Virtual functions and exception handling(16-08-24)

Author: ThanhTH10
Date: 16/08/2024

1. Design a ticket booking system for various modes of transport(bus,train,flight) where each mode of transport has a different fare calculation method. The system should use runtime polymorphism to dynamically calculate and display the fare based on the type of ticket

```cpp
#include <iostream>
using namespace std;

class Transport
{
public:
    Transport() {}
    virtual double calculateFare(float distance) = 0;
    virtual ~Transport() {}
};
class Bus : public Transport
{
public:
    double calculateFare(float distance) override
    {
        return distance * 100;
    }
};
class Train : public Transport
{
public:
    double calculateFare(float distance) override
    {
        return distance * 200;
    }
};
Flight : public Transport
{
public:
    double calculateFare(float distance) override
    {
        return distance * 500;
    }
};
void bookingTicket(Transport *transport, float distance)
{
    float fare = transport->calculateFare(distance);
    cout << "Total fare for your journey: $" << fare << endl;
}
int main(int argc, char const *argv[])
{
    float distance;
    cout << "Enter the distance of your journey: ";
    cin >> distance;
    Transport *transport;
    int choice;
    do
    {
        cout << "Transport Mode: \n"
```

```
            << "1. Bus | 100$/km\n"
            << "2. Train | 200$/km\n"
            << "3. Flight | 500$/km\n"
            << "Choose a transport mode: ";
        cin >> choice;
        switch (choice)
        {
        case 1:
            transport = new Bus();
            break;
        case 2:
            transport = new Train();
            break;
        case 3:
            transport = new Flight();
            break;
        default:
            cout << "Invalid choice! Please try again." << endl;
        }
    } while (choice < 1 || choice > 3);
    bookingTicket(transport, distance);
    delete transport;
    return 0;
}
```

Output:

```
Enter the distance of your journey: 123
Transport Mode:
1. Bus | 100$/km
2. Train | 200$/km
3. Flight | 500$/km
Choose a transport mode: 5
Invalid choice! Please try again.
Transport Mode:
1. Bus | 100$/km
2. Train | 200$/km
3. Flight | 500$/km
Choose a transport mode: 3
Total fare for your journey: $61500
```

2. Write a program to create classes for following
Create class called Employee, with the following members
Data Members:
a) age of int type
b) name of string type
e) emp_id of integer
c) email_id of string type
d) contact no of string type
f) salary of float type.
Member functions:
1. Constructor for initialization.
2. Print for Printing the data members . 3. Calculate Salary 4. Destructor.

Derive a class called Permanent Employee from Employee with following members.
Data members:
basic of float type
da of float type

it of float type
gross_salary of float type ,
net_salary of float type
Member function:
1. Constructor for initialization.
2. Print for Printing the data members .
3. Calculate Salary
Note : (DA = 52% of Basic and IT = 30% of the gross salary).
gross salary = basic + da;
net_salary = (basic + da) - it; 4. Destructor

