TASK

Deadline: Dec 15th,2021

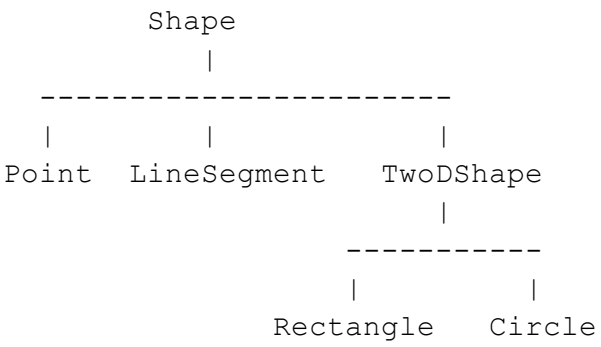
This task is from inheritance and virtual functions, operator overloading, exceptions, STL and smart pointers.

The Problem

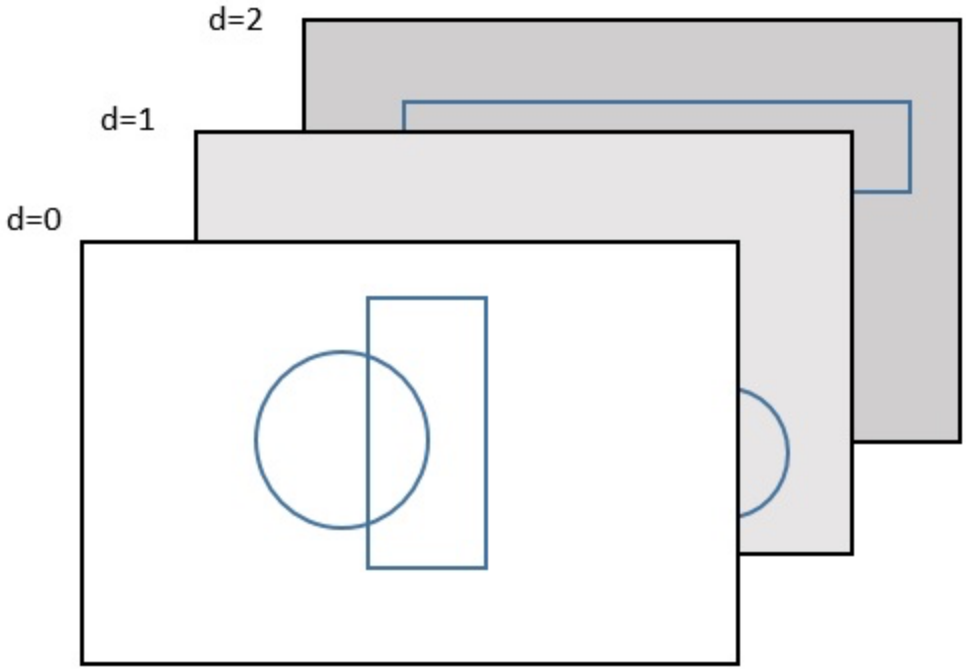
You will complete the code for a number of C++ classes that model a number of (up to) two-dimensional objects in the standard x-y coordinate system (see the first picture in this), together with a number of transformations that can be applied to them, and some other operations.

Overview and the class hierarchy

A number of different classes are used to model different geometric shapes, all derived from a common base class called `Shape`. The class hierarchy is as shown in this diagram:



In addition to attributes specific to each type of objects, all objects have a "depth". The *depth* of an object is a non-negative integer indicating which "layer" or "plane" it is in: imagine that there are multiple "planes" in the coordinate system, or like "layers" in some photo-editing software; see figure below.



Constructors of Shape and its subclasses

The following explains each of the classes and their constructors:

- `Shape(int d)`:
  This is the base class and should be abstract (no object of this class should be constructed). Nevertheless, it has a constructor with a parameter d specifying the depth of the object. If d is negative, the constructor should throw a `std::invalid_argument` exception.
- `Point(float x, float y, int d=0)`:
  This class models a point, which is a zero-dimensional object specified by its x-coordinate, y-coordinate and its depth. The =0 here indicates that d is a *default argument*, namely that it will by default set to 0 if it is not supplied when the constructor is invoked. (See e.g. https://www.geeksforgeeks.org/default-arguments-c/ for some explanation.) You don't need to do anything about it (and it has nothing to do with pure virtual functions!)
- `LineSegment(const Point& p, const Point& q)`:
  This class models a line segment, which is a one-dimensional object, the portion of a straight line passing between the two points. Here we model

"axis-aligned" line segments only, i.e. the line segment must be parallel to either the x-axis or the y-axis. The constructor specifies the two endpoints of the line segment. The two points p and q are not necessarily given in any order; the line segment has no "direction". The two endpoints should have the same depth, and the resulting line segment has a depth equal to that of its endpoints. If the two endpoints have different depths, or if both their x- and y-coordinates are different (line not horizontal/vertical), or if both their x- and y-coordinates are the same (the two endpoints coincide), the constructor should throw a `std::invalid_argument` exception.

- `TwoDShape(int d)`:
  The class models any two-dimensional object, and should be an abstract class. The parameter d specifies the depth of the object.
- `Rectangle(const Point& p, const Point& q)`:

  This class models a rectangle, which is a two-dimensional object specified by two **opposite** corners (not adjacent corners) p and q. Note that the two points could be either the top-left and bottom-right corners, or the top-right and bottom-left corners; and in no particular order. Only axis-aligned rectangles are allowed, which means all edges are parallel to either the x- or y-axis; thus two corners are sufficient to define a rectangle. For example, if two of the corners are (1,2) and (3,4), then the other two corners must be (3,2) and (1,4).
  The two points p and q should have the same depth which is also the depth of the rectangle. If the two given points have different depths, or have the same x-coordinate and/or y-coordinate (which means they are on the same horizontal/vertical line or are even the same point, and no rectangle can be formed), the constructor should throw a `std::invalid_argument` exception.
- `Circle(const Point& c, float r)`:
  This models a circle, specified by its centre point c and its radius r. The depth of the circle is the same as that of c. If the radius is 0 or negative, the constructor should throw a `std::invalid_argument` exception.

Other functions of Shape and its subclasses

The `Shape` class (and all its subclasses) should support the following functions:

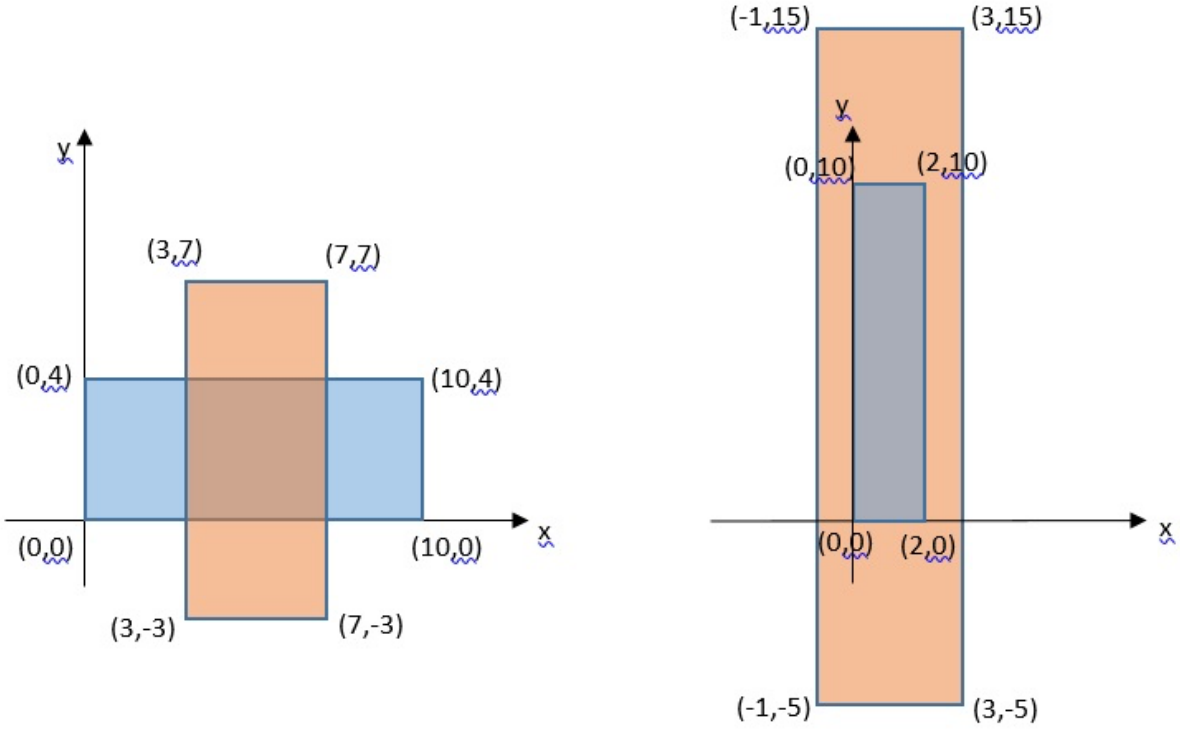
- `int getDepth()`
  `bool setDepth(int d)`
  Get/set the depth of the object. If d is negative, return false and do not update the depth.
- `int dim()`:
  Return the dimension (0, 1 or 2) of the object.
- `void translate(float x, float y)`:
  Translate, i.e. move, the whole object, to the right by a distance of x, and to the top by a distance of y. A negative x or y value means it will move to the left or the bottom, respectively.
- `void rotate()`:
  Rotate the object 90 degrees around its centre. Since all objects under consideration are "symmetric", it makes no difference whether it is rotated clockwise or anticlockwise.
  For example, for a line segment with two endpoints (0,0) and (10,0), rotating will change its endpoints to (5,5) and (5,-5); for a rectangle with four corners (0,0), (10,0), (0,4) and (10,4), rotating will change the corners to (3,-3), (7,-3), (3,7) and (7,7). Rotation has no effect (but is still a valid operation) on Point or Circle.
- `void scale(float f)`:
  Scale up/down the size of the object by a factor f, relative to its centre. A factor f > 1 indicates the object becomes bigger, and a factor 0 < f < 1 indicates the object becomes smaller. If f is zero or negative, throw a `std::invalid_argument` exception, and do not change the object. "Relative to its centre" means that the object's centre remains at the same position.
  For example, if a rectangle with four corners (0,0), (0,10), (2,0) and (2,10) is scaled up by a factor of 2, the corners become (-1,-5), (3,15), (3,-5) and (-1,15). And a circle with centre (1,2) and radius 10, scaled by a factor f = 0.5, will still have centre (1,2) but the radius becomes 5. Scaling has no effect (but is still a valid operation) on Point.
- `bool contains(const Point& p)`:

  Return true or false indicating whether the point p is "inside" the current object (let's call it o here) or not. The depths of o and p are disregarded for the purpose of this function. That is, even if o and p have different depths, this function should return true if p is "inside" o as if they had the same depth.
  If this object o is a Point, the function returns true if and only if o and p have

the same x- and y-coordinates. If o is a line segment, it returns true if and only if p lies between the two endpoints of o, including the two endpoints themselves. If o is a rectangle or circle, it returns true if and only if p is on the inside or the boundary of the rectangle/circle. Just to remind you, the distance between two points (x1,y1) and (x2,y2) is given by the formula $\sqrt{(x1-x2)^2 + (y1-y2)^2}$).

- Destructor, copy constructor and copy assignment operator are assumed to be not required (i.e. the default is adequate). If for some reason they are not adequate for your implementation then you should supply them.

The figures below illustrates how rotate and scale work on a rectangle: (left) blue rectangle rotates to become the orange one (and vice versa); (right) blue rectangle scaled up with f=2 to become the orange one (or the orange one scaled down with f=1/2 to become the blue one).



Other functions of Point

In addition to those inherited from the parent class, the Point class should support the following operations:

- `float getX()`: return the x-coordinate of the point.
- `float getY()`: return the y-coordinate of the point.

Other functions of LineSegment

In addition to those inherited from the parent class, the LineSegment class should support the following operations:

- `float getXmin()`: return the x-coordinate of the left endpoint (or both endpoints if the line is vertical).
- `float getXmax()`: return the x-coordinate of the right endpoint (or both endpoints if the line is vertical).
- `float getYmin()`: return the y-coordinate of the bottom endpoint (or both endpoints if the line is horizontal).
- `float getYmax()`: return the y-coordinate of the top endpoint (or both endpoints if the line is horizontal).
- `float length()`: return the length of this line segment.

Other functions of TwoDShape and its subclasses

In addition to those inherited from the parent class, all TwoDShape objects should support the following operation:

- `float area()`: return the area of the object. (In case you need reminding, for rectangles it is the width multiplied by the height, and for circle it is πr^2 where r is the radius and the constant PI is defined in the Shape class.)

Other functions of Rectangle

In addition to those inherited from the parent classes, the Rectangle class should support the following operations:

- `float getXmin()`: return the x-coordinate of the left edge of the rectangle.
- `float getXmax()`: return the x-coordinate of the right edge of the rectangle.
- `float getYmin()`: return the y-coordinate of the bottom edge of the rectangle.
- `float getYmax()`: return the y-coordinate of the top edge of the rectangle.

Other functions of Circle

In addition to those inherited from the parent classes, the Circle class should support the following operations:

- `float getX()`: return the x-coordinate of the centre.
- `float getY()`: return the y-coordinate of the centre.
- `float getR()`: return the radius.

The Scene class

This class stores a collection of shared pointers to some Shape objects, so that the objects can be "drawn" on the screen. It should support the following operations:

`void addObject(shared_ptr<Shape> p)`:
Add the shared pointer p that points to some Shape (or its subclasses) object to this Scene object. After the call, the pointer p (and the object that it points to) must remain "intact", and both the caller of this function and this Scene object share the "ownership" of the object being pointed to. In other words, any changes made to one of them is reflected in the other. For example, in the following code:

```
shared_ptr<Rectangle> rp = make_shared<Rectangle>(...);
Scene s;
s.addObject(rp);
rp->translate(1,2);
```

- After the rectangle pointed to by rp is translated, the rectangle added to s is translated as well.
- `void setDrawDepth(int d)`:
  Set the "drawing depth" to d, which means that when operator<< is called (see the next bullet point), it draws all objects at depth d or less. For example, if d=2, it draws all objects with depth 0, 1, 2, but not those with depth 3, 4, 5, etc. If this function is never called, operator<< should draw all objects of any depth.

`ostream& operator<<(ostream& out, const Scene& s)`:
Overloaded output stream redirection operator, to be implemented as a friend function (not a member function) of this class. It "draws" all objects to the screen, as follows. There is a rectangular "drawing area" of size defined in the constants WIDTH and HEIGHT in the Scene class. This function outputs to the stream a number of lines equal to HEIGHT, where each line has exactly WIDTH characters.

These lines are joined by the newline ('\n') character, and the final line is followed by a newline character as well. The j-th line from the **bottom**, and the i-th character from the left in this line (count starts from 0), correspond to the point with integer coordinates (i,j). That is, the rectangular area corresponds to the coordinates from (0,0) in the bottom left, to (WIDTH-1, HEIGHT-1) in the top right.
Each character in the above lines is the character `'*'`, if the corresponding integer coordinates (i,j) is inside part of any object (as defined by the `contains()` function) and that object has depth no more than the drawing depth (see previous bullet point); otherwise it is the blank space `' '` character.
Effectively, all objects of different depths (that are at most the drawing depth) are "overlayed" on top of each other. Also note that some objects may be partially or wholly outside of the drawing area, and they (or those parts) are not drawn.
For example, with the default WIDTH and HEIGHT values of 20 and 60, a scene with a point (0,0), a line segment with endpoints (0,19) and (59,19), a rectangle with opposite corners (59,0) and (55,19), and a circle with centre (30,0) and radius 10, all with depth 0, may be drawn like this if the drawing depth is also 0:

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

- (yes, I know it doesn't look like a semi-circle...) Due to the floating point nature of the coordinates, this function will have the interesting "feature" that it only draws things with integer coordinates; for example a line segment between (1,1) and (1,5) will be drawn, while one between (1.5, 1) and (1.5, 5) will not.

In all places that require comparing floating point values, you can ignore the issue of floating-point inaccuracies. That is, you can just compare two floats with x==y.

What needs to change and what cannot be changed

All the classes have no member variables at the moment; you will need to decide what protected/private data members to add to the classes. There are multiple ways to represent the various geometric objects, and there is not necessarily a "best" way. For example, a rectangle may be represented by the coordinates of its four corners, or in fact only two opposite corners are enough; or the coordinates of its centre plus its height and width; or many other ways. You will also need some data structure (preferably from STL) to store the shared pointers in the Scene class.

The classes Shape, Point, LineSegment, TwoDShape, Rectangle and Circle, as well as their inheritance relationships, have been defined for you in the Geometry.h file. However, all the required member functions are only declared in the highest class in the class hierarchy where they are relevant. You will need to decide whether some subclasses should override some of the functions, where in the hierarchy should they be implemented, and whether each function should be made virtual. In addition, all non-leaf classes should be abstract; only instances of Point, LineSegment, Rectangle and Circle should be allowed to be created. This means each non-leaf class must have some pure virtual member functions.

