Contents

1 Purpose

- Implement an inheritance hierarchy of classes in C++
- Learn about virtual functions, overriding, and polymorphism in C++
- Use two-dimensional arrays using `vector<T>`, one of the simplest container class templates in the C++ Standard Template Library (STL)
- Use modern C++ smart pointers, which automate the process of resource deallocation

2 Overview

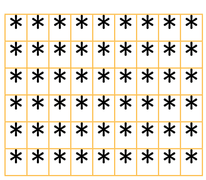
Using simple two-dimensional geometric shapes, this assignment will give you practice with the fundamental principles of object-oriented programming (OOP).

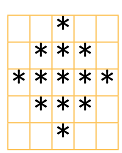
The assignment starts by abstracting the essential attributes and operations common to four geometric shapes of interest in this assignment, namely, rhombus, rectangle, and two kinds of triangle shapes.

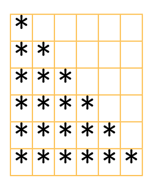
You will then be tasked to implement the shape abstractions using the C++ features that support encapsulation, information hiding, inheritance and polymorphism.

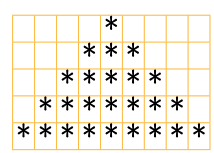
In addition to implementing the shape classes, you will be tasked to implement a `Canvas` class whose objects can be used by the shape objects to draw on.

The four geometric shapes of interest in this assignment can be textually rendered into visually identifiable images on the computer screen; for example:


Rectangle, 6×9


Rhombus, 5×5


Right Triangle, 6×6


Acute Triangle, 5×9

3 Modeling 2D Geometric Shapes

3.1 Common Attributes: Data

height the length of the vertical attribute of the shape, a positive integer

width the length of the horizontal attribute of the shape, a positive integer

name a string object; for example, "Book" for a rectangular shape

pen a character to draw the shape with

ID number a unique positive integer, distinct from that of all the other shape objects

3.2 Common Operations: Interface

1. A constructor that accepts as parameters the initial values of a shape's `height`, `width`, `name`, and `pen` data members
2. Five accessor (getter) member-functions, one for each attribute
3. Four mutator (setter) member-functions for setting the `name`, `height`, `width` and `pen` data members
4. A `toString()` member-function that returns a `std::string` representation of the `Shape` object invoking it
5. An overloaded **polymorphic** output operator `<<`
6. A member-function `areaGeo()` that computes and returns the shape's geometric area
7. A member-function `preimeterGeo()` that computes and returns the shape's geometric perimeter
8. A member-function `areaScr()` that computes and returns the shape's *screen area*, the number of characters forming the textual image of the shape
9. A member-function `preimeterScr()` that computes and returns the shape's *screen perimeter*, the number of characters on the borders of the textual image of the shape
10. A member-function 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++:

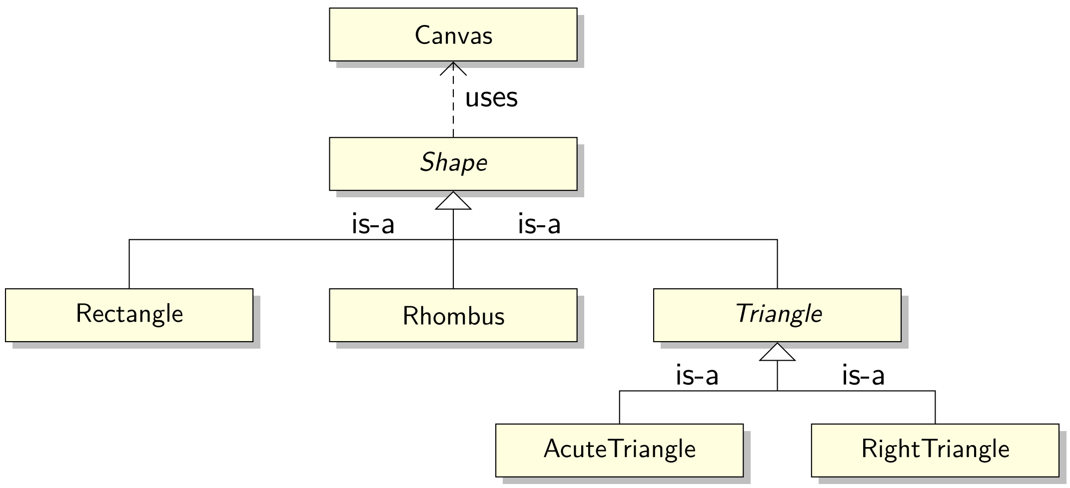


Figure 1: A UML class diagram showing an inheritance hierarchy specified by two abstract classes `Shape` and `Triangle`, and by four concrete classes `Rectangle`, `Rhombus`, `AcuteTriangle`, and `RightTriangle`.

