Lab #6: Inheritance and Virtual Functions

1. Objectives

In this lab you will gain experience with inheritance and virtual functions, which are powerful aspects of C++ that allow new types of code re-use. You will do this by enhancing a program that can store, manipulate, and draw a database of shapes.

2. Problem Statement

The bulk of the processing necessary to manage the database of shapes already exists, and without changing this code you will add new functionality simply by adding three new classes (Circle, Rectangle and Polygon). This is an example of adding new features to an existing *framework.* The framework provides the "higher-level" decision making of the program, and you add new features by creating new classes that the framework will call when appropriate, via virtual functions. This is a powerful form of code re-use; the framework is making use of objects that did not exist when it was originally written.

3. Command Reference

Your program should accept and execute the commands listed below. The parsing code and the general "framework" processing code has already been written for you. The framework code checks the input for basic errors and outputs appropriate messages, which are not listed below. Then the framework code executes the commands by calling appropriate functions on Shape objects.

In the commands below, `console` font indicates a keyword that must be typed exactly as shown, while *italics* indicate arguments that should be replaced by the appropriate strings or numbers.

- `tri` *name colour x1 y1 x2 y2 x3 y3*: Creates a new `Triangle` object, with the specified name and colour, and with 3 vertices given by (*x1, y1*), (*x2, y2*) and (*x3, y3*). *name* and *colour* are strings, while *x1* through *y3* are floats. *name* can be any string, and is used to identify the object. *colour* must be one of `"white", "black", "grey55", "grey75", "blue", "green", "yellow", "cyan", "red", "darkgreen", or "magenta"`.

- `circ` *name colour xcen ycen radius*: Creates a new `Circle` object, with the specified *name* and *colour*, and with a center at (*xcen,* ycen) and the specified *radius.* name and *colour* are strings, while *xcen, ycen* and *radius* are floats.

- `rect` *name colour xcen ycen width height*: Creates a new `Rectangle` object, with the specified name and colour, a center at (*xcen,* ycen) and the specified *width* and *height.* name and *colour* are strings, while *xcen, ycen, width* and *height* are floats.

- `polygon` *name colour x1 y1 x2 y2 x3 y3 x4 y4 ...* x_n y_n

- `printall:` Prints out all the shapes in the database. See Section 5 for example output. For each shape, the following should be printed, all on one line. Text in `console` font should be output exactly

as shown, while values in *italics* should be replaced by the appropriate data in the output. All floating point numbers should be printed with **1 digit** after the decimal place and all spaces shown in the text below are single spaces. All shapes should print their name, colour and center, and then each shape prints its type and remaining data as specified.

  - *name colour* `center: (`*xcen*`,`*ycen*`)`
  - `circle radius: `*radius*
  - `rectangle width: `*width* `height: `*height*
  - `polygon (`*x1*`,`*y1*`) (`*x2*`,`*y2*`) ... (`x_n`,`y_n`)`

- `remove` *name:* Removes the shape with the specified *name* from the database, or prints an error message if no shape with that name exists.

- `scale` *name scaleFactor:* Scales the size of the shape with the specified *name* by the specified *scaleFactor*. *name* is a string, while *scaleFactor* is a float. A *scaleFactor* of 2 would make a rectangle twice as wide and twice as tall, or would double the radius of a Circle, for example. The center of the object does not move. For a polygon, the center of the object is the average of all its vertices.

- `translate` *findX findY Xshift Yshift:* All parameters are floats. This command executes in two parts. First it determines if location (*findX, findY*) falls within any shape. If the location falls within a shape, that shape is moved by shifting its center by (*Xshift, Yshift)*. If the location does not fall within a shape, an error message is printed. If multiple shapes overlap at *(findX, findY)* the shape which was inserted in the database last is considered to be "on top" and is the one moved.

- `area:` Computes the total area of all the shapes in the database and outputs this sum. The computation is done in floating point, and is printed out with one digit after the decimal point.

- `perimeter:` Computes the total perimeter of all the shapes in the database and outputs this sum. The computation is done in floating point, and is printed out with one digit after the decimal point.

- `draw:` Creates (if it doesn't exist yet) a graphics window, and draws all the shapes to that window. The graphics window is then "in control" of the program until the ***Proceed*** button is pressed – no more commands can be entered until the ***Proceed*** button is pressed in the graphics window.

4. Graphics Reference

When the draw command is entered for the first time, the graphics window below will be created, as shown in Figure 1. The shapes that have been entered will be drawn and control will pass to the graphics window. You can click on a shape, then click on where you would like to move its center, and the shape will move – basically this is invoking the same functionality as the translate command. You can also click on the various buttons on the right hand side of the window to pan and zoom the display of shapes; Figure 2 gives a description of what each button does.

While the graphics window is active, you cannot enter more commands in the regular command window. Your program is not waiting for "events", or data, from cin – instead it is waiting for events (mouse clicks, etc.) from the graphics window. When you click the **Proceed** button, the shape program will resume processing keyboard input from cin, and the graphics window will no longer respond to mouse clicks.

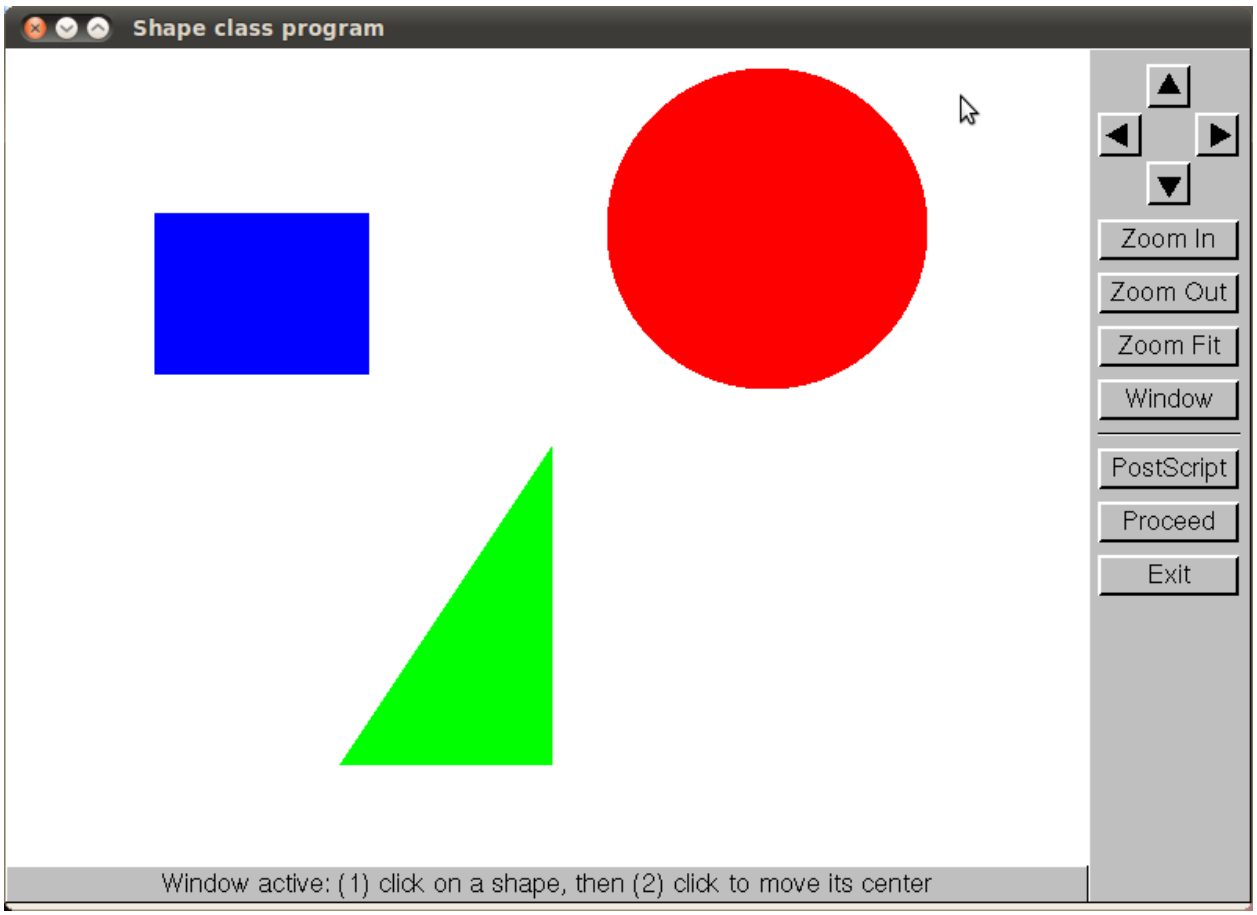
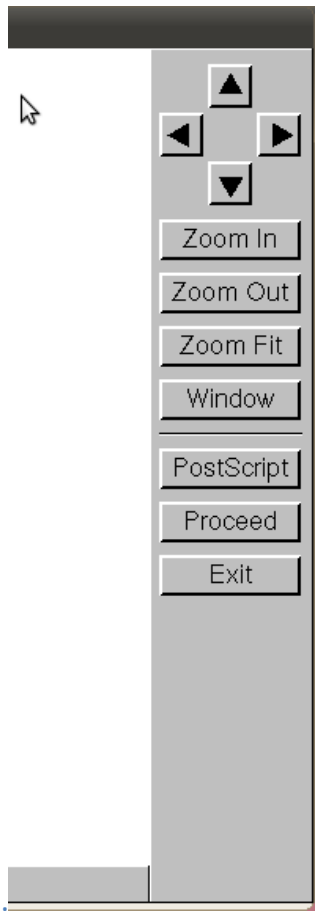


Figure 1: Graphics window; click on a shape, then click where you would like to move its center.



Arrow Buttons: pan (shift) image.

Zoom In: focuses on center of image.

Zoom Out: shows more image.

Zoom Fit: shows entire image (as defined by the maximum and minimum world coordinates in the last set_world_coordinates).

Window: Click on the diagonally-opposite corners of a box in which to zoom.

Postscript: writes image to pic1.ps (first click), pic2.ps (2nd click) etc.

Proceed: returns from gl_event_loop ().

Exit: Ends program.

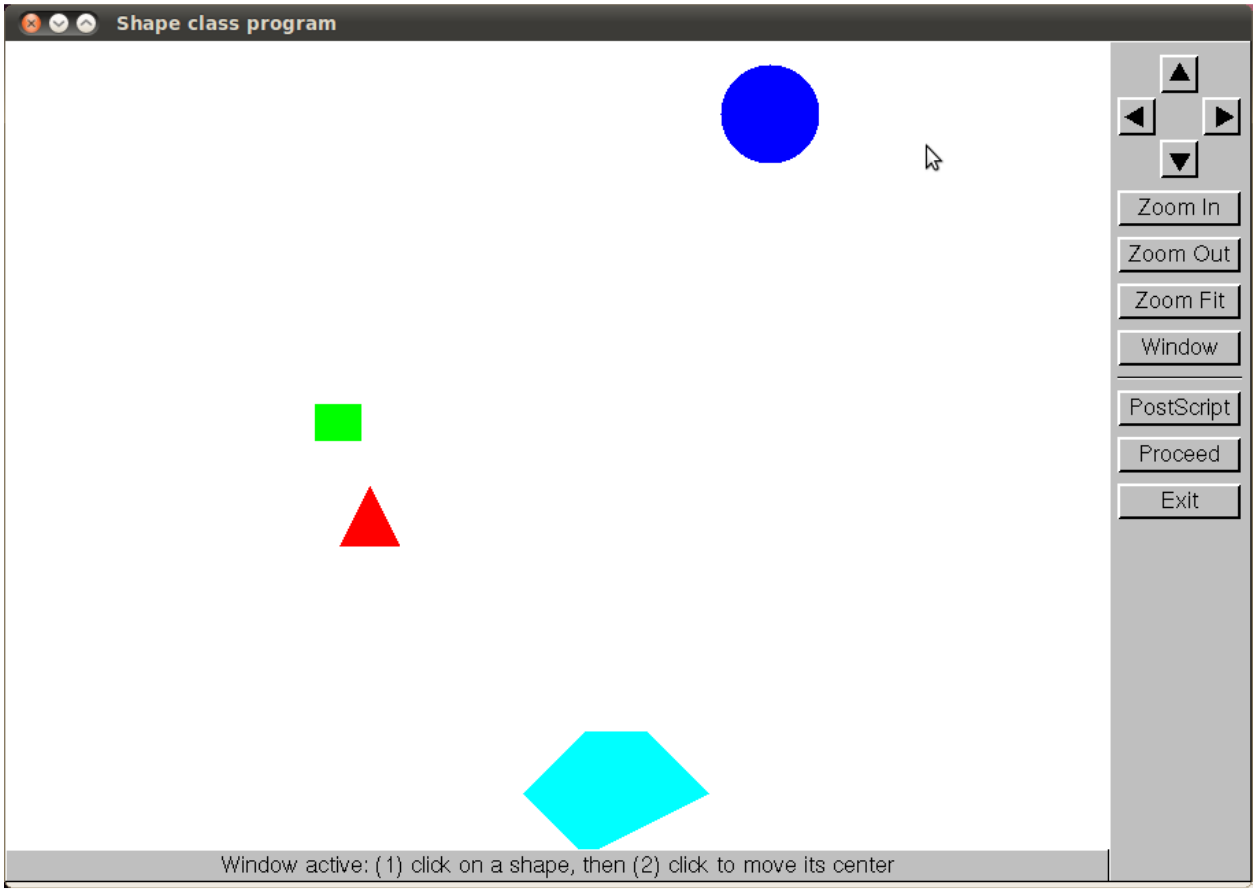
Moving the mouse scrollwheel will also zoom in and out, and moving the mouse while you hold down the scrollwheel will pan (shift) the image.

Figure 2: Graphics buttons and their functions.

5. Sample Session Output

```
> tri t1 red -2 0 -1 0 -1.5 1
Success
> circ c1 blue 2 3 0.8
Success
> rect r1 green -2 2 1.5 1.2
Success
> poly p1 cyan 2 -3 3 -3 4 -4 2 -5 1 -4
Success
> printall
t1 red center: (-1.5,0.3) triangle (-2.0,0.0) (-1.0,0.0) (-1.5,1.0)
c1 blue center: (2.0,3.0) circle radius: 0.8
r1 green center: (-2.0,2.0) rectangle width: 1.5 height: 1.2
p1 cyan center: (2.4,-3.8) Polygon (2.0,-3.0) (3.0,-3.0) (4.0,-4.0) (2.0,-5.0) (1.0,-
4.0)
> area
Total area: 7.8
> perimeter
Total perimeter: 21.1
> translate 2 2.8 3 4
Success
```

```
> scale r1 0.5
Success
> printall
t1 red center: (-1.5,0.3) triangle (-2.0,0.0) (-1.0,0.0) (-1.5,1.0)
c1 blue center: (5.0,7.0) circle radius: 0.8
r1 green center: (-2.0,2.0) rectangle width: 0.8 height: 0.6
p1 cyan center: (2.4,-3.8) Polygon (2.0,-3.0) (3.0,-3.0) (4.0,-4.0) (2.0,-5.0) (1.0,-
4.0)
> draw
Passing control to graphics window. Click Proceed to return control to this command
window
```




```
Control returned to command window
> <Ctrl+D>
```

5. Coding Specification

Download the files listed below from blackboard. You are to implement three classes, `Rectangle`, `Circle`, and `Polygon` in the files specified. You will also have to add a small amount of code to `Main.cpp` to create new Rectangle, Circle, and Polygon objects when the `rect`, `circ` and `poly` commands are parsed, respectively. The code to parse those commands is already in `Main.cpp` – you need only create the appropriate objects.

Files that you do not need to change:

- **Shape.h and Shape.cpp:** Base class for shapes. It contains the data common to all shapes: name, colour, and the shape center (xcen, ycen). It also contains definitions of functions that will work for all Shape objects, such as getName(). Finally, it contains virtual functions defining the interface that derived classes must provide; for example draw and computeArea functions.
- **ShapeArray.h and ShapeArray.cpp:** Stores the array of shapes, as an array of Shape* pointers. This class stores the shapes, parses most commands, and manipulates the array of shapes and outputs text and graphics by calling the appropriate Shape functions (some regular and some virtual).
- **Triangle.h and Triangle.cpp:** A derived class that inherits from Shape, and extends it to implement Triangles. Triangle adds data members for 3 vertices (all stored relative to the Shape center). All virtual functions defined by Shape are implemented in Triangle, in a way appropriate for Triangles.
- **easygl.h, easygl_constants.h, graphics.h, easygl.cpp and graphics.cpp:** Define the graphics library used to display graphics, handle panning and zooming and so on.

Files you must modify or create:

- **Main.cpp:** Basic parsing and setup code. You must add a small amount of code to construct a Circle, Rectangle and a Polygon in the place indicated in the file, when a circ, rect or poly command has been parsed, respectively.
- **`Rectangle.h and Rectangle.cpp`**: Create these files and implement the Rectangle class. This class must inherit from the Shape class. You will have to add appropriate data members (which must be of private type), and implement Rectangle versions of all the virtual functions defined in Shape.
- **`Circle`.h and Circle.cpp:** Create these files and implement the Circle class. It must also inherit from Shape, and will require some additional private data members and must implement Circle versions of all the virtual functions defined in Shape.
- **`Polygon`.h and Polygon.cpp:** Also inherits from Shape, just like Circle. You have to handle **only polygons without "holes"** them (formally: simple, non-self-intersecting polygons). This simplifies computing the polygon area and determining if a point is inside a polygon or not.

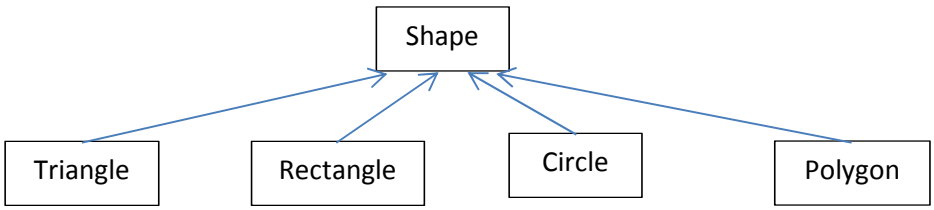


Figure 3: Inheritance Hierarchy.

Shape.h listing:

```
#define PI 3.141593   // This constant may be useful to you
#include <string>
#include "easygl.h"   // To get at graphics.
using namespace std;

class Shape {
private:
```

```
   string name;    // Name of this shape, can be used as its identifier

   // A string giving the colour of the shape. Valid values are:
   // "white", "black", "grey55", "grey75" "blue", "green", "yellow",
   // "cyan", "red", "darkgreen", "magenta"
   string colour;

   float xcen, ycen;  // x and y coordinates of the center of the shape.

public:
   // Accessor and mutator functions implemented by Shape
   string getName () const;
   void setName (string _name);
   string getColour () const;
   void setColour (string _colour);
   float getXcen () const;
   void setXcen (float _xcen);
   float getYcen () const;
   void setYcen (float _ycen);


   // Virtual functions that *may* be overriden by the derived classes if
   // they wish below. The functions below have a reasonable implementation
   // in Shape, and derived classes may wish to use that implementation
   // rather than overriding it.
   //*****************************************************************

   // Shift the object's center by the specified amount. Derived classes
   // should be defined so that shifting the object center shifts where the
   // whole object is drawn.
   // Implemented by Shape.
   virtual void translate (float xShift, float yShift);

   // Print out the object. Shape::print() will print the basic info contained
   // in the Shape base class. Made virtual so you get the right print method
   // (print extra info) for derived classes.
   virtual void print () const;

   // Below are pure virtual interface functions; these *must* be implemented in each
   // class derived from Shape.
   //*****************************************************************

   // Scale the object size by the specified factor.
   // E.g. a scaleFac of 1 would change nothing, 0.5 would shrink the object
   // to half its original size, and 2 would grow the object to twice its
   // original size. The object stays centered at the same xcen, ycen point.
   virtual void scale (float scaleFac) = 0;

   // Returns the area of the Shape.
   virtual float computeArea () const = 0;

   // Returns the perimeter of the Shape.
   virtual float computePerimeter () const = 0;

   // Draws the object, using the easygl drawing commands.
   virtual void draw (easygl* window) const = 0;

   // Returns true if the given (x,y) point is inside the Shape.
   // Otherwise returns false. Used to determine when we click on a shape.
   virtual bool pointInside (float x, float y) const = 0;

   // virtual destructor, in case we have different data to clean up in
   // different derived classes.
```

```
   virtual ~Shape ();

protected:
   // Constructors protected, so they can only be invoked from Derived
   // classes as part of building a Derived object.  No other classes can
   // create a Shape anyway, since it is an abstract base class (cannot be
   // instantiated).
   Shape ();
   Shape (string _name, string _colour, float _xcen, float _ycen);
};
```

6. Compiling with Graphics

Since this program includes graphics, it requires the compiler (specifically the *link* step of compilation) to include the low-level graphics library, which is called the X11 library, in the list of libraries it searches on Linux. This requires one extra option to be passed to the compiler.

- **NetBeans on Linux (e.g. ECF):** As shown in **Figure 4**, type **–lX11** in the *File* | *Project Properties* | *Build* | *Linker* | *Additional Options* field.
- **Command line on Linux (e.g. ECF):**
  ```
  g++ -g -Wall -lX11 *.cpp -o shape
  ```
- **Apple MacIntosh:** add –lX11 to your linker options, in the same way as for Linux computers.
- **Microsoft Windows with Visual Studio:** X11 is not needed; instead you should enter WIN32 in the *Project | Properties | Preprocessor | Configuration Properties | C/C++ | Preprocessor | Preprocessor Definitions* box in your MS Visual Studio project.
- **Microsoft Windows with NetBeans:**
  1. Tell the easygl graphics package you are compiling for MS Windows by adding **WIN32** to *File* | *Project Properties* | *C++ Compiler* | *Proceprocessor Definitions*.
  2. Tell NetBeans where to find the low-level MS Windows graphics library to which easygl interfaces. Click on *File* | *Project Properties* | *Linker* | *Libraries* | *Add Library* and select the path to your gdi32 library. With the default cygwin installation this would be in c:\cygwin\lib\w32api\libgdi32.a.

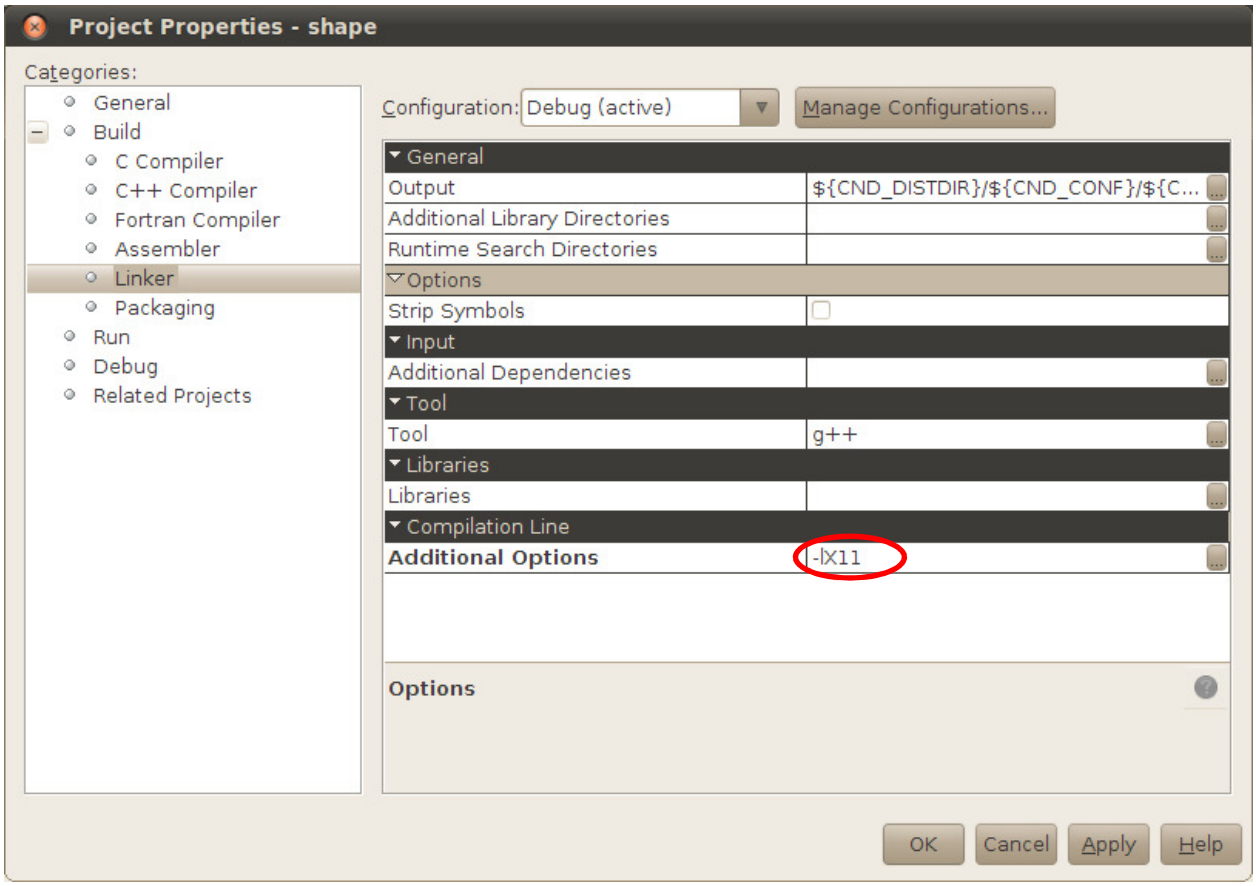


Figure 4: Compiling with graphics in NetBeans on a Linux system: make the circled setting.

7. Deliverables and Suggested Approach

- Create a lab6 directory, download all the files from blackboard, and create a NetBeans project called **shape**. Enter the **–lX11** setting in your Project Properties as described in Section 6. Compile and run the shape program; you should be able to enter, scale, translate, draw etc. Triangles (but not Circles, Rectangles or Polygons yet).
- Read the Shape.h and Triangle.h files carefully. Shape.h defines the class and interface you must extend, and Triangle gives a good example of how to extend that class.
- Implement the Rectangle class, and the small amount of code in Main.cpp needed to create Rectangles. To draw a Rectangle, the following graphics functions will be helpful:

```
easygl* window;   // A pointer to the window object (passed into draw)
window->gl_setcolor(colorStr);  // Sets color for subsequent drawing commands
window->gl_fillrect (xleft, ybottom, xright, ytop);
```

- Implement the Circle class, and the small amount of code in Main.cpp needed to create Circles. To draw a Circle, the following graphics function will be helpful:

```
window->gl_fillarc (xcen, ycen, radius, 0, 360);
```

- Implement the Polygon class, and the small amount of code in Main.cpp needed to create Polygons. To draw a Polygon, the following graphics function will be helpful:

```
window->gl_fillpoly(arrayOfVertices, numPoints);
```

- Computing the area of an arbitrary polygon is non-trivial. See http://www.mathopenref.com/coordpolygonarea2.html for a good method and tutorial.
- Determining whether a point is inside an arbitrary polygon or not is also non-trivial. The generalization of the pointInside routine in `Triangle.cpp` to N points will work. The test in Triangle.cpp uses the "crossing count" algorithm described at http://geomalgorithms.com/a03-_inclusion.html.
- Your program should pass exercise and other text-based tests, and also should draw shapes and respond to mouse clicks properly. Note that you will always have to click "Proceed" to return from the graphics when running exercise (and hence pass exercise).
- Your program must not leak memory; use exercise and valgrind to test. Note that when the graphics are invoked with the draw command during a run, some memory deep in the graphics library is still on the heap at the end of the program, so valgrind will report that some heap memory is still in use at exit. However, no memory leaks occur, and valgrind should report **definitely lost: 0 bytes** for your program in all cases.
- Submit your program using the command

```
~ece244i/public/submit 6
```

7. Bonus Part of Lab

An optional bonus for this lab will be released on blackboard shortly. The bonus is worth 5 additional marks on this lab if it works perfectly. In the bonus portion, you will implement an additional *composite* type of shape, which is the union of an arbitrary number of other shapes.

8. To Go Further

With the winter break approaching, we are sure many students are wondering how they will avoid boredom without programming assignments. Fear not! You can extend this lab in many ways if you get bored over the holidays.

1. Add a new "Scale" button that lets you click on a shape and then type in a number to scale the object.
2. (After completing the bonus part): add a new "Composite" button that lets you click on a series of shapes; all the shapes you click on before clicking away from any shape are used to create a new composite type shape.
3. And many more ...

To do these optional assignments, you will have to learn more about the graphics library (easygl) you are using. You can download a manual giving an overview of easygl's use and capabilities (and an intro to event-driven graphics) from http://www.eecg.toronto.edu/~vaughn/easygl/easygl.html. The easygl.h and easygl_constants.h header files give a highly commented list of the funtions available to set up and interact with the graphics.