1 Objectives

- Practice fundamental object-oriented programming (OOP) concepts
- Implement an inheritance hierarchy of classes
- Use virtual functions, overriding, and polymorphism in C++
- Use two-dimensional arrays using **array** and **vector**, the two simplest container class templates in the C++ Standard Template Library (STL)
- Use C++ smart pointers

2 Modeling Geometric Shapes

Using simple geometric shapes, this assignment will give you practice with fundamental concepts of OOP, namely, the concepts of abstraction, encapsulation, information hiding, inheritance, and polymorphism.

The geometric shapes considered are simple two-dimensional shapes that can be reasonably depicted textually on the computer screen; specifically, rhombus , rectangle, and two kinds of triangles. As depicted in the UML class diagram on page 1, the classes that model our shapes of interest in this assignment form a single inheritance hierarchy, with the most generalized class **Shape** at the top.

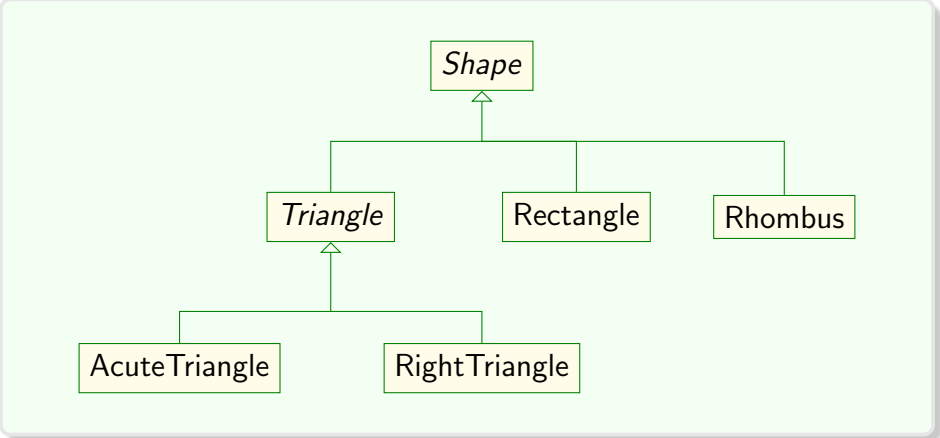


Figure 1: A UML class diagram showing **Shape** inheritance hierarchy.

So let's begin with class **Shape**; that is, let's begin by specifying the common characteristics of our simple geometric shapes.

2.1 Common Attributes of Shape Objects: State (Data)

- A integer *identity number*, distinct from that of the rest of the **Shape** objects.
- An optional user supplied *name* for the shape, such as "Swimming Pool."
- An optional user supplied *description* for the shape, such as "Montreal's Olympic Stadium".

2.2 Common Operations of Shape Objects: Interface

1. A constructor that optionally accepts initial values for the shape's name and description.
2. Three accessor (getter) methods, one for each attribute;
3. Two mutator (setter) methods to set the object's name and description;
4. A method that generates and returns a string representation for the shape object;
5. A method to compute the object's geometric area;
6. A method to compute the object's geometric perimeter;
7. A method to compute the object's *screen area*; that is, *the number of characters that form the textual image of the shape;*
8. A method to compute the object's *screen perimeter*; that is, *the number of characters on the borders of the textual image of the shape;*
9. A method that *draws* a textual image for the shape object on a two dimensional grid, and then returns that grid.
10. Two methods returning, respectively, the *height* and the *width* of the object's *bounding box*; that is, *the smallest rectangular box enclosing the textual image of the shape.*

3 Abstract Shapes

The UML class[1] diagram shown on page 1 represents two abstract classes: **Shape** and **Triangle**.

Encapsulating the attributes and operations common to its objects, class **Shape** is *abstract* because the shapes it models are so general that it would not know how to implement most of the operations it is required to provide; for example, operation 9, to name just one example. As an abstract class, **Shape** not only serves as a common interface to all classes in the inheritance hierarchy, but also makes polymorphism possible through variables of types **Shape*** and **Shape&**.

Based on **Shape**, class **Triangle** models triangular shapes with their bases oriented horizontally. The height of a triangle is length of the line perpendicular to the base from the intersection of the other two sides.