Encapsulating the attributes and operations common to all shapes, the *`Shape`* class must necessarily be *abstract* because the shapes it models are so general that it simply would not know how to implement several of the operations specified in section 3.2.

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

As an abstract class, *`Shape`* enables polymorphism, allowing variables of types *`Shape*`* and *`Shape&`* to make polymorphic calls.

Similarly, the class *`Triangle`* must be abstract, since it has no knowledge about the shape-dependent data and operations of the shapes it generalizes.

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

5 Concrete Shapes

The specific characteristic properties of our concrete shapes are listed in the following table.

Properties	Concrete Shapes			
	Rectangle	Rhombus	Right Triangle	Acute Triangle
attributes	h, w	d	b	b
Invariants	$h \geq 1,\ w \geq 1$	d is odd, $d \geq 1$	$b \geq 1$	b is odd, $b \geq 1$
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 if Height>1 and Width>1	$2(h+w)-4$	$2(d-1)$	$3(h-1)$	$4(h-1)$
textual perimeter if Height=1 or Width=1	hw	1	1	1
Sample textual images of the concrete shapes and their dimensions, w (width) and h (height)	********* ********* ********* ********* *********	* *** ***** *** *	* ** *** **** *****	* *** ***** ******* *********
	$w=9,\ h=5$	$d=5$	$b=5,\ h=b$	$b=9,\ h=\frac{b+1}{2}$
Default name	Door	Diamond	Ladder	Wedge
Default pen character	*	*	*	*

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

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.

You may add private member functions to facilitate your operations, but you may not add data members other than those given in the attribute row of Table on page 4.

6.1 Requirements

- The unit of length is a single character; thus, all shape attributes such as height, width, base, and diagonal 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.
- The constructor of `Rhombus` must select the next integer if the supplied value for its diagonal is not odd.
- The constructor of `AcuteTriangle` must select the next integer if the supplied value for its base length is not odd.

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 overloaded output operator itself internally would call `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 height and width, 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 using `this`[1]instead of `p`. You need to include the <`typeinfo`> header for this.

Lines 12-13 show that `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.

Your C++ compiler may generate different text to indicate the static and dynamic types of a pointer. Microsoft VC++ 2022 produces a more readable output as shown below.

[1]During the call `rect.toString()`, inside `toString()`, the object `rect` is represented by the pointer `this`, which points to `rect`.

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

```
Shape Information
-----------------
id:                   1
Shape name:           Rectangle
Pen character:        *
Height:               5
Width:                7
Textual area:         35
Geometric area:       35.00
Textual perimeter:    20
Geometric perimeter:  24.00
Static type:          class Shape const * __ptr64
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;
```

```
Shape Information
-----------------
id:                   2
Shape name:           Ace of diamond
Pen character:        v
Height:               17
Width:                17
Textual area:         145
Geometric area:       144.50
Textual perimeter:    32
Geometric perimeter:  48.08
Static type:          class Shape const * __ptr64
Dynamic type:         class Rhombus
```

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

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

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

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

```
Shape Information
-----------------
id:                  3
Shape name:          Wedge
Pen character:       *
Height:              9
Width:               17
Textual area:        81
Geometric area:      76.50
Textual perimeter:   32
Geometric perimeter: 41.76
Static type:         class Shape const * __ptr64
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;
```

```
Shape Information
-----------------
id:                  4
Shape name:          Carpenter's square
Pen character:       L
Height:              10
Width:               10
Textual area:        55
Geometric area:      50.00
Textual perimeter:   27
Geometric perimeter: 34.14
Static type:         class Shape const *
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 overloaded output operator:

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

Specifically, `rt` in line 23 is bound to the parameter `shp`, which is a reference, and as such, `shp` can call virtual functions of `Shape` polymorphically; in other words, the decision as to which virtual function to run 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.areaGeo()` binds to `Rhombus::areaGeo()`, if `shp` references a `Rectangle` object, then `shp.areaGeo()` binds to `Rectangle::areaGeo()`, and so on.

Now, consider the call `rt.toString()` on line 25. Since, `Shape::toString()` is non-virtual, the call `rt.toString()` is bound at compile time (early binding). However, the object `rt` in the call `rt.toString()` is represented inside the function `Shape::toString()` through `this`, a pointer of type `Shape*`, which can in fact call virtual functions of `Shape` polymorphically.

7.2 Shape's Draw Function

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

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

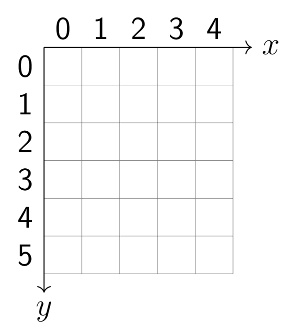
Defining a local `Canvas` object like so

