COMP 5421/1-BB | Assignment 4 | Due by August 1, 2021

1 Purpose

- Practice fundamental object-oriented programming (OOP) concepts
- Implement an inheritance hierarchy of classes `C++`
- Learn about virtual functions, overriding, and polymorphism in `C++`
- Use two-dimensional arrays using `array` and `vector`, the two simplest container class templates in the `C++` Standard Template Library (STL)
- Use modern `C++` smart pointers, avoiding calls to the `delete` operator for good!

2 Overview

Using simple two-dimensional geometric shapes, this assignment will give you practice with the fundamental principles of object-oriented programming (OOP).

The assignment starts by abstracting the essential attributes and operations common to the geometric shapes of interest in this assignment, namely, rhombus, rectangle, and two kinds of triangle shapes.

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

In addition to implementing the shape classes, you will be tasked to implement a `Canvas` class whose objects can be used by the shape objects to draw on.

The geometric shapes of interest in this assignment are four simple two-dimensional shapes which can be textually rendered into visually identifiable images on the computer screen; specifically: rhombus , rectangle, and two special kinds of isosceles triangles.

Here are examples of the specific shapes of interest:

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

Rectangle, 6×9

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

Rhombus, 5×5

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

Right Triangle, 6×6

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

Acute Triangle, 5×9

3 Modeling 2D Geometric Shapes

3.1 Common Attributes: Data

The 2D shapes of interest to us have five attributes in common. Specifically, each shape has

- a *name*, a string object; for example, "Book" for a rectangular shape
- an *identity number*, a unique positive integer, distinct from that of all the other shapes
- a *pen character*, the single character to use when drawing the shape
- a *height*, a non-negative integer
- a *width*, a non-negative integer

Note Here, we assume that the height and width of a shape measure, respectively, that shape's vertical and horizontal attributes, although they may be called by a different name for different shapes; for example, both attributes for a rhombus are called "diagonal", and the horizontal attribute of a triangle is called "base."

3.2 Common Operations: Interface

Listed below are the services that every concrete 2D geometric object is expected to provide.

3.2.1 General Operations

1. A constructor that accepts as parameters the initial values of a shape's `height`, `width`, `pen`, and `name`, in that order.
2. Five accessor (getter) methods, one for each attribute;
3. Two mutator (setter) methods for setting the `name` and `pen` members;
4. A `toString()` method that forms and returns a string representation of the `this` shape

3.2.2 Shape-Specific Operations

5. Two mutator (setter) methods for setting the `height` and `width` members;
6. A method to compute and return the shape's geometric area;
7. A method to compute and return the shape's geometric perimeter;
8. A method to compute the shape's *textual area*, which is the number of characters that form the textual image of the shape;
9. A method to compute the shape's *textual perimeter*, which is the number of characters on the borders of the textual image of the shape;
10. A method that *draws* a textual image of the shape on a `Canvas` object using the shape's pen character.

4 Modeling Specialized 2D Geometric Shapes

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

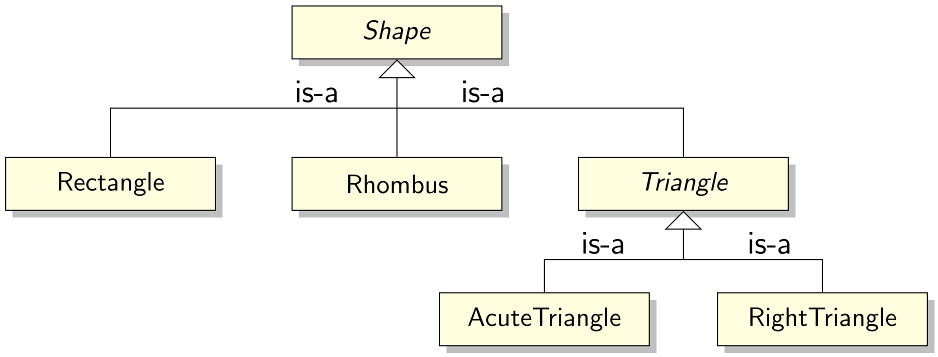


Figure 1: A UML class diagram showing an inheritance hierarchy specified by two abstract classes *`Shape`* and *`Triangle`*, and by four concrete classes `Rectangle`, `Rhombus`, `AcuteTriangle`, and `RightTriangle`.