Obviously, class **Triangle** must also be abstract as it too lacks information to implement some of the operations it inherits from **Shape**, including operation 9.

[1] Recall that in UML the name of an abstract class is written in an italic font.

4 Concrete Shapes

Rectangle, **Rhombus**, **RightTriangle** and **AcuteTriangle** represent concrete shape classes, picked specifically because they each can be textually rendered into visually identifiable patterns.

The specific features of these concrete shapes are listed in the following table.

Specialized Features of Concrete Shapes

Features	Concrete Shapes			
Shape name	Rectangle	Rhombus	Right Triangle	Acute Triangle
Construction values	h, w	d, if d is even set $d \leftarrow d+1$	b	b, if b is even set $b \leftarrow b+1$
Computed values			$h = b$	$h = (b+1)/2$
Height of bounding box	h	d	h	h
Width of bounding box	w	d	b	b
Geometric area	hw	$d^2/2$	$hb/2$	$hb/2$
Screen area	hw	$2n(n+1)+1$, $n = \lfloor d/2 \rfloor$	$h(h+1)/2$	h^2
Geometric perimeter	$2(h+w)$	$(2\sqrt{2})d$	$(2+\sqrt{2})h$	$b + 2\sqrt{0.25b^2 + h^2}$
Screen perimeter	$2(h+w)-4$	$2(d-1)$	$3(h-1)$	$4(h-1)$
Sample visual pattern	********* ********* ********* ********* *********	* *** ***** *** *	* ** *** **** *****	* *** ***** ******* *********
Sample pattern dimensions	$w = 9$, $h = 5$	$d = 5$	$b = 5$, $h = b$	$b = 9$, $h = \frac{b+1}{2}$

4.1 Shape Notes

- The unit length is a character; thus, the lengths of the vertical and horizontal attributes of a shape are measured in characters.
- The height and width of a shape's bounding box are *NOT* stored anywhere; they are computed on demand.
- At construction, a **Rectangle** shape requires the values of both its height and width, whereas the other three concrete shapes each require a single value for the length of their respective horizontal attribute.

  In addition, at construction, all shape objects accept optional user specified name and description. The default values are given in Table 1.

Shape		Default shape name	Default shape description
Rhombus		Diamond	A parallelogram with equal sides
Rectangle		Four-sided	Four right angles
Triangle	Acute	Acute triangle	All acute angels
	Right	Right triangle	One right and two acute angles

Table 1: Default shape names and descriptions

5 Task 1 of 2

Implement the **Shape** inheritance class hierarchy described above. It is completely up to you to decide which operations should be virtual, pure virtual, or non-virtual, provided that it satisfies a few simple requirements. Feel free to add your own private methods, if you think they facilitate your implementaion of the class interface.

The amount of coding required for this task is not a lot as your shape classes will be small. Be sure that common behavior (shared methods) and common attributes (shared data) are pushed toward the top of your class hierarchy.

5.1 Some Examples

Sourse code

```
1 Rectangle rect{ 5, 7 };
2 cout << rect.toString() << endl;
3 // or equivalently
4 // cout << rect << endl;
```

Output

```
1  Shape Information
2  -----------------
3  id:             1
4  Shape name:     Four-sided
5  Description:    Four right angles
6  B. box width:   5
7  B. box height:  7
8  Scr area:       35
9  Geo area:       35.00
10 Scr perimeter:  20
11 Geo perimeter:  24.00
12 Static type:    PK5Shape
13 Dynamic type:   9Rectangle
```

The call **rect.toString()** on line 2 of source code generates the entire output shown. However, note that line 4 would produce the same output, as the output operator overload itself internally calls **toString()**.

Line 3 of the output shows that **rect**'s ID number is 1. The ID number of the next shape will be 2, the one after 3, and so on. These unique ID numbers are generated and assigned when shape objects are first constructed.

Line 4-5 of the output show object **rect**'s name and description, and lines 6-7 show the width and height of **rect**'s bounding box, respectively.

Now let's see how the static and dynamic types in lines 12-13 of the output are produced.

Recall that a C++ pointer (or reference) to a class with a virtual member function has two types: *static* type and *dynamic* type; the *static* type is its type as defined in the source code, and thus cannot change, and the *dynamic* type is the type of the object it points at (or references) at runtime, and thus may change during the runtime period.

