COMP0002 Programming Principles

Programming Notes and Exercises 2

Example Answers for a Selection of Questions

Core Questions

Q2.1 Write a program using a while loop to display the thirteen times table, like this:

```
1 * 13 = 13
2 * 13 = 26
3 * 13 = 39
```

and so on.

Note: the multiplication operator is *.

Example Answer:

This program will print out the table:

```
#include <stdio.h>

int main(void)
{
  int count = 1;
  while (count < 13)
  {
    printf("%i * 13 = %i\n", count, count * 13);
    count++;
  }
  return 0;
}
```

And the table looks like this:

```
1 * 13 = 13
2 * 13 = 26
3 * 13 = 39
4 * 13 = 52
5 * 13 = 65
6 * 13 = 78
7 * 13 = 91
8 * 13 = 104
9 * 13 = 117
10 * 13 = 130
11 * 13 = 143
12 * 13 = 156
```

The formatting of the printed table can be improved by getting the columns to line up. This can be done by adding field width values to the conversion characters in the `printf` formatting string. Or, in plain English(!), you can specify how many characters to use to display each number. For example, using `"%2i"` with the 2 between the '`%`' and '`i`' means use two characters to display the integer number. Any character positions not needed by the number are filled with spaces. By default the spaces fill, or 'pad', to the left but inserting a '`-`' pads to the right. Changing the `printf` statement to:

```
printf("%2i * 13 = %-3i\n", count, count * 13);
```

results in this table being displayed:

```
 1 * 13 = 13
 2 * 13 = 26
 3 * 13 = 39
 4 * 13 = 52
 5 * 13 = 65
 6 * 13 = 78
 7 * 13 = 91
 8 * 13 = 104
 9 * 13 = 117
10 * 13 = 130
11 * 13 = 143
12 * 13 = 156
```

The while loop in the program uses a variable called `count`, which is incremented at the end of the loop body. A common practice in C programming is to put increments or decrements into the boolean expression controlling a loop. This would give:

```
#include <stdio.h>

int main(void)
{
  int count = 0;
  while (count++ < 12)
  {
    printf("%2i * 13 = %-3i\n", count, count * 13);
  }
  return 0;
}
```

Notice that two other changes were needed to get the counting correct. Firstly the variable `count` is now initialised to `0` and secondly the boolean expression checks whether `count` is less than `12`. Remember that the ++ following the variable is the post-increment version, meaning that the increment is performed after the boolean expression is evaluated. Statements inside the loop body then see the value of `count` *after* it has been incremented.

Q2.2 Repeat Q2.1 using a for loop.

Example Answer:

A for loop version looks like this:

```
#include <stdio.h>

int main(void)
{
  int count;
  for (count = 1 ; count < 13 ; count++)
```

```
  {
    printf("%2i * 13 = %-3i\n", count, count * 13);
  }
  return 0;
}
```

The main issue here is to use the for loop syntax correctly, with the three expressions in parentheses to initialise the counter, provide a boolean expression to determine if the loop body should be evaluated, and the post loop body expression evaluated each time after the loop body is evaluated. Also care has to be taken to get the counting correct, so `count` is initialised to `1` and compared to `13`.

The increment expression could be written as `++count`, using the pre-increment version of ++. In this loop it would make no difference the result. Depending on the quality of the code generated by your C compiler, `++count` might be very slightly more efficient.

Q2.3 Repeat Q2.1 using do loop.

Example Answer:

```
#include <stdio.h>

int main(void)
{
  int count = 1;
  do
  {
    printf("%2i * 13 = %-3i\n", count, count * 13);
  } while (++count < 13);
  return 0;
}
```

Remember that a do loop body is always evaluated at least once, as the boolean test is done after the body is evaluated. Here using `++count` rather than `count++` does make a difference.

Q2.4 Write a program to display the following:

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

You may display only one character at a time. You cannot have a statement such as:

```
printf(“*****”);
```

to output an entire line at once. A loop must be used to write characters one at a time. Space characters should be output to display the spaces inside the shape.

Hint: nested loops.

Example Answer:

```
#include <stdio.h>

int main(void)
{
  int row, column;
```

```
  for (row = 0 ; row < 4 ; row++)
  {
    if ((row == 0) || (row == 3))
    {
      for (column = 0 ; column < 5 ; column++)
      {
        printf("*");
      }
    }
    else
    {
      printf("*");
      for (column = 0 ; column < 3 ; column++)
      {
        printf(" ");
      }
      printf("*");
    }
    printf("\n");
  }
  return 0;
}
```

This illustrates a common strategy for displaying these patterns of having an outer loop counting through the rows and one or more nested loops counting through columns.

