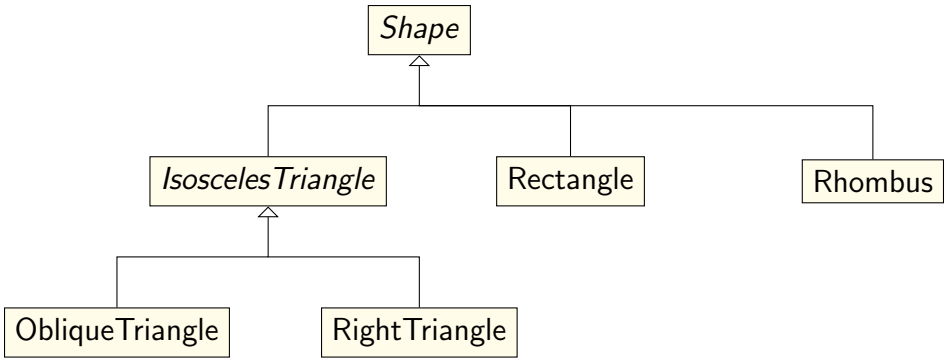
1 Objectives

- To practice fundamental object-oriented programming (OOP) concepts
- To learn how to define an inheritance hierarchy of classes implementing a common interface
- To learn how to define and use virtual functions and how to override them in order to make runtime polymorphism possible in C++
- To learn how to define and use two-dimensional arrays using **array** and **vector**, the two simplest container class templates in the C++ Standard Template Library (STL)
- To provide an opportunity for you to practice programming!

2 Geometric Shape Modeling

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

The geometric shapes considered are simple two-dimensional shapes that can be reasonably depicted textually on the computer screen; namely, squares, rectangles, and specific kinds of triangles and rhombuses. The actual types of our geometric shapes form a single inheritance hierarchy of classes, with the most generalized class **Shape** at the top of the class hierarchy.



So let's begin by specifying characteristics common to all shape objects in the class hierarchy.

2.1 Common Attributes of Shape Objects

- A distinct identity number, an integer, which is to be generated automatically at construction.

- An optional user supplied name, such as "Swimming Pool" for the shape object; defaults to the object's class name.
- An optional user supplied description of the shape object, such as "Montreal's Olympic Stadium"; defaults to the word "Class" followed by the object's class name.

2.2 Common Operations of Shape Objects

1. A constructor that optionally accepts initial values for the shape's description and name.
2. Three accessor (getter) methods, one for each attribute;
3. Two mutator (setter) methods to set the object's name and description;
4. A method that generates and returns a string representation for the shape object;
5. A method to compute the object's geometric area;
6. A method to compute the object's geometric perimeter;
7. A method to compute the object's *screen area*: the number of characters that form the textual image of the shape;
8. A method to compute the object's *screen perimeter*: the number of characters on the borders of the textual image of the shape;
9. A method to draw a textual image for the shape object on a given drawing surface named **Canvas**, which will be introduced in section 6.
10. Two methods returning, respectively, the height and the width of the object's bounding box: the smallest rectangular box enclosing the textual image of the shape.

3 Abstract Shapes

There are two abstract classes in the inheritance hierarchy on page 1: **Shape** and **Isosceles Triangle**.

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

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

Obviously, class **IsoscelesTriangle** must remain abstract as it too lacks information to implement some of **Shape**'s operations, including operation 9.

4 Concrete Shapes

Classes **Rectangle**, **Rhombus**, **RightTriangle** and **ObliqueTriangle** are concrete geometric shapes, picked specifically because they each can be textually rendered into visually identifiable patterns. The specific features of these concrete shapes are listed in the following table.

Specialized Features of Concrete Shapes

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

4.1 Shape Notes

- The lengths of the vertical and horizontal attributes of a shape are measured in character units.

- At construction, a **Rectangle** shape requires the values of both its height and width, whereas the other three shapes each require a single value for the length of their respective horizontal attribute.
- The height and width of a shape's bounding box are *not* stored anywhere; they are computed on demand.

5 Task 1 of 2

Implement the **Shape** inheritance class hierarchy described above, except only for operation 9, which is covered in section 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 methods) and common attributes (shared data) are pushed toward the top of your class hierarchy.

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

```
1 Rectangle rect1(5, 7);
2 cout << rect1.toString() << endl;
3 // or equivalently
4 // cout << rect1 << endl;
```

```
Shape Information
-----------------
Static type:   PK5Shape
Dynamic type:  9Rectangle
Shape name:    Rectangle
Description:   Class Rectangle
id:            1
B. box width:  5
B. box height: 7
Scr area:      35
Geo area:      35.00
Scr perimeter: 20
Geo perimeter: 24.00
```

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 when shape objects are first constructed.

