C++concepts-26/6

1. 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;

}

2.

#include <iostream>

using namespace std;

class Animal {

public:

void eat() {

std::cout<<"Eating..."<<std::endl;

}

};

class Dog: public Animal

{

public:

void bark(){

std::cout<<"Barking..."<<std::endl;

}

};

class BabyDog: public Dog{

public:

void weep() {

std::cout<<"weeping..."<<std::endl;

}

};

int main(void) {

BabyDog d1;

d1.eat();

d1.bark();

d1.weep();

return 0;

}

3.

#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;

}

4. #include <iostream>

using namespace std;

// Superclass

class Animal {

public:

void eat() {

cout << "This animal is eating." << endl;

}

};

// Subclass 1

class Dog : public Animal {

public:

void bark() {

cout << "The dog barks." << endl;

}

};

// Subclass 2

class Cat : public Animal {

public:

void meow() {

cout << "The cat meows." << endl;

}

};

int main() {

Dog dog;

Cat cat;

dog.eat(); // Inherited from Animal

dog.bark(); // Specific to Dog

cat.eat(); // Inherited from Animal

cat.meow(); // Specific to Cat

return 0;

}

5 . example :

#include <iostream>

using namespace std;

// Base class

class Animal {

public:

void eat() {

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

}

};

// Derived class 1

class Dog : public Animal {

public:

void bark() {

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

}

};

// Derived class 2

class Cat : public Animal {

public:

void meow() {

cout << "Meowing..." << endl;

}

};

int main() {

Dog d;

Cat c;

d.eat(); // Calling function from base class

d.bark(); // Calling function from derived class Dog

c.eat(); // Calling function from base class

c.meow(); // Calling function from derived class Cat

return 0;

}

6. Sir given task……

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

Code:

#include <iostream>

#include <vector>

#include <string>

using namespace std;

// Base class

class Person {

protected:

string name;

int age;

public:

Person(string n, int a) : name(n), age(a) {}

virtual void getDetails() {

cout << "Name: " << name << "\nAge: " << age << endl;

}

};

// Derived class from Person (Single Inheritance)

class Student : public Person {

protected:

int studentId;

string major;

public:

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

void setMajor(string m) {

major = m;

}

string getMajor() {

return major;

}

void getDetails() override {

Person::getDetails();

cout << "Student ID: " << studentId << "\nMajor: " << major << endl;

}

};

// Derived class from Person (Single Inheritance)

class Faculty : public Person {

protected:

string department;

int employeeId;

public:

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

void setDepartment(string d) {

department = d;

}

string getDepartment() {

return department;

}

void getDetails() override {

Person::getDetails();

cout << "Employee ID: " << employeeId << "\nDepartment: " << department << endl;

}

};

// Derived class from Student (Multilevel Inheritance)

class TeachingAssistant : public Student {

protected:

vector<string> coursesTeaching;

public:

TeachingAssistant(string n, int a, int id, string m, vector<string> courses)

: Student(n, a, id, m), coursesTeaching(courses) {}

void setCoursesTeaching(vector<string> courses) {

coursesTeaching = courses;

}

vector<string> getCoursesTeaching() {

return coursesTeaching;

}

void getDetails() override {

Student::getDetails();

cout << "Courses Teaching: ";

for (const string& course : coursesTeaching) {

cout << course << " ";

}

cout << endl;

}

};

// Derived class from Person (Hierarchical Inheritance)

class ResearchAssistant : public Person {

protected:

string researchArea;

string supervisor;

public:

ResearchAssistant(string n, int a, string ra, string s)

: Person(n, a), researchArea(ra), supervisor(s) {}

void setResearchArea(string ra) {

researchArea = ra;

}

string getResearchArea() {

return researchArea;

}

void setSupervisor(string s) {

supervisor = s;

}

string getSupervisor() {

return supervisor;

}

void getDetails() override {

Person::getDetails();

cout << "Research Area: " << researchArea << "\nSupervisor: " << supervisor << endl;

}

};

// Derived class from both Student and TeachingAssistant (Hybrid Inheritance)

class GraduateStudentTA : public TeachingAssistant {

public:

GraduateStudentTA(string n, int a, int id, string m, vector<string> courses)

: TeachingAssistant(n, a, id, m, courses) {}

void getDetails() override {

TeachingAssistant::getDetails();

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

}

};

int main() {

Student s("John Doe", 20, 12345, "Computer Science");

Faculty f("Dr. Smith", 45, 6789, "Engineering");

TeachingAssistant ta("Alice Johnson", 25, 54321, "Mathematics", {"Calculus", "Algebra"});

ResearchAssistant ra("Bob Brown", 30, "Machine Learning", "Dr. White");

GraduateStudentTA gta("Charlie Black", 27, 98765, "Physics", {"Quantum Mechanics", "Thermodynamics"});

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

s.getDetails();

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

f.getDetails();

cout << "\nTeaching Assistant Details:" << endl;

ta.getDetails();

cout << "\nResearch Assistant Details:" << endl;

ra.getDetails();

cout << "\nGraduate Student TA Details:" << endl;

gta.getDetails();

return 0;

}

Sirs 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;

}

Scenario 2:

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.

Solution for this:

#include <iostream>

#include <vector>

#include <string>

using namespace std;

// Base class Person

class Person {

protected: // Protected access specifier allows derived classes to access these members

string name;

int id;

public:

// Constructor to initialize name and id

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

// Virtual destructor for proper cleanup of derived class objects

virtual ~Person() {}

// Method to display basic information

void displayInfo() {

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

}

};

// Derived class Student from Person

class Student : public Person {

private:

string major;

double gpa;

public:

// Constructor to initialize name, id, major, and gpa

Student(string n, int i, string m, double g) : Person(n, i), major(m), gpa(g) {}

// Method to calculate the semester GPA

double calculateSemesterGPA(vector<double> grades) {

double total = 0.0;

for(double grade : grades) {

total += grade;

}

gpa = total / grades.size();

return gpa;

}

// Override displayInfo to include major and gpa

void displayInfo() override {

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

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

}

};

// Derived class Faculty from Person

class Faculty : public Person {

private:

string department;

string title;

public:

// Constructor to initialize name, id, department, and title

Faculty(string n, int i, string d, string t) : Person(n, i), department(d), title(t) {}

// Method to simulate teaching a course

void teachCourse(string courseName) {

cout << title << " " << name << " is teaching " << courseName << " in the " << department << " department." << endl;

}

// Override displayInfo to include department and title

void displayInfo() override {

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

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

}

};

// Main function to test the classes

int main() {

// Create a Student object

Student student("Alice", 123, "Computer Science", 3.5);

student.displayInfo();

vector<double> grades = {3.7, 3.8, 3.6};

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

// Create a Faculty object

Faculty faculty("Dr. Smith", 456, "Engineering", "Professor");

faculty.displayInfo();

faculty.teachCourse("Physics 101");

return 0;

}