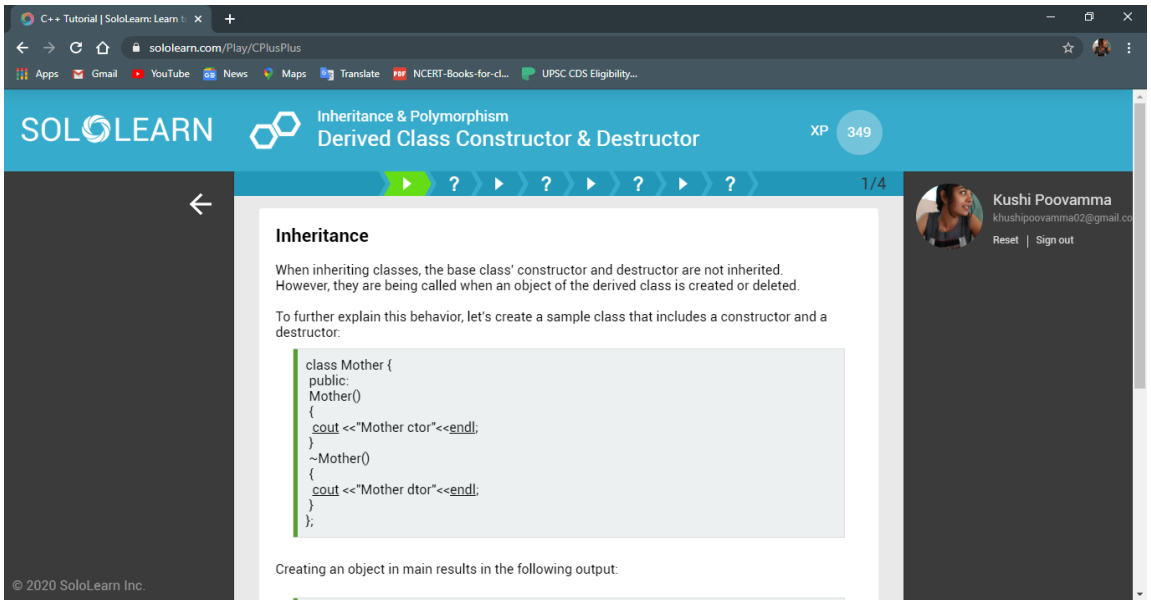
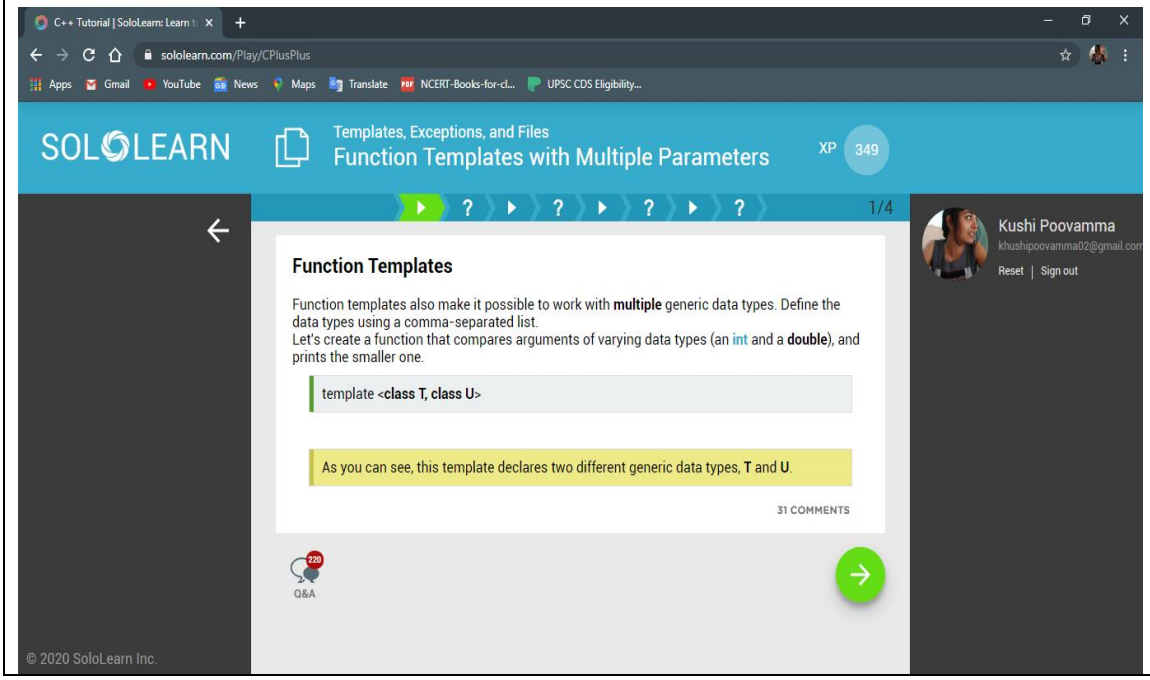
DAILY ASSESSMENT

