National University of Computer and Emerging Sciences

Lab Manual 12
Object Oriented Programming

Course Instructor	Ms. Hafsa Tariq
Lab Instructor (s)	Ms Sonia Anum Ms Yusra Arshad
Section	BSCS-2J
Semester	Spring 2022

Department of Computer Science
FAST-NU, Lahore, Pakistan

Objectives:

- Declare virtual functions and destructors
- See polymorphism in action
- Abstract classes

Exercise 1:

Write a program to practice memory management alongside polymorphism.
You are not allowed to change function prototypes
Implement following class structure. In addition to this you are to implement destructors in all classes below to ensure dynamically allocated memory is properly deleted.

Person (Base Class)	Employee (Derived)	Student (Derived)

//member variables
{
Private:
String fullName;
Int height;
 	
Public:
 	
Person(string name,Int height) //constructor
 	
Virtual void printInfo();
//this function is to print all private varaibles
 	
//destructor to be implemented alongside type of class eg cout<<"person destructor" (5)
 	
}
	//member variables
	{
	Private:
	String departement;
	Int ID;
	
	Public:
	
	Employee(string name,Int height,string departement,Int id) : Person(name, height)
	//constructor
	
	void printInfo();
	//this function is to print all private variables alongside type of class
	
	//destructor to be implemented alongside type of class eg cout<<"employee destructor"
	}
		//member variables
		{
		Private:
		String schoolName;
		
		Public:
		
		Student **(string name,Int height, string SchoolName) : Person(name, height)**
		//constructor
		
		void printInfo();
		//this function is to print all private variables alongside type of class
		
		//destructor to be implemented alongside type of class eg cout<<"student destructor"
		}

Main Program:

1. Initialize each of the base class member with employee and student object respectively.
2. Call delete operator on the array of base class to test the memory management.

Exercise 2:
For the first part of this lab, we are going to see if base class pointers can point to an object of derived class and vice versa. Perform the following steps

- Create a class called `Animal`.
- An animal can speak so create a public method named speak in the which returns a `char *`
- Modify the definition of the speak method so that it returns the string `"speak() called."`.
- A Dog is an Animal. Create a class named `Dog`. Use public inheritance.
- The `Dog` class will inherit the `speak` method from `Animal`. Override this method in the `Dog` class so that it returns "woof!" when called.
- Add the following lines in the main function of your program, note the output and paste it in the space provided below.

```
Animal objAnimal;
Dog objDog;
Animal *ptrAnimal = &objAnimal;
Dog *ptrDog = &objDog;

cout << objAnimal.speak() << endl;
cout << objDog.speak() << endl;
cout << ptrAnimal->speak() << endl;
cout << ptrDog->speak() << endl;
```

You can see that we have created two objects, one for each class and two pointers in the same manner. The pointer to `Animal` is pointing to an object of the class `Animal` and the pointer to `Dog` is pointing to the object of class `Dog`. In this example, `ptrAnimal` called the `speak` method of the class `Animal` where as `ptrDog` called the `speak` method of the class Dog. If we want to use the definition of the base class method that we have overloaded in a derived class from a derived class pointer, we have to follow this syntax.

```
ptrDog->Animal::speak();
```

Modify the last line of your program to use the syntax above, execute it and paste the output in the box below.

Exercise 2:

Now we will see what happens if we change this. Change the main function so that `ptrAnimal` is pointing to the object of `Dog`, execute your program and paste the output below

Now modify the code so that `ptrDog` points to `objAnimal` and compile your program. It will not compile successfully. Paste the error in the space below. What does this show?

You can see that there was no problem pointing to an object of a derived class by a pointer of the base class but when calling a function through this pointer, the definition of the base class is used. In the other case, there was a compilation error which showed that it is not possible to point a derived class pointer to an object of base class.

Exercise 3:

Change the main function of your program and replace it with the code below. Execute it and paste the output in the box below.

```
Dog lassie;
Animal *myPet = &lassie;
cout << myPet->speak() << endl;
```

You will see that as expected, the `speak` method of the base class was called. What if we wanted to use the definition of the derived class function? To accomplish this, we can add the keyword virtual to the declaration of the `speak()` method in the `Animal` class. Specifying a function as `virtual` makes sure that whenever we use a base class pointer pointing to an object of a derived class to call a function, the definition of the method declared in the derived class is used. Modify the speak method of the `Animal` class as shown below, compile your code, execute it and paste the output in the space provided below.

```
virtual char *  speak()
{
      return "speak() called.";
}
```

Exercise 4:

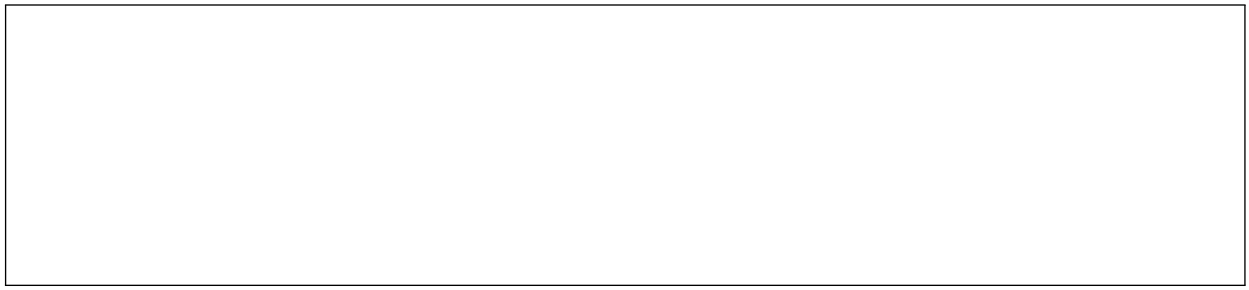
In the above exercises, we have seen a very simple implementation of Polymorphism. The real power of this feature is realized when we have a collection of objects of multiple derived classes and we use a pointer of the base class to call their respective overloaded methods. A Cat is an animal too. Let's see how we can use an array of base class pointers to utilize the essence of polymorphism.

- Define a class `Cat`. Inherit publicly from `Animal` just like we did in class `Dog`.
- Overload the `speak`() method so that it returns "mew!".
- Modify the `main` function as shown below.
- Compile, execute and paste the output in the space given below.

```
const int size = 2;
Animal * myPets[size];
Cat whiskers;
Dog mutley;

myPets[0] = &whiskers;
myPets[1] = &mutley;

for(int i=0; i<size; i++)
      cout << myPets[i]->speak() << endl;
```



We can see that although we used the base class pointer to call the `speak`() method, the overloaded definitions of the derived class was used. We can also see that the "virtualness" of the function declared in `Animal` was inherited in the new derived class Cat as well so once a function is declared `virtual`, we don't need to add the keyword virtual to all derived class definitions.

Exercise 5:

Modify the main function so that the size of array `myPets` is 5. Display a menu to the user asking him the type of pet for each of the 5 pets. For each input, create the object of the respective class dynamically and point to it by the corresponding pointer of the array `myPets`. Once you have taken all 5 inputs, use a loop to call the `speak()` method and delete each of the objects. You can use the following code to take the input. The declaration of `getch()` is in the header file `conio.h`.

```
int i = 0;
while (i<size)
{
	cout << "Press 1 for a Dog and 2 for a Cat." << endl ;
	switch (getch())
	{
		case '1':
			myPets[i] = new Dog;
			cout << "Dog added at position "<< i <<endl<<endl;
			i++;
			break;
		case '2':
			myPets[i] = new Cat;
			cout << "Cat added at position "<< i <<endl<<endl;
			i++;
			break;
		default:
			cout<<"Invalid input. Enter again." <<endl<<endl;
			break;
	}
}
```

Exercise 6:

Although things seem to be fine on the surface, there is a problem in the program we just wrote. To observe this problem, we must add destructors for all classes. Paste the following inline definitions of the destructors in their corresponding classes, execute the program and paste the output below.

```
~Animal()   { cout << "~Animal() called."<<endl;      }
```

```
~Cat()      { cout << "~Cat() called."<<endl;         }
```

```
~Dog()      { cout << "~Dog() called."<<endl;         }
```

Can you see what went wrong? When using delete to deallocate memory, only the base class destructor is called whereas the derived class destructor is not called at all. Although this is fine in the example we are using here but it will create memory leaks if there are any dynamically allocated variables in any of the derived class. To avoid this we declare the base class destructor as `virtual`. Doing this will make sure that the derived class destructor is called even if you are using a base class pointer to call the destructor. Now change the definition of the base class destructor to make it virtual, execute the program. Make sure you can see the derived class destructors being called in the output.

Exercise 7:

Create a class named ***Person***, which contains

- A pure virtual function named ***print()***
- Two data fields i.e. ***personName*** and ***age***

A class named ***Patient*** inherits ***Person*** class, which contains

- Two data fields i.e. ***diseaseType*** and ***recommendedMedicine***
- Overridden function ***print()*** to display all details relevant to a patient

A class named ***MedicarePatient*** inherited from class ***Patient***, which holds

- A data field representing the ***name of the hospital***
- A data filed representing the ***name of the ward***
- A data field representing ***room number***
- Overridden function ***print()*** to display all details relevant to a patient

In the ***main*** function, create instances of derived classes to access respective ***print()*** function using ***dynamic binding***.