COMP0002 Programming Principles

Programming Notes and Exercises 1

Example Answers

Core questions

Q1.1

a) Type in, compile and run this program:

Example Answer:

Just follow what the question asks you to do.

Q1.2 Modify the program in question 1.1 to display the address of the Computer Science Department rather than just your name. Each line of the address will need to be stored in a separate variable. For example,

```
char name[] = “Dept. of Computer Science”;
char address1[] = “Malet Place Engineering Building”;
etc.
```

You will need to give your program a new name and save it to a new file, otherwise you will overwrite your answer to Q1.1.

Example Answer:

```
#include <stdio.h>

int main(void)
{
  char name[] = "Dept. of Computer Science";
  char address1[] = “66-72 Gower Street";
  char address2[] = "London";
  char address3[] = "WC1E6BT";

  printf("%s\n", name);
  printf("%s\n", address1);
  printf("%s\n", address2);
  printf("%s\n", address3);

  return 0;
}
```

This program defines four character array variables (type char[ ]) and *initialises* them with strings giving the address of the department. They are then printed out using printf statements using a *formatting string* of `“%s\n”`. The %s within the string is a conversion character that acts as a

marker of where a string will be substituted to complete the actual output string. The `\n` is an escape character representing newline.

The four printf statements could be replaced by a single version, with a more complex formatting string:

```
printf("%s\n%s\n%s\n%s\n", name, address1, address2, address3);
```

This results in exactly the same output:

```
Dept. of Computer Science
Malet Place Engineering Building
Gower Street, London
WC1E6BT
```

Q1.3 Type in the octagon drawing program presented earlier. Save it to a file and then run the program. Make sure the shape displayed is the same as the one in the picture.

Example Answer:

Follow the instructions in the question.

Q1.4 Modify your program to draw a heptagon (7 sides). Don't forget to save it to a new file, otherwise your answer to 1.3 will get overwritten. Make sure the heptagon is drawn accurately, don't just guess at the co-ordinates so sit seems to look correct! Hint: Heptagons are surprisingly tricky to draw, check Wikipedia for more information.

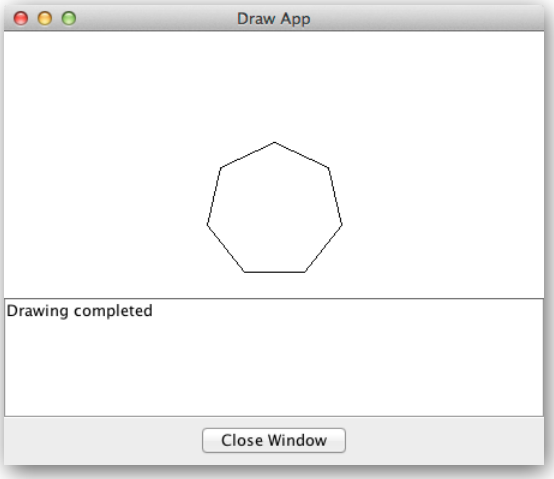
Example Answer:

To draw the heptagon correctly, each side must be the same length and the angle between each pair of sides must be the same. Heptagon's are awkward to draw but you can work out a set of co-ordinates for the start and end of each line in a straightforward way by exploiting right-angled triangles and simple trigonometry.

This program will display a suitable heptagon:

```
#include "graphics.h"

int main(void)
{
  drawLine(200,200,250,200);
  drawLine(250,200,281,161);
  drawLine(281,161,270,113);
  drawLine(270,113,225,92);
  drawLine(225,92,180,113);
  drawLine(180,113,169,161);
  drawLine(169,161,200,200);
  return 0;
}
```

While this code will draw a heptagon, it is not a very convenient way of drawing such shapes; the level of abstraction is too low as you have to do all the calculation to determine the coordinates of the lines making up the sides of the heptagon!

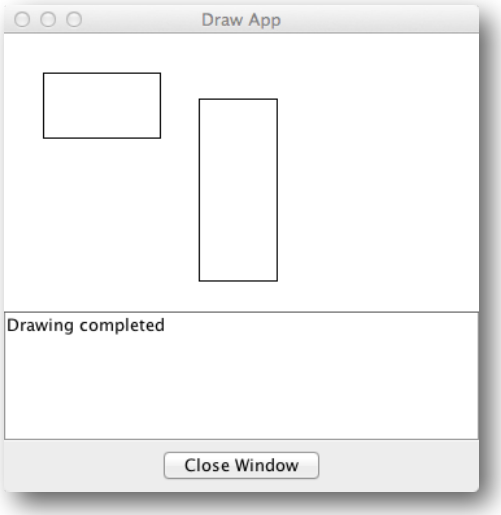


Q1.5 Write a program to draw two rectangles:
One with with top left corner at (30,30) and horizontal sides of length 90 and vertical sides of length 50, and another with top left corner at (150,50) and horizontal sides of length 60 and vertical sides of length 140.

Draw the first rectangle with drawLine and the second with drawRect.

Example Answer:

```
#include "graphics.h"

int main(void)
{
  drawLine(30,30,120,30);
  drawLine(120,30,120,80);
  drawLine(120,80,30,80);
  drawLine(30,80,30,30);
  drawRect(150,50,60,140);

  return 0;
}
```



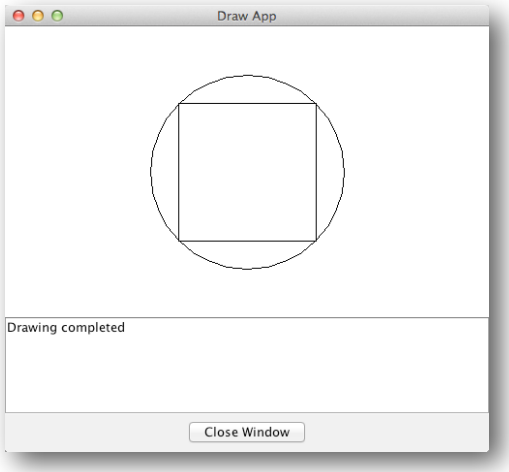
Q1.6 Write a program to draw a square inside a circle, such that the corners of the square touch the circle.

Use drawArc to draw the circle.

Example Answer:

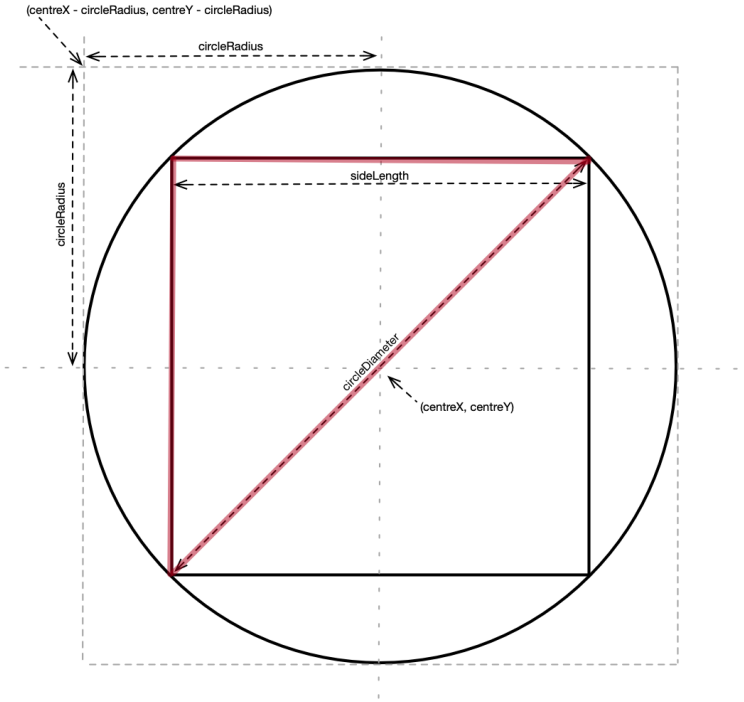
```
#include "graphics.h"

int main(void)
{
  drawArc(150, 50, 200, 200, 0, 360);
  drawRect(179, 79, 142, 142);

  return 0;
}
```

The program displays:



This program works but is another example of where you have to do most of the work of calculating coordinates, which also makes it harder to change the position and size of the shapes or display several copies of the shapes. Further if you squint at the program output above you wonder if the corners of the square are really in the right positions! The resolution of the drawing window makes this hard to determine.

We can improve on the first version of the program by doing some planning to find out how the coordinates needed can be calculated in a systematic way. Look at the diagram below:



If you decide on the size of the square, you have the length of a side, and the length of the diagonal between opposite corners can be calculated as you have a right-angled triangle (shown in red). This is, of course, done using the very familiar $a^2 = b^2 + c^2$ equation for right angled triangles. The square diagonal is the diameter of the circle, half that is the radius, and with the radius you can find the coordinates of the top left corner of the boundary square needed by `drawArc` to draw the circle. Once you have this worked out, you can draw the square in the circle given any centre point as the position of the shapes.

This gives the following code:

```
#include "graphics.h"
#include <math.h>

const int WIDTH_X = 500;
const int HEIGHT_Y = 300;

int main(void)
{
  int centreX = WIDTH_X / 2;
  int centreY = HEIGHT_Y / 2;
  int sideLength = 150;
  int circleDiameter =
    (int)sqrt(sideLength * sideLength + sideLength * sideLength);
  int halfSideLength = sideLength / 2;
  int circleRadius = circleDiameter / 2;

  drawArc(centreX - circleRadius, centreY - circleRadius,
          circleDiameter, circleDiameter, 0, 360);
  drawRect(centreX - halfSideLength, centreY - halfSideLength,
           sideLength, sideLength );

  return 0;
}
```

The code uses some features of C not really covered at this stage of the module, but is included here to illustrate the direction we need to go in. The two variables outside of the main function (`WIDTH_X` and `WIDTH_Y`) give the dimensions of the drawing window. They are declared as const meaning they are constant values that cannot be changed by assigning a different value. This is because the size of the drawing window is fixed and the values should not be changed. Using const allows a programmer to explicitly enforce the no change decision. If code is written to assign a different value to a const variable, the compiler will report an error.

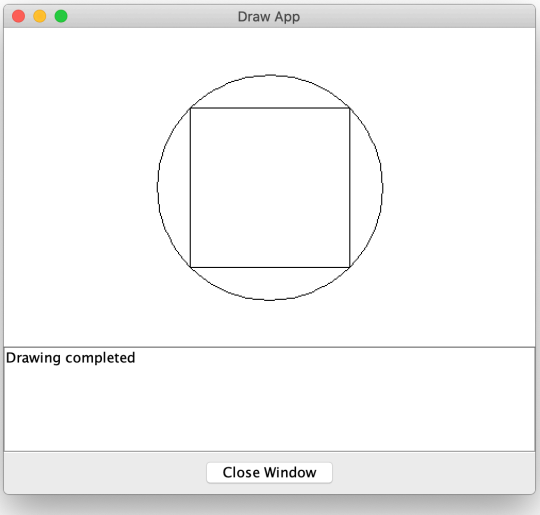
The variables inside the main function store the various values that need to be calculated in order to draw the shapes. The strategy of using a number of variables, each storing a distinct value, makes the code more readable and avoids the calculation expressions getting over-complex. The two variables `centreX` and `centreY` specify the position of the centre the circle and square, determining where they will be drawn.

The program uses the C maths library function `sqrt` to calculate the square root needed. This function is declared in the header file `math.h`, which is why the code has `#include<math.h>` at the start. When the code is compiled the `-lm` flag (link maths) may be needed on the command line to make sure that the code implementing the `sqrt` function is found, for example:

```
gcc -o squareInCircle squareInCircle.c graphics.c -lm
```

Whether the `-lm` flag is needed will depend on the compiler version and operating system you are using.

The program displays:



We can still do better though! While this is again going well beyond what has been covered at this stage of the module, it is worth introducing some further program design concepts now as you will need to understand and use them later. If you don't follow this discussion now, then come back later in the module to review this section.

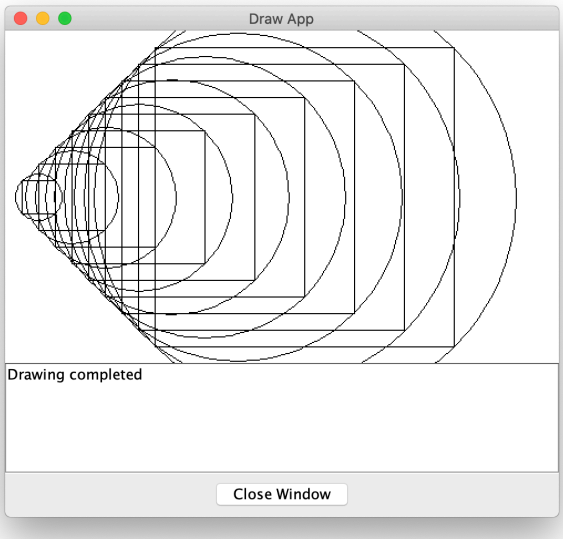
It would be a good improvement to move the code into a new function, which takes the mid-point and the length of side as parameters. The `main` method then calls the new function. This gives:

```
#include "graphics.h"
#include <math.h>

const int WIDTH_X = 500;
const int HEIGHT_Y = 300;

void drawShapes(int centreX, int centreY, int sideLength)
{
  int circleDiameter =
    (int)sqrt(sideLength * sideLength + sideLength * sideLength);
  int halfSideLength = sideLength / 2;
  int circleRadius = circleDiameter / 2;

  drawArc(centreX - circleRadius, centreY - circleRadius,
          circleDiameter, circleDiameter, 0, 360);
  drawRect(centreX - halfSideLength, centreY - halfSideLength,
           sideLength, sideLength );
}

int main(void)
{
  drawShapes(WIDTH_X / 2, HEIGHT_Y / 2, 200);

  return 0;
}
```

The program displays the same output as the previous version, but we can now start easily experimenting with displaying multiple copies of the circle in square in different sizes and positions. For example, using this code in the `main` method:

```
int main(void)
{
  int n;
  for (n = 30 ; n < 300 ; n += 30)
  {
    drawShapes(n, HEIGHT_Y / 2, n);
  }

  return 0;
}
```

displays this pattern of square in circle shapes:



Q1.7 Write a program to draw a representation of the BT Tower (many people still call it the Post Office Tower), this is the tall thin round tower visible from many parts of the main UCL campus.

This will need a longer sequence of drawing statements. Plan your drawing before trying to write your code.

Just be creative to answer this!

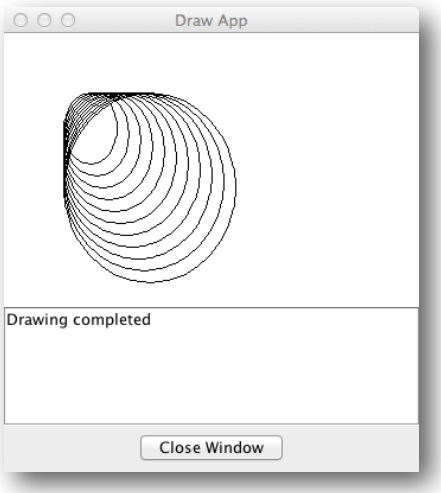
Q1.8 Write a program to draw a series of ovals of increasing size. Use `drawOval` for this.

Example Answer:

```
#include <stdio.h>
#include "graphics.h"

int main(void)
{
  drawOval(50, 50, 45, 60);
  drawOval(50, 50, 55, 70);
  drawOval(50, 50, 65, 80);
  drawOval(50, 50, 75, 90);
  drawOval(50, 50, 85, 100);
  drawOval(50, 50, 95, 110);
  drawOval(50, 50, 105, 120);
  drawOval(50, 50, 115, 130);
  drawOval(50, 50, 125, 140);
  drawOval(50, 50, 135, 150);
  drawOval(50, 50, 145, 160);
  return 0;
}
```

While this answers the question the code has unwanted duplication due to the use of a sequence of statements where a loop could be used instead, requiring only one `drawOval` function call statement. A core programming principle is to always eliminate duplication wherever and in whatever form it occurs.

```
#include <stdio.h>
#include "graphics.h"

int main(void)
{
  int n;
  for (n = 0 ; n < 110 ; n += 10)
  {
    drawOval(50, 50, 45 + n, 60 + n);
  }
  return 0;
}
```



Q1.9 Write a program to display the CS department address (see the CS web page at https://www.ucl.ac.uk/computer-science/) in a drawing. Each part of the address should appear on a separate line in the normal way.

Use `drawString` to do this.

Example Answer:

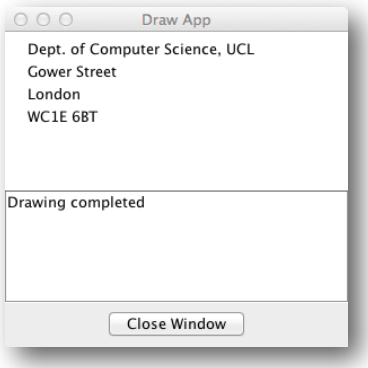
```
#include <stdio.h>
#include "graphics.h"

int main(void)
{
  drawString("Dept. of Computer Science, UCL", 20, 20);
  drawString("Gower Street", 20, 40);
  drawString("London", 20, 60);
  drawString("WC1E 6BT", 20, 80);
  return 0;
}
```

The main goal here is familiarisation with the `drawString` function. Did you work out the relationship between the coordinate given in the function call and where the string is actually displayed?

