Comp 5421 BB	Assignment 4	Due by 24 July 2016

Objectives

- To practice object-oriented programming (OOP) concepts such as inheritance, abstract classes, virtual functions, late-binding, and polymorphism
- To learn how to implement and use two-dimensional arrays using `vector<T>`, the simplest container class template in the Standard Template Library (STL)

Geometric Shape Modeling

This assignment makes use of simple geometric shape objects to practice and illustrate OOP concepts. The geometric shape objects considered are simple two-dimensional shapes that can reasonably be depicted textually; namely, squares, rectangles, and specific kinds of triangles and rhombuses.

Rather than creating different, unrelated classes for each of these shapes, we first explore whether there is some relationship between them. We quickly find that relationship from the previous paragraph: all geometric shapes named above have a name (an attribute of shapes), and formulas for computing their areas and perimeters (operations on shapes).

Shared Properties of Shape Objects

In this assignment, we require that our simple geometric shapes share the following general properties:

Attributes:
1. a fixed unique identification number
2. a fixed generic name
3. a variable descriptive name

Operations:
1. Access to all attributes
2. Modify the descriptive name
3. Generate a string representation of the shape
4. Scale by a given integer factor
5. Compute geometric area and geometric perimeter.
6. Compute *screen area* and *screen perimeter*.

   The *screen area* is the number of characters that form the textual image of a shape, and the *screen perimeter* is the number of characters on the borders of the textual image of a shape.

7. Compute the *width* and *height* of the shape's *bounding box*.
   The *bounding box* of a shape is the smallest rectangle that encloses the entire textual image of the shape.

8. Draw a textual image for the shape on a drawing surface.

Abstract Class Shape

The class of all objects with the properties listed above is named **Shape** in this assignment. In OOP, **Shape** is considered an *abstract* class, because **Shape**'s operations 3-8 are so general that it cannot possibly know how to implement them. An abstract class is designed to serve as a common interface to *concrete* classes that implement its unimplemented operations.

In C++, a class with a *pure virtual* function is called an *abstract* class. A pure virtual function of a class is a virtual member function in that class whose declaration ends with the curious = **0;** syntax.

In C++, you cannot declare an ordinary variable of a class that has at lease one pure virtual member function, but you can declare pointers and references of the class type:

```
class Shape   // an abstract class
{
   public: virtual void area() = 0; // a pure virtual function
   // ...
};
void f1(Shape*);        // ok
void f2(Shape&);        // ok
void f3(Shape);         // error; can't create objects of an abstract class
Shape shp1;             // error; can't create objects of an abstract class
```

Concrete Shapes

The geometric shape objects of interest in this assignment are rectangles, right isosceles triangles, and specific forms of isosceles triangles and rhombuses. Deriving from **Shape**, the concrete classes of these shapes each provide their own special attributes and special operations, including overrides.

Here are some of the specifics, where lengths are measured in character units:

Rectangle shapes		Sample Image
Construction values:	width w and height h	
Sample image:	a rectangle with $w = 9$ and $h = 5$	
How to scale(n)	set w to $w+n$ and h to $h+n$, provided that both $w+n \geq 1$ and $h+n \geq 1$; otherwise, no scale.	

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

Isosceles triangles with height h and base $b = 2h - 1$		Sample Image
Construction values:	height h	
Sample image:	An isosceles triangle with $h = 5$	
How to scale(n)	Set h to $h+n$ and b to $2h-1$, in that order, provided that $h+n \geq 1$; otherwise, no scale.	

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

Right isosceles triangles with height h and base $b = h$		Sample Image
Construction values:	height h	
Sample image:	a right isosceles triangle with $h = 5$.	
How to scale(n)	Set both h and b to $h+n$, provided that $h + n \geq 1$; otherwise, no scale.	

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

Rhombus shapes with both equal and odd diagonal lengths d		Sample Image
Construction values:	diagonal d	
Sample image:	a rhombus with $d = 5$	
How to scale(n)	set d to $d + n$, provided n is even and $d + n \geq 1$; otherwise, no scale.	

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

Thus, at construction, a **Rectangle** shape requires values of both its height and width, whereas the other three shapes each require a single value for the length of their respective vertical attribute.

