**Data encapsulation**

All C++ programs are composed of the following two fundamental elements:

* **Program statements (code):** This is the part of a program that performs actions and they are called functions.
* **Program data:** The data is the information of the program which affected by the program functions.

Encapsulation is an Object Oriented Programming concept that binds together the data and functions that manipulate the data, and that keeps both safe from outside interference and misuse. Data encapsulation led to the important OOP concept of **data hiding**.

**Data encapsulation** is a mechanism of bundling the data, and the functions that use them and **data abstraction** is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called **classes**. We already have studied that a class can contain **private, protected** and **public** members. By default, all items defined in a class are private. For example:

class Box

{

public:

double getVolume(void)

{

return length \* breadth \* height;

}

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

The variables length, breadth, and height are **private**. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class **public** (i.e., accessible to other parts of your program), you must declare them after the **public** keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

**Data Encapsulation Example:**

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example:

#include <iostream>

using namespace std;

class Adder{

public:

// constructor

Adder(int i = 0)

{

total = i;

}

// interface to outside world

void addNum(int number)

{

total += number;

}

// interface to outside world

int getTotal()

{

return total;

};

private:

// hidden data from outside world

int total;

};

int main( )

{

Adder a;

a.addNum(10);

a.addNum(20);

a.addNum(30);

cout << "Total " << a.getTotal() <<endl;

return 0;

}

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

Total 60

Above class adds numbers together, and returns the sum. The public members **addNum** and **getTotal** are the interfaces to the outside world and a user needs to know them to use the class. The private member **total** is something that is hidden from the outside world, but is needed for the class to operate properly.

**Designing Strategy:**

Most of us have learned through bitter experience to make class members private by default unless we really need to expose them. That's just good **encapsulation**.

This wisdom is applied most frequently to data members, but it applies equally to all members, including virtual functions.

# Interfaces in C++ (Abstract Classes)

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. A pure virtual function is specified by placing "= 0" in its declaration as follows:

class Box

{

public:

// pure virtual function

virtual double getVolume() = 0;

private:

double length; // Length of a box

double breadth; // Breadth of a box

double height; // Height of a box

};

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**.

## Abstract Class Example:

Consider the following example where parent class provides an interface to the base class to implement a function called **getArea()**:

#include <iostream>

using namespace std;

// Base class

class Shape

{

public:

// pure virtual function providing interface framework.

virtual int getArea() = 0;

void setWidth(int w)

{

width = w;

}

void setHeight(int h)

{

height = h;

}

protected:

int width;

int height;

};

// Derived classes

class Rectangle: public Shape

{

public:

int getArea()

{

return (width \* height);

}

};

class Triangle: public Shape

{

public:

int getArea()

{

return (width \* height)/2;

}

};

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;

}

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

Total Rectangle area: 35

Total Triangle area: 17

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.

## Designing Strategy:

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 all 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.