Date:	25-06-2020	Name:	K B KUSHI
Course:	C++ Programming	USN:	4AL17EC107
Topic:	Inheritance and Polymorphism Templates,exceptions and files	Semester & Section:	6th & B
GitHub Repository:	https://www.github.com/alvas-education-foundation/KUSHI-COURSES.git		

FORENOON SESSION DETAILS

Image of session





Report:

One of the most important concepts in object-oriented programming is that of inheritance. Inheritance allows us to define a class in terms of another class, which makes it easier to create and maintain an application. This also provides an opportunity to reuse the code functionality and fast implementation time.

When creating a class, instead of writing completely new data members and member functions, the programmer can designate that the new class should inherit the members of an existing class. This existing class is called the base class, and the new class is referred to as the derived class.

The idea of inheritance implements the is a relationship. For example, mammal IS-A animal, dog IS-A mammal hence dog IS-A animal as well and so on.

Base and Derived Classes

A class can be derived from more than one classes, which means it can inherit data and functions from multiple base classes. To define a derived class, we use a class derivation list to specify the base class(es). A class derivation list names one or more base classes and has the form −

class derived-class: access-specifier base-class

Where access-specifier is one of public, protected, or private, and base-class is the name of a previously defined class. If the access-specifier is not used, then it is private by default.

Consider a base class Shape and its derived class Rectangle as follows −x

```
#include <iostream>

using namespace std;

// Base class
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
};
```

```
// Derived class
class Rectangle: public Shape {
  public:
    int getArea() {
      return (width * height);
    }
};

int main(void) {
  Rectangle Rect;

  Rect.setWidth(5);
  Rect.setHeight(7);

  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;

  return 0;
}
```

When the above code is compiled and executed, it produces the following result –

Total area: 35

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class.

We can summarize the different access types according to - who can access them in the following way –

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions –

- **Constructors, destructors and copy constructors of the base class.**

- **Overloaded operators of the base class.**
- **The friend functions of the base class.**

Type of Inheritance

When deriving a class from a base class, the base class may be inherited through public, protected or private inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use protected or private inheritance, but public inheritance is commonly used. While using different type of inheritance, following rules are applied −

- **Public Inheritance − When deriving a class from a public base class, public members of the base class become public members of the derived class and protected members of the base class become protected members of the derived class. A base class's privatemembers are never accessible directly from a derived class, but can be accessed through calls to the public and protected members of the base class.**
- **Protected Inheritance − When deriving from a protected base class, public and protected members of the base class become protected members of the derived class.**
- **Private Inheritance − When deriving from a private base class, public and protectedmembers of the base class become private members of the derived class.**

Multiple Inheritance

A C++ class can inherit members from more than one class and here is the extended syntax −

class derived-class: access baseA, access baseB....

