Objectives

- To practice using advanced object-oriented programming concepts such as inheritance, abstract classes, virtual functions, late-binding, and polymorphism.
- To learn how to implement such object-oriented techniques in C++ by developing two class hierarchies for representing and displaying two-dimensional geometric shapes.
- To learn about using the C++ standard library class template **vector** to create and manipulate two-dimensional arrays.[1]

Geometric Shape Modeling

Geometric shapes provide a simple model for practicing object-oriented programming concepts.

The shape objects of interest in this assignment model two-dimensional geometric shapes characterized by their common attributes and common operations:

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

Operations:
1. Accessors for the attributes.
2. Mutator for the descriptive name.
3. Generate a string representation.
4. Scale shape by a given integer factor.
5. Compute geometric area and geometric perimeter.
6. Compute *screen area*[2] and *screen perimeter*.[3]
7. Compute shape's horizontal and vertical extents.[4]
8. Draw a textual image for the shape on a drawing surface.[5]

The class of all such objects will be called **Shape** in this assignment.

Evidently, **Shape**'s characterization of geometric shapes is so general that it lacks the specifics needed for implementing most of its operations, implying that **Shape** represents an abstract class. However, **Shape** can be used as an interface or as a base for classes that specialize **Shape**. Any specialization of **Shape** must either implement **Shape**'s unimplemented operations, or itself be abstract.

[1] No built-in arrays, please. Use **vector<vector<T> >** and resize to desired dimensions, for example.

[2] The number characters consumed by the textual image of a shape

[3] The number of characters used on the borders of the textual image of a shape.

[4] Associated with each shape is a *conceptual* bounding box that encloses the textual image of the shape; the width and height of this bounding box represent the horizontal and vertical extents of the shape, respectively.

[5] See page 5.

Specialized Shapes

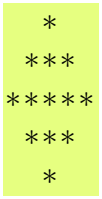
The specialized shapes of interest consist of four basic geometric shapes: rectangles, and specific forms of rhombuses, isosceles triangles, and right isosceles triangles. Deriving from **Shape**, the concrete classes of these shapes each provide their own special attributes and special operations, including overrides. Here are some of the specifics:

- Rectangles

  Attributes: Width w and height h, measured in character units.
  Operations: Accessors/mutators for width and height.
  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.
  Sample visual image shown: A rectangle with $w = 9$ and $h = 5$.

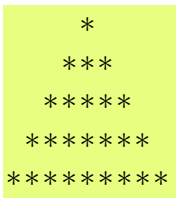
- Rhombus shapes that have *diagonal*s of the same length d, such that d is an odd integer.

  Attributes: Diagonal d, measured in character units.
  Operations: Accessors/mutators for diagonal.
  How to scale(n): Set d to $d+n$, provided $d+n \geq 1$. otherwise, no scale.
  Sample visual image shown: A rhombus with $d = 5$.



- Isosceles triangles that have the length of their base b related to their height h such that $b = 2 \times h - 1$.

  Attributes: Height, measured in character units.
  Operations: Accessors/mutators for height.
  How to scale(n): Set h to $h+n$ and b to $2h-1$, provided that $h+n \geq 1$. otherwise, no scale.
  Sample visual image shown: An isosceles triangle with $h = 5$.



- Right Isosceles triangles with equal base b and height h.

  Attributes: Height, measured in character units.
  Operations: Accessors/mutators for height.
  How to scale(n): Set both h and b to $h + n$, provided that $h + n \geq 1$. otherwise, no scale.
  Sample visual image shown: A right isosceles triangle with $h = 5$.

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

Shape Type	Required Values	Computed Values	Vert. Extent	Horiz. Extent	Geometric Area	Screen Area	Geometric Perimeter	Screen Perimeter
Rectangle	h, w		h	w	hw	hw	$2(h+w)$	$2(h+w)-4$
Rhombus	d	$d = d+1$ if d even	d	d	$d^2/2$	$2n(n+1)+1$, $n = \lfloor d/2 \rfloor$	$(2\sqrt{2})d$	$2(d-1)$
Right Triangle	h	$b = h$	h	h	$h^2/2$	$h(h+1)/2$	$(2+\sqrt{2})h$	$3(h-1)$
Isosceles Triangle	h	$b = 2h-1$	h	b	$hb/2$	h^2	$b + 2\sqrt{0.25b^2 + h^2}$	$4(h-1)$

Thus, a rectangle shape requires input values for both its height and width, whereas the other three shapes each require a single input for the length of their respective vertical attribute.

Your Task 1 of 3

Your first task is to design and implement a class hierarchy of the shapes above, *without* **Shape**'s *draw* operation.[6] 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.

Here are a couple of examples:

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

```
Shape Information
-----------------
Type of this:  class Shape const *
Type of *this: class Rectangle
Generic name:  Rectangle
Description:   Generic Rectangle
id:            1
H extent:      10
V extent:      15
Scr area:      150
Geo area:      150.00
Scr perimeter: 46
Geo perimeter: 50.00
```

The type names for **this** and ***this** are obtained through the calls **typeid(this).name()** and

[6] You will later implement a dawning surface on which **Shape** objects can render their textual images (see page 5).

typeid(*this).name(), respectively; you need to include the standard header **<typeinfo>** for this.

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
-----------------
Type of this:  class Shape const *
Type of *this: class Rhombus
Generic name:  Rhombus
Description:   Ace of diamond
id:            2
H extent:      17
V extent:      17
Scr area:      145
Geo area:      144.00
Scr perimeter: 32
Geo perimeter: 48.08
```

Note that lines 2 and 4 of the code segments above show equivalent ways for printing shape information. The **toString()** function call on line 4 generates a string representation for the shape object **ace**. Note also that in 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 more examples of **Shape** objects.

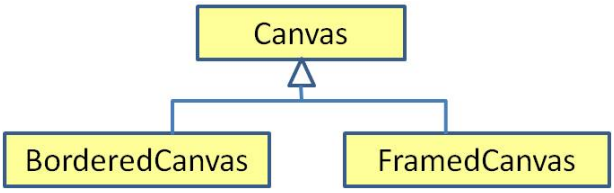
```
5 Isosceles iso1(10);
6 Shape * iso1ptr = &iso1;
7 cout << iso1ptr->toString() << endl;
```

```
Shape Information
-----------------
Type of this:  class Shape const *
Type of *this: class Isosceles
Generic name:  Isosceles
Description:   Generic Isosceles Triangle
id:            3
H extent:      19
V extent:      10
Scr area:      100
Geo area:      95.00
Scr perimeter: 36
Geo perimeter: 46.59
```

