EECS402, Winter 2016, Project 5

Overview:

In sticking with this semester's theme of image-based projects, out last project will again involve working with images. For this project, though, you will be developing a set of classes for generating a "map" of a very simple city, given a description of the city.

You will be making use of inheritance and polymorphism such that different city entities will be drawn on the map differently, depending on the type of entity they represent. This means there will be a common interface for adding a city entity to a map, but the implementation for each entity will be different.

Due Date and Submitting:

This project is due on **Monday, April 18 at 4:30pm**. Early submissions are allowed with corresponding bonus points, according to the policy described in detail in the course syllabus. However, since the project is due on the last day of class, ***late submissions are not allowed*** for this project. No submissions will be accepted after Monday April 18 at 4:30.

For this project, your solution must be organized into multiple files, where each class definition is in its own header file (extension ".h"), and each class' member functions are defined in a source file with a ".cpp" extension. A valid and functional Makefile, with a "clean" target is also required.

Details:

Much like one of our earlier projects, you will be developing a framework of classes to support building a map of a city – you will *not* be writing an "interactive application". Given the short turnaround time for this project, I am trying to keep the amount of work needed minimal, while still conveying the use and benefits of both inheritance and polymorphism. Templates are not used for this project, so no .inl files will be needed or allowed.

Class Hierarchy:

At the highest level of the class hierarchy will be a class named "MapEntityClass". This class is purposely generically named such that it will serve as a base class for more specifically typed classes. We will put any attributes that apply to all map entities at this level so they are automatically obtained by all subclasses. Attributes that apply only to a subset of specific map entities will not belong to MapEntityClass. In addition to the common attributes at the MapEntityClass level, we will also put common functionality at the MapEntityClass level. The implementations of some of those functions that are common, though, will be specific to the subclasses. Therefore, we'll make them virtual to allow the specific implementation to be utilized via the common interface at the MapEntityClass level. An example of this common-interface-but-different-implementation-method is the method to draw a map entity onto a map. We need to be able to simply say "draw this particular map entity on the map", since, as you'll see below, different entities are drawn in different ways, but all of them need to be able to be drawn with an interface that is the same, and can be called at the MapEntityClass level as opposed to at the lowest subclass level.

Our simplified city maps will consist only of houses, roads, schools, and parks. The specific entities are drawn as follows:

- HouseClass: Houses are drawn as full red (RGB 255, 0, 0) squares, where the size of the square is dependent on the value of the house – that is, houses that are worth more are drawn as larger squares than houses that are worth less. Houses are always square, and the dimension of the house on the map will be a minimum of one pixel in size, with one additional pixel for each $15,000 of value (that is, houses with values from $0 to $14,999 will be one pixel, houses from $15,000 to $29,999 will be 2x2, houses from $30,000 to $44,999 will be 3x3, etc.).
- SchoolClass: Schools are drawn as light-ish blue (RGB 63, 63, 255) squares, and their size depends on the number of students in the school. Schools are always square, and the dimension of the school on the map will be a minimum of one pixel in size, with one additional pixel for each 50 students enrolled in the school (that is, schools with 0 to 49 students will be one pixel, schools with 50 to 99 students will be two pixels, etc.).
- RoadClass: Roads are drawn as half-gray (RGB 127, 127, 127) rectangles, and either run vertically or horizontally. In our simple cities, there are only straight roads, with no curves or diagonal roads, etc. Roads are rectangular in shape, and the length of the road will be specified directly, but the width of the road is based on the number of lanes the road has. Each lane will result in 3 pixels of width for the road, so a 4 lane road with a length of 100 will be displayed with a width of 12 pixels and a length of 100.
- ParkClass: Parks are drawn as full green (RGB 0, 255, 0) circles and the size of the circle is determined based on the number of acres the park consists of. Parks are drawn differently than the other classes described above, which are all rectangular in shape. The park's circle is centered at the park's specified location, and the circle's radius will be determined by having a minimum radius of one pixel with one additional pixel of radius for every 25 acres of land the park consists of (that is, a park consisting of 0 to 24 acres will have a radius of 1 pixel, while a park of 25 to 49 acres will have a radius of 2 pixels, etc.).