Encapsulating the attributes and operations common to all shapes, class *`Shape`* must necessarily be *abstract*[1] because the shapes it models are so general that it simply would not know how to implement the operations specified in section 3.2.2.

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

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

Similarly, class *`Triangle`* must be abstract, since it would have no knowledge about the specific triangular shapes it generalizes.

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

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

Any class derived from an abstract class will itself be abstract unless it overrides all the pure virtual functions it inherits.

[2] A pointer (or reference) to an object with a virtual member function has two types: *static* and *dynamic*.

- *static* type: refers to its type as defined in the source code and thus cannot change. For example, the static type of the pointer variable `pf` as in `Foo* pf;` is `Foo*`, a pointer to `Foo`, a type that cannot be changed, in the sense that `pf` will always remain a pointer to `Foo`.
- *dynamic* type: refers to the type of the object the pointer points to (or references) at runtime and thus can change during runtime. For example, although `pf` points to (stores the address of) any shape in the inheritance hierarchy, it may point to different shapes during its lifespan.

5 Concrete Shapes

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

Properties↓ Shapes→	Rectangle	Rhombus	Right Triangle	Acute Triangle
Construction values	h, w	d, if d is odd; else $d \leftarrow d+1$	b	b, if b is odd; else $b \leftarrow b+1$
Height	h	d	b	$(b+1)/2$
Width	w	d	b	b
Geometric area	hw	$d^2/2$	$hb/2$	$hb/2$
Geometric perimeter	$2(h+w)$	$(2\sqrt{2})d$	$(2+\sqrt{2})h$	$b+\sqrt{b^2+4h^2}$
textual area	hw	$2n(n+1)+1$, $n = \lfloor d/2 \rfloor$	$h(h+1)/2$	h^2
textual perimeter	$2(h+w)-4$	$2(d-1)$	$3(h-1)$	$4(h-1)$
Sample textual images of shapes and their dimensions	********* ********* ********* ********* *********	* *** ***** *** *	* ** *** **** *****	* *** ***** ******* *********
	$w=9,\ h=5$	$d=5$	$b=5,\ h=b$	$b=9,\ h=\frac{b+1}{2}$
Default name	Rectangle	Diamond	Ladder	Wedge
Default pen character	*	*	*	*

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

5.1 Shape Notes

- The unit of length is a single character; thus, both the height and width of a shape are measured in characters.
- At construction, a `Rectangle` shape requires the values of both its height and width, whereas the other three concrete shapes each require a single value for the length of their respective horizontal attribute.

6 Task 1 of 2

Implement the `Shape` inheritance class hierarchy described above. It is completely up to you to decide which operations should be virtual, pure virtual, or non-virtual, provided that it satisfies a few simple requirements.

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

6.1 Modeling 2D Triangle Shapes

6.1.1 Common Attributes

The common attributes of triangles are their heights and bases which are already being represented by `height` and `width` data members in *`Shape`*.

6.1.2 Common Operations

- A method to return a geometric area of the triangle

  Note: Without knowledge about its shape, a `Triangle` object can compute its area based on its height and width.

6.1.3 Type-Specific Operations

- A method to return a triangle's height which may depend on is base
- A method to return a triangle's width which may depend on is height
- A method to return a geometric perimeter of the triangle

  Note: Without knowledge about its shape, a `Triangle` object is unable to compute its perimeter based on its height and width.

7 Some Examples

Sourse code

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

Output

```
1  Shape Information
2  -----------------
3  id:                  1
4  Shape name:          Rectangle
5  Pen character:       *
6  Height:              5
7  Width:               7
8  Textual area:        35
9  Geometric area:      35.00
10 Textual perimeter:   20
11 Geometric perimeter: 24.00
12 Static type:         PK5Shape
13 Dynamic type:        9Rectangle
```

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

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

Lines 4-5 of the output show object `rect`'s name and pen character, and lines 6-7 show `rect`'s width and height, respectively.

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

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

As you can see on lines 12-13, `rect`'s static type name is `PK5Shape` and it's dynamic type name is `9Rectangle`. The actual names returned by these calls are implementation defined. For example, the output above was generated under g++ (GCC) 10.2.0, where `PK` in `PK5Shape` means "pointer to ~~konst~~ `const`", and `5` in `5Shape` means that the name "`Shape`" that follows it is `5` character long.

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

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

```
1  Shape Information
2  -----------------
3  id:                  1
4  Shape name:          Rectangle
5  Pen character:       *
6  Height:              5
7  Width:               7
8  Textual area:        35
9  Geometric area:      35.00
10 Textual perimeter:   20
11 Geometric perimeter: 24.00
12 Static type:         class Shape const *
13 Dynamic type:        class Rectangle
```

Here is an example of a `Rhombus` object:

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

```
14 Shape Information
15 -----------------
16 id:                  2
17 Shape name:          Ace of diamond
18 Pen character:       v
19 Height:              17
20 Width:               17
21 Textual area:        145
22 Geometric area:      144.50
23 Textual perimeter:   32
24 Geometric perimeter: 48.08
25 Static type:         class Shape const *
26 Dynamic type:        class Rhombus
```

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

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

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

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

```
27 Shape Information
28 -----------------
29 id:                  3
30 Shape name:          Wedge
31 Pen character:       *
32 Height:              9
33 Width:               17
34 Textual area:        81
35 Geometric area:      76.50
36 Textual perimeter:   32
37 Geometric perimeter: 41.76
38 Static type:         class Shape const *
39 Dynamic type:        class AcuteTriangle
```

```
21 RightTriangle
22    rt{ 10, 'L', "Carpenter's square" }
23 cout << rt << endl;
24 // or equivalently
25 // cout << rt.toString() << endl;
```

```
40 Shape Information
41 -----------------
42 id:                  4
43 Shape name:          Carpenter's square
44 Pen character:       L
45 Height:              10
46 Width:               10
47 Textual area:        55
48 Geometric area:      50.00
49 Textual perimeter:   27
50 Geometric perimeter: 34.14
51 Static type:         class Shape const *
52 Dynamic type:        class RightTriangle
```

7.1 Polymorphic Magic

Note that on line 22 in the source code above, `rt` is a regular object variable, as opposed to a pointer (or reference) variable pointing to (or referencing) an object; as such, `rt` cannot make polymorphic calls. That's because in C++ the calls made by a regular object, such as `rect`,`ace`, `at`, and `rt`, to any function (virtual or not) are bound at compile time (early binding).

Polymorphic magic happens through the second argument in the calls to the output `operator<<` at lines 4, 9, 11, and 23. For example, consider the call `cout << rt` on line 23 which is equivalent to `operator<<(cout, rt)`. The second argument in the call, `rt`, corresponds to the second parameter of the output operator overload:

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

Specifically, `rt` in line 23 gets bound to parameter `shp` which is a reference and as such, can call virtual functions of `Shape` polymorphically. That means, the decision as to which function to invoke depends on the type of the object referenced by `shp` at run time (late binding).

For example, if `shp` references a `Rhombus` object, then the call `shp.geoArea()` binds to `Rhombus::geoArea()`, if `shp` references a `Rectangle` object, then `shp.geoArea()` binds to `Rectangle::geoArea()`, and so on.

However, consider `rt` on line 25; although `rt` is not a reference or a pointer, it is the invoking object in the call `rt.toString()` which is represented inside `Shape::toString()` by the `this` pointer, which in fact can call virtual functions of `Shape` (the base class) polymorphically.

7.2 Shape's Draw Function

```
virtual Canvas draw() const = 0; // concrete derived classes must implement it
```

Introduced in `Shape` as a pure member function, the `draw()` function forces concrete derived classes to implement it.