The example above is formatted using a style that spreads the code across a fairly large number of lines. By modifying the style conventions the number of lines can be reduced. For example, this style allows the for loop bodies to be on the same line as the for keyword:

```
#include <stdio.h>

int main(void)
{
  int row, column;
  for (row = 0 ; row < 4 ; row++)
  {
    if ((row == 0) || (row == 3))
    {
      for (column = 0 ; column < 5 ; column++) { printf("*"); }
    }
    else
    {
      printf("*");
      for (column = 0 ; column < 3 ; column++) { printf(" "); }
      printf("*");
    }
    printf("\n");
  }
  return 0;
}
```

Moving the opening braces to the end of the line compacts the code a bit more:

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

```
int main(void) {
  int row, column;
  for (row = 0 ; row < 4 ; row++) {
    if ((row == 0) || (row == 3)) {
      for (column = 0 ; column < 5 ; column++) { printf("*"); }
    }
    else {
      printf("*");
      for (column = 0 ; column < 3 ; column++) { printf(" "); }
      printf("*");
    }
    printf("\n");
  }
  return 0;
}
```

The position of opening braces (end of line or next line) is a long running “debate” between programmers. It is the cause of many bitter arguments, has probably started several wars and can completely disrupt a project. Whichever style you choose, be consistent. No specific style is the best and each has trade-offs between line count and readability. If working in a team, use the team style.

Notice that a star is always displayed at the start and end of a line for this shape. A further version of the program could be this:

```
#include <stdio.h>

int main(void) {
  int row, column;
  for (row = 0 ; row < 4 ; row++) {
    printf("*");
    if ((row == 0) || (row == 3)) {
      for (column = 0 ; column < 3 ; column++) { printf("*"); }
    }
    else {
      for (column = 0 ; column < 3 ; column++) { printf(" "); }
    }
    printf("*");
    printf("\n");
  }
  return 0;
}
```

If you are prepared to omit the braces where the body of an `if` or `else` is a *single* statement, you can remove two more lines from the code:

```
#include <stdio.h>

int main(void) {
  int row, column;
  for (row = 0 ; row < 4 ; row++) {
    printf("*");
    if ((row == 0) || (row == 3))
      for (column = 0 ; column < 3 ; column++) { printf("*"); }
    else
      for (column = 0 ; column < 3 ; column++) { printf(" "); }
```

```
    printf("*");
    printf("\n");
  }
  return 0;
}
```

Does this increase readability and understandability of the code? It does break the defensive programming rule of always including braces.

Q2.5 Write a program to display the following:

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

You may display only one character at a time.

Example Answer:

```
#include <stdio.h>

int main(void)
{
  int row, column;
  for (row = 0 ; row < 6 ; row++)
  {
    for (column = 0 ; column < row ; column++) { printf(" "); }
    for (column = 0 ; column < 6 - row ; column++) { printf("*"); }
    printf("\n");
  }
  return 0;
}
```

The key here is to identify the relationship between the row number (counting from zero don't forget) and the number of stars or spaces needed on each row. The number of spaces is simply the same as the row number while the expression `6 - row` gives the numbers of stars.

An alternative approach is to use two variables to store the number of spaces and stars to output on each row. As the outer loop counts through the rows, the number of spaces is incremented and the number of stars decremented.

```
#include <stdio.h>

int main(void)
{
  int spaces = 0;
  int stars = 6;
  int row,column;
  for (row = 0 ; row < 6 ; row++)
  {
    for (column = 0 ; column < spaces ; column++) { printf(" "); }
    for (column = 0 ; column < stars ; column++) { printf("*"); }
    spaces++;
    stars--;
```

```
    printf("\n");
  }
  return 0;
}
```

This solution is a bit more verbose and needs two more variables. Whether it is better or not really comes down to a matter of opinion and whether you feel it is easier to understand or not.

Q2.6 Write a program to display the following:

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

You may display only one character at a time.

Example Answer:

The key here is knowing how many spaces to print at the start of each row. The number of spaces can be stored in a variable that is updated at the end of each iteration.

```
#include <stdio.h>

int main(void)
{
  int spaces = 0;
  int row, column;

  for (row = 0 ; row < 9 ; row++)
  {
    for (column = 0 ; column < spaces ; column++) { printf(" "); }
    for (column = 0 ; column < 5 ; column++) { printf("*"); }
    printf("\n");
    int step = ( row == 0 || row == 7 ) ? 2 : 1;
    spaces = ( row < 4 ) ? spaces + step : spaces - step;
  }
  return 0;
}
```

Notice that this program makes use of the ternary conditional operator `?:`. Ternary means that the operator has three arguments (a binary operator has two, a unary operator one). The conditional operator provides another way of expressing if-else and is interpreted like this:

```
(boolean expression) ? value if true : value if false
```

If the boolean expression is true the expression following the '`?`' is evaluated and returned as the value of the operation. If false, the value of the expression following the '`:`' is returned. Hence, the statement:

```
int step = ( row == 0 || row == 7 ) ? 2 : 1;
```

is equivalent to:

```
int step;
```

```
if (row == 0 || row == 7)
{
  step = 2;
}
else
{
  step = 1;
}
```

Q2.7 Write a program to display the following:

```
********
*      *
* #### *
* #  # *
* #### *
*      *
********
```

You may print only one character at a time.

Hint: Find out about the if statement.

