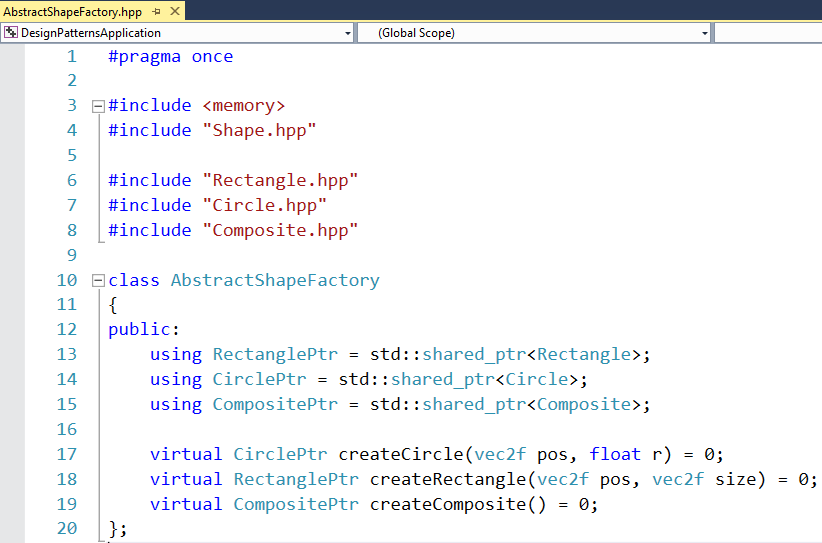
Factory Pattern

In the previous assignment, we introduced the builder pattern to encapsulate the creation of complex object graphs, in our example, a scene. In this example, we will encapsulate the creation of the individual parts of the graph. The reason for this is that we want to be able to switch types used by the builders without having to change any of the builder code itself. We therefore introduce a new type hierarchy with a base class called AbstractShapeFactory that contains 3 methods, each one for creating a different shape type (Circle, Rectangle and Composite):

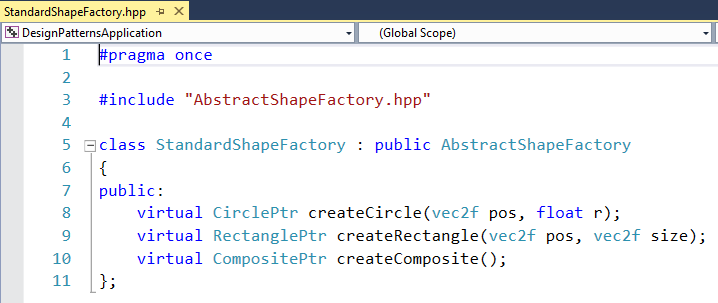
# Introduce AbstractShapeFactory

Add a new header, AbstractShapeFactory.hpp, containing the abstract functions for creating individual shapes:



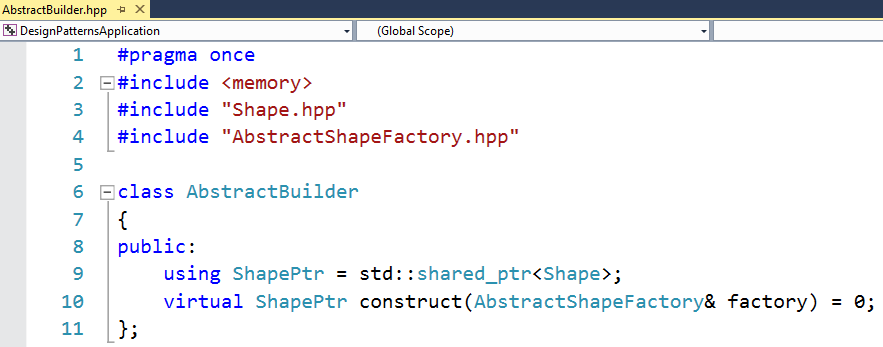
# StandardShapeFactory

The shape factory is used to construct new shapes. The default implementation will do just that, create the standard shapes:

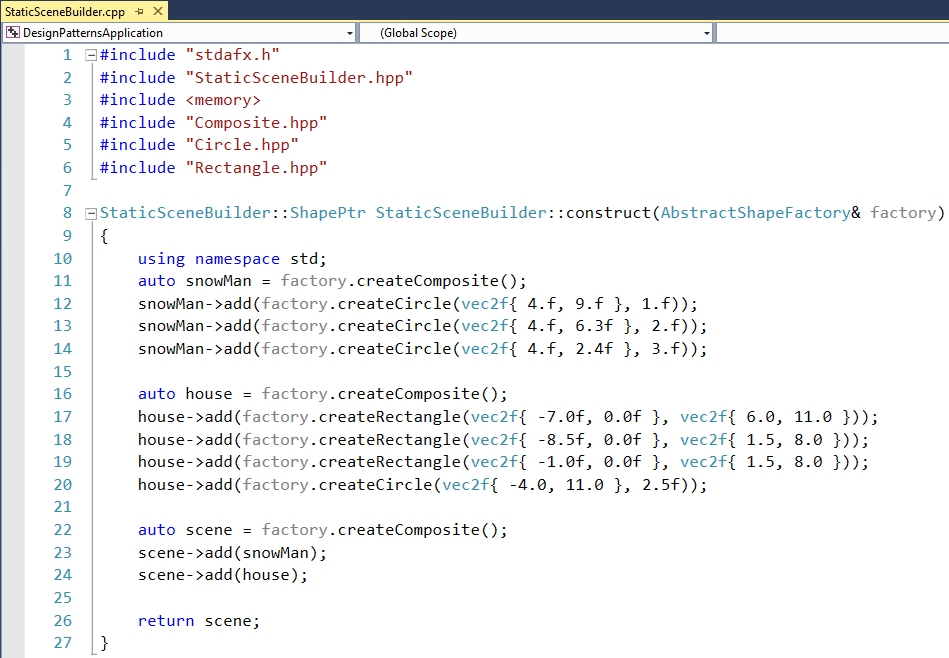


# Employing the Factory

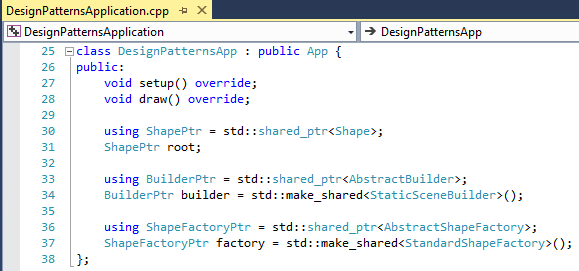
To use the new shape factory machinery we have to change the Builder interface to accept a reference to a factory that produces the individual parts it uses to construct a scene:

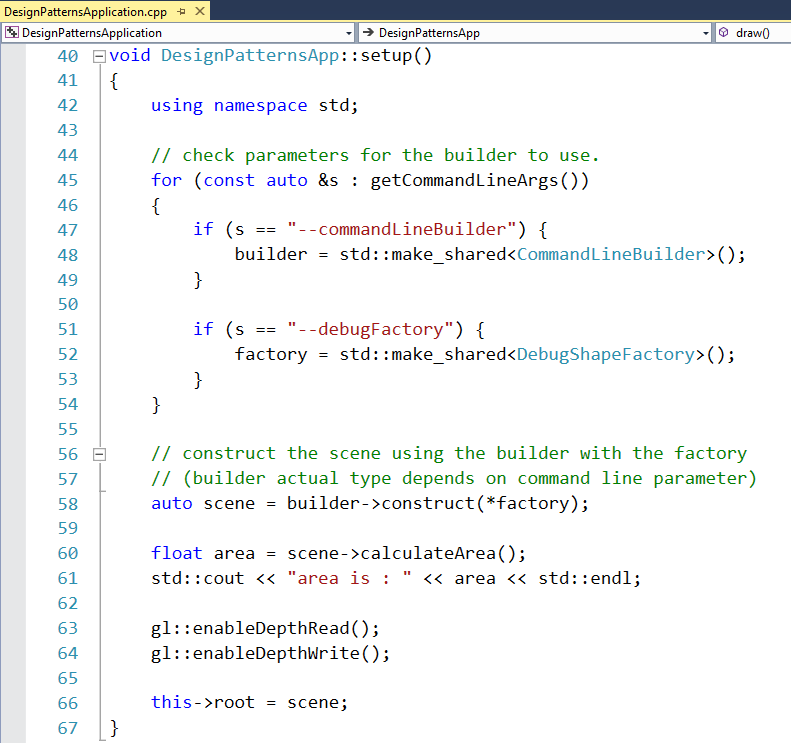


Accordingly, the StaticSceneBuilder and CommandlineBuilder implementations will have to be changed to forward object instantiation to the factory:



The application code will have to pass a factory to the builder. This enables us to switch the factory type depending on a commandline parameter:





We can now add the DebugShapeFactory. This factory creates instances of derived classes from Circle, Rectangle and Composite respectively. In this example, the debug variations print extra debugging information in the calculateArea method before after calling their base class implementation:

