COMP5421/1-BB	Assignment 4	Due by July 31, 2020

1 Purpose

- Practice fundamental object-oriented programming (OOP) concepts
- Implement an inheritance hierarchy of classes C++
- Learn about 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 modern C++ smart pointers, avoiding calls to the `delete` operator for good!

2 Overview

Using simple two-dimensional geometric shapes, such as rhombus, rectangle, and triangles, this assignment will give you practice with fundamental concepts of OOP; namely, abstraction, inheritance and polymorphism, and encapsulation and information hiding.

The assignment starts by abstracting the essential attributes and operations common to the geometric shapes of interest; recall that abstraction is a process that involves modeling a physical entity or activity, focusing on the interface rather than on the implementation.

Once the desired common operations are specified, the assignment moves on to abstracting the desired operations specific to each of the shapes of interest.

Next, you will be tasked to implement the shape abstractions using the C++ features that support encapsulation, information hiding, inheritance and polymorphism.

Finally, you will be tasked to demonstrate polymorphism using modern C++ smart pointers.

3 Modeling 2D Geometric Shapes

3.1 Common Attributes: State (Data)

For simplicity, we specify only three attributes:

- a unique *identity number*,
- a *name*, such as "Swimming Pool", and
- a *description* for the shape, such as "Montreal's Olympic Stadium".

3.2 Common Operations: Interface (Operations)

The following is a wish list of operations that every 2D geometric object of interest in this assignment is expected to provide:

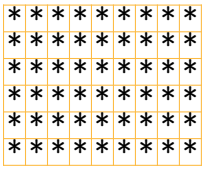
1. A constructor that accepts and sets the initial values of the shape's name and description.
2. Three accessor (getter) methods, one for each attribute;
3. Two mutator (setter) methods for each attribute except for the identity number;
4. A method that generates and returns a string representation of the shape;
5. A method to compute and return the shape's area;
6. A method to compute and return the shape's perimeter;
7. A method to compute the shape's *screen area*, which is the number of characters that form the textual image of the shape;
8. A method to compute the shape's *screen perimeter*, which is the number of characters on the borders of the textual image of the shape;
9. A method that *draws* a textual image of the shape object on a two dimensional grid that represents the shape's bounding box.

   The bounding box of a shape is a rectangular grid that is aligned with the x and y axes and is large enough to exactly bound the shape.
10. A method to return the height of the shape's bounding box;
11. A method to return the width of the shape's bounding box;

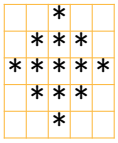
4 Modeling Specialized 2D Geometric Shapes

The geometric shapes considered are four simple two-dimensional shapes which can be textually rendered into visually identifiable patterns on the computer screen, such as rhombus , rectangle, and two kinds of isosceles triangles.

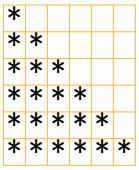
Here are examples of such patterns:



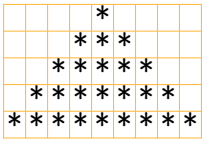
Rectangle, bounded by 6×9 box



Rhombus, bounded by 5×5 box



Right Triangle, bounded by 6×6 box



Acute Triangle, bounded by 5×9 box

There are several ways to classify 2D shapes, but we use the following, which is specifically designed for you to gain experience with implementing inheritance and polymorphism in C++:

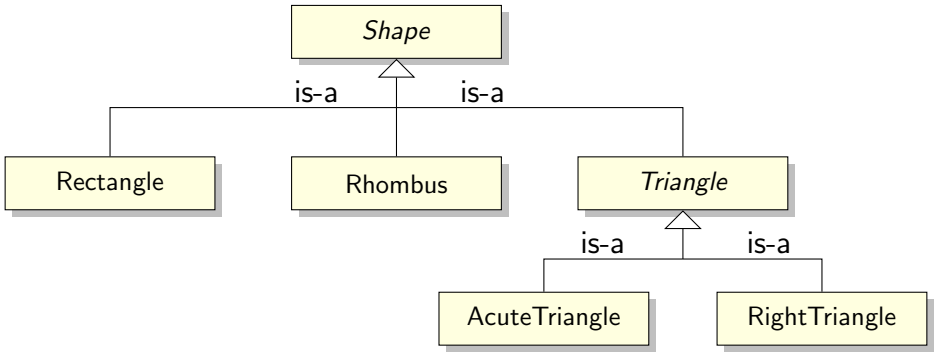


Figure 1: A UML class diagram showing an inheritance hierarchy specified by two abstract classes *Shape* and *Triangle*, and by four concrete classes Rectangle, Rhombus, AcuteTriangle, and RightTriangle.

