**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 <string>

using namespace std;

class Person { // Base class Person

protected:

string name;

int age;

public:

Person(string name, int age) : name(name), age(age) {} // Virtual function

virtual void getDetails() {

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

}

};

class Student : public Person { // Derived class Student

protected:

int studentId;

string major;

public:

Student(string name, int age, int studentId) : Person(name, age), studentId(studentId) {}

void setMajor(string major) {

major = major; //Assign directly

} string getMajor() {

return major; }

void getDetails() override { // Override getDetails to include student-specific information

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

Person::getDetails();

cout << "Student ID: " << studentId << ", Major: " << major << endl;

}

};

class Faculty : public Person { // Derived class Faculty

protected:

string department;

int employeeId;

public:

Faculty(string name, int age, int employeeId) : Person(name, age), employeeId(employeeId) {}

void setDepartment(string department) {

department = department;// Assign directly

} string getDepartment() {

return department;

}

void getDetails() override { // Override getDetails to include faculty-specific information

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

Person::getDetails();

cout << "Employee ID: " << employeeId << ", Department: " << department << endl;

}

};

class TeachingAssistant : public Student { // (Multilevel Inheritance)

protected:

string coursesTeaching;

public:

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

void setCoursesTeaching(string courses) {

coursesTeaching = courses;

}

string getCoursesTeaching() {

return coursesTeaching;

}

void getDetails() override { // Override getDetails to include TA-specific information

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

Student::getDetails();

cout << "Courses Teaching: " << coursesTeaching << endl;

}

};

int main() {

Student s("Manogn", 24, 544);

s.setMajor("Computer Science and Engineering");

Faculty f("Rahul", 25, 543);

f.setDepartment("Computer Science and Engineering");

TeachingAssistant ta("sri ram", 24, 523 );

ta.setMajor("Computer Science and Engineering");

ta.setCoursesTeaching("CN, CG");

Person\* people[] = {&s, &f, &ta};

for (auto person : people) {

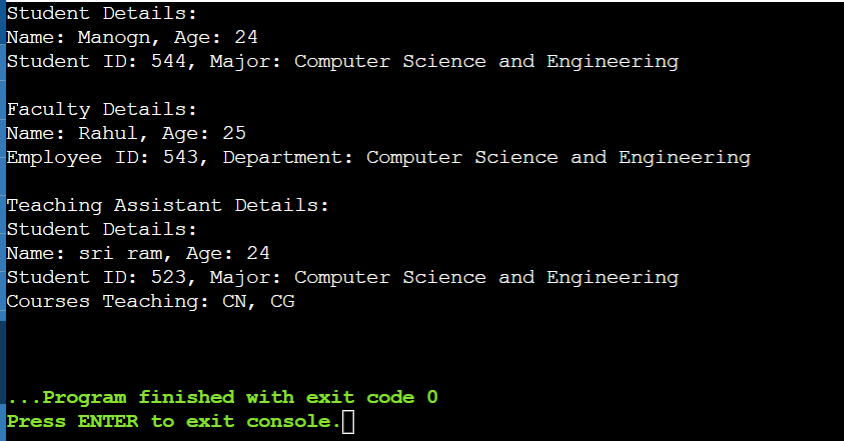
person->getDetails();

cout << endl;

} return 0;

}

**Output**



**Flowchart for above program:**

# 

**Write the single inheritance code using public, private, protected access specifiers**

1. **Public Inheritance from Public Base**

#include <iostream>

using namespace std;

class Account {

public:

float salary = 6000;

};

class Programmer : public Account {

public:

float bonus = 5000;

};

int main(void) {

Programmer p1;

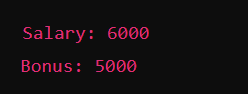
cout << "Salary: " << p1.salary << endl;

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

return 0;

}

**Ouput**

****

1. **Protected Inheritance from Public Base**

#include <iostream>

using namespace std;

class Account {

public:

float salary = 6000;

};

class Programmer : protected Account {

public:

float bonus = 5000;

};

int main(void) {

Programmer p1;

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

// cout << "Salary: " << p1.salary << endl; // This will cause a compilation error

return 0;

}

**Output**



Explanation: salary is public in Account but becomes protected in Programmer.

1. **Private Inheritance from Public Base**

#include <iostream>

using namespace std;

class Account {

public:

float salary = 6000;

};

class Programmer : private Account {

public:

float bonus = 5000;

};

int main(void) {

Programmer p1;

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

// cout << "Salary: " << p1.salary << endl; // This will cause a compilation error

return 0;

}

**Output**

****

Explanation: salary is public in Account but becomes private in Programmer.

