Inheritance and Class Hierarchies

©Stewart Weiss

Inheritance is a feature that is present in many object-oriented languages such as C++, Eiffel, Java, Ruby, and Smalltalk, but each language implements it in its own way. This chapter explains the key concepts of the *C++ implementation of inheritance*.

1 Deriving Classes

Inheritance is a feature of an object-oriented language that allows classes or objects to be defined as extensions or specializations of other classes or objects. In C++, classes inherit from other classes. Inheritance is useful when a project contains many similar, but not identical, types of objects. In this case, the task of the programmer/software engineer is to find commonality in this set of similar objects, and create a class that can serve as an archetype for this set of classes.

1.1.2 Examples

- Squares, triangles, circles, and hexagons are all 2D shapes; hence a ***Shape*** class could be an archetype.
- Faculty, administrators, office assistants, and technical support staff are all employees, so an ***Employee*** class could be an archetype.
- Cars, trucks, motorcycles, and buses are all vehicles, so a ***Vehicle*** class could be an archetype.

When this type of relationship exists among classes, it is more efficient to create a class hierarchy rather than replicating member functions and properties in each of the classes. Inheritance provides the mechanism for achieving this.

1.1.3 Syntax

The syntax for creating a derived class is very simple. (You will wish everything else about it were so simple though.)

```
class A
{/* ... stuff here ... */};

class B: [access-specifier] A
{/* ... stuff here ... */};
```

in which an *access-specifier* can be one of the words, **`public`**, **`protected`**, or **`private`**, and the square brackets indicate that it is optional. *If omitted, the inheritance is **private***.

In this example, A is called the ***base class*** and B is the ***derived class***. Sometimes, the base class is called the *superclass* and the derived class is called a *subclass*.

<u>Five Important Points (regardless of access specifier):</u>

1. The constructors and destructors of a base class are not inherited.
2. The assignment operator is not inherited.
3. The friend functions and friend classes of the base class are not inherited.
4. The derived class does not have access to the base class's private members.

5. The derived class has access to all public and protected members of the base class.

Public inheritance expresses an ***is-a*** relationship: a B ***is a*** particular type of an A, as a car ***is a*** type of vehicle, a manager ***is a*** type of employee, and a square ***is a*** type of shape.

Protected and private inheritance serve different purposes from public inheritance. Protected inheritance makes the public and protected members of the base class protected in the derived class. Private inheritance makes the public and protected members of the base class private in the derived class. This is summarized in Table 1. These notes do not examine protected and private inheritance.

Access in Base Class	Base Class Inheritance Method	Access in Derived Class
public protected private	public	public protected no access allowed
public protected private	protected	protected protected no access
public protected private	private	private private no access

Table 1Summary of Accesses in Inheritance

1.1.4 Example

In this example, a Shape class is defined and then many different kinds of shapes are derived from it.

```
class Shape {
private:
    Point  Centroid;
public:
    void Move(Point newCentroid);  // move Shape to new centroid
/* more stuff here */
};

class Square   : public Shape {/* stuff here */ };
class Triangle : public Shape {/* stuff here */ };
class Hexagon  : public Shape {/* stuff here */ };
/* and so forth */
```

This set of classes can be depicted schematically by a directed graph in which each class is a node and an edge is directed from node A to node B if A is derived from B directly, as illustrated below.

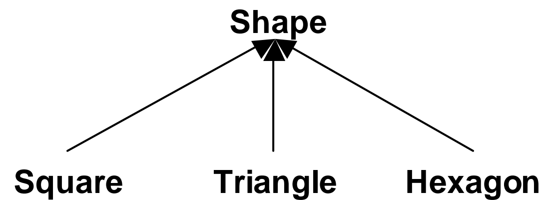


Figure 1 Shape Class Hierarchy

1.2 Implicit Conversion of Classes

The C++ language allows certain assignments to be made even though the types of the left and right sides are not identical. For example, it will allow an integer to be assigned to a floating-point type without an explicit cast. However, it will not in general let a pointer of one type be assigned to a pointer of another type. One exception to this rule has to do with assigning pointers to classes. I will begin this section by stating its conclusion. If you do not want to understand why it is true or get a deeper understanding of the nature of inheritance, you can then just skip to the next section.