```
8 RightIsosceles iso2(10);
9 cout << iso2.toString() << endl;
```

```
Shape Information
-----------------
Type of this:  class Shape const *
Type of *this: class RightIsosceles
Generic name:  Right Isosceles
Description:   Generic Right Isosceles Triangle
id:            4
H extent:      10
V extent:      10
Scr area:      55
Geo area:      50.00
Scr perimeter: 27
Geo perimeter: 34.14
```

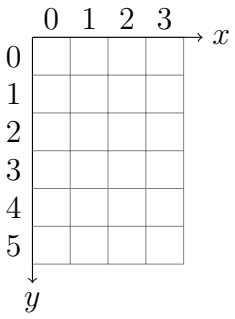
Your Task 2 of 3: More Polymorphism



Having completed your **Shape** class hierarchy, you are now ready to create drawing areas for the **Shape** objects to draw on. Specifically, your next task is to model the concept of drawing canvas with tool palette, implementing an inheritance hierarchy of three classes: **Canvas**, **BorderedCanvas**, and **FramedCanvas**.

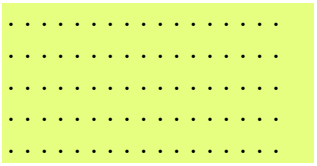
Objects of **Shape** associate with objects of **Canvas**, but not vice versa.

Objects of **Canvas** each store a rectangular grid of characters with rows and columns, and provide a simple interface that allows their clients to manipulate (*draw*) the contents of the grid cells. The origin of the grid is located at the top-left gid cell at row 0 and column 0. 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. Thus, $(x, y) = (col, row)$.

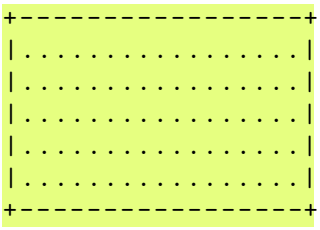


The area of the grid a client of **Canvas** can draw on is referred to as the *client area*. The size of the client area is supplied by the client in terms of its width (number of columns) and the height (number of rows).

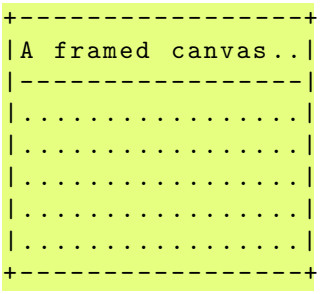
The client area for a plain **Canvas** object coincides with the entire grid it maintains.



BorderedCanvas represents **Canvas** objects with borders around their client area.



FramedCanvas represents **Canvas** objects with a title bar, a title, and borders.



Thus, the actual size of the grid maintained by a **Canvas** object may actually be larger than the size of its client area, depending on whether the canvas object itself consumes some number of rows and columns for decorating its client area.

The tool palette part of the concept being modeled is very simple: it consists of a pen and a set of characters. Positioned over a grid cell, the pen tool can read from or write to that grid cell. Because of its simplicity, the pen tool concept is not required to be implemented as a full blown class here; instead, it is implemented directly by `Canvas`.

The interface of canvas classes should provide the following services:

- A constructor — Receives as parameters the width and height of the desired client area, and possibly other specific information. Sets the dimensions of the associated grid so as to house both the client area and possibly borders, frames, etc.
- virtual void clear(char ch = BLANK) — Fills the client area with **ch**.
- int geth() const — Returns the height (number of rows) of this canvas.
- int getw() cons — Returns the width (number of columns) of this canvas.
- virtual void put(int c, int r, char ch = STAR) — Puts **ch** at column **c** and row **r**.
- virtual char get(int c, int r) const — Returns the character at column **c** and row **r**.
- virtual void decorate() — Decorate the client area with borders, frame, etc. according to canvas type.

If you like to practice two dimensional operations, implement the following! This part is entirely optional.

- virtual void flipHorizontal() Flips the contents of client area horizontally.
- virtual void flipVertical() Flips the contents of client area vertically.
- virtual void rotateLeft() Rotates the contents of client area 90 degrees counterclockwise.
- virtual void rotateRight() Rotates the contents of client area 90 degrees clockwise.

Drawing shapes on a canvas

The prototype for the **Shape**'s `draw` function is:

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

This pure virtual member function draws the invoking **Shape** object, mapping the top-left corner of the **Shape** object's conceptual bounding box to column **c** and row **r** in the client area of the supplied **canvas**.

Examples

```
1 Canvas canvas(20, 10); // a plain canvas of width 20 and height 10
2 canvas.clear('-'); // clear canvas with the '-' character
3 cout << canvas << endl; // print canvas
```

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

The constructor calls in line 1 and 5 clear the client area using the blank character.

```
4 // a bordered canvas with client area of width 20 and height 10
5 BorderedCanvas bCanvas(20, 10);
6 cout << bCanvas << endl;
```

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

```
+--------------------+
|A framed canvas     |
|--------------------|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
|^^^^^^^^^^^^^^^^^^^^|
+--------------------+
```

```
7  // a framed canvas with client area of width 20 and height 10
8  FramedCanvas fCanvas(20, 10, "A framed canvas");
9  fCanvas.clear('^');
10 cout << fCanvas << endl;
```

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

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

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

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

```
18 // draw shp1 above at col 3 row 0 on fCanvas
19 shp1.draw(3, 0, fCanvas,'O');
20 cout << fCanvas << endl;
```

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

```
21 // redraw shp1 above at col 16 row 6 on fCanvas
22 shp1.draw(16, 6, fCanvas);
23 cout << fCanvas << endl;
```

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

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

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

