End-Semester Project Report(48)

Team: Mokshitha(CS23B2048), Sumit kumar(CS23b2008),jatin(CS23B2045)

December 7, 2024

Post Mid-Semester Topics Project:

Inheritance,Polymorphism,class templates,exception handling

Inheritance

Code 1

```
class User {
protected:
    string id;          //
    string name;        //
    string email;       //
    string phoneNumber; /
```

Figure 1: Base class of code which will be inherited further

Inference:

1. This is the base class named user which will be further inherited by the derived classes and the data inside the user is protected hence if we inherit the user as public it remains protected inside the derived class

```
class Alumni:public User {
private:
    int graduationYear;
    string degree;
    string department;
    string currentPosition;
    string company;
    string location;
```

```
class Administrator:public User {
private:
    string role;
```

Derived class 1

```
The class alumni is the first derived class of user class
```

Inference:

1. This is the derived class of name Alumni inheriting user as public.
2. So the user class data members and functions can be used by Alumni .
3. As the data members of user class are protected the become protected in the Alumni class

Derived class 2

Code

```
The administrator class is the second derived class of administartor
```

Inference:
the inferences:

1. second derived class is administrator which inherits the base class user as public

```
class Alumni:public User {
private:
    int graduationYear;
    string degree;
    string department;
    string currentPosition;
    string company;
    string location;

public:
    // Constructor
    Alumni() {}
    Alumni(string id, string name, int graduationYear, string degree,
           string department, string email, string phoneNumber,
           string currentPosition, string company, string location)
        : User(id, name, email, phoneNumber), graduationYear(graduationYear),
          degree(degree), department(department), currentPosition(currentPosition),
          company(company), location(location) {}
```

The Above pic of alumni class shows how it is using and inheriting the user class

In the above screenshot we can see how the alumni is using baseclass

Inference:

1. so all the data members id,name,email,phone number belongs to base class and this alumni class is inheriting the class publicly and using the datamembers inside its class
2. Attribute Inheritance: The Alumni class inherits common user attributes like id, name, email, and phoneNumber from the User class, allowing it to manage user-specific information.
3. Constructor Inheritance: The Alumni class calls the User constructor to initialize inherited attributes while adding its own specific fields like graduationYear and currentPosition.

Polymorphism

Code 1

Code:

the next page contains the polymorphism

```cpp
// Overriding displayInfo (Polymorphism)
void displayInfo() const override {
    User::displayInfo(); // Display common user info
    cout << "Role: " << role << endl;
}
```

```cpp
// Overriding displayInfo (Polymorphism)
void displayInfo() const override {
    User::displayInfo();
    cout << "Graduation Year: " << graduationYear << ", Degree: " << degree
         << ", Department: " << department << ", Current Position: " << currentPosition
         << ", Company: " << company << ", Location: " << location << endl;
}
```

Inference:

1. Polymorphism allows objects of different classes to be treated as instances of a common superclass, enabling a single method to behave differently based on the object's actual class. It is achieved through method overriding (run-time) and method overloading (compile-time).

2. Polymorphism: The Alumni class overrides the displayInfo() method from User to include alumni-specific details, showcasing polymorphism.

Code 2

`the second pic of this page shows overriding`

inference

1. The virtual void displayInfo() method enables run-time polymorphism, allowing derived classes to override it and provide specific behavior.

2. It ensures the correct method is called based on the actual object type, even when accessed through a base class reference or pointer..

```
try{
     int optn = stoi(choice);
      switch (optn) {
          case 1:
              handleViewAllAlumni(db);
              break;
          case 2:
              if (isAdmin(currentUser)) {
                  handleAddAlumni(dynamic_cast<Administrator*>(currentUser), db);
              }
              else {
                  cout<<"Access denied. Admin privileges required."<<endl;
                  }

              break;
          case 3:
              handleSearchAlumniByID(db);
              break;
          case 4:
              handleSearchAlumniByName(db);
              break;
          case 5:
              if (isAdmin(currentUser)) {
                  handleUpdateAlumniInfo(dynamic_cast<Administrator*>(currentUser), db);
              }
              else {
                  cout<<"Access denied. Admin privileges required."<<endl;
                  }
              break;
          case 6:
              if (isAdmin(currentUser)) {
                  handleRemoveAlumni(dynamic cast<Administrator*>(currentUser), db);
```

Exception handling

code:

`the above pic is the code for the exception handling`

Inference:

1. The stoi(choice) conversion may throw exceptions (like std::invalid argument) if the input is invalid, so proper exception handling is needed to prevent crashes.

2. The program checks for admin privileges before performing certain actions, and uses dynamic cast to cast currentUser to Administrator; if the cast fails, null pointers should be handled to avoid undefined behavior.

exception handling(catch)

```
    catch (const invalid_argument& e) {
        cout << "Error: Invalid input, please enter a valid number." << endl;
    } catch (const out_of_range& e) {
        cout << "Error: Input is out of range." << endl;
    }
    // To prevent the program from exiting prematurely due to input errors
    if (running) {
        displayMenu(currentUser);  // Display the menu again
    }
}
```

the above pic is the continuous code after try

Inference:

1. The catch blocks handle specific exceptions: invalid argument for invalid input (non-numeric input) and out of range for input that exceeds the acceptable range, displaying appropriate error messages.

2. After handling the exceptions, the if (running) condition ensures that the program does not exit prematurely and prompts the user to enter valid input again by re-displaying the menu.