Contents

1 Purpose

- Practice fundamental object-oriented programming (OOP) concepts
- 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

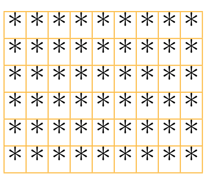
Using simple two-dimensional geometric shapes, this assignment will give you practice with the fundamental principles of object-oriented programming (OOP).

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.

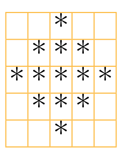
You will then be tasked to implement the shape abstractions using the `C++` features that support encapsulation, information hiding, inheritance and polymorphism.

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.

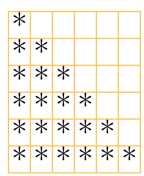
The four geometric shapes of interest in this assignment can be textually rendered into visually identifiable images on the computer screen; for example:



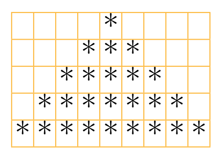
Rectangle, 6×9



Rhombus, 5×5



Right Triangle, 6×6



Acute Triangle, 5×9

3 Modeling 2D Geometric Shapes

3.1 Common Attributes: Data

- a ***name***, a string object; for example, "Book" for a rectangular shape
- a ***pen*** character, a character to draw the shape with
- an ***identity 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 `pen` and `name`
2. Three accessor (getter) member-functions, one for each attribute
3. Two mutator (setter) member-functions for setting the `name` and `pen` data members
4. A `toString()` member-function that returns a `std::string` representation of the `Shape` object invoking it
5. An overloaded output operator `<<`, which invokes the `toString()` function above polymorphically and then outputs the `string` object it returns to a supplied output stream.
6. Two accessor member-functions `getHeight()` and `getWidth()`, each returning a non-negative integer, measuring the height and width of the shape's bounding box

   Note: The bounding box of a shape is essentially a rectangle that encloses or surrounds the shape. For example, the lengths of the height and base of a triangle shape represent the height and width of the triangle's bounding box, and the lengths of the vertical and horizontal diagonals of a rhombus shape represent the height and width of the rhombus's bounding box, respectively.
7. A member-function `areaGeo()` that computes and returns the shape's geometric area
8. A member-function `preimeterGeo()` that computes and returns the shape's geometric perimeter
9. A member-function `areaScr()` that computes and returns the shape's *screen area*, the number of characters forming the textual image of the shape
10. 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
11. 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++`:

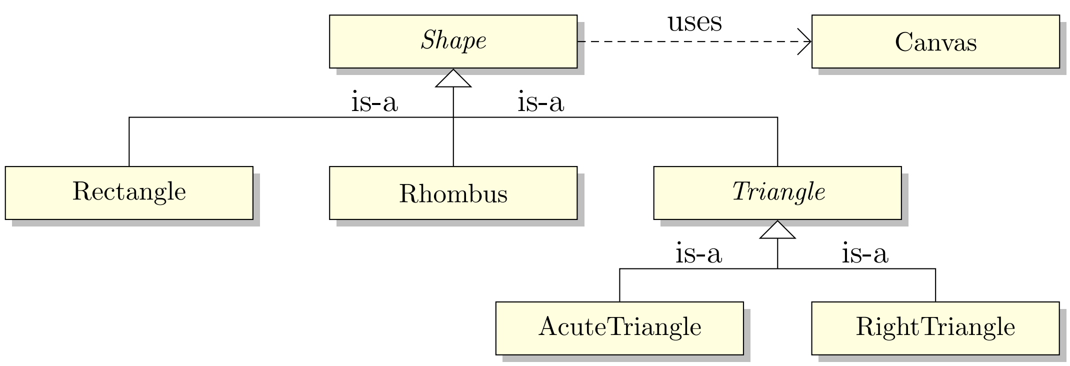


Figure 1: A UML class diagram showing an inheritance hierarchy specified by two abstract classes *`Shape`* and *`Triangle`*, and by four concrete classes `Rectangle`, `Rhombus`, `AcuteTriangle`, and `RightTriangle`.

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 the operations 6 through 11 specified in section 3.2.

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&`*.

Similarly, class *`Triangle`* must be abstract, since it has 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.

