Chapter 12
Variables and Operators

Basic C Elements
Variables
• Named, typed data items

Operators
• Predefined actions performed on data items
• Combined with variables to form expressions, statements

Statements and Functions
• Group together operations

Data Types
C has several basic data types

- `int` integer (at least 16 bits, commonly 32 bits)
- `long` integer (at least 32 bits)
- `float` floating point (at least 32 bits)
- `double` floating point (commonly 64 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 -- 32 bits for most modern processors

Signed vs unsigned:
• Default is 2's complement signed integers
• Use "unsigned" keyword for unsigned numbers

Variable Names
Any combination of letters, numbers, and underscore (_)

Case sensitive
• "sum" is different than "Sum"

Cannot begin with a number
• Usually, variables beginning with underscore
  are used only in special library routines

Only first 31 characters are definitely used
• Implementations can consider more characters if they like
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

• 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 variable is declared outside all blocks

Examples

#include <stdio.h>
int itsGlobal = 0;

main()
{
    int itsLocal = 1;  /* local to main */
    printf("Global %d Local %d\n", itsGlobal, itsLocal);
    
    itsGlobal = 4;  /* change global variable */
    /* local to this block */
    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

Literals

Integer

123 /* decimal */
-123
0x123 /* hexadecimal */

Floating point

6.023
6.023e23 /* 6.023 x 10^{23} */
5E12 /* 5.0 x 10^{12} */

Character

'c'
'\n' /* newline */
'\xA' /* ASCII 10 (0xA) */
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 & z + 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 formed with 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 it do?

(2) Precedence
  • In which order are operators combined?
  • Example:
    "a * b + c + d" is the same as "(a * b) + (c + d)"
    because multiply (*) has a 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/sub 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.
Assignment Operator

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 the 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.
- \( y = (x = 3); \)

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 \div y )</td>
<td>6</td>
<td>l-to-r</td>
</tr>
<tr>
<td>%</td>
<td>modulo</td>
<td>( x \mod y )</td>
<td>6</td>
<td>l-to-r</td>
</tr>
<tr>
<td>+</td>
<td>addition</td>
<td>( x + y )</td>
<td>7</td>
<td>l-to-r</td>
</tr>
<tr>
<td>-</td>
<td>subtraction</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 \(+ -\).

Example
- \( 2 + 3 \times 4 \) versus
- \((2 + 3) \times 4\)

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.
- \( x \div 3 \)
  - if \( x \) is int and \( x=5 \), result is 1 (not 1.666666...)

Modulo -- result is remainder.
- \( x \mod 3 \)
  