OOPs

19 August 2024 22:25

The **Object oriented programming** is a programming paradigm based on the concepts of Objects and classes.

Principles of OOPS

1. **Abstraction** is the ability to hide complex / important/ internal logic hidden from normal users. For e.g. In a vehicle, only seats, steering , chassis frame are visible and its mechanism, engines are well hidden in the frame.
2. **Encapsulation** is the ability to encapsule/ Wrap the data as a single unit.
3. **inheritance** allows a child class to inherit from the parent class.
4. **Polymorphism** is the ability of object to take on different forms or behave in different ways depending on the context they are used. e.g. if we want to learn driving then do you learn BMW, Maruti, Suzuki? No, we learn Car and thereafter we can manage to drive any car.

Q. Write a class Quadrilateral.

```
#include<iostream>
using namespace std;
class Quadrilateral
{
private:
    int side1, side2, side3, side4, angle;
public:
    // Constructor
    Quadrilateral(int side1, int side2, int side3, int side4, int angle){
        this->side1 = side1;
        this->side2 = side2;
        this->side3 = side3;
        this->side4 = side4;
        this->angle = angle;
    }
    bool is_rect(){
        if (side1 == side3 && side2 == side4 && angle == 90){
            cout << "true" << endl;
            return true;
        }
        cout << "False" << endl;
        return false;
    }
    bool is_square(){
        if (side1 == side2 && side2 == side3 && side3 == side4 && angle ==
90){
            cout << "true" << endl;
            return true;
        }
        cout << "False" << endl;
```

```
        return false;
    }
};
int main() {
    Quadrilateral rectangle(40, 50, 40, 50, 90);
    cout << rectangle.is_rect() << endl;
    cout << rectangle.is_square() << endl;
    Quadrilateral square(50, 50, 50, 50, 90);
    cout << square.is_rect() << endl;
    cout << square.is_square() << endl;
    return 0;
}
```

Pointer to an Object

```
#include<iostream>
using namespace std;
class Rectangle{
public:
    int length, breadth;
    int perimeter(){
        return 2 * (length + breadth);
    }
    int area(){
        return length * breadth;
    }
};
int main() {
    Rectangle *r = new Rectangle(); // Allocate memory for the Rectangle
object in heap memory
    r->length = 6; // -> is arrow operator to visit length var space at
address r, so basically a dereferencing operator
    r->breadth = 4;
    cout << "Perimeter: " << r->perimeter() << endl;
    cout << "Area: " << r->area() << endl;
    delete r; // Free the allocated memory
    return 0;
}
```

Constructors

Constructor is something that is called during the initialization of an object.

- Constructor is a member function of a class
- It will have same name as class name
- It will not have return type
- Its should be public
- It can be declared as private also in some cases
- It is called when object is created
- It is used for initializing an object
- It can be overloaded
- If its not defined then class will have a default constructor
- Constructor can take default arguments

Destructors

Destructor is an instance member function that is invoked automatically whenever an objects is going to be deleted.

```
class Rectangle{
private:
    int length;
    int width;
public:
    // Constructor: Called when an object of the class is created
    Rectangle(int l, int w) {
        length = l;
        width = w;
    }
    // Destructor: Called when an object of the class is destroyed
    ~Rectangle() {
        // Clean-up code, if needed, would go here
        cout << "Rectangle object is being deleted" << endl;
    }
};
```

Types of Constructors:

1. **Default / Built-in Constructors** are the constructors that are provided by the compiler that takes no arguments or all arguments having default values.

```
class Rectangle {
private:
    int length;
    int width;
public:
    // Default constructor
    Rectangle() {
        length = 0;
        width = 0;
    }
};
```

2. **Parameterized Constructors** are the user defined constructors that takes one or more arguments.

```
class Rectangle {
private:
    int length;
    int width;
public:
    // Parameterized constructor
    Rectangle(int l, int w) {
        length = l;
        width = w;
    }
};
```

3. **Non - parameterized Constructors** are the type of constructors as the name suggest do

not take any arguments.

```
class Rectangle {
private:
    int length;
    int width;
public:
    // Non-parameterized (default) constructor
    Rectangle() {
        length = 1; // Default length
        width = 1;  // Default width
    }
};
```

4. Copy Constructers is a constructor that takes a reference to an object of the same class as a parameter.

```
class Rectangle {
private:
    int length;
    int width;
public:
    // Copy constructor
    Rectangle(const Rectangle &rect) {
        length = rect.length;
        width = rect.width;
    }
};
```

Types of Data Copying

1. **Shallow Copy** is type of copy where any change to copied variable may lead to change in Original passed Data.

```
#include<iostream>
using namespace std;
class Shallow {
public:
    int* data;
    Shallow(int value) {
        data = new int(value);
    }
    // Shallow copy constructor
    Shallow(const Shallow &source) : data(source.data) {
        cout << "Shallow copy constructor - shallow copy made" << endl;
    }
    void displayData() const {
        cout << "Data: " << *data << endl;
    }
    // Destructor is called, leading to potential double deletion
    ~Shallow() {
        delete data;
        cout << "Destructor freeing data" << endl;
    }
};
```

```
int main() {
    Shallow obj1(10);
    Shallow obj2 = obj1; // Shallow copy
    obj1.displayData();
    obj2.displayData();
    return 0;
}
```

2. **Deep Copy** is type of copy where any change to copied variable will not lead to change in Original passed Data.

```
#include<iostream>
using namespace std;
class Deep {
public:
    int* data;
    Deep(int value) {
        data = new int(value);
    }
    // Deep copy constructor
    Deep(const Deep &source) {
        data = new int(*source.data);
        cout << "Deep copy constructor - deep copy made" << endl;
    }
    void displayData() const {
        cout << "Data: " << *data << endl;
    }
    ~Deep() {
        delete data;
        cout << "Destructor freeing data" << endl;
    }
};
int main() {
    Deep obj1(10);
    Deep obj2 = obj1; // Deep copy
    obj1.displayData();
    obj2.displayData();
    return 0; // Destructor is called, but no issues as each object manages
its own resource
}
```

All types of Member Functions

1. **Constructors** - called when object is created
2. **Accessors** - used for knowing the value of data members
3. **Mutators** - used for changing value of data member
4. **Facilitator** - actual functions of class
5. **Enquiry** - used for checking if an object satisfies some condition
6. **Destructor** - used for releasing resources used by object

Scope Resolution Operator

:: is a Scope Resolution Operator and it is used to access a global variable when there is a local variable with the same name, or to define a function outside a class or namespace. It helps in distinguishing between different scopes and namespaces in your code.

```
class Rectangle{
private:
    int length;
    int width;
public:
    Rectangle(int l, int w){
        length = l;
        width = w;
    }
    void area();
    void change_length(int l){ // this will automatically become inline
Function so its better to write it inside with scope resolution operator
        length = l;
    }
};
void Rectangle::area(){
    printf("%d \n", length * width);
}
```

Function inside Classes `will` automatically become inline Function so its better to write it inside with scope resolution operator.