```
24 // show scaling of a Rhombus and an Isosceles objects
25 fCanvas.clear(); // clear the canvas
26 Rhombus rhom(9); // a Rhombus with diagonal length = 9
27 rhom.draw(1, 1, fCanvas); // draw the rhombus
28 Isosceles iso(1); // an Isosceles with height 1
29 // draw iso on fCanvas at column 10 row 4 using 'O'
30 iso.draw( 10, 4,fCanvas, 'O');
31 cout << fCanvas << endl;
32 for(int k = 0; k < 3; ++k )
33 {
34    fCanvas.clear(); // clear the canvas
35    rhom.scale(−1); // scale down the rhombus
36    rhom.draw(1, 1, fCanvas); // draw the rhombus
37    iso.scale(+1); // scale up the isosceles
38    iso.draw(10, 4, fCanvas, 'O'); // draw the isosceles
39    cout << fCanvas << endl;  // write contents fCanvas to screen
40 }
```

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

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 scales 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.

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

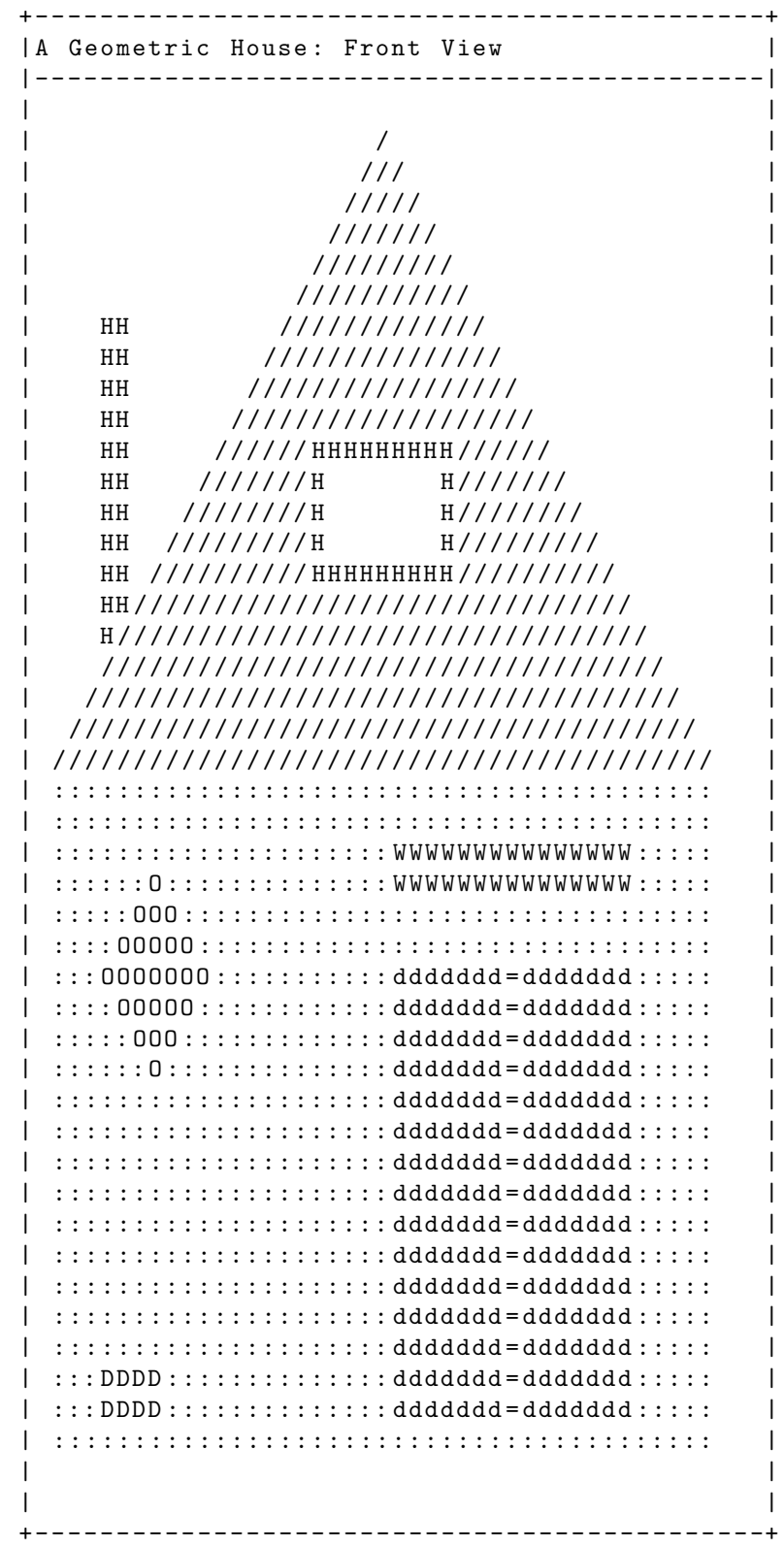
The following program shows how to construct a compound shape that looks like a house.

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

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

```
void drawHouse()
{
    // draw a house front view
    FramedCanvas canvas(45, 50, "A Geometric House: Front View"); // a canvas to draw on
    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, 22, canvas, ':'); // draw front wall

    Rectangle leftDoors(7, 15, "Front Left Door"); // A 7c x 15r rectangle door
    leftDoors.draw(22, 28, canvas, 'd'); // Draw left door

    Rectangle rightDoors(7, 15, "Front Right Door"); // A 7c x 15r rectangle door
    rightDoors.draw(30, 28, canvas, 'd'); // Draw right door