Implicit Conversion of Classes: *The address of an object of a derived class can be assigned to a pointer declared to the base class, but the base class pointer can be used only to invoke member functions of the base class.*

Because a `Square` is a specific type of `Shape`, a `Square` is a `Shape`. But because a `Square` has other attributes, a `Shape` is not necessarily a `Square`. But consider a `Shape*` pointer. A `Shape*` is a pointer to a `Shape`. A `Square*` is a pointer to a `Square`. A `Square*` is not the same thing as a `Shape*`, since one points to `Squares` and the other points to `Shapes`, and so they have no inherent commonality other than their "pointerness."

However, since a `Square` is also a `Shape`, a `Square*` can be used wherever a `Shape*` can be used. In other words, `Squares`, `Triangles`, and `Hexagons` are all `Shapes`, so whatever kinds of operations can be applied to `Shapes` can also be applied to `Squares`, `Triangles`, and `Hexagons`. Thus, it is reasonable to be able to invoke any operation that is a member function of the `Shape` class on any dereferenced `Shape*`, whether it is a `Square`, a `Triangle`, or a `Hexagon`. This argument explains why, in C++, ***the address of any object derived from a `Shape` can be assigned to a `Shape` pointer; e.g., a `Square*` can be assigned to a `Shape*`***.

The converse is not true; a `Shape*` cannot be used wherever a `Square*` is used because a `Shape` is not necessarily a `Square`! Dereferencing a `Square*` gives access to a specialized set of operations that only work on `Squares`. If a `Square*` contained the address of a `Shape` object, then after dereferencing the `Square*`, you would be allowed to invoke a member function of a `Square` on a `Shape` that does not know what it is like to be a `Square`, and that would make no sense.

The need to make the preceding argument stems from the undecidability of the Halting Problem and the need for the compiler designer to make sensible design decisions. If you are not familiar

with the Halting Problem, you can think of it as a statement that there are problems for which no algorithms exist. One consequence of the Halting Problem is that it is not possible for the ***compiler*** to know whether or not the address stored in a pointer is always going to be the address of any specific object. To illustrate this, consider the following code fragment, and assume that the `Square` class is derived from the `Shape` class.

```
1.  Square* pSquare;
2.  Shape* pShape;
3.  void*   ptr;
4.  Shape   someShape;
5.  Square someSquare;
6.  /* .. ..... */
7.  if ( some condition that depends on user input )
8.        ptr = (void*) &someShape;
9.  else
10.       ptr = (void*) &someSquare;
11. pShape = (Shape*) ptr;
```

The compiler cannot tell at compile time whether the true or false branch of the if-statement will always be taken, ever be taken, or never taken. If it could, we could solve the Halting Problem. Thus, at compile-time, it cannot know whether the assignment in line 11 will put the address of a `Square` or a `Shape` into the variable `pShape`. Another way to say this is that, at compile-time, the compiler cannot know how to bind an object to a pointer. The designer of C++ had to decide what behavior to give to this assignment statement. *Should it be allowed? Should it be a compile time error? If it is allowed, what operations can then be performed on this pointer?*

The only sensible and safe decision is to allow the assignment to take place, but to play it "safe" and allow the dereferenced `pShape` pointer to access only the member functions and members of the `Shape` class, not any derived class. This semantics will be revisited in §1.5.

1.3 Multiple Inheritance

Suppose that the shapes are not geometric abstractions but are instead windows in an art-deco building supply store. Then what they also have in common is the property of being a window, assuming they are all the same type of window (e.g., double hung, casement, sliding, etc.). Then geometric window shapes really inherit properties from different archetypes, i.e., the property of being a shape and the property of being a window. In this case we need ***multiple inheritance***, which C++ provides:

1.3.2 Example

```
class Shape {
private:
    Point  Centroid;
public:
    void Move(Point newCentroid);  // move Shape to new centroid
/* more stuff here about being a Shape */
};
```

```
class Window
 {/* stuff here about being a Window */ };

class SquareWin :   public Shape, public Window {/* stuff here */ };
class TriangleWin : public Shape, public Window {/* stuff here */ };
class HexagonWin  : public Shape, public Window {/* stuff here */ };
/* and so forth */
```

Note the syntax. The derived class is followed by a single colon (:) and a comma-separated list of base classes, with the inheritance qualifier (public) attached to each base class. The set of classes created by the above code creates the hierarchy depicted in Figure 2.