1. **Public Inheritance from Protected Base**

#include <iostream>

using namespace std;

class Account {

protected:

float salary = 6000;

};

class Programmer : public Account {

public:

float bonus = 5000;

};

int main(void) {

Programmer p1;

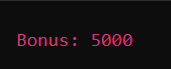
cout << "Bonus: " << p1.bonus << endl;

// cout << "Salary: " << p1.salary << endl; // This will cause a compilation error

return 0;

}

**Output**

****

Explanation: salary is protected in Account and remains protected in Programmer.

1. **Protected Inheritance from Protected Base**

#include <iostream>

using namespace std;

class Account {

protected:

float salary = 6000;

};

class Programmer : protected Account {

public:

float bonus = 5000;

};

int main(void) {

Programmer p1;

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

// cout << "Salary: " << p1.salary << endl; // This will cause a compilation error

return 0;

}

**Output**

****

Explanation: salary is protected in Account and remains protected in Programmer.

1. **Private Inheritance from Protected Base**

#include <iostream>

using namespace std;

class Account {

protected:

float salary = 6000;

};

class Programmer : private Account {

public:

float bonus = 5000;

};

int main(void) {

Programmer p1;

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

// cout << "Salary: " << p1.salary << endl; // This will cause a compilation error

return 0;

}

**Output**

****

Explanation: salary is protected in Account but becomes private in Programmer.

1. **Public Inheritance from Private Base**

#include <iostream>

using namespace std;

class Account {

private:

float salary = 6000;

};

class Programmer : public Account {

public:

float bonus = 5000;

};

int main(void) {

Programmer p1;

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

// cout << "Salary: " << p1.salary << endl; // This will cause a compilation error

return 0;

}

**Output**

****

Explanation: salary is private in Account and is not accessible in Programmer.

1. **Protected Inheritance from Private Base**

#include <iostream>

using namespace std;

class Account {

private:

float salary = 6000;

};

class Programmer : protected Account {

public:

float bonus = 5000;

};

int main(void) {

Programmer p1;

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

// cout << "Salary: " << p1.salary << endl; // This will cause a compilation error

return 0;

}

**Output**

****

Explanation: salary is private in Account and is not accessible in Programmer.

1. **Private Inheritance from Private Base**

#include <iostream>

using namespace std;

class Account {

private:

float salary = 6000;

};

class Programmer : private Account {

public:

float bonus = 5000;

};

int main(void) {

Programmer p1;

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

// cout << "Salary: " << p1.salary << endl; // This will cause a compilation error

return 0;

}

**Output**

****

Explanation: salary is private in Account and is not accessible in Programmer.

**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 <numeric> // For std::accumulate

#include <string>

using namespace std;

// Base class Person

class Person {

protected:

string name;

int id;

public:

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

void getDetails() {

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

}

};

// Derived class Student

class Student : public Person {

private:

string major;

double gpa;

public:

Student(string name, int id, string major) : Person(name, id), major(major), gpa(0.0) {}

void calculateSemesterGPA(const vector<double>& grades) {

if (!grades.empty()) {

gpa = accumulate(grades.begin(), grades.end(), 0.0) / grades.size();

}

}

double getGPA() {

return gpa;

}

void getDetails() {

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

Person::getDetails();

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

}

};

// Derived class Faculty

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) {}

void teachCourse(const string& courseName) {

cout << title << " " << name << " is teaching course: " << courseName << endl;

}

void getDetails() {

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

Person::getDetails();

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

}

};

int main() {

// Creating a Student object and setting details

Student student("Manogn", 1001, "Computer Science");

student.calculateSemesterGPA({3.5, 3.7, 4.0, 3.8});

student.getDetails();

cout << endl;

// Creating a Faculty object and setting details

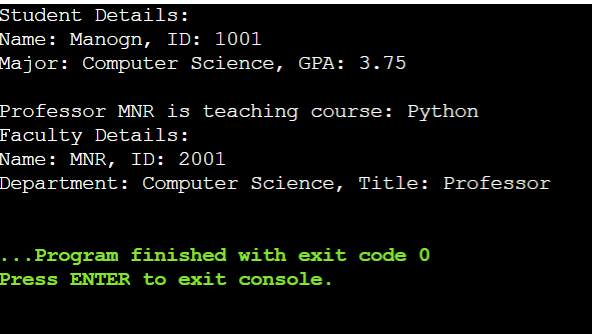
Faculty faculty("MNR", 2001, "Computer Science", "Professor");

faculty.teachCourse("Python");

faculty.getDetails();

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;

}

};

int main(void) {

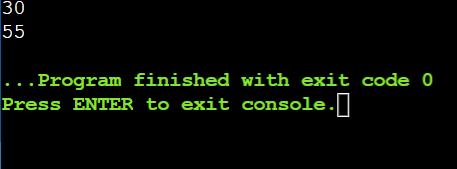
Cal C;

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

cout << C.add(12,20,23);

return 0;

}

**Output**  
  


**Function Overloading with add, sub, multiply, divide**   
  
#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**

