

CS 202

Assignment #9

Purpose: Learn class inheritance, dynamic data allocation, and operator overloading.
Due: Thursday (3/21)
Points: 150

Assignment:

Calculations with complex number¹ can generate distinctive patterns when plotted and colored in two-dimensional space. The coloring scheme we will use is referred to as domain coloring².

We will write some classes to generate complex number images. Since the images require calculations with complex numbers, we will implement a class with complex numbers that supports operator overloading in order to more easily perform these calculations. Design and implement two C++ classes;

- ***complexType*** to provide complex operations
- ***complexPlot*** to implement the complex number fractal algorithm.

The UML class specifications are provided below. A main will be provided that uses the the ***complexType*** and ***complexPlot*** classes.

- Complex Type Class
The complex type class will implement the overloaded functions to support complex numbers.

complexType
-realPart: double
-imaginaryPart: double
+complexType (double=0.0, double=0.0)
<<friend>> +operator << (ostream&, const complexType&): ostream&
<<friend>> +operator >> (istream&, complexType&): istream&
+operator + (const complexType&): complexType
+operator - (const complexType&): complexType
+operator * (const complexType&): complexType
+operator / (const complexType&): complexType
+operator == (const complexType&): bool
+angle(): double
+abs(): double
+sine(): complexType
+cose(): complexType
+cSqrt(): complexType
+setComplex(const double&, const double&): void
+getComplex(double&, double&) const: void

Note, the stream operators must be overloaded as non-member functions.

1 For more information, refer to: http://en.wikipedia.org/wiki/Complex_number

2 For more information, refer to: http://en.wikipedia.org/wiki/Domain_coloring

A complex number is a number comprising a real and imaginary part. It can be written in the form (a,b) , where a and b are real numbers, and b represents the imaginary part. The complex numbers contain the ordinary real numbers, but extend them by adding in extra numbers and correspondingly expanding the understanding of addition and multiplication.

Complex Operations:

If $z_1 = (a,b)$ and $z_2 = (c,d)$ are complex numbers:

$$(a,b) + (c,d) = (a + c, b + d)$$

$$(a,b) - (c,d) = (a - c, b - d)$$

$$(a,b) * (c,d) = ((ac - bd), (ad + bc))$$

$$\sqrt{(a,b)} = \left(\sqrt{\frac{a + \sqrt{a^2 + b^2}}{2}}, \frac{b}{|b|} \sqrt{\frac{-a + \sqrt{a^2 + b^2}}{2}} \right)$$

$$\text{sine}(a,b) = (\sin(a) \cosh(b), \cos(a) \sinh(b))$$

$$\text{cose}(a,b) = (\cos(a) \cosh(b), \sin(a) \sinh(b))$$

If (c,d) is non-zero

$$(a,b) / (c,d) = ((ac + bd) / (c^2 + d^2), (-ad + bc) / (c^2 + d^2))$$

Additionally, some complex functions returning a double result:

$$|(a,b)| = \sqrt{a^2 + b^2} \quad // \text{ absolute value function}$$

$$\theta(a,b) = \arctan(b, a) \quad // \text{ angle function}$$

The a and b values are represented with doubles within the complex type class. The **arctan()** function is **atan2()** in <cmath>.

- Complex Plot Class

The complex plot class will support the creation of complex number images and provide a series of basic utility functions. The complex plot class should inherit the bitmap image class, **bitmapImage**, and utilize the and the **complexType** class (for the complex calculations).

complexPlot
-fileName: string
-funcNumber: int
-W_MIN=300, W_MAX=12000: static const int
-H_MIN=300, H_MAX=12000: static const int
-FUNC_LIMIT=11: static const int
-PI = 3.14159265358: static const double
-E = 2.71828182845: static const double
+complexPlot()
+complexPlot(int, int, int=0, string="")
+complexPlot(const complexPlot&)
+readFileName(): void
+setFileName(string): void
+getFileName() const: string
+readImageSize(): void

+setSize(int, int): void
+functionCount(): int
+getFunctionNumber(): int
+setFunctionNumber(int): void
+readFunctionNumber(): void
+createComplexPlotImage(): void
+createImageFile(): void
-func1(complexType, complexType): complexType
-func2(complexType, complexType): complexType
-func3(complexType, complexType): complexType
-func4(complexType): complexType
-func5(complexType): complexType
-func6(complexType): complexType
-func7(complexType): complexType
-func8(complexType): complexType
-func9(complexType): complexType
-func10(complexType): complexType
-func11(complexType): complexType
-setHSV(complexType, double&, double&, double&): void
-setRGB(double, double, double, int&, int&, int&): void

You may add additional private functions as needed.