Therefore, you will need to decide whether each function should have the `virtual`, `override`, `final` keywords and/or the `= 0` pure specifier. You may need to duplicate some function declarations and/or their implementations (bodies) to some subclasses, or move them to some other classes (even when the comments say IMPLEMENT ME). Please see further notes in the marking criteria about this.

You must not change the existing public interface of the classes, other than in relation to inheritance, virtual-ness and pure-ness as explained above. You are allowed to add other public/protected/private member functions, should you want to.

Files Provided

- Geometry.h
- Geometry.cpp

These are the only two files you need to modify/submit. All code written by you should be in there.

- main.cpp

This is just an example that illustrates how the functions can be called.

- GeometryTester.h
- GeometryTester.cpp
- GeometryTesterMain.cpp

They are used for the execution testing part (see the next section).

- makefile

This is a makefile that will compile the main executable and the testing suite executable.

Marking Criteria and Test Suite

See Mark Distribution for the marking criteria.

To use the test suite (which is used in the execution testing part of the marking), simply type "make" in a linux terminal (with all the above files in the same folder). It will (assuming you did not break Geometry.h or Geometry.cpp) produce a GeometryTesterMain executable file. Run the program by typing

```
./GeometryTesterMain a
```

or

```
./GeometryTesterMain a b c d
```

which runs a single test case or multiple test cases respectively. We will also demonstrate its use in class.

The given Geometry.h and Geometry.cpp files (without your contributions) are already compilable with the test suite. It might even pass a few test cases. So, whatever you do, please don't break them...

The test suite may not cover all corner cases, so passing all test cases does not guarantee your program is 100% correct. Also, unfortunately we cannot isolate the testing of the various functions. For example to pass those test cases meant for `LineSegment` you need to also implement `Point` at least partially correctly.

Submission Instructions

Submit only the files Geometry.h and Geometry.cpp. Just upload them as two separate files with those exact names. DO NOT change their names including upper/lowercase, DO NOT change the extension to .txt or .cpp.txt or some such, DO NOT put them in a Word or pdf file (yes, someone actually did that), DO NOT put them in a zip archive, and DO NOT upload the entire project folder of whatever IDE you are using.

While you may want to change the main.cpp file for your own testing, it is not part of the submission. The test suites and the makefiles also should not be submitted. If you submit them, they will be ignored and I will use my own version for testing.

This is an **individual task**, and collaboration is not permitted.

Mark Distribution

- 50% of the marks go to execution of the test suite;
- 40% of the marks go to code correctness;
- 10% of the marks go to readability of the code and use of good C++ coding style.

Each are explained in a separate section below.

Execution Testing (50%)

Marks are awarded for the correct observed bahaviour of your program, judged solely by the passing (or not) of the test cases in the given test suite.

Since C++ programs may behave differently in different systems, **the departmental linux system is the definitive compilation and execution environment for the testing**. So you would want to make sure your programs can be compiled and run in the departmental linux system with the given makefile and test files. Obviously, if it doesn't compile it cannot pass any test case!

If you find that to be too much trouble, it should be possible to run the test suite on your own development environment, linux or not, by making small adaptations. You might even choose to not use the test suite at all (you are not obliged to use them, but presumably you want to know whether you pass those test cases - you will know how many marks you get there before I do).

There are 25 test cases, each worth 2 marks:

- if the test result says pass you get 2 marks;
- if you get one of error codes 1 or 2 but not both, you get 1 mark;
- if you get error code 0, or both error codes 1 and 2, you get 0 marks;
- if it crashes (with or without error code), you also get 0 marks.

Code Correctness (40%)

Here we will judge the general correctness of your code, by manual inspection or other means. This allows you to get some substantial marks, if you fail many test cases in the test suite for some naive mistakes. Equally there are almost certainly issues that the published test cases won't find, but may be reflected in the mark here.

Half of this (20%) is awarded based on whether your implementation is broadly speaking correct and implements the required functionalities. A range of methods may be used, such as additional test cases not released to you, or manual inspection.

The other half (20%) is specifically awarded to the handling of inheritance and virtual functions, that may not be identified directly by the test suite. This may involve running new test cases, manual inspection of code, or other means. In particular:

- Some of the classes should be abstract, as indicated in the assignment specification.
- Member variables or functions, or the implementation of (part of) the functions, should be placed at the highest point in the class hierarchy, where appropriate. In other words, you should "factor" the common parts to the base classes to avoid unnecessarily repetitive code in child classes. Note that there may be multiple approaches to the design and there may not be one single "correct" or "best" way to place these variables/functions/implementations.

