**Single Inheritance:**

#include <iostream>

using namespace std;

class Account{

public:

float salary = 6000;

};

class Programmer:public Account{

public:

float bonus = 5050;

};

int main (void){

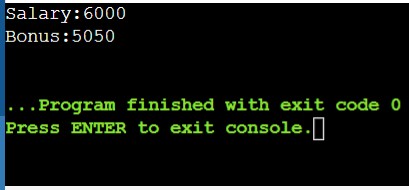
Programmer p1;

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

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

}

**Output:**



**Multilevel Inheritance:**

#include <iostream>

using namespace std;

class Animal{

public:

void eat(){

cout<<"Eating...."<<endl;

}

};

class Dog:public Animal

{

public:

void bark(){

cout<<"Barking......"<<endl;

}

};

class BabyDog:public Dog

{

public:

void weep(){

cout<<"Weeping.....";

}

};

int main(void){

BabyDog d1;

d1.eat();

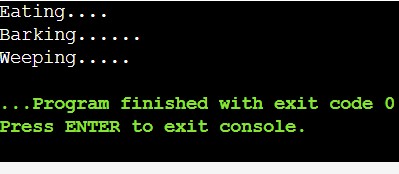
d1.bark();

d1.weep();

return 0;

}

**Output:**



**Multiple Inheritance:**

#include <iostream>

using namespace std;

class A{

protected:

int a;

public:

void get\_a(int n)

{

a = n;

}

};

class B{

protected:

int b;

public:

void get\_b(int n)

{

b = n;

}

};

class C:public A , public B

{

public:

void display(){

std::cout<<"The value of a is:"<<a<<std::endl;

std::cout<<"The value of b is:"<<b<<std::endl;

std::cout<<"Addition of a and b is:"<<a+b;

}

};

int main()

{

C c;

c.get\_a(10);

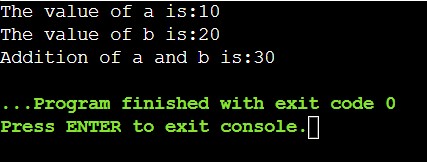
c.get\_b(20);

c.display();

return 0;

}

**Output:**



**Hybrid Inheritance:**

#include <iostream>

using namespace std;

class A{

protected:

int a;

public:

void get\_a()

{

std::cout<<"Enter the value of 'a':"<<std::endl;

cin>>a;

}

};

class B : public A{

protected:

int b;

public:

void get\_b()

{

std::cout<<"Enter the value of 'b':"<<std::endl;

cin>>b;

}

};

class C{

protected:

int c;

public:

void get\_c()

{

std::cout<<"Enter the value of c is:"<<std::endl;

cin>>c;

}

};

class D : public B,public C{

protected:

int d;

public:

void mul()

{

get\_a();

get\_b();

get\_c();

std::cout<<"Multiplication of a,b,c is:"<<a\*b\*c<<std::endl;

cin>>c;

}

};

int main()

{

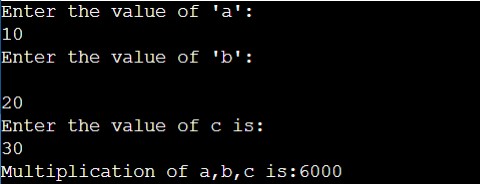
D d;

d.mul();

return 0;

}

**Output:**



**Hierarchical inheritance:**

#include <iostream>

using namespace std;

class Person {

protected:

string name;

int age;

public:

void set\_details(string n, int a) {

name = n;

age = a;

}

void display\_details() {

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

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

}

};

class Student : public Person {

private:

int studentID;

public:

void set\_student\_details(string n, int a, int id) {

set\_details(n, a);

studentID = id;

}

void display\_student() {

display\_details();

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

}

};

class Teacher : public Person {

private:

string subject;

public:

void set\_teacher\_details(string n, int a, string sub) {

set\_details(n, a);

subject = sub;

}

void display\_teacher() {

display\_details();

cout << "Subject: " << subject << endl;

}

};

int main() {

Student s;

Teacher t;

s.set\_student\_details("kavya", 10, 2345);

t.set\_teacher\_details("Jagadeesh", 40, "Mathematics");

cout << "Student Details:" << endl;

s.display\_student();

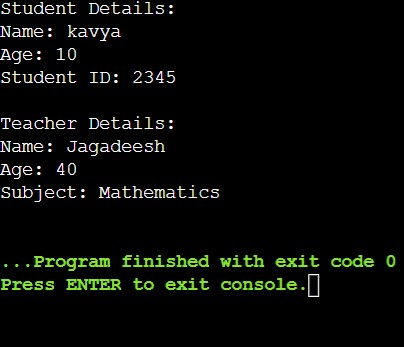
cout << "\nTeacher Details:" << endl;

t.display\_teacher();

return 0;

