LAB 7 - Functions

- Before we get started, let's review the arithmetic operators that you should know at this point.

+	add the left operand and the right operand, providing the sum.
-	subtract the left operand and the right operand, providing the difference.
*	multiply the left operand and the right operand, providing the product.
/	divide the left operand and the right operand, providing the quotient. When dividing one integer by another, the remainder is simply discarded.
%	divide the left operand by the right operand, providing the remainder. This operator may only be used with two integer operands.

- You've learned about functions in math classes. Do you remember what a function is in math? Recall that a mathematical function takes some number of inputs, and somehow maps those inputs to produce a single output. Consider the addition operator, `+`. It is effectively a function that requires two inputs -- the two numbers to be added -- and it produces a result which is the sum of those two inputs. For example, 2 + 3 produces a result of 5.

- In C, a function is defined by showing each of its input types (and the names of a *variables* to hold those inputs) and its output type. We could write the addition function in C, like this:

```
// result  function    1st input             2nd input
//  type    name    type     name          type        name
//   ↓       ↓       ↓        ↓              ↓            ↓

 int add(int leftInput, int rightInput)
 {
     return leftInput + rightInput;
 }
```

  Here, we are defining a function called `add` that takes as input an `int` that will be stored in a memory location called `leftInput`, and an *int* that will be stored in a memory location called `rightInput`. This function produces an `int` result.

- Open the **Chrome** browser, and sign in to https://learn.cs.uml.edu.

- Program *lab07a.c* is a complete program that defines and calls the `add` function. Open and carefully review the comments in the program (shown in light blue and green text), to try to understand what is happening.

- Let's walk through this program. Place a breakpoint at line 32, which will cause your program to stop just before calling the `add` function. Run the program. When your program stops at line 32, open the Memory View, and inspect the values of your variables. As you would expect, there is some garbage value in the variable 'sum', since no value has yet been assigned to that variable prior to line 32. The result returned by the `add` function, after that function executes, will be stored in 'sum'.

- Click on Step, and notice that you are entering the `add` function. Press Step until you reach line 19. When you look at the Memory View now, you will see a new "Activation Record" for the function `add`. An activation record is created each time a function is called. An activation record contains three components. **You should know the three components of an activation record:**

    - A **<u>copy</u> of the <u>values</u> of the arguments** which were passed to the function are *pushed* onto the runtime stack. To *push* onto the stack means to add the value at the next lower unused memory location on the runtime stack.
    - The **return address**, i.e., where to return to when the function completes, is pushed onto the runtime stack.
    - **Space** is **reserved** on the runtime stack **for** any ***automatic local variables***: those variables declared within the function (and not declared as `static`).

- Notice that the value of the variable 'resultSum' is not yet set. It contains some garbage value. Press Step once more, to execute line 19, and see that 'resultSum' now has the value 11, which is the sum of the two arguments.

- We are about to execute line 21, where we specify to `return` a value. That value will be available to the caller of the function. When you now press Step, you'll see that we have exited the `add` function and are back in the function `main`, at line 35. Notice, in the Memory View, that the variable 'sum', which was assigned the value returned by `add`, now also contains the value 11.

- Also notice, at this point, that the activation record for the call to `add` has been destroyed. The only remaining activation record is for `main`, which is still active -- we are still executing in `main`.

- When you press Continue, you'll see the `printf()` output, showing the result of the addition.

- Functions can be defined to return a value of a particular type. In the example we just examined, the return value of `add` was declared to be `int`. Sometimes, we want to declare a function that executes a number of statements, simply because those statements will be executed repeatedly, and it is easier to read code that calls a function to execute those statements repeatedly, than to read code that has the whole set of statements copied repeatedly each time those statements are to be executed. To declare a function that does not return a value, you may use the return type **`void`**.

- Open program *lab07b.c*. This program calls a function called `digitSum()` whose purpose is to accept an integer parameter, and calculate the sum of the digits of that number. For example, if the user enters the number **`245`**, the function should calculate that 2 + 4 + 5 is 11, so the main program should print that the sum of the digits of 245 is 11. This program does not work correctly, however. It displays incorrect values. Your task is: by setting breakpoints and examining the Memory View, (a) determine any line numbers at which problems in this program lie, and (b) fix the program so it works correctly. <u>Only a very small amount of change to the program is required, to fix it.</u> If you have identified what the problem is, but do not know how to fix it, discuss it with your TA.

- Let's revisit the work you completed in the prior lab. There you used the drawing functions to generate a number of shapes. First, review the drawing features that are available to you, included at the end of this lab.