You must use inheritance and virtual functions to achieve the intended behaviour. For example, you should not simply include a member variable indicating the "type" of objects (or any other means to deduce the exact subclass types), then simply implement everything in the base class like

```
void translate(...) {
  if (type == "Point") { // point-translation code }
  else if (type == "LineSegment") { // line-translation code }
  else if ...
}
```

- That would cost you most or all the marks here. One way of thinking this is that your code should require almost no changes if some other subclasses are added: if someone decides to add a Triangle class in the future, you should not need to find all such if-then-else in your codebase and add triangle-related code there.
- Contrary to what is stated in the general list on readability/coding styles below, the use of keywords `override` and `final` in relation to overriding virtual functions is required in this assignment. Absence of them will lead to a deduction of marks here (not in the readability/coding style part).

In both parts, "stepped marking" is used, so the mark should be one of 0/4/8/12/16/20.

Also in both parts, if additional test cases are used, they will be applied to everyone's submissions, and you will see those test cases and results afterwards. This will not affect your mark of the execution testing part.

Coding Style / Readability (10%)

This is worth 10 points. See list for issues that affect the mark.

List of Issues

- Comments: please ONLY include useful comments. Too many comments is likely to cost you marks. Most of the assignment tasks are quite strict, there are not too many different ways to write them and so there should be little need for comments. Good code should be self-documenting.
  Do not write comments that just translate your code back into English.
- But if you have added new public/private functions, add a brief comment about what they do. Same for private variables.
- Correct indentation, i.e. not misleading readers that something is inside a block when it isn't, or similar.
  Also you may wish to check that you have not accidentally used a mix of tab and space characters for indentation, which may make it look weird on an editor of different tab size. Some editors can automatically convert any tabs you type into spaces.
  If you use actual spaces rather than tabs, the tab size (number of spaces per level of indentation) should be at least 2 and not more than 8. Just so you know, I may be reading your code on an editor with tab size 4 or 8.

- Meaningful variable / function names, but not too long especially those used only locally. Private class members should use some kind of underscore notation such as size_ or m_size so they can be distinguished from local variables. Should use a consistent capitalisation scheme such as thisIsAVariable or this_is_a_variable.

Reasonable spacing. There is no one common standard here, but

```
void f(){for(int i=0;i<99;i++)for(int j=0;j<99;j++)std::cout<<"i="<<i<<"j="<<j<<std::endl;}
```

- is probably not particularly eye-pleasing.
- Avoid unnecessarily broad scope, e.g. global variables.
- Enforce encapsulation, i.e. class members should not be public unless necessary, similarly for friendships.
- Class/function design or factoring (if appropriate). Note that for many of the tasks / assignments the design is given to you and you cannot change so this is irrelevant. You do not need to go for the "best" refactoring; please apply common sense. Marks will only be deducted if things are "clearly" bad.
- Clearly redundant or indirect ways of doing things, and at a significant level, may incur deductions. Again, small amounts of repetitions or poor ways of doing things will not be penalised.
- Issues that compilers sometimes warn you about, such as unused variables, dangling else or fallthrough switch that are either intentionally clever or unintended but just happens to work. Any intentionally clever tricks should come with comments (and may not lead to better marks).
- Use of const, override, final, noexcept etc. where appropriate. Again, some of these may be fixed in the assignment. Absence of those will not lead to deduction (unless otherwise specified in the assignment), but appropriate presence may be awarded marks.
- Use of outdated things such as NULL when nullptr should be used instead, will incur deductions.
- ...

What is not included:

- Efficiency (speed or memory usage) of your program, within reason.
- Memory leaks, not implementing the correct virtualness/abstractness, etc: these are counted in the code inspection part.

You might also want to look at the Google C++ style guide (but don't treat it religiously, and you are not working at Google).

Generally speaking the marks will be given as follows:

- 9 - You start at here: you get this if no coding style / readability issues, or anything particularly commendable, were identified.
- 10 - You go up by at most 1 point if there are something particularly commendable (see list above).
- 5,6,7,8 - 1 mark deducted for each systemic/widespread readability/coding style issues in the above list. Annoying but not unreadable.
- 2,3,4: Issues are so severe that makes marking significantly more difficult. Will certainly confuse and mislead readers.
- 1 - You should enter the IOCCC.
- 0 - There is too little code (that is your own), it is impossible to be unreadable