**Q1.**

Static variables in C++ are variables that are declared with the static keyword. They are initialized once, when the program starts, and they keep their value until the program terminates. Static variables are shared by all objects of a class, and they can be accessed outside of the class using the class name and the **scope resolution operator ::**

Static functions in C++ are functions that are declared with the static keyword. They are not associated with any particular object of a class, and they can be called even without creating an object of the class. Static functions can access only static data members and other static member functions of the class.

#include <iostream>

using namespace std;

class *MyClass* {

    private:

*static* int count;

    public:

*static* void incrementCount() {

            count++;

        }

*static* int getCount() {

            return count;

        }

};

int *MyClass*::count = 0;

int main() {

*MyClass* data;

    data.incrementCount();

    data.incrementCount();

    cout << data.getCount() << endl;

    return 0;

}

**Q2.**

#include <iostream>

using namespace std;

class *classA*{

    public:

        void fun(int x){

            cout << x<<"^2 = "<<x\*x << endl;

        }

        void fun(float a, float b){

            cout << a << " + " << b << " = " << a+b << endl;

        }

};

int main(){

*classA* User;

    User.fun(10);

    User.fun(15.7,17.2);

    return 0;

}

**Q3.**

#include <iostream>

using namespace std;

class *myClass*{

    private:

        int x, y;

    public:

        void inputSave(int a, int b){

            x = a;

            y = b;

        }

*friend* void output(*myClass* mClass){

            cout << "x = " << mClass.x << endl;

            cout << "y = " << mClass.y << endl;

        }

};

int main(){

*myClass* User;

    User.inputSave(15,20);

    output(User);

    return 0;

}

**Q4.**

**Binary Operator**

#include <iostream>

using namespace std;

class *Complex* {

    public:

        int real;

        int img;

        Complex(int real, int img) {

            this->real = real;

            this->img = img;

        }

*Complex* operator+(*const* *Complex&* other) {

            return *Complex*(this->real + other.real, this->img + other.img);

        }

};

int main(){

    cout << endl;

*Complex* a(1.0, 2.0);

*Complex* b(3.0, 4.0);

*Complex* c = a + b;

    cout << a.real << " + " << a.img << "i" << endl;

    cout << b.real << " + " << b.img << "i" << endl;

    cout << "------" << endl;

    cout << c.real << " + " << c.img << "i" << endl;

    return 0;

}

**Alternative**

#include<iostream>

using namespace std;

class *sum*{

    public:

        int n1,n2;

        sum(){}

        sum(int p, int q){

            n1=p;

            n2=q;

        }

        void result(){

            cout << n1 << " + " << n2 << "i" << endl;

        }

*sum* operator + (*sum* s1){

*sum* s2;

            s2.n1= n1 + s1.n1;

            s2.n2= n2 + s1.n2;

            return s2;

        }

};

int main(){

*sum* s1(2,3);

*sum* s2(4,5);

*sum* s3;

    s1.result();

    s2.result();

    s3 = s1 + s2;

    cout << "------" << endl;

    s3.result();

    return 0;

}

**Unary Operator**

#include <iostream>

using namespace std;

class *Counter* {

    private:

        int count;

    public:

        Counter(int count = 0) {

            this->count = count;

        }

*Counter* operator++() {

            ++count;

            return \*this;

        }

*Counter* operator--() {

            --count;

            return \*this;

        }

        int getCount() *const* {

            return count;

        }

};

int main() {

*Counter* C;

    ++C;

    ++C;

    cout << C.getCount() << endl;

    --C;

    cout << C.getCount() << endl;

    return 0;

}

**Q5.**

**Here are the pros and cons of constructors in C++ in a table with 5 small points each:**

|  |  |  |
| --- | --- | --- |
| Aspect | Pros | Cons |
| Initialization | 1. Initialize object state | 1. Can be complex and error-prone |
| Multiple | 2. Multiple constructors with different parameter lists can be defined (overloading) | 2. Can lead to ambiguity if parameter lists are similar |
| Inheritance | 3. Constructors of base classes are called before derived class constructors | 3. Can result in complex construction order issues |
| Implicit Call | 4. Constructors are automatically called when objects are created | 4. Overhead if not needed for certain objects |
| Exception Handling | 5. Constructors can handle exceptions and indicate object initialization failure | 5. Exception handling adds complexity and can affect performance |

**Here are the pros and cons of destructors in C++ summarized in a table with five small points:**

|  |  |
| --- | --- |
| Destructors in C++ | |
| Pros | Cons |
| 1. Automatic Cleanup | 1. Lack of Control: Destructors are automatic and can't be explicitly called. |
| 2. Resource Management | 2. Potential Exceptions: Destructors should not throw exceptions to avoid undefined behavior. |
| 3. Simplifies Cleanup | 3. Inheritance Complexity: The order of execution in inheritance hierarchies can be complex. |
| 4. Allows RAII | 4. Can't be Overloaded: Unlike constructors, destructors cannot be overloaded. |
| 5. Supports Polymorphism | 5. No Return Type: Destructors have no return type, not even void. |

**Q6.**

#include <iostream>

using namespace std;

class *A*{

    public:

        A(){

            cout << "Default Constructor Invoked" << endl;

        }

};

class *B*{

    public:

        B(int x=0){

            cout << "Value of argument x = " << x << endl;

        }

};

int main(){

*A* a;

*B* b(10);

    return 0;

}

#include <iostream>

using namespace std;

class *MyClass* {

private:

    int x;

    int y;

public:

    MyClass(int x, int y) {

        this->x = x;

        this->y = y;

    }

*// Copy constructor*

    MyClass(*const* *MyClass&* other) {

        this->x = other.x;

        this->y = other.y;

    }

    int getX() *const* {

        return x;

    }

    int getY() *const* {

        return y;

    }

};

int main() {

*MyClass* obj1(1, 2);

*MyClass* obj2(obj1);

    cout << "Obj1 -> value of x: " << obj1.getX() << endl;

    cout << "Obj1 -> value of y: " << obj1.getY() << endl;

    cout << endl;

    cout << "Obj2 -> value of x: " << obj2.getX() << endl;

    cout << "Obj2 -> value of y: " << obj2.getY() << endl;

    return 0;

}