Note, points will be deducted for especially poor style or inefficient coding.

Function Descriptions

- The *complexPlot()* constructor must initialize the class variables (NULL or zero's as appropriate) and may assume a 0 for width and height (which would be passed to the based class constructor).
- The *complexPlot(width, height, functionNumber, fileName)* constructor should verify the arguments (as specified in the below set functions).
 - ensure width is between W_MIN (300) and W_MAX (12000)
 - ensure height is between H_MIN (300) and H_MAX (12000)
 - ensure the aspect ratio (width/height) is between 0.50 and 2.0
 and set the *bitmapImage* base class width and height variables accordingly.
 - If non-empty, ensure the filename has a '.bmp' extension.
- The *complexPlot(const complexPlot&)* function is the copy constructor and should create a new image based on the passed complex plot image.
- The *readFileName()* function should read a file name from input, and ensure that it is valid (long enough) and that it has a '.bmp' extension. The *setFileName(string)* function sets the file name class variable (including verification). The *getFileName()* function returns the file name.
- The *readImageSize()* function should read the width and height from input (same limits and the constructor) and set the *bitmapImage* base class width and height variables accordingly. The routine should re-prompt until valid input is received. The *setSize(int,int)* function should also verify the values and set the base class width and height variables (in that order).

- The *createImageFile()* function create the image file using the class variable for file name with a 24-bit depth (by using the appropriate base class functions).
- The *createComplexPlotImage()* function should play the implement the complex plot algorithm as described below.
- The *functionCount()* function should return the maximum count of functions implemented (FUNC_LIMIT). If new complex functions are added to the class, the function would be added sequentially and the FUNC_LIMIT updated accordingly.
- The *getFunctionNumber()* should return the current function number. The *setFunctionNumber()* should set the current function number, including verifying that function number is between 0 and FUNC_LIMIT. The *readFunctionNumber()* function should read the function number from input and ensure it is between 0 and FUNC_LIMIT (inclusive).
- The private *setHSV(complexType, double &, double &, double &)* function should calculate the hue saturation, and brightness values based on the provided algorithm.
- The private *setRGB(double, double, double, int &, int &, int &)* function should accept the hue saturation, and brightness values and convert them to RGB (red, green, and blue).
- The private *func1()* through *func11()* functions should implement the complex calculations specified in the complex functions table.

Refer to the example executions for output formatting. Make sure your program includes the appropriate documentation. See Program Evaluation Criteria for CS 202 for additional information.

Complex Plot Algorithm:

A complex number fractal image is generated by computing a function using a complex number which is initialized based on the (x,y) point for a image. The calculation for the function is repeated for each (x,y) point in the plot area (width by height) and based on the behavior of that calculation, a hue, saturation, and brightness for that point are set based on the below algorithm.

Given the initializations:

```
const double rmi = -3;
const double rma = 3;
const double imi = -3;
const double ima = 3;
```

For each pixel in the screen, the following computations must be performed:

```
for every screen pixel:
    im = ima - (ima - imi) * j / (height - 1)
    re = rma - (rma - rmi) * i / (width - 1)

    c = v = (re,im)
    v = func<functionNumber>(v,c) || func<functionNumber>(v)

    setHSV(v, hue, sat, val)
    setRGB(hue, sat, val, red, green, blue)
    setPixel(row, col) = red, green, blue
```

Where *v* and *c* are complex numbers. The red, green, and blue variables are integers. All other variables are doubles.

Complex Functions:

The plot of a complex function will depend on the specific function used. The program should allow the user to select one, of multiple, functions to be plotted. The functions supported should include:

Function Number	Function
0	$f0(z) = z$
1	$f1(z, c) = z^2 + c$
2	$f2(z, c) = f1(z, z) + c$
3	$f3(z, c) = f2(z, z) + c$
4	$f4(z) = cose(\pi z)$
5	$f5(z) = z^2 + 1$
6	$f6(z) = \frac{sine(z^3 - 1)}{z}$
7	$f7(z) = f6\left(f8(z) + \frac{1}{f8(z)}\right)$
8	$f8(z) = f9\left(\frac{(z^2 - 1)(z - 2 - i)^2}{z^2 + 2 + 2i}\right)$
9	$f9(z) = z + \frac{1}{z}$
10	$f10(z) = \sqrt{z}$
11	$f11(z) = \frac{z^2 + 1}{z^2 - 1}$

Note, variables z and c are complex numbers. The value of 1 as a complex number is (1.0, 0.0), the value of 2 as a complex number is (2.0, 0.0), the value of π is (π , 0.0), and the value of i as a complex number is (0.0, 1.0).

Set Hue, Saturation, and Brightness³:

The hue, saturation, and brightness values are set based on the current value of the complex number in the following manner:

```
hue = angle(v)
while (hue < 0.0)
    hue += 2.0  $\pi$ 
hue /= 2.0  $\pi$ 

m = abs(v)
ranges = 0.0
rangee = 1.0
```

3 For more information, refer to: http://en.wikipedia.org/wiki/HSL_and_HSV

```

while (m > rangee)
    ranges = rangee
    rangee *= E

k = (m - ranges) / (rangee - ranges)

if (k < 0.5) then
    sat = 2.0 k
else
    sat = 1.0 - 2.0(k - 0.5)
sat = 1.0 - (1.0 - sat)3
sat = 0.4 + 0.6 sat

if (k < 0.5)
    val = 2.0 k
else
    val = 1.0 - 2.0(k - 0.5)

val = 1.0 - val
val = 1.0 - (1.0-val)3
val = 0.6 + 0.4 val

```

Where the variable *v* is a complex number passed into the routine.

Hue, Saturation, and Brightness⁴ Conversion:

The hue, saturation, and brightness calculated by the algorithm must be converted into red, green, and blue values for the image. Assuming that hue (*hue*), saturation (*sat*), and brightness (*val*) are doubles and passed into the routine, the following algorithm is used to perform the conversion.

- If the saturation is 0.0, the red, green, and blue values are each set to the brightness value.
- Otherwise
 - if the hue is 1.0, reset the hue to 0.0
 - compute the following:

```

z = [ 6.0 hue ]
f = 6.0 hue - z
p = val(1.0 - sat)
q = val(1.0 - sat f)
t = val(1.0 - sat(1.0 - f))

```

- based on the formulas, set the red, green, and blue values as follows:
 - if z is 0, set red = val, green = t, and blue = p
 - if z is 1, set red = q, green = val, and blue = p
 - if z is 2, set red = p, green = val, and blue = t
 - if z is 3, set red = p, green = q, and blue = val
 - if z is 4, set red = t, green = p, and blue = val
 - if z is 5, set red = val, green = p, and blue = q
 - scale the red, green, and blue values (multiply each by 256).
 - convert the red, green, and blue values to integer
 - if any color (red, green or blue) is > 255, set the final color value to 255

Note, the algorithm assumes all calculations are real and cast to integer at the last step.

4 For more information, refer to: http://en.wikipedia.org/wiki/HSL_and_HSV

- **Bit Map Image Class**

There are many programs that can write information to a simple, device and OS independent bitmap image format. The *bitmapImage* class (**bitmapImage.h** and **bitmapImage.so**) provides a simple interface for the creation of an uncompressed BMP format bitmap file.

bitmapImage
-width: int
-height: int
-**pixelData: int
-freePixelData()
+bitmapImage()
+bitmapImage(const char*)
+bitmapImage(int, int)
+~bitmapImage(): virtual
+setSize(int, int): void
+loadFromBitmapFile(const char*): bool
+saveToBitmapFile(string) const: void
+saveToBitmapFile(string, int) const: void
+getHeight() const: int
+getWidth() const: int
+getRed(int color) const: int
+getGreen(int color) const: int
+getBlue(int color) const: int
+getColorForRGB(int, int, int) const: int
+getPixel(int, int) const: int
+setPixel(int x, int y, int color): void
+fillWith(int color): void
+fillRect(int, int, int, int, int): void
+setPixelRGB(int, int, int, int, int): void

The bitmapImage class (object file) and header file will be provided.

Make File:

With the provided make file, you can type:

make

Which should create the executable.

Submission:

- Submit a compressed zip file of the program source files, header files, and makefile via the on-line submission by 23:50.

All necessary files must be included in the ZIP file. The grader will download, uncompress, and type **make** (so you must have a valid, working *makefile*).

Example Execution:

Below is an example program execution for the main, including the error testing.

```
ed-vm% ./main
```

```
-----  
Part 1: Complex Number Tests.
```

```
    num1 = (23, 34)  
    num2 = (3, 4)  
    num3 = (26, 38)  
    num1+num2 = (26, 38)  
    num1-num2 = (20, 30)  
    num1*num2 = (-67, 194)  
    num1/num2 = (8.2, 0.4)  
    angle(num1) = 0.976037  
    abs(num1) = 41.0488  
    sine(num1) = (-2.46869e+14, -1.55444e+14)  
    cose(num1) = (-1.55444e+14, -2.46869e+14)  
    cSqrt(num1) = (5.65901, 3.00406)  
    num1 != num2  
    num2 = num2
```

```
-----  
Part 2: Complex Plot Image Creation.
```

```
Error, file name must be '.bmp' extension.
```

```
-----  
Constructor Error Testing (size):
```

```
Error, height must be between 300 and 12000.  
Error, width must be between 300 and 12000.  
Error, width must be between 300 and 12000.  
Error, height must be between 300 and 12000.  
Error, height must be between 300 and 12000.  
Error, width must be between 300 and 12000.
```

```
Constructor Error Testing (file name):
```

```
Error, file name must be '.bmp' extension.  
Error, invalid file name.
```

```
setSize() Error Testing:
```

```
Error, height must be between 300 and 12000.  
Error, width must be between 300 and 12000.  
Error, width must be between 300 and 12000.  
Error, height must be between 300 and 12000.  
Error, height must be between 300 and 12000.  
Error, width must be between 300 and 12000.
```

```
setSize() Aspect Ratio Error Testing:
```

```
Error, invalid aspect ratio. Must be between 0.5 and 2.0.  
Error, invalid aspect ratio. Must be between 0.5 and 2.0.
```

```
setFileName() Error Testing:
```

```
Error, file name must be '.bmp' extension.  
Error, invalid file name.  
Error, invalid file name.
```

```
setFunctionNumber() Error Testing:
```

```
Error, function number must be between 0 and 11.  
Error, function number must be between 0 and 11.  
Error, function number must be between 0 and 11.  
Error, function number must be between 0 and 11.
```

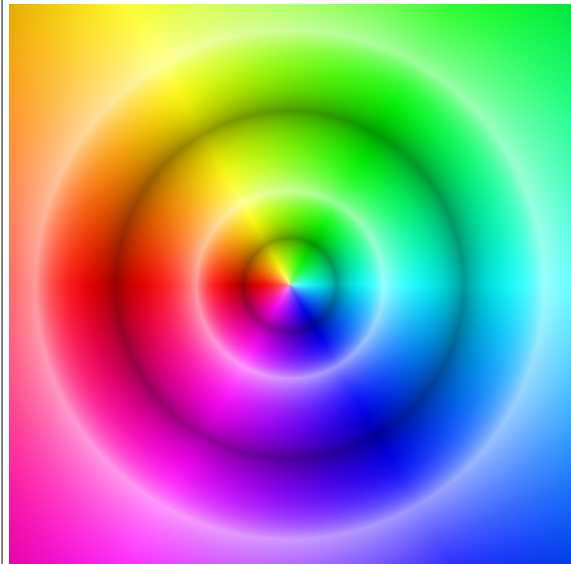

```
-----
```


Enter Image Information:
Enter Function Number (1 - 11): -3
Error, function number must be between 0 and 11.
Enter Function Number (1 - 11): 31
Error, function number must be between 0 and 11.
Enter Function Number (1 - 11): 11
Enter image size (width, height): -3 300
Error, width must be between 300 and 12000.
Please re-enter.
Error, invalid aspect ratio. Must be between 0.5 and 2.0.
Please re-enter.
Enter image size (width, height): 300 -3
Error, height must be between 300 and 12000.
Please re-enter.
Error, invalid aspect ratio. Must be between 0.5 and 2.0.
Please re-enter.
Enter image size (width, height): 0 0
Error, width must be between 300 and 12000.
Please re-enter.
Error, height must be between 300 and 12000.
Please re-enter.
Enter image size (width, height): 12000 300
Error, invalid aspect ratio. Must be between 0.5 and 2.0.
Please re-enter.
Enter image size (width, height): 300 1200
Error, invalid aspect ratio. Must be between 0.5 and 2.0.
Please re-enter.
Enter image size (width, height): 300 300
Enter Output File Name: *
Error, invalid file name.
Please re-enter.
Enter Output File Name: tmp
Error, invalid file name.
Please re-enter.
Enter Output File Name: tmp.bpm
Error, file name must be '.bmp' extension.
Please re-enter.
Enter Output File Name: .bmp
Error, invalid file name.
Please re-enter.
Enter Output File Name: tmp.bmp

ed-vm%

Example Output:

Below are some examples of the final output of the complex plot program.
Most image viewers will be able to display a '.bmp' file.

Complex Plot Sample Images	
Function 0	Function 7
	
Function 8	Function 11