    // visually split the two doors
    Rectangle doorsMiddle(1, 15, "Vertical center panel between front doors");
    doorsMiddle.draw(29, 28, canvas, '='); // draw the middle vertical rectangle

    // windows above front door
    Rectangle doorTop = Rectangle(15, 2, "Top door window"); // A 4c by 2r rectangle
    doorTop.draw(22, 24, canvas, 'W'); // Draw top door window


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

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

    cout << canvas << endl; // polymorphic call

        // All polymorphic calls
    cout << chimney << endl;
    cout << roof << endl;
```

```
47      cout << skylightFrame << endl;
48      cout << skylight << endl;
49      cout << front << endl;
50      cout << leftDoors << endl;
51      cout << rightDoors << endl;
52      cout << doorsMiddle << endl;
53      cout << doggyDoor << endl;
54      cout << window << endl;
55
56  }
```


A Geometric House: Front View

```
Shape Information
-----------------
Type of this:  class Shape const *
Type of *this: class Rectangle
Generic name:  Rectangle
Description:   Chimeny on the roof
id:            4
H extent:      2
V extent:      14
Scr area:      28
Geo area:      28.00
Scr perimeter: 28
Geo perimeter: 32.00


Shape Information
-----------------
Type of this:  class Shape const *
Type of *this: class Isosceles
Generic name:  Isosceles
Description:   House roof
id:            5
H extent:      41
V extent:      21
Scr area:      441
Geo area:      430.50
Scr perimeter: 80
Geo perimeter: 99.69


Shape Information
-----------------
Type of this:  class Shape const *
Type of *this: class Rectangle
Generic name:  Rectangle
Description:   Frame around skylight
id:            6
H extent:      9
V extent:      5
Scr area:      45
Geo area:      45.00
Scr perimeter: 24
Geo perimeter: 28.00


Shape Information
-----------------
Type of this:  class Shape const *
Type of *this: class Rectangle
Generic name:  Rectangle
Description:   skylight
id:            7
H extent:      7
```

```
V extent:      3
Scr area:      21
Geo area:      21.00
Scr perimeter: 16
Geo perimeter: 20.00


Shape Information
-----------------
Type of this:  class Shape const *
Type of *this: class Rectangle
Generic name:  Rectangle
Description:   Front wall
id:            8
H extent:      41
V extent:      22
Scr area:      902
Geo area:      902.00
Scr perimeter: 122
Geo perimeter: 126.00


Shape Information
-----------------
Type of this:  class Shape const *
Type of *this: class Rectangle
Generic name:  Rectangle
Description:   Front Left Door
id:            9
H extent:      7
V extent:      15
Scr area:      105
Geo area:      105.00
Scr perimeter: 40
Geo perimeter: 44.00


Shape Information
-----------------
Type of this:  class Shape const *
Type of *this: class Rectangle
Generic name:  Rectangle
Description:   Front Right Door
id:            10
H extent:      7
V extent:      15
Scr area:      105
Geo area:      105.00
Scr perimeter: 40
Geo perimeter: 44.00


```

```
Shape Information
-----------------
Type of this:  class Shape const *
Type of *this: class Rectangle
Generic name:  Rectangle
Description:   Vertical center panel between front doors
id:            11
H extent:      1
V extent:      15
Scr area:      15
Geo area:      15.00
Scr perimeter: 28
Geo perimeter: 32.00


Shape Information
-----------------
Type of this:  class Shape const *
Type of *this: class Rectangle
Generic name:  Rectangle
Description:   A rectangle  doggy door
id:            13
H extent:      4
V extent:      2
Scr area:      8
Geo area:      8.00
Scr perimeter: 8
Geo perimeter: 12.00


Shape Information
-----------------
Type of this:  class Shape const *
Type of *this: class Rhombus
Generic name:  Rhombus
Description:   Diamond shape window on front wall
id:            14
H extent:      7
V extent:      7
Scr area:      25
Geo area:      24.00
Scr perimeter: 12
Geo perimeter: 19.80
```

Marking scheme

30%	Program correctness: **Shape** class hierarchy
30%	Program correctness: : **Canvas** class hierarchy
20%	Program design, encapsulation, information hiding, code reuse, inheritance, polymorphism, proper use of C++ concepts and the STL.
10%	Format, clarity, completeness of output
10%	Concise documentation of nontrivial steps in code, choice of variable names, indentation and readability of program