The shape's name defaults to the shape's class name. The shape's description defaults to the word **Class** followed by the class name:

The display box at right shows the output generated in line 2. Note that line 4 would produce the same output, as the output operator overload itself internally calls **toString()**.

Now let's see how the static and dynamic type names at the top of the box are produced.

In general, to get the name of the *static* type of a pointer **p** at runtime you use **typeid(p).name()**, and to get the name of **p**'s *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, **rect1**'s static type name is **PK5Shape** and it's dynamic type name in **toString()** is **9Rectangle**.

The actual names returned by these calls are implementation defined. For example, the output above was generated under g++ 5.4.0, where **PK** in **PK5Shape** means "pointer to ~~**konst**~~ **const**", and **5** in **PK5Shape** means that the type name that follows it is **5** character long.

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

```
1 Rectangle rect1(5, 7);
2 cout << rect1.toString() << endl;
3 // or equivalently
4 // cout << rect1 << endl;
```

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

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

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

```
Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rhombus
Shape name:    Ace
Description:   Ace of diamond
id:            1
B. box width:  17
B. box height: 17
Scr area:      145
Geo area:      144.50
Scr perimeter: 32
Geo perimeter: 48.08
```

Notice that in line 5, 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 6 and 8 would produce the same output; the difference is that the call to **toString()** is implicit in line 8.

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

```
9     Oblique ob(17);
10    cout << ob << endl;
11
12     /* equivalently:
13
14    Shape *obptr = &ob;
15    cout << *obptr << endl;
16
17    Shape &obref = ob;
18    cout << obref << endl;
19    */
```

```
Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Oblique
Shape name:    Oblique
Description:   Class Oblique
id:            3
B. box width:  17
B. box height: 9
Scr area:      81
Geo area:      76.50
Scr perimeter: 32
Geo perimeter: 41.76
```

```
21 RightTriangle rt(10, "Carpenter's square");
22 cout << rt << endl;
```

```
Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class RightTriangle
Shape name:    RightTriangle
Description:   Carpenter's square
id:            1
B. box width:  10
B. box height: 10
Scr area:      55
Geo area:      50.00
Scr perimeter: 27
Geo perimeter: 34.14
```

It is important to note that none of the variables **rect1**, **ob**, **rt**, and **ace** above understands polymorphism because they are all non-pointer and non-reference variables. The polymorphic magic happens through the second argument in the calls to the output **operator**<< at lines 2, 8, 10, and 22. For example, consider the call **cout** << **rt** at lines 22 which can be equivalently written as **operator**<<**(cout, rt)**. The second argument in the call, **rt**, corresponds to the second parameter on the output operator overload:

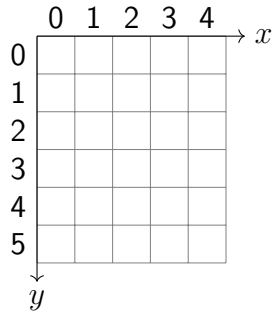
```
ostream& operator<< (ostream& out, const Shape& shape);
```

Specifically, **rt** in the expression **cout** << **rt** in line 22 binds to the second parameter of the output operator overload, named **shape**, which is intentionally a reference variable of type **Shape&** defined to behave polymorphically. Using the reference variable **shape**, the output operator can call shape methods such as **shape.geoArea()**, **shape.geoPerimeter()**, etc., all polymorphically; that is, if **shape** references a rhombus, **shape.geoArea()** calls rhombus's **geoArea()**, if **shape** references a rectangle, **shape.geoArea()** calls rectangle's **geoArea()**, and so on.

6 Task 2 of 2

Let a drawing surface be defined as follows:

A drawing surface is a rectangular array of cells, with a fixed number of rows and columns. The rows are parallel to the x-axis, with row numbers increasing down. The columns are parallel to the y-axis, with column numbers increasing to the right. The origin of the drawing surface is located at $(0, 0)$, the top-left cell at row 0 and column 0.



Implement a class named **Canvas** that models a drawing surface in which the cells each store a character. Your **Canvas** class implementation should provide the following features:

- Internally, the **Canvas** class should use a **std::vector<std::vector<char>>** to represent the cells.
- The class should provide a **Canvas(int rows, int cols, char fillCh=' ')** constructor. The parameters **rows** and **cols** represent, respectively, the number of rows and columns of the canvas under construction. This constructor should initialize every cell with the fill character **fillCh**, which defaults to a blank.
- Subscript operator `[]` overloads, both **const** and non-**const** versions. These operators do not check against bounds, as they effectively reflect the corresponding operators of the underlying vectors.
- A **put(int r, int c, char ch='*')** function that writes **ch** in the cell at row **r** and column **c**. This function checks against bounds, and it simply ignores (*clips*) writes that land outside its boundaries.
- A **getHeight()** method that returns the canvas height (rows).
- A **getWidth()**method that returns the canvas width (columns).
- An **inBounds(int, int)** method that determines whether the specified row and column positions are inside the bounds of this canvas.

  A **clear()** method that takes an optional fill character and writes the cells with that character. The fill character defaults to a blank.
- An output operator overload that writes the entire canvas cells to a given **ostream**.

The **draw** function can now be prototyped in class **Shape** as follows:

```
virtual void draw(Canvas & canvas, // a drawing surface
        int row, // the y (row) coordinate of the bounding box to be drawn.
        int col, // the x (column) coordinate of the bounding box to be drawn.
        char foreChar = '*', // foreground char
        char backChar = ' ') // background char
        const = 0;
```

For example, the **Rectangle** class could implement its **draw()** method as follows:

```
1  void Rectangle::
2  draw(Canvas & canvas, int row, int col, char fChar, char bChar) const
3  {
4     for (int r = 0; r < this->getHeight(); r++) // rows
5     {
6        for (int c = 0; c < this->getWidth(); c++) // columns
7        {
8           canvas[row + r][col + c] = foreChar; // unchecked by vector
9           // canvas.put(row + r, col + c, fChar); // checked by Canvas
10       }
11    }
12 }
```

The following box shows an example of calling **draw()** both polymorphically and non-polymorphically:

Early and Late Binding Example in C++

```
1  // define 4 shapes, one of each shape type
2  Rectangle rect(5, 7);                  // a 5x7 = (width x height) rectangle
3  Oblique pizzaSlice(7);                 // an isosceles with base = 7
4  RightTriangle right(8);                // a right triangle with base = 8
5  Rhombus rhom(7);                       // a rhombus with diagonal = 7
6
7  Canvas poster(10, 75, '.');  //  a 10x75 (rows x columns) canvas filled with dots
8
9  // print above 4 shapes using different foreground characters
10 // Non-polymorphic calls to draw()
11 rect.draw(poster, 0, 30, 'H');          // draw rect at (0, 30) = (row, col)
12 pizzaSlice.draw(poster, 2, 1);          // draw pizzaSlice at (2, 1)
13 right.draw(poster, 2, 10, '\\');        // draw right at (2, 10)
14 rhom.draw(poster, 2, 20, 'o');          // draw rhom at (2, 20)
15
16 // Polymorphic calls to draw()
17 Shape *shapePtr{ nullptr }; // one pointer to draw any shape!
18
19 // print above 4 shapes using different locations, fore/background characters
20 shapePtr = &rect;
21 shapePtr->draw(poster, 2, 45, 'H', '*');  // draw rect at (2, 45) = (row, col)
22
23 shapePtr = &pizzaSlice;
24 shapePtr->draw(poster, 2, 36, '*', '-');  // draw pizzaSlice at (2, 36)
25
26 Shape& rightShapeRef = right; // now let's try a reference
27 rightShapeRef.draw(poster, 1, 65, '\\', '/'); // draw right at (2, 65)
28
29 Shape& rhomShapeRef = rhom;  // ditto
30 rhomShapeRef.draw(poster, 2, 55, ' ', 'o');  // draw rhom at (2, 55)
31
32 cout << poster;
```

Output

```
.............................HHHHH.........................................
.............................HHHHH.............................\///////..
.   *   ..\       ..   o   ...HHHHH.---*---..HHHHH.....ooo ooo...\\//////..
.  ***  ..\\      ..  ooo  ...HHHHH.--***--..HHHHH.....oo   oo...\\\/////..
. ***** ..\\\     .. ooooo ...HHHHH.-*****-..HHHHH.....o     o...\\\\////..
.*******..\\\\    ..ooooooo...HHHHH.*******..HHHHH.....       ...\\\\\///..
.........\\\\\   .. ooooo ...HHHHH..........HHHHH.....o     o...\\\\\\//..
.........\\\\\\  ..  ooo  ..................HHHHH.....oo   oo...\\\\\\/..
.........\\\\\\\ ..   o   ..................HHHHH.....ooo ooo...\\\\\\\..
.........\\\\\\\.............................................................
```

Note that none of the variables **rect**, **pizzaSlice**, **right**, and **rhom** is a pointer or a reference, and hence none of the calls to **draw()** on lines 11-14 is polymorphic. In fact there is no need for polymorphism there as these variables **rect**, **pizzaSlice**, **right**, and **rhom** are each bound to

their corresponding **draw()** function at compile time (early binding).

By contrast, the calls to **draw()** on lines 21, 24, 27, and 30 are all polymorphic because the first two calls are made through shape pointers and that last two through shape references.

7 Deliverables

Header files:	Shape.h, Rectangle.h, Rhombus.h, Isosceles.h, Oblique.h, Canvas.h, RightTriangle.h
Implementation files:	Shape.cpp, Rectangle.cpp, Rhombus.cpp, Isosceles.cpp, Oblique.cpp, RightTriangle.cpp, Canvas.cpp, ShapeTestDriver.cpp
README.txt	A text file, as described in the course outline.

8 Marking scheme

60%	Program correctness: 42% `Shape class hierarchy` 18% `Canvas`
15%	Program design, encapsulation, information hiding, code reuse, proper use of C++ concepts.
10%	No use of operator **new** and operator **delete**. No C-style coding and memory functions such as **malloc**, **alloc**, **realloc**, **free**, etc.
5%	Format, clarity, completeness of output
10%	Javadoc style documentation before introduction of every class and function, Concise documentation of nontrivial steps in code, choice of variable names, indentation and readability of program

9 Shape's Test Driver

```cpp
#include<iostream>
using std::cout;
using std::cin;
using std::endl;
//#include "Shape.h"
//#include "Isosceles"
#include "Rhombus.h"
#include "Rectangle.h"
#include "Oblique.h"
#include "RightTriangle.h"
#include "Canvas.h"

// function prototypes
void drawHouseEalrlyBinding();
void drawHouseLateBinding();

int main()
{
   drawHouseEalrlyBinding();
   drawHouseLateBinding();
   return 0;
}

void drawHouseEalrlyBinding()
{
   // draw a house front view on a 50-column and 50-row titles Canvas
   Canvas poster(50, 50);

   std::string title("a geometric house: front view");
   int pos = 8;
   for (auto ch : title)
   {
      poster.put(0, pos, ch);
      ++pos;
   }

   Oblique roof(41, "house roof");
   roof.draw(poster, 1, 1, '/');// house roof

   Rectangle chimney(2, 10, "chimeny on the roof");
   chimney.draw(poster, 8, 4, '/'); // chimeny on the roof

   Rectangle skylightFrame(9, 5, "frame around skylight");
   skylightFrame.draw(poster, 11, 17, 'h');// frame around skylight

   Rectangle skylight(7, 3, "skylight on the roof");
   skylight.draw(poster, 12, 18, ' '); // skylight on the roof
```

```
48
49      Rectangle frontWall(41, 22, "front wall");
50      frontWall.draw(poster, 23, 1, ':');  // front wall
51
52      Rectangle top_bottom_left_brackets(21, 1, "top and bottom left square brackets");
53      top_bottom_left_brackets.draw(poster, 22, 1, '['); // top left square brackets
54      top_bottom_left_brackets.draw(poster, 44, 1, '['); // bottom left square brackets
55
56      Rectangle top_bottom_right_brackets(20, 1, "top
    and bottom right square brackets");
57      top_bottom_right_brackets.draw(poster, 22, 22, ']'); // top right square brackets
58      top_bottom_right_brackets.draw(poster, 44, 22, ']');// bottom right square brackets
59
60      Rectangle right_wall_brackets(2, 22, "right wall brackets");
61      right_wall_brackets.draw(poster, 23, 40, ']'); // right wall brackets
62
63      Rectangle left_wall_brackets(2, 22, "left wall brackets");
64      left_wall_brackets.draw(poster, 23, 1, '['); // left wall brackets
65
66      Rectangle rightDoor(6, 7, "front right door");
67      rightDoor.draw(poster, 36, 30, '-');// front left door
68
69      Rectangle rightDoorFrame(1, 7, "front right door hinge");
70      rightDoorFrame.draw(poster, 36, 35, 'H');// front left door
71
72      Rectangle leftDoor(6, 7, "front left door");
73      leftDoor.draw(poster, 36, 23, '-');// front left door
74
75      Rectangle leftDoorFrame(1, 7, "front left door hinge");
76      leftDoorFrame.draw(poster, 36, 22, 'H');// front left door
77
78      Rectangle doorsMiddle(2, 7, "vertical center panel between front doors");
79      doorsMiddle.draw(poster, 36, 28, '|'); // vertical center panel between front doors
80
81      Rectangle doorTopBottomBar(14, 1, "door top and bottom bar");
82      doorTopBottomBar.draw(poster, 35, 22, 'H'); // door top bar
83      doorTopBottomBar.draw(poster, 43, 22, '='); // door bottom bar
84
85      Rectangle doorKnobs(2, 1, "door knobs");
86      doorKnobs.draw(poster, 40, 28, 'O'); // vertical center panel between front doors
87
88                                          // Triagle windows above front door
89      Oblique Triagle_above_front_door(8, "triagle door top");
90      Triagle_above_front_door.draw(poster, 24, 22, '*'); // triagle door top
91
92      Rectangle doggyDoor = Rectangle(4, 3, "doggy door");
93      doggyDoor.draw(poster, 40, 3, '~');// doggy door
94
95      Rhombus diamond_shape_window_on_front_wall(7, "diamond shape window on front wall");
96      diamond_shape_window_on_front_wall.draw(poster, 25, 4, 'o', ':');// diamond shape wind
```

```
97
98      Rectangle StairSlash(41, 1, "front stairs slashes");
99      StairSlash.draw(poster, 45, 1, '\\'); // row of front stairs slashes
100     StairSlash.draw(poster, 46, 2, '\\'); // row of front stairs slashes
101     StairSlash.draw(poster, 47, 3, '\\'); // row of front stairs slashes
102     StairSlash.draw(poster, 48, 4, '\\'); // row of front stairs slashes
103
104     Rectangle pole(1, 12, "flag pole");
105     pole.draw(poster, 10, 41, 'i'); // flag pole
106
107     RightTriangle flag(6, "flag");
108     flag.draw(poster, 11, 42, '\\'); // flag
109
110     cout << poster << endl;
111     cout << chimney << endl;
112     cout << roof << endl;
113     cout << skylightFrame << endl;
114     cout << skylight << endl;
115     cout << frontWall << endl;
116     cout << top_bottom_left_brackets << endl;
117     cout << top_bottom_right_brackets << endl;
118     cout << top_bottom_right_brackets << endl;
119     cout << top_bottom_left_brackets << endl;
120     cout << leftDoor << endl;
121     cout << rightDoor << endl;
122     cout << leftDoorFrame << endl;
123     cout << rightDoorFrame << endl;
124     cout << doorsMiddle << endl;
125     cout << Triagle_above_front_door << endl;
126     cout << doggyDoor << endl;
127     cout << diamond_shape_window_on_front_wall << endl;
128     cout << StairSlash << endl;
129     cout << pole << endl;
130     cout << flag << endl;
131 }
```

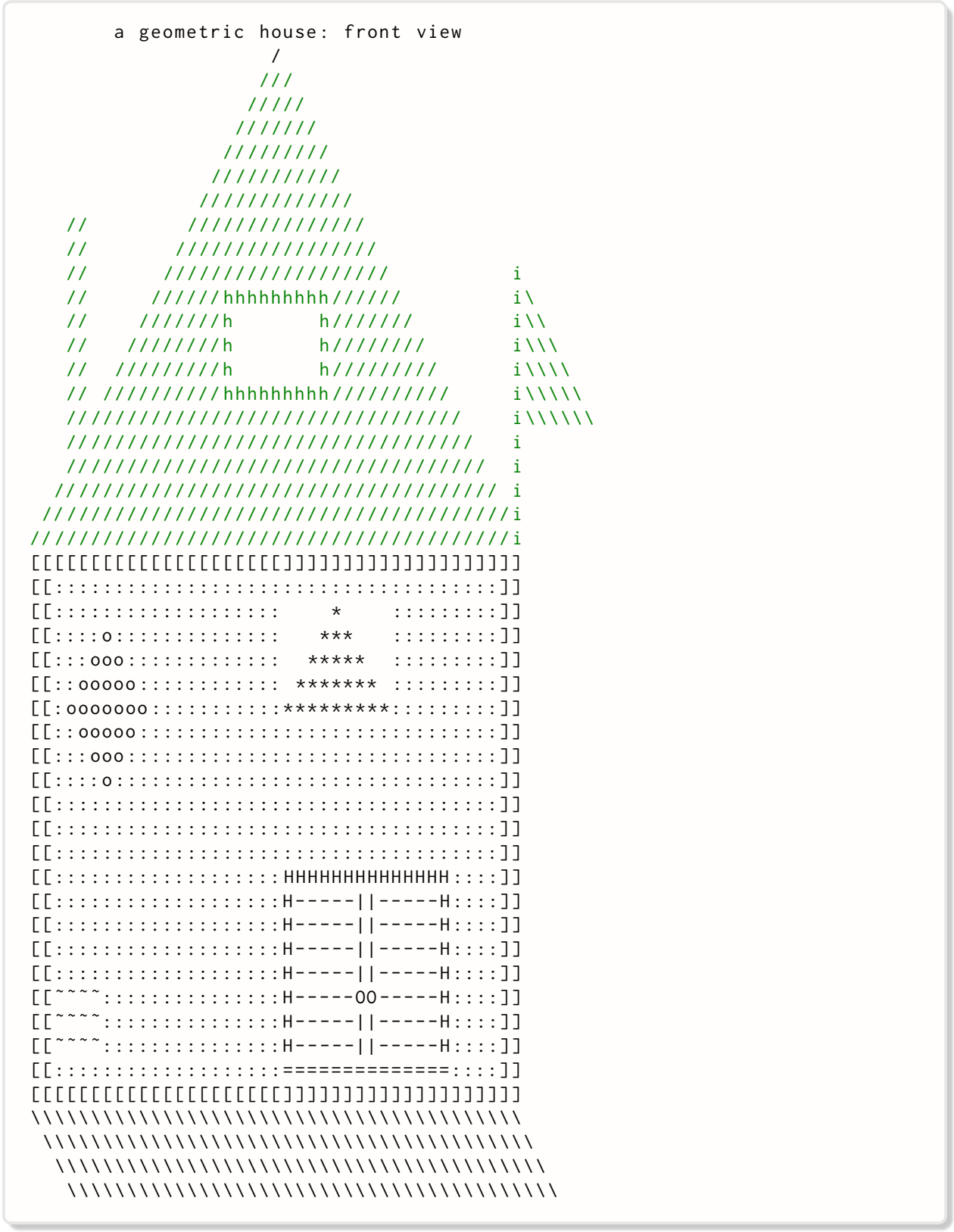
```
132
133 void drawHouseLateBinding()
134 {
135     // draw a house front view on a 50-column and 50-row titles Canvas
136     Canvas poster(50, 50);
137
138     std::string title("a geometric house: front view");
139     int pos = 8;
140     for (auto ch : title)
141     {
142         poster.put(0, pos, ch);
143         ++pos;
144     }
145
146     Shape* shapePtr{ nullptr }; // a pointer to draw any shape
```

```
147
148     Oblique roof(41, "house roof");
149     shapePtr = &roof;
150     shapePtr->draw(poster, 1, 1, '/');// house roof
151
152     Rectangle chimney(2, 10, "chimeny on the roof");
153     shapePtr = &chimney;
154     shapePtr->draw(poster, 8, 4, '/'); // chimeny on the roof
155
156     Rectangle skylightFrame(9, 5, "frame around skylight");
157     shapePtr = &skylightFrame;
158     shapePtr->draw(poster, 11, 17, 'h');// frame around skylight
159
160     Rectangle skylight(7, 3, "skylight on the roof");
161     shapePtr = &skylight;
162     shapePtr->draw(poster, 12, 18, ' '); // skylight on the roof
163
164     Rectangle frontWall(41, 22, "front wall");
165     shapePtr = &frontWall;
166     shapePtr->draw(poster, 23, 1, ':');  // front wall
167
168     Rectangle top_bottom_left_brackets(21, 1, "top and bottom left square brackets");
169     shapePtr = &top_bottom_left_brackets;
170     shapePtr->draw(poster, 22, 1, '['); // top left square brackets
171     shapePtr->draw(poster, 44, 1, '['); // bottom left square brackets
172
173     Rectangle top_bottom_right_brackets(20, 1, "top
    and bottom right square brackets");
174     shapePtr = &top_bottom_right_brackets;
175     shapePtr->draw(poster, 22, 22, ']'); // top right square brackets
176     shapePtr->draw(poster, 44, 22, ']');// bottom right square brackets
177
178     Rectangle right_wall_brackets(2, 22, "right wall brackets");
179     shapePtr = &right_wall_brackets;
180     shapePtr->draw(poster, 23, 40, ']'); // right wall brackets
181
182     Rectangle left_wall_brackets(2, 22, "left wall brackets");
183     shapePtr = &left_wall_brackets;
184     shapePtr->draw(poster, 23, 1, '['); // left wall brackets
185
186     Rectangle rightDoor(6, 7, "front right door");
187     shapePtr = &rightDoor;
188     shapePtr->draw(poster, 36, 30, '-');// front left door
189
190     Rectangle rightDoorFrame(1, 7, "front right door Frame");
191     shapePtr = &rightDoorFrame;
192     shapePtr->draw(poster, 36, 35, 'H');// front left door
193
194     Rectangle leftDoor(6, 7, "front left door");
195     shapePtr = &leftDoor;
196     shapePtr->draw(poster, 36, 23, '-');// front left door
```

```
197
198     Rectangle leftDoorFrame(1, 7, "front left door Frame");
199     shapePtr = &leftDoorFrame;
200     shapePtr->draw(poster, 36, 22, 'H');// front left door
201
202     Rectangle doorsMiddle(2, 7, "vertical center panel between front doors");
203     shapePtr = &doorsMiddle;
204     shapePtr->draw(poster, 36, 28, '|'); // vertical center panel between front doors
205
206     Rectangle doorTopBottomBar(14, 1, "door top and bottom bar");
207     shapePtr = &doorTopBottomBar;
208     shapePtr->draw(poster, 35, 22, 'H'); // door top bar
209     shapePtr->draw(poster, 43, 22, '='); // door bottom bar
210
211     Rectangle doorKnobs(2, 1, "door knobs");
212     shapePtr = &doorKnobs;
213     shapePtr->draw(poster, 40, 28, 'O'); // vertical center panel between front doors
214
215                                          // Triagle windows above front door
216     Oblique Triagle_above_front_door(8, "triagle door top");
217     shapePtr = &Triagle_above_front_door;
218     shapePtr->draw(poster, 24, 22, '*'); // triagle door top
219
220     Rectangle doggyDoor = Rectangle(4, 3, "doggy door");
221     shapePtr = &doggyDoor;
222     shapePtr->draw(poster, 40, 3, '~');// doggy door
223
224     Rhombus diamond_shape_window_on_front_wall(7, "diamond shape window on front wall");
225     shapePtr = &diamond_shape_window_on_front_wall;
226     shapePtr->draw(poster, 25, 4, 'o', ':');// diamond shape window on front wall
227
228     Rectangle StairSlash(41, 1, "stair slash"); // a row of  stair slashes
229     shapePtr = &StairSlash;
230     shapePtr->draw(poster, 45, 1, '\\'); // row of front stairs slashes
231     shapePtr->draw(poster, 46, 2, '\\'); // row of front stairs slashes
232     shapePtr->draw(poster, 47, 3, '\\'); // row of front stairs slashes
233     shapePtr->draw(poster, 48, 4, '\\'); // row of front stairs slashes
234
235     Rectangle pole(1, 12, "flag pole");
236     shapePtr = &pole;
237     shapePtr->draw(poster, 10, 41, 'i'); // flag pole
238
239     RightTriangle flag(6, "flag");
240     shapePtr = &flag;
241     shapePtr->draw(poster, 11, 42, '\\'); // flag
```

```
242
243
244     cout << poster << endl;
245     //cout << chimney << endl;
246     //cout << roof << endl;
247     //cout << skylightFrame << endl;
248     //cout << skylight << endl;
249     //cout << frontWall << endl;
250     //cout << top_bottom_left_brackets << endl;
251     //cout << top_bottom_right_brackets << endl;
252     //cout << top_bottom_right_brackets << endl;
253     //cout << top_bottom_left_brackets << endl;
254     //cout << leftDoor << endl;
255     //cout << rightDoor << endl;
256     //cout << leftDoorFrame << endl;
257     //cout << rightDoorFrame << endl;
258     //cout << doorsMiddle << endl;
259     //cout << Triagle_above_front_door << endl;
260     //cout << doggyDoor << endl;
261     //cout << diamond_shape_window_on_front_wall << endl;
262     //cout << StairSlash << endl;
263     //cout << pole << endl;
264     //cout << flag << endl;
265 }
```