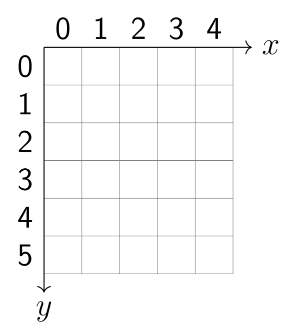
Defining a `Canvas` object like so

```
Canvas  can { getHeight(), getWidth() };
```

the `draw` function draws on `can` using its `put` members function, something like this:

```
can.put(r, c, penChar); // write penChar in cell at row r and column c
```

A `Canvas` object models a two-dimensional grid as abstracted in the Figure at right. The rows of the grid are parallel to the x-axis, with row numbers increasing down. The columns of the grid 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 $(0, 0)$ at row 0 and column 0.



7.3 Examples Continued

```

Canvas aceCan = ace.draw();
cout << aceCan << endl;
```

```
        v
       vvv
      vvvvv
     vvvvvvv
    vvvvvvvvv
   vvvvvvvvvvv
  vvvvvvvvvvvvv
 vvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvv
 vvvvvvvvvvvvvvv
  vvvvvvvvvvvvv
   vvvvvvvvvvv
    vvvvvvvvv
     vvvvvvv
      vvvvv
       vvv
        v
```

```

Canvas rectCan = rect.draw();
cout << rectCan << endl;
```

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

```

at.setPen('^');
Canvas atCan = at.draw();
cout << atCan << endl;
```

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

```
Canvas rtCan = rt.draw();
cout << rtCan << endl;
```

```
L
LL
LLL
LLLL
LLLLL
LLLLLL
LLLLLLL
LLLLLLLL
LLLLLLLLL
LLLLLLLLLL
```

A Canvas object can be flipped both vertically and horizontally:

```
rt.setPen('O');
Canvas rtQuadrant_1 = rt.draw();
cout << rtQuadrant_1 << endl;
```

```
O
OO
OOO
OOOO
OOOOO
OOOOOO
OOOOOOO
OOOOOOOO
OOOOOOOOO
OOOOOOOOOO
```

```
Canvas rtQuadrant_2 = rtQuadrant_1.flip_horizontal();
cout << rtQuadrant_2 << endl;
```

```
         O
        OO
       OOO
      OOOO
     OOOOO
    OOOOOO
   OOOOOOO
  OOOOOOOO
 OOOOOOOOO
OOOOOOOOOO
```

```

Canvas rtQuadrant_3 = rtQuadrant_2.flip_vertical();
cout << rtQuadrant_3 << endl;
```

```
0000000000
 000000000
  00000000
   0000000
    000000
     00000
      0000
       000
        00
         0
```

```

Canvas rtQuadrant_4 = rtQuadrant_3.flip_horizontal();
cout << rtQuadrant_4 << endl;
```

```
0000000000
000000000
00000000
0000000
000000
00000
0000
000
00
0
```

Now, let's create a polymorphic vector of shapes and draw them polymorphically:

```cpp

// first, create a polymorphic
// vector<smart pointer to Shape>:
std::vector<std::unique_ptr<Shape>> shapeVec;

// Next, add some shapes to shapeVec
shapeVec.push_back
   (std::make_unique<Rectangle>(5, 7));
shapeVec.push_back
   (std::make_unique<Rhombus>(16, 'v', "Ace"));
shapeVec.push_back
   (std::make_unique<AcuteTriangle>(17));
shapeVec.push_back
   (std::make_unique<RightTriangle>(10, 'L'));

// now, draw the shapes in shapeVec
for (const auto& shp : shapeVec)
   cout << shp->draw() << endl;
```

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

        v
       vvv
      vvvvv
     vvvvvvv
    vvvvvvvvv
   vvvvvvvvvvv
  vvvvvvvvvvvvv
 vvvvvvvvvvvvvvv
vvvvvvvvvvvvvvvvv
 vvvvvvvvvvvvvvv
  vvvvvvvvvvvvv
   vvvvvvvvvvv
    vvvvvvvvv
     vvvvvvv
      vvvvv
       vvv
        v

        *
       ***
      *****
     *******
    *********
   ***********
  *************
 ***************
*****************

L
LL
LLL
LLLL
LLLLL
LLLLLL
LLLLLLL
LLLLLLLL
LLLLLLLLL
LLLLLLLLLL
```

8 Task 2 of 2

Implement a Canvas class using the following declaration. Feel free to introduce other private member functions of your choice to facilitate the operations of the other members of the class.