Example Answer:

There are four different row patterns to print out, so the basic strategy is to count through each row and use an `if` statement to select the correct pattern based on the row number. `If` statements can be used to do this but the `switch` statement leads to a cleaner solution:

```
#include <stdio.h>

int main(void)
{
  int row, column;
  for (row = 0 ; row < 7 ; row++)
  {
    printf("*");
    switch(row)
    {
      case 0:
      case 6:
        for (column = 1 ; column < 7 ; column++) { printf("*"); }
        break;
      case 1:
      case 5:
        for (column = 1 ; column < 7 ; column++) { printf(" "); }
        break;
      case 2:
      case 4:
        printf(" ");
        for (column = 2 ; column < 6 ; column++) { printf("#"); }
        printf(" ");
        break;
      case 3:
        printf(" ");
```

```
        printf("#");
        for (column = 3 ; column < 5 ; column++) { printf(" "); }
        printf("#");
        printf(" ");
        break;
    }
    printf("*");
    printf("\n");
  }
  return 0;
}
```

The `switch` statement is another form of selection statement like `if-else`. It allows one from many statement sequences to be selected for evaluation, whereas `if-else` is limited to selected one out of two statement sequences for evaluation.

To evaluate a `switch` statement the value of the integer expression in parentheses following the `switch` keyword is used to select a labelled *case* inside the `switch` statement body. For example, in the program above, if `row` has the value `4`, a jump will be made to `case 4:` and the statements following the label will be evaluated. The `break` keyword causes a jump to the statement following the `switch` statement to avoid falling through and evaluating another case in error.

Several case labels can label the same statement sequence, so `case 2` and `case 4` in the example answer both result in the same pattern being displayed.

Q2.8 Write a program to display the following:

```
*#*#*#
#*#*#*
*#*#*#
#*#*#*
*#*#*#
#*#*#*
```

You may display only one character at a time.

Example Answer:

The pattern here is an alternating sequence of '*' and '#' characters, so printing a line is just a question of toggling between the two characters for each column. This just leaves selecting the correct character at the start of each line.

```
#include <stdio.h>

int main(void)
{
  int row;
  int column;
  for (row = 0 ; row < 6 ; row++)
  {
    char c = (row % 2) ? '#' : '*';
    for (column = 0 ; column < 6 ; column++)
    {
      printf("%c",c);
      c = (c == '*') ? '#' : '*';
    }
```

```
    printf("\n");
  }
  return 0;
}
```

The expression `row % 2` is used to determine what character a line starts with. Remember that in C an expression that evaluates to zero is false and non-zero true. Hence, if `row % 2` returns zero, or false, a line starts with '*'.

Q2.9 Write a program to display the following:

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

You may print only one character at a time.

Example Answer:

```
#include<stdio.h>

int main(void)
{
  int row, column, spaces = 0;
  printf("*");
  printf("\n");
  for (row = 1 ; row < 8 ; row++)
  {
    printf("*");
    for (column = 0 ; column < spaces ; column++) { printf(" "); }
    spaces = (row > 3) ? spaces-1 : spaces+1;
    printf("*");
    printf("\n");
  }
  printf("*");
  printf("\n");
  return 0;
}
```

The first and last lines are treated as separate cases by just printing a star followed by a newline. The rest of the lines all consist of a star, followed by zero or more spaces, followed by a star and the main issue is to keep track of the number of spaces. This is done using the technique seen before of using a variable to keep track of the number of characters to print, and then incrementing or decrementing it as needed.

Q2.10 Write a program to display the following:

```
*******
#*    *
##*   *
# #*  *
```

```
#  #* *
#   #**
#    #*
#######
```

You may display only one character at a time.

Example Answer:

```
#include <stdio.h>

int main(void)
{
  int row, column;
  for(row = 0 ; row < 8 ; row++)
  {
    for(column = 0 ; column < 7 ; column++)
    {
      if (row == 0)
      {
        printf("*");
      }
      else
      {
        if (row == 7)
        {
          printf("#");
        }
        else
        {
          if (row == column)
          {
            printf("*");
          }
          else
          {
            if(column == row-1)
            {
              printf("#");
            }
            else
            {
              if (column == 0)
              {
                printf("#");
              }
              else
              {
                if (column == 6)
                {
                  printf("*");
                }
                else
                {
```

```
                printf(" ");
              }
            }
          }
        }
      }
    }   // Yuk, look at all these closing braces for the nested if statements
    printf("\n");
  }
  return 0;
}
```

Well, this code works but it's pretty horrible. All those nested if statements make the code hard to read and understand. Experienced programmers will say that this code smells; a code smell is used to describe any piece of code that is sub-standard. Smelly code should always be fixed using a process called *refactoring*[1].

The strategy that will be followed to refactor this code is to make use of *functions*. Small sections of code will be extracted into functions to eliminate the `if` statements and give the program a much better structure. The functions will be kept short and given descriptive names to make them easier to read. Here is the result of the first few stages of splitting the program into functions by using the Extract Function refactoring:

```
#include <stdio.h>

void printSecondLine()
{
  int n;
  printf("#");
  printf("*");
  for (n = 0 ; n < 4 ; n++) { printf(" "); }
  printf("*");
  printf("\n");
}

void printLastButOneLine()
{
  int n;
  printf("#");
  for (n = 0 ; n < 4 ; n++) { printf(" "); }
  printf("#");
  printf("*");
  printf("\n");
}

void printMiddleLine(int line)
{
  int n;
  printf("#");
  for (n = 0 ; n < line - 2 ; n++) { printf(" "); }
  printf("#");
  printf("*");
```