Encapsulating the attributes and operations common to all shapes, the class *Shape* must be *abstract*[1] because by design the shapes it models are so general that it simply would not know how to implement most of the operations it specifies; to name just one example, it would not know how to draw any shapes without knowledge about the specific shapes.

As a base class, *Shape* serves as a common interface to all classes in the inheritance hierarchy.

As an abstract class, *Shape* makes polymorphism in C++ possible through variables of types *Shape** and *Shape&*.[2]

Similarly, the Triangle class must be abstract, since it would have no idea how to, for example, draw the specific shapes it generalizes.

5 Concrete Shapes

The classes classes Rectangle, Rhombus, RightTriangle and AcuteTriangle are concrete classes because they each fully implement their respective interface.

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

[1] Recall that a C++ class is said to be abstract if it has at least one pure virtual function.
You cannot define an object of an abstract class Foo, but you can define variables of types Foo* and Foo&. The compiler ensures that all calls to a virtual function (pure or not) via Foo* and Foo& are polymorphic calls.
Any class derived from an abstract class will itself be abstract unless it overrides all the pure virtual functions it inherits.

[2] A pointer (or reference) to a class object with a virtual member function has two types:

- *static* type: refers to its type as defined in the source code and thus cannot change, and
- *dynamic* type: refers to the type of the object it points at (or references) at runtime and thus can change during runtime.

Features	Rectangle	Rhombus	Right Triangle	Acute Triangle
Construction values	h, w	d, if d is odd; else $d \leftarrow d+1$	b	b, if b is odd; else $b \leftarrow b+1$
Dependent attribute			$h = b$	$h = (b+1)/2$
Height of bounding box	h	d	h	h
Width of bounding box	w	d	b	b
Geometric area	hw	$d^2/2$	$hb/2$	$hb/2$
Geometric perimeter	$2(h+w)$	$(2\sqrt{2})d$	$(2+\sqrt{2})h$	$b+\sqrt{b^2+4h^2}$
Screen area	hw	$2n(n+1)+1$, $n = \lfloor d/2 \rfloor$	$h(h+1)/2$	h^2
Screen perimeter	$2(h+w)-4$	$2(d-1)$	$3(h-1)$	$4(h-1)$
Sample visual pattern	********* ********* ********* ********* *********	* *** ***** *** *	* ** *** **** *****	* *** ***** ******* *********
Sample pattern dimensions	$w = 9$, $h = 5$	$d = 5$	$b = 5$, $h = b$	$b = 9$, $h = \frac{b+1}{2}$
Default name	Rectangle	Diamond	Ladder	Wedge
Default description	Four right angles	Parallelogram with equal sides	One right and two acute angles	Three acute angles

h : height, w : width, b : base, d : diagonal

5.1 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.

6 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.

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 operations) and common attributes (shared data) are pushed toward the top of your class hierarchy; for example:

6.1 Modeling 2D Triangle Shapes

6.1.1 Common Attributes: State (Data)

- Base, the horizontal attribute of a triangle
- Height, the vertical attribute of a triangle

6.1.2 Common Operations: Interface (Operations)

- A method to return the triangle's height
- A method to return the triangle's width
- A method to return the geometric area of the triangle

7 Some Examples

Sourse code

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

Output

```
Shape Information
-----------------
id:             1
Shape name:     Rectangle
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:    PK5Shape
Dynamic type:   9Rectangle
```

The call `rect.toString()` on line 2 of the 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.

Lines 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 `rect`'s static and dynamic types are produced on lines 12-13 of the output.