With a bit more knowledge about C and using arrays, the following program is possible:

```
#include "graphics.h"

int main(void)
{
  char *address[4] = { "Dept. of Computer Science, UCL",
                       "Gower Street",
                       "London",
                       "WC1E 6BT"
                     };
  int y;
  for (y = 0 ; y < 4 ; y++)
  {
    drawString(address[y], 20, 20 + (y * 20));
  }
  return 0;
}
```

At first sight this looks more complicated than the previous version and it is. However, it eliminates the duplication of multiple `drawString` function calls and, more usefully, makes it easy to add or remove lines from the address. You don't have to change the display coordinates of each string, instead strings can just be added or removed from the array. A better design if more actual code.



Q1.10 Create a program to draw a bar chart with labels. This will need quite a long list of drawing statements. Here is an example:

Example Answer:

```
#include <stdio.h>
#include “graphics.h"

int main(void)
{
  drawLine(40, 250, 290, 250);
  drawLine(40, 250, 40, 50);

  drawString("CDs", 50, 270);
  drawString("DVDs", 98, 270);
  drawString("Books", 146, 270);
  drawString("Clothes", 190, 270);
  drawString("Shoes", 248, 270);

  char buf[20];
  int y;
  for (y = 0 ; y < 200 ; y += 50)
  {
    sprintf(buf,"%i",y);
    drawLine(40, 250 - y, 35, 250 - y);
    drawString(buf, 10, 255 - y);
  }

  setColour(blue);
  fillRect(40, 190, 50, 60);
  setColour(green);
  fillRect(90, 110, 50, 140);
  setColour(red);
  fillRect(140, 130, 50, 120);
  setColour(yellow);
  fillRect(190, 155, 50, 95);
```

```
  setColour(magenta);
  fillRect(240, 110, 50, 140);

  return 0;
}
```

Note how the integer counter `y` in the `for` loop is converted to a string so it can be displayed. The `sprintf` function is like `printf` but writes to a string rather than the standard output (`stdout`).

Given the comments about duplicated code in some of the previous answers you should be looking at the program above and thinking that it contains duplication – so let's get rid of it! Duplication does not just mean identical lines of code but also similar lines that are doing essentially the same job, such as the `drawString`s and the `setColour/fillRect` sequence. We should also use functions to make the code more readable and introduce some structure.

As a first step the code is packaged into functions to give a better structure. Note how the main method has shrunk and is much more understandable.

```
#include <stdio.h>
#include "graphics.h"

void drawXAxis()
{
  drawLine(40, 250, 290, 250);
  drawString("CDs", 50, 270);
  drawString("DVDs", 98, 270);
  drawString("Books", 146, 270);
  drawString("Clothes", 190, 270);
  drawString("Shoes", 248, 270);
}

void drawYAxis()
{
  drawLine(40, 250, 40, 50);
  char buf[20];
  int y;
  for (y = 0 ; y < 200 ; y += 50)
  {
    sprintf(buf,"%i",y);
    drawLine(40, 250 - y, 35, 250 - y);
    drawString(buf, 10, 255 - y);
  }
}

void drawBars()
{
  setColour(blue);
  fillRect(40, 190, 50, 60);
  setColour(green);
  fillRect(90, 110, 50, 140);
  setColour(red);
  fillRect(140, 130, 50, 120);
  setColour(yellow);
  fillRect(190, 155, 50, 95);
  setColour(magenta);
```

```
  fillRect(240, 110, 50, 140);
}

int main(void)
{
  drawXAxis();
  drawYAxis();
  drawBars();
  return 0;
}
```

Next the `drawXAxis` function can be rewritten to remove all the 'magic numbers' and to make it easier to change the number of bars and labels the graph displays. `#define`s are used to declare proper constant values, so the coordinates no longer need to appear as literal values (e,g,. 20, 50 ,150) in the `drawLine` and `drawString` function calls.

```
#include <stdio.h>
#include "graphics.h"

#define XORIGIN 40
#define YORIGIN 250
#define XAXISLENGTH 250
#define BARWIDTH 50
#define XLABELYORIGIN YORIGIN + 20

char *xAxisLabels[] = {"  CDs  ", "  DVDs ", " Books ", "Clothes", " Shoes ",
NULL};

void drawXAxis()
{
  drawLine(XORIGIN, YORIGIN, XORIGIN + XAXISLENGTH, YORIGIN);
  int xPosition = XORIGIN;
  char **label = xAxisLabels;
  while (*label)
  {
    drawString(*label++, xPosition, XLABELYORIGIN);
    xPosition += BARWIDTH;
  }
}

void drawYAxis()
{
  drawLine(40, 250, 40, 50);
  char buf[20];
  int y;
  for (y = 0 ; y < 200 ; y += 50)
  {
    sprintf(buf,"%i",y);
    drawLine(40, 250 - y, 35, 250 - y);
    drawString(buf, 10, 255 - y);
  }
}
```

```
void drawBars()
{
  setColour(blue);
  fillRect(40, 190, 50, 60);
  setColour(green);
  fillRect(90, 110, 50, 140);
  setColour(red);
  fillRect(140, 130, 50, 120);
  setColour(yellow);
  fillRect(190, 155, 50, 95);
  setColour(magenta);
  fillRect(240, 110, 50, 140);
}

int main(void)
{
  drawXAxis();
  drawYAxis();
  drawBars();
  return 0;
}
```

The same process can then be repeated for the other functions to give this version:

```
#include <stdio.h>
#include "graphics.h"

#define XORIGIN 40
#define YORIGIN 250
#define XAXISLENGTH 250
#define YAXISLENGTH 200
#define BARWIDTH 50
#define XLABELYORIGIN YORIGIN + 20
#define YLABELINCREMENT 50
#define YLABELXORIGIN XORIGIN - 30
#define YAXISTICKLENGTH 5

char *xAxisLabels[] = {"  CDs  ", "  DVDs ", " Books ", "Clothes", " Shoes ",
NULL};

void drawXAxis()
{
  drawLine(XORIGIN, YORIGIN, XORIGIN + XAXISLENGTH, YORIGIN);
  int xPosition = XORIGIN;
  char **label = xAxisLabels;
  while (*label)
  {
    drawString(*label++, xPosition, XLABELYORIGIN);
    xPosition += BARWIDTH;
  }
}
```

```
void drawYAxis()
{
  drawLine(XORIGIN, YORIGIN, XORIGIN, YORIGIN - YAXISLENGTH);
  char buf[20];
  int y;
  for (y = 0 ; y < YAXISLENGTH ; y += YLABELINCREMENT)
  {
    sprintf(buf,"%i",y);
    drawLine(XORIGIN, YORIGIN - y, XORIGIN - YAXISTICKLENGTH, YORIGIN - y);
    drawString(buf, YLABELXORIGIN, YORIGIN - y);
  }
}

void drawBar(colour barColour, int position, int height)
{
  setColour(barColour);
  fillRect(XORIGIN + (BARWIDTH * position), YORIGIN - height, BARWIDTH, height);
}

void drawBars()
{
  drawBar(blue,0,60);
  drawBar(green,1,140);
  drawBar(red,2,120);
  drawBar(yellow,3,95);
  drawBar(magenta,4,140);
}

int main(void)
{
  drawXAxis();
  drawYAxis();
  drawBars();
  return 0;
}
```

The `drawBars` function still contains some duplication, which can be removed by creating another simple data structure to hold the data representing the colour and height of each bar. This gives:

```
#include <stdio.h>
#include "graphics.h"

#define XORIGIN 40
#define YORIGIN 250
#define XAXISLENGTH 250
#define YAXISLENGTH 200
#define BARWIDTH 50
#define XLABELYORIGIN YORIGIN + 20
#define YLABELINCREMENT 50
#define YLABELXORIGIN XORIGIN - 30
#define YAXISTICKLENGTH 5
```

```
char *xAxisLabels[] = {"  CDs  ", "  DVDs ", " Books ", "Clothes", " Shoes ",
NULL};

struct bar
{
  colour barColour;
  int height;
};

#define NUMBEROFBARS 5
struct bar bars[NUMBEROFBARS] = {
  {blue, 60},
  {green, 140},
  {red, 120},
  {yellow, 95},
  {magenta, 140}
};

void drawXAxis()
{
  drawLine(XORIGIN, YORIGIN, XORIGIN + XAXISLENGTH, YORIGIN);
  int xPosition = XORIGIN;
  char **label = xAxisLabels;
  while (*label)
  {
    drawString(*label++, xPosition, XLABELYORIGIN);
    xPosition += BARWIDTH;
  }
}

void drawYAxis()
{
  drawLine(XORIGIN, YORIGIN, XORIGIN, YORIGIN - YAXISLENGTH);
  char buf[20];
  int y;
  for (y = 0 ; y < YAXISLENGTH ; y += YLABELINCREMENT)
  {
    sprintf(buf,"%i",y);
    drawLine(XORIGIN, YORIGIN - y, XORIGIN - YAXISTICKLENGTH, YORIGIN - y);
    drawString(buf, YLABELXORIGIN, YORIGIN - y);
  }
}

void drawBar(colour barColour, int position, int height)
{
  setColour(barColour);
  fillRect(XORIGIN + (BARWIDTH * position), YORIGIN - height, BARWIDTH, height);
}

void drawBars()
{
  int n;
```

```
  for (n = 0 ; n < NUMBEROFBARS ; n++)
  {
    struct bar aBar = bars[n];
    drawBar(aBar.barColour, n, aBar.height);
  }
}

int main(void)
{
  drawXAxis();
  drawYAxis();
  drawBars();
  return 0;
}
```

The version above uses a C *struct* to create a type `bar` that combines two values, the colour and height of a bar.

If we review what has happened to the code in this program we can see that the data values to be displayed have been removed from being coded into statements in the function bodies, and moved into proper data structures. The functions have split the code up into more manageable units and the 'magic numbers' have been turned into constant values.

Overall the program is better organised, easier to understand and easier to modify. However, there is still more that could be done. There are always opportunities to improve your code. If the program was developed further the next step would be to move the data into a data file, so it is not coded into the source code at all.

Challenge Questions

These questions are pose challenges. Try searching on the web for information – there are lots of interesting websites and examples to be found.

Several of the questions below require the use of maths functions such as sin and cos. The standard C library provides implementations of these, see here for the documentation for the GNU C version: http://www.gnu.org/s/hello/manual/libc/index.html.

To use a maths function you should add the following line at the top of your source code file:

```
#include <math.h>
```

When you compile your code you need to add an additional *flag* -lm to tell the compiler to link the maths library code with your code.

```
gcc –o graph graph.c graphics.c –lm
```

This means that the code implementing the maths functions gets added to the executable program you are creating. If the flag is missing then gcc will give linker error messages stating that there are undefined symbols as it will not be able to locate the function definitions (code) needed.

Q1.11 Write a program to draw a sine wave. The C math library provides a sine function with this *signature*: `double sin(double x)`. This means it takes a double, or floating point, value as an argument and returns a double result. To use the `sin` function in your code, the statement

```
#include <math.h>
```

needs to appear at the top of you source code file. The `sin` function works with *radians*, so you either need to work with radians yourself or use degrees and convert to radians when calling the `sin` function (multiply by pi/180).

Double values produced by the `sin` function needs to be converted to integers, type `int`, when passed as values to drawing functions. This can be done using a `cast` expression:

```
double y = 1.23;
int x = (int)y;
```

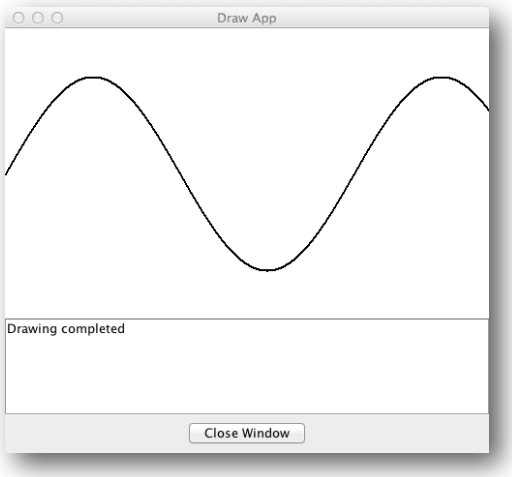
The value 1 will be assigned to x, as the digits after the decimal point are discarded.

Example Answer:

This program draws a sequence of points (rectangles of width and height both one) close enough together so that they appear to form a continuous curve.

```
#include "graphics.h"
#include <math.h>

#define PI 3.14159265

int main(void)
{
  int width = 500;
  int height = 300;
  int x;
  for (x = 0 ; x < width*2 ; x++)
  {
    double y = (height / 2) - sin(x*(PI/180)) * 100;
    drawRect(x,(int)y,1,1);
  }
  return 0;
}
```

Note the use of a `#define` to give the value of pi. This is a mechanism in C to define constant values. In fact, the maths library already provides a `#define` to give the value of pi, which is called `M_PI`. The program displays this:

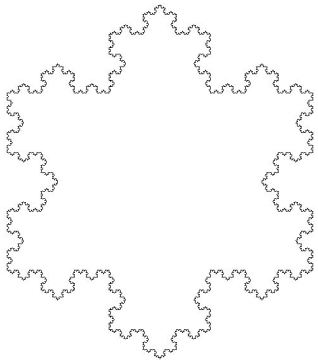


Q1.12 Write a program to draw a fractal shape using a recursive algorithm, like this one:

Note that the shape of the sub-elements reflect the shape of the whole.

Example Answer:

This shape is what is commonly known as a Koch snowflake. What you see is actually a fractal curve first documented by the Swedish mathematician Helge von Koch in 1904. Wikipedia has some good articles on the Koch snowflake, fractals and related subjects.



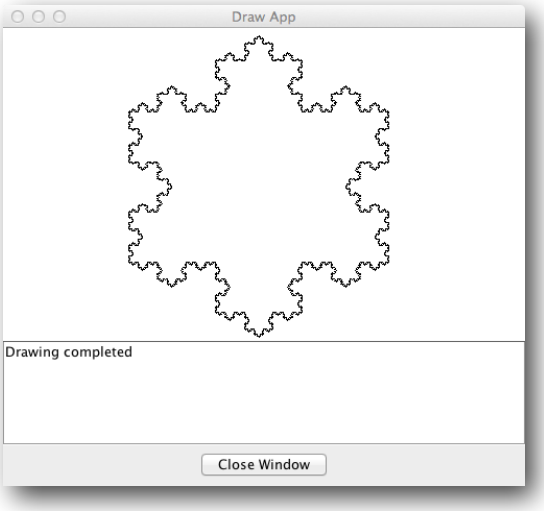
a) Version 1

The program below draws a snowflake by calculating the line segments needed to draw the boundary line. As fractal shapes like this are recursive structures, using a recursive function (a function that calls itself) gives a good solution.

```
#include <math.h>
#include "graphics.h"

double x,y;

double convertToRadians(int angle)
{
  return (M_PI / 180) * angle;
}

// This recursive function draws the line segments that make up the core
// shape of the fractal.
void segment(int level, double lengthOfSide, double angle)
{
  // The base case (where recursion stops) draws a single line with the required
  // position and angle. Only the lines at this level are actually drawn.
  if (level == 0)
  {
    double newX = (cos(convertToRadians(angle)) * lengthOfSide) + x;
    double newY = (sin(convertToRadians(angle)) * lengthOfSide) + y;
    drawLine((int)x, (int)y, (int)newX, (int)newY);
    x = newX;
    y = newY;
  }
  else
  {
    // The recursive calls create the fractal structure by calculating four
    // lines, each one third of the length of the current line. The current line
    // is effectively folded up in the fractal pattern.
    double newLength = lengthOfSide / 3;
    segment(level - 1, newLength, angle);
    segment(level - 1, newLength, angle - 60);
    segment(level - 1, newLength, angle + 60);
    segment(level - 1, newLength, angle);
  }
}
```

```
// The basic snowflake is made of three fractal lines
// drawn round an equilateral triangle. This function sets up
// the basic shape and then recursion is used to repeat the same
// basic structure on progressively smaller scales (the self
// similarity).
void snowflake(int level, double lengthOfSide)
{
  segment(level, lengthOfSide, 0);
  segment(level, lengthOfSide, 120);
  segment(level, lengthOfSide, 240);
}

int main(void)
{
  x = 120.0;
  y = 80.0;
  snowflake(5,250.0);
}
```

This displays:

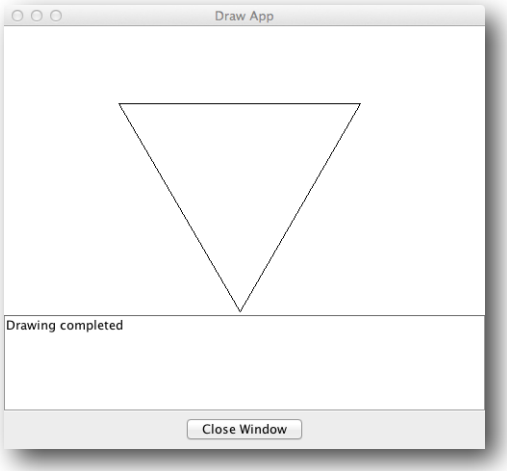


To get a better idea of what is going on, the `main` function can be modified to allow the depth of recursion to be specified:

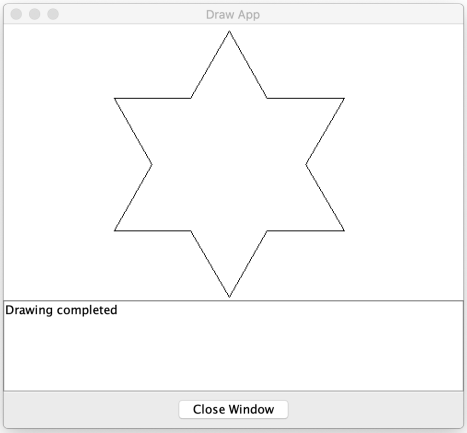
```
#include <stdio.h> // Add this now as terminal IO is used
int main(void)
{
  x = 120.0;
  y = 80.0;
  int complexity = 5;
  fprintf(stderr, "Enter depth of recursion: ");
  scanf("%i", &complexity);
  snowflake(complexity, 250.0);
}
```

Note that `fprintf` with `stderr` is used to display the input prompt in the terminal window. If the normal printf is used the prompt would be displayed on the standard output, `stdout`, and not be seen as the standard output is redirected into the `drawapp` program.