To get the name of the *static* type of a pointer **p** at runtime you use **typeid(p).name()**, and to get its *dynamic* type you use **typeid(*p).name()**. That's exactly what **toString()** does at line 2, using **this** instead of **p**. You need to include the <**typeinfo**> header for this.

As you can see on lines 12-13, **rect**'s static type name is **PK5Shape** and it's dynamic type name is **9Rectangle**. The actual names returned by these calls are implementation defined. For example, the output above was generated under g++ (GCC) 7.4.0, where **PK** in **PK5Shape** means "pointer to ~~**konst**~~ **const**", and **5** in **5Shape** means that the name "**Shape**" that follows it is **5** character long.

Microsoft VC++ produces a more readable output as shown below.

```
1 Rectangle rect{ 5, 7 };
2 cout << rect.toString() << endl;
3 // or equivalently
4 // cout << rect << endl;
```

```
Shape Information
-----------------
id:             1
Shape name:     Four-sided
Description:    Four right angles
B. box width:   5
B. box height:  7
Scr area:       35
Geo area:       35.00
Scr perimeter:  20
Geo perimeter:  24.00
Static type:    class Shape const *
Dynamic type:   class Rectangle
```

Note that in the source code above the object **rect** is NOT a pointer (or reference) and hence NOT polymorphic; however, the call **rect.toString()** on line 2 represents **rect** inside **toString()** by the *this* pointer, where object **this* is polymorphic.

Here is an example of a **Rhombus** object:

```
5 Rhombus
6    ace{ 16, "Ace", "Ace of diamond" };
7 // cout << ace.toString() << endl;
8 // or, equivalently:
9 cout << ace << endl;
```

```
Shape Information
-----------------
id:             2
Shape name:     Ace
Description:    Ace of diamond
B. box width:   17
B. box height:  17
Scr area:       145
Geo area:       144.50
Scr perimeter:  32
Geo perimeter:  48.08
Static type:    class Shape const *
Dynamic type:   class Rhombus
```

Notice that in line 6, the supplied height, 16, is invalid because it is even; to correct it, **Rhombus**'s constructor uses the next odd integer, 17, as the diagonal of object **ace**.

Again, lines 7 and 9 would produce the same output; the difference is that the call to **toString()** is implicit in line 9.

Here are examples of **AcuteTriangle** and **RightTriangle** shape objects.

```
10      AcuteTriangle at{ 17 };;
11      cout << at << endl;
12
13       /*equivalently:
14
15      Shape *atptr = &at;
16      cout << *atptr << endl;
17
18      Shape &atptr = at;
19      cout << atptr << endl;
20      */
```

```
Shape Information
-----------------
id:             3
Shape name:     Acute triangle
Description:    All acute angels
B. box width:   17
B. box height:  9
Scr area:       81
Geo area:       76.50
Scr perimeter:  32
Geo perimeter:  41.76
Static type:    class Shape const *
Dynamic type:   class AcuteTriangle
```

```
21  RightTriangle
22      rt{ 10, "Carpenter's square" };
23  cout << rt << endl;
```

```
Shape Information
-----------------
id:             4
Shape name:     Carpenter's square
Description:    One right and two acute angles
B. box width:   10
B. box height:  10
Scr area:       55
Geo area:       50.00
Scr perimeter:  27
Geo perimeter:  34.14
Static type:    class Shape const *
Dynamic type:   class RightTriangle
```

5.2 Polymorphic Magic

It is important to note that none of the objects **rect**, **ace** , **at**, and **rt** is polymorphic because none of them is a pointer or a reference. Polymorphic magic happens through the second argument in the calls to the output **operator**<< at lines 2, 8, 11, and 23. For example, consider the call **cout**<<**rt** at lines 23, which can be equivalently written as **operator**<<**(cout, rt)**. The second argument in the call, **rt**, corresponds to the second parameter of the output operator overload:

```
ostream& operator<< (ostream& out, const Shape& shape);
```

Specifically, in line 23, **rt** in the expression **cout**<<**rt** binds to parameter **shape**, which is a reference, and thus the object it references is polymorphic. That means, for example, that if **shape** references a rhombus object, then the call **shape.geoArea()** calls rhombus's **geoArea()**, if **shape** references a rectangle, then **shape.geoArea()** calls rectangle's **geoArea()**, and so on.

5.3 Shape's Draw Function

Finally, the **draw** member function, prototyped as a pure virtual member function in **Shape**,

```
virtual vector<vector<char>> draw(char fChar = '*', char bChar = ' ') const = 0;
```

forcing all concrete derived classes to implement the function. Using the values of the parameters **fChar** and **bChar** as foreground and background characters, respectively, the function "draws" an image of the shape on "a two dimensional grid" representing the shape's bounding box, and then returns that grid.

5.4 The Grid

The two dimensional grid must be named and implemented as follows:

```
using Grid = vector<vector<char>>; // a vector of vectors of chars
```

One way to print the grid is to overload the **operator**<< as follows:

```
ostream& operator<< (ostream& sout, const Grid& grid)
{
    for (const auto& row : grid) // for each row vector in the grid
    {
        for (const auto& ch : row) // for each char in the row vector
        {
            sout << ch;
        }
        sout << endl; // line break
    }
    return sout;
}
```

5.5 Examples Continued

```
24
25 Grid aceBox = ace.draw('+', '.');
26 cout << aceBox << endl;
```

```
........+........
.......+++.......
......+++++......
.....+++++++.....
....+++++++++....
...+++++++++++...
..+++++++++++++..
.+++++++++++++++.
+++++++++++++++++
.+++++++++++++++.
..+++++++++++++..
...+++++++++++...
....+++++++++....
.....+++++++.....
......+++++......
.......+++.......
........+........
```

```
27
28 Grid rectBox = rect.draw();
29 cout << rectBox << endl;
```

```
*****
*****
*****
*****
*****
*****
*****
```

```
30
31 Grid atBox = at.draw('^');
32 cout << atBox << endl;
```

```
         ^
        ^^^
       ^^^^^
      ^^^^^^^
     ^^^^^^^^^
    ^^^^^^^^^^^
   ^^^^^^^^^^^^^
  ^^^^^^^^^^^^^^^
 ^^^^^^^^^^^^^^^^^
^^^^^^^^^^^^^^^^^^^
```

```
33
34 Grid rtBox = rt.draw('|');
35 cout << rtBox << endl;
```

```
|
||
|||
||||
|||||
||||||
|||||||
||||||||
|||||||||
||||||||||
```

Note that a shape object can be redrawn using different foreground and background characters:

```
37 rtBox = rt.draw('\\', 'o');
38 cout << rtBox << endl;
```

```
\oooooooooo
\\oooooooo
\\\ooooooo
\\\\oooooo
\\\\\ooooo
\\\\\\oooo
\\\\\\\ooo
\\\\\\\\oo
\\\\\\\\\o
\\\\\\\\\\
```

```
39 aceBox = ace.draw('o');
40 cout << aceBox << endl;
```

```
          o
         ooo
        ooooo
       ooooooo
      oooooooo
     ooooooooo
    ooooooooooo
   oooooooooooo
  ooooooooooooo
 oooooooooooooo
  ooooooooooooo
   oooooooooooo
    ooooooooooo
     oooooooooo
      ooooooooo
       ooooooo
        ooooo
         ooo
          o
```

6 Task 2 of 2

A slot machine is a classic coin-operated rip-off gambling machine. Traditional slot machines have three reels and one payline, with each reel labeled with about two dozen symbols. To use one, you insert coins into a slot and pull a handle that activates the spinning of the reels. After spinning a random number of times, the reels come to rest, showing three symbols lined up across the *payline*. If two or more of the symbols on the payline match, you will win a cash payout, which the slot machine dispenses back to you.

Your task is to design and implement a class that models a simple slot machine, using the geometric shapes you created in Task 1 as the visual symbols on the reels. The slot machine is to have three reels, each with 4 symbols (shapes), and each symbol in 25 available sizes.

6.1 Sample Run

```
auto main() -> int
{   SlotMachine slot_machine;// create a slot machine object
    slot_machine.run();      // run our slot machine until the player decides
       return 0;             // to stop, or until the player runs out of tokens
}
```

```

Welcome to 3-Reel Lucky Slot Machine Game!
Each reel will randomly display one of four shapes, each in 25 sizes.
To win 3 x bet Get 2 similar shapes AND 2 shapes with equal Scr Areas
To win 2 x bet Get 3 similar shapes
To win 1 x bet Get (Middle) Scr Area > (Left + Right) Scr Areas
To win or lose nothing Get same Left and Right shapes
Otherwise, you lose your bet.
You start with 10 free slot tokens!

How much would you like to bet (enter 0 to quit)? 1
+----+-----+---+
| *  | *   | * |
| ** | **  |   |
|    | *** |   |
+----+-----+---+
(Right triangle, 2, 2) (Right triangle, 3, 3) (Diamond, 1, 1)
Middle > Left + Right, in Screen Areas
Congratulations! you win your bet: 1
You now have 11 tokens
```

```
How much would you like to bet (enter 0 to quit)? 2
+---------------+--------------+---------------+
|       *       | *            |       *       |
|      ***      | **           |      ***      |
|     *****     | ***          |     *****     |
|    *******    | ****         |    *******    |
|   *********   | *****        |   *********   |
|  ***********  | ******       |  ***********  |
| ************* | *******      | ************* |
|  ***********  | ********     |               |
|   *********   | *********    |               |
|    *******    | **********   |               |
|     *****     | ***********  |               |
|      ***      | ************ |               |
|       *       |              |               |
+---------------+--------------+---------------+
(Diamond, 13, 13) (Right triangle, 12, 12) (Acute triangle, 7, 13)
Oh No!
You lose your bet
You now have 9 tokens

How much would you like to bet (enter 0 to quit)?5
```

```
+-------------+---------------+-----------+
| *           |       *       | *         |
| **          |      ***      | **        |
| ***         |     *****     | ***       |
| ****        |    *******    | ****      |
| *****       |   *********   | *****     |
| ******      |  ***********  | ******    |
| *******     | ************* | *******   |
| ********    |  ***********  | ********  |
| *********   |   *********   | ********* |
| **********  |    *******    |           |
| ********** |     *****     |           |
|             |      ***      |           |
|             |       *       |           |
+-------------+---------------+-----------+
(Right triangle, 11, 11) (Diamond, 13, 13) (Right triangle, 9, 9)
Lucky this time!
You don't win, you don't lose, your are safe!
You now have 9 tokens

How much would you like to bet (enter 0 to quit)?
```

6.2 Slot Machine Algorithm

The **run()** method called in line 2 of the source code on page 11 performs the following algorithm:

Algorithm 1 Slot Machine's run() Algorithm

1: Prepare an array R of three elements to represent the three reels of the slot machine.
   Each element (reel) is a pointer to a **Shape** object managed dynamically.
2: **while** user has slot tokens and wants to play **do**
3: Prompt for and read a bet ▷ an integer representing slot tokens
4: **for each** $k = 0, 1, 2$ **do** ▷ have reel R_k point at a random shape
5: Generate a random integer n, $0 \leq n \leq 3$
6: Generate a random width w, $1 \leq w \leq 25$
7: **if** $n = 0$ **then**
8: Let reel R_k point to a **Rhombus** object of width w
9: **else if** $n = 1$ **then**
10: Let reel R_k point to a **AccuteTriangle** object of width w
11: **else if** $n = 2$ **then**
12: Let reel R_k point to a **RightTriangle** object of width w
13: **else**
14: Generate a random height h, $1 \leq h \leq 25$
15: Let reel R_k point to a**Rectangle** object of width w and height h
16: **end if**
17: **end for**
18: Print the three reels side by side horizontally. ▷ simulates the symbols on the payline
19: Based on the symbols on the payline, compute the prize payout (tokens won) as follows:

$$\text{tokens won} = \begin{cases} 3 \times \text{bet} & \text{if any two symbols are similar in shape and any two match in screen areas} \\ 2 \times \text{bet} & \text{if all three symbols are similar in shape} \\ 1 \times \text{bet} & \text{if screen area of middle symbol}>\text{sum of screen areas of left and right symbols} \\ 0 \times \text{bet} & \text{if the left and right symbols are similar} \\ -1 \times \text{bet} & \text{otherwise, player loses the bet} \end{cases}$$

20: Update player's slot tokens using the prize payout
21: Display whether the player has won, lost, or neither
22: **end while**