```
class Canvas
{
public:
   // all special members are defaulted because 'grid',
   // the only data member, is self-sufficient and efficient; that is,
   // it is equipped to handle the corresponding operations efficiently
   Canvas()                             = default;
   virtual ~Canvas()                    = default;
   Canvas(const Canvas&)                = default;
   Canvas(Canvas&&)                     = default;
   Canvas& operator=(const Canvas&)     = default;
   Canvas& operator=(Canvas&&)          = default;

protected:
   vector<vector<char> > grid{};              // the only data member
   bool check(int r, int c)const;             // validates row r and column c
   void resize(size_t rows, size_t cols);     // resizes this Canvas's dimensions

public:
   // creates this canvas's (rows x columns) grid filled with blank characters
   Canvas(int rows, int columns, char fillChar = ' ');

   int getRows()const;                  // returns height
   int getColumns()const;               // returns width
   Canvas flip_horizontal()const;       // flips this canvas horizontally
   Canvas flip_vertical()const;         // flips this canvas vertically
   void print(ostream&) const;          // prints to ostream
   char get(int r, int c) const;        // returns char at row r and column c
   void put(int r, int c, char ch);     // puts ch at row r and column c; this is the
                                        // only function used by a shape's draw functon;
                                        // returns doing nothing if r or c is invalid

   // draws text starting at row r and col c on the canvas
   void drawString(int r, int c, const std::string text);

   // copies the non-blank characters of "can" onto the invoking canvas;
   // maps can's origin to row r and column c on the invoking canvas
   void overlap(const Canvas& can, size_t r, size_t c);
};
ostream& operator<< (ostream& sout, const Canvas& f);
```

8.1 FYI

To make the assignment workload lighter, the following features were dropped from the original version of `Canvas`. They are listed here so that you might want to implement them some time after the exam to enhance your `Canvas` class.

- Allow the user to index both rows and column from 1
- Overload the function call operator as a function of two `size_t` arguments to write on a canvas, similar to `put`. For example:

```
char ch {'*'};
can(1, 2) = ch;
// similar to
can.put(1, 2, ch);

ch = can(1, 2);
// similar to
ch = can.get(1,2)
```

  To serve both `const` and non-`const` objects of `Canvas`, provide two version of the operator.

- Overload the subscript operator to to support this code segment:

```
char ch {'*'};
can[1][2] = ch;
// similar to
can.put(1, 2, ch);

ch = can[1][2];
// similar to
ch = can.get(1,2)
```

  To serve both `const` and non-`const` objects of `Canvas`, provide two version of the operator.

- Overload the binary `operator+` to join two `Canvas` objects horizontally. The returning `Canvas` object will be large enough to accommodate both `Canvas` objects.

Deliverables

Header files:	Shape.h, Triangle.h, Rectangle.h, Rhombus.h, AcuteTriangle.h, RightTriangle.h, Canvas.h,
Implementation files:	Shape.cpp, Triangle.cpp, Rectangle.cpp, Rhombus.cpp, AcuteTriangle.cpp, RightTriangle.cpp, Canvas.cpp, and ShapeTestDriver.cpp
README.txt	A text file (see the course outline).

9 Grading scheme

Task 1: 70% Shape classes

Task 2: 30% Slot machine class

Each task is graded as follows:

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

10 Sample Test Driver

10.1 ShapeTestDriver.cpp

```
#include<iostream>
#include<vector>

#include "Rhombus.h"
#include "Rectangle.h"
#include "AcuteTriangle.h"
#include "RightTriangle.h"
#include "Canvas.h"
#include "ShapeTestDriver.h"

using std::cout;
using std::endl;

void drawHouse();
void drawHouseElement(Canvas& can, Shape& shp, int row, int col);

int main()
{
   drawHouse();
   return 0;
}
```

10.2 Preparing to Make Polymorphic Calls

To reduce repetitive code, we define the following function which draws a given shape on a given canvas polymorphically.

Notice that the `shp` parameter is a reference of type `Shape&`, enabling `shp` to handle polymorphic shape calls.

Note that there is no need to pass the `shp` parameter by `Shape*` unless we are prepared to take charge of managing the storage it points to.

If the `shp` parameter must be a pointer, use a smart pointer such as `unique_ptr<Shape>`.

```

void drawHouseElement(Canvas& house_canvas, Shape& shp, int row, int col)
{
   cout << shape << "\n========================================\n";
   Canvas can_shape = shape.draw();
   house_canvas.overlap(can_shape, row, col);
}
```

10.3 Drawing Front View of a House

In the following functions, the offsets declared on lines 33 and 34 allow the programmer to index both rows and columns from 1; that is, the programmer considers the top-left grid cell located at (1,1). However, these offsets are used to convert the programmer's 1-based counting to C++'s 0-based counting. (Ideally, Canvas should provide this service but that is not a requirement in this assignment).

```
29
30 // Using our four geometric shapes, draws a pattern that looks line a house
31 void drawHouse()
32 {
33    int row_offset = -1;
34    int col_offset = -1;
35
36    // create a 50-row by 72-column Canvas as a host canvas
37    Canvas hostCan(50, 72);
38    hostCan.drawString(row_offset + 1, col_offset + 10, "a geometric house: front view");
39
40    RightTriangle roof(20, '\\', "Right half of roof");
41    Canvas roof_right_can = roof.draw();
42    hostCan.overlap(roof_right_can, row_offset + 4, col_offset + 27);
43
44    roof.setPen('/');
45    Canvas roof_left_can = roof.draw().flip_horizontal();
46    hostCan.overlap(roof_left_can, row_offset + 4, col_offset + 7);
47
48    hostCan.drawString(row_offset + 23, col_offset + 8,
49             "[] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []");
50
51    Rectangle chimneyL(5, 1, '|', "left edge chimeny");
52    drawHouseElement(hostCan, chimneyL, row_offset + 14, col_offset + 12);
53
54    Rectangle chimneyR(4, 1, '|', "left edge chimeny");
55    drawHouseElement(hostCan, chimneyR, row_offset + 14, col_offset + 13);
56
57    Rectangle antenna_stem(11, 1, 'I', "antenna stem");
58    drawHouseElement(hostCan, antenna_stem, row_offset + 11, col_offset + 45);
59
60    RightTriangle antenna(5, '=', "Right antenna wing");
61    Canvas antenna_Q1 = antenna.draw();
62    Canvas antenna_Q2 = antenna_Q1.flip_horizontal();
63    Canvas antenna_Q3 = antenna_Q2.flip_vertical();
64    Canvas antenna_Q4 = antenna_Q1.flip_vertical();
```

```
hostCan.overlap(antenna_Q3, row_offset + 11, col_offset + 40);
hostCan.overlap(antenna_Q4, row_offset + 11, col_offset + 46);

Rectangle wall(18, 1, '[', "vertical left and right brackets");
drawHouseElement(hostCan, wall, row_offset + 24, col_offset + 8);
drawHouseElement(hostCan, wall, row_offset + 24, col_offset + 44);
wall.setPen(']');
drawHouseElement(hostCan, wall, row_offset + 24, col_offset + 9);
drawHouseElement(hostCan, wall, row_offset + 24, col_offset + 45);

Rectangle line(1, 66, '-', "horizontal lines depicting the ground");
for (size_t c = 1; c <= 6; c++)
{
   drawHouseElement(hostCan, line, row_offset + 40 + c, col_offset + 7 - c);
}
hostCan.drawString(row_offset + 40, col_offset + 8,
         "[][][][][][][][][][][][][][][][][][][]");
hostCan.drawString(row_offset + 41, col_offset + 8,
         "[][][][][][][][][][][][][][][][][][][]");