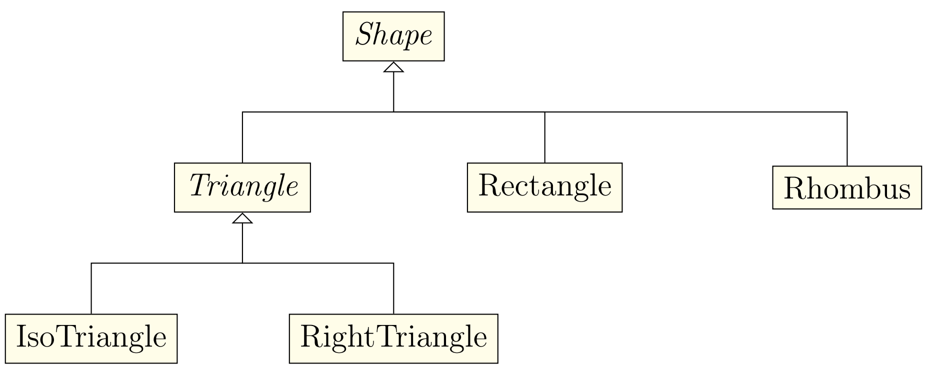
The remaining specifics of the shapes above are summarized in the following table.

Concrete Shape Specifics

	Rectangle	Rhombus	Right Triangle	Isosceles Triangle
Construction Values	h, w	d	h	h
Computed Values		if d is even set $d = d + 1$	$b = h$	$b = 2h - 1$
Bounding Box Height	h	d	h	h
Bounding Box Width	w	d	b	b
Geo 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
Geo Perimeter	$2(h+w)$	$(2\sqrt{2})d$	$(2+\sqrt{2})h$	$b + 2\sqrt{0.25b^2 + h^2}$
Screen Perimeter	$2(h+w) - 4$	$2(d-1)$	$3(h-1)$	$4(h-1)$

Task 1 of 3

Implement the following a class hierarchy:



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 code) and common attributes (shared data) are pushed toward the top of your class hierarchy.

Note: for this task, you should ignore operation 8 in **Shape**'s interface; you will implement that operation in your next task.

Here are a couple of examples alongside the output they each generate:

```cpp
Rectangle shape1(10, 15);
cout << shape1 << endl;
```

```
Shape Information
-----------------
Static type:   PK5Shape
Dynamic type:  9Rectangle
Generic name:  Rectangle
Description:   Generic Rectangle
id:            1
Bound. box W:  10
Bound. box H:  15
Scr area:      150
Geo area:      150.00
Scr perimeter: 46
Geo perimeter: 50.00
```

To get the name of the *static* type of a pointer **p** at runtime use **typeid(p).name()**, and to get the name of its *dynamic* type use **typeid(*p).name()**. You need to include the standard header **<typeinfo>** for this. The actual names returned by these calls are implementation defined. For example, the output above was generated under MinGW 4.9.2, where **PK** in **PK5Shape** means "pointer to ~~**konst**~~ **const** ", and the **5** in where **PK5** means that the type name that follows it is **5** character long. Microsoft VC++ produces more readable output as shown below.

```cpp
Rectangle shape1(10, 15);
cout << shape1 << endl;
```

```
Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
Description:   Generic Rectangle
id:            1
Bound. box W:  10
Bound. box H:  15
Scr area:      150
Geo area:      150.00
Scr perimeter: 46
Geo perimeter: 50.00
```

The ID number 1 for the shape is assigned during the construction of the object. 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 automatically when shape objects are constructed.

The generic name for a shape is the name of its class, and is automatically set when the shape object is constructed.

The descriptive name for a shape defaults to the word `Generic` followed by the class name, but can be supplied when the shape object is created:

```
3 Rhombus ace(16, "Ace of diamond");
4 cout << ace.toString() << endl;
```

```
Shape Information
-----------------
Static type:    PK5Shape
Dynamic type:   7Rhombus
Generic name:   Rhombus
Description:    Ace of diamond
id:             2
Bound. box W:   17
Bound. box H:   17
Scr area:       145
Geo area:       144.50
Scr perimeter:  32
Geo perimeter:  48.08
```

Note 1: Lines 2 and 4 of the code segments above show equivalent ways for printing shape information. The **toString()** function call at line 4 generates a string representation for the shape object **ace**.

Note 2: At line 3, the supplied height, 16, is invalid because it is even; to correct the invalid height, **Rhombus**'s constructor uses instead the next odd integer, 17, as the diagonal of object **ace**.

Here are two other examples of **Shape** objects.

```
5 Isosceles iso1(10);
6 Shape * iso1ptr = &iso1;
7 cout << iso1ptr->toString()
8      << endl;
```

```
Shape Information
-----------------
Static type:    PK5Shape
Dynamic type:   9Isosceles
Generic name:   Isosceles
Description:    Generic Isosceles
id:             3
Bound. box W:   19
Bound. box H:   10
Scr area:       100
Geo area:       95.00
Scr perimeter:  36
Geo perimeter:  46.59
```

```cpp
RightIsosceles iso2(10,
            "Carpenter's square");
cout << iso2.toString() << endl;
```

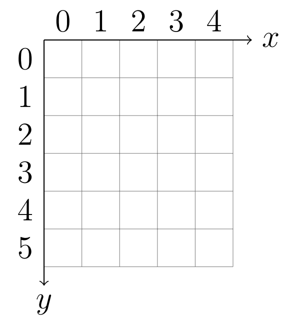
```
Shape Information
-----------------
Static type:   PK5Shape
Dynamic type:  14RightIsosceles
Generic name:  Right Isosceles
Description:   Carpenter's square
id:            4
Bound. box W:  10
Bound. box H:  10
Scr area:      55
Geo area:      50.00
Scr perimeter: 27
Geo perimeter: 34.14
```

Task 2 of 3: A Canvas to Draw on

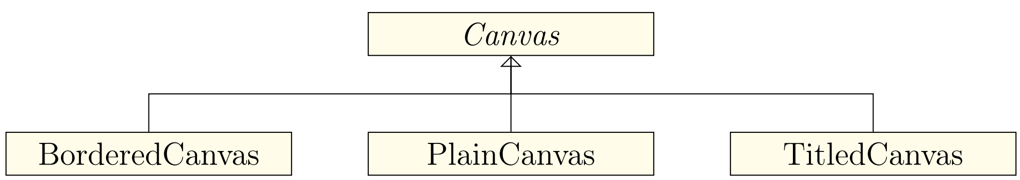
Having implemented your **Shape** class hierarchy, you are now ready to create drawing areas for the **Shape** objects to draw on. Specifically, you will model the concept of a drawing area, named **Canvas** here, described as follows:

A **Canvas** object stores a two-dimensional grid of character cells and provides a simple interface that allows its clients to manipulate the contents of the grid cells.

The grid has specified numbers of rows and columns. The grid rows are parallel to the x-axis, with row numbers increasing down. The grid columns are parallel to the y-axis, with column numbers increasing to the right. The origin of the grid is located at the top-left grid cell at $(x, y) = (col, row) = (0, 0)$. For clarity, we write an 'n-column by m-row' grid instead of an 'n by m' grid.



We are interested in implementing three concrete classes based on **Canvas**:



So, let's first specify the shared properties of our canvas objects.

Shared Properties of Canvas Objects

Attributes:
1. Number of grid rows, m
2. Number of grid columns, n
3. A two dimensional grid of characters with n columns and m rows.

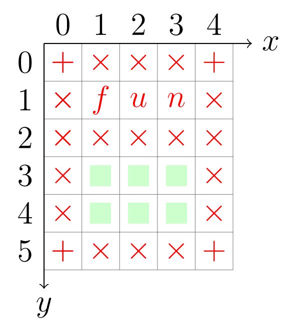
Operations:
1. Get canvas height (number of grid rows)
2. Get canvas width (number of grid columns).
3. Write canvas grid area to an **std::istream**.
4. Clear canvas grid area using a given character **ch**.
5. Put a given character **ch** in the grid cell at column **c** and row **r**.
6. Get the character **ch** from the grid cell at column **c** and row **r**.
7. Decorate the canvas, doing nothing by default.
8. Validate a given column **c** and row **r**.

Obviously, **Canvas** can implement operations 1-3 directly. However, even though it can implement operations 4-8, it is going to allow future subclasses to override them. That way **Canvas** promotes code reuse and polymorphism.

The concrete classes in the class inheritance hierarchy above differ from each other by the algorithms they use to frame and decorate the grid cells around their client area.

The Client Area of a Canvas

The client area of a **Canvas** object is a rectangular area on the grid where a **Shape** object can draw on. The dimensions of the client area may or may not be the same as the dimensions of of the underlying grid area, depending on whether the **Canvas** object uses any rows and/or columns to decorate the cells around its client area. The Figure at right depicts a 3-column by 2-row client area within a 5-column by 6-row grid, with its origin of the client area at $(3, 1)$.



Class PlainCanvas

A **PlainCanvas** object does not consume grid rows and/or columns for decorating its client area. In other words, The client area for a **PlainCanvas** object coincides with the entire grid area it stores, fixing the origin of its client area at the grid cell $(0, 0)$. The image at right depicts a 15-column by 5-row grid area for a **PlainCanvas(15,5)** whose client area has been cleared using the dot character. The row numbers are for reference only and not part of the output.

```
0   . . . . . . . . . . . . . . . . .
1   . . . . . . . . . . . . . . . . .
2   . . . . . . . . . . . . . . . . .
3   . . . . . . . . . . . . . . . . .
4   . . . . . . . . . . . . . . . . .
```

Class BorderedCanvas

A **BorderedCanvas** represents a **Canvas** whose client area is bordered by one row at the top, one row at the bottom, one column at the left, and one column at the right edge of the client area. To accommodate its borders, **BorderedCanvas** acquires a grid area that extends its client area by 2 rows and 2 columns, fixing the origin of its client area at the grid cell $(1, 1)$. The image at right depicts a 17-column by 7-row grid area for a **BorderedCanvas(15,5)** whose client area has been cleared using the dot character.

```
0  +-----------------+
1  |.................|
2  |.................|
3  |.................|
4  |.................|
5  |.................|
6  +-----------------+
```

Class TitledCanvas

A **TitledCanvas** represents a **Canvas** whose client area is decorated by three rows at the top, one row at the bottom, one column at the left, and one column at the right edge of the client area. The non-border cells on the second row houses a textual title. To accommodate its borders, **TitledCanvas** acquires a grid area that extends its client area by 4 rows and 2 columns, fixing the origin of its client area at the grid cell $(3, 1)$. The image at right depicts a 17-column by 9-row grid area for a **TitledCanvas(15,5)** whose client area has been cleared using the dot character.

```
0 +-----------------+
1 |A titled canvas..|
2 |-----------------|
3 |.................|
4 |.................|
5 |.................|
6 |.................|
7 |.................|
8 +-----------------+
```

Use **std::vector** to implement the grid for **Canvas**:

Two-Dimensional Gird (Array) in C++

```
// an empty 2-D array of ints with 0 rows and 0 cols
vector<vector<int> > array; // can later be resized to user specified dimensions

// a 2-D grid with 10 rows and 20 cols, filled with blank chars
const char BLANK = ' ';
vector< vector<char> > grid(10, vector<char>(20, BLANK));
...
// resize array to 10 rows and 20 cols, filled with zeros
const int ZERO = 0;
array.resize(10, vector<int>(20, ZERO));
// int x = array[i][j];      // read an array element
// array[i][j] = 99;      // write an array element
```

Drawing shapes on a canvas

Objects of **Shape** associate with objects of **Canvas**, but not vice versa.

The prototype for the **Shape**'s draw operation (8) is:

```
virtual void draw(int c, int r, Canvas & canvas, char ch = '*') const = 0;
```

This polymorphic member function draws the invoking **Shape** object at column **c** and row **r** relative to the origin of the client area of the **canvas** object. Specifically, the **canvas** object in use maps the point (c, r) to gird cell at position $(c, r) + (x, y)$, where (x, y) denotes the origin of **canvas**'s client area.

Examples

```
1  // a plain canvas with 20 columns (width) by 10 row (height)
2  PlainCanvas pCanvas(20, 10);
3  pCanvas.clear('-'); // clear canvas with the '-' character
4  cout << pCanvas << endl;           // print canvas
```

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

```
5  //bordered canvas with 20 columns (width) by 10 row (height)
6  BorderedCanvas bCanvas(20, 10);
7  cout << bCanvas << endl;
```

```
+--------------------+
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
|                    |
+--------------------+
```