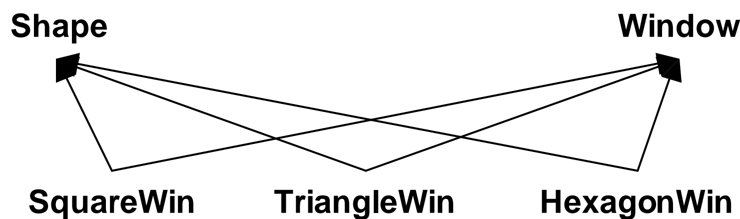


Figure 2 Class Hierarchy with Mulitple Inheritance

We could, in fact, make this more interesting by assuming that the Window class is also a base class from which are derived three different window styles: DoubleHung, Casement, and Sliding. Ignoring the physical impossibilities of certain of these combinations (it is pretty hard to make a triangular double-hung window), there are now nine possible window classes, being the product of three different shapes and three different styles. The hierarchy would be the one depicted in Figure 3.

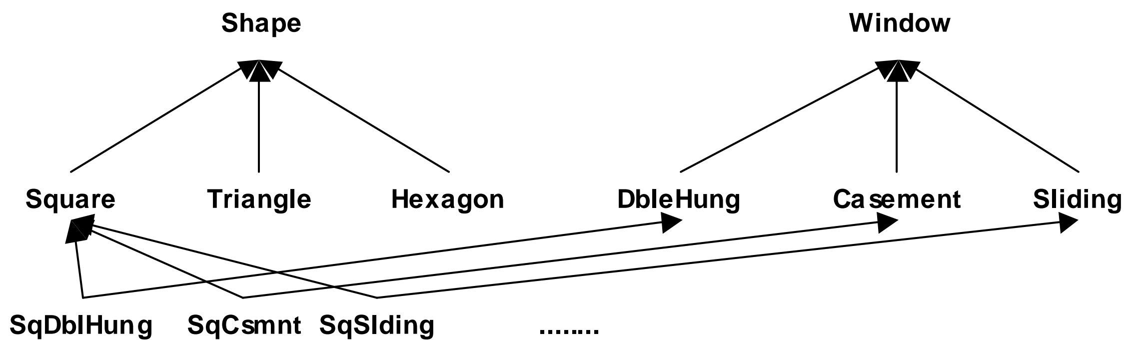


Figure 3 Three Level Multiple Inheritance

1.4 Extending Functionality of Derived Classes with Member Functions

Inheritance would be relatively useless if it did not allow the derived classes to make themselves different from the base class by the addition of new members, whether data or functions. The derived class can add members that are not in the base class, making it like a subcategory of the base class.

1.4.2 Example

The `Square` class below will add a member function not in the base class.

```
class Shape
{
private:
    Point  Centroid;
public:
    void Move(Point newCentroid);  // move Shape to new centroid
    Point getCentroid() const;
    /* more stuff here */
};

class Rectangle : public Shape
{
private:
    float length, width;
public:
     /* some stuff here too */
    float LengthDiagonal() const;// functionality not in Shape class
};
```

The `Rectangle` class can add a `LengthDiagonal` function that was not in the base class since it did not makes sense for it to be in the base class and only the `Rectangle` class has all diagonals of the same length. Remember though that the member functions of the derived class cannot access the private members of the base class, so the base class must either make protected or public accessor functions for its subclasses if they need access to the private members.

1.5 Redefining Member Functions in Derived Classes

Derived classes can redefine member functions that are declared in the base class. This allows subclasses to specialize the functionality of the base class member functions for their own particular needs. For example, the base class might have a function `print()` to display the fields of an employee record that are common to all employees, and in a derived class, the `print()` function might display more information than the `print()` function in the base class.

