

# Abstract Classes and Interfaces

Ali Haider

[syedalihaider.ciit@gmail.com](mailto:syedalihaider.ciit@gmail.com)

Department of Computer Science IUB

# Overview

- Interfaces
- Pure Virtual Function
- Abstract Classes

# Interface

- An interface describes the behavior or capabilities of a C++ class without committing to a particular implementation of that class.
- The C++ interfaces are implemented using **abstract classes**
- And these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.
- A class is made abstract by declaring at least one of its functions as **pure virtual** function.

# Pure Virtual Function

- A pure virtual function is specified by placing "= 0" in its declaration
- class Box {
- private:
- double length; // Length of a box
- double breadth; // Breadth of a box
- double height; // Height of a box
- public:
- // pure virtual function
- virtual double getVolume() = 0;
- };

# Abstract Class

- The purpose of an abstract class (often referred to as an ABC) is to provide an appropriate base class from which other classes can inherit.
- Abstract classes cannot be used to instantiate objects and serves only as an interface.
- Attempting to instantiate an object of an abstract class causes a compilation error.
- Thus, if a subclass of an ABC needs to be instantiated, it has to implement each of the virtual functions, which means that it supports the interface declared by the ABC.
- Failure to override a pure virtual function in a derived class, then attempting to instantiate objects of that class, is a compilation error.
- Classes that can be used to instantiate objects are called concrete classes.

# Why Need Abstract Classes

- An object-oriented system might use an abstract base class to provide a common and standardized interface appropriate for all the external applications.
- Then, through inheritance from that abstract base class, derived classes are formed that operate similarly.
- The capabilities (i.e., the public functions) offered by the external applications are provided as pure virtual functions in the abstract base class.
- The implementations of these pure virtual functions are provided in the derived classes that correspond to the specific types of the application.
- This architecture also allows new applications to be added to a system easily, even after the system has been defined.

# Example

- Parent class provides an interface to the base class to implement a function called **getArea()**

```
#include <iostream>
using namespace std;
// Base class
class Shape {
protected:
    int width;
    int height;
public:
    // pure virtual function providing interface framework.
    virtual int getArea() = 0;
    void setWidth(int w) {
        width = w;
    }
    void setHeight(int h) {
        height = h;
    }
};
```

Cont...

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

class Triangle: public Shape {
public:
    int getArea() {
        return (width * height)/2;
    }
};
```

# Cont...

```
int main(void) {
    Rectangle Rect;
    Triangle Tri;

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

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

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

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

    return 0;
}
```

# Cont...

- You can see how an abstract class defined an interface in terms of `getArea()` and two other classes implemented same function but with different algorithm to calculate the area specific to the shape