```
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 the 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 rectCan{ rect.draw() };
cout << rectCan << endl;
```

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

```
Canvas aceCan{ ace.draw() }; // or, Canvas aceCan = ace.draw();
cout << aceCan << endl;
```

```
        V
       VVV
      VVVVV
     VVVVVVV
    VVVVVVVVV
   VVVVVVVVVVV
  VVVVVVVVVVVVV
 VVVVVVVVVVVVVVV
VVVVVVVVVVVVVVVVV
 VVVVVVVVVVVVVVV
  VVVVVVVVVVVVV
   VVVVVVVVVVV
    VVVVVVVVV
     VVVVVVV
      VVVVV
       VVV
        V
```

```
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
```

7.4 Flipping Canvas Objects

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;
```

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

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

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

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

        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
```

7.5 Using Smart Pointers to `Shape` objects

Now, let's create a vector of smart pointers pointing to concrete shape objects and draw them polymorphically:

```

// create a vector of smart pointers 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
for (const auto& shp : shapeVec)
{
   cout << shp->draw() << endl;
}
// referncing a unique_ptr object that point to a
// concrete shape object, shp behaves like a pointer,
// calling the virtual function draw() polymorphically
```

Notice the absence of the operators `new` and `delete` in the code above.

8 Task 2 of 2

Implement a Canvas class using the following declaration. Feel free to introduce other private member functions, but no data members, of your choice to facilitate the operations of the other members of the class.

```
class Canvas {
public:
   // all special members are defaulted because 'grid',
   // a 2D std::vector, is self-sufficient and efficient,
   // designed to handle the corresponding special 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{};            // a 2D vector representing a canvas
   char fillChar{ ' ' };                    // fill or clear character
   bool check(int r, int c)const;           // validates row r and column c, 0-based
   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 = ' ');

   char getFillChar()const;
   void setFillChar(char ch);

   int getRows()const;                  // returns height of this Canvas object
   int getColumns()const;               // returns width of this Canvas object
   Canvas flip_horizontal()const;       // flips this canvas horizontally
   Canvas flip_vertical()const;         // flips this canvas vertically
   void print(ostream&) const;          // prints this Canvas to ostream
   char get(int r, int c) const;        // returns char at row r and column c, 0-based;
                                        // throws std::out_of_range{ "Canvas index out of range" }
                                        // if r or c is invalid.
   void put(int r, int c, char ch);     // puts ch at row r and column c, 0-based;
                                        // the only function used by a shape's draw functon;
                                        // throws std::out_of_range{ "Canvas index out of range" }
                                        // if r or c is invalid.

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

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

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 Specific Grading scheme

Task 1: 60% The Shape classes

Task 2: 40% The `Canvas` class

10 General Grading scheme

Functionality	• Correctness of execution of your program • Proper implementation of all specified requirements • Efficiency	60%
OOP style	• Encapsulating only the necessary data inside 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 `memset()`, `memmove`, `memcpy`, `memcmp`, `malloc`, `alloc`, `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%

11 Sample Test Driver

11.1 ShapeTestDriver.cpp

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

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

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

void drawHouse();      // draws front view of a house image

int main()
{
   drawHouse();

   return 0;
}
```

11.2 Drawing Front View of a House

```cpp
// Using our four geometric shapes,
// draws a pattern that looks like the front view of a house
void drawHouse()
{
   // create a vector of smart pointers to Shape
   std::vector<std::unique_ptr<Shape>> shapeVec;

   // create a 47-row by 72-column Canvas
   Canvas houseCanvas(47, 72);
   houseCanvas.drawString(1, 10, "a geometric house: front view");

   shapeVec.push_back(std::make_unique<RightTriangle>(20, '\\', "Right half of roof"));
   Canvas roof_right_can = shapeVec.back()->draw();
   houseCanvas.overlap(roof_right_can, 4, 27);

   shapeVec.back()->setPen('/');
   Canvas roof_left_can = shapeVec.back()->draw().flip_horizontal();
   houseCanvas.overlap(roof_left_can, 4, 7);

   houseCanvas.drawString(23, 8,
      "[] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []");

   shapeVec.push_back(std::make_unique<Rectangle>(5, 1, '|', "left chimeny edge"));
   Canvas chimneyL = shapeVec.back()->draw();
   houseCanvas.overlap(chimneyL, 14, 12);