}

**Output:**



**Scenario:**

**Design a class hierarchy to represent various entities in a university system.**

**Base Class (Person):**

**Data members: name (string), age (int)**

**Member functions: getDetails(), a virtual function to print basic person details**

**Derived Class (Student): (Single Inheritance)**

**Inherits from Person**

**Data members: studentId (int), major (string)**

**Member functions:**

**setMajor(string) to set the student's major**

**getMajor() to retrieve the major**

**Override getDetails() to include student-specific information**

**Derived Class (Faculty): (Single Inheritance)**

**Inherits from Person**

**Data members: department (string), employeeId (int)**

**Member functions:**

**setDepartment(string) to set the faculty member's department**

**getDepartment() to retrieve the department**

**Override getDetails() to include faculty-specific information**

**Derived Class (TeachingAssistant): (Multilevel Inheritance)**

**Inherits from Student (inherits indirectly from Person as well)**

**Data member: coursesTeaching (array/vector of strings)**

**Member functions:**

**setCoursesTeaching(string[]) to set the courses the TA is teaching**

**getCoursesTeaching() to retrieve the list of courses**

**Override getDetails() to include TA-specific information (e.g., courses)**

**Derived Class (ResearchAssistant): (Hierarchical Inheritance)**

**Inherits from Person (separate inheritance from Student)**

**Data members: researchArea (string), supervisor (string)**

**Member functions:**

**setResearchArea(string) to set the research area**

**getResearchArea() to retrieve the research area**

**setSupervisor(string) to set the research supervisor**

**getSupervisor() to retrieve the supervisor**

**Override getDetails() to include RA-specific information**

**Derived Class (GraduateStudentTA): (Hybrid Inheritance)**

**Inherits from both Student and TeachingAssistant (combines functionality)**

**Might have additional data members or functions specific to graduate student Tas**

#include <iostream>

#include <vector>

#include <string>

using namespace std;

// Base class Person

class Person {

protected:

string name;

int age;

public:

Person(const string& name, int age) : name(name), age(age) {}

virtual void getDetails() const {

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

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

}

};

// Derived class Student

class Student : public Person {

protected:

int studentId;

string major;

public:

Student(const string& name, int age, int studentId)

: Person(name, age), studentId(studentId) {}

void setMajor(const string& major) {

this->major = major;

}

string getMajor() const {

return major;

}

void getDetails() const override {

Person::getDetails();

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

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

}

};

// Derived class Faculty

class Faculty : public Person {

protected:

string department;

int employeeId;

public:

Faculty(const string& name, int age, int employeeId)

: Person(name, age), employeeId(employeeId) {}

void setDepartment(const string& department) {

this->department = department;

}

string getDepartment() const {

return department;

}

void getDetails() const override {

Person::getDetails();

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

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

}

};

// Derived class TeachingAssistant

class TeachingAssistant : public Person {

protected:

vector<string> coursesTeaching;

public:

TeachingAssistant(const string& name, int age)

: Person(name, age) {}

void setCoursesTeaching(const vector<string>& courses) {

coursesTeaching = courses;

}

vector<string> getCoursesTeaching() const {

return coursesTeaching;

}

void getDetails() const override {

Person::getDetails();

cout << "Teaching Assistant" << endl;

cout << "Courses Teaching:";

for (const auto& course : coursesTeaching) {

cout << " " << course;

}

cout << endl;

}

};

// Derived class ResearchAssistant

class ResearchAssistant : public Person {

protected:

string researchArea;

string supervisor;

public:

ResearchAssistant(const string& name, int age)

: Person(name, age) {}

void setResearchArea(const string& researchArea) {

this->researchArea = researchArea;

}

string getResearchArea() const {

return researchArea;

}

void setSupervisor(const string& supervisor) {

this->supervisor = supervisor;

}

string getSpervisor() const {

return supervisor;

}

void getDetails() const override {

Person::getDetails();

cout << "Research Assistant" << endl;

cout << "Research Area: " << researchArea << endl;

cout << "Supervisor: " << supervisor << endl;

}

};

// Derived class GraduateStudentTA

class GraduateStudentTA : public Student, public TeachingAssistant {

public:

GraduateStudentTA(const string& name, int age, int studentId)