6.2.1 Implementation of Step 1 of Slot Machine Algorithm

Using modern C++ smart pointers, implement step 1 as follows:

```
std::array<std::unique_ptr<Shape>, 3>  reel{};
```

so that the three reels are represented by three unique pointers **reel[0]**, **reel[1]**, **reel[2]**, all currently pointing at **nullptr**. The actual shape objects are created and assigned at steps 8, 10, 12, ans 15. For example, step 8 involves the following step:

```
reel[k].reset(new Rhombus(w));
```

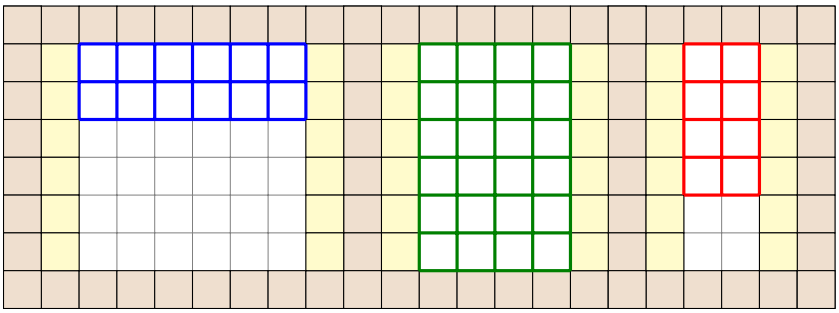
The unique pointer **reel[k]** automatically "**delete**s" the object it currently manages (points at), before resetting the pointer to a new dynamically allocated **Shape** object. That is, you the programmer can enjoy life without having to be concerned about "*deleting*" dynamically created resources ever again. Welcome to modern C++!

6.2.2 Implementation of Step 18 of Slot Machine Algorithm

This step must obviously obtain the bounding boxes corresponding to the three **shape** objects currently pointed at by the three reel objects:

```
Grid box_0 = reel[0]->draw();
Grid box_1 = reel[1]->draw();
Grid box_2 = reel[2]->draw();
```

This step then prints the three bounding boxes side by side in a row, with their top borders aligned. To make the output look a bit nicer, **slot_machine** should decorate the output as indicated in the following figure. The blue, green, and red grids in the figure represent the bounding boxes; yellow girds represent blank vertical margins; brown squares represent border characters. See output on page 12 for specifics.



6.3 Class SlotMachine

Despite its lengthy description, the **SlotMachine** class is straightforward:

```
class SlotMachine
{
   std::array<std::unique_ptr<Shape>, 3>  reel{};
   void make_shapes();      // Step 4
   void make_shape(int k); // Steps 5-16
   void display();          // Step 18
public:
   ShapeSlotMachine() = default;
   SlotMachine(const SlotMachine&)  = delete; // copy ctor
   SlotMachine(SlotMachine&&)       = delete; // move ctor
   SlotMachine& operator=(const SlotMachine&)  = delete; // copy assignment
   SlotMachine& operator=(SlotMachine&&)       = delete; // move assignment
   virtual ~ShapeSlotMachine() = default;
   void run();  // see algorithm on page 13
};
```

6.4 FYI

The following two features were dropped from the original version of the slot machine program in order to make the assignment workload lighter:

- Simulate the spinning of the Slot's reels
- Implement the concepts of a slot machine reel into a **Reel** class whose objects are responsible for managing their own internal needs, including the use of dynamic memory.

6.5 Sample Run Continued

```
How much would you like to bet (enter 0 to quit)? 3
+---------------+------------------------+---------------+
|       *       |           *            |       *       |
|      ***      |          ***           |      ***      |
|     *****     |         *****          |     *****     |
|    *******    |        *******         |    *******    |
|   *********   |       *********        |   *********   |
|  ***********  |      ***********       |  ***********  |
| ************* |     *************      |               |
|               |    ***************     |               |
|               |   *****************    |               |
|               |  *******************   |               |
|               | ********************* |               |
+---------------+------------------------+---------------+
(Acute triangle, 7, 13) (Acute triangle, 11, 21) (Acute triangle, 6, 11)
Three similar shapes
Congratulations! you win 2 times your bet: 6
You now have 15 tokens
```