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) 9.3.0, 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;
```

```
1  Shape Information
2  -----------------
3  id:             1
4  Shape name:     Rectangle
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:    class Shape const *
13 Dynamic type:   class Rectangle
```

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 &atref = at;
19    cout << atref << endl;
20    */
```

```
Shape Information
-----------------
id:             3
Shape name:     Wedge
Description:    All acute angles
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
```

7.1 Polymorphic Magic

Note that on line 1 in the source code above, `rect` is a regular object variable, as opposed to a pointer (or reference) variable pointing to (or referencing) an object; as such, `rect` cannot make polymorphic calls. That's because in C++ the calls made by any regular object, such as `rect`,`ace`, `at`, and `rt`, to any function (virtual or not) are bound at compile time (early binding).

Polymorphic magic happens through the second argument in the calls to the output `operator<<` at lines 4, 9, 11, and 23. For example, consider the call `cout << rt` on line 23, which is equivalent to `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, `rt` in line 23 binds to parameter `shape`, which is a reference, and as such, can call virtual functions of `Shape` polymorphically. That means, the decision as to which function to invoke depends on the type of the object referenced by `shape` at run time (late binding).

For example, if parameter `shape` references a rhombus object, then the call `shape.geoArea()` will call rhombus's `geoArea()`, if `shape` references a rectangle, then `shape.geoArea()` will call rectangle's `geoArea()`, and so on.

However, although `rect` on line 2 is not a reference or a pointer, `rect`, the invoking object in the call `rect.toString()`, is represented inside `Shape::toString()` by the `this` pointer, which in fact can call virtual functions of `Shape`, the base class, polymorphically.

7.2 Shape's Draw Function

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

```
virtual Grid 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.

7.3 The Grid

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

7.4 Examples Continued

```

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

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

```

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

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

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

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

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

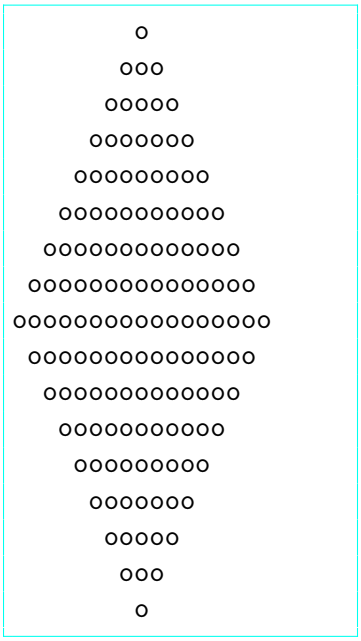
```
-
--
---
----
-----
------
-------
--------
---------
----------
```

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

```
36
37 rtBox = rt.draw('\\', 'o');
38 cout << rtBox << endl;
```

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

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



8 Polymorphism in C++

Notice that none of the calls to the virtual function `draw` on lines 25, 28, 31 ,34, 37, and 40 is polymorphic, because the invoking objects in the calls are all represented by regular variables.

To draw the same shapes polymorphically, the variables that call the virtual function `draw` must be either a reference or a pointer.

Here is an example of calling the draw functions polymorphically through `Shape&`:

```
1  void poly_draw_shape_by_ref(const Shape& shape, char foreground= '*', char background= ' ')
2  {  Grid shape_box = shape.draw(foreground, background);
3     cout << shape_box << endl;
4  }
5  int main()
6  {   Rectangle rect{ 5, 7 };
7     Rhombus   ace{ 16, "Ace", "Ace of diamond" };
8     AcuteTriangle at{ 17 };
9     RightTriangle rt{ 10, "Carpenter's square" };
10    poly_draw_shape_by_ref(rect);
11    poly_draw_shape_by_ref(ace, '+', '.');
12    poly_draw_shape_by_ref(at, '^');
13    poly_draw_shape_by_ref(rt, '-');
14    return 0;
15 }
```

Here is an example of calling the draw functions polymorphically through `Shape*`. Avoid this approach, because it MUST explicitly call `delete`; instead, prefer using `std::unique_ptr`.

