**Lecture 5 - Chapter 3: Structured Program Development in C – Wed Sept 6 or Thurs Sept 7**

**Announcements**

Reading:

* Chapter 3

Assignments:

* Graded: Assignment #1
* Due: Assignment #2
* Assign: Assignment #3 (while Loops) –due on **Sept 13** (MW class) or **Sept 14** (T/R class)
* **No late assignments accepted**

**Today’s Goals**

1. Assignment #1 feedback
2. While Loops
3. Assignment Operators
4. Increment and Decrement Operators
5. Case Studies
   1. Top-Down -Stepwise Refinement
   2. Pseudocode Examples
6. Secure C Programming

**Assignment #1 Feedback**

* Please turn in .c files – NO zip files please – it is extra work to unzip everyone’s file
* Make sure your file is completely submitted. Several people’s file was “in progress” so we could not get to the .c file. I was told this occurs if you are submitting over a wireless connection and you lose the connection or it times out.
* Turn in a **hardcopy version** of your code.
* Review the Programming Assignments Policy about what is expected on your assignments for the 4 areas we are grading: documents, quality, specifications, and correctness. There were some issues with indentations and white space so if you received a comment about those go back over the policy.
* Pseudocode is now required on all assignments unless otherwise stated
* Check your work! May people did not check if their formulas were correct. Being diligent is part of the engineering mindset that is important to develop.
* If you want to do more:
  + I’m fine with students wanting to do something beyond the assignment.  I believe that if your understanding is beyond what we are currently covering in class then doing something more only helps develop stronger skills.
  + When student decide to do this, it is still a requirement that the implementation correctly address the structures/concepts the assignment requires.  For example, in assignment #2, that would be the proper use of if-statements.
  + It is also a requirement that you follow the programming policy for the advanced work. For example, if you include methods then you must comment the methods as stated in the policy.
  + Finally, simple clean code is the expectation in industry not code that is overly complicated for the task.  The thought process behind Agile development is “what is the simplest thing that works.”  Code that is clever/tricky/over done for the task is not acceptable.  Clean and clear – not clever code – is what will be excepted.

**Today’s Terminology**

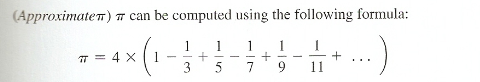
**Terminology**

* Flow of Control
  + The order in which statements are executed in a program
* Selection Statement
  + Way to make decision
  + If statements
  + Switch statements
* Iteration Statement
  + Way to repeatedly execute code
  + While loops
  + Do-while loops
  + For loops
* Loop Body
  + Statements that are repeated within the loop
* Loop Iteration
  + One complete execution of the loop body
* Infinite Loop
  + A loop that runs forever - stuck executing the body over and over because the condition tested never becomes false
* Sentinel Value
  + A value that signifies the end of a loop
* Pretest Loop
  + Loops where the continuation condition is checked **before** the loop body is executed
* Posttest Loop
  + Loops where the continuation condition is checked **after** the loop body is executed
* Coding Incrementally
  + A problem-solving approach
  + Write one part of the code - get it working and tested - then add a little more
  + Helpful with loops
* Casting
  + When *explicitly* tell the complier to convert a variable from one data type to another data type

**While Loops**

**Purpose**

* Usually some part of your algorithm needs to be execute repetitively.
* A "while-loop" is one programming structure that performs repeated execution (looping)
* What if you were asked to approximate PI using this formula using the 1st 10 fractions – not bad!



* What if I asked you to add the next 1000 fractions to the equation?
* If you had to do it manually - major pain!
* If you use computer - easy!

**General Form**

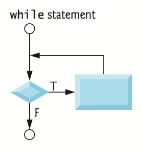
* A while loop has the following general form:

while (boolean expression) { // loop condition

statement(s); // loop body

}

* Flowchart:



**How It Works**

* The loop condition - boolean expression - is evaluated
* If the condition is true, then the statements in the loop body are executed
* When execution of loop body statements is complete, control returns to the loop condition
* The loop condition is evaluated again
* When the loop condition is false, control is transferred to the 1st statement following the loop
* Note: if the loop condition evaluates to false the 1st time, the entire while loop is skipped

**Rules for While Loops**

* The loop condition must be a boolean expression
  + Boolean expression must be in parentheses
  + Boolean expressions are formed using **relational** and **logical** operators (chapter 4)
* Loop condition
  + Generally, some statement **before** the while loop "initializes" the loop condition to true
  + Some statement within the loop body must eventually change the condition to false
* If the condition is never changed to false, the program is forever stuck in the loop
  + This is called an "infinite loop"
* Curly braces are not necessary if only one statement in loop
  + But best practice is to always include curly braces