Note: by default, the constructor calls at lines 2 and 6 clear the client area using the blank character.

```
8  //titled canvas with 20 columns (width) by 10 row (height)
9  TitledCanvas tCanvas(20, 10, "A titled canvas");
10 tCanvas.clear('^');
11 cout << tCanvas << endl;
```

```
+--------------------+
|A titled canvas     |
|--------------------|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
+--------------------+
```

```
12 // draw a right triangle with height 6
13 // at col 8  row 2 on canvas
14 RightIsosceles shp1 = RightIsosceles(6);
15 shp1.draw( 8, 2, pCanvas);
16 cout << pCanvas << endl;
```

```
--------------------
--------------------
--------*-----------
--------**----------
--------***---------
--------****--------
--------*****-------
--------******------
--------------------
--------------------
```

```
17 // draw shp1 above at col 9 row 4 on bCanvas
18 shp1.draw(9, 4, bCanvas);
19 cout << bCanvas << endl;
```

```
+--------------------+
|                    |
|                    |
|                    |
|                    |
|         *          |
|         **         |
|         ***        |
|         ****       |
|         *****      |
|         ******     |
+--------------------+
```

```
20 // draw shp1 above at col 3 row 0 on tCanvas
21 shp1.draw(3, 0, tCanvas,'O');
22 cout << tCanvas << endl;
```

```
+--------------------+
|A titled canvas     |
|--------------------|
|^^^O^^^^^^^^^^^^^^^^|
|^^^OO^^^^^^^^^^^^^^^|
|^^^OOO^^^^^^^^^^^^^^|
|^^^OOOO^^^^^^^^^^^^^|
|^^^OOOOO^^^^^^^^^^^^|
|^^^OOOOOO^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
+--------------------+
```

Any shape object can draw on any canvas object. The following code redraws **shp1** on the same canvas **tCanvas** but at a different location:

```
23 // redraw shp1 above at col 16 row 6 on tCanvas
24 shp1.draw(16, 6, tCanvas);
25 cout << tCanvas << endl;
```

```
+--------------------+
|A titled canvas     |
|--------------------|
|^^^O^^^^^^^^^^^^^^^^|
|^^^OO^^^^^^^^^^^^^^^|
|^^^OOO^^^^^^^^^^^^^^|
|^^^OOOO^^^^^^^^^^^^^|
|^^^OOOOO^^^^^^^^^^^^|
|^^^OOOOOO^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^*^^^|
|^^^^^^^^^^^^^^^^**^^|
|^^^^^^^^^^^^^^^^***^|
|^^^^^^^^^^^^^^^^****|
+--------------------+
```

Although the **shp1** object draws its entire shape on the **tCanvas** object , it is **tCanvas**'s responsibility to ignore (*clip*) writes that land outside its client area; that is why the bottom two rows of **shp1** (drawn at line 24) are missing at the bottom right corner of **tCanvas**.

Here is an example of scaling two shapes on the same *canvas*:

```
// show scaling of a  Rhombus and  an Isosceles objects
tCanvas.clear();   // clear the canvas
Rhombus rhom(9);   // a Rhombus with diagonal length = 9
rhom.draw(1, 1, tCanvas); // draw the rhombus
Isosceles iso(1); // an Isosceles with height 1
// draw iso on tCanvas at column 10 row 4 using 'O'
iso.draw( 10, 4,tCanvas, 'O');
cout << tCanvas << endl;
for(int k = 0; k < 3; ++k )
{
    tCanvas.clear();                    // clear the canvas
    rhom.scale(-1);                 // scale down the rhombus
    rhom.draw(1, 1, tCanvas);          // draw the rhombus
    iso.scale(+1);                   // scale up the isosceles
    iso.draw(10, 4, tCanvas, 'O');// draw the isosceles
    cout << tCanvas << endl;// write tCanvas to screen
}
```

```
+--------------------+
|A titled canvas     |
|--------------------|
|                    |
|     *              |
|    ***             |
|   *****            |
|  ******* O         |
| *********          |
|  *******           |
|   *****            |
|    ***             |
|     *              |
+--------------------+
```

```
+--------------------+
|A titled canvas     |
|--------------------|
|                    |
|    *               |
|   ***              |
|  *****             |
| *******    O       |
|  *****    OOO      |
|   ***              |
|    *               |
|                    |
|                    |
+--------------------+
```

```
+--------------------+
|A titled canvas     |
|--------------------|
|                    |
|   *                |
|  ***               |
| *****              |
|  ***        O      |
|   *        OOO     |
|           OOOOO    |
|                    |
|                    |
|                    |
+--------------------+
```

```
+--------------------+
|A titled canvas     |
|--------------------|
|                    |
|  *                 |
| ***                |
|  *                 |
|              O     |
|             OOO    |
|            OOOOO   |
|           OOOOOOO  |
|                    |
|                    |
+--------------------+
```

The initial diagonal length of the rhombus shape is 9, and initial height of the isosceles shape is 1. The rhombus is scaled down by 1 unit, and the isosceles is scaled up by 1 unit, in that order, three times. Since the diagonal of rhombus shapes must be odd, it make no sense to scale it by 1 unit (up or down); so, **Rhombus**'s **scale** function chooses the next best length. The isosceles, however, scales by any number of units. The scaling process never scales down the length of an attribute to below 1.

The following program shows how to construct a compound shape that looks like the front view of a house.

Task 3 of 3: Putting It All Together

Drawing a House

```
// Uses different shapes to draw a textual image that looks line a house
void drawHouse()
{
    // draw a house front view on a 44-column and 50-row titles canvas
    TitledCanvas canvas(44, 50, "A Geometric House: Front View");

    Rectangle chimney(2, 14, "Chimeny on the roof"); // A vertical 14 x 2 chimney
    chimney.draw(4, 7, canvas, 'H'); // Draw chimney on canvas

    Isosceles roof(21, "House roof"); // A triangular roof of height 21
    roof.draw(1, 1, canvas, '/');// Draw roof

    Rectangle skylightFrame(9, 5, "Frame around skylight"); // A 9c x 5r skylight frame
    skylightFrame.draw(17, 11, canvas, 'H');// Draw skylight frame

    Rectangle skylight(7, 3, "skylight"); // A 7c x 3r skylight
    skylight.draw(18, 12, canvas, ' ');// Draw skylight

    Rectangle front(41, 22, "Front wall");// A 41c x 22r rectangular front wall
    front.draw(1, 23, canvas, ':'); // draw front wall

    Rectangle top_left_brackets(21, 1, "top Left Brackets "); // top left square bracke
    top_left_brackets.draw(1, 22, canvas, '['); // top left square brackets

    Rectangle bottom_left_brackets(21, 1, "Bottom Left Brackets "); // bottom left sq
    bottom_left_brackets.draw(1, 44, canvas, '['); // bottom left square brackets

    Rectangle top_right_brackets(20, 1, "top Right Brackets "); // top brackets right s
    top_right_brackets.draw(22, 22, canvas, ']'); // Draw top brackets right square bracket

    Rectangle bottom_right_brackets(20, 1, "Bottom Right Brackets "); // a bottom rig
    bottom_right_brackets.draw(22, 44, canvas, ']');// Draw bottom right square brackets

    Rectangle right_right_brackets(2, 22, "Right Right Brackets "); // right right sq
    right_right_brackets.draw(40, 23, canvas, ']');// Draw right right square brackets

    Rectangle lef_tleft_brackets(2, 22, "Left Left Brackets "); // left left square bra
    lef_tleft_brackets.draw(1, 23, canvas, '[');// Draw left left square brackets

    Rectangle leftDoors(7, 10, "Front Left Door"); // A rectangle left door
    leftDoors.draw(22, 33, canvas, 'd');// Draw left door

    Rectangle rightDoors(7, 10, "Front Right Door"); // A rectangle right door
```

```
    rightDoors.draw(30, 33, canvas, 'b');// Draw right door

    // visually split the two doors
    Rectangle doorsMiddle(1, 10, "Vertical center panel between front doors");
    doorsMiddle.draw(29, 33, canvas, '='); // draw the middle vertical rectangle

    // Triagle windows above front door
    Isosceles Triagle_above_front_door(8, "Triagle Door Top"); // triagle above front c
    Triagle_above_front_door.draw(22, 24, canvas, '*');

    Rectangle doggyDoor = Rectangle(4, 2, "A rectangle  doggy door");// A rectangle  doggy door
    doggyDoor.draw(13, 41, canvas, 'D');// Draw doggy door

    Rhombus diamond_shape_window_on_front_wall(7, "Diamond shape window on front
    diamond_shape_window_on_front_wall.draw(4, 25, canvas, 'O');// Draw rhombus window

    Rectangle StairSlash(40, 1, "Stair Slash"); // A row of 40 slashes
    StairSlash.draw(1, 45, canvas, '/'); // Draw row of 40 slashes
    StairSlash.draw(1, 46, canvas, '/'); // Draw row of 40 slashes
    StairSlash.draw(1, 47, canvas, '/'); // Draw row of 40 slashes
    StairSlash.draw(1, 48, canvas, '/'); // Draw row of 40 slashes

    RightAngle rightAngle(4, "RightAngle "); // right angle at bottom left of hous
    rightAngle.draw(3, 40, canvas, '\\');

    cout << canvas << endl;
    cout << chimney << endl;
    cout << roof << endl;
    cout << skylightFrame << endl;
    cout << skylight << endl;
    cout << front << endl;
    cout << top_left_brackets << endl;
    cout << bottom_left_brackets << endl;
    cout << top_right_brackets << endl;
    cout << bottom_right_brackets << endl;
    cout << right_right_brackets << endl;
    cout << lef_tleft_brackets << endl;
    cout << leftDoors << endl;
    cout << rightDoors << endl;
    cout << doorsMiddle << endl;
    cout << Triagle_above_front_door << endl;
    cout << doggyDoor << endl;
    cout << diamond_shape_window_on_front_wall << endl;
    cout << StairSlash << endl;
    cout << rightAngle << endl;
}
```

```
+--------------------------------------------+
|A Geometric House: Front View               |
|--------------------------------------------|
|                                            |
|                     /                      |
|                    ///                     |
|                   /////                    |
|                  ///////                   |
|                 /////////                  |
|                ///////////                 |
|    HH         /////////////                |
|    HH        ///////////////               |
|    HH       /////////////////              |
|    HH      ///////////////////             |
|    HH     //////HHHHHHHHH//////            |
|    HH    ///////H       H///////           |
|    HH   ////////H       H////////          |
|    HH  /////////H       H/////////         |
|    HH //////////HHHHHHHHH//////////        |
|    HH///////////////////////////////       |
|    H/////////////////////////////////      |
|    ///////////////////////////////////     |
|   /////////////////////////////////////    |
|  ///////////////////////////////////////   |
| /////////////////////////////////////////  |
| [[[[[[[[[[[[[[[[[[[[[]]]]]]]]]]]]]]]]]]]]  |
| [[:::::::::::::::::::::::::::::::::::::]]  |
| [[::::::::::::::::::::::::::*::::::::::]]  |
| [[::::0::::::::::::::::::::***:::::::::]]  |
| [[:::000::::::::::::::::::*****::::::::]]  |
| [[::00000::::::::::::::::*******:::::::]]  |
| [[:0000000::::::::::::::*********::::::]]  |
| [[::00000::::::::::::::***********:::::]]  |
| [[:::000::::::::::::::*************::::]]  |
| [[::::0::::::::::::::***************:::]]  |
| [[:::::::::::::::::::::::::::::::::::::]]  |
| [[:::::::::::::::::::ddddddd=bbbbbbb:::]]  |
| [[:::::::::::::::::::ddddddd=bbbbbbb:::]]  |
| [[:::::::::::::::::::ddddddd=bbbbbbb:::]]  |
| [[:::::::::::::::::::ddddddd=bbbbbbb:::]]  |
| [[:::::::::::::::::::ddddddd=bbbbbbb:::]]  |
| [[:::::::::::::::::::ddddddd=bbbbbbb:::]]  |
| [[:::::::::::::::::::ddddddd=bbbbbbb:::]]  |
| [[\::::::::::::::::::ddddddd=bbbbbbb:::]]  |
| [[\\::::::::DDDD:::::ddddddd=bbbbbbb:::]]  |
| [[\\\:::::::DDDD:::::ddddddd=bbbbbbb:::]]  |
| [[\\\\:::::::::::::::::::::::::::::::::]]  |
| [[[[[[[[[[[[[[[[[[[[[]]]]]]]]]]]]]]]]]]]]  |
| ////////////////////////////////////////   |
| ////////////////////////////////////////   |
| ////////////////////////////////////////   |
| ////////////////////////////////////////   |
|                                            |
+--------------------------------------------+
```

```
Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
Description:   Chimeny on the roof
id:            1
Bound. box W:  2
Bound. box H:  14
Scr area:      28
Geo area:      28.00
Scr perimeter: 28
Geo perimeter: 32.00



Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Isosceles
Generic name:  Isosceles
Description:   House roof
id:            2
Bound. box W:  41
Bound. box H:  21
Scr area:      441
Geo area:      430.50
Scr perimeter: 80
Geo perimeter: 99.69



Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
Description:   Frame around skylight
id:            3
Bound. box W:  9
Bound. box H:  5
Scr area:      45
Geo area:      45.00
Scr perimeter: 24
Geo perimeter: 28.00



Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
```

```
Description:   skylight
id:            4
Bound. box W:  7
Bound. box H:  3
Scr area:      21
Geo area:      21.00
Scr perimeter: 16
Geo perimeter: 20.00



Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
Description:   Front wall
id:            5
Bound. box W:  41
Bound. box H:  22
Scr area:      902
Geo area:      902.00
Scr perimeter: 122
Geo perimeter: 126.00



Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
Description:   top Left Brackets
id:            6
Bound. box W:  21
Bound. box H:  1
Scr area:      21
Geo area:      21.00
Scr perimeter: 40
Geo perimeter: 44.00



Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
Description:   Bottom Left Brackets
id:            7
Bound. box W:  21
Bound. box H:  1
Scr area:      21
Geo area:      21.00
```

```
Scr perimeter: 40
Geo perimeter: 44.00


Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
Description:   top Right Brackets
id:            8
Bound. box W:  20
Bound. box H:  1
Scr area:      20
Geo area:      20.00
Scr perimeter: 38
Geo perimeter: 42.00


Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
Description:   Bottom Right Brackets
id:            9
Bound. box W:  20
Bound. box H:  1
Scr area:      20
Geo area:      20.00
Scr perimeter: 38
Geo perimeter: 42.00


Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
Description:   Right Right Brackets
id:            10
Bound. box W:  2
Bound. box H:  22
Scr area:      44
Geo area:      44.00
Scr perimeter: 44
Geo perimeter: 48.00

Press any key to continue . . .
```

```
Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
Description:   Left Left Brackets
id:            11
Bound. box W:  2
Bound. box H:  22
Scr area:      44
Geo area:      44.00
Scr perimeter: 44
Geo perimeter: 48.00


Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
Description:   Front Left Door
id:            12
Bound. box W:  7
Bound. box H:  10
Scr area:      70
Geo area:      70.00
Scr perimeter: 30
Geo perimeter: 34.00


Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
Description:   Front Right Door
id:            13
Bound. box W:  7
Bound. box H:  10
Scr area:      70
Geo area:      70.00
Scr perimeter: 30
Geo perimeter: 34.00


Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
Description:   Vertical center panel between front doors
```

```
id:             14
Bound. box W:   1
Bound. box H:   10
Scr area:       10
Geo area:       10.00
Scr perimeter:  18
Geo perimeter:  22.00



Shape Information
-----------------
Static type:    class Shape const *
Dynamic type:   class Isosceles
Generic name:   Isosceles
Description:    Triagle Door Top
id:             15
Bound. box W:   15
Bound. box H:   8
Scr area:       64
Geo area:       60.00
Scr perimeter:  28
Geo perimeter:  36.93



Shape Information
-----------------
Static type:    class Shape const *
Dynamic type:   class Rectangle
Generic name:   Rectangle
Description:    A rectangle  doggy door
id:             16
Bound. box W:   4
Bound. box H:   2
Scr area:       8
Geo area:       8.00
Scr perimeter:  8
Geo perimeter:  12.00



Shape Information
-----------------
Static type:    class Shape const *
Dynamic type:   class Rhombus
Generic name:   Rhombus
Description:    Diamond shape window on front wall
id:             17
Bound. box W:   7
Bound. box H:   7
Scr area:       25
Geo area:       24.50
Scr perimeter:  12
```

```
Geo perimeter: 19.80


Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Generic name:  Rectangle
Description:   Stair Slash
id:            18
Bound. box W:  40
Bound. box H:  1
Scr area:      40
Geo area:      40.00
Scr perimeter: 78
Geo perimeter: 82.00


Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class RightAngle
Generic name:  Right Isosceles
Description:   RightAngle
id:            19
Bound. box W:  4
Bound. box H:  4
Scr area:      10
Geo area:      8.00
Scr perimeter: 9
Geo perimeter: 13.66
```

Evaluation

Program correctness: **Shape** class hierarchy	30%
Program correctness: : **Canvas** class hierarchy	30%
Practice of OOP principles; proper use of C++ constructs; Minimal use of C	20%
Format, clarity, and completeness of output	10%
Documentation of purpose of functions and classes including pre- and post- conditions before function headings, Comments on nontrivial steps inside function body, Choice of meaningful variable names, Indentation and readability of program. No *magic* numbers!	10%