Note: *A function in a derived class overrides a function in the base class* ***only if*** *its signature is identical to that of the base class except for some minor differences in the return type.* It must have the same parameters and same qualifiers. Thus,

```
void print();
void print() const;
```

are not the same and the compiler will treat them as different functions, whereas

```
void print() const;
void print() const;
```

are identical and one would override the other. Continuing with the `Shape` example, suppose that a `Shape` has the ability to print only its `Centroid` coordinates, but we want each derived class to print out different information. Consider the `Rectangle` class again, with a `print()` member function added to `Shape` and `Rectangle` classes.

```
class Shape
{
private:
    Point  Centroid;
public:
    void Move(Point newCentroid);  // move Shape to new centroid
    Point getCentroid() const;
    void print() const;    //  prints just Centroid coordinates
    /* more stuff here */
};

class Rectangle : public Shape
{
private:
    float length, width;
public:
     /* some stuff here too */
    float LengthDiagonal() const;// functionality not in Shape class
    void print() const
    { /* print stuff that Rectangle class has here */ }
};

/* .... */
Shape   myShape;
Rectangle myRect;
myRect.print();
myShape.print();
```

The call to `myRect.print()` will invoke the `print()` member function of the `Rectangle` class, since `myRect` is bound to a `Rectangle` at compile time. Similarly, the call to `myShape.print()` will invoke the `Shape` class's `print()` function. But what happens here:

```
Shape* pShape;
pShape = new Rectangle;
pShape->print();
```

In this case, the address of the dynamically allocated, anonymous `Rectangle` object is assigned to a `Shape` pointer, and referring back to §1.2 above, the dereferenced `pShape` pointer will point to the `print()` member function in the `Shape` class, since the compiler binds a pointer to the member functions of its own class. Even though it points to a `Rectangle` object, `pShape` cannot invoke the `Rectangle`'s `print()` function. This problem will be overcome in §2 below when virtual functions are introduced.

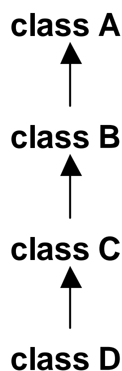
1.6 Revisiting Constructors and Destructors

1.6.2 Constructors

Let us begin by answering two questions.

***When does a class need a constructor*?** If a class must initialize its data members, then the class needs a user-defined constructor because the compiler-generated constructor will not be able to do this.

***If a class is derived from other classes, when does it need a user-defined constructor*?** To understand the answer, it is necessary to understand what happens at run time when an object is created. Class objects are always constructed from the bottom up, meaning that the lowest level base class object is constructed first, then any base class object immediately derived from that is constructed, and so on, until the constructor for the derived class itself is called. To make this concrete, suppose that four classes have been defined, and that they form the hierarchy depicted below, in which **D** is derived from **C,** which is derived from **B,** which is derived from **A**.



Then when an object of class **D** is created, the run time system will recursively descend the hierarchy until it reaches the lowest level base class (**A**), and construct **A**, then **B**, then **C**, and finally **D**.

From this discussion, it should be clear that a constructor is required for every class from which the class is derived. If a base class's constructors require arguments, then there must be a user-supplied constructor for that class, and any class derived from it must explicitly call the constructor of the base class, supplying the arguments to it that it needs. If the base class has at least one default constructor, the derived class does not need to call it explicitly, because the default constructor can be invoked implicitly as needed. In this case, the derived class may not

need a user-defined constructor, because the compiler will arrange to have the run time system call the default constructors for each class in the correct order. But in any case, when a derived class object is created, a base class constructor must ***always*** be invoked, whether or not the derived class has a user-defined constructor, and whether or not it requires arguments. The short program that follows demonstrates the example described above.

Example

```
class A  {
public:
    A() {cout << "A constructor called\n";}
};
class B : public A  {
public:
    B() {cout << "B constructor called\n";}
};
class C : public B  {
public:
    C() {cout << "C constructor called\n";}
};
class D : public C  {};
void main() {
    D  d;
}
```

This program will display

```
A constructor called
B constructor called
C constructor called
```

because the construction of d implicitly requires that **C**'s constructor be executed beforehand, which in turn requires that **B**'s constructor be executed before **C**'s, which in turn requires that **A**'s constructor be executed before **B**'s. To make this explicit, you would do so in the initializer lists:

```
class A {
public:
    A() {}
};
class B : public A  {
public:
    B(): A() {}
};
class C : public B  {
public:
    C(): B() {}
};
class D : public C  {
public:
    D(): C() {}
};
```

This would explicitly invoke the **A()**, then **B()**, then **C()**, and finally **D()** constructors.