Interestingly, we have noticed that houses, schools, and roads are all drawn as rectangles (well, technically, houses and schools are squares, but that's just a special case of a rectangle, so…) Therefore, rather than duplicate drawing code in all of these classes, we will introduce a mid-level class named "RectangularEntityClass". Since this class represents map entities, it will inherit from MapEntityClass. Note that it would probably be a good idea to generate a "CircularEntityClass" as well, in case we want to add more specific entity types that will be drawn as circles in the future. For this project, though, we will assume that the ParkClass will be the only circular entity expected, so the CircularEntityClass will not be used.

Figure 1 shows a visual representation of the class hierarchy for this project and some of the important attributes and methods of each class. This diagram shows highlights of the classes, and is not meant to fully define the classes shown, but the attributes and methods shown must be implemented as members of the class as shown in the diagram, so while this diagram is not necessarily complete, you must implement your solution according to what is described in the diagram.

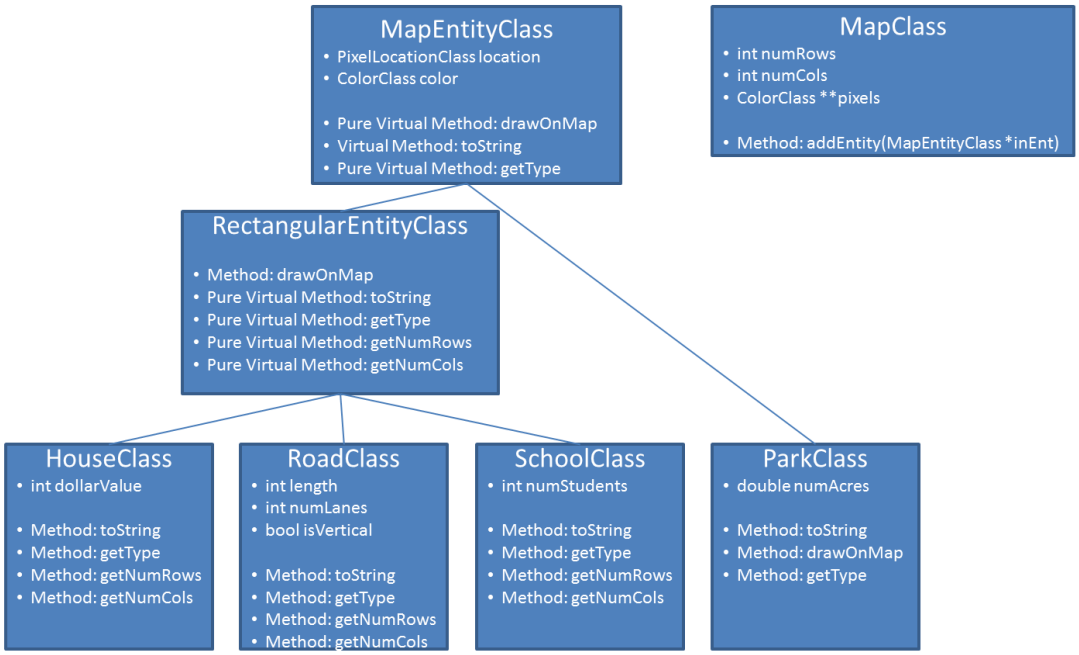

Figure 1: Class hierarchy with important attributes and methods listed

Some Descriptions:

The "toString" method simply returns a string representation of the object – that is the object's attributes are "printed" to a string in a readable way and the string is returned. Since the method is virtual, you'll be able to call is on a MapEntityClass pointer and it is expected to return a string describing the low-level class object. To avoid duplicating code, the higher-level classes should be responsible for adding the attributes at that level to the string.

The "getType" method just returns a string that indicates the type of the class. This method is "pure virtual" and will only be implemented at the lowest level classes that will be instantiated to actual objects. There's no need to store the type as an attribute, because the existence of this method will allow you to obtain the string without having to store it as a per-object attribute.