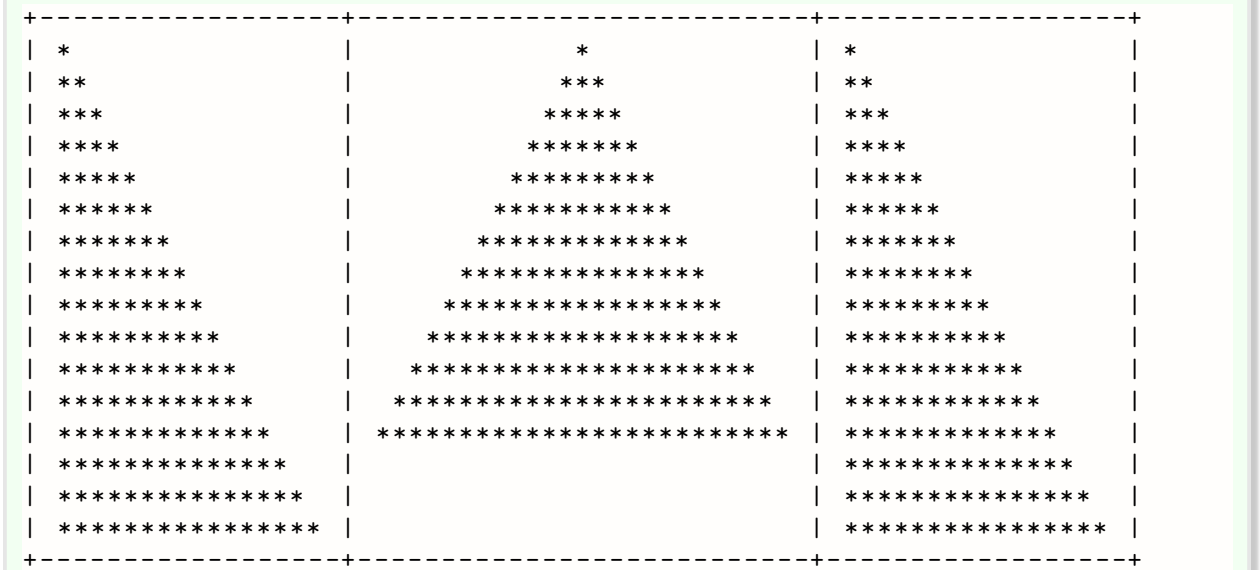
```
How much would you like to bet (enter 0 to quit)? -1
No negative bets, try again!

How much would you like to bet (enter 0 to quit)? -2
No negative bets, try again!

How much would you like to bet (enter 0 to quit)? 50
You can't bet more than 15, try again!

How much would you like to bet (enter 0 to quit)? 40
You can't bet more than 15, try again!

How much would you like to bet (enter 0 to quit)? 1
```




```
Jackpot! 2 Similar Shapes AND 2 Equal Screen Areas
Congratulations! you win 3 times your bet: 3
You now have 18 tokens

How much would you like to bet (enter 0 to quit)? 0
Thank you for playing, come back soon!
Be sure you cash in your remaing 18 tokens at the bar!
Game Over.
```

Deliverables

Header files: Shape.h, Triangle.h, Rectangle.h, Rhombus.h, AcuteTriangle.h, RightTriangle.h, SlotMachine.h,

Implementation files: Shape.cpp, Triangle.cpp, Rectangle.cpp, Rhombus.cpp, AcuteTriangle.cpp, RightTriangle.cpp, SlotMachine.cpp, and shape_slotmachine_driver.cpp

README.txt A text file (see the course outline).

7 Evaluation Criteria

Evaluation Criteria

Functionality	Testing correctness of execution of your program, Proper implementation of all specified requirements, Efficiency	60%
OOP style	Encapsulating only the necessary data inside your objects, Information hiding, Proper use of C++ constructs and facilities. No use of operator operator **delete**. No C-style coding and memory functions such as **malloc**, **alloc**, **realloc**, **free**, etc.	20%
Documentation	Description of purpose of program, Javadoc comment style for all methods and fields, comments on non-trivial pieces of code in submitted programs	10%
Presentation	Format, clarity, completeness of output, user friendly interface	5%
Code readability	Meaningful identifiers, indentation, spacing, localizing variables	5%