Where access is one of public, protected, or private and would be given for every base class and they will be separated by comma as shown above. Let us try the following example −

```
#include <iostream>

using namespace std;

// Base class Shape
class Shape {
  public:
    void setWidth(int w) {
      width = w;
    }
    void setHeight(int h) {
      height = h;
    }

  protected:
    int width;
    int height;
```

```
};

// Base class PaintCost
class PaintCost {
  public:
    int getCost(int area) {
      return area * 70;
    }
};

// Derived class
class Rectangle: public Shape, public PaintCost {
  public:
    int getArea() {
      return (width * height);
    }
};

int main(void) {
  Rectangle Rect;
  int area;

  Rect.setWidth(5);
  Rect.setHeight(7);

  area = Rect.getArea();

  // Print the area of the object.
  cout << "Total area: " << Rect.getArea() << endl;

  // Print the total cost of painting
  cout << "Total paint cost: $" << Rect.getCost(area) << endl;

  return 0;
}
```

When the above code is compiled and executed, it produces the following result –

```
Total area: 35
Total paint cost: $2450
```

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

Consider the following example where a base class has been derived by other two classes −

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

class Shape {
  protected:
    int width, height;

  public:
    Shape( int a = 0, int b = 0){
      width = a;
      height = b;
    }
    int area() {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
class Rectangle: public Shape {
  public:
    Rectangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
      cout << "Rectangle class area :" <<endl;
      return (width * height);
    }
};

class Triangle: public Shape {
  public:
    Triangle( int a = 0, int b = 0):Shape(a, b) { }

    int area () {
      cout << "Triangle class area :" <<endl;
      return (width * height / 2);
    }
};

// Main function for the program
int main() {
  Shape *shape;
  Rectangle rec(10,7);
  Triangle  tri(10,5);
```

```
  // store the address of Rectangle
  shape = &rec;

  // call rectangle area.
  shape->area();

  // store the address of Triangle
  shape = &tri;

  // call triangle area.
  shape->area();

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Parent class area :
Parent class area :
```

The reason for the incorrect output is that the call of the function area() is being set once by the compiler as the version defined in the base class. This is called static resolution of the function call, or static linkage - the function call is fixed before the program is executed. This is also sometimes called early binding because the area() function is set during the compilation of the program.

But now, let's make a slight modification in our program and precede the declaration of area() in the Shape class with the keyword virtual so that it looks like this −

```
class Shape {
  protected:
    int width, height;

  public:
    Shape( int a = 0, int b = 0) {
      width = a;
      height = b;
    }
    virtual int area() {
      cout << "Parent class area :" <<endl;
      return 0;
    }
};
```

After this slight modification, when the previous example code is compiled and executed, it produces the following result −

```
Rectangle class area
Triangle class area
```

This time, the compiler looks at the contents of the pointer instead of it's type. Hence, since addresses of objects of tri and rec classes are stored in *shape the respective area() function is called.

As you can see, each of the child classes has a separate implementation for the function area(). This is how polymorphism is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

Virtual Function

A virtual function is a function in a base class that is declared using the keyword virtual. Defining in a base class a virtual function, with another version in a derived class, signals to the compiler that we don't want static linkage for this function.

What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as dynamic linkage, or late binding.

Pure Virtual Functions

It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

We can change the virtual function area() in the base class to the following −

```
class Shape {
  protected:
    int width, height;

  public:
    Shape(int a = 0, int b = 0) {
      width = a;
      height = b;
    }

    // pure virtual function
    virtual int area() = 0;
};
```

The = 0 tells the compiler that the function has no body and above virtual function will be called pure virtual function.

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.

A template is a blueprint or formula for creating a generic class or a function. The library containers like iterators and algorithms are examples of generic programming and have been developed using template concept.

There is a single definition of each container, such as vector, but we can define many different kinds of vectors for example, vector <int> or vector <string>.