**Trace Simple While Loop**

* Example of how control flows through a while loop

**int** count = 1; // Initializes the loop control variable

**while** (count <= 5) {

**printf** ("The value of count is %d\n", count);

count = count + 1; // Changes the loop control variable

}

**Trace:**

count is initialized to 1

count is compared to 5 - is count less than or equal to 5 - result is true - control enters loop body

Display => **The value of count is 1**

count is incremented to 2

count is compared to 5 - is count less than or equal to 5 - result is true - control enters loop body

Display => **The value of count is 2**

count is incremented to 3

count is compared to 5 - is count less than or equal to 5 - result is true - control enters loop body

Display => **The value of count is 3**

count is incremented to 4

count is compared to 5 -is count less than or equal to 5 - result is true - control enters loop body

Display => **The value of count is 4**

count is incremented to 5

count is compared to 5 - is count less than or equal to 5 - result is true - control enters loop body

Display => **The value of count is 5**

count is incremented to 6

count is compared to 5 - is count less than or equal to 5 - result is false - loop ends!

**Example: Infinite While Loop**

* In this example, nothing in the loop body changes the value of the control variable

count = 1; // Initializes the loop control variable

**while** (count <= 5) {

**printf** ("The value of count is %d\n", count);

}

This is an infinite loop because (count <= 5) will always be true

Nothing changes the value of count in the loop body

If you accidentally create infinite loop, use terminate button (red square) to make it stop! (in Eclipse)



**Off by One Errors**

* Common issue with loops!
* Loop body executes one more or one less than expected
* Make sure your loop condition is executing the expected number of times
* Example: why is this off by one?

puts ("I'm going to count to five, ready set....");

count = 1;

**while** (count < 5) {

printf (count);

count++;

}

Displays

I'm going to count to five, ready set....

1

2

3

4

**Controlling Loop with Sentinel Value**

* Many times, we don't know how many times a loop needs to be executed
* But we may know that a certain value signifies the end of the loop
* This value is called the **sentinel value**
* Write a loop that reads a string of characters one character at a time and keeps a running count of the characters until a period is reached.

**unsigned** **int** characterCounter = 0;

**char** userInput;

**puts** ("Enter a string of characters followed by a period to end input");

**scanf** ("%c", &userInput);

**while** (userInput != '.') {

characterCounter = characterCounter + 1;

**scanf** ("%c", &userInput);

}

**printf** ("The number of characters entered = %d", characterCounter);

**Displays**

Enter a string of characters followed by a period to end input

abcdefg.

The number of characters entered = 7

**New Data Types**

* **unsigned int**
  + Use when you know for certain you only need numbers 0 or greater
* **float** 
  + Use when dealing with numbers with a decimal point

**Common Errors and Pitfalls**

* **Equality testing of floats**
  + Do **not** perform equality checking with floating point numbers in the loop control
  + Floating point numbers are an approximation
  + Checking for equality could lead to inaccurate results
  + In this example, there is no guarantee that item will ever be zero – it most likely is an infinite loop

**float** item = 1.0;

**float** floatingPoint = 0;

**while** (item != 0) {

floatingPoint = floatingPoint + item;

item = item - 0.1;

**printf** ("Item = %2.20f\n", item);

} // end while

Item = 0.89999997615814209000 // Values are starting to get off

Item = 0.79999995231628418000

Item = 0.69999992847442627000

Item = 0.59999990463256836000

Item = 0.49999991059303284000

Item = 0.39999991655349731000

Item = 0.29999992251396179000

Item = 0.19999992847442627000

Item = 0.09999992698431015000 // Off enough to miss 0 on next iteration

**Item = -0.00000007301569127094 // Yep 0.0 was missed so in infinite loop**

Item = -0.10000007599592209000

Item = -0.20000007748603821000

Item = -0.30000007152557373000

Item = -0.40000006556510925000

Item = -0.50000005960464478000

Item = -0.60000008344650269000

Item = -0.70000010728836060000

Item = -0.80000013113021851000

Item = -0.90000015497207642000

Item = -1.00000011920928960000

Item = -1.10000014305114750000

* **Wrong Placement of Semicolon** 
  + Placing a semicolon at the end of the while-clause creates an infinite loop - be careful!