   shapeVec.push_back(std::make_unique<Rectangle>(4, 1, '|', "right chimeny edge"));
   Canvas chimneyR = shapeVec.back()->draw();
   houseCanvas.overlap(chimneyR, 14, 13);

   shapeVec.push_back(std::make_unique<Rectangle>(11, 1, 'I', "antenna stem"));
   Canvas antenna_stem = shapeVec.back()->draw();
   houseCanvas.overlap(antenna_stem, 11, 45);

   shapeVec.push_back(std::make_unique<RightTriangle>(5, '=', "Right antenna wing"));
   Canvas antenna_Q1 = shapeVec.back()->draw();
   Canvas antenna_Q2 = antenna_Q1.flip_horizontal();
   Canvas antenna_Q3 = antenna_Q2.flip_vertical();
   Canvas antenna_Q4 = antenna_Q1.flip_vertical();
   houseCanvas.overlap(antenna_Q3, 11, 40);
   houseCanvas.overlap(antenna_Q4, 11, 46);

   shapeVec.push_back(std::make_unique<Rectangle>(18, 1, '[', "vertical left and right brackets"));
   Canvas wall = shapeVec.back()->draw();
   houseCanvas.overlap(wall, 24, 8);
   houseCanvas.overlap(wall, 24, 44);

   shapeVec.back()->setPen(']'); // use the same wall shape
```

```
houseCanvas.overlap(wall, 24, 9);
houseCanvas.overlap(wall, 24, 45);

shapeVec.push_back(std::make_unique<Rectangle>(1, 66, '-', "horizontal lines depicting the ground"))
Canvas line = shapeVec.back()->draw();
for (int c = 1; c <= 6; c++)
{
   houseCanvas.overlap(line, 40 + c, 7 - c);
}
houseCanvas.drawString(40, 8,
   "[] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []");
houseCanvas.drawString(41, 8,
   "[] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] [] []");

shapeVec.push_back(std::make_unique<Rectangle>(1, 12, '/', "door step"));
Canvas door_step = shapeVec.back()->draw();
houseCanvas.overlap(door_step, 39, 21);

shapeVec.push_back(std::make_unique<Rectangle>(12, 12, '|', "door"));
Canvas door = shapeVec.back()->draw();
houseCanvas.overlap(door, 27, 21);

shapeVec.push_back(std::make_unique<Rectangle>(1, 10, '=', "door top/bottom edge"));
Canvas door_edge = shapeVec.back()->draw();
houseCanvas.overlap(door_edge, 27, 22);
houseCanvas.overlap(door_edge, 38, 22);

shapeVec.push_back(std::make_unique<Rectangle>(1, 1, 'O', "door knob"));
Canvas door_knob = shapeVec.back()->draw();
houseCanvas.overlap(door_knob, 33, 22);

houseCanvas.drawString(26, 25, "5421");

shapeVec.push_back(std::make_unique<Rhombus>(5, '+', "left window"));
Canvas window = shapeVec.back()->draw();
houseCanvas.overlap(window, 28, 14);
houseCanvas.overlap(window, 28, 35);

shapeVec.push_back(std::make_unique<Rectangle>(5, 3, 'H', "tree trunk"));
Canvas tree_trunk = shapeVec.back()->draw();
houseCanvas.overlap(tree_trunk, 36, 60);

shapeVec.push_back(std::make_unique<AcuteTriangle>(7, '*', "top level leaves"));
Canvas leaves = shapeVec.back()->draw();
houseCanvas.overlap(leaves, 21, 58);

shapeVec.push_back(std::make_unique<AcuteTriangle>(11, '*', "middle level leaves"));
Canvas middleLeaves = shapeVec.back()->draw();
houseCanvas.overlap(middleLeaves, 23, 56);
```

```
119     shapeVec.push_back(std::make_unique<AcuteTriangle>(19, '*', "bottom level leaves"));
120     Canvas bottomLeaves = shapeVec.back()->draw();
121     houseCanvas.overlap(bottomLeaves, 26, 52);
122 
123     houseCanvas.drawString(13, 11, "\\||/");
124     houseCanvas.drawString(12, 11, "_/\\_");
125 
126     // finally, reveal the house image
127     cout << houseCanvas;
128 
129     // print the string representation of each shape
130     for (const auto& shp : shapeVec)
131     {
132         cout << *shp << endl;
133     }
134 
135     return;
136 }
```

11.3 Output

For the sake of brevity, the string representation of the shape objects printed on line 135 are not shown.

Output

```
         a geometric house: front view


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