```
1  void poly_draw_shape_by_ptr(const Shape* pShape, char foreground= '*', char background= ' ')
2  {  Grid shape_box = pShape->draw(foreground, background);
3     cout << shape_box << endl;
4  }
5  int main()
6  {  Shape* pRect = new Rectangle{ 5, 7 };
7     Shape* pAce = new Rhombus   { 16, "Ace", "Ace of diamond" };
8     Shape* pAt = new AcuteTriangle { 17 };
9     Shape* pRt = new RightTriangle { 10, "Carpenter's square" };
10    poly_draw_shape_by_ptr(pRect);
11    poly_draw_shape_by_ptr(pAce, '+', '.');
12    poly_draw_shape_by_ptr(pAt, '^');
13    poly_draw_shape_by_ptr(pRt, '-');
14    delete pRect;
15    delete pAce;
16    delete pAt;
17    delete pRt;
18    return 0;
19 }
```

Here is an example of calling the draw functions polymorphically using `std::unique_ptr`:

```
int main()
{
   std::unique_ptr<Shape> rectShape{ new Rectangle{5, 7} };
   Grid rectBox = rectShape->draw();
   cout << rectBox << endl;

   std::unique_ptr<Shape> aceShape{ new Rhombus{16, "Ace", "Ace of diamond"} };
   Grid aceBox = aceShape->draw('+', '.');
   cout << aceBox << endl;

   std::unique_ptr<Shape> atShape{ new AcuteTriangle(17)};
   Grid atBox = atShape->draw('^');
   cout << atBox << endl;

   std::unique_ptr<Shape> rtShape{ new RightTriangle(10, "Carpenter's square")};
   Grid rtBox = rtShape->draw('-');
   cout << rtBox << endl;

   // no need to delete any resources!
   // Notice that are 4 calls to the new operator but NONE to the delete operator
   return 0;
}
```

9 Task 2 of 2

9.1 Background

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.

9.2 Your Task

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.

9.3 Sample Run

```
int main()
{
   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)? 2
+---+-------+---------------------+
| * |   *   |          *          |
|   |  ***  |         ***         |
|   | ***** |        *****        |
|   |  ***  |       *******       |
|   |   *   |      *********      |
|   |       |     ***********     |
|   |       |    *************    |
|   |       |   ***************   |
|   |       |  *****************  |
|   |       | ******************* |
|   |       |  *****************  |
|   |       |   ***************   |
|   |       |    *************    |
|   |       |     ***********     |
|   |       |      *********      |
|   |       |       *******       |
|   |       |        *****        |
|   |       |         ***         |
|   |       |          *          |
+---+-------+---------------------+
(Diamond, 1, 1) (Diamond, 5, 5) (Diamond, 19, 19)
Three similar shapes
Congratulations! you win 2 times your bet: 4
You now have 14 tokens

How much would you like to bet (enter 0 to quit)? 5
+------------------+-------------+-------+
| **************** | *           |   *   |
| **************** | **          |  ***  |
|                  | ***         | ***** |
|                  | ****        |       |
|                  | *****       |       |
|                  | ******      |       |
|                  | *******     |       |
|                  | ********    |       |
|                  | *********   |       |
|                  | **********  |       |
|                  | ********** |       |
+------------------+-------------+-------+
(Rectangle, 2, 17) (Ladder, 11, 11) (Wedge, 3, 5)
Middle > Left + Right, in Screen Areas
Congratulations! you win your bet: 5
You now have 19 tokens
```

```
How much would you like to bet (enter 0 to quit)? 9
+-----+----+----------+
|  *  | *  | *        |
| *** | ** | **       |
|     |    | ***      |
|     |    | ****     |
|     |    | *****    |
|     |    | ******   |
|     |    | *******  |
|     |    | ******** |
+-----+----+----------+
(Wedge, 2, 3) (Ladder, 2, 2) (Ladder, 8, 8)
Oh No!
You lose your bet
You now have 10 tokens