[1] Doing refactoring properly requires a full set of test code, so this is just a basic example.

```
  for (n = 0 ; n < 5 - line ; n++) { printf(" "); }
  printf("*");
  printf("\n");
}

int main(void)
{
  int row, column;
  for (column = 0 ; column < 7 ; column++) { printf("*"); }
  printf("\n");
  printSecondLine();
  for (row = 2 ; row < 6 ; row++) { printMiddleLine(row); }
  printLastButOneLine();
  for (column = 0 ; column < 7 ; column++) { printf("#"); }
  printf("\n");
  return 0;
}
```

This refactoring was actually done one function at a time with the end result as shown. The strategy was to create a function for each different pattern needed in the shape, except for the first and last rows. Three patterns were identified:

- one hash, two stars plus spaces.
- two hashes, two stars plus spaces.
- two hashes, one star plus spaces.

These correspond to the patterns that most of the if statements were looking for. But notice that *all the if statements have now disappeared* and choosing suitable names for the functions helps understand what the code is doing.

A good start but more can be done. Look at all the `for` loops. Except for the second loop in the main function, all the `for` loops are doing the same thing – printing the same character zero or more times. Duplication!!! Eliminate duplicated code[2].

We can apply the Extract Function refactoring again to extract a single function to print zero or more characters. The program now looks like this:

```
#include <stdio.h>

void printChars(char c, int count)
{
  int n;
  for (n = 0 ; n < count ; n++) { printf("%c",c); }
}

void printSecondLine()
{
  printf("#");
  printf("*");
  printChars(' ', 4);
  printf("*");
  printf("\n");
}
```

[2] Remember that duplication includes similar code, not just identical code.

```
void printLastButOneLine()
{
  printf("#");
  printChars(' ', 4);
  printf("#");
  printf("*");
  printf("\n");
}

void printMiddleLine(int line)
{
  printf("#");
  printChars(' ', line - 2);
  printf("#");
  printf("*");
  printChars(' ', 5 - line);
  printf("*");
  printf("\n");
}

int main(void)
{
  int row;
  printChars('*', 7);
  printf("\n");
  printSecondLine();
  for (row = 2 ; row < 6 ; row++) { printMiddleLine(row); }
  printLastButOneLine();
  printChars('#', 7);
  printf("\n");
  return 0;
}
```

The next step is to clean up the `main` function by extracting some more functions to deal with the first, last and middle lines.

```
#include <stdio.h>

void printChars(char c, int count)
{
  int n;
  for (n = 0 ; n < count ; n++) { printf("%c",c); }
}

void printFirstLine()
{
  printChars('*', 7);
  printf("\n");
}

void printSecondLine()
{
  printf("#");
```

```
  printf("*");
  printChars(' ', 4);
  printf("*");
  printf("\n");
}

void printMiddleLine(int line)
{
  printf("#");
  printChars(' ', line - 2);
  printf("#");
  printf("*");
  printChars(' ', 5 - line);
  printf("*");
  printf("\n");
}

void printMiddleLines()
{
  int row;
  for (row = 2 ; row < 6 ; row++) { printMiddleLine(row); }
}

void printLastButOneLine()
{
  printf("#");
  printChars(' ', 4);
  printf("#");
  printf("*");
  printf("\n");
}

void printLastLine()
{
  printChars('#', 7);
  printf("\n");
}

int main(void)
{
  printFirstLine();
  printSecondLine();
  printMiddleLines();
  printLastButOneLine();
  printLastLine();
  return 0;
}
```

Look at what has happened to the `main` function. It is much more readable as it consists of a sequence of function calls to functions with names that can be read in order to understand what the sequence does (ignore the `return` statement, it is just infrastructure that has to be present).

Choosing good function and variable names is important. A name should make the intent of purpose of the thing being named as clear as possible. It can be hard to find the right name, so be prepared to change names as you think of better names.

But...

Doesn't the last program still seem too long and complex? OK, it's an artificial problem to be solved, and there is the constraint that characters have to be printed one at a time, but isn't there a simpler way of solving the problem?

Yes, look at this:

```
#include <stdio.h>

void printLine(char c[], int count)
{
  int n;
  for (n = 0 ; n < count ; n++) { printf("%c",c[n]); }
  printf("\n");
}

int main(void)
{
  printLine("*******", 7);
  printLine("#*    *", 7);
  printLine("##*   *", 7);
  printLine("# #*  *", 7);
  printLine("#  #* *", 7);
  printLine("#   #**", 7);
  printLine("#    #*", 7);
  printLine("#######", 7);

  return 0;
}
```

This meets the constraint of printing one character at a time. It lifts the level of abstraction from thinking about different patterns of characters to thinking about sequences of lines.

Moreover, this version could be used to print all the other shapes in the preceding questions! All the special cases, `if` and `switch` statements, and other code disappears. You don't even need to really think about how to display a shape or look for patterns...

Why didn't you think of this earlier[3], before you wrote all that code to display specific shapes :-)

Q2.11 Write the following programs:

a) To print the squares of the numbers from 1 to 100.

b) To print the squares of the even numbers between 1 and 101.

c) To print the prime numbers between 1 and 100.