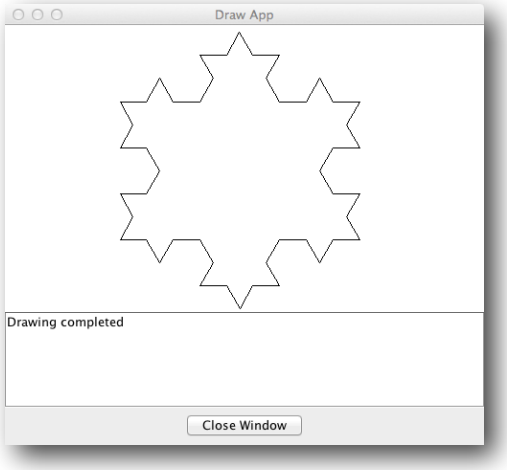
If the depth of recursion is set to zero you see the basic triangle that the snowflake is based on:



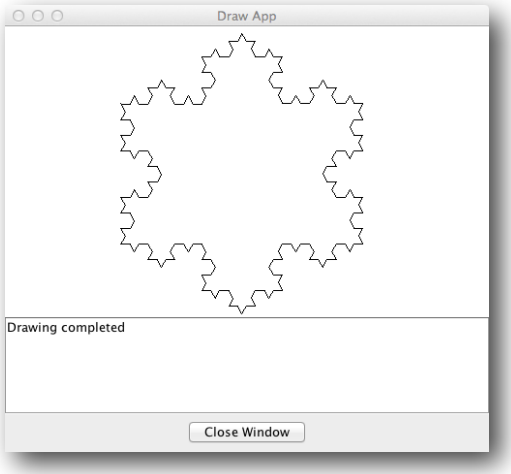
With depth one you can see how the fractal folding of the edges works, so that a triangle is folded into the middle of each side of the triangle:



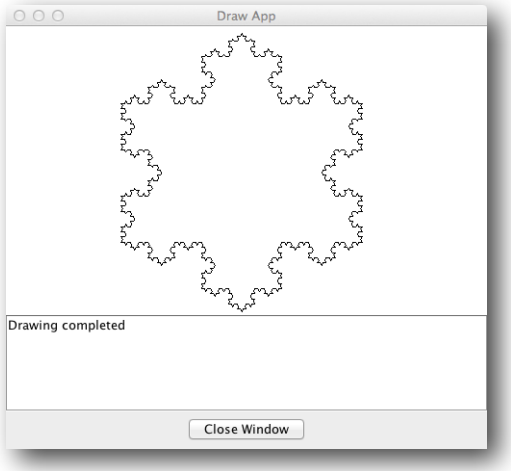
Depth two recursion then repeats the process on all the sides; the self-similarity effect:



At depth three recursion the snowflake shape becomes more obvious:



At depth four you have a snowflake:



The first image showing the output from the program was at recursive depth 5. At greater depths of recursion the line segments lengths become too small to display so the appearance of the snowflake looks the same. Greater depths of recursion will also take longer and longer to compute (try fifteen or greater if you want if you want to stare at the screen doing nothing for a long time!).

b) Version 2

The first version demonstrates a working solution and could be considered ‘good enough’.

BUT…

There are things I don’t like, in particular the way that the variables `x` and `y` are being used. They are effectively global variables (actually file scope variables), meaning that they are declared outside of any functions and can be used by any function declared in the same source code file. Sharing variables between functions like this can easily lead to problems in the future if the code were to be developed further as part of a larger program. The use of global variables is a strong indicator of a weak design and they should be removed where possible. We will revisit this whole issue in more detail when we start Java programming, though, as it is quite a bit more complex than it might seem.

In addition, using `x` and `y` like this is compromising the use of recursion. Each recursive call is sharing the same two variables, whereas the values needed within each call should be local to that call only. In other words `x` and `y` should be passed as parameters, not shared.

This version of the program stops using the global `x` and `y`, and instead passes the values as parameters to the recursive call, and each call returns the new `x` and `y` (the end of the line segment it has just drawn). The tricky part of this is that the recursive function has to return two values, the `x` and `y`, but C functions can return only a single value.

This is resolved by using a C struct. A struct is a structure that can be treated as a single value but actually contains multiple values. This is the struct definition to hold a pair of values representing a coordinate, which is what `x` and `y` actually are:

```
struct coordinate
{
  double x;
  double y;
}
```

This defines a type called '`struct coordinate`' that can be used to declare variables or function return types, and a value of the type is composed of two `double` variables. A struct definition is usually used with a `typedef` statement to provide a more convenient name for the struct type. In this case it looks like:

```
typedef struct coordinate
{
  double x;
  double y;
} coordinate;
```

This states that the type `struct coordinate` (in italics) is given an alias, or alternate name, of just `coordinate` (in bold). Given this we can define a variable of type `coordinate` and use it:

```
coordinate point;
point.x = 10.0;
point.y = 20.0;
```

This also illustrates the use of the dot ('`.`') operator to access the values held in the struct.

Defining and initialising a struct can be shortened to:

```
coordinate point = {10.0, 20.0};
```

Using structs the recursive function can be rewritten as you see below. The opportunity has also been taken to rename a number of variables, re-order the function parameters lists, and use `for` loops instead of a sequence of similar function calls.

```
#include <math.h>
#include <stdio.h>
#include "graphics.h"

typedef struct coordinate
{
  double x;
  double y;
} coordinate;

double convertToRadians(int angle)
{
  return (M_PI / 180) * angle;
}

// This recursive function draws the line segments that make up the core
// shape of the fractal.
coordinate segment(coordinate start, int sideLength, double angle,  int depth)
```

```
{
  // The base case (where recursion stops) draws a single line with the required
  // position and angle. Only the lines at this depth are actually drawn.
  if (depth == 0)
  {
    coordinate endOfLine =
    {
      (cos(convertToRadians(angle)) * sideLength) + start.x,
      (sin(convertToRadians(angle)) * sideLength) + start.y
    };
    drawLine((int)start.x, (int)start.y, (int)endOfLine.x, (int)endOfLine.y);
    return endOfLine;
  }
  else
  {
    // The recursive calls create the fractal structure by calculating four
    // lines, each one third of the length of the current line. The current line
    // is effectively folded up in the fractal pattern.
    double newLength = sideLength / 3;
    coordinate nextStart = start;
    double angles[] = {angle, angle - 60, angle + 60, angle};
    for (int n = 0 ; n < 4 ; n++)
    {
      nextStart = segment(nextStart, newLength, angles[n], depth - 1);
    }
    return nextStart;
  }
}

// The basic snowflake is made of three fractal lines
// drawn round an equilateral triangle. This function sets up
// the basic shape and then recursion is used to repeat the same
// basic structure on progressively smaller scales (the self
// similarity).
void snowflake(coordinate start, int sideLength, int depth)
{
  coordinate nextStart = start;
  for (int angle = 0 ; angle < 360; angle += 120)
  {
    nextStart = segment(nextStart, sideLength, angle, depth);
  }
}

int getDepth()
{
  int depth;
  fprintf(stderr, "Enter the maximum depth (0-7): ");
  scanf("%i", &depth);
  return depth;
}

int main(void)
```

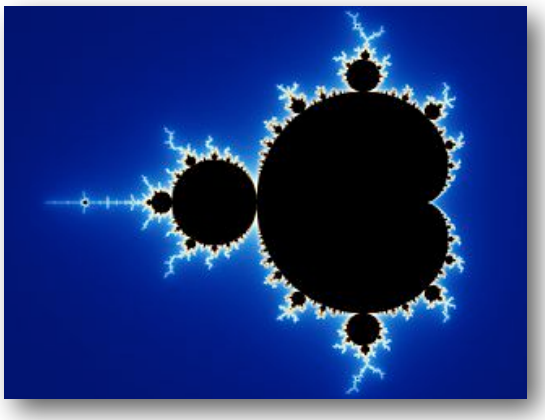
```
{
  setWindowSize(800,800);
  coordinate start = {120.0, 200.0};
  int depth = getDepth();
  int sideLength = 550;
  snowflake(start, sideLength, depth);
}
```

For future iterations of the code, the recursive function `segment` is a bit too long and a lot of the code could be extracted out into two supporting functions called from the function. Also a better name than `segment` could be found.

Q1.13 Write a program to display the Mandelbrot set in colour.

Drawing a basic Mandelbrot is actually quite straightforward and needs only a few lines of code. When working on this question get a basic working program first and then extend it.

A Mandelbrot looks like this:



Search the web for an explanation of what a Mandelbrot is and how to draw one.

Hint: try Wikipedia.

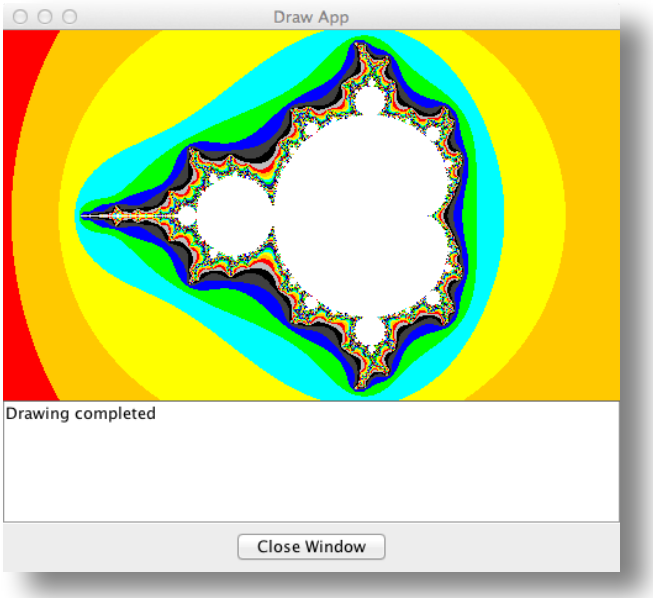
Example Answer:

As noted in the question, the code for calculating the values in the Mandelbrot set is straightforward. The harder part is mapping values to colours so that the set can be displayed. The program below uses an array of colour values, using those defined in `graphics.h`, and selects each colour in sequence from the array when a point in the set is drawn. When the end of the colour sequence is reached it simply wraps around to the beginning.

```
#include "graphics.h"

int main(void)
{
  const int width = 500;
  const int height = 300;
  const int iterations = 128;
  const int limit = 6;
  const double yOffset = 1.2;
  const double xOffset = 2.5;
  const double scale = 0.008;
  const int numberOfColours = 9;
  const colour colours[] =
```

```
    {lightgray, red, orange, yellow, cyan, green, blue, darkgray, black};

  int column;
  int row;
  for (column = 0 ; column < width ; column++)
  {
    for (row = 0 ; row < height ; row++)
    {
      double x = 0.0;
      double y = 0.0;
      int count;
      for (count = 0 ; count < iterations ; count++)
      {
        if ((x * x + y * y) > limit)
        {
          setColour(colours[count % numberOfColours]);
          drawLine(column, row, column, row);
          break;
        }
        double newy = 2 * x * y + row * scale - yOffset;
        x = x * x - y * y + column * scale - xOffset;
        y = newy;
      }
    }
  }
}
```

The program displays this:



The colours can be changed by modifying the `colours` array.