**OOP FINAL**

**1. Encapsulation**

**Definition**: Wrapping data (variables) and code (methods) together into a single unit (class) and restricting direct access to some of the object’s components.

**Purpose**: Improves data security and prevents unintended interference.

## ****2. Inheritance****

**Definition**: The process by which one class (child/derived) inherits the properties and behaviours (methods) of another class (parent/base).

**Purpose**: Promotes code reusability and logical hierarchy.

## ****3. Polymorphism****

**Definition**: The ability of different objects to respond in their own way to the same function call.

There are two types:

* **Compile-time (Function Overloading)**
* #include <iostream>
* using namespace std;
* class Calculator {
* public:
* int add(int a, int b) {
* return a + b;
* }
* double add(double a, double b) {
* return a + b;
* }
* int add(int a, int b, int c) {
* return a + b + c;
* }
* };
* int main() {
* Calculator calc;
* cout << "Add 2 ints: " << calc.add(2, 3) << endl; // add(int, int)
* cout << "Add 2 doubles: " << calc.add(2.5, 3.1) << endl; // add(double, double)
* cout << "Add 3 ints: " << calc.add(1, 2, 3) << endl; // add(int, int, int)
* return 0;
* }
* **Run-time (Function Overriding with Virtual Functions)**
* #include <iostream>
* using namespace std;
* class Animal {
* public:
* virtual void speak() {
* cout << "Animal makes a sound" << endl;
* }
* };
* class Dog : public Animal {
* public:
* void speak() override {
* cout << "Dog barks" << endl;
* }
* };
* class Cat : public Animal {
* public:
* void speak() override {
* cout << "Cat meows" << endl;
* }
* };
* int main() {
* Animal\* a1 = new Dog(); // Base pointer to derived object
* Animal\* a2 = new Cat();
* a1->speak(); // Dog barks (determined at runtime)
* a2->speak(); // Cat meows
* delete a1;
* delete a2;
* return 0;
* }

## ****4. Abstraction****

**Definition**: Hiding complex implementation details and showing only the essential features of the object.

**Purpose**: Reduces complexity and increases efficiency. You can do it by just making a function in parent class a pure virtual function like this virtual void draw() = 0;

**Types of Constructors (with full names):**

| **Constructor Type** | **Description** |
| --- | --- |
| 1. Default Constructor | No parameters; used to create objects with default values |
| 2. Parameterized Constructor | Takes arguments; used to initialize objects with specific values |
| 3. Copy Constructor | Initializes an object by copying another object of the same class |
| 4. Dynamic/Move Constructor (C++11) | Moves resources instead of copying (not always needed) |
| 5. Constructor with Default Arguments | Combines default and parameterized behaviors |
| 6. Delegating Constructor (C++11) | One constructor calls another within the same class |
| 7. Explicit Constructor (C++11) | Prevents implicit conversions during object creation |

**EXAMPLE CODE**

#include <iostream>

using namespace std;

class Person {

private:

string name;

int age;

public:

// 1. Default Constructor

Person() {

name = "Unknown";

age = 0;

cout << "Default Constructor called\n";

}

// 2. Parameterized Constructor

Person(string n, int a) {

name = n;

age = a;

cout << "Parameterized Constructor called\n";

}

// 3. Copy Constructor

Person(const Person &p) {

name = p.name;

age = p.age;

cout << "Copy Constructor called\n";

}

// 4. Constructor with Default Arguments

Person(string n = "Anonymous") {

name = n;

age = 18;

cout << "Constructor with Default Arguments called\n";

}

void show() {

cout << "Name: " << name << ", Age: " << age << endl;

}

};

int main() {

Person p1; // Default constructor

p1.show();

Person p2("Alice", 25); // Parameterized constructor

p2.show();

Person p3 = p2; // Copy constructor

p3.show();

Person p4("John"); // Constructor with default age

p4.show();

Person p5; // Calls default constructor again

p5.show();

return 0;

}

#include <iostream>

#include <cstring> // for strcpy, strlen

using namespace std;

class StudentProfile {

private:

char\* name;

int\* age;

float\* gpa;

char\* department;

public:

// Default Constructor

StudentProfile() {

name = new char[1]{ '\0' };

age = new int(0);

gpa = new float(0.0f);

department = new char[1]{ '\0' };

cout << "Default Constructor called\n";

}

// Parameterized Constructor

StudentProfile(const char\* n, int a, float g, const char\* d) {

name = new char[strlen(n) + 1];

strcpy(name, n);

age = new int(a);

gpa = new float(g);

department = new char[strlen(d) + 1];

strcpy(department, d);

cout << "Parameterized Constructor called\n";

}

// Copy Constructor (Deep Copy)

StudentProfile(const StudentProfile& other) {

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

age = new int(\*other.age);

gpa = new float(\*other.gpa);

department = new char[strlen(other.department) + 1];

strcpy(department, other.department);

cout << "Copy Constructor called\n";

}

**CONSTRUCTORS OF POINTERS**

#include <iostream>

#include <cstring> // for strcpy, strlen

using namespace std;

class StudentProfile {

private:

char\* name;

int\* age;

float\* gpa; // Pointer to array of GPAs

int gpaCount; // Number of GPAs

char\* department;

public:

// Default Constructor

StudentProfile() {

name = new char[1]{ '\0' };

age = new int(0);

gpaCount = 0;

gpa = nullptr;

department = new char[1]{ '\0' };

cout << "Default Constructor called\n";

}

// Parameterized Constructor

StudentProfile(const char\* n, int a, const float\* g, int count, const char\* d) {

name = new char[strlen(n) + 1];

strcpy(name, n);

age = new int(a);

gpaCount = count;

gpa = new float[gpaCount];

for (int i = 0; i < gpaCount; i++) {

gpa[i] = g[i];

}

department = new char[strlen(d) + 1];

strcpy(department, d);

cout << "Parameterized Constructor called\n";

}

// Copy Constructor (Deep Copy)

StudentProfile(const StudentProfile& other) {

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

age = new int(\*other.age);

gpaCount = other.gpaCount;

gpa = new float[gpaCount];

for (int i = 0; i < gpaCount; i++) {

gpa[i] = other.gpa[i];

}

department = new char[strlen(other.department) + 1];

strcpy(department, other.department);

cout << "Copy Constructor called\n";

}

// Copy Assignment Operator

StudentProfile& operator=(const StudentProfile& other) {

if (this == &other)

return \*this;

delete[] name;

delete age;

delete[] gpa;

delete[] department;

name = new char[strlen(other.name) + 1];

strcpy(name, other.name);

age = new int(\*other.age);

gpaCount = other.gpaCount;

gpa = new float[gpaCount];

for (int i = 0; i < gpaCount; i++) {

gpa[i] = other.gpa[i];

}

department = new char[strlen(other.department) + 1];

strcpy(department, other.department);

cout << "Copy Assignment Operator called\n";

return \*this;

}

// Move Constructor

StudentProfile(StudentProfile&& other) noexcept {

name = other.name;

age = other.age;

gpa = other.gpa;

gpaCount = other.gpaCount;

department = other.department;

// Nullify source

other.name = nullptr;

other.age = nullptr;

other.gpa = nullptr;

other.department = nullptr;

other.gpaCount = 0;

cout << "Move Constructor called\n";

}

// Move Assignment Operator

StudentProfile& operator=(StudentProfile&& other) noexcept {

if (this == &other)

return \*this;

delete[] name;

delete age;

delete[] gpa;

delete[] department;

name = other.name;

age = other.age;

gpa = other.gpa;

gpaCount = other.gpaCount;

department = other.department;

other.name = nullptr;

other.age = nullptr;

other.gpa = nullptr;

other.department = nullptr;

other.gpaCount = 0;

cout << "Move Assignment Operator called\n";

return \*this;

}

// Destructor

~StudentProfile() {

delete[] name;

delete age;

delete[] gpa;

delete[] department;

cout << "Destructor called\n";

}

// Display Function

void display() const {

cout << "Name: " << name << "\nAge: " << \*age << "\nGPA(s): ";

for (int i = 0; i < gpaCount; i++) {

cout << gpa[i] << (i < gpaCount - 1 ? ", " : "");

}

cout << "\nDepartment: " << department << endl;

}

};

int main() {

float gpas[] = {3.5f, 3.7f, 3.9f};

cout << "\n--- s1: Parameterized Constructor ---\n";

StudentProfile s1("Alice", 20, gpas, 3, "Computer Science");

s1.display();

cout << "\n--- s2: Copy Constructor ---\n";

StudentProfile s2 = s1;

s2.display();

cout << "\n--- s3: Copy Assignment ---\n";

StudentProfile s3;

s3 = s1;

s3.display();

cout << "\n--- s4: Move Constructor ---\n";

StudentProfile s4 = std::move(s1);

s4.display();

cout << "\n--- s5: Move Assignment ---\n";

StudentProfile s5;

s5 = std::move(s2);

s5.display();

return 0;

}

### **Example of Shallow Copy:**

#include <iostream>

using namespace std;

class Shallow {

private:

int\* data;

public:

Shallow(int value) {

data = new int(value);

}

// Default copy constructor (shallow copy)

void show() {

cout << "Value: " << \*data << " at address " << data << endl;

}

~Shallow() {

delete data;

}

};

int main() {

Shallow obj1(10);

Shallow obj2 = obj1; // Shallow copy

obj1.show();

obj2.show();

// Modifying obj2 will affect obj1 because they share the same memory

return 0; // Crash likely here: both try to delete the same memory

}

### ❌ Problem:

* Both obj1 and obj2 point to the **same memory**.
* When destructors are called, **double deletion** occurs, leading to **undefined behavior or crash**.

### **Example of Deep Copy:**

#include <iostream>

using namespace std;

class Deep {

private:

int\* data;

public:

Deep(int value) {

data = new int(value);

}

// Custom copy constructor (deep copy)

Deep(const Deep& other) {

data = new int(\*other.data);

}

void show() {

cout << "Value: " << \*data << " at address " << data << endl;

}

~Deep() {

delete data;

}

};

int main() {

Deep obj1(20);

Deep obj2 = obj1; // Deep copy

obj1.show();

obj2.show(); // Different addresses!

return 0; // No crash, both destructors safe

}