**Lecture 6 - Chapter 4: C Program Control – Mon Sept 11 or Tues Sept 12**

**Announcements**

Reading:

* Chapter 4

Assignments:

* Due: Assignment #3 is due on **Sept 13** (MW class) or **Sept 14** (T/R class) **(no late assignments accepted)**

Quiz #1:

* Sept 20th (MW) or Sept 21 (TR)

**Today’s Goals**

1. Iteration Essentials
2. Counter Controlled Iteration
3. For Loop
4. Switch Statement

**Today’s Terminology**

**Terminology**

* Iteration - Looping
  + Way to repeatedly execute code
  + While loops
  + Do-while loops
  + For loops
* Counter-controlled Iteration
  + Loop for a known number of iterations
  + Also called *definite iteration*
* Sentinel-controlled Iteration
  + Loop until a certain value is reached
  + Also called *indefinite iteration*
* 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
* Variable Scope
  + The area of a program a variable can be referenced.
* 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

**Iteration Essentials**

**Types of Loops**

* Counter Controlled Loops
  + When the number of iterations is known before the loop starts
  + A **counter** variable is tested in the loop condition and incremented in the loop body
  + Generally written with for-loops
* Sentinel Controlled Loops
  + When the number of iterations is NOT known before the loop starts
  + A specified value is tested for in the loop condition and obtained in the loop body
  + The sentinel value must be distinct from the rest of the data examined in the loop
  + Written with while-loops and do-while loops

**Counter Controlled Iteration**

**What’s Required**

* Loop counter - a variable serves as the loop counter
* Initial starting point - loop counter is initialized to some starting value
* Modification of loop counter – loop counter is incremented or decremented in loop body
* Loop counter test - determination if we’ve iterated the desired number of times

**Example**

**unsigned** **int** count = 1; // Counter set to starting value

**while** (count <= 5) { // Loop performs 5 iterations

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

++count; // Counter incremented

}

**Output**

The value of count is 1

The value of count is 2

The value of count is 3

The value of count is 4

The value of count is 5

**For Loop**

**Purpose**

* Same purpose as while loops => a programming structure that performs repeated execution (looping)
* For loops are used to execute a loop a **preset** number of times – counter-controlled!
* For loops are built to handle all the countered controlled details automatically

**General Form**

* A for loop has the following general form:

for (initialize control; test control against end value; update control) {

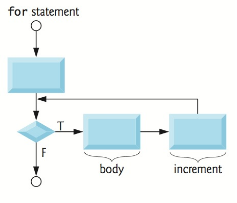
statement;

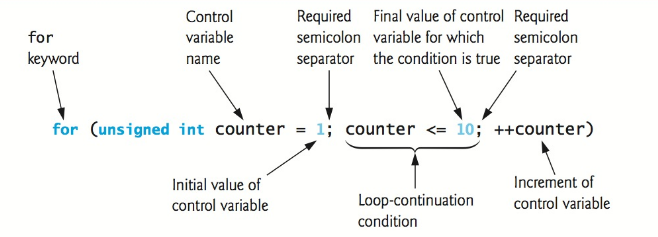
...

statement;

}

* Flowchart:





**How It Works**

1. The control variable is initialized to a start value
2. The control variable is compared to the end value
3. If the test of the control variable evaluates to true then **loop body is executed**
4. The control variable is updated **after** loop body completes
5. Steps 2-4 are repeated as long as control test evaluates to true

**Rules for For-Loops**

* The control structure of the for-loop needs to be in parentheses

for **(i=0; i<= 5; i++)** { statements; }

* The loop condition (i <= 5) must be a boolean expression
* Curly braces are not necessary if only one statement in loop
  + Best practice is to always include curly braces

**Trace Simple For-Loop**

* Example of how control flows through a for-loop

**unsigned** **int** i; // Declaring loop control outside loop

**for** (i = 1; i <= 3; i++) {

**printf** ("This is iteration %u of the for loop\n", i);

}

i is initialized to 1

i is compared to 3 - result is true - control enters loop body

Display => This **is iteration 1 of the for-loop**

i increments to 2

i is compared to 3 - result is true - control enters loop body

Display => **This is iteration 2 of the for-loop**

i increments to 3

i is compared to 3 - result is true because i <= 3 - control enters loop body

Display => **This is iteration 3 of the for-loop**

i incremented to 4

i is compared to 3 - result is false - loop ends!

**Notes**

* You may declare the control variable within the for-loop

**for** (**unsigned** **int** i = 1; i <= 3; i++) {

**printf** ("This is iteration %u of the for loop\n", i);

}

* Variable scope
  + Scope - the area of a program a variable can be referenced.
  + Declaring control variable **before the for loop** cause its **scope** to be inside and outside for-loop
  + Declaring the control variable **in the for-loop** causes its **scope** to be only inside the for loop
    - If I tried to use the variable i outside the for-loop - complier error
    - Declare control variables in for-loop when control variable is only used in for-loop
* Expressions can be used for any of the 3 parts - expression must evaluate to same type as control variable

**unsigned** **int** startValue = 0;

**for** (**unsigned** **int** i = startValue+5; i <= 10; i++) {

**printf** ("For loop iteration = %u\n", i);

}

**Displays**

For loop iteration = 5

For loop iteration = 6

For loop iteration = 7

For loop iteration = 8

For loop iteration = 9

For loop iteration = 10

* The control variable **should not** be changed within the for-loop body
  + Not prevented by complier but not good practice
  + Control variable automatically changed by for-loop
  + Leads to unpredictable results
* These two options are legal but generally not done in practice
  + The **initial-control** can be a list of zero or more comma -separated expression
  + The **update-control** can be a list of zero or more comma-separate statements

**unsigned** **int** x;

**unsigned** **int** y; Legal but not recommended

**for** (i=0, j=0; i+j <= 5; i++, j++) {

**printf** ("i = %u and j = %u\n", i, j);

}

**Displays**

i = 0 and j = 0

i = 1 and j = 1

i = 2 and j = 2

* It is perfectly legal to leave off any of the 3 parts of the for-loop
  + The 3 expressions in the for loop are optional
  + If the loop condition is left off, it is implicitly true and we get an infinite loop

**for** (**unsigned** k = 0; ; k++) {

**printf** ("The value of k is %u\n", k);

}

**unsigned** **int** k = 0;

**for** (; ;) {

**printf** ("The value of k is %u\n", k);

}

* Adding semicolon to end of the for-loop clause causes logic errors

**unsigned** **int** iteration;

**for** (iteration = 0; iteration <= 10; iteration++); {

**printf** ("For loop iteration with misplaced semicolon %u\n", iteration);

}

**Displays**

For loop iteration = 11

This next case can drive you crazy because it causes a compiler error! “Symbol Iteration could not be resolved” Why?

**for** (**unsigned** **int** iteration = 0; iteration <= 10; iteration++); {

**printf** ("For loop iteration %u\n", iteration);

}

* For-loop is the **one** place in your code where variable names like "i" or "j" are allowed and acceptable!

**Switch Statement**

**Purpose**

* Like If-Statements, provides ability to select a path of execution from a **group** of possibilities
* Good when for multiple selection situations

**General Form**

switch (expression) {

case value1: statement(s);

break;

case value2: statement(s);

break;

. . .

case valueN: statement(s);

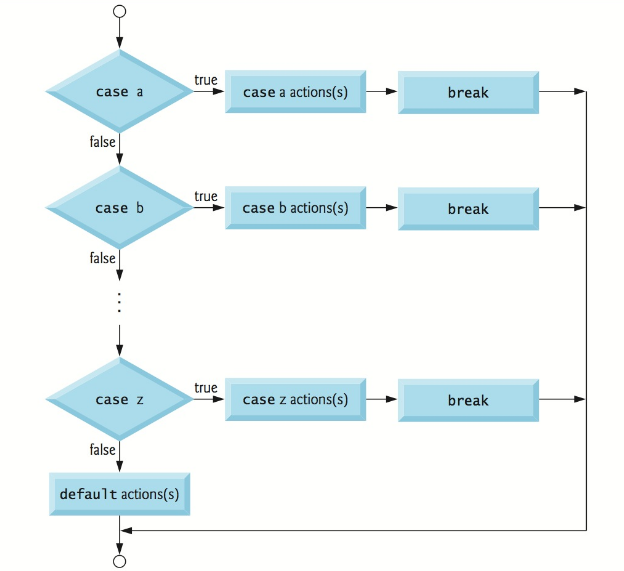
break;

default: statement(s); // optional

}

**How it Works**

* The switch "expression" is evaluated
* If one of the cases "values" matches the value of the switch "expression"
  + The statements for that case are executed
  + Execution stops **when a break statement** is reached or the end of the switch statement is reached
  + Break statements cause program control to continue with the 1st statement after the switch
* If no case "values" match the value of the switch "expression"
  + a default case is executed if it exists
* This flowchart is a good visual of the switch statement with breaks:

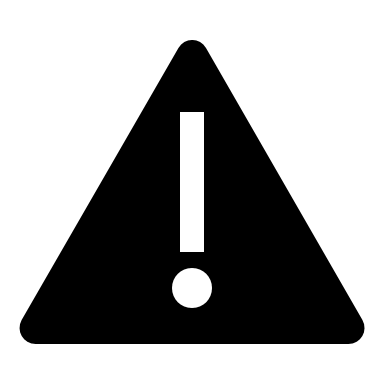


**Rules for Switch Statements**

* The switch "expression"
  + must always be in parentheses
  + must evaluate to a value of type **char, short, int, long**

switch (x > 1) // Not allowed - evaluates to a boolean value

switch (x == 2) // Not allowed - another boolean expression

The above two cases produce a **warning** in Eclipse 

The code still runs but **no cases will match in switch statement except default**!

switch (x = 2) // Allowed because this evaluates to 2 an **int**

* + the date type of the "expression" must match data type of “value1", "value2", etc.
    - Remember **char** can be stored in type **int**
    - If switch expression is type **int,** cases can be type **char**

**unsigned** **int** grade = 70;

**switch** (grade) {

**case** 'a':

**puts** ("You got a 90");

**break**;

**case** 'b':

**puts** ("You got a 80");

**break**;

**case** 70:

**puts** ("You got a 70");

**break**;

**default**:

**puts** ("No matches");

} // end switch

* The case values - "value1", "value2", etc.
  + are integer constants expressions
  + they cannot contain variables
  + they cannot contain expressions

case 0: // valid

puts ("......");

case x: // not valid – cannot be variable

puts ("....");

case (x+1): // not valid – cannot be expression

puts ("....");

* + this is valid way to write the cases

**switch** (value) {

**case** 1: **case** 2: **case** 3:

**puts** ("Case is 1, 2, or 3");

**break**;

**case** 4:

**puts** ("Case 4");

**break**;

**default**:

**puts** ("No matching case");

} // end switch

or

**switch** (value) {

**case** 1:

**case** 2:

**case** 3: **puts** ("Case 1, 2, or 3"); **break**;

**case** 4: **puts** ("Case 4"); **break**;

**default**: **puts** ("No matching case");

} // end switch

* + Note that **braces are not required** around multiple case action statements
* The "break" statement
  + is optional
  + ends a case
  + in most cases, each case statement should have a break to terminate the reminder of the switch
  + once a case is entered execution contains **until** a break occurs
  + if the case that was entered does not contain a break - the code continues to next case
  + we say the code **falls through** to next case

// Fall through behavior

**unsigned** **int** day = 0;

**switch** (day) {

**case** 1:

**case** 2:

**case** 3:

**case** 4:

**case** 5: **puts** ("Weekday"); **break**;

**case** 0:

**case** 6: **puts** ("Weekend"); **break**;

**default**: **puts** ("Something is not right");

} // end switch

**Displays**

Weekend

* "The "default" statement
  + is optional
  + used to perform some action if none of the case match the switch expression

**Which Selection Statement Should I Use - If or Switch?**

* Use an If statement
  + Need to test for boolean conditions
* Use a Switch statement
  + Situations where looking for a constant value and the cases are simple

**Character Input**

**Char Data Type**

* Values: one single character
* Use single quote to represent a character (doubles quotes are for Strings)

char middleInitial = 'M';

char numCharacter = '4'; // Assigns the digit character 4 to numCharacter

printf(“%c”, numCharacter); // Displays the character 4

printf(“%d”, numCharacter); // Display 52 - the ASCII code for the character 4

**Unicode and ASCII Code**

* Encoding
  + When you encode you are using something to represent something else
  + With Unicode and ASCII code a **binary** representation maps to a **char**

01100010 represents the char 'b' (ASCII coding scheme - binary representation)

01100010 = 98 (2^7 + 2^6 + 2^5 + 2^4 + 2^3 + 2^2 + 2^1 + 2^1)

0 + 1 + 1 + 0 + 0 + 0 + 1 + 0

64 32 2

64+32+2=98

* + ASCII chart in Appendix B - shows where lowercase 'b' is represented by the decimal number 98
* Unicode and ASCII code are two encoding schemes
* ASCII (American Standard Code for Information Interchange)
  + 8-bit character scheme - shown above for the char 'b'
  + Provides encoding for 128 characters
  + Based on ordering of English alphabet
  + Superseded by UTF-8
* Unicode
  + 16-bit character scheme (which can store the **char** data type)
  + Encodes every possible language - so huge!
  + Requires more space than ASCII
  + Includes ASCII code
  + UTF-8 one of the widely-used encodings
  + Basically, Unicode is a way to assign to each character a unique number, or code point
  + Want more information - Read http://kunststube.net/encoding/

**Input Example – using getchar()**

* Use function getchar() to read on character from keyboard
* Found in stdio.h
* It returns the character the user entered as an **int**!
* Characters normally stored in variables of type **char**
* Characters can also be stored in integer data types because they are represented as one-byte integers
* Characters can be treated as either **int** or **char** in C
  + depending on how you want to use it
  + if as character then use **char**

**puts** ("Enter a character");

**char** someCharacter = **getchar**();

**printf** ("The value entered by user was %c\n",someCharacter);

Displays

Enter a character

b

The value entered by user was b

* + if as the character’s numeric representation (ASCII code) as **int**

**puts** ("Enter a character");

**int** someInteger = **getchar**();

**printf** ("The value entered by user was %d\n",someInteger);

**Displays**

Enter a character

b

The value entered by user was 98