How much would you like to bet (enter 0 to quit)? 5
+--------+----------------------+--------+
| *      |           *          | *      |
| **     |          ***         | **     |
| ***    |         *****        | ***    |
| ****   |        *******       | ****   |
| *****  |       *********      | *****  |
| ****** |      ***********     | ****** |
|        |     *************    |        |
|        |    ***************   |        |
|        |   *****************  |        |
|        |  ******************** |        |
|        | ********************** |        |
+--------+----------------------+--------+
(Ladder, 6, 6) (Wedge, 11, 21) (Ladder, 6, 6)
Jackpot! 2 Similar Shapes AND 2 Equal Screen Areas
Congratulations! you win 3 times your bet: 15
You now have 25 tokens

How much would you like to bet (enter 0 to quit)? 5
+--------------+---------+----------------+
| ************ |    *    | ************** |
| ************ |   ***   | ************** |
| ************ |  *****  | ************** |
| ************ | ******* | ************** |
| ************ |  *****  | ************** |
| ************ |   ***   | ************** |
```

```
| ************** |     *     | ***************** |
| ************** |           | ***************** |
|                |           | ***************** |
|                |           | ***************** |
|                |           | ***************** |
|                |           | ***************** |
|                |           | ***************** |
|                |           | ***************** |
|                |           | ***************** |
|                |           | ***************** |
|                |           | ***************** |
|                |           | ***************** |
|                |           | ***************** |
|                |           | ***************** |
|                |           | ***************** |
|                |           | ***************** |
+----------------+-----------+-------------------+
(Rectangle, 8, 13) (Diamond, 7, 7) (Rectangle, 24, 15)
Lucky this time!
You don't win, you don't lose, your are safe!
You now have 25 tokens
```

```

How much would you like to bet (enter 0 to quit)? 1
+------------------+------------------+--------------------+
|         *        |         *        |          *         |
|        ***       |        ***       |         ***        |
|       *****      |       *****      |        *****       |
|      *******     |      *******     |       *******      |
|     *********    |     *********    |      *********     |
|    ***********   |    ***********   |     ***********    |
|   *************  |   *************  |    *************   |
| *************** | *************** |  ***************  |
|                  |                  | ***************** |
+------------------+------------------+--------------------+
(Wedge, 8, 15) (Wedge, 8, 15) (Wedge, 9, 17)
Jackpot! 2 Similar Shapes AND 2 Equal Screen Areas
Congratulations! you win 3 times your bet: 3
You now have 28 tokens
```

```
How much would you like to bet (enter 0 to quit)? 50
You can't bet more than 28, try again!

How much would you like to bet (enter 0 to quit)? 28
+-----+--------+--------------------------+
| *** | ****** |             *            |
| *** | ****** |            ***           |
| *** | ****** |           *****          |
| *** | ****** |          *******         |
| *** | ****** |         *********        |
| *** | ****** |        ***********       |
| *** |        |       *************      |
| *** |        |      ***************     |
| *** |        |     *****************    |
| *** |        |    *******************   |
| *** |        |   *********************  |
| *** |        |  ********************** |
| *** |        | ***********************|
| *** |        |  ********************** |
| *** |        |   *********************  |
| *** |        |    *******************   |
| *** |        |     *****************    |
| *** |        |      ***************     |
| *** |        |       *************      |
|     |        |        ***********       |
|     |        |         *********        |
|     |        |          *******         |
|     |        |           *****          |
|     |        |            ***           |
|     |        |             *            |
+-----+--------+--------------------------+
(Rectangle, 19, 3) (Rectangle, 6, 6) (Diamond, 25, 25)
Oh No!
You lose your bet
You now have 0 tokens
You just ran out of tokens. Better luck next time!
Game Over.
```

9.4 Another sample run