Rectangle door_step(1, 12, '/', "door step");
drawHouseElement(hostCan, door_step, row_offset + 39, col_offset + 21);

Rectangle door(12, 12, '|', "door");
drawHouseElement(hostCan, door, row_offset + 27, col_offset + 21);

Rectangle door_edge(1, 10, '=', "door top/bottom edge");
drawHouseElement(hostCan, door_edge, row_offset + 27, col_offset + 22);
drawHouseElement(hostCan, door_edge, row_offset + 38, col_offset + 22);

Rectangle door_knob(1, 1, 'O', "door knob");
drawHouseElement(hostCan, door_knob, row_offset + 33, col_offset + 22);

hostCan.drawString(row_offset + 26, col_offset + 25, "5421");

Rhombus window(5, '+', "left window");
drawHouseElement(hostCan, window, row_offset + 28, col_offset + 14);
drawHouseElement(hostCan, window, row_offset + 28, col_offset + 35);

Rectangle tree_trunk(5, 3, 'H', "tree trunk");
drawHouseElement(hostCan, tree_trunk, row_offset + 36, col_offset + 60);

AcuteTriangle leaves(7, '*', "top level leaves");
drawHouseElement(hostCan, leaves, row_offset + 21, col_offset + 58);
leaves.setWidth_cols(11);
```

```
110     drawHouseElement(hostCan, leaves, row_offset + 23, col_offset + 56);
111     leaves.setWidth_cols(19);
112     drawHouseElement(hostCan, leaves, row_offset + 26, col_offset + 52);
113 
114     hostCan.drawString(row_offset + 13, col_offset + 11, "\\||/");
115     hostCan.drawString(row_offset + 12, col_offset + 11, "_/\\_");
116 
117     // finally, reveal the house image
118     cout << hostCan << "-------------------------------" << endl;
119     return;
120 }
```

10.4 Output

For the sake of brevity, string representation of the shape objects printed on line 25 are not shown.

```
         a geometric house: front view


                         /\
                        //\\
                       ///\\\
                      ////\\\\
                     /////\\\\\
                    //////\\\\\\
                   ///////\\\\\\\
                  ////////\\\\\\\\     =====I=====
          _/\_   /////////\\\\\\\\\     ====I====
          \||/  //////////\\\\\\\\\\     ===I===
           ||  ///////////\\\\\\\\\\\     ==I==
           || ////////////\\\\\\\\\\\\     =I=
           ||/////////////\\\\\\\\\\\\\     I
           ||/////////////\\\\\\\\\\\\\\    I
           |//////////////\\\\\\\\\\\\\\\   I
          ////////////////\\\\\\\\\\\\\\\\  I
         /////////////////\\\\\\\\\\\\\\\\\ I
        //////////////////\\\\\\\\\\\\\\\\\\I               *
       ///////////////////\\\\\\\\\\\\\\\\\\\              ***
      /[][][][][][][][][][][][][][][][][][][]\            *****
       []                                  []            *******
       []                                  []             *****
       []               5421               []            *******
       []           |==========|           []           *********
       []      +    ||||||||||||    +      []          ***********
       []     +++   ||||||||||||   +++     []            *******
       []    +++++  ||||||||||||  +++++    []           *********
       []     +++   ||||||||||||   +++     []          ***********
       []      +    ||||||||||||    +      []         *************
       []           |O||||||||||           []        ***************
       []           ||||||||||||           []       *****************
       []           ||||||||||||           []      *******************
       []           ||||||||||||           []              HHH
       []           ||||||||||||           []              HHH
       []           |==========|           []              HHH
       []           ////////////           []              HHH
       [][][][][][][][][][][][][][][][][][][]              HHH
     --[][][][][][][][][][][][][][][][][][][]---------------------------
    -------------------------------------------------------------------
   -------------------------------------------------------------------
  -------------------------------------------------------------------
 -------------------------------------------------------------------
-------------------------------------------------------------------
```