**Pass\_By\_Value\_Program:**

#include <iostream>

using namespace std;

class Point{

public:double x,y;

};

void offsetPoint(Point p,double x,double y){

p.x += x;

p.y += y;

}

int main()

{

Point p;

p.x=3.0;

p.y=4.0;

offsetPoint(p,1.0,2.0);

cout<<"("<<p.x<<","<<p.y<<")";

}

**Question:**

Write a class Student with a default constructor that initializes the student's name to "Unknown" and age to 0. Add a method display to print the student's details.

Solution:

#include <iostream>

#include <string>

class Student {

std::string name;

int age;

public:

// Default constructor

Student() : name("Unknown"), age(0) {}

// Method to display student's details

void display() const {

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

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

}

};

int main() {

Student student;

student.display();

return 0;

}

**Single Inheritance:**

#include <iostream>

using namespace std;

class Account{

public:

float salary=60000;

};

class programmer:public Account{

public:

float bonus=5000;

};

int main(void)

{

programmer p1;

cout<<"salary: "<<p1.salary<<endl;

cout<<"bonus: "<<p1.bonus<<endl;

return 0;

}

**Code:**

#include <iostream>

#include <string>

using namespace std;

class Person {

private:

string name;

int age;

public:

// Constructor with validation (optional)

Person(const string& n, int a) {

if (a < 0) {

throw invalid\_argument("Age cannot be negative.");

}

name = n;

age = a;

}

virtual ~Person() {} // Virtual destructor for proper cleanup

// Accessors (getters)

string getName() const { return name; }

int getAge() const { return age; }

// Mutators (setters) with validation (optional)

void setName(const string& n) {

if (n.empty()) {

throw invalid\_argument("Name cannot be empty.");

}

name = n;

}

void setAge(int a) {

if (a < 0) {

throw invalid\_argument("Age cannot be negative.");

}

age = a;

}

// Virtual function for details (can be overridden)

virtual void getDetails() const {

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

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

}

};

class Student : public Person {

private:

int studentId;

string major;

public:

Student(const string& n, int a, int id, const string& m) : Person(n, a), studentId(id), major(m) {}

// Accessors

int getStudentId() const { return studentId; }

string getMajor() const { return major; }

// Mutators

void setMajor(const string& m) {

if (m.empty()) {

throw invalid\_argument("Major cannot be empty.");

}

major = m;

}

// Override getDetails to include student-specific information

void getDetails() const override {

Person::getDetails(); // Call base class getDetails

cout << "Student ID: " << studentId << endl;

cout << "Major: " << major << endl;

}

};

class Faculty : public Person {

private:

string department;

int employeeId;

public:

Faculty(const string& n, int a, const string& d, int id) : Person(n, a), department(d), employeeId(id) {}

// Accessors

string getDepartment() const { return department; }

int getEmployeeId() const { return employeeId; }

// Mutators

void setDepartment(const string& d) {

if (d.empty()) {

throw invalid\_argument("Department cannot be empty.");

}

department = d;

}

// Override getDetails to include faculty-specific information

void getDetails() const override {

Person::getDetails(); // Call base class getDetails

cout << "Department: " << department << endl;

cout << "Employee ID: " << employeeId << endl;

}

};

int main() {

Person p1("John Doe", 30); // Create a Person object

Student s1("Jane Smith", 22, 12345, "Computer Science"); // Create a Student object

s1.getDetails();

Faculty f1("Alice Jones", 45, "Mathematics", 54321); // Create a Faculty object

f1.getDetails();

return 0;

}