Summary

The important rules regarding constructors are:

- A base class constructor is ALWAYS called if an object of the derived class is constructed, even if the derived class does not have a user-defined constructor.
- If the base class does not have a default constructor, then the derived class must have a constructor that can invoke the appropriate base class constructor with arguments.
- The constructor of the base class is invoked before the constructor of the derived class.
- If the derived class has members in addition to the base class, these are constructed after those of the base class.

1.6.3 Destructors

Destructors are slightly more complicated than constructors. The major difference arises because destructors are ***rarely*** called explicitly. They are invoked for only one of two possible reasons:

1. The object was not created dynamically and execution of the program left the scope containing the definition of the class object, in which case the destructors of all objects created in that scope are implicitly invoked, or
2. The ***delete*** operator was invoked on a class object that was created dynamically, and the destructors for that object and all of its base classes are invoked.

Notes

- When a derived class object must be destroyed, for either of the two reasons above, it will always cause the base class's destructor to be invoked implicitly.
- Destructors are always invoked in the reverse of the order in which the constructors were invoked when the object was constructed. In other words, the derived class destructor is invoked before the base class destructor, recursively, until the lowest level base class destructor is called.
- If a class used the ***new*** operator to allocate dynamic memory, then the destructor should release dynamic memory by called the ***delete*** operator explicitly.
- From the preceding statements, it can be concluded that the derived class releases its dynamic memory before the classes from which it was derived.

To illustrate with an example, consider the following program. The derived class has neither a constructor nor a destructor, but the base class has a default constructor and a default destructor, each of which prints a short message on the standard output stream.

```
class A
{
public:
    A() {cout << "base constructor called.\n";}
    ~A() {cout << "base destructor called.\n";}
};

class B : public A  // has no constructor or destructor
{ };

void main()
{
    B* pb;        //pb is a derived class pointer
    pb = new B;   // allocate an object of the derived class
    delete pb;    // delete the object
}
```

This program will display two lines of output:

```
base constructor called.
base destructor called.
```

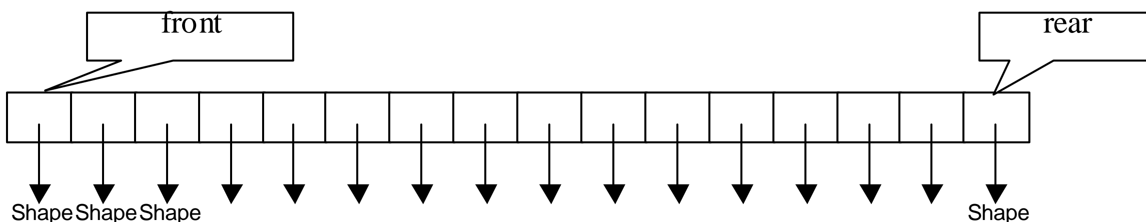
This confirms that the base class destructor and constructor are called implicitly.

2 Virtual Functions

In §1.5 above, I stressed that the compiler always binds pointers to the member functions of the class to which they are declared to point to in their declarations (i.e., at compile time.) This is not a problem if derived objects are never created dynamically. In this case, inheritance is really only buying a savings in the amount of code needed in the program; it does not give the language ***polymorphism***. Polymorphism occurs when objects can alter their behavior dynamically. This is the reason for the ***virtual function specifier***, "virtual."

2.1.2 Example

A program needs a queue that can hold `Shape` objects of all kinds. Since you do not know in advance which cells of the queue will hold which objects, the queue must be able to change what it can hold dynamically, i.e., at run time. If the queue element type is `Shape*`, then each queue element can point to a `Shape` of a different class. However, because the pointers are of type `Shape*`, the program will only be able to access the member functions of the `Shape` class through this pointer, not the member functions of the derived classes.



