Exercise Set 10

Instructions

Each exercise has a different difficulty level. You need to do three of the exercises and turn in all associated code (.cpp, .hpp files).

Make sure to keep it organized – zip up your projects with names like “ex10-part1.zip” so I can tell which files belong to which project.

Table of Contents

Just Inheritance (Level 1)

Classes, Inheritance

Write a simple program with 3 classes: A, B, and C.

A is a **base class;** it does not inherit from anything.

B is a **child of A**.

C is a **child of B**.

All 3 classes have a function named **Output**. The return type is void and it takes no parameters. Each version of the Output function returns a different value based on the class:

- A: output “a”
- B: output “ba”
- C: output “cb”

main function should look like this:

Just Inheritance Output (Part 1)

main()	Output

`int main()`
`{`
`  A myA;`
`  B myB;`
`  C myC;`
 	
`  myA.Output();`
`  myB.Output();`
`  myC.Output();`
 	
`  return 0;`
`}`
	`a`
	`ba`
	`cb`

Continued on the next page.

Just Inheritance, continued

After you've implemented it this way, you should make modifications to the Output function. Within the Output function, you should call the parents' version of Output after its own output:

- A: does not call a parent, only outputs “a”.
- B: outputs “b”, then calls A's version of Output.
- C: outputs “c”, then calls B's version of Output.

Now the output should look different:

Just Inheritance Output (Part 2)

main()	Output

`int main()`
`{`
`    A myA;`
`    B myB;`
`    C myC;`
 	
`    myA.Output();`
`    myB.Output();`
`    myC.Output();`
 	
`    return 0;`
`}`
	`a`
	`ba`
	`cba`

Doggies (Level 1)

Classes, Inheritance, protected members

You will create a family of Dog objects. There will be the base class, **Dog**, as well as two specializations: **Pomeranian** and **Wolf**.

The Dog class will have the following member variable:

- m_name, a string, protected access

*For a **protected** variable, any child classes can access it. If it were a private variable, no children could access it, only the Dog class itself.*

The **Dog** class will have the following member functions:

- SetName Return type: void Parameters: name, a const string reference
  This will set the protected m_name variable's value to the passed in value of name.
- GetName Return type: string Parameters: none
  This returns the value of the protected m_name variable.
- Breed Return type: string Parameters: none
  This returns the string “Mutt” for the Dog class.
- Speak Return type: string Parameters: none
  This returns the string “Woof!” for the Dog class.

*With the Dog class, the SetName and GetName functions will be reused for all Dogs. However, different types of classes will return different values for Breed and Speak, so we will be **overriding** these two functions.*

The **Pom** (or Pomeranian) class will inherit from Dog and override Breed and Speak. Breed is “Pomeranian” and Speak is “Yip!”.

The **Wolf** class will inherit from Dog and override Breed and Speak. Breed is “Wolf” and speak is “AAWWOO!”.

Doggies, continued

Doggies Output

main()	Output

`int main()`
`{`
`    Dog myMutt;`
`    Pom myPom;`
`    Wolf myWolf;`
 	
`    myMutt.SetName( "Daisy" );`
`    myPom.SetName( "Fluffy" );`
`    myWolf.SetName( "D-Dog" );`
 	
`    cout << "Dog 1: "`
`        << myMutt.GetName() << endl;`
`    cout << "Breed: "`
`        << myMutt.Breed() << endl;`
`    cout << "Bark:  "`
`        << myMutt.Speak() << endl << endl;`
 	
`    cout << "Dog 2: "`
`        << myPom.GetName() << endl;`
`    cout << "Breed: "`
`        << myPom.Breed() << endl;`
`    cout << "Bark:  "`
`        << myPom.Speak() << endl << endl;`
 	
`    cout << "Dog 3: "`
`        << myWolf.GetName() << endl;`
`    cout << "Breed: "`
`        << myWolf.Breed() << endl;`
`    cout << "Bark:  "`
`        << myWolf.Speak() << endl << endl;`
 	
`    return 0;`
`}`
	`Dog 1: Daisy`
	`Breed: Mutt`
	`Bark:  Woof!`
	
	`Dog 2: Fluffy`
	`Breed: Pomeranian`
	`Bark:  Yip!`
	
	`Dog 3: D-Dog`
	`Breed: Wolf`
	`Bark:  AAWWOO!`

Quizzer (Level 2)

Classes, Inheritance, Arrays

You will write a family of Question objects. The base class, **Question**, will have reusable functions that the child questions can use, but Question itself won't be used in main() since it is just an “interface”.

The Question class will have the following member variable:

- m_question, a protected string

The Question class will have the following member function:

- SetQuestion Return type: void Parameters: value, a const string reference
  Sets the value of the protected m_question variable.

Then, we'll declare a **TrueFalseQuestion** and **MultipleChoiceQuestion**. These will have different types of correct answers, and handle getting the user's choice in different ways.

The TrueFalseQuestion inherits from Question, and has the following member variable:

- m_correctAnswer, a boolean, private or protected.

TrueFalseQuestion has the following member functions:

- SetCorrectAnswer Return type: void Parameters: value, a boolean
  Sets the value of m_correctAnswer
- AskQuestion Return type: bool Parameters: none
  Display the question, as well as a prompt for T or F for true or false.
  Have the user enter their answer. Check to see if it is correct or not.
  If the answer is correct, return **true**. Otherwise, return **false**.
  - NOTE: The user will have entered a char or a string, so you will need to compare text to a boolean, which you cannot do directly with ==. You can create a temp variable to store “true” or “false” based on what the user entered, then compare it with m_correctAnswer. You could also use if/else if statements to figure out if the answer is correct or not.

Quizzer, continued

The MultipleChoiceQuestion class inherits from Question. It has the following member variables:

- m_answers, an array of strings of size 4.
- m_correctAnswer, an integer that corresponds to the index of one of the m_answers.

MultipleChoiceQuestion has the following member functions:

- SetAnswer Return type: void Parameters: index (int), value (const string ref)
  For an element of the m_answers array, at the appropriate index, set the element's value.
  Have error checking – if the index is invalid, do NOT set a value; just ignore or return.
- SetCorrectAnswer Return type: void Parameters: index (int)
  Set the value of m_correctAnswer to the index passed in.
- AskQuestion Return type: bool Parameters: none
  Display the question, display a numbered list of all possible answers, then ask the user to enter the # of the correct answer.
  If the answer is correct, return **true**. Otherwise, return **false**.

Quizzer Output

Output

```
The capital of KS is Topeka
[T]rue or [F]alse? t

Which state has the capital Jefferson City?
0. Missouri
1. Oklahoma
2. Arkansas
3. Washington
Which answer is correct? 0

You got 2 out of 2!
```

Quizzer, continued

main()

```
int main()
{
    TrueFalseQuestion       question1;
    MultipleChoiceQuestion  question2;

    question1.SetQuestion( "The capital of KS is Topeka" );
    question1.SetCorrectAnswer( true );

    question2.SetQuestion( "Which state has the capital Jefferson
City?" );
    question2.SetAnswer( 0, "Missouri" );
    question2.SetAnswer( 1, "Oklahoma" );
    question2.SetAnswer( 2, "Arkansas" );
    question2.SetAnswer( 3, "Washington" );
    question2.SetCorrectAnswer( 0 );

    int score = 0;

    if ( question1.AskQuestion() )
    {
        score++;
    }

    if ( question2.AskQuestion() )
    {
        score++;
    }

    cout << endl << "You got " << score << " out of 2!" << endl;

    return 0;
}
```

Customers and Employees (Level 3)

Implement a program that will keep track of customers and employees. There should be a **Person** class, with all the common variables needed: name, address, phone number. Remember to give these protected access.

A **Customer** will have the following information tracked: a count of transactions made by the customer (int), and the amount of credit on their account (float) such as from returns or gift cards.

An **Employee** will have the following information tracked: their wage per hour (float), months worked there (int), and job title (string). An Employee may also have up to 3 underlings, so an Employee object should also contain an array of Employees, as well as variables to keep track of the amount of underlings this employee has.

A Person will need common functions like Getters and Setters for the name, address, and phone #. These will be inherited by the Customer and Employee classes.

The Customer needs functions that will adjust their credit, and add to transactions (no need to be able to decrease transactions), as well as a function to display their information and credit/transactions.

An Employee will need to be able to have their wage set, increment amount of months worked (can't go down), and job title. There should also be functionality to Add an underling, and to Display all underlings for an employee.

Note that underlings can go deeper than one level: There might be the Manager, and under them the Assistant Manager, and under that the Sales Floor Coordinator, and a set of Sales People.

When you output the Manager's underlings, it should display the Assistant Manager, then proceed to display all of the Assistant Manager's underlings, and so on until the Sales People.

All classes may need Constructors, with information initialized. For example, Customers' transaction and credit will be initialized to 0, and an Employee's underling count should be 0 as well.

In main(), initialize a set of customers as an array. Create just one Employee (the Manager), but set up all other Employees under the Manager (Manager – Assistant Manager – Coordinator – Sales, etc.)

After everybody has been set up, display all Customer information and all Employee information.

Battles (Level 3)

For this program, the old battle game program will be updated (you might want to rewrite from scratch, though.)

There will be a **Character** class, which will contain the common variables and functionality, such as name, hp, and atk, and functions to GetAttack and GetHurt.

- The GetAttack function will get an attack value. If the character is attacking, it will be the atk amount. If they are defending, it will be 0.
- The GetHurt function will take the attack value from the opposite character. If the active character is defending, the damage amount will be reduced before subtracting it from their HP. If the character is attacking, the full damage amount will be deducted from their HP.

Then, there will be an **Enemy** class, which will have a function to decide whether to attack or to defend. It will use random numbers to decide.

And there is a **Player** class, which will have a function to decide whether to attack or to defend. It will display a menu to the user and ask them to enter something.

In the game, you will create an Enemy and a Player. Each round, they will choose to attack or defend. The game will continue running until one or both of their HP is less than or equal to 0.