- In that lab, you drew some arcs. You had likely added calls to the `draw_` functions directly to your `main` function. Here is an example, though, of drawing a special type of arc, a circle, by calling a single function within the program. The function takes three arguments, the x and y coordinates of the circle's center, and the radius of the circle. It returns nothing, as indicated by the return type, **void**. This function is called by main, and displays a filled circle. Review program *lab07c.c* now.

- Create a new function in your program called `arc` that will draw an arc of a specified color, and optionally fill it. Your function should take the following parameters:

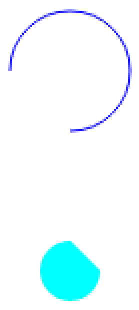
    - (int) The x coordinate of the center of the arc's circle
    - (int) The y coordinate of the center of the arc's circle
    - (int) The radius of the arc's circle
    - (double) the starting angle, in radians, measured from the circle's 3 o'clock position
    - (double) the ending angle, in radians, measured from the circle's 3 o'clock position
    - (int) A boolean, 'isAnticlockwise', that indicates whether the arc is drawn from the starting angle to the ending angle in a clockwise (if zero) or anticlockwise (if non-zero) direction
    - (int) A boolean, 'isFilled', which indicates whether the circle should be filled or unfilled
    - (int) A boolean, 'isDark', which indicates the color of the circle (true=blue, false=cyan)

- Be sure that you provide function documentation above the function, that describes the parameters, and good comments within the function, describing what your code is doing. Refer to the `circle` function for an example.

- Comment out the call to `circle` in your `main` function.

- Modify your main function so that it creates two arcs by calling your new `arc` function:
    - a blue, unfilled arc of radius 30, centered at (220, 50), which begins at the 9 o'clock position and draws in a clockwise direction to the 6 o'clock position
    - a cyan, filled arc of radius 15, centered at (220, 150), which begins at the 12 o'clock position and draws in an anticlockwise direction to the 3 o'clock position.

Your result should look like this:



- We would like for the user to be able to enter the start and end angles via the terminal. People, however, don't think easily in terms of radians, and entering a fractional value like Pi is not easy to do. We, people, much more typically think in terms of degrees around a circle. Since there are 360 degrees in a full circle, and 2Π radians in a complete circle, the formula to convert *d* degrees into radians is $radians = 2\Pi d \div 360$

- Write a function called `degreesToRadians` that accepts an `int` parameter, the number of degrees, and returns a `double` which is the equivalent angle in radians. For example, if you were to pass 0 to this function, it should return 0. If you were to pass 180, it should return the floating point value of Π, approximately 3.1415926. Your function should accept, convert, and return any whole number of degrees between 0 and 360, inclusive.

- The `degreesToRadians` function returns a value. Your documentation block, above the function, must describe the meaning of the return value, in addition to the purpose of the function and the meaning of the parameters as you saw with `circle`, and did yourself with `arc`. See the `add` function that we looked at earlier, once again, for an example of documenting the return value, in *lab70d.c*.

- Modify your `main` function so that it first prompts the user to enter the start angle, in degrees, and then waits for the user to enter the start angle. It then prompts the user to enter the end angle, in degrees, and waits for the user to enter the end angle. You program should then draw the open blue arc using the user-provide start and end angles. Don't forget that the `draw_arc` requires angles in radians, not degrees.

  You do not need to make any changes to the filled cyan arc.

- One of the shapes you drew in the prior lab was a square. As with the arcs, you likely had just written the statements that generated a square directly in your `main` function. Here is an example of drawing a square by calling a function. The function takes three arguments: the x and y coordinates of the top-left corner of the square, and the square's size (which is both its width and its height).

```
/**
 * Draw a square
 *
 * @param x
 *   The x coordinate of the top-left corner of the square
 *
 * @param y
 *   The y coordinate of the top-right corner of the square
 *
 * @param size
 *   The size (width and height) of the square, in pixels
 */
 void square(int x, int y, int size)
```

```
{
     // Begin drawing the path
     draw_begin();

     draw_setColor("red");

     // Move to the top-left corner, our starting position
     draw_moveTo(x, y);

     // Draw the top line
     draw_lineTo(x + size, y);

     // Draw the right side line
     draw_lineTo(x + size, y + size);

     // Draw the bottom line
     draw_lineTo(x, y + size);

     // Finally, close the path by drawing the left side line
     draw_lineTo(x, y);

     // Render this square, filled
     draw_finish(1);
}
```

- Notice how `draw_begin` and `draw_finish` act somewhat like parentheses, surrounding the "shape" being drawn. Be sure to always begin drawing a shape with a call to `draw_begin`, and finish drawing the shape with a call to `draw_finish`.

