Mastering OOP in C++: From Fundamentals to Advanced Applications

Author: Abdelrahman Ramadan

1. Introduction

Object-Oriented Programming (OOP) is a paradigm in software design that revolves around objects and classes.
In C++, OOP empowers developers to model real-world entities and relationships in an organized and reusable manner.
This article explores OOP concepts and advanced C++ features through practical examples and implementations.

2. Core Concepts of OOP

2.1 Full Example Demonstrating Core OOP Concepts

This example combines abstraction, encapsulation, access modifiers, inheritance, and polymorphism in one unified Employee system.

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

// ===== Abstraction + Encapsulation + Access Modifiers =====
class Employee {
private:
    string name;
    int id;
    double salary;

protected:
    string role;
```

```
public:
    Employee(string n, int i, double s) {
        name = n;
        id = i;
        salary = s;
    }

    string getName() { return name; }
    void setName(string n) { name = n; }

    double getSalary() { return salary; }
    void setSalary(double s) {
        if (s >= 0)
            salary = s;
    }

    virtual void work() {
        cout << name << " is working as a general employee.\n";
    }

    void showInfo() {
        cout << "Name: " << name << ", ID: " << id << ", Salary: " << salary << ", Role: "
<< role << endl;
    }
};

// ===== Inheritance + Overriding =====
class Developer : public Employee {
private:
    string programmingLanguage;
public:
    Developer(string n, int i, double s, string lang) : Employee(n, i, s) {
        role = "Developer";
        programmingLanguage = lang;
    }

    void work() override {
        cout << getName() << " is coding in " << programmingLanguage << ".\n";
    }
```

```
};

class Manager : public Employee {
private:
    int teamSize;
public:
    Manager(string n, int i, double s, int t) : Employee(n, i, s) {
        role = "Manager";
        teamSize = t;
    }

    void work() override {
        cout << getName() << " is managing a team of " << teamSize << " people.\n";
    }
};

int main() {
    Developer dev("Alice", 101, 5000, "C++");
    Manager mgr("Bob", 102, 7000, 5);

    Employee* staff[2] = { &dev, &mgr };

    for (int i = 0; i < 2; i++) {
        staff[i]->showInfo();
        staff[i]->work();
        cout << endl;
    }

    return 0;
}
```

2. Static Members and the 'this' Pointer

Static members belong to the class rather than any individual object. They are shared across all instances. The 'this' pointer refers to the current object.

```
class Counter {
private:
    static int count;
    int id;
public:
    Counter() {
        id = ++count;
    }
    void showID() {
        cout << "Object ID: " << this->id << endl;
    }
};
int Counter::count = 0;
```

3. Constructors and Destructors with Memory Allocation

Constructors are used to initialize objects. If memory is allocated using 'new' in a constructor, it must be released with 'delete' in the destructor.

```
class MemoryDemo {
private:
    int* data;
public:
    MemoryDemo(int size) {
        data = new int[size]; // Constructor allocates memory
    }
    ~MemoryDemo() {
        delete[] data; // Destructor frees memory
    }
};
```

4. Constructor Overloading and Overriding

Overloading allows multiple constructors with different parameters. Overriding happens when a derived class redefines a base class method.

```
class Person {
public:
    Person() {
        cout << "Default constructor" << endl;
    }
    Person(string name) {
        cout << "Name: " << name << endl;
    }
};

class Student : public Person {
public:
    Student() : Person("Student") {
        cout << "Derived class constructor" << endl;
    }
};
```

5. Operator Overloading

Operators can be redefined to work with user-defined types like classes.

```
class Complex {
private:
    int real, imag;
public:
    Complex(int r, int i) : real(r), imag(i) {}

    Complex operator+(const Complex& other) {
        return Complex(real + other.real, imag + other.imag);
    }

    void display() {
        cout << real << " + " << imag << "i" << endl;
    }
};
```

6. Exception Handling

C++ supports error handling using try, catch, and throw.

```
void divide(int a, int b) {
    if (b == 0) throw "Division by zero!";
    cout << "Result: " << a / b << endl;
}

int main() {
    try {
        divide(10, 0);
    } catch (const char* msg) {
        cout << "Error: " << msg << endl;
    }
}
```

7. File Input/Output (Student Example)

Reading and writing data to a file using fstream.

```
class Student {
private:
    string name;
    int age;
public:
    void input() {
        cout << "Enter name: "; cin >> name;
        cout << "Enter age: "; cin >> age;
    }
    void saveToFile() {
        ofstream out("students.txt", ios::app);
        out << name << " " << age << endl;
        out.close();
    }

    void readFromFile() {
        ifstream in("students.txt");
        string n; int a;
        while (in >> n >> a) {
            cout << "Name: " << n << ", Age: " << a << endl;
        }
        in.close();
    }
};
```

8. Bitwise Analysis Program

Convert an integer to binary and analyze the number of 1s and 0s.

```
class BitAnalyzer {
private:
    int number;
public:
    void input() {
        cout << "Enter a number: ";
        cin >> number;
    }

    void analyze() {
        int ones = 0, zeros = 0;
        int n = number;
        string binary = "";

        while (n > 0) {
            int bit = n % 2;
            binary = to_string(bit) + binary;
            if (bit == 1) ones++;
            else zeros++;
            n /= 2;
        }

        cout << "Binary: " << binary << endl;
        cout << "1s: " << ones << ", 0s: " << zeros << endl;
        float total = ones + zeros;
        cout << "1s %: " << (ones / total) * 100 << "%, 0s %: " <<
(zeros / total) * 100 << "%" << endl;
```

```
        }
};
```