10 Test Driver Output



```
Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Shape name:    Rectangle
Description:   chimeny on the roof
id:            2
B. box width:  2
B. box height: 10
Scr area:      20
Geo area:      20.00
Scr perimeter: 20
Geo perimeter: 24.00


Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Oblique
Shape name:    Oblique
Description:   house roof
id:            1
B. box width:  41
B. box height: 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
Shape name:    Rectangle
Description:   frame around skylight
id:            3
B. box width:  9
B. box height: 5
Scr area:      45
Geo area:      45.00
Scr perimeter: 24
Geo perimeter: 28.00
```

```
132
133
134
135 Shape Information
136 -----------------
137 Static type:   class Shape const *
138 Dynamic type:  class Rectangle
139 Shape name:    Rectangle
140 Description:   skylight on the roof
141 id:            4
142 B. box width:  7
143 B. box height: 3
144 Scr area:      21
145 Geo area:      21.00
146 Scr perimeter: 16
147 Geo perimeter: 20.00
148
149
150
151 Shape Information
152 -----------------
153 Static type:   class Shape const *
154 Dynamic type:  class Rectangle
155 Shape name:    Rectangle
156 Description:   front wall
157 id:            5
158 B. box width:  41
159 B. box height: 22
160 Scr area:      902
161 Geo area:      902.00
162 Scr perimeter: 122
163 Geo perimeter: 126.00
164
165
166
167 Shape Information
168 -----------------
169 Static type:   class Shape const *
170 Dynamic type:  class Rectangle
171 Shape name:    Rectangle
172 Description:   top and bottom left square brackets
173 id:            6
174 B. box width:  21
175 B. box height: 1
176 Scr area:      21
177 Geo area:      21.00
178 Scr perimeter: 40
179 Geo perimeter: 44.00
```

```
Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Shape name:    Rectangle
Description:   top  and bottom right square brackets
id:            7
B. box width:  20
B. box height: 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
Shape name:    Rectangle
Description:   top  and bottom right square brackets
id:            7
B. box width:  20
B. box height: 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
Shape name:    Rectangle
Description:   top and bottom left square brackets
id:            6
B. box width:  21
B. box height: 1
Scr area:      21
Geo area:      21.00
Scr perimeter: 40
Geo perimeter: 44.00
```

```
228
229
230
231 Shape Information
232 -----------------
233 Static type:   class Shape const *
234 Dynamic type:  class Rectangle
235 Shape name:    Rectangle
236 Description:   front left door
237 id:            12
238 B. box width:  6
239 B. box height: 7
240 Scr area:      42
241 Geo area:      42.00
242 Scr perimeter: 22
243 Geo perimeter: 26.00
244
245
246
247 Shape Information
248 -----------------
249 Static type:   class Shape const *
250 Dynamic type:  class Rectangle
251 Shape name:    Rectangle
252 Description:   front right door
253 id:            10
254 B. box width:  6
255 B. box height: 7
256 Scr area:      42
257 Geo area:      42.00
258 Scr perimeter: 22
259 Geo perimeter: 26.00
260
261
262
263 Shape Information
264 -----------------
265 Static type:   class Shape const *
266 Dynamic type:  class Rectangle
267 Shape name:    Rectangle
268 Description:   front left door hinge
269 id:            13
270 B. box width:  1
271 B. box height: 7
272 Scr area:      7
273 Geo area:      7.00
274 Scr perimeter: 12
275 Geo perimeter: 16.00
```

```
Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Shape name:    Rectangle
Description:   front right door hinge
id:            11
B. box width:  1
B. box height: 7
Scr area:      7
Geo area:      7.00
Scr perimeter: 12
Geo perimeter: 16.00


Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Shape name:    Rectangle
Description:   vertical center panel between front doors
id:            14
B. box width:  2
B. box height: 7
Scr area:      14
Geo area:      14.00
Scr perimeter: 14
Geo perimeter: 18.00


Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Oblique
Shape name:    Oblique
Description:   triagle door top
id:            17
B. box width:  9
B. box height: 5
Scr area:      25
Geo area:      22.50
Scr perimeter: 16
Geo perimeter: 22.45
```

```
Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Shape name:    Rectangle
Description:   doggy door
id:            18
B. box width:  4
B. box height: 3
Scr area:      12
Geo area:      12.00
Scr perimeter: 10
Geo perimeter: 14.00



Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rhombus
Shape name:    Rhombus
Description:   diamond shape window on front wall
id:            19
B. box width:  7
B. box height: 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
Shape name:    Rectangle
Description:   front stairs slashes
id:            20
B. box width:  41
B. box height: 1
Scr area:      41
Geo area:      41.00
Scr perimeter: 80
Geo perimeter: 84.00
```

```
Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class Rectangle
Shape name:    Rectangle
Description:   flag pole
id:            21
B. box width:  1
B. box height: 12
Scr area:      12
Geo area:      12.00
Scr perimeter: 22
Geo perimeter: 26.00


Shape Information
-----------------
Static type:   class Shape const *
Dynamic type:  class RightTriangle
Shape name:    RightTriangle
Description:   flag
id:            22
B. box width:  6
B. box height: 6
Scr area:      21
Geo area:      18.00
Scr perimeter: 15
Geo perimeter: 20.49
```

```
        a geometric house: front view
                      /
                     ///
                    /////
                   ///////
                  /////////
                 ///////////
                /////////////
    //         ///////////////
    //        /////////////////
    //       ///////////////////           i
    //      //////hhhhhhhhh//////          i\
    //     ///////h        h///////         i\\
    //    ////////h        h////////        i\\\
    //   /////////h        h/////////       i\\\\
    //  //////////hhhhhhhhh//////////      i\\\\\
    ////////////////////////////////      i\\\\\\
    ////////////////////////////////    i
    ////////////////////////////////   i
   ////////////////////////////////////i
  /////////////////////////////////////i
 //////////////////////////////////////i
[[[[[[[[[[[[[[[[[[[[]]]]]]]]]]]]]]]]]]]
[[:::::::::::::::::::::::::::::::::::::]]
[[::::::::::::::::::    *    ::::::::::]]
[[::::o::::::::::::::   ***   ::::::::::]]
[[:::ooo:::::::::::::  *****  ::::::::::]]
[[::ooooo:::::::::::: ******* ::::::::::]]
[[:ooooooo::::::::::*********::::::::::]]
[[::ooooo:::::::::::::::::::::::::::::::]]
[[:::ooo:::::::::::::::::::::::::::::::::]]
[[::::o:::::::::::::::::::::::::::::::::]]
[[:::::::::::::::::::::::::::::::::::::]]
[[:::::::::::::::::::::::::::::::::::::]]
[[:::::::::::::::::::::::::::::::::::::]]
[[::::::::::::::::::HHHHHHHHHHHHHH::::]]
[[::::::::::::::::::H-----||-----H::::]]
[[::::::::::::::::::H-----||-----H::::]]
[[::::::::::::::::::H-----||-----H::::]]
[[::::::::::::::::::H-----||-----H::::]]
[[~~~~::::::::::::::H-----OO-----H::::]]
[[~~~~::::::::::::::H-----||-----H::::]]
[[~~~~::::::::::::::H-----||-----H::::]]
[[::::::::::::::::::==============::::]]
[[[[[[[[[[[[[[[[[[[[]]]]]]]]]]]]]]]]]]]
\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
 \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
  \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
   \\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\\
```