- Create your own `square` function that it takes all of the parameters in the `square` function above, plus one new parameter: a boolean (integer) called 'isDark'. When 'isDark' is true (any non-zero value), the function should display the square in color blue. When 'isDark' is false (zero), the square should display in color cyan. Don't forget to document the new parameter above the function.

- Modify your `main` function to call your `square` function. Be sure to test your function with various parameter combinations to ensure it can display both dark and non-dark squares of various sizes.

- Using the technique you used in the prior lab, for displaying a row of squares of alternating colors, now modify your `main` function to display a row of 10 squares of alternating blue and cyan, beginning at coordinates (10, 10), by calling your `square` function to display each of the squares.

- Using the technique you used in the prior lab, for displaying a row of squares of alternating colors, create a new function called `rowOfSquares` which takes these parameters:

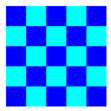
    - (int) The x coordinate of the top-left corner of the row of squares
    - (int) The y coordinate of the top-left corner of the row of squares

- (int) The size, in pixels, of each edge of a square
- (int) The length, i.e., the number of squares to display in a row
- (int) A boolean, ‘isDark’, which indicates the color of the first square in the row.

This function must display a row of squares of alternating blue/cyan, beginning with the specified color, using the given coordinates, square size, and length. Ea**ch square must be drawn by a call to your `square` function. You should not call any draw_ functions from within `rowOfSquares`.** Remember that a function must be declared within the program before any calls to that function. Be sure to comment your function, both its purpose and parameters (above the function), and the individual groups of statements within the function.

- Modify your `main` function so that it calls `rowOfSquares`. Your program should print a row of squares that has its top-left corner at (50, 50), each square of size 10 pixels, and 5 squares in the row.

- Now that you have a working `rowOfSquares` function, write a new function called `checkerboard` that uses `rowOfSquares` to generate a checkerboard pattern. **Each row of the checkerboard must be drawn by a call to your `rowOfSquares` function. You should not call your `square` function, nor any `draw_` functions from within `checkerboard`.** The pattern should look something like this:

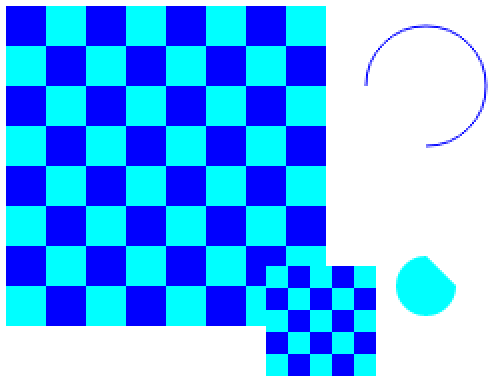


The function should accept the following parameters:
- (int) The x coordinate of the top-left corner of the row of squares
- (int) The y coordinate of the top-left corner of the row of squares
- (int) The size, in pixels, of each edge of a square
- (int) The length, i.e., the number of squares to display in each row and column
- (int) A boolean, ‘isDark’, which indicates the color of the first square in the first row.

- Modify your `main` function so that instead of calling `rowOfSquares`, it now has two calls to `checkerboard`. Your program should print the two checkerboards of squares as follows:

  - top-left corner at (10, 10), each square of size 20 pixels, 8 rows of 8 squares per row, with the top-left square the dark color.
  - top-left corner at 140, 140), each square of size 11, and 5 rows of 5 squares per row, with the top-left square the light color.

The result of running your program should now look like this:



Think about how much work it would have been to accomplish this, and how messy your code would be, if you didn't have the ability to create your own functions, and had to draw each of these individual squares using only the primitive `draw_` features.

- Checklist:
  A. digitSum
     1. Demonstrate how you debugged the digitSum program to discover its error(s), and what you did to fix it.

  B. drawing program
     1. Your program is properly indented, no ragged indents.
     2. Each function has a documentation block above it, describing the purpose of the function, the meaning of the parameters, and if applicable, the meaning of the return value.
     3. Each function implementation is well commented.
     4. The user is prompted to enter the start and end angles
     5. The blue arc is rendered using the user-provided angles, in the correct location.
     6. The cyan, filled arc, with top-right portion “cut off,” is rendered in the correct location.
     7. A checkerboard of 5 squares height and width, each 11-pixels, is rendered on top of a checkerboard of 8 squares height and width, each 20-pixels, with the proper overlap.
     8. The large checkerboard begins with a blue square in its top-right corner
     9. The small checkerboard begins with a cyan square in its top-right corner
     10. Describe (a) what an *activation record* is, and (b) its three parts