: Student(name, age, studentId), TeachingAssistant(name, age) {}

void getDetails() const override {

Student::getDetails();

TeachingAssistant::getDetails();

}

};

int main() {

// Example usage

Student student("Kavya", 29, 1);

student.setMajor("Devops");

Faculty faculty("Jagadeesh", 26, 2);

faculty.setDepartment("Chemistry");

TeachingAssistant ta("Jahnavi", 25);

ta.setCoursesTeaching({"Fullstack java"});

ResearchAssistant ra("Nani", 28);

ra.setResearchArea("Big Data");

ra.setSupervisor("Er. ");

GraduateStudentTA gradTA("Vamsi", 24, 1);

gradTA.Student::setMajor("Mathematics");

gradTA.TeachingAssistant::setCoursesTeaching({"Advance Graphs", "Trignometry"});

// Printing details

cout << "Student Details:" << endl;

student.getDetails();

cout << endl << "Faculty Details:" << endl;

faculty.getDetails();

cout << endl << "Teaching Assistant Details:" << endl;

ta.getDetails();

cout << endl << "Research Assistant Details:" << endl;

ra.getDetails();

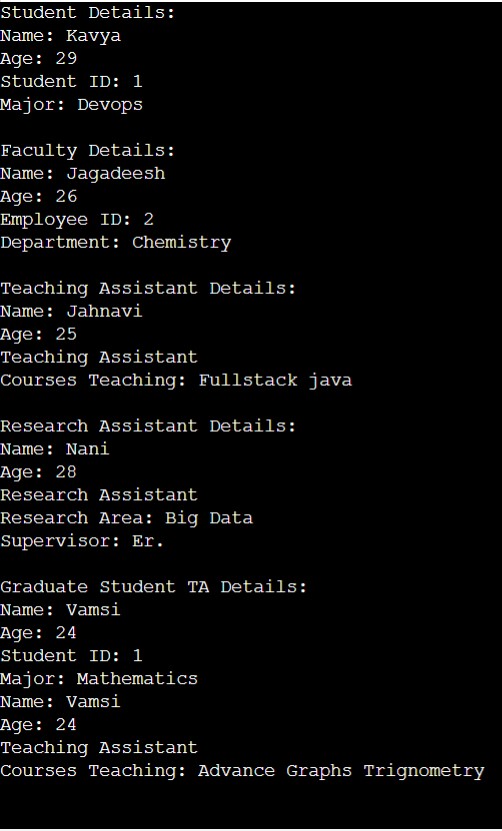
cout << endl << "Graduate Student TA Details:" << endl;

gradTA.getDetails();

return 0;

}

**Output:**



#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();

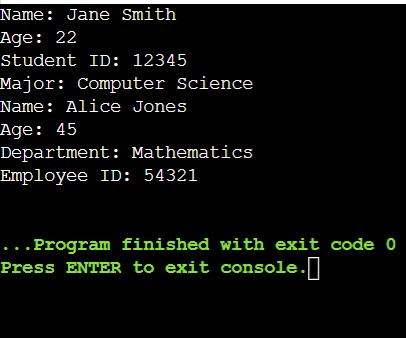
Faculty f1("Alice Jones", 45, "Mathematics", 54321); // Create a Faculty object

f1.getDetails();

return 0;

}

**Output:**



**Ambiguity in Inheritance:**

#include <iostream>

using namespace std;

class A

{

public:

void display()

{

std::cout<<"Class A"<<std::endl;

}

};

class B

{

public:

void display()

{

std::cout<<"Class B"<<std::endl;

}

};

class C:public A,public B

{

Public:

void view()

{

A::display();

B::display();

}

};

int main()

{

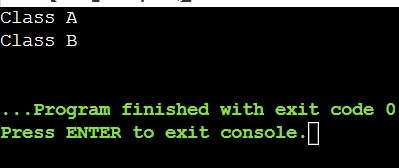
C c;

c.display();

return 0;

}

**Output:**



**Scenario:**

**Imagine you're developing a university management system. You have a base class named Person that stores basic information about individuals associated with the university, such as:**

**name (string)**

**id (int)**

**Question:**

**Design a class hierarchy using inheritance to represent different types of people within the university. Consider the following categories:**

**Student: Inherits from Person and has additional attributes like:**

**major (string)**

**gpa (double)**

**A method calculateSemesterGPA(vector<double> grades) that takes a vector of grades (doubles) and calculates the semester GPA.**