What is needed is a way to allow the pointer to the base class to access the members of the derived class. This is the purpose of a ***virtual function***. Declaring a function to be virtual in the base class means that if a function in a derived class overrides it and a base class pointer is dereferenced, that pointer will access the member function in the derived class.

2.1.3 Example

```
class CBase
{
public:
    virtual void print()
    { cout<< "Base class print() called. " << endl; }
};

class CDerived : public CBase
{
public:
    void print()
    { cout<< "Derived class print function called." << endl; }
};

void main()
{
    CBase*  baseptr;
    baseptr = new CDerived;
    baseptr->print();
}
```

When this program is run, even though the type of `baseptr` is `CBase*`, the function invoked by the dereference "`baseptr->print()`" will be the `print` function in the derived class, `CDerived`, because the run time environment bound `baseptr->print()` to the derived object when it was assigned a pointer of type `CDerived` and it knew that `print()` was virtual in the base class.

2.2 Virtual Constructors and Destructors

Constructors cannot be virtual. Each class must have its own constructor. Since the name of the constructor is the same as the name of the class, a derived class cannot override it. Furthermore, constructors are not inherited anyway, so it makes little sense.

On the other hand, destructors are rarely invoked explicitly and surprising things can happen in certain circumstances if a destructor is not virtual. Consider the following program.

```
class CBase {
public:
    CBase()
    { cout << "Constructor for CBase called." << endl;}

    ~CBase()
    { cout << "Destructor for CBase called." << endl;}
};
```

```
class CDerived: public Cbase {
public:
    CDerived()
    { cout << "Constructor for CDerived called." << endl;}

    ~CDerived()
    { cout << "Destructor for CDerived called." << endl;}
};

void main() {
    CBase *ptr = new CDerived();
    delete ptr;
}
```

When this is run, the output will be

```
Constructor for CBase called.
Constructor for CDerived called.
Destructor for CBase called.
```

The destructor for the `CDerived` class was not called because the destructor was not declared to be a virtual function, so the call "**`delete`** `ptr`" will invoke the destructor of the class of the pointer itself, i.e., the `CBase` destructor. This is exactly how non-virtual functions work. Now suppose that we make the destructor in the base class virtual. Even though we cannot actually override a destructor, we still need to use the virtual function specifier to force the pointer to be bound to the destructor of the most-derived type, in this case `CDerived`.

```
class CvirtualBase {
public:
    CVirtualBase()
    { cout << "Constructor for CVirtualBase called." << endl; }

    virtual ~CVirtualBase()  // THIS IS A VIRTUAL DESTRUCTOR!!!
    { cout << "Destructor for CVirtualBase called." << endl; }
};

class CDerived: public CvirtualBase {
public:
    CDerived()
    { cout << "Constructor for CDerived called." << endl; }

    ~CDerived()
    { cout << "Destructor for CDerived called." << endl; }
};

void main() {
    CVirtualBase *ptr = new CDerived();
    delete ptr;
}
```

The output of this program will be:

```
Constructor for CVirtualBase called.
Constructor for CDerived called.
Destructor for CDerived called.
Destructor for CVirtualBase called.
```

This is the correct behavior.

In summary, a class C must have a virtual destructor if both of the following conditions are true:

- A pointer p of type C* may be used as the argument to a delete call, and
- It is possible that this pointer may point to an object of a derived class.

There are no other conditions that need to be met. You do not need a virtual destructor if a derived class destructor is called because it went out of scope at run time. You do not need it just because the base class has some virtual functions (which some people will tell you.)

3 Pure Virtual Functions

Suppose that we want to create an `Area` member function in the `Shape` class. This is a reasonable function to include in this base class because every closed shape has area. Every class derived from the `Shape` class can override this member function with its own area function, designed to compute the area of that particular shape. However, the `Area` function in the base class has no implementation because a `Shape` without any particular form cannot have a function that can compute its area. This is an example of a ***pure virtual function***. A pure virtual function is one that has no possible implementation in its own class.

