possible to have more than one constructor in a class.

Types of constructors

There are three basic categories of Constructors:

- i. **Default Constructor:** It takes no arguments and performs no processing other than reservation of memory. It will always call by the compiler, if no user defined constructor is being provided.

Syntax :- class-name () { code } ;

- ii. **Parameterized Constructor:** The constructors that can take arguments are called parameterized constructors.

Syntax :- class-name (parameter-list) { code } ;

- iii. **Copy Constructor:** It is used to declare and initialize an object from another object. It takes a reference to an object of the same class as itself as an argument.

Syntax :- class-name (reference of the same class) { code } ;

DEFAULT CONSTRUCTOR

Example 1:

```
#include<iostream>
using namespace std;
class Cube
{
    public:
    int side;
    Cube()
    {
        side=10;
    }
};

int main()
{
    Cube c;
    cout << c.side;
}
```

Output:



In this case, as soon as the object is created the constructor is called which initializes its data members. A default constructor is so important for initialization of object members, that even if we do not define a constructor explicitly, the compiler will provide a default constructor implicitly.

Example 2:

```
#include<iostream>
using namespace std;
class Cube
{
    public:
    int side;
};

int main()
{
    Cube c;
    cout << c.side;
    return 0;
}
```

Output:



In this case, default constructor provided by the compiler will be called which will initialize the object data members to default value, or any random integer value in this case.

PARAMETERIZED CONSTRUCTOR

Example:

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

```
class Cube
{
    public:
    int side;
    Cube(int x)
    {
        side=x;
    }
};
```

```
int main()
{
    Cube c1(10);
    Cube c2(20);
    Cube c3(30);
    cout << c1.side<<"\t";
    cout << c2.side<<"\t";
    cout << c3.side<<"\t";
    return 0;
}
```

Output:

```
10    20    30
```

By using parameterized constructor in above case, we have initialized 3 objects with user defined values. We can have any number of parameters in a constructor.

COPY CONSTRUCTOR

Example:

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

```
class Cube
{
    int side;
    public:
    Cube(){}           //constructor
    Cube(int x)        //constructor again
    {
        side=x;
    }
    Cube(Cube & x)     //copy constructor
```

```

{
    side=x.side;
}
void display()
{
    cout<<side;
}
};
int main()
{
    Cube c1(10);    //object c1 is created and initialized
    Cube c2(c1);    //copy constructor called
    Cube c3=c1;     //copy constructor called again
    Cube c4;        //c4 is created, not initialized
    c4=c1;          //copy constructor not called
    cout << "\n Side of c1:"; c1.display();
    cout << "\n Side of c2:"; c2.display();
    cout << "\n Side of c3:"; c3.display();
    cout << "\n Side of c4:"; c4.display();
    return 0;
}

```

Output:

```

Side of c1:10
Side of c2:10
Side of c3:10
Side of c4:10

```

DYNAMIC CONSTRUCTORS

The constructors can also be used to allocate memory while creating objects. This will enable the system to allocate the right amount of memory for each object when the objects are not of the same size, thus, resulting in the saving of memory. The allocation of Memory to objects at the time of their construction is known as **dynamic construction of objects**. The memory is allocated at run time with the help of 'new' operator.

Example:

```

#include<iostream>
using namespace std;
class code
{
    int * p;
public:
    code()
    {
        p=new int;
        *p=10;
    }
    code(int v)
    {
        p=new int;
        *p=v;
    }
}

```

```

int dis()
{ return(*p);
}
};

int main()
{
code c1, c2(9);
cout<<"The value of object c1's p is:";
cout<<c1.dis();
cout<<"\nThe value of object c2's p is:"<<c2.dis();
return 0;
}

```

Output:

```

The value of object c1's p is:10
The value of object c2's p is:9

```

EXPLICIT CONSTRUCTORS

The explicit function specifier controls unwanted implicit type conversions. It can only be used in declarations of constructors within a class declaration. For example, except for the default constructor, the constructors in the following class are conversion constructors.

Example 1:

```

#include<iostream>
using namespace std;
class Base
{
    int b_var;
public:
    Base() { }
    Base(int var)
    {
        b_var=var;
    }
    void print ()
    {
        cout<<b_var<<endl;
    }
    void fun(Base b)
    {
        b.print();
    }
};

int main()
{
    Base obj1(10); //normal call to constructor
    Base obj2=20; //implicit call to constructor
    obj1.fun(obj1); //normal call to constructor
    obj1.fun(30); //implicit call to constructor
}

```

```
return 0;
}
```

Output:

```
10
30
```

If you do not want implicit call to constructor then you have to use explicit keyword in front of constructor.

Example 2:

```
#include<iostream>
using namespace std;
class Base
{
    int b_var;
public:
    Base()
    {

    }

    explicit Base(int var)
    {
        b_var=var;
    }
    void print ()
    {
        cout<<b_var<<endl;
    }
    void fun(Base b)
    {
        b.print();
    }
};

int main()
{
    Base obj1(10); //normal call to constructor
    Base obj2=20; //implicit call to constructor
    obj1.fun(obj1); //normal call to constructor
    obj1.fun(30); //implicit call to constructor
    return 0;
}
```

It will give error.

DESTRUCTOR

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.

The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a **tilde** ~ sign as prefix to it.

Example 1:

```
class A
{
    public:
    // defining destructor for class
    ~A()
    {
        // statement
    }
};
```

Example 2:

```
#include<iostream>
using namespace std;
class A
{
    // constructor
    public:
    A()
    {
        cout << "Constructor called"<<endl;
    }

    // destructor
    ~A()
    {
        cout << "Destructor called"<<endl;
    }
};

int main()
{
    A obj1;    // Constructor Called
    int x=1;
    if(x)
    {
        A obj2;    // Constructor Called
    }    // Destructor Called for obj2
    return 0;
} // Destructor called for obj1
```

Output:

```
Constructor called
Constructor called
Destructor called
Destructor called
```

CONSTRUCTOR AND DESTRUCTORS WITH STATIC MEMBERS

Every object has its own set of data members. When a member function is invoked, only copy of data member of calling object is available to the function. Sometimes, it is necessary for all the objects to share fields which are common for all the objects. If the member variable is declared as static, only one copy of such member is created for entire class. All objects access the same copy of static variable. The static

member variable can be used to count the number of objects declared for a particular class. The following program helps you to count the number of objects declared for a class.

Example:

```
#include<iostream>
using namespace std;
class man
{
    static int no;
    char name;
    int age;
public:
    man()
    {
        no++;
        cout<<"\n number of objects exists:  "<<no;
    }
    ~man()
    {
        --no;
        cout<<"\n number of objects exists:  "<<no;
    }
};

int man :: no=0;
int main()
{
    man A, B,C;
    cout<<"\n press any key to destroy object";
    return 0;
}
```

Output:

```
number of objects exists: 1
number of objects exists: 2
number of objects exists: 3
press any key to destroy object
number of objects exists: 2
number of objects exists: 1
number of objects exists: 0
```

INITIALIZER LISTS

Initializer list is used to initialize data members of a class. The list of members to be initialized is indicated with constructor as a comma separated list followed by a colon. The initializer list does not end in a semicolon.

Syntax:

```
Constructorname(datatype value1, datatype value2):datamember(value1),datamember(value2)
{
    ...
}
```

Uses of Initializer List in C++

There are situations where initialization of data members inside constructor doesn't work and Initializer List must be used. Following are such cases:

1) For initialization of non-static const data members:

const data members can be initialized only once, so it must be initialized in the initialization list.

Example

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

class Base
{
    private:
        const int x;
    public:
        Base(int y): x(y)
        {
            cout << "Value is " << x;
        }
};

int main()
{
    Base il(10);
}
```

Output:

```
Value is 10
```

2) When reference type is used:

Reference members must be initialized using Initializer List

Example

If you have a data member as reference type, you must initialize it in the initialization list. References are immutable hence they can be initialized only once.

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

```
class Base
{
    private:
    int &y;
    public:
    Base(int &n) : y(n)
    {
        cout << "Value is " << y;
    }
};
```

```
int main()
{
    int m=10;
    Base il(m);
    return 0;
}
```

Output:**Value is 10****3) When data member and constructor's parameter have same name:**

If constructor's parameter name is same as data member name then the data member must be initialized using Initializer List.

Example

```
#include<iostream>
using namespace std;
class Base
{
    private:
    int x;
    public:
    Base(int x):x(x)
    {
        cout << "Value is " << x;
    }
};
```

```
int main()
{
    Base il(10);
    return 0;
}
```

Output:**Value is 10****4) For improving performance:**

It is better to initialize all class variables in Initializer List instead of assigning values inside body.

CONSTRUCTOR OVERLOADING IN C++

Just like other member functions, constructors can also be overloaded. Infact when you have both default and parameterized constructors defined in your class you are having Overloaded Constructors, one with no parameter and other with parameter.

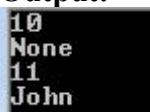
You can have any number of Constructors in a class that differ in parameter list.

Example:

```
#include<iostream>
using namespace std;
class Student
{
    int rollno;
    string name;
public:
    // first constructo
    Student(int x)
    {
        rollno=x;
        name="None";
    }
    // second constructor
    Student(int x, string str)
    {
        rollno=x ;
        name=str ;
    }

    void show()
    {
        cout<<rollno<<endl;
        cout<<name<<endl;
    }
};

int main()
{
    // student A initialized with roll no 10 and name None
    Student A(10);
    A.show();
    // student B initialized with roll no 11 and name John
    Student B(11, "John");
    B.show();
}
```

Output:


```
10
None
11
John
```

In above case we have defined two constructors with different parameters, hence overloading the constructors.

One more important thing, if you define any constructor explicitly, then the compiler will not provide default constructor and you will have to define it yourself.

In the above case if we write Student S; in **main()**, it will lead to a compile time error, because we haven't defined default constructor, and compiler will not provide its default constructor because we have defined other parameterized constructors.

INLINE FUNCTION

An inline function looks like a regular function but, is preceded by the keyword “**inline**”. Inline functions are short length functions which are expanded at the point of its invocation, instead of being called. Let's understand inline functions with an example.

Example:

```
#include <iostream>
using namespace std;
class example{
int a, b;
public:
inline void initialize( int x, int y)
{ a=x;
b=y ;
}
void display()
{ cout<< a <<" "<<b <<" \n";
} //automatic inline
};
int main( )
{
example e;
e.initialize(10, 20);
e.display();
}
```

Output:



In above program, I declared and defined, the function initialize(), as an inline function in the class “example”. The code of the initialization() function will expand where it is invoked by the object of the class “example”. The function display(), defined in the class example is not declared inline but it may be considered inline by the compiler, as in C++ the function defined inside the class are automatically made inline by the compiler considering the length of the function.

Key Differences Between Inline and Macro

1. The basic difference between inline and macro is that an inline functions are parsed by the compiler whereas, the macros in a program are expanded by preprocessor.
2. The keyword used to define an inline function is “**inline**” whereas, the keyword used to define a macro is “**#define**”.
3. Once inline function is declared inside a class, it can be defined either inside a class or outside a class. On the other hand, a macro is always defined at the start of the program.
4. The argument passed to the inline functions are evaluated only once while compilation whereas, the macros argument are evaluated each time a macro is used in the code.

5. The compiler may not inline and expand all the functions defined inside a class. On the other hand, macros are always expanded.
6. The short function that are defined inside a class without inline keyword are automatically made inline functions. On the other hand, Macro should be defined specifically.
7. A function that is inline can access the members of the class whereas, a macro can never access the members of the class.
8. To terminate a inline function a closed curly brace is required whereas, a macro is terminated with the start of a new line.
9. Debugging become easy for inline function as it is checked during compilation for any error. On the other hand, a macro is not checked while compilation so, debugging a macro becomes difficult.
10. Being a function an inline function bind its members within a start and closed curly braces. On the other hand, macro do not have any termination symbol so, binding becomes difficult when macro contains more than one statements.

Question Bank

Apr-May-2018

3. (a) What is pointer in C++? Give its merits and demerits. [2]
- (b) What is copy constructor? Explain it with example. [7]
- (c) What do you mean by memory allocation? How is dynamic allocation done? Explain with example. [7]
- (d) Explain about *this* pointer. Illustrate with example. [7]

Nov-Dec-2017

3. (a) How to declare and initialize the pointer ? [2]
- (b) Explain dynamic memory management using new and delete operator with program. [7]
- (c) What is the pointer? Also explain pointer arithmetic with example. [7]
- (d) What are constructors and destructors and why is it needed ? [7]

Apr-May-2017

3. (a) What do you mean by pointer to object ? [2]
- (b) Write a program to handle a multidimensional array by pointer. [7]
- (c) Write a program to find maximum memory available for dynamic allocation. [7]
- (d) Explain the following operators : [$3\frac{1}{2}+3\frac{1}{2}=7$]
- (i) New
- (ii) Delete

Nov-Dec-2016

3. (a) What do you mean by Destructors?

- (b) Write a program to demonstrate constructor overloading. 7
- (c) What is dynamic memory management? Explain with an example. 7
- (d) What is 'this' pointer? Explain the use of 'this' operator. 7

Apr-May-2016

3. (a) Write difference between constructor and destructor. 2
- (b) Explain dynamic memory management in C++. 7
- (c) Explain copy constructure with program. 7
- (d) Write short notes on :
- (i) this pointer 2
- (ii) wild pointer 2
- (iii) null pointer 1
- (iv) void pointer 2



Nov-Dec-2015

3. (a) Describe the importance of destructor.	2
(b) What is constructor? List some of its special properties.	7
(c) Explain this pointer in detail.	7
(d) Explain memory allocation in detail.	7

Apr-May-2015

3. (a) What is a Default Constructor?	
(b) What is this pointer explain with suitable example and also explain the need of this pointer?	7
(c) What is Inline? Write down difference between Macro and Inline Substitution.	7
(d) Write an OOP's program to add two complex number using constructor overloading.	7

Nov-Dec-2014

Q. 3. (a) What is pointer in C++, give its merits & demerits. 2

(b) The C++ program segments are not valid. Please explain the error in each program : 7

(i). `int *p;`
`int **q;`
`p=&q;`

(ii). `int **p;`
`int **q;`
`p=&q;`

(iii). `int **p;`
`float *q;`
`p=&q;`

(c) What is the output of following Program – justify

```
#include <iostream>

using namespace std;

int main()
{
    int a = 5, b = 10, c = 15;
    int *arr[] = {&a, &b, &c};
    cout << arr[1];
    return 0;
}
```

(d) What is constructor, Different types of Constructors ? How it is defined and called explain with suitable example. 7

Apr-May-2014

Q. 3. (a) What is a inline function ? 2

(b) What do you mean by memory allocation ?

How dynamic allocation is done, explain

with example.



7

(c) Explain about *this* pointer, illustrate with example. 7

(d) What is a constructor and explain different types of constructor. 7