**int** iteration = 1;

**while** (iteration <= 10)**;** {

**printf** ("Iteration = %d", iteration);

iteration = iteration + 1;

}

The above it equivalent to writing:

**int** iteration = 1;

**while** (iteration <= 10) {}

{

**printf** ("Iteration = %d", iteration);

iteration = iteration + 1;

}

**Casting**

**Casting**

* When you *explicitly* tell the complier to convert a variable from one data type to another data type

**Example**

* Numbers with **decimals** are called **floats**
* We defined **average** as a floating-point number since it will be a decimal number
* In lecture 2 we learned
  + Division with integers results in an integer; the fraction part is truncated
  + If we write

Average = total / counter;

* + - we will **NOT** get the correct average
    - total and counter are both integers so result of division is an integer
    - the decimal part would be truncated before result is stored in average (a float)
    - Example: average should be 74.33

Enter quiz grade or a -1 to end program

95

Enter quiz grade or a -1 to end program

53

Enter quiz grade or a -1 to end program

75

Enter quiz grade or a -1 to end program

-1

Class average is 74.00

* + If we write Casts total to a float

Average = **(float)** total / counter;

* + - we will get the correct average
    - we used a cast on total to convert it to a float just during the operation!

Enter quiz grade or a -1 to end program

95

Enter quiz grade or a -1 to end program

53

Enter quiz grade or a -1 to end program

75

Enter quiz grade or a -1 to end program

-1

Class average is 74.33

Notes:

* More about data types and casting will be in next lecture and chapter 5

**Augmented Assignment Operators – Increment and Decrement Operators**

**Augmented Assignment Operators**

* On Tuesday, we saw that we have the following operators

|  |  |
| --- | --- |
| Addition | + |
| Subtraction | - |
| Multiplication | \* |
| Division | / |
| Remainder | % |

* Each of these operators can be combined with the assignment operator
* For example, the addition operator (+) combined with the assignment operator (=) becomes +=

numStudents = numStudents + 1; Can be rewritten as

numStudents += 1;

**Increment and Decrement Operators**

* Two more operators to add to our list
  + Increment ++
  + Decrement --
* Operator can be placed before or after variables
* Example (**prefix** increment/decrement)

int i = 1;

int j = 3;

++i; // Same as i = i + 1; i will become 2

--j; // Same as j = j – 1; j will become 2

* Example (**postfix** increment/decrement):

int i = 1;

int j = 3;

i++; // Same as i = i + 1; i will become 2

j--; // Same as j = j – 1; j will become 2

* Placement of operator (prefix or postfix) cause different results **in an expression** so be careful!!!
* Prefix increment vs Postfix increment

int x = 1;

int y = 3;

y = ++x; // do increment **then** assignment => y becomes 2, x becomes 2

y = x++; // do assignment **then** increment => y becomes 1, x becomes 2

* Examples in expressions
  + **Simple example**

**int** x, j, k;

x = 10;

j = 20;

k = x-- + (x + j); // k = 39, x = 9

* + **How it can get confusing**

**int** numDays = 100;

**printf** ("%d\n", numDays); // 100

**printf** ("%d\n", ++numDays); // 101

**printf** ("%d\n", numDays++); // 101

**printf** ("%d\n", numDays); // 102

Use sparingly – can make expressions difficult to read and complex

**Fitting into Order of Operators**

* Anything in parentheses
* expr++ expr-- (postfix)
* ++expr --expr (prefix)
* \* / % (multiplication, division, remainder)
* + - (addition, subtraction)
* **< <= > >= (relational operators)**
* **== != (equality)**
* = += -= \*= /= %= (assignment, augmented assignment)

**Case Study – Sections 3.8, 3.9, 3.10**

In lecture 4 we talked about algorithms and pseudocode

**Algorithm vs Pseudocode**

* Algorithm is the set of steps needed to solve a problem
* Pseudocode is the language to represent the algorithm

**Top-Down, Stepwise Refinement**

* Also, called divide and conquer
* Strategy to design code
* Breaking the code down into subcomponents
* Benefits
  + Simpler program
  + Reusing functions
  + Easier development, debugging, and testing
  + Facilitates teamwork

**Example**

* Problem from Section 3.9
  + Develop a class-average program that will process an arbitrary number of quiz grades each time the program runs.
* **Start with a top-level statement**

*Determine the class average for the quiz*

* **1st refinement** - break the statement down into a few refined steps

*Initialize variables  
Input, sum, and count the quiz grades  
Calculate and print the class average*

* **2nd refinement** - break each of the steps in 1st refinement down
  + *Initialize variables* 
    - Refine this step into more specific details – what variable need to be initialized?