**Faculty: Inherits from Person and has additional attributes like:**

**department (string)**

**title (string) - e.g., "Professor", "Lecturer"**

**A method teachCourse(string courseName) that simulates assigning a faculty member to teach a specific course.**

**Additional Considerations:**

**You can introduce further derived classes if you think of more specific roles within the university (e.g., Staff, Administrator).**

**Think about access specifiers (public, private, protected) for member variables and methods in the base and derived classes.**

**Consider virtual functions (especially in the context of polymorphism) if there's common functionality that might have different implementations in derived classes.**

**Guiding Tips:**

**Focus on code clarity and maintainability.**

**Use meaningful variable and method names.**

**Add comments to explain your design choices.**

**Test your code to ensure it works as expected.**

#include <iostream>

#include <vector>

#include <string>

#include <numeric>

using namespace std;

// Base class Person

class Person {

private:

string name;

int id;

public:

Person(string name, int id) : name(name), id(id) {}

string getName() const {

return name;

}

int getId() const {

return id;

}

void setName(string name) {

this->name = name;

}

void setId(int id) {

this->id = id;

}

void displayInfo() const {

cout << "Name: " << name << ", ID: " << id << endl;

}

};

// Derived class Student from Person

class Student : public Person {

private:

string major;

int gpa; // GPA stored as an integer percentage

public:

Student(string name, int id, string major, int gpa)

: Person(name, id), major(major), gpa(gpa) {}

string getMajor() const {

return major;

}

int getGPA() const {

return gpa;

}

void setMajor(string major) {

this->major = major;

}

void setGPA(int gpa) {

this->gpa = gpa;

}

int calculateSemesterGPA(const vector<int>& grades) const {

if (grades.empty()) return 0;

int total = accumulate(grades.begin(), grades.end(), 0);

return total / grades.size();

}

void displayInfo() const {

Person::displayInfo();

cout << "Major: " << major << ", GPA: " << gpa << "%" << endl;

}

};

// Derived class Faculty from Person

class Faculty : public Person {

private:

string department;

string title;

public:

Faculty(string name, int id, string department, string title)

: Person(name, id), department(department), title(title) {}

string getDepartment() const {

return department;

}

string getTitle() const {

return title;

}

void setDepartment(string department) {

this->department = department;

}

void setTitle(string title) {

this->title = title;

}

void teachCourse(string courseName) const {

cout << title << " " << getName() << " is now teaching " << courseName << endl;

}

void displayInfo() const {

Person::displayInfo();

cout << "Department: " << department << ", Title: " << title << endl;

}

};

int main() {

// Creating a Student object

Student student("Kavya", 1, "Computer Science", 89);

student.displayInfo();

// Calculating semester GPA

vector<int> grades = {76, 77, 78, 79};

cout << "Semester GPA: " << student.calculateSemesterGPA(grades) << "%" << endl;

// Creating a Faculty object

Faculty faculty("Vamsi", 2000, "CSE", "Professor");

faculty.displayInfo();

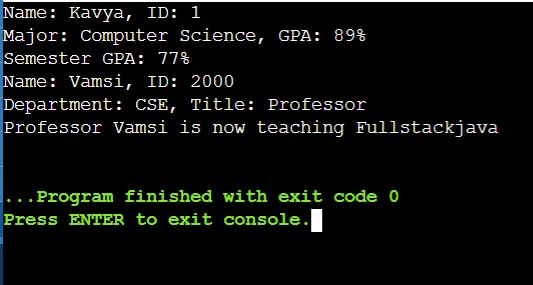
// Assigning a course to teach

faculty.teachCourse("Fullstackjava");

return 0;

}

**Output:**



**Function Overloading:**

#include <iostream>

using namespace std;

class Cal{

public:

static int add(int a, int b){

return a + b;

}

static int add(int a, int b, int c)

{

return a + b + c;

}

static int sub(int a, int b){

return a - b;

}

static int sub(int a, int b, int c){

return a - b - c;

}

static int mul(int a, int b){

return a \* b;

}

static int mul(int a, int b, int c){

return a \* b \* c;

}

static int divide(int a, int b){

return a / b;

}

static int divide(int a, int b, int c){

return a / b / c;

}

};

int main(void){

Cal c;

cout<<c.add(10,20)<<endl;

cout<<c.add(12,20,23)<<endl;

cout<<c.sub(10,20)<<endl;

cout<<c.sub(12,30,40)<<endl;

cout<<c.mul(10,20)<<endl;

cout<<c.mul(27,12,4)<<endl;

cout<<c.divide(20,10)<<endl;

cout<<c.divide(20,30,10)<<endl;

return 0;

}

**Output:**