Derive a class called Contract Employee with following members.
Data Members:
A.) wage of float type. (amount per hour)
B). total hours of float type
C). total wage of float type.
Member Functions:
1. Constructor for initialization.
2. Print for Printing the data members.
3. Calculate Salary
Note : salary=wage*total hours
Use runtime polymorphism, to calculate the salary and also Print.
If we store the Permanent employee object in Employee pointer calculations should be done
according to Permanent employee and print also according to this class.
If we store the Contract Employee object in in Employee pointer calculations should be done
according to Contract employee and Print also according to this class.

```
#include <iostream>
using namespace std;

class Employee
{
protected:
    int age;
    string name;
    int emp_id;
    string email;
    string contact_no;
    float salary;
public:
    Employee(int age, string name, int emp_id, string email, string contact_no)
    {
        this->age = age;
        this->name = name;
        this->emp_id = emp_id;
        this->email = email;
        this->contact_no = contact_no;
    }
    virtual void display()
    {
        cout << "Id: " << emp_id << "\t| Name: " << name << "\t| Age: " << age << "\t| Email:
" << email
             << "\t| Contact: " << contact_no << "\t| True Salary: " << salary << endl;
    }
```

```
    virtual void calculateSalary() = 0;
    virtual ~Employee() {}
};
class PermanentEmployee : public Employee
{
protected:
    float basic;
    float da;
    float it;
    float gross_salary;
    float net_salary;
public:
    PermanentEmployee(int age, string name, int emp_id, string email_id, string contact_no,
float basic)
        : Employee(age, name, emp_id, email_id, contact_no)
    {
        this->basic = basic;
    }
    void display() override
    {
        Employee::display();
        cout << "Basic: " << basic << "\t\t| DA: " << da << "\t| IT: " << it << "\t\t\t| Gross
Salary: " << gross_salary
             << "\t| Net Salary: " << net_salary << endl;
    }
    void calculateSalary() override
    {
        da = 0.52 * basic;
        gross_salary = basic + da;
        it = 0.30 * gross_salary;
        net_salary = gross_salary - it;
        salary = net_salary;
    }
    ~PermanentEmployee() {}
};
class ContractEmployee : public Employee
{
protected:
    float wage;
    float total_hours;
    float total_wage;
public:
    ContractEmployee(int age, string name, int emp_id, string email_id, string contact_no,
float wage, float total_hours)
        : Employee(age, name, emp_id, email_id, contact_no)
    {
        this->wage = wage;
        this->total_hours = total_hours;
    }
    void display() override
    {
        Employee::display();
        cout << "Wage: " << wage << "\t\t| Total hours: " << total_hours << "\t\t\t\t| Total
wage: " << total_wage << endl;
    }
    void calculateSalary() override
    {
        total_wage = wage * total_hours;
        salary = total_wage;
    }
    ~ContractEmployee() {}
```

```
};
int main(int argc, char const *argv[])
{
    Employee *emp1 = new PermanentEmployee(30, "Thanh", 101, "Thanh@example.com", "1234567890",
50000);
    Employee *emp2 = new ContractEmployee(25, "Nhu", 102, "Nhu@example.com", "9876543210", 20,
40);
    emp1->calculateSalary();
    emp1->display();
    cout << endl;
    emp2->calculateSalary();
    emp2->display();
    cout << endl;
    delete emp1;
    delete emp2;
    return 0;
}
```

Output:

```
Id: 101 | Name: Thanh   | Age: 30       | Email: Thanh@example.com      | Contact: 1234567890   | True Salary: 53200
Basic: 50000           | DA: 26000     | IT: 22800                     | Gross Salary: 76000   | Net Salary: 53200

Id: 102 | Name: Nhu    | Age: 25       | Email: Nhu@example.com        | Contact: 9876543210   | True Salary: 800
Wage: 20               | Total hours: 40                               | Total wage: 800
```

3.consider that the base class stack is available.It does not take care of situations such as overflow or underflow.Enhance this class to MyStack which raises an exception whenever overflow or underflow error occurs.

```
#include <iostream>
#include <stdexcept>

class Stack
{
protected:
    int *arr;
    int top;
    int max_size;
public:
    Stack(int size) : max_size(size), top(-1)
    {
        arr = new int[max_size];
    }
    virtual void push(int value)
    {
        arr[++top] = value; // error overflow maybe here
    }
    virtual int pop()
    {
        return arr[top--]; // error underflow maybe here
    }
    int peek() const
    {
        return arr[top];
    }
    bool isFull() const
    {
        return top == max_size - 1;
    }

    bool isEmpty() const
```

```
    {
        return top == -1;
    }
    ~Stack()
    {
        delete[] arr;
    }
};
// Stack upgrade class
class MyStack : public Stack
{
public:
    MyStack(int size) : Stack(size) {}
    void push(int value) override
    {
        if (isFull())
        {
            throw std::overflow_error("Stack Overflow: Stack full.");
        }
        Stack::push(value);
    }
    int pop() override
    {
        if (isEmpty())
        {
            throw std::underflow_error("Stack Underflow: Stack empty.");
        }
        return Stack::pop();
    }
};
int main()
{
    MyStack stack(3); // Size = 3
    try
    {
        stack.push(10);
        stack.push(20);
        stack.push(30);
        // Overflow error
        stack.push(40);
    }
    catch (const std::overflow_error &e)
    {
        std::cerr << e.what() << std::endl;
    }
    try
    {
        std::cout << "Element pop: " << stack.pop() << std::endl;
        std::cout << "Element pop: " << stack.pop() << std::endl;
        std::cout << "Element pop: " << stack.pop() << std::endl;
        // Underflow error
        stack.pop();
    }
    catch (const std::underflow_error &e)
    {
        std::cerr << e.what() << std::endl;
    }
    return 0;
}
```

Output:

```
Stack Overflow: Stack full.
Element pop: 30
Element pop: 20
Element pop: 10
Stack Underflow: Stack empty.
```