51). Differentiate between constructor and destructor.

->

constructor	Destructor
1) It has the same name as the class name.	1) It has the same name as the class name pr-ceded by tilde.
2) It is automatically called when an object is Created.	2) It is automaticallt when an object gets out of Scope car is destroyed.
3) It can be overloaded.	3) It cannot be overloaded.
4) It cannot be virtual.	4) It can be virtual.
5) it can have argument but no return type.	5) It neither has argument nor return type,
6) Constructor invocation order is same as Object creation order.	6) Destructor invocation order is reverse of obj-Ect creation order.

7)

For eg-

```
Class test
{
Private:
//…..
Public:
 Test()
{
…….
}
```

7)

For eg-:

```
class test
{
private
   //……
  public:
  ~test()
{
   ……..
}
```

52). WAP to show pointer to object.

```cpp
#include <iostream>
using namespace std;
class MyClass {
public:
    int data;
    MyClass(int d) {
        data = d;
    }
    void display() {
        cout << "Data: " << data << endl;
    }
};
int main() {
    MyClass obj(10); // Create an object
    // Pointer to the object
    MyClass *ptr = &obj;
    // Accessing members using pointer
    cout << "Accessing through pointer:" << endl;
    ptr->display(); // Using arrow operator
    cout << "Data: " << ptr->data << endl;
    // Dereferencing the pointer
    cout << "Accessing through dereferenced pointer:" << endl;
    (*ptr).display(); // Using dot operator
    cout << "Data: " << (*ptr).data << endl;
    return 0;
}
```

53). Write the syntax for DMA for objects and object arrays. WAP to show DMA for objects and object arrays.

DMA stands for Dynamic Memory Allocation. In C++, DMA is done through the keywords new and the memory deallocation is done through the delete keyword. The syntax for DMA for objects and objects array is as follows:

```
//syntax for DMA for single object
class_name = *pointer_name;
pointer_name = new class_name;

//syntax for DMA for array of object
class_name = *pointer_name;
pointer_name = new class_name[size];
```

For single object:

```cpp
#include <iostream>
using namespace std;
class num
{
private:
int data;
public:
int getdata()
{
    cout<<"Enter the data: ";
    cin>>data;
    return 0;
}
int display()
{
    cout<<"The data is: "<<data;
    return 0;
}
};
int main()
{
    num *a;
    a = new num;
    a->getdata();
    a->display();
    delete a;
    return 0;
}
```

For many objects:

```cpp
#include <iostream>
using namespace std;
class num
{
private:
int data;
public:
int getdata()
{
    cin>>data;
    return 0;
}
int display()
{
    cout<<"The data is: "<<data;
    return 0;
}
};
```

```
int main()
{
    int n;
    cout<<"Enter the no. of objects to be created: ";
    cin>>n;
    num *a;
    a = new num[n];
    cout<<"Inserting the data for different objects:\n";
    for(int i = 0; i < n;i++ )
        {
            cout<<"Enter the data for object: ";
            (a+i)->getdata();
        }
    for(int i = 0; i < n;i++ )
        {
            cout<<"The data of object"<<i<<":";
            (a+i)->display();
            cout<<endl;
        }
    delete a;
    return 0;
}
```

54). List the properties for constant member function and constant object. Write their syntax. Also write a program to show constant member functions and constant objects.

Constant Data Member Properties

- **Read-only:** Once initialized, the value of a constant data member cannot be changed.
- **Initialization:** Must be initialized during declaration or in the constructor's initializer list.
- **No direct assignment:** Cannot be assigned a value after declaration.
- **Scope:** Has the same scope as other data members within the class.
- **Access specifiers:** Can be public, private, or protected like other data members.
- **Const correctness:** Can be used to ensure data integrity and prevent accidental modifications.

Constant Member Function Properties

- **Read-only:** Guarantees that the function will not modify the object's state.
- **Const keyword:** Declared with the const keyword after the parameter list.
- **Access to non-const data members:** Can access only constant data members and other constant member functions.
- **Can be called on const objects:** Essential for working with constant objects.
- **Overloading:** Can be overloaded with non-const versions for different behaviors.
- **Const correctness:** Promotes code reliability and prevents unintended modifications.

```
//constant data member
class MyClass {
public:
    // ...
private:
    const int constantValue = 10;
};

//constant member function
class MyClass {
public:
    int getValue() const {
        return constantValue;
    }
    // ...
private:
    // ...
};
```

Program to show constant member functions and constant objects.

```cpp
#include <iostream>
class Circle {
public:
    Circle(double r) : radius(r) {}

    double getArea() const {
        return 3.14159 * radius * radius;
    }
    double getRadius() const {
        return radius;
    }
private:
    const double radius;
};
int main() {
    const Circle c1(5); // Constant object
    Circle c2(3);
    std::cout << "Area of c1: " << c1.getArea() << std::endl;
    std::cout << "Radius of c2: " << c2.getRadius() << std::endl;

    // c1.radius = 6; // Error: cannot modify constant object
    // c2.getArea() = 10; // Error: cannot assign to function call

    return 0;
}
```

55). What is this pointer? What is its importance? WAP showing the use of this pointer.

In C++, the this pointer is a special pointer that implicitly exists within the scope of non-static member functions of a class. It points to the current object, i.e., the object whose member function is being called.

Importance of the this pointer:

1. **Accessing object's data members:** It allows one to access the object's data members from within its member functions. This is essential for manipulating the object's state.
2. **Disambiguating between local variables and data members:** If a local variable and a data member have the same name, the this pointer can be used to explicitly access the data member.
3. **Returning a reference to the object itself:** One can use the this pointer to return a reference to the current object, enabling method chaining.
4. **Passing the current object to other functions:** One can pass the this pointer as an argument to other functions, giving them access to the object's members.

