c++ Assignment

OOPs**-**Introduction

SUBMITTED BY: Rahul Kumar (85)

**Section 1: Classes, Objects, Constructors, Destructors**

**1. Understanding Classes and Objects (Student)**

#**include** <iostream>

#**include** <string>

**class** Student {

**public**:

std::string name;

**int** age;

**char** grade;

**void** displayDetails() **const** {

std::cout << "Name: " << name << std::endl;

std::cout << "Age: " << age << std::endl;

std::cout << "Grade: " << grade << std::endl;

}

};

**int** main() {

Student student1;

student1.name = "Alice";

student1.age = 20;

student1.grade = 'A';

student1.displayDetails();

**return** 0;

}

**Example Output:**

Name: Alice

Age: 20

Grade: A

**2. Constructors and Destructors (Car)**

#**include** <iostream>

#**include** <string>

**class** Car {

**public**:

std::string brand;

std::string model;

**int** year;

Car(std::string brand, std::string model, **int** year) : brand(brand), model(model), year(year) {

std::cout << "Car constructor called for " << brand << " " << model << std::endl;

}

~Car() {

std::cout << "Car destructor called for " << brand << " " << model << std::endl;

}

};

**int** main() {

Car car1("Toyota", "Camry", 2022);

{

Car car2("Honda", "Civic", 2023);

} *// car2 is destroyed at the end of this block*

**return** 0; *// car1 is destroyed at the end of main()*

}

**Example Output:**

Car constructor called for Toyota Camry

Car constructor called for Honda Civic

Car destructor called for Honda Civic

Car destructor called for Toyota Camry

**3. Dynamic Memory Allocation (Book)**

#**include** <iostream>

#**include** <string>

**class** Book {

**public**:

std::string title;

**double** price;

Book(std::string title, **double** price) : title(title), price(price) {}

**void** displayDetails() **const** {

std::cout << "Title: " << title << std::endl;

std::cout << "Price: " << price << std::endl;

}

};

**int** main() {

Book\* bookPtr = **new** Book("The C++ Programming Language", 49.99);

bookPtr->displayDetails();

**delete** bookPtr;

bookPtr = **nullptr**;

**return** 0;

}

**Example Output:**

Title: The C++ Programming Language

Price: 49.99

**10. Constructor Overloading (Person)**

#**include** <iostream>

#**include** <string>

**class** Person {

**public**:

std::string name;

**int** age;

Person() : name("Unknown"), age(0) {

std::cout << "Default constructor called" << std::endl;

}

Person(std::string name) : name(name), age(0) {

std::cout << "Constructor with name called" << std::endl;

}

Person(std::string name, **int** age) : name(name), age(age) {

std::cout << "Constructor with name and age called" << std::endl;

}

**void** displayDetails() **const** {

std::cout << "Name: " << name << std::endl;

std::cout << "Age: " << age << std::endl;

}

};

**int** main() {

Person person1;

person1.displayDetails();

Person person2("Bob");

person2.displayDetails();

Person person3("Charlie", 30);

person3.displayDetails();

**return** 0;

}

**Example Output:**

Default constructor called

Name: Unknown

Age: 0

Constructor with name called

Name: Bob

Age: 0

Constructor with name and age called

Name: Charlie

Age: 30

**Section 2: Function and Operator Overloading**

**4. Function Overloading (MathOperations)**

#**include** <iostream>

#**include** <string>

**class** MathOperations {

**public**:

**int** add(**int** a, **int** b) {

**return** a + b;

}

**double** add(**double** a, **double** b) {

**return** a + b;

}

std::string add(std::string a, std::string b) {

**return** a + b;

}

};

**int** main() {

MathOperations math;

std::cout << "Sum of integers: " << math.add(5, 10) << std::endl;

std::cout << "Sum of doubles: " << math.add(5.5, 3.2) << std::endl;

std::cout << "Concatenation of strings: " << math.add("Hello, ", "World!") << std::endl;

**return** 0;

}

**Example Output:**

Sum of integers: 15

Sum of doubles: 8.7

Concatenation of strings: Hello, World!

**7. Operator Overloading (+ Operator) (Complex)**

#**include** <iostream>

**class** Complex {

**public**:

**double** real;

**double** imaginary;

Complex(**double** real = 0.0, **double** imaginary = 0.0) : real(real), imaginary(imaginary) {}

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

**return** Complex(real + other.real, imaginary + other.imaginary);

}

**void** display() **const** {

std::cout << real << " + " << imaginary << "i" << std::endl;

}

};

**int** main() {

Complex c1(1.0, 2.0);

Complex c2(3.0, 4.0);

Complex c3 = c1 + c2;

c3.display();

**return** 0;

}

**Example Output:**

4 + 6i

**8. Operator Overloading (== Operator) (Point)**

#**include** <iostream>

**class** Point {

**public**:

**int** x, y;

Point(**int** x = 0, **int** y = 0) : x(x), y(y) {}

**bool** **operator**==(**const** Point& other) **const** {

**return** (x == other.x) && (y == other.y);

}

};

**int** main() {

Point p1(1, 2);

Point p2(1, 2);

Point p3(3, 4);