Example Answer:

a)

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

[3] If you did think of this or something similar, well done!

```
int main(void)
{
  int n;
  for (n = 1 ; n < 101 ; n++)
  {
    printf("%d^2 = %d\n", n, n*n);
  }
  return 0;
}
```

b)

```
#include <stdio.h>

int main(void)
{
  int n;
  for (n = 2 ; n < 101 ; n += 2)
  {
    printf("%d^2 = %d\n", n, n*n);
  }
  return 0;
}
```

c)

Finding prime numbers is a popular programming exercise and a wide range of solutions have been published. The program below implements a basic algorithm that simply tries to divide each candidate number by all the numbers less than it (excluding 1). It looks at only odd numbers and can be further optimised in several ways, for example by limiting the maximum divisor checked to the square root of the candidate number.

```
#include <stdio.h>

int main(void)
{
  int n;
  for (n = 1 ; n < 100 ; n += 2)
  {
    int isPrime = 1;
    int y;
    for (y = 3 ; y < n ; y++)
    {
      if ((n % y) == 0)
      {
        isPrime = 0;
        break;
      }
    }
    if (isPrime)
    {
      printf("%d\n", n);
    }
  }
  return 0;
}
```

Below is a second program using the Sieve of Eratosthenes approach to finding prime numbers (see Wikipedia for a good explanation http://en.wikipedia.org/wiki/Sieve_of_Eratosthenes). The array `numbers` is used to record whether each value between 2 and 100 is prime or not (0 indicates prime, -1 not), with suitable adjustment to take into account that array indexing starts from zero.

```
#include <stdio.h>

int main(void)
{
  int numbers[100];
  int divisor;
  for (divisor = 2 ; divisor < 51 ; divisor++)
  {
    int n = divisor + divisor;
    while (n < 101)
    {
      numbers[n-1] = -1;
      n += divisor;
    }
  }

  int x;
  for (x = 1 ; x < 100 ; x++)
  {
    if (numbers[x] != -1)
    {
      printf("%d\n", x+1);
    }
  }

  return 0;
}
```

Drawing Question

Q2.12 Write a drawing program that draws a graph showing the curves y=sin(x), y=cos(x) and y = tan(x). Include properly labelled axes, to look like the example below:

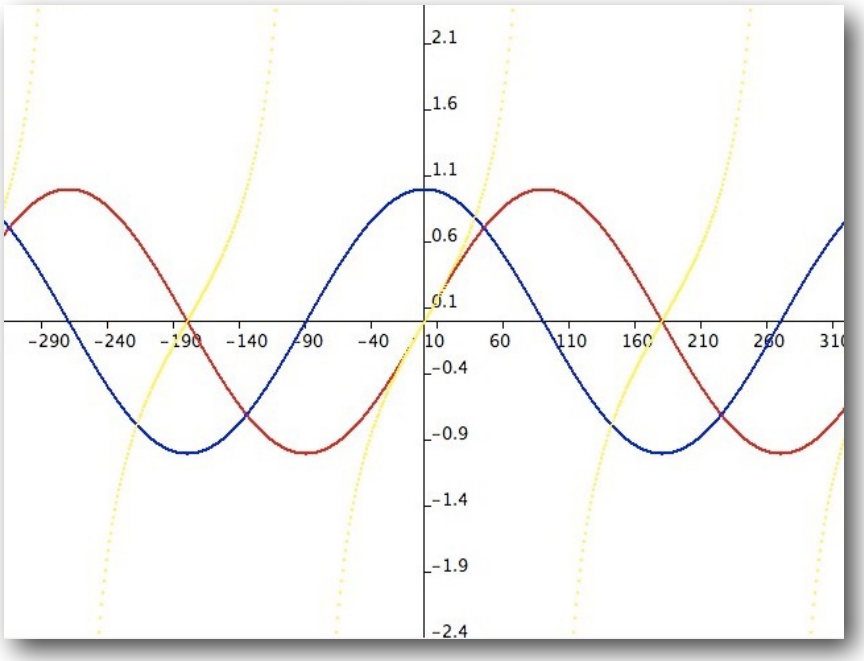
Choose suitable ranges for the axes of the graph to show the curves to the best effect.

Hints: Break the drawing of the graph down into a series of steps: draw the x-axis, label the x-axis, draw the y-axis, label the y-axis, etc. Use loops to draw the curves using points. A point can be drawn using a 1x1 rectangle. Remember that on the window co-ordinate system (0,0) is top left, so you will have to take care when you compute the coordinates of lines and points.

How do you compute Sin, Cos, Tan? See the functions in the standard C library. For example, double sin(double) will return the sin of double value (note value needs to be specified in radians or convert degrees to radians).

Example Answer:

This program requires some planning in order to organise the code. The best way to structure the program is to have a collection of functions, where each function is responsible for drawing a part of the overall diagram. The functions include those to draw the axes, label each axis and draw each different kind of curve. A control function is also needed to call each of the functions that draw each part of the diagram. In addition, decisions need to be made on the scaling of the diagram onto the pixel-based drawing area.



The program listed below draws a version of the diagram (with the labelling of the axes made rather more sensible than in the example diagram shown in the question!):

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

#define PI 3.14159265
#define PIXELWIDTH 500
#define PIXELHEIGHT 300
#define PIXELMIDY (PIXELHEIGHT / 2)
#define PIXELMIDX (PIXELWIDTH / 2)

void drawAxes(void);
void labelXAxis(int);
void labelYAxis(int);
void drawGraph(void);
void plotSin(int,int);
void plotCos(int,int);
void plotTan(int,int);

void drawAxes(void)
{
  drawLine(0, PIXELMIDY, PIXELWIDTH, PIXELMIDY) ;
  drawLine(PIXELMIDX,0,PIXELMIDX,PIXELHEIGHT) ;
}

void labelXAxis(int limit)
{
  char buffer[10];
  int n = -limit;
  int gap = PIXELWIDTH / (limit * 2);
  int xPosition;

  for(xPosition = 0 ; xPosition < PIXELWIDTH ; xPosition += gap)
```

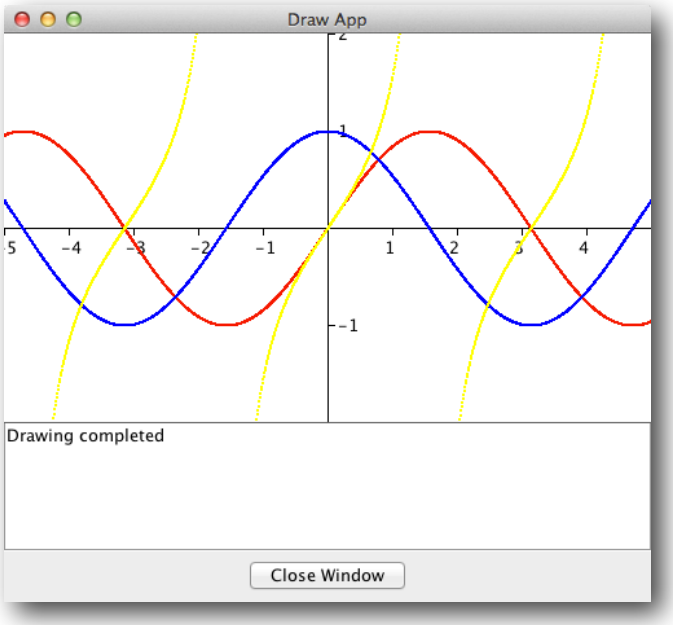
```
  {
    if (n != 0)
    {
      drawLine(xPosition,PIXELMIDY,xPosition,PIXELMIDY+5) ;
      sprintf(buffer,"%d", n);
      drawString(buffer,xPosition-6,PIXELMIDY+20) ;
    }
    n++;
  }
}

void labelYAxis(int limit)
{
  char buffer[10];
  int n = limit;
  int gap = PIXELHEIGHT / (limit * 2);
  int yPosition;

  for(yPosition = 0 ; yPosition < PIXELHEIGHT ; yPosition += gap)
  {
    if (n != 0)
    {
      drawLine(PIXELMIDX,yPosition,PIXELMIDX+5,yPosition) ;
      sprintf(buffer,"%d", n);
      drawString(buffer,PIXELMIDX+8,yPosition+5);
    }
    n--;
  }
}

void plotSin(int xLimit, int yLimit)
{
  double x,y;
  double xscale = PIXELMIDX / xLimit;
  double yscale = PIXELMIDY / yLimit;

  setColour(red);
  for(x = -xLimit ; x < xLimit ; x += 0.01)
  {
    y = sin(x);
    int pixelX = (int) (PIXELMIDX + (x * xscale));
    int pixelY = (int) (PIXELMIDY - (y * yscale));
    drawRect(pixelX, pixelY, 1, 1);
  }
}

void plotCos(int xLimit, int yLimit)
{
  double x,y;
  double xscale = PIXELMIDX / xLimit;
  double yscale = PIXELMIDY / yLimit;
```

```
  setColour(blue);
  for(x = -xLimit ; x < xLimit ; x += 0.01)
  {
    y = cos(x);
    int pixelX = (int) (PIXELMIDX + (x * xscale));
    int pixelY = (int) (PIXELMIDY - (y * yscale));
    drawRect(pixelX, pixelY, 1, 1);
  }
}

void plotTan(int xLimit, int yLimit)
{
  double x,y;
  double xscale = PIXELMIDX / xLimit;
  double yscale = PIXELMIDY / yLimit;

  setColour(yellow);
  for(x = -xLimit ; x < xLimit ; x += 0.01)
  {
    y = tan(x);
    int pixelX = (int) (PIXELMIDX + (x * xscale));
    int pixelY = (int) (PIXELMIDY - (y * yscale));
    drawRect(pixelX, pixelY, 1, 1);
  }
}

void drawGraph()
{
  int xLimit = 5;
  int yLimit = 2;
  drawAxes();
  labelXAxis(xLimit);
  labelYAxis(yLimit);
  plotSin(xLimit, yLimit);
  plotCos(xLimit, yLimit);
  plotTan(xLimit, yLimit);
}

int main(void)
{
  drawGraph();
  return 0;
}
```

This is the drawing displayed:

The drawing functions in the program have parameters used to specify the limit, or maximum value, on each axis. In the drawing above the x-axis maximum value is 5 and the y-axis maximum value is 2. The graph is drawn with (0,0) in the middle of the drawing area, so the x-axis actually shows the range -5 to 5 and the y-axis -2 to 2.

Since the functions have the ability to work with different limits it would nice to allow the limits to be specified on the command line when the program is run, letting the user select the version of the drawing they want. For example, the program could be run using this command:



./a.out 3 1 | java -jar drawapp.jar

where the 3 and 1 following a.out (the default executable program name) specify the x limit to be 3 and the y limit to be 1. The `main` function can be modified to read the values from the command line like this:

```
int main(int argc, char* argv[])
{
  int xLimit, yLimit;
  if (argc != 3)
  {
    fprintf(stderr,"Usage: %s xlimit ylimit\n", argv[0]);
    return 1;
  }
  sscanf(argv[1], "%d", &xLimit);
  sscanf(argv[2], "%d", &yLimit);
  drawGraph(xLimit, yLimit);
  return 0;
}
```

The signature of the `drawGraph` function has to be updated to have two `int` parameters, `xLimit` and `yLimit`, replacing the local variables of the same name.

Not done yet... Remember, duplication is bad! Look at the `plotSin`, `plotCos` and `plotTan` functions. Most of the code in each function is identical – duplication! This can be eliminated by creating a single function that can draw any of the curves. It will need two additional parameters, the colour to draw with and the function used to calculate the points on each curve.

How can a function be passed as a parameter to another function? By using a function pointer! As well as pointers to data in memory, C also allows pointers to functions. Dereferencing a function pointer allows the function to be called.

The main 'trick' to using function pointers is to understand how the type of a function can be declared. For this program, we need pointers to the `sin`, `cos` and `tan` functions, all of which have a function signature with this pattern: `double f(double)`. The type of a pointer to a function with this signature is `double(*f)(double)`. This declares a variable `f` that is a pointer to a function that takes a `double` parameter and returns a `double` result. The parentheses around the name `f` are necessary to specify that a pointer to a function is being declared and not type `double*`. The function pointed to by `f` can be called by dereferencing the pointer and supplying a parameter list: `(*f)(1.2)`.

The new function called `plotCurve`, replacing `plotSin`, `plotCos` and `plotTan`, is listed below, along with an updated `drawGraph` method showing how the function is called. Note that a function pointer can be passed to the `plotCurve` function simply by giving the function name. The C compiler will do the type checking needed to check that the function matches the type specified in the parameter list.

```
void plotCurve(int xLimit, int yLimit, colour curveColour, double(*f)(double))
{
  double x,y;
  double xscale = PIXELMIDX / xLimit;
  double yscale = PIXELMIDY / yLimit;

  setColour(curveColour);
  for(x = -xLimit ; x < xLimit ; x += 0.01)
  {
    y = (*f)(x);
    int pixelX = (int) (PIXELMIDX + (x * xscale));
    int pixelY = (int) (PIXELMIDY - (y * yscale));
    drawRect(pixelX, pixelY, 1, 1);
  }
}

void drawGraph(int xLimit, int yLimit)
{
  drawAxes();
  labelXAxis(xLimit);
  labelYAxis(yLimit);
  plotCurve(xLimit, yLimit, red, sin);
  plotCurve(xLimit, yLimit, blue, cos);
  plotCurve(xLimit, yLimit, yellow, tan);
}
```

On the subject of eliminating duplication, go back and look at the `labelXAxis` and `labelYAxis` functions. They look pretty similar, can they be replaced by a single function?

Hmmm, not so easy as although both functions have the same pattern of code there are a lot of differences in the details. The higher level abstraction of labelling an axis (for each label on the axis, draw a short line and the label value), is too dependent on those details to be cleanly separated out to provide common code. You could create a single function with a large number of parameters to specify each of the details but it is poor design to have functions with more than four parameters. Also you would lose readability.

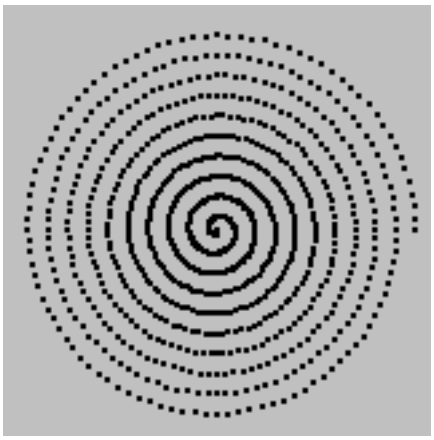
The bottom line is that everything is a compromise – this is what engineering is about, finding the best trade-off between factors such as cost, efficiency, clean design and time spent. So despite our desire to eliminate duplication we will compromise here and leave the functions unchanged.

Q2.13 Write a drawing program to plot a spiral using a series of points, like this:

Hint: If you can draw a circle, just keep increasing the radius.

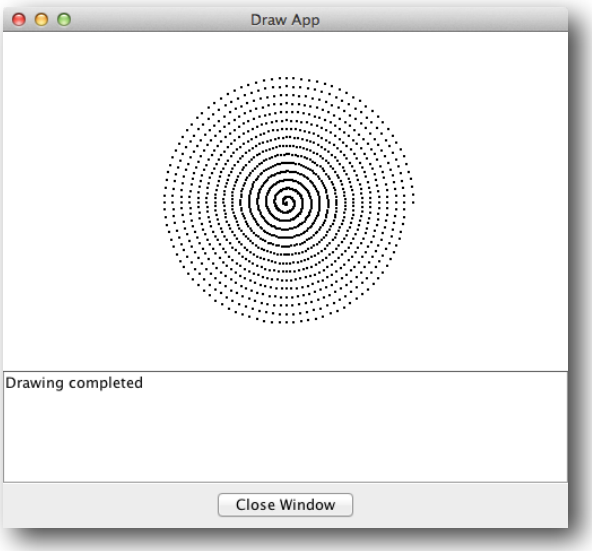
Example Answer:

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