Program showing the use of this pointer.

```cpp
#include <iostream>
using namespace std;
class MyClass {
public:
    MyClass(int x) : data(x) {}
    void printData() {
        cout << "Data: " << data << std::endl;
    }
    void incrementData() {
        (*this).data++;
    }
    MyClass& operator++() {
        data++;
        return *this;
    }
    MyClass operator++(int) {
        MyClass temp(*this);
        data++;
        return temp;
    }
};
```

```cpp
int main() {
    MyClass obj(5);
    cout << "Before increment: ";
    obj.printData();
    obj.incrementData();
    cout << "After increment: ";
    obj.printData();
    obj++;
    cout << "After post-increment: ";
    obj.printData();
    ++obj;
    cout << "After pre-increment: ";
    obj.printData();
    return 0;
}
```

56). Justify with example:

a) In the case of friendship between classes, friendship is not mutual.

In C++, friendship between classes is a one-way relationship. If class A is a friend of class B, it doesn't automatically imply that class B is a friend of class A. This means that class A can access the private and protected members of class B,but class B cannot access the private and protected members of class A unless it's explicitly declared as a friend.

```
#include <iostream>
using namespace std;
class ClassB {
public:
    int public_data = 10;
private:
    int private_data = 20;
};
class ClassA {
public:
    void accessB(ClassB &b) {
        cout << "ClassA accessing public data of ClassB: " << b.public_data <<endl;
        cout << "ClassA accessing private data of ClassB: " << b.private_data <<endl;
    }
};
int main() {
    ClassA a;
    ClassB b;
    a.accessB(b);
}
```

b) Friend function and classes breach the wall of OOP.

Object-Oriented Programming (OOP) is built on the principles of encapsulation, inheritance, and polymorphism.Encapsulation, in particular, emphasizes the idea of hiding implementation details within a class and providing access to them through well-defined interfaces (public methods). **Friend functions and classes** are exceptions to this principle. They are granted special privileges to access the private members of a class, bypassing the encapsulation barrier. This can be argued to breach the wall of OOP.

```cpp
#include <iostream>
using namespace std;
class MyClass {
private:
    int data = 10;
public:
    friend void friendFunction(MyClass obj);
};
void friendFunction(MyClass obj) {
    cout << "Friend function accessing private data: " << obj.data << std::endl;
}
int main() {
    MyClass obj;
    friendFunction(obj);
    return 0;
}
```

c) **A friend function is not a member of any classes but has full access to the members of the class where it is declared as friend.**

A friend function is indeed not a member of any class, but it is granted special privileges to access the private and protected members of the class that declares it as a friend.

```cpp
#include <iostream>
using namespace std;
class MyClass {
private:
    int data = 10;
public:
    friend void friendFunction(MyClass obj);
};
void friendFunction(MyClass obj) {
    cout << "Friend function accessing private data: " << obj.data <<endl;
}
int main() {
    MyClass obj;
    friendFunction(obj);
    return 0;
}
```

Explanation:

- friendFunction is declared as a friend of MyClass.
- It's defined outside the class, making it a global function, not a member of any class.
- However, due to the friend declaration, it can directly access the private member data of MyClass.

57). Write a program designing a class called midpoint to find the midpoint

between two points by returning an object from a member function using this pointer.

```
#include <iostream>
using namespace std;
class point
{
private:
int x;
int y;
public:
void getdata(){
cout<<"Enter x-coordinate: ";
cin>>x;
cout<<"Enter y-coordinate: ";
cin>>y;
}
void display()
{
cout<<"("<<x<<","<<y<<")"<<endl;
}
point midpoint(point a,point b)
{
point p;
p.x=(a.x + b.x)/2;
p.y=(a.y + b.y)/2;
return p;
}
};
```

```
int main()
{
    point p1,p2,*p3;
    p3 = new point;
    p1.getdata();
    p2.getdata();
    *p3 = p1.midpoint(p1,p2);
    cout<<"The midpoint is: ";
    p3->display();
    cout<<endl;
    return 0;
}
```

58). Explain the invocation order of constructor and destructor with an example.

Invocation Order of Constructors and Destructors

Constructor Invocation

- **Object creation:** When an object of a class is created, its constructor is automatically called.
- **Base class constructors:** If a class inherits from a base class, the base class constructor is called *before* the derived class constructor. This follows a top-down approach.
- **Member objects:** If a class contains member objects, their constructors are called before the class constructor.
- **Initialization list:** Members can be initialized in the constructor's initialization list, which is executed before the constructor body.

Destructor Invocation

- **Object destruction:** When an object goes out of scope or is explicitly deleted, its destructor is called.
- **Derived class destructor:** The destructor of the derived class is called *before* the base class destructor. This is the reverse order of constructor invocation.
- **Member object destructors:** Destructors of member objects are called before the class destructor.

```
#include <iostream>
using namespace std;
int count = 0;
class A
{
public:
A()
{
  count++;
    cout<<"Constructor is called for object: "<<count<<endl;
}
~A()
{
    cout<<"Destructor is called for object: "<<count<<endl;
    count--;
}
};
int main()
{
    A a,b;
    {
        A c;
        {
            A d;
        }
    }
    return 0;
}
```

```
Constructor is called for object: 1
Constructor is called for object: 2
Constructor is called for object: 3
Constructor is called for object: 4
Destructor is called for object: 4
Destructor is called for object: 3
Destructor is called for object: 2
Destructor is called for object: 1
```