4.1 FYI

- 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 such as `Shape`, but you can define variables of types `Shape*` and `Shape&`. The compiler ensures that all calls to a virtual function (pure or not) via `Shape*` and `Shape&` are polymorphic calls.

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

- 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 `pShp` as in `Shape* pShp;` is `Shape*`, a pointer to `Shape`, a type that cannot be changed, in the sense that `pShp` will always remain a pointer to `Shape`.

  dynamic type refers to the type of the object the pointer points to (or the reference binds to) at runtime and thus can change during runtime. For example, even though `pShp` is of the type `Shape*`, it may point to different shape objects during its lifespan.

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
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 the concrete 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, 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.

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

Define all shape-independent attributes (such as height and base) in the `Triangle` class.

6.1.2 Common Operations

Define all shape-independent operations in the `Triangle` class.

6.1.3 Type-Specific Operations

Define all shape-dependent data and operations in the specific triangle shape classes.

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.

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.

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] Pointing to `rect`, the object invoking `toString()` in line 2, the `this` pointer represents `rect` during the call `rect.toString()` inside the function `toString()`.

```
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 */
```

```
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 * __ptr64
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 * __ptr64
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 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.

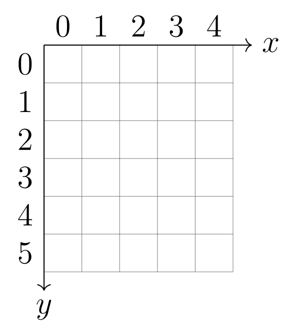
Defining a local `Canvas` object like so

```
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

```
26
27 Canvas rectCan{ rect.draw() };
28 cout << rectCan << endl;
```

```
53 *******
54 *******
55 *******
56 *******
57 *******
```

```
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
```

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:

```cpp

// 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.

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

        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.

```cpp
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, 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 = ' ');

   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-blank 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);
```

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 using 1-based indexing
- 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% The Shape classes

Task 2: 30% The `Canvas` 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 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, 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 outpu • 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 shape_examples(); // the examples shown in the assignment description
void drawHouse();      // draw front view of a house
// a helper function
void drawHouseElement(Canvas& house_canvas, const Shape& shp, int row, int col);

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

10.2 Preparing to Make Polymorphic Calls

Looking at the code of the function `drawHouse()`, you will notice that it contains many concrete shape objects but no pointers or references to any shape objects. Although the concrete shape objects could each be used directly to output their respective textual image and their string representations, it would at least double the size of the code in `drawHouse()`.

To reduce repetitive code in `drawHouse()`, we define the following function which draws a given shape together with its string representation polymorphically on a given canvas.

```

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);
}
```

Notice that the `shp` parameter is a reference of type `Shape&`, enabling `shp` to handle calls to virtual member functions of `Shape` polymorphically. If the `shp` parameter were to be a pointer, we would use a smart pointer type such as `unique_ptr<Shape>`.

10.3 Drawing Front View of a House

```

// Using our four geometric shapes,
// draws a pattern that looks like the front view of a house
void drawHouse()
{
   // create a 47-row by 72-column Canvas
   Canvas houseCanvas(47, 72);
   houseCanvas.drawString(1, 10, "a geometric house: front view");

   RightTriangle roof(20, '\\', "Right half of roof");
   Canvas roof_right_can = roof.draw();
   houseCanvas.overlap(roof_right_can, 4, 27);

   roof.setPen('/');
   Canvas roof_left_can = roof.draw().flip_horizontal();
   houseCanvas.overlap(roof_left_can, 4, 7);

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

   Rectangle chimneyL(5, 1, '|', "left chimeny edge");
   drawHouseElement(houseCanvas, chimneyL, 14, 12);

   Rectangle chimneyR(4, 1, '|', "right chimeny edge");
   drawHouseElement(houseCanvas, chimneyR, 14, 13);

   Rectangle antenna_stem(11, 1, 'I', "antenna stem");
   drawHouseElement(houseCanvas, antenna_stem, 11, 45);

   RightTriangle antenna(5, '=', "Right antenna wing");
   Canvas antenna_Q1 = antenna.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);

   Rectangle wall(18, 1, '[', "vertical left and right brackets");
   drawHouseElement(houseCanvas, wall, 24, 8);
   drawHouseElement(houseCanvas, wall, 24, 44);

   wall.setPen(']'); // use the same wall shape
   drawHouseElement(houseCanvas, wall, 24, 9);
   drawHouseElement(houseCanvas, wall, 24, 45);
```

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

   Rectangle door_step(1, 12, '/', "door step");
   drawHouseElement(houseCanvas, door_step, 39, 21);

   Rectangle door(12, 12, '|', "door");
   drawHouseElement(houseCanvas, door, 27, 21);

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

   Rectangle door_knob(1, 1, 'O', "door knob");
   drawHouseElement(houseCanvas, door_knob, 33, 22);

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

   Rhombus window(5, '+', "left window");
   drawHouseElement(houseCanvas, window, 28, 14);
   drawHouseElement(houseCanvas, window, 28, 35);

   Rectangle tree_trunk(5, 3, 'H', "tree trunk");
   drawHouseElement(houseCanvas, tree_trunk, 36, 60);

   AcuteTriangle leaves(7, '*', "top level leaves");
   drawHouseElement(houseCanvas, leaves, 21, 58);

   AcuteTriangle middleLeaves(11, '*', "middle level leaves");
   drawHouseElement(houseCanvas, middleLeaves, 23, 56);

   AcuteTriangle bottomLeaves(19, '*', "bottom level leaves");
   drawHouseElement(houseCanvas, bottomLeaves, 26, 52);

   houseCanvas.drawString(13, 11, "\\||/");
   houseCanvas.drawString(12, 11, "_/\\_");

   // finally, reveal the house image
   cout << houseCanvas;
   return;
}
```

10.4 Output

For the sake of brevity, the string representation of the shape objects printed on line 28 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
      --[][][][][][][][][][][][][][][][][][][]--------------------------
     ------------------------------------------------------------------
    ------------------------------------------------------------------
   ------------------------------------------------------------------
  ------------------------------------------------------------------
 ------------------------------------------------------------------
```

10.5 Example Code

The following function contains the sample code used in the description of this assignment. You can use it during the development of your program; otherwise, you may ignore it altogether.

```

void shape_examples()
{
   Rectangle rect{ 5, 7 };
   cout << rect.toString() << endl;
   // or equivalently
   //cout << rect << endl;
   //-----------------------------------------------------------
   Rhombus
      ace{ 16, 'v', "Ace of diamond" };
   // cout << ace.toString() << endl;
   // or, equivalently:
   cout << ace << endl;
   //-----------------------------------------------------------
   AcuteTriangle at{ 17 };
   cout << at << endl;

   /*
     // equivalently:

     Shape *atPtr = &at;
     cout << *atPtr << endl;

     Shape &atRef = at;
     cout << atRef << endl;
  */
  //-----------------------------------------------------------
   RightTriangle
      rt{ 10, 'L', "Carpenter's square" };
   cout << rt << endl;
   // or equivalently
   // cout << rt.toString() << endl;
   //-----------------------------------------------------------
   Canvas aceCan = ace.draw();
   cout << aceCan << endl;
   //-----------------------------------------------------------
   Canvas rectCan = rect.draw();
   cout << rectCan << endl;
   //-----------------------------------------------------------
   at.setPen('^');
   Canvas atCan = at.draw();
   cout << atCan << endl;
   //-----------------------------------------------------------
   Canvas rtCan = rt.draw();
   cout << rtCan << endl;
   //-----------------------------------------------------------
```

```cpp
    rt.setPen('O');
    Canvas rtQuadrant_1 = rt.draw();
    cout << rtQuadrant_1 << endl;
    //------------------------------------------------------------
    Canvas rtQuadrant_2 = rtQuadrant_1.flip_horizontal();
    cout << rtQuadrant_2 << endl;
    //------------------------------------------------------------
    Canvas rtQuadrant_3 = rtQuadrant_2.flip_vertical();
    cout << rtQuadrant_3 << endl;
    //------------------------------------------------------------
    Canvas rtQuadrant_4 = rtQuadrant_3.flip_horizontal();
    cout << rtQuadrant_4 << endl;
    //------------------------------------------------------------
    // 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;
    //------------------------------------------------------------
}
```