```
#define PI 3.14159265

int main(void)
{
  double radius;
  double theta;
  double increment = 2.0 * PI/100.0;
  int step;

  for (step = 0 ; step < 1500 ; ++step)
  {
    theta += increment;
    radius = 75.0 * step / 1000.0;
    int x = 250 + (int)(radius * cos(theta));
    int y = 150 + (int)(radius * sin(theta));
    drawRect(x,y,1,1);
  }

  return 0;
}
```

The program above displays this:

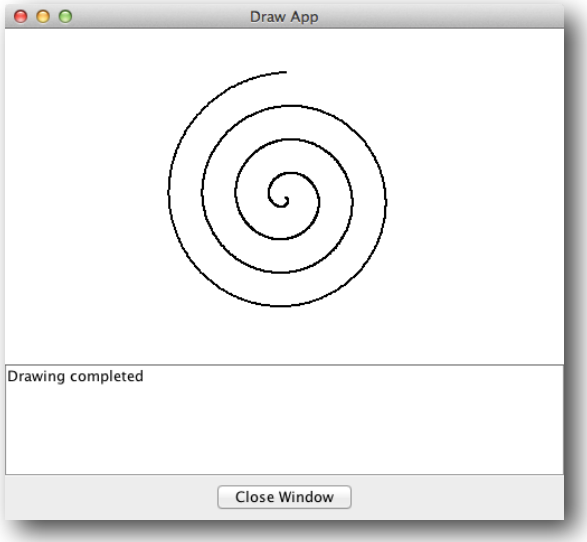


The program works by drawing a circle, by plotting points (1x1 rectangles) on the circumference, but increasing the radius by a small amount when displaying each point (attempt to draw a circle and miss).

Adjusting the increment can give the appearance of a smoother curve. For example, using:

```
  double increment = 0.5 * PI/100.0;
```

displays this:



Challenges

Q2.14 Write a *function* to display rectangles of any character like the following:

```
*****
*****    %%%%%%%%
*****    %%%%%%%%
***** or %%%%%%%%
```

The function parameters should give the number of rows and columns, and the character to use.

Use the function to display various rectangles of different sizes.

Example Answer:

The function needs to use two loops, one nested inside the other to count through the rows and columns to display each rectangle.

```
#include <stdio.h>

void rectangle(char, int, int);

void rectangle(char c, int width, int height)
{
  int row, column;
  for (row = 0 ; row < width ; row++)
  {
    for (column = 0 ; column < height ; column++)
    {
      printf("%c",c);
    }
    printf("\n");
  }
}
```

```
int main(void)
{
  rectangle('*', 5, 4);
  rectangle('%', 3, 9);
  return 0;
}
```

Q2.15 Write a drawing program that includes functions for drawing rectangles and triangles of any size and at any location. Draw a picture using the functions.

Example Answer:

This question is primarily about practice in writing functions. The example code below illustrates two basic functions to draw shapes. There are many other functions and variations that could be added. The key issue is to recognise the value of using functions as units of abstraction, so that once a family of shape drawing functions are defined, a drawing can be defined in terms of shapes, rather than the individual lines that shapes are made from.

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

void drawRectangle(int,int,int,int);
void drawEquilateralTriangle(int, int, int, int);
void drawPicture();

void drawRectangle(int x, int y, int width, int height)
{
  drawLine(x, y, x + width, y);
  drawLine(x + width, y, x + width, y + height);
  drawLine(x + width, y + height, x, y + height);
  drawLine(x, y + height, x, y);
}

void drawEquilateralTriangle(int x, int y, int width, int height)
{
  drawLine(x, y, x + width, y);
  drawLine(x, y, x + (width / 2), y - height);
  drawLine(x + (width / 2), y - height, x + width, y);
}

void drawPicture()
{
  drawRectangle(150, 120, 200, 170);
  drawRectangle(230, 230, 40, 60);
  drawRectangle(170, 230, 40, 30);
  drawRectangle(290, 230, 40, 30);
  drawRectangle(170, 160, 40 , 30);
  drawRectangle(290, 160, 40 , 30);
  drawEquilateralTriangle(150, 120, 200, 70);
}

int main(void)
{
```

```
  drawPicture();
  return 0;
}
```