```
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)? 4
+-------------------------+----------+----------+
|            *            | *        | ******** |
|           ***           | **       | ******** |
|          *****          | ***      | ******** |
|         *******         | ****     |          |
|        *********        | *****    |          |
|       ***********       | ******   |          |
|      *************      | *******  |          |
|     ***************     | ******** |          |
|    *****************    |          |          |
|   *******************   |          |          |
|  *********************  |          |          |
| ********************** |          |          |
|  *********************  |          |          |
|   *******************   |          |          |
|    *****************    |          |          |
|     ***************     |          |          |
|      *************      |          |          |
|       ***********       |          |          |
|        *********        |          |          |
|         *******         |          |          |
|          *****          |          |          |
|           ***           |          |          |
|            *            |          |          |
+-------------------------+----------+----------+
(Diamond, 23, 23) (Ladder, 8, 8) (Rectangle, 3, 8)
Oh No!
You lose your bet
You now have 6 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 remaining 6 tokens at the bar!
Game Over.
```

9.5 Slot Machine Algorithm

The `run()` method called on line 4 of the source code on page 14 implements the following algorithm:

Algorithm 1 run(t)

1: Allow an optional starting number of player's slot tokens t, using $t = 10$ if t is not supplied.
2: Let `current tokens` $\leftarrow t$
3: Prepare an array `reel` of three elements to represent the three reels of the slot machine. Each array element is a *smart* pointer to a `Shape` object (why are we using pointers?).
4: **while** player's `current tokens` > 0 and player wants to play **do**
5: Prompt for and read a bet ▷ an integer number of slot tokens
6: **for each** $k = 0, 1, 2$ **do** ▷ have `reel[k]` point to a random shape
7: Generate a random integer n, $0 \leq n \leq 3$
8: Generate a random width w, $1 \leq w \leq 25$
9: **if** $n = 0$ **then**
10: Let `reel[k]` point to a `Rhombus` object of width w
11: **else if** $n = 1$ **then**
12: Let `reel[k]` point to an `AccuteTriangle` object of width w
13: **else if** $n = 2$ **then**
14: Let `reel[k]` point to a `RightTriangle` object of width w
15: **else**
16: Generate a random height h, $1 \leq h \leq 25$
17: Let reel `reel[k]` point to a `Rectangle` object of width w and height h
18: **end if**
19: **end for**
20: Print the three reels side by side horizontally. ▷ simulate the payline
21: Based on the symbols on the payline, compute the tokens won (prize payout) as follows:

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

22: Update player's `current tokens`
23: Display whether the player has won, lost, or neither
24: **end while**
25: Display an appropriate message if `current tokens` is zero and another message otherwise.

9.5.1 Implementation of Step 3 of Slot Machine Algorithm

Using modern C++ smart pointers, implement step 3 as follows:

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

so that the three reels are represented by three `std::unique_ptr<Shape>` objects `reel[0]`, `reel[1]`, `reel[2]`, all three currently pointing to `nullptr`. The actual shape objects are created and assigned at steps 10, 12, 14 and 17. For example, step 10 is implemented as follows:

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

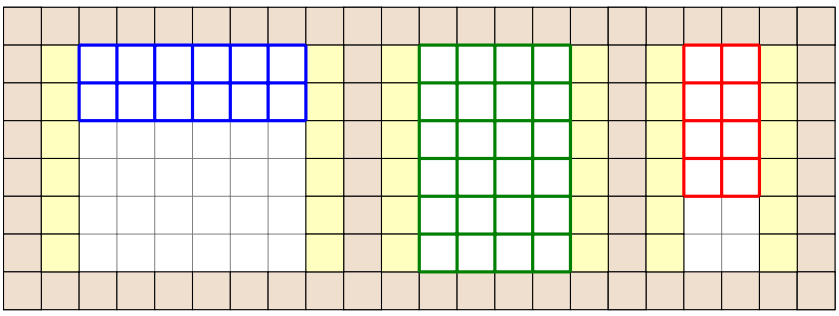
The unique pointer `reel[k]` automatically "`delete`s" the object it currently manages (points at), before resetting the pointer to a new dynamically allocated `Shape` object.

9.5.2 Implementation of Step 20 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 displays 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 19 for specifics.



9.6 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 6-19
   void make_shape(int k); // Steps 7-18
   void display();         // Step 23
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();
};
```

9.7 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
- Abstract the concept 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.

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).

10 Evaluation Criteria

Task 1: 70% Shape classes
Task 2: 30% Slot machine class

Each task is graded as follows:

Functionality	• 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 global variables • No use of operator operator delete. • No C-style 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 for non-trivial code segments	10%
Presentation	• Format, clarity, completeness of output, • User friendly interface	5%
Code readability	Meaningful identifiers, indentation, spacing	5%