**Lecture 10 - Chapter 5: Functions – Mon Sept 25 or Tues Sept 26**

**Announcements**

Reading:

* Chapter 5

Assignments:

* Exam #1 Graded
* Due: Assignment #4 - due on **Sept 27** (MW class) or **Sept 28** (TR class) (no late assignments accepted)

**Today’s Goals**

1. Exam #1
2. Function Prototypes
3. Headers
4. Passing Arguments by Value
5. Data Types and Casting

**Exam #1**

Exam #1:

* Exams: 67
* Average: 80.4
* Highest Score: 97
* %A – 28%
* %B – 25%
* %C – 30%
* %D – 12%
* %F – 4%

Changes I made:

* Question 1.c. – threw out - the wording made many people think something other than I intended.

What’s left in semester:

* Assignments – 45%
* Exam #1 – 15%
* Exam #2 – 15% (Around Oct 18th/19th - Chapters 5, 6, 7)
* Final Exam – 25%

Key Points:

* Understanding the difference between the logical operators **AND (&&)** and **OR (||)**
  + If you are struggling with this go back to section 4.10
* Understanding **while loops** and **for loops** and when each is appropriate to use
  + We want to learn idiomatic code – code that does a common task in a common way
  + This is like design patterns
  + Programming languages are built on core structures
    - Selection statements – if, switch
    - Iterations statements – while, for
  + If you learn these 4 core structures, **WHEN** to use them, and **HOW** to use them properly then you will have the tools you need!
    - Identify the problem type
    - Determine what pattern to use
  + Don’t get caught up in the – “C lets me do it”
    - In English, can combine words in fashion that is syntactically correct but don’t make sense
      * “Colorless green ideas sleep furiously” – Noam Chomsky composed this to show how you can create grammatically correct sentence but semantically nonsensical
    - Just because C allows something syntactically, doesn’t mean it is the correct thing to do
      * for (int k = 0; k < 10 || number != 0; k++)
  + Problem 10 should have been written with a while loop
    - People tried to use a **for-loop** for a problem that is not a pattern for a for loop
      * most people were unsuccessful
      * those that got it to work did so using bad practices
    - For loops should iterate through all iterations!
      * You should not “force” a for loop to stop early
      * You should not exit a **for loop** with a return or break statement
    - Do-while loops were also not the correct pattern
      * The body always executes once with a do-while loop
      * For this problem, the user could enter a zero and there would be no list
      * Using a do-while now causes you to add extra code to handle that situation
    - Some people combined while loops and for loops
      * I think you were on the correct path in your thinking but you should have asked yourself what you were accomplishing with nesting those two loop structures.
    - People also tried to solve with arrays and thought you needed arrays to solve the problem
      * We haven’t even discussed arrays and I would never put a problem on an exam about something we have not discussed
      * If you thought you needed an array, it indicates a lack of understanding of how to use variables properly.

Exam #1 Question 10:

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

// Question #10

// Write a loop that asks the user for a series of positive integers. The integers must be

// read one at a time. When the user enters the integer 0 or after 10 values have been read,

// the code will display the following information. (25 pts)

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

**puts** ("");

**puts** ("Question #10 - while loop");

**puts** ("Ask user for a list of integers that is terminated by a 0 or when 10 values are read");

// Setup +3pts

**int** intValue;

**puts** ("Enter an integer value: ");

**scanf** ("%u", &intValue);

// Correct initialization of variables +5pts

**int** counter = 0;

**int** sum = 0;

**float** average = 0;

**int** largest = intValue;

**int** smallest = intValue;

// Using a while loop and setting up boolean expression correctly +5pts

**while** (intValue != 0 && counter < 10) {

// Correct computations +2pts

counter++;

sum = sum + intValue;

// Correct if statements +4pts

**if** (intValue > largest) {

largest = intValue;

}

**if** (intValue < smallest) {

smallest = intValue;

}

// Correctly getting next value +1pts

**puts** ("Enter an integer value");

**scanf** ("%u", &intValue);

}

// Correctly dealing with empty list (counter = 0)

// Computation of average outside of loop

// Correct average computation +2pt

// Correct output +4pts

**if** (counter != 0) {

average = (**float**)sum/counter;

**printf** ("The number of integers entered - %d\n", counter);

**printf** ("The average of the integer values - %.2f\n", average);

**printf** ("The largest integer entered - %d\n", largest);

**printf** ("The smallest integer entered - %d\n", smallest);

}

**else** {

**puts** ("No values entered");

}

**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 values sending to a function
* 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
* Storage duration
  + Period during which the identifiers exists in memory
* Scope
  + Section of a program that a identifier can be referred to (i.e., is visible in)
* Linkage
  + In multiple-source-file programs, where identifier is known
  + Only in current source file or in any source file

**Function Prototypes**

**Function Prototype (Declaration)**

* Feature that was not in pre-standard C
* Added to take advantage of type-checking capabilities
  + Complier uses to validate functions calls
* Placed at the top of your file **before** main
* Just as you have declarations for variables before they are used you have a declaration for functions
* Compiler compares **function prototype** to **function call** and ensures
  + number of arguments is same
  + data type of arguments is same
  + argument types are in correct order
  + return type is consistent with the context in which function is called
* A function call that does not match function prototype will generate a compilation error

**Functions Before or After Main – why the fuss?**

* Use the approach in the book
  + Function prototypes before main
  + Functions after main!
* Why use function prototypes to declare function
  + With prototype
    - When arguments don’t match – conversion occurs for you
  + Without prototype
    - Compiler gives **warning message** when function returns **int** because compiler assumes return type is int
    - Compiler give **error message** when functions returns **non-int** because compiler assumes return type is int! So that is good!
    - When **arguments** that don’t match **parameter types** – unexpected values are placed in parameters
    - If ignore warning - can lead to unexpected behavior!
* As a developer
  + Look at main first
  + See what functions are invoked
  + See prototypes without searching code for functions
* Later when you create your own header files, you will be moving functions to those files.

**Automatic Conversations**

* In general, argument values in a function call that **DO NOT** correspond precisely to parameter types in the function prototype are converted to the proper type before the function is called.
* **Example** 
  + Assume there is a function prototype

**int** **autoConvertToInt** (**int** first, **int** second);

* + This works even though the function is expecting integer arguments and we are passing floats.

**int** **autoConvertToInt** (**int** first, **int** second) {

**return** first + second;

}

* Assume you have this code in main

// Works, no complier error

**float** first = 10.5;

**float** second = 5.5;

**float** third = autoConvertToInt (first, second);

**printf**("What did the auto convert do? third = %.2f\n", third);

* The output will show that the floats were converted to **int** and we **lost the .5 off each value**

What did the auto convert do? third = 15.00

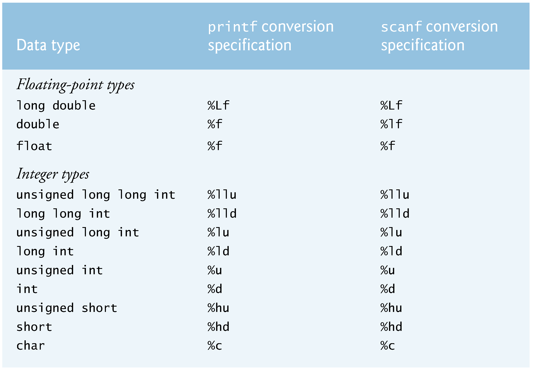
* **Note: automatic conversion can lead to unexpected results (especially logic errors)**
* Problems occur when a type with a **bigger range** is converted to a type with a **smaller range**

**Automatic Mixed Expressions Conversions**

* Mixed expressions contain values of two or more different data types
* Compiler converts values to the “highest” type
  + If one value is long double -> other converted to long double
  + If one value is double -> other converted to double
  + If one value is float -> other converted to float
  + Same with integers
* The standard discussed this situation in section 6.3 if you are interested
  + <http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1570.pdf>

**Overview**

* Arguments in **function ca**ll are automatically converted to **type of matching parameter**
  + The compiler makes a **copy** of the value/variable and converts the **copy**!
* Values in **expressions** are automatically converted to “higher” type if mixed types exist
* Figure 5.5 – p.167 - shows the conversion specifications for printf and scanf
  + In ***most cases***, the integer types lower in this figure are converted to types higher in the figure
    - In most cases!!!
    - As the story goes, yes in most cases, until not!



* **Be careful with format specifiers!!!**
  + In last example if I write this:

**float** first = 10.5;

**float** second = 5.5;

**float** third = autoConvertToInt (first, second);

**printf**("What did the auto convert do? third = %d\n", third);

Eclipse gives me a warning but if I ignore it:

* The output is: What did the auto convert do? third = 0

Because we accidently used **%d** instead of **%f** we got 0 which is an unexpected result!

* Clean up the warnings in your code!

**Pragmatic Programming Rule - Don’t live with broken windows!**

* A broken window in a home, left unrepaired, creates a sense of abandonment!
* This can lead to an increase in crime.
* The “Broken Window Theory” is used by police in many cities
  + Cracking down on small stuff prevents big stuff!
* The “Broken Window Theory” applies to code!
  + **Don’t leave bad designs, wrong decisions, or poor code left unrepaired**!

**Headers**

**Header**

* Standard libraries have a header that contains the **functions prototypes** for all the functions in that library
* Constants are also specified in the header

**Pre-defined Headers**

* <math.h>
* <stdio.h>

**Custom-defined Headers**

* You can create customer headers which you can then include in your programs

**Passing Arguments by Value**

We have looked at

* *value-returning* functions and
* *void* functions

Now let's discuss the process of **passing information** to functions

**Sending Information to a Function**

* Caller ***passes*** arguments
  + *swap*(**number1**, **number2**);
* Function ***uses*** parameters
  + **void** swap (**int** **num1**, **int** **num2**)
* Main rule!

Order and number of **arguments** much match

order and number of **parameters** declared in the function

**Example**

* Two ***arguments***
  + number1
  + number2
* Two **parameters**
  + num1
  + num2
* Arguments and parameters match in **order**, **number** and **type**

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

**puts** ("Pass by Value");

**int** number1 = 7;

**int** number2 = 82;

**printf** ("Before swap number1 = %d\n", number1);

**printf** ("Before swap number2 = %d\n", number2);

swap (number1, number2);

**printf** ("After swap number1 = %d\n", number1);

**printf** ("After swap number2 = %d\n", number2);

} // main

// Function to swap two integers values

**void** **swap** (**int** a, **int** b) {

**puts** ("\*\*\*Inside swap function\*\*\*\n");

**printf** ("Before swap a = %d\n", a);

**printf** ("Before swap b = %d\n", b);

// Swap the values

**int** tmp = a;

a = b;

b = tmp;

**printf** ("After swap a = %d\n", a);

**printf** ("After swap b = %d\n", b);

**puts** ("\*\*\*End of swap function\*\*\*\n");

} // swap

* To call the function we must "pass" two integer arguments
* When we pass values in C we ***pass-by-value***

**int** number1 = 7;

**int** number2 = 82;

swap (number1, number2); <- Passing values in number1 and number2 to function **swap**

* Displays

Before swap number1 = 7

Before swap number2 = 82

\*\*\*Inside swap function\*\*\*

Before swap a = 7

Before swap b = 82

After swap a = 82

After swap b = 7

\*\*\*End of swap function\*\*\*

After swap number1 = 7

After swap number2 = 82

**Pass-By-Value**

* **Pass by value** which means
  + A **copy of the argument** is sent to the function using this process:
    - The argument is fully evaluated
    - A copy of that value is placed into the parameter variable
  + Because only a copy is sent, the ***actual value*** of the argument is **NOT** changed by the function!
* In above example, the values of **number1** and **number2** are NOT changed. Why?
  + Only a copy of the variables sent to the function!
  + Whatever happens to variables inside the function does not affect variables outside the function
* But, what happens if we make the **argument’s** name match **parameter’s** name?
  + Will the swap work now?
  + That is, will the values of num1 and num2 be changed outside of function?
  + Why or why not?

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

**puts** ("Pass by Value");

**int** number1 = 7;

**int** number2 = 82;

**printf** ("Before swap number1 = %d\n", number1);

**printf** ("Before swap number2 = %d\n", number2);

swap (number1, number2);

**printf** ("After swap number1 = %d\n", number1);

**printf** ("After swap number2 = %d\n", number2);

} // main

// Function to swap two integers values

**void** **swap** (**int** number1, **int** number2) {

**puts** ("\*\*\*Inside swap function\*\*\*\n");

**printf** ("Before swap number1 = %d\n", number1);

**printf** ("Before swap number2 = %d\n", number2);

// Swap the values

**int** tmp = number1;

number1 = number2;

number2 = tmp;

**printf** ("After swap number1 = %d\n", number1);

**printf** ("After swap number2 = %d\n", number2);

**puts** ("\*\*\*End of swap function\*\*\*\n");

} // swap

**Displays**

Before swap number1 = 7

Before swap number2 = 82

\*\*\*Inside swap function\*\*\*

Before swap number1 = 7

Before swap number2 = 82

After swap number1 = 82

After swap number2 = 7

\*\*\*End of swap function\*\*\*

After swap number1 = 7

After swap number2 = 82

**Rules**

* If a function declares parameters, you must pass something and it better match in type!
* The name of the **argument** and **parameter** may be the ***same or different*** - it does not matter
  + Parameters are variables that live (have scope) in the function
  + When function is invoked - memory for parameter is created push on call stack
  + When function completes - memory released and parameter no longer exists (pop from call stack)
* Arguments must match the parameters in
  + order
  + number
  + type
* Automatic Conversations
  + In general, argument values in a function call that **DO NOT** correspond precisely to parameter types in the function prototype are converted to the proper type before the function is called.

* Does C only **pass *by value*** or does C allow ***pass by reference***?
  + By default, C is pass by value.
  + But C does have pass by reference
    - In chapter 7 (Pointers) we will be learning about pass-by-reference.
  + With pass by reference the parameter is acting as an **alias**
    - Whatever the function does to the parameter it is doing it to the argument
    - If the value is changed by the function, this change will be see outside the function

**Data Types**

**Data Type**

* Tells compiler **what type** of data that is stored in a variable
* You must **declare** the type of each variable
  + This means assign a specific type to the variable – it’s an integer
* Looked at these data types:
  + 4 integer types –**short, int, long**
  + 2 floating types – **float and double**
* Strongly typed language!
  + This means that once you declare a variable to be a certain it will behave as that type
  + Type safety- you can’t put a floating-point value into an integer unless you explicitly tell it

**Cup Analogy**

* Each data type has a range of values that are valid for that type
* Declaring a variable a certain type sets aside (allocates) a certain amount of memory for that variable
* Think of a Variable as a Cup

Cup is a container -> Variable is a container

Cup holds something -> Variable holds a value

Cup has a size -> Variable has a size

Variable has a name

* Primitive Types (not looking at unsigned versions – values can vary based on platform implementation)

**short** **int** **long**

Range -32,768 to 32,767 See note below -2,147,483,648 to 2,147,483,647

Storage 16 bits 16 or 32 bits 32 bits

**float** **double long double**

Range ~ 1.2E-038 to 3.4E+038 ~ 2.2E-308 to 1.8E+308 ~3.4E-4932 to 1.1E+4932

Precision 6 decimal places 15 decimal places 19 decimal places Size

Storage 4 bytes (32 bits) 8 bytes (64 bits) 12 bytes (96 bits)

* Note: range of values for **int** is
  + Greater than or equal to that of a short and
  + Less than or equal to that of a long

**Rules**

* When declaring variables - be sure it’s potential values can fit into the type you assign (i.e. cup)
* Pick the best type for your variable, although, if there are no concerns about memory
  + int – generally used as default type for integral values
  + double – generally used as default type for decimal values
* You can put the contents of a smaller variable (cup) into a larger variable (cup)

short int

* Putting the contents from a larger variable (cup) into a smaller variable (cup) **CAUSES ISSUES**
  + You will also lose information when you do this
  + To fix this issue you must do something special – casting

**NOT ENOUGH ROOM** to put contents

From big cup (int) into little cup (short) Overflow occurs!

Int short

**Overflow**

* Example
  + short smallCup ; Note this value is max value that’s fits in a short!
  + int biggerCup = 32767;
  + smallCup = biggerCup; This works just fine!

**Value in smallCup is now 32767!**

* + short smallCup ; Note this value is one more than fits in a short!
  + int biggerCup = 32768;
  + smallCup = biggerCup; This causes an issue!!! It causes overflow!

Java will catch this but C allows it!

**Value in smallCup is now -32768!**

* + biggerCup = smallCup; This can be done

The bigger cup can take on what’s in smaller cup

* General rules for variables:
  + When declaring variables be sure the potential values can fit into the type you assign (i.e. cup)
  + You can put the contents of a smaller variable (cup) into a larger variable (cup)
  + You can put the contents from a larger variable (cup) into a smaller variable (cup) but might get overflow
    - In Java, you had to explicitly cast when place values from bigger cup into smaller cup
    - In C, it does not care - you do not have to cast! C let’s you get into trouble if not careful!

**Casting**

* When you *explicitly* tell the complier to convert a variable from one data type to another data type
* We did this in lecture 5 when we were calculating an average using two integers
* Type cast (or casting)
  + Explicitly convert a variable of one type into another type
    - **(type)value**
  + Casting will **widen** or **narrow** the type
    - **Widening**
      * Casting a value with a smaller range to one with a bigger range
      * C does this automatically
      * Example

double d = 3; // implicit casting

printf (“Value in d = **%f**\n”, d); // Display result

* + - * Display

Value in d = 3.000000

* + - **Narrowing**
      * Casting a value with a bigger range to one with a smaller range
      * Example

int number = (int)3.0; // explicit casting

printf (“Value in number = **%d**\n, number); // Display result

* + - * Displays

Value in number = 3

* + printf (“**%d**\n”, 5.6);
    - Displays 1717986918
    - Gives warning
    - Garbage value – have a %d with a float – unexpected results
  + printf (“**%d**\n”, **(int)**5.6);
    - Displays 5
    - This casted a **float** to an **int** – the fractional part was truncated
  + printf (“**%d**\n”, 10/4);
    - Displays 2
    - Integer division – fractional part was truncated
  + printf (“**%f**”\n, **(float**)10/4);
    - Displays 2.500000
    - Casted **int** 10 into a **float,** no longer integer division
  + printf (“**%f**\n”, (float)(10/4));
    - Displays 2.000000
    - Performed integer division then casted the result **int** to a **float**
* Example with cups
  + short smallCup;

int bigCup = 10; 10 will fit into the smallCup

smallCup = bigCup; Notice the format specifier for short (p.167)

printf (“%hd\n”, smallCup); **Displays 10**

* Example with issue
  + bigCup = 32768; 32768 fits just fine into a int!

smallCup = (short)bigCup; Even cast it here!

printf (“%hd\n”, smallCup); **Displays -32768**

.

* Notes
  + Need to be careful with casting!
  + Can result in loss of information --- can cause inaccurate results!
  + Casting DOES NOT change the variable being cast.

float x = 5.6;

printf (“%d\n”, (int)x); **x is still a float**

* + Be care with format specifiers as well!

float x = 5.6;

printf (“%f\n”, x); Displays 5.600000 as expected

printf (“%d\n”, x); Eclipse gives warning – if I ignore

Displays 1610612736 on my system

%d is problem – gibberish because no automatic cast

Internal representation of floats and int differ!

printf (“%d\n”, (int)x); Displays 5 – as expected result with casting