**if** (p1 == p2) {

std::cout << "p1 and p2 are equal" << std::endl;

} **else** {

std::cout << "p1 and p2 are not equal" << std::endl;

}

**if** (p1 == p3) {

std::cout << "p1 and p3 are equal" << std::endl;

} **else** {

std::cout << "p1 and p3 are not equal" << std::endl;

}

**return** 0;

}

**Example Output:**

p1 and p2 are equal

p1 and p3 are not equal

**9. Overloading Unary ++ Operator (Counter)**

#**include** <iostream>

**class** Counter {

**private**:

**int** value;

**public**:

Counter(**int** value = 0) : value(value) {}

*// Pre-increment*

Counter& **operator**++() {

++value;

**return** \***this**;

}

*// Post-increment*

Counter **operator**++(**int**) {

Counter temp = \***this**;

++value;

**return** temp;

}

**int** getValue() **const** {

**return** value;

}

};

**int** main() {

Counter c1(5);

std::cout << "Initial value: " << c1.getValue() << std::endl;

Counter c2 = c1++; *// Post-increment*

std::cout << "Post-increment value of c1: " << c1.getValue() << std::endl;

std::cout << "Value of c2 (post-increment): " << c2.getValue() << std::endl;

Counter c3 = ++c1; *// Pre-increment*

std::cout << "Pre-increment value of c1: " << c1.getValue() << std::endl;

std::cout << "Value of c3 (pre-increment): " << c3.getValue() << std::endl;

**return** 0;

}

**Example Output:**

Initial value: 5

Post-increment value of c1: 6

Value of c2 (post-increment): 5

Pre-increment value of c1: 7

Value of c3 (pre-increment): 7

**12. Operator Overloading (<< and >> for Input/Output Stream) (Time)**

#**include** <iostream>

**class** Time {

**public**:

**int** hours;

**int** minutes;

Time(**int** hours = 0, **int** minutes = 0) : hours(hours), minutes(minutes) {}

**friend** std::ostream& **operator**<<(std::ostream& os, **const** Time& time) {

os << time.hours << ":" << time.minutes;

**return** os;

}

**friend** std::istream& **operator**>>(std::istream& is, Time& time) {

std::cout << "Enter hours: ";

is >> time.hours;

std::cout << "Enter minutes: ";

is >> time.minutes;

**return** is;

}

};

**int** main() {

Time t1;

std::cin >> t1; *// Input time*

std::cout << "The time is: " << t1 << std::endl; *// Output time*

**return** 0;

}

**Example Interactions:**

* **Input:**
  + Enter hours: 10
  + Enter minutes: 30
* **Output:**
  + The time is: 10:30

**Section 3: Friend Functions and Pass by Value/Reference**

**5. Friend Function (Rectangle)**

#**include** <iostream>

**class** Rectangle {

**private**:

**int** length;

**int** width;

**public**:

Rectangle(**int** length = 0, **int** width = 0) : length(length), width(width) {}

**friend** **int** calculateArea(**const** Rectangle& rect);

};

**int** calculateArea(**const** Rectangle& rect) {

**return** rect.length \* rect.width;

}

**int** main() {

Rectangle rect(5, 10);

**int** area = calculateArea(rect);

std::cout << "Area: " << area << std::endl;

**return** 0;

}

**Example Output:**

Area: 50

**6. Pass by Value vs. Pass by Reference (Number)**

#**include** <iostream>

**class** Number {

**public**:

**int** value;

Number(**int** value = 0) : value(value) {}

**void** modifyValue(Number num) { *// Pass by value*

num.value = 100;

std::cout << "Inside modifyValue: " << num.value << std::endl;

}

**void** modifyReference(Number& num) { *// Pass by reference*

num.value = 200;

std::cout << "Inside modifyReference: " << num.value << std::endl;

}

};

**int** main() {

Number n(50);

std::cout << "Original value: " << n.value << std::endl;

n.modifyValue(n);

std::cout << "After modifyValue: " << n.value << std::endl;

n.modifyReference(n);

std::cout << "After modifyReference: " << n.value << std::endl;

**return** 0;

}

**Example Output:**

Original value: 50

Inside modifyValue: 100

After modifyValue: 50

Inside modifyReference: 200

After modifyReference: 200

**11. Friend Function with Two Classes (ClassA and ClassB)**

#**include** <iostream>

**class** ClassB; *// Forward declaration*

**class** ClassA {

**private**:

**int** valueA;

**public**:

ClassA(**int** valueA = 0) : valueA(valueA) {}

**friend** **int** sumObjects(**const** ClassA& a, **const** ClassB& b);

};

**class** ClassB {

**private**:

**int** valueB;

**public**:

ClassB(**int** valueB = 0) : valueB(valueB) {}

**friend** **int** sumObjects(**const** ClassA& a, **const** ClassB& b);

};

**int** sumObjects(**const** ClassA& a, **const** ClassB& b) {

**return** a.valueA + b.valueB;

}

**int** main() {

ClassA a(10);

ClassB b(20);

**int** sum = sumObjects(a, b);

std::cout << "Sum: " << sum << std::endl;

**return** 0;

}

**Example Output:**

Sum: 30