**Lecture 8 - Chapter 5: Functions – Mon Sept 18 or Tues Sept 19**

**Announcements**

Reading:

* Chapter 5

Assignments:

* Assignment #3 graded
* Due: Assignment #4 - due on **Sept 27** (MW class) or **Sept 28** (TR class) (no late assignments accepted)

Exam:

* Exam #1 on Sept 20th (MW - class) or Sept 21th (TR - class)

**Today’s Goals**

1. Assignment 3 Feedback
2. Modularizing Programs in C
3. Math Library Functions
4. Functions
5. Headers
6. Exam #1 Review (go over any questions from review sheet)

**Assignment #3 Feedback**

1. Pseudocode
   1. A few people didn’t hand in pseudocode
   2. Should be in initial comment! Not hand written!
   3. Should use same formatting – indentation – as we do with code.
   4. Some people made it too sentence like – writing paragraphs – see examples in lecture 5 notes
   5. Some people made it too code like – don’t just copy your code
   6. Add more details – some pseudocode lacked details
   7. Question to ask – **if I handed this to someone could they write code that solves the problem**?
2. Commenting
   1. Yes, you must comment your code.
   2. Place those comments above sections of code.
   3. Comments should be indented at same level as code.
   4. Comments to the right are hard to read unless on variable declarations.
   5. Comment sections of code, not individual lines, unless they are super complicated.
   6. If you are going to write functions – they need comment above explain what the functions does
3. Indentation and spacing
   1. Indentation is getting a lot better
   2. Indent code and comments at same level
   3. Add white space to code to help with readability
   4. Some code could use better use of space - white space helps readability
4. Variables
   1. Saw several assignments that used one or two-letter identifiers.
   2. Don’t get lazy, use meaningful names
   3. Initialize before use! Very dangerous not to initialize variables before you use them.
5. If statements
   1. Watch where you are writing

If (menu == 1) {

}

If (menu == 2) {

}

If (menu == 3) {

}

**When you should be writing!**

If (menu == 1) {

}

else if (menu == 2) {

}

else if (menu == 3) {

}

1. Specification
   1. Read a 4-digit number , not 4 one-digit numbers
   2. The key was to use division and remainder operators to get the individual bits out of 4-digit number
   3. Not handling invalid menu option and looping until menu option is valid
   4. Not allowing the user to run code an unspecified number of times – while loop

**Today’s Terminology**

**Terminology**

* Modular Programming
  + Breaking down of a problem into smaller, simpler and more manageable parts and then coding those parts into independent units.
  + Makes code easier to maintain and debug.
* Functions
  + Group of statements that perform a specific task
  + Procedure and methods are terms people use as well
* Function Prototype
  + Combination of the function’s return type, name and parameter list
  + Not part of pre-standard C
* Function Definition
  + The return type, function name, parameters and body
* Return Value Type
  + The data type of the value the function returns
* Function Name
  + A user defined name that describes the function
  + In this class, these need to be meaningful names
* Parameters
  + The values that are specified in the function definition
  + How information is passed to a function
* Arguments
  + The values that are specified in the function invocation (call)
  + The information we want to send to a function
* Parameter list
  + Comma separated list of variables passed to a function (definition)
* Invoking a Function
  + Calling a function
* Local Variable
  + Variable defined within a function
  + Can only be accessed within the function
* Value Returning Function
  + A function that returns a value
* Void Function
  + A function that performs some task without returning a value

**Modularizing Programs in C**

**Purpose**

* Provides a way to organize and simplify code
* Place code that is used over and over into a function

**Why Use Functions**

* Once code is in a function it can be called many times
* Enables code reuse!
  + Reduces redundant code
  + Solves the don’t repeat yourself (DRY) issue
* Improves code quality!
* Breaking code into units makes it easier to plan, code, test and modify

**C library – Pre-define Functions**

* Have already used C Library functions – printf, scanf, getchar
* Use the functions in the C standard library to help make programs more portable
* Provides functions for performing
  + Mathematical calculations
  + String manipulations
  + Character manipulations
  + Input/output
  + And more…

**Programming-defined Functions**

* Now it's time to create our own programmer-defined functions

**Math Library Functions**

Using functions that have been pre-defined in **C** to perform mathematical functions

**Math Functions**

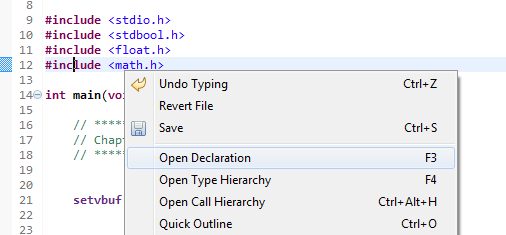
* Functions that perform mathematical calculations beyond the basic operators +, -, \*, /, %
* To use the Math functions - **#include <math.h>**
* All functions in the math library that return decimal values return data type **double**

**Constants**

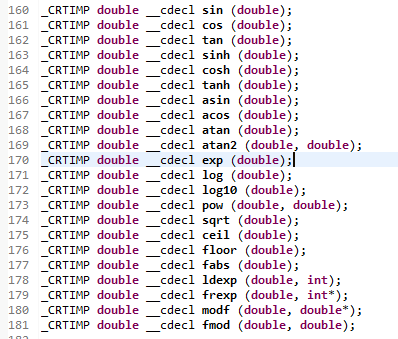
* There are constants in the Math library
* Two common ones:
  + M\_PI => 3.14159...
  + M\_E => 2.71828... (base of natural log)

**Note**

* Use Eclipse to see what is in math.h
* Select **#include <math.h>**
* Right click on selection
* Select **Open Declaration** in the dropdown



* Scrolling down you will see the list of math functions available



**Trigonometric Functions**

* sin(radians), cos(radians), tan(radians)
  + Give these functions an angle in radians
  + The returned value is in radians

**printf** ("sin(0.0) = %f\n", **sin**(0.0)); // Displays 0.0

**printf** ("sin(0) = %f\n", **sin**(0)); // Displays 0.0

**Exponent Functions**

* exp(x)
  + Exponential function **ex**
* log(x)
  + Natural log of x (base e)
* log10(x)
  + Log of x (base 10)

**Rounding Functions**

* ceil(x)
  + Rounds x up to nearest integer
* floor(x)
  + Rounds x down to nearest integer

**Other Functions**

* fabs(x)
  + Returns a double that is the absolute value of the floating-point argument

**printf** ("Absolute value of -81 = %.2f\n", **fabs**(-81)); // Returns 81.00

* pow(a, b)
  + Returns the value of **a** raised to the power of **b** – that is - **ab**

**printf** ("10 raised to the power of 3 = %.2f\n", **pow**(10,3));

**printf** ("10 raised to the power of 3 = %d\n", **pow**(10,3));// Warning in eclipse

**printf** ("10 raised to the power of 3 = %d\n", (**int**)**pow**(10,3));

**int** x = **pow**(10,3); // Double returned converted through assignment

**double** y = **pow**(10,3);

**printf** ("x = pow(10.3) where x is declared an int = %d\n", x);

**printf** ("y = pow(10.3) where y is declared a double = %.2f\n", y);

**Displays**

10 raised to the power of 3 = 1000.00

10 raised to the power of 3 = 0

10 raised to the power of 3 = 1000

x = pow(10.3) where x is declared an int = 1000

y = pow(10.3) where y is declared a double = 1000.00

* sqrt(x)
  + Returns the square root of x (x >= 0)

**printf** ("Square root of 45 = %.2f\n", **sqrt**(45));

**Programmer Defined Functions**

**Function Definition**

* When you define a function, you are defining
  + what value it returns (if any)
  + what values it takes (if any)
  + what the function does
* A function has the following general form:

returnType **FunctionName** (parameter list) {

statement; // function body

....

statement;

}

* **returnType** - data type for value that is returned (**int, double, bool**, etc.) or **void** (nothing to return)
* **functionName** - a user defined name
* **parameter lis**t - values that are being passed to the function (must *explicitly* specify type for each value)

**Function Invocation**

* General form of "invoking" or "calling" a function

**functionName**(argument list);

**Function Prototype**

* General form of prototype (also called declaration)

returnType functionName(parameter list); Looks like definition but no function body - { }

**Example**

* Function that sums all the values from 1 to N

// Sums all values from 1 to N

**int** **sum** (**int** n) {

**int** sum = 0;

// Compute the sum of all numbers from 1 to N

**for** (**unsigned** **int** i = 1; i <= n; i++) {

sum = sum + i;

}

**return** sum;

} // sum

NOTE: the function name is ***sum*** and we also declared a variable ***sum*** inside the function

This is legal in C but could be confusing so writing it as follows is recommended:

// Sums all values from 1 to N

**int** **sum** (**int** n) {

**int** result = 0;

// Compute the sum of all numbers from 1 to N

**for** (**unsigned** **int** i = 1; i <= n; i++) {

result = result + i;

}

**return** result;

} // sum

**Calling a Function**

Calling a function = invoking a function

Two ways to call a function

* Treating the function call as a value
* Treating the function call as a statement

**Value-Returning Functions**

**Value-Returning Functions**

* Return a value to the caller
* Call can be treated as a value or a statement (it’s unusual to treat as statement)

**Trace of Function Invocation**

* Example of treating the function call as a value

**int** **sum** (**int** n); // Function prototype

**int** **main**(**void**) {

// Prompt user for a positive integer value and store in the integer variable n

**puts** ("Enter a positive number:");

**scanf** ("%d", &n);

**int** totalSum = sum(n); // Value returned by sum is assigned to totalSum

**printf** ("Sum of numbers from 1 to %d = %d \n", n, totalSum);

} // main

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Sums all values from 1 to N

// Input: Integer representing N

// Return: Sum of numbers from 1 to N

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**int** **sum** (**int** n) {

**int** result = 0;

// Compute the sum of all numbers from 1 to N

**for** (**unsigned** **int** i = 1; i <= n; i++) {

result = result + i;

}

**return** result;

} // sum

**Displays**

Enter a positive number

3

The sum of the numbers from 1 to 3 is 6

**Trace of Function**

* The main function is where the code starts
* The user is prompted for a positive number
  + Assume the user enters 3
* The number is read form the console and stored in a variable
  + 3 is then stored in variable called **n**
* The function **sum** is called
  + The value entered by the user is the argument
  + This means, the value stored in **n** (5) is sent to the function
  + Control now enters the **sum** function
* Substitute argument (i.e. **number**) into the parameter (i.e. **n**)
  + n = 3
  + The function now has a value for "n" that can be used within the function
* Declare and initialize variables
  + result = 0
* Process for loop
* The **return** statement forces control to return to the caller
  + In this case the main function
* The **return** statement sends to caller an integer value
  + In this case, value in variable result (i.e. 6)

**void Function**

**Void Function**

* Does not return a value to the caller
* Call must be made as a statement

**Trace of Function Invocation**

* Example of treating the function call as a statement

**void** **printSum** (**int** n); // Function prototype

**int** **main**(**void**) {

// Prompt user for a positive integer value and store in the integer variable n

**puts** ("Enter a positive number:");

**scanf** ("%d", &n);

printSum(n);

} // main

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Prints the sum of the numbers from 1 to N

// Input: Integer representing N

// Return: void

// \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**void** **printSum** (**int** n) {

**int** sum = 0;

// Compute the sum of all numbers from 1 to N

**for** (**unsigned** **int** i = 1; i <= n; i++) {

sum = sum + i;

}

**printf** ("Sum of the numbers from 1 to %d = %d\n", n, sum);

} // printSum

**Displays**

Enter a positive number

3

The sum of the numbers from 1 to 3 is 6

**Notes**

* Programs should be written as a collection of small functions.
* A function should perform **one** task.
* Arguments and parameters in the function prototype, function definition and function call must agree in
  + Number
  + Order
  + Type
* Function can return a value or nothing.
  + When a value is returned:
    - Function must specify the type of value it returns
      * If not specified defaults to **int**
    - A return statement is required
      * If left off it generates a warning and possibly incorrect result
    - The return statement is the point at which control returns to the caller
  + When no value is returned:
    - The return type must be set to void
    - No return statement is used
  + Control returns to caller when the function’s closing curly brace is reached
* **Note:** main returns an int but if omitted it *implicitly* returns a 0 (C-standard)