The "drawOnMap" method draws a map entity onto a map. For this project, out-of-bounds cases will be handled as follows: Every time an out-of-bounds pixel is attempted to be drawn to, a descriptive error message is printed, but the drawing process continues such that all "in-bounds" pixels are drawn even is part of the entity ends up being out-of-bounds. However, if, after drawing as much of the entity as possible, any out-of-bounds pixels were reached, then the drawOnMap method will throw an exception so the caller can know this happened and can handle the exception in whatever way they want to. Since you are only developing a framework here, you're not responsible for handling the exception, but you are responsible for determining when an exception should be thrown and throwing an appropriate exception. Since all rectangular entities are drawn with the same algorithm and we don't want duplicated code, the implementation of the virtual method "drawOnMap" will be at the RectangularEntityClass-level and will *not* be in the subclasses. Note that since ParkClass is drawn differently, it has its own implementation of drawOnMap. To reiterate – you may *not* have an implementation of drawOnMap in HouseClass, RoadClass, or SchoolClass.

The RectangularEntityClass has two additional virtual methods – a "getNumRows" and "getNumCols" that will help in knowing how to draw an entity. Unlike the drawOnMap described above, these methods will have a unique implementation for each specific rectangular entity, so they will be implemented at the lowest level, but will be used (i.e. called) from within the RectangularEntityClass's drawOnMap method so that method knows how many rows and columns to draw the rectangle as.

Output:

Your framework will support two types of output – console output and imagery output. Imagery output will consist of a PPM-formatted image that consists of the city "map" with map entities that have been drawn on it. Console output will consist of output from the class methods, including detected problems that come up during the drawing process and/or via the polymorphic "toString" interface that is called on a MapEntityClass pointer but prints out specific entity type details, such as house values, park sizes, etc.

Important Restriction:

For this project, you are required to use inheritance and polymorphism to their full extent. To enforce this requirement, there is a *very important restriction* that **must** be adhered to in order to receive credit for this project.

You may not declare any variables (objects) or pointers to objects having the following types: RectangularEntityClass, HouseClass, RoadClass, SchoolClass, or ParkClass. You also may not use type casting to type cast to any of these types. You may not have any code that essentially says "if the type of this thing is a <something>, then do X otherwise do Y".

Also, you may not do any crazy "workarounds" that effectively do any of those restricted things in a different way. Let polymorphism do the work of deciding what type something is and using the type-specific implementation as needed – that's what it's for!

It is perfectly allowed (and expected/required) that MapEntityClass pointers will be declared and set directly to instantiations of the lower-level classes in the main function (provided to you). Finally, you won't be able to declare actual MapEntityClass objects (as opposed to pointers) since the MapEntityClass has pure virtual functions and is therefore an abstract class, meaning objects of that type cannot be declared (or instantiated).

Provided Stuff:

One of your colleagues started working on this project, but got pulled off onto another project, so its not finished. Luckily, he seems to have started with the most complex parts of the project, but you need to finish the project from where he left off. Specifically, you will start with the full MapEntityClass, the full MapClass, the full ParkClass, and a main function for testing. None of these provided items should need to be changed.

"Specific Specifications":

These "specific specifications" are meant to state whether or not something is allowed. A "no" means you definitely may NOT use that item. In general, you can assume that you should not be using anything that has not yet been covered in lecture (as of the first posting of the project).

- Use of Goto: No
- Global Variables / Objects: No
- Global Functions: Yes, if needed
- Use of Friend Functions / Classes: No

- Use of Structs: No
- Use of Classes: Yes
- Public Data In Classes: **No** (all data members must be private)
- Use of Inheritance / Polymorphism: Yes, required
- Use of Arrays: Yes
- Use of C++ "string" Type: Yes
- Use of C-Strings: No
- Use of Pointers: Yes
- Use of STL Containers: Yes
- Use of Makefile / User-Defined Header Files / Multiple Source Code Files: Yes, required
- Use of exit(): No
- Use of overloaded operators: Yes
- Use of float type: **No** (That is, all floating point values should be type double, not float)