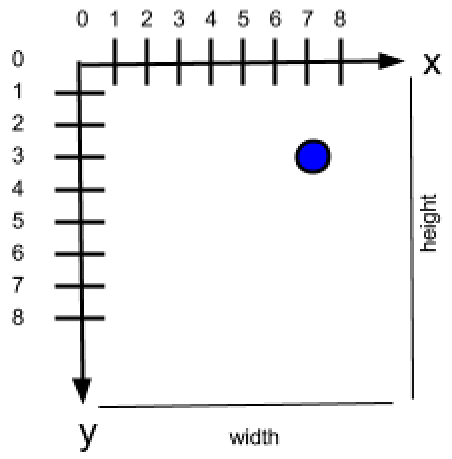
- Show the program to your TA for **sign-off** of checklist items**.** ⚑

Review of Drawing Features

- The *drawing canvas* is a grid of *pixels* (individual dots that can each display a color, or display nothing). The grid is numbered from left to right, beginning at the top-left corner, starting at 0. This is the *x axis*. Similarly, the grid is numbered from top to bottom, beginning at the top-left corner, starting at 0. This is the *y axis*. The *drawing canvas* also has a width and a height, which depends on the size of your LearnCS! window.

- A pixel is identified by its *x* location, and its *y* location.

- The following drawing shows a small portion of the top-left corner of the drawing canvas. You can see that it is numbered along the *x axis* beginning with zero, and along its *y axis* beginning with zero. There is a single pixel shown, at coordinates (7, 3), i.e., it is at position 7 on the *x axis*, and position 3 on the *y axis*.



- The available drawing functions are:
    - **draw_begin**
        - Begin defining a path. This function accepts no arguments. This path ends when **draw_finish** is called. *Each path is rendered in a single color.*

    - **draw_moveTo**
        - Move directly to the specified coordinates.
        - Arguments:
            1. (int) the *x* coordinate to move to
            2. (int) the *y* coordinate to move to

    - **draw_lineTo**
        - Draw a line from the current coordinates to the specified, new coordinates.
        - Arguments:
            1. (int) the new *x* coordinate
            2. (int) the new *y* coordinate

    - **draw_arc**

        - Draw an arc (some portion of a circle). The arc is drawn according to the argument values.
        - Arguments:
            1. (int) the *x* position of the center of the circle on which the arc is drawn
            2. (int) the *y* position of the center of the circle on which the arc is drawn
            3. (int) the radius of the circle on which the arc is drawn
            4. (double) the starting angle, measured from the circle's 3 o'clock position
            5. (double) the ending angle, measured from the circle's 3 o'clock position
            6. (int) a boolean that indicates "anticlockwise", i.e., whether the arc is drawn from the starting angle to the ending angle in a clockwise (if zero) or anticlockwise (if non-zero) direction
        - Angles are measured in radians. There are 2Π radians in a complete circle, just as there are 360 degrees in a complete circle.
        - You can obtain the value of Π by `#include <math.h>` and then referencing the constant `M_PI`.
        - To draw a complete circle centered at coordinates (100, 200), radius 50, you could issue this function call:

```
draw_arc(100,         // x coordinate of circle center
         200,         // y coordinate of circle center
         50,          // radius
         0,           // angle of start point of arc
         M_PI * 2,    // angle of end point of arc
         0);          // FALSE, meaning not anticlockwise
```

    - **draw_finish**
        - Finish defining a path, and draw it.
        - Arguments:
            1. (int) This argument specifies whether the path should be "filled" or not. If the argument is 0 (false), only the path itself -- an outline -- will be drawn. If the argument is any non-zero value (true), the path will be filled in. In either case, the path is drawn in the most recently-specified color, or black if no color has been specified.
        - Note: a straight line, drawn with **draw_line**, will not display if filled. When drawing a single straight line, the argument to **draw_finish** should be false, requesting no fill.

    - **draw_setColor**
        - Specify the color for the path currently being defined.
        - Arguments:
            1. (char *) A string representing the color. There are a small number of known textual color strings, such as `"red"` and `"black"` and `"cyan"`. Additionally, colors can be defined by their *mixture* of red, green, and blue, where each component is a value from 0 to 255. There are a number of available formats, for now, you can specify a string that looks like this:

```
"rgb(255, 165, 0)"
```

which means that the red component is 255 (as much red as possible), the green component is 165, and the blue component is 0 (no blue at all). This combination yields the color orange.

- **draw_clearRectangle**
    - This function should be called outside of defining a path, to clear a rectangular region of the drawing canvas.
    - Arguments:
        1. (int) The *x* coordinate of the top-left corner of the rectangle to be cleared
        2. (int) The *y* coordinate of the top-left corner of the rectangle to be cleared
        3. (int) The width, in pixels, of the rectangle to be cleared
        4. (int) The height, in pixels, of the rectangle to be cleared

End of lab 7