Variables and Operators

2/20/01  Lecture #6  16.070

- Variables, their characteristics and their uses
- Operators, their characteristics and their uses
Variables

- **Variables** enable you to associate a name with a memory location
- Variables hold values upon which a program acts
- Variables have 3 characteristics
  - **Type** indicates properties of values contained in variable-- int, char, double
  - **Identifier** -- symbolic, "descriptive" name
    - Allows programmer to refer to values symbolically, by name vs by storage location
    - Most high-level languages have rules for choosing variable names
    - Letters (upper vs lower case), numbers, special characters (e.g., "_")
  - **Scope** - implicitly indicates to compiler where variable is accessible within program
Variable Declaration

• All variables (in C) must be defined prior to use
  ➢ Format:
    
    <type> <identifier>;

  ➢ Examples:
    
    int thrust_command; /*expected range: >0 */
    declares variable called thrust_command
    that will contain an integer value

    float meters_per_second; /*expected range: -3.2e4 to 3.2e4 */
    declares variable called
    meters_per_second
    that will contain a float value

• Compiler reserves memory locations for variables at compile time
• Operating system creates memory locations for each variable and
  ensures that values are appropriately stored in them
Variable Initialization

• Set up initial value of variable

• Without initialization, memory location contains garbage value
  \[ \text{int thrust\_command; \hspace{1cm} 100 \quad 01100010} \]

• Compile-time initialization
  ➢ Included in variable declaration
  ➢ At run time, OS assigns reserved location to variable and places defined
    value in memory location
  \[ \text{int thrust\_command = 0; \hspace{1cm} 100 \quad 00000000} \]

• Run-time initialization
  \[ \text{int thrust\_command; \hspace{1cm} 100 \quad 01010010} \]
  \[ \text{thrust\_command = 0; \hspace{1cm} 100 \quad 00000000} \]
Constant Variables

- **Constants** are variables whose values do not and can not change during execution of program

- Value specified during variable declaration - Compile-time initialization
  
  - Defined by the `const` qualifier before the type specifier
    
    ```
    const double PI = 3.14159;
    ...
    area = PI * radius * radius;
    circumference = 2 * PI * radius;
    ```

- Why use constants?
  
  - Readability
  
  - Maintenance - easier to change
Variable Scope

• Region of program where variable is accessible
• Specified implicitly by the place in the code where declaration occurs
• **Global variables** are known to several functions
  ➢ Declared at front of file, outside of functions, outside of all blocks `{}`
  ➢ Visible to, and usable by, all functions
  ➢ Can be accessed and modified from any block within the program
  ➢ Retain storage throughout duration of program

• **Local variables** are known only to the functions containing them
  ➢ Declared at the beginning of a block `{}`
  ➢ Accessible only within the block in which they are declared
  ➢ Lose storage and value after block execution completes
Global and Local Variables

- Pros and cons for using global vs local variables

<table>
<thead>
<tr>
<th></th>
<th>Pro</th>
<th>Con</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Global Variables</strong></td>
<td><strong>Can produce faster code</strong></td>
<td><strong>Increase program's complexity</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Difficult to maintain - look in every source file to see if variable is used</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Potential conflict between modules</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>Undermines modularity</strong></td>
</tr>
<tr>
<td><strong>Local Variables</strong></td>
<td><strong>Interfaces between modules clearly defined</strong></td>
<td><strong>Parameter lists can be long</strong></td>
</tr>
<tr>
<td></td>
<td><strong>In some languages (e.g., Ada), variable use specified - input/output/both</strong></td>
<td></td>
</tr>
</tbody>
</table>

- Rule of thumb: use global variables only when timing warrants it
Global and Local Variables

```c
char user_input; /* global variable */
int main(void)
{
    ...
    user_input = print_menu(void);
    move_robot (user_input);
    ...
}

void move_robot(void)
{
    ...
    /* move robot in direction */
    /* specified by passed */
    /* argument */
    ...
}
```
A View of Memory

- Memory organization when a program is running

<table>
<thead>
<tr>
<th>X000</th>
<th>⋮</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Memory-mapped I/O and DMA</td>
</tr>
<tr>
<td></td>
<td>Program Area</td>
</tr>
<tr>
<td></td>
<td>contains executable code of program</td>
</tr>
<tr>
<td></td>
<td>- OS</td>
</tr>
<tr>
<td></td>
<td>- applications software</td>
</tr>
<tr>
<td></td>
<td>- constants</td>
</tr>
<tr>
<td></td>
<td>Global Data Section</td>
</tr>
<tr>
<td></td>
<td>Run-Time Stack</td>
</tr>
<tr>
<td></td>
<td>For context saving and automatic variables</td>
</tr>
<tr>
<td></td>
<td>⋮</td>
</tr>
</tbody>
</table>
Operators

- **Operators** are tokens that result in computation or action when applied to variables or other elements in an expression
  - Assignment
  - Arithmetic
  - Relational
  - Logical (Chapter 9)
  - Bitwise manipulation (listed in Appendix B)
Assignment Operator

• Arithmetic Expression consists of operators and operands
  1) 8  
  2) 5 + x  
  3) steps = 10

• Arithmetic Expressions do not always change values in memory
  ➢ What effect will this expression have?
    
    \[(a + b) \times c;\]

  ➢ What effect will this expression have?
    
    \[\text{thrust} = (a + b) \times c;\]

  ➢ Assignment operator '=' does not mean "equals"!
  ➢ '=' assigns the value that is calculated on RHS to variable on LHS
    
    \[i = i + 1;\]
Arithmetic Operators

• C uses operators (~40) to represent arithmetic operations (see Appendix B)

• Operators act on operands and result in computation or action
  ➢ Unary, requires one operand
    ♦ + Plus, has no effect on result
    ♦ - Minus, reverses the sign of the operand
    ♦ ++ Increment
    ♦ -- Decrement
  ➢ Binary, requires two operands (integer or float)
    ♦ + Addition, adds the two operands
    ♦ - Subtraction, subtracts the second operand from the first
    ♦ * Multiplication, multiplies the two operands
    ♦ / Division, divides the first operand by the second operand
    ♦ % Modulus, equals remainder of first operand divided by second. Operands must be integers
Increment Operator (Analogous comments for Decrement)

• Increment operator increments value of operand by 1
  \[ \text{counter}++ \Leftrightarrow \text{counter} = \text{counter} + 1 \]

• Operand must be variable
  \[ 10++ \text{ ??} \]

• Pre-increment/Prefix mode - increment operand and then use the result
  \[
  \text{counter} = 3; \\
  \text{printf} \left( "\%d", ++\text{counter} \right); \Rightarrow 4 \\
  \text{printf} \left( "\%d", \text{counter} \right); \Rightarrow 4
  \]

• Post-increment/Suffix mode - use operand and then increment operand
  \[
  \text{counter} = 3; \\
  \text{printf} \left( "\%d", \text{counter}++ \right); \Rightarrow 3 \\
  \text{printf} \left( "\%d", \text{counter} \right); \Rightarrow 4
  \]
Side Effects of Increment and Decrement Operators

- Increment/Decrement operators cause **side effects**: change the value of a variable as well as result in a value.

- Not always possible to predict order in which side effects occur.
  - Caution: order of argument evaluation is not specified
    
    ```c
    newval = oldval + oldval++;  
    ```

  - C language does not specify which multiplication operand to evaluate first.

- Rule of thumb:
  - Avoid operators that have side effects.
  - If using a side effect operator in an expression, do not use the affected variable anywhere else in the expression
    
    ```c
    newval = oldval + oldval;  
    ++oldval;  
    ```
Operator Precedence Property (Section C4.2)

- All programming languages have Precedence Rule
- Precedence Rule defines order in which different operators are applied
  - Square (not represented in C)
  - Unary (++ and --)
  - Unary (+ and -)
  - Multiplicative (* and /)
  - Additive (+ and -)
    - i) \( x = 2 + 3 \times 4 \)
    - ii) \( \text{area} = 3.14159 \times \text{radius} \times \text{radius} \)
    - iii) \( 2 + 3 \times 4 \Leftrightarrow 3 \times 4 + 2 \)
Operator Associativity Property (Section C4.2)

- All programming languages have Associativity Rule
- Associativity Rule specifies order in which same operators are applied
  - **Left associativity** means from left to right; e.g., in C:
    i) \( a \times x \times x + b \times x + c \) evaluated left to right
    
    ii) \( x = 2 + 3 - 4 + 5; \) evaluated left to right
    
    iii) \( \text{area} = 3.14159 \times \text{radius} \times \text{radius}; \) evaluated left to right
  
  - **Right associativity** means from right to left; e.g., in C:
    i) \( x = y = 3; \) evaluated right to left
Operator Precedence and Associativity

- Recommendation: Do not rely on your human memory for operator precedence!
- Parentheses override evaluation rules by specifying explicitly which operations to perform first
- Evaluation always begins at the innermost set of parentheses
- Use parentheses to force desired operator precedence
  - Ensures correct execution: $a + b * c \Rightarrow (a + b) * c$ or $a + (b * c)$?
  - Improves readability/maintainability -- stylistically preferable!
    $(a * (x **2)) + (b * x) + c$
Side Effects of Increment and Decrement Operators

• Increment/Decrement operators cause side effects: change the value of a variable as well as result in a value

• Not always possible to predict order in which side effects occur

  ➢ Caution: order of argument evaluation is not specified
    newval = oldval + oldval++; /*eval LH or RH operand 1st? */

  ➢ C language does not specify which multiplication operand to evaluate first

• Rule of thumb:

  ➢ Avoid operators that have side effects -- they are not portable

  ➢ If using a side effect operator in an expression, do not use the affected variable anywhere else in the expression:
    newval = oldval + oldval;

    ++oldval;
Relational Operators

- Relational Operators are used to test relationship between two values
- Result of comparison is an integer, either 1 (≡TRUE) or 0 (≡FALSE)
- Any non-zero result is evaluated as TRUE
- All relational operators:
  - "<"   less than
  - ">"   greater than
  - ">="   greater than or equal to
  - ">="   less than or equal to
  - ">="   equal to
  - ">="   equal to
  - ">="   equal to
- Example: test for equality ">="
  
x = 2;
y = (x == 2);
if (x = 3)
calculate_thrust(void);

➤ Is y equal to 0 or 1 or 2?
➤ Will calculate_thrust execute? (Careful!)
Review

- Described the concept of variables and can use variables in C
- Described the concept of operators and can use operators in C

Readings:
  - For Wednesday, read C4
  - For Friday, read C9

Reminder: No lab sessions this week