To declare that a virtual function is pure, use the following syntax:

> ***virtual*** *return-type function-name(parameter-list) = 0;*

For example, in the `Shape` class, we can include a pure `Area` function by writing

```
class Shape
{
private:
    Point  Centroid;
public:
    void Move(Point newCentroid);
    Point getCentroid() const;
    virtual double Area() = 0;        // pure Area() function
};
```

4 Abstract Classes

A class with at least one pure virtual function cannot have any objects that are instances of it, because at least one function has no implementation. Such a class is called an ***abstract class***. In contrast, a class that can have objects is called a ***concrete class***. An abstract class can serve as a class interface that can have multiple implementations, by deriving classes from it that do not add any more functionality but provide implementations of the pure virtual functions. It can also serve as an abstraction that is extended in functionality by deriving more specific classes from it.

4.1 Abstract Classes as Interfaces

The following code demonstrates how an abstract class can act like a class interface.

```
template <class T>
class Stack         // an abstract stack base class
{
public:
    Stack() {}
    virtual void pop()                = 0; // all pure
    virtual T    top() const          = 0;
    virtual void push(const T& t)     = 0;
    virtual unsigned int size() const = 0;

};

template <class T>
class VStack : public Stack<T>   // a vector-based implementation
{
private:
    vector<T>  vec;
public:
    VStack() {}
    void pop()                { vec.pop_back(); }
    T    top() const          { return vec[vec.size() - 1];}
    void push (const T& t)    { vec.push_back(t);}
    unsigned int size() const { return vec.size(); }
};
```

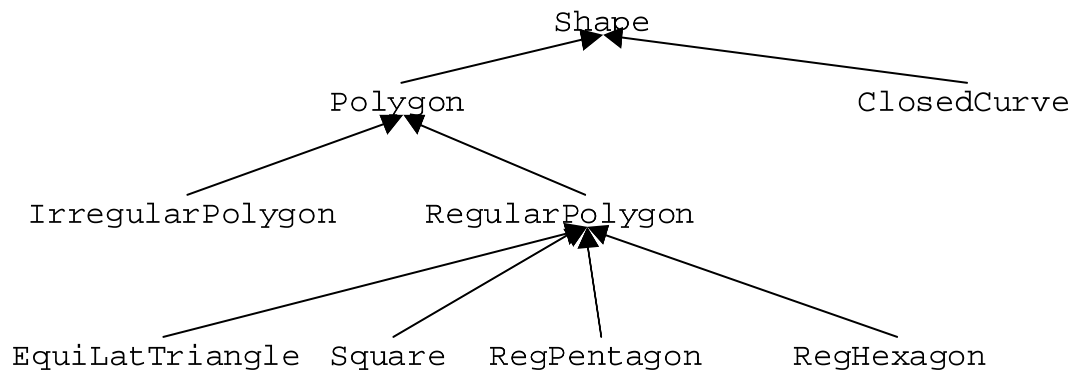
The `VStack` class is derived from the `Stack` abstract base class. Note that both are class templates, not actual classes. Note too the placement of the **`const`** member function qualifier; putting it anyplace else will lead to errors. The `VStack` class overrides all of the pure virtual functions and provides one possible implementation of it. I could also define a linked list implementation, an array implementation, and so on. Each implementation must conform to my abstract base class's interface, but it is free to add private members.

4.2 Abstract Class Hierarchies

When a class is derived from an abstract class and it does not redefine all of the pure virtual functions, it too is an abstract class, because it cannot have objects that represent it. This is often exactly what you want.

Using the `Shape` example again, imagine that we want to build an extensive collection of two-dimensional shapes with the `Shape` class at the root of a tree of derived classes. We can subdivide shapes into polygons and closed curves. Among the polygons we could further partition them into regular and irregular polygons, and among closed curves, we could have other shapes such as ellipses and circles. The class `Polygon` could be derived from the `Shape` class and yet be abstract, because just knowing that a shape is a polygon is still not enough

information to implement an `Area` member function. In fact, the `Area` member function cannot be implemented until the shape is pretty much nailed down. This leads to a multi-level hierarchy that takes the shape of a general tree with the property that the interior nodes are abstract classes and the leaf nodes are concrete classes. A portion of that hierarchy is shown below.



4.2.2