*Initialize total to zero   
Initialize counter to zero*

* + *Input, sum, and count the quiz grades*
    - Looking at this step we need a way to repeatedly obtain the quiz grades
    - We will need a loop

*Input the first grade  
While the user has not as yet entered the sentinel  
 Add this grade to the running total  
 Add one to the grade counter  
 Input the next grade*

* + *Calculate and print the class average*

*If the counter is not equal to zero  
 Set the average to the total divided by the counter  
 Print the average  
else  
 Print “No grades were entered”*

* **Final pseudocode**

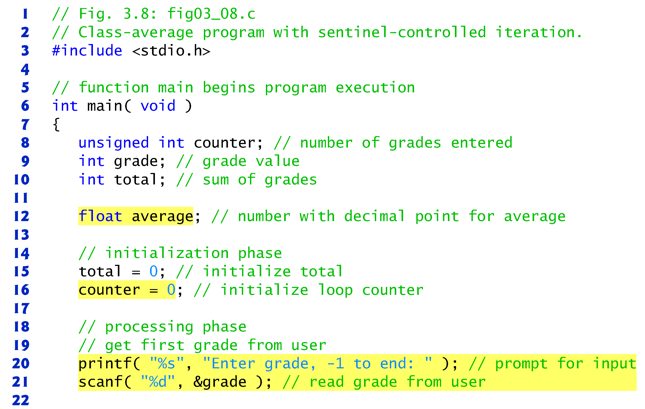
*Initialize total to zero   
Initialize counter to zero*

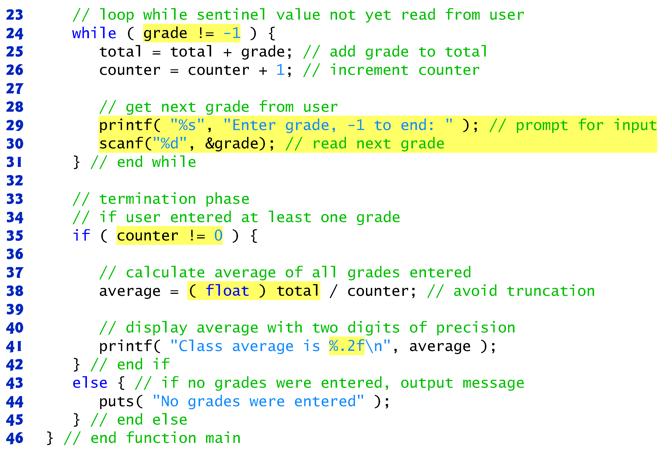
*Input the first grade  
While the user has not as yet entered the sentinel  
 Add this grade to the running total  
 Add one to the grade counter  
 Input the next grade*

*If the counter is not equal to zero  
 Set the average to the total divided by the counter  
 Print the average  
else  
 Print “No grades were entered”*

**Notes**

* + This last step is the type of pseudocode I expect with your assignments
  + It is very English like – it is NOT code like
* **C Code**





**Another Pseudocode Example**

Here is another example that was not covered in class.

**Example 1:**

Write a program that prompts the user to enter three points (x1, y1), (x2, y2), (x3, y3) of a triangle and displays the triangle's area. The formula for computing the area of a triangle is:

s = (side1 + side 2 + side3)/2

area =

**Pseudocode**

* **Start with a top-level statement**

*Determine the area of a triangle given 3 points*

* **1st refinement** - break the statement down into a few refined steps

*Initialize variables*

*Input 3 points and calculate area of triangle*

*Print the 3 points and the area of the triangle*

* **2nd refinement** - break each of the steps in 1st refinement down
  + *Initialize variables* 
    - Refine this step into more specific details – what variables need to be initialized?

*Initialize area to zero*

*Initialize side1 to zero*

*Initialize side2 to zero*

*Initialize side3 to zero*

*Initialize area to zero*

* + *Input 3 points and calculate area of triangle*

*Prompt user for the 1st point*

*Input the 1st point*

*Prompt user for the 2nd point*

*Input the 2nd point*

*Prompt the user for the 3rd point*

*Input the 3rd point*

*Set* ***side1*** *to distance between 1st and 2nd points*

*Set* ***side 2*** *to distance between 2nd and 3rd points*

*Set* ***side 3*** *to distance between 1st and 3rd points*

*Set semi-perimeter s (side1 + side2 + side 3) / 2*

*Set area to*

* + *Print the 3 points and the area of triangle*

*Print the 1st point, 2nd point, and 3rd point*

*Print the area of triangle*

* **Final pseudocode**

*Initialize area to zero*

*Initialize side1 to zero*

*Initialize side2 to zero*

*Initialize side3 to zero*

*Initialize semi-perimeter s to zero*

*Prompt user for the 1st point - (x1,y1)*

*Input the 1st point*

*Prompt user for the 2nd point - (x2,y2)*

*Input the 2nd point*

*Prompt the user for the 3rd point - (x3,y3)*

*Input the 3rd point*

*Set* ***side1*** *to distance between 1st and 2nd points*

*Set* ***side 2*** *to distance between 2nd and 3rd points*

*Set* ***side 3*** *to distance between 1st and 3rd points*

*Set semi-perimeter s (side1 + side2 + side 3) / 2*

*Set* *area to*

*Print the 1st point, 2nd point, and 3rd point*

*Print the area of triangle*

* The following pseudocode is too code-like

area = 0;

side1 = 0;

side2 = 0;

side3 = 0;

s = 0;

Prompt user for the three points of a triangle

scanf (“%d”, x1);

scanf (“%d”, y1);

scanf (“%d”, x2);

scanf (“%d”, y2);

scanf (“%d”, x3);

scanf (“%d”, y3);

side1 = compute length of (x1,y1) and (x2,y2)

side2 = compute length of (x2,y2) and (x3,y3)

side 3 = compute length of (x1,y1) and (x3,y3)

s = (side1 + side2 + side3)/2

area =

print (x1,y1), (x2,y2), (x3,y3)

print area

Don’t just copy your code and call it your pseudocode!

* **C Code**

**float** x1, y1;

**float** x2, y2;

**float** x3, y3;

**puts** ("Enter three points for a triangle");

**puts** ("1st point (x1,y1)");

**puts** ("Enter 1st point x-coordinate");

**scanf** ("%f", &x1);

**puts** ("Enter 1st point y-coordinate");

**scanf** ("%f", &y1);

**puts** ("2nd point (x2,y2)");

**puts** ("Enter 2nd point x-coordinate");

**scanf** ("%f", &x2);

**puts** ("Enter 2nd point y-coordinate");

**scanf** ("%f", &y2);

**puts** ("3rd point (x3,y3)");

**puts** ("Enter 3rd point x-coordinate");

**scanf** ("%f", &x3);

**puts** ("Enter 3rd point y-coordinate");

**scanf** ("%f", &y3);

// Computer the length of the 3 sides

// Function pow(x,y) calculates the value of x raised to the yth power

**float** side1 = **pow** ((x1-x2) \* (x1-x2) + (y1-y2) \* (y1-y2), 0.5);

**float** side2 = **pow** ((x1-x3) \* (x1-x3) + (y1-y3) \* (y1-y3), 0.5);

**float** side3 = **pow** ((x3-x2) \* (x3-x2) + (y3-y2) \* (y3-y2), 0.5);

**float** s = (side1 +side2 + side3)/2;

**float** area = **pow** ((s \* (s-side1) \* (s-side2) \* (s-side3)), 0.5);

**printf** ("Area of triangle with points

(%.2f,%.2f), (%.2f,%.2f), and (%.2f,%.2f) is %.2f\n",

x1, y1, x2, y2, x3, y3, area);

**Displays**

Enter three points for a triangle

1st point (x1,y1)

Enter 1st point x-coordinate

1.5

Enter 1st point y-coordinate

-3.4

2nd point (x2,y2)

Enter 2nd point x-coordinate

4.6

Enter 2nd point y-coordinate

5

3rd point (x3,y3)

Enter 3rd point x-coordinate

9.5

Enter 3rd point y-coordinate

-3.4

Area of triangle with points (1.50,-3.40), (4.60,5.00), and (9.50,-3.40) is 33.60

**Secure C Programming**

**Secure Programming**

* To write code that uses techniques that can stand up to attacks
* This topic is an entire class so we won’t be focusing on this topic
* We will discuss some of the techniques

**CERT C Secure Coding Standard**

* CERT – Computer Emergency Response Team - [www.cert.org](http://www.cert.org)
* Publishes and promotes secure coding standards
* Standard for C
  + <https://www.securecoding.cert.org/confluence/display/c/SEI+CERT+C+Coding+Standard>
* Standard for other languages:
  + <https://www.securecoding.cert.org/confluence/display/seccode/SEI+CERT+Coding+Standards>

**Arithmetic Overflow**

* When you perform addition and the result is too large for the variable -> arithmetic overflow
* Can leave systems open to attack
* Make sure your arithmetic calculations do not cause overflow!
* If interested, there is code on the CERT site for doing this
  + Search INT32-C

**Unsigned Integers**

* Use unsigned integers for any integer values that store non-negative values

**More secure versions of printf and scanf**

* The C11 standard introduces:
  + printf\_s
  + scanf\_s
* These are optional so some implementations of C may not have these functions