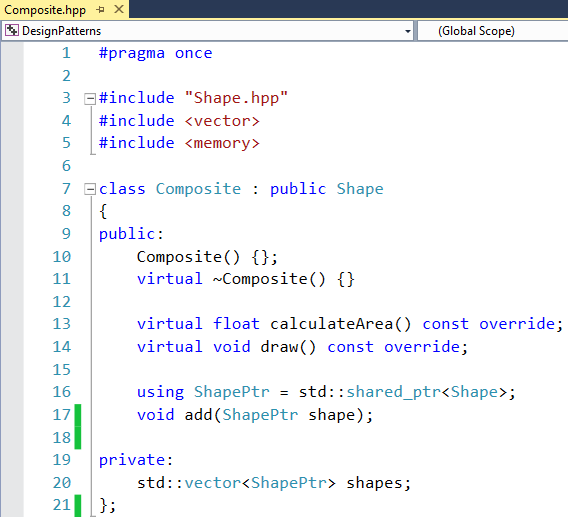
Composite Pattern

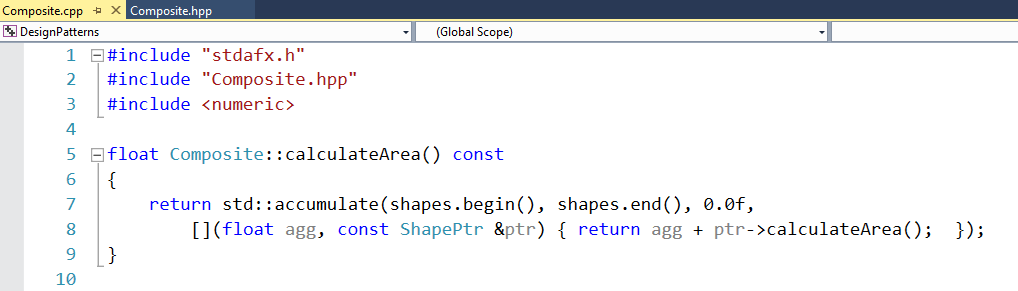
In this assignment we will refactor the root scene, a collection of shapes, into a composite structure. The composite is a commonly used design pattern that represents a collection of other elements and an operation performed on the composite is forwarded to the individual shapes depending on the operation. As with other shapes, a Composite also derives from the base class that other elements derive from (in our example Shape) and provides the same operations.

# Add the Composite class

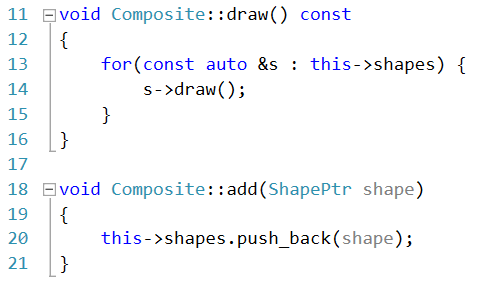
Add two new files, Composite.hpp and Composite.cpp. The composite adds a collection holding the element shapes and methods to modify the collection (add(ShapePtr shape)) in this case:



The implementation uses an STL algorithm (from the <numeric> header) to sum up the areas of the individual shapes:

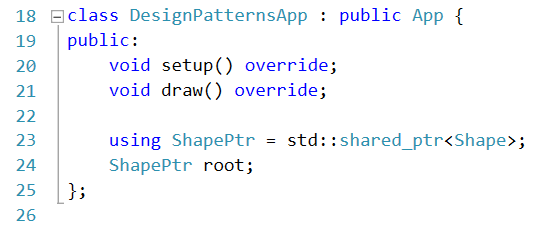


The add(ShapePtr shape) method simply pushes the shape onto the end of the shapes collection and the draw() method iterates all elements to call as was done before in the application code for iterating all element of the scene:

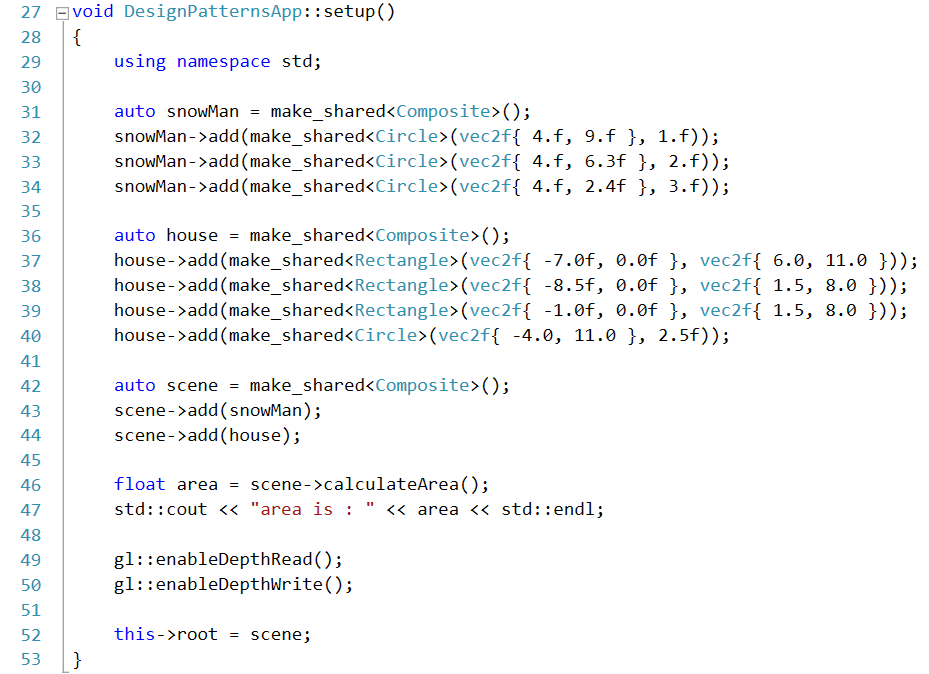


# Use the Composite Class

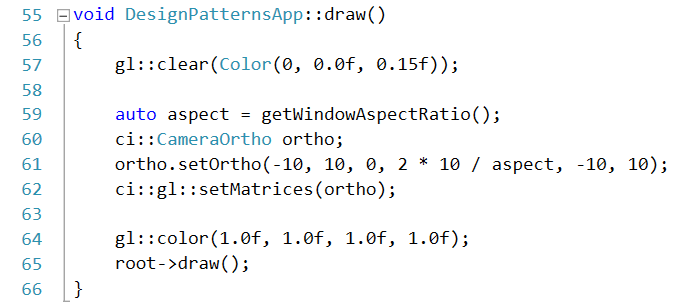
We can now modify the application to have an abstract ShapePtr as the root pointer to the scene. In the setup method we can now construct a more complex scene by compositing multiple sub-shapes to a parenting scene shape. Note that, because Composite is inheriting from Shape we created a recursive data structure which allows many desirable nesting features:



The setup method is now creates a more complex scene involving a snowman and a structure, somewhat resembling an oriental building:



The draw method is now simpler, as the loop for drawing all scene elements has been moved to the composite shape:



Compiling and starting the application reveals the scene:

