Chapter 12
Variables and Operators

Original slides from Gregory Byrd, North Carolina State University
Modified slides by Chris Wilcox, Colorado State University

Basic C Elements

Variables
- named, typed data items

Operators
- predefined actions performed on data items
- combined with variables to form expressions, statements

Rules and usage
- Implementation using LC-3 instructions

Data Types
- C has three basic data types
  - int: integer (at least 16 bits)
  - double: floating point (at least 32 bits)
  - char: character (at least 8 bits)

- Exact size can vary, depending on processor
  - int is supposed to be "natural" integer size, for LC-3 that's 16 bits, LC-3 does not have double
  - int on a modern processor is usually 32 bits, double is usually 64 bits

Variable Names
- Any combination of letters, numbers, and underscore (_)
- Case matters
  - "sum" is different than "Sum", this is also true of function names
- Cannot begin with a number
  - usually variables beginning with underscore are used only in special library routines
- Restricted length?
  - compiler dependent, older implementations recognized as few as 31 characters
Examples

*Legal*
- i
- wordsPerSecond
- words_per_second
- _green
- aReally_longName_moreThan31chars
- aReally_longName_moreThan31characters

*Illegal*
- 10sdigit
- ten'sdigit
- done?
- double

Scope: Global and Local

- Where is the variable accessible?
  - **Global**: accessed anywhere in program
  - **Local**: only accessible in a particular region
- Compiler infers scope from where variable is declared in the program
  - programmer doesn't have to explicitly state
- **Variable is local to the block in which it is declared**
  - block defined by open and closed braces {}
  - can access variable declared in any "containing" block
  - global variables are declared outside all blocks

Examples

```c
#include <stdio.h>
int itsGlobal = 0;
main()
{
  int itsLocal = 1; /* local to main */
  printf("Global %d Local %d\n", itsGlobal, itsLocal);
  int itsLocal = 2; /* local to this block */
  itsGlobal = 4; /* change global variable */
  printf("Global %d Local %d\n", itsGlobal, itsLocal);
  printf("Global %d Local %d\n", itsGlobal, itsLocal);
}
```

Output

```
Global 0 Local 1
Global 4 Local 2
Global 4 Local 1
```
Operators

- Programmers manipulate variables using the operators provided by the high-level language.
- Variables and operators combine to form expressions and statements.
- These constructs denote the work to be done by the program.
- Each operator may correspond to many machine instructions.
  - Example: The multiply operator (*) typically requires multiple LC-3 ADD instructions.

Expression

- Any combination of variables, constants, operators, and function calls
  - every expression has a type, derived from the types of its components (according to C typing rules)
- Examples:
  - `counter >= STOP`
  - `x + sqrt(y)`
  - `x & x + 3 || 9 - w-- % 6`

Statement

- Expresses a complete unit of work
  - executed in sequential order
- Simple statement ends with semicolon
  - `z = x * y; /* assign product to z */`
  - `y = y + 1; /* after multiplication */`
  - `; /* null statement */`
- Compound statement groups simple statements using braces.
  - syntactically equivalent to a simple statement
    - `{ z = x * y; y = y + 1; }`

Operators

Three things to know about each operator:

1. **Function**
   - what does the operator do?
2. **Precedence**
   - in which order are operators combined?
     - Example: `a * b + c * d` is the same as `((a * b) + (c * d))` since multiply has higher precedence than addition
3. **Associativity**
   - in which order are operators of the same precedence combined?
     - Example: `a - b - c` is the same as `(a - b) - c` because add and subtract associate left-to-right
Assignment Operator

- Changes the value of a variable.

\[ x = x + 4; \]

1. Evaluate right-hand side.
2. Set value of left-hand side variable to result.

All expressions evaluate to a value, even ones with the assignment operator.

For assignment, the result is the value assigned.
- Usually (but not always) the value of right-hand side
- Type conversion might make assigned value different than computed value

Assignment associates right to left.

\[ y = x = 3; \]
- \( y \) gets the value 3, because \( x = 3 \) evaluates to the value 3.

Arithmetic Operators

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>multiply</td>
<td>( x \times y )</td>
<td>6</td>
<td>l-to-r</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>( x / y )</td>
<td>6</td>
<td>l-to-r</td>
</tr>
<tr>
<td>%</td>
<td>modulo</td>
<td>( x % y )</td>
<td>6</td>
<td>l-to-r</td>
</tr>
<tr>
<td>+</td>
<td>add</td>
<td>( x + y )</td>
<td>7</td>
<td>l-to-r</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
<td>( x - y )</td>
<td>7</td>
<td>l-to-r</td>
</tr>
</tbody>
</table>

- All associate left to right.
- \( * / \% \) have higher precedence than \( + - \).
- Full precedence chart on page 602 of textbook

Arithmetic Expressions

- If mixed types, smaller type is "promoted" to larger.
  \( x + 4.3 \)
  - If \( x \) is int, converted to double and result is double

- Integer division -- fraction is dropped.
  \( x / 3 \)
  - If \( x \) is int and \( x = 5 \), result is 1 (not 1.666666...)

- Modulo -- result is remainder.
  \( x \% 3 \)
  - If \( x \) is int and \( x = 5 \), result is 2.
**Bitwise Operators**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>bitwise NOT</td>
<td></td>
<td>4</td>
<td>r-to-l</td>
</tr>
<tr>
<td>&lt;&lt;=</td>
<td>left shift</td>
<td></td>
<td>8</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&gt;&gt;=</td>
<td>right shift</td>
<td></td>
<td>8</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td></td>
<td>11</td>
<td>l-to-r</td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
<td></td>
<td>12</td>
<td>l-to-r</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise OR</td>
<td></td>
<td>13</td>
</tr>
</tbody>
</table>

- Operate on variables bit-by-bit.
- Like LC-3 AND and NOT instructions.
- Shift operations are logical (not arithmetic).
- Operate on values -- neither operand is changed.

**Logical Operators**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>!</td>
<td>logical NOT</td>
<td></td>
<td>4</td>
<td>r-to-l</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td></td>
<td>14</td>
<td>l-to-r</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Logical OR</td>
<td></td>
</tr>
</tbody>
</table>

- Treats entire variable (or value) as TRUE (non-zero) or FALSE (zero).
- Result of a logical operation is always either TRUE (1) or FALSE (0).

**Relational Operators**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>greater than</td>
<td></td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater or equal</td>
<td></td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td></td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less or equal</td>
<td></td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>==</td>
<td>equals</td>
<td></td>
<td>10</td>
<td>l-to-r</td>
</tr>
<tr>
<td>!=</td>
<td>not equals</td>
<td></td>
<td>10</td>
<td>l-to-r</td>
</tr>
</tbody>
</table>

- Result is 1 (TRUE) or 0 (FALSE).
- **Note:** Don’t confuse equality (==) with assignment (=).

**Special Operators: ++ and --**

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>++</td>
<td>postincrement</td>
<td></td>
<td>2</td>
<td>r-to-l</td>
</tr>
<tr>
<td>--</td>
<td>postdecrement</td>
<td></td>
<td>2</td>
<td>r-to-l</td>
</tr>
<tr>
<td>++</td>
<td>preincrement</td>
<td></td>
<td>3</td>
<td>r-to-l</td>
</tr>
<tr>
<td>--</td>
<td>predecrement</td>
<td></td>
<td>3</td>
<td>r-to-l</td>
</tr>
</tbody>
</table>

- Changes value of variable before (or after) its value is used in an expression.
  - Pre: Increment/decrement variable before using its value.
  - Post: Increment/decrement variable after using its value.
Using ++ and --

\[ x = 4; \]
\[ y = x++; \]
\[ x = 4; \]
\[ y = ++x; \]

Results: \( x = 5, y = 4 \)
(because \( x \) is incremented after assignment)

Results: \( x = 5, y = 5 \)
(because \( x \) is incremented before assignment)

Practice with Precedence

Assume \( a=1, b=2, c=3, d=4 \).

\[ x = a * b + c * d / 2; \quad \text{// same as:} \]
\[ x = (a * b) + ((c * d) / 2); \]

For long or confusing expressions, use parentheses, because reader might not have memorized precedence table.

Note: Assignment operator has lowest precedence, so operations on the right-hand side are evaluated before assignment.

Special Operator: Conditional

\[ x \rightarrow y \quad z \]

If \( x \) is TRUE (non-zero), result is \( y \); else, result is \( z \).

Like a MUX, with \( x \) as the select signal.

Special Operators: +=, *=, etc.

Arithmetic and bitwise operators can be combined with assignment operator.

\[ x += y; \]
\[ x -= y; \]
\[ x *= y; \]
\[ x /= y; \]
\[ x &= y; \]
\[ x |= y; \]
\[ x ^= y; \]
\[ x <<= y; \]
\[ x >>= y; \]

All have same precedence and associativity as \( = \) and associate right-to-left.
Symbol Table

- Like assembler, compiler needs to know information associated with identifiers
  - in assembler, all identifiers were labels and information is address
- Compiler keeps more information
  - Name (identifier)
  - Type
  - Location in memory
  - Scope

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>amount</td>
<td>int</td>
<td>0</td>
<td>main</td>
</tr>
<tr>
<td>hours</td>
<td>int</td>
<td>-3</td>
<td>main</td>
</tr>
<tr>
<td>minutes</td>
<td>int</td>
<td>-4</td>
<td>main</td>
</tr>
<tr>
<td>rate</td>
<td>int</td>
<td>-1</td>
<td>main</td>
</tr>
<tr>
<td>seconds</td>
<td>int</td>
<td>-5</td>
<td>main</td>
</tr>
<tr>
<td>time</td>
<td>int</td>
<td>-2</td>
<td>main</td>
</tr>
</tbody>
</table>

Allocating Space for Variables

- **Global data section**
  - All global variables stored here
  - R4 points to beginning
- **Run-time stack**
  - Used for local variables
  - R6 points to top of stack
  - R5 points to top frame on stack
  - New frame for each block (goes away when block exited)
  - Offset = distance from beginning of storage area

  - Global: `LDR R1, R4, #4`
  - Local: `LDR R2, R5, #3`

Local Variable Storage

- Local variables are stored in an activation record, also known as a stack frame.
- Symbol table “offset” gives the distance from the base of the frame.
  - R5 is the frame pointer – holds address of the base of the current frame.
  - A new frame is pushed on the run-time stack each time a block is entered.
  - Because stack grows downward, base is the highest address of the frame, and variable offsets are <= 0.

Variables and Memory Locations

- In our examples, a variable is always stored in memory.
- When assigning to a variable, must store to memory location.
- A real compiler would perform code optimizations that try to keep variables allocated in registers.
Example: Compiling to LC-3

```c
#include <stdio.h>
int inGlobal;

main()
{
    int inLocal; /* local to main */
    int outLocalA;
    int outLocalB;
    /* initialize */
    inLocal = 5;
    inGlobal = 3;
    /* perform calculations */
    outLocalA = inLocal++ & ~inGlobal;
    outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);
    /* print results */
    printf("The results are: outLocalA = %d, outLocalB = %d\n", outLocalA, outLocalB);
}
```

Example: Symbol Table

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Offset</th>
<th>Scope</th>
</tr>
</thead>
<tbody>
<tr>
<td>inGlobal</td>
<td>int</td>
<td>0</td>
<td>global</td>
</tr>
<tr>
<td>inLocal</td>
<td>int</td>
<td>0</td>
<td>main</td>
</tr>
<tr>
<td>outLocalA</td>
<td>int</td>
<td>-1</td>
<td>main</td>
</tr>
<tr>
<td>outLocalB</td>
<td>int</td>
<td>-2</td>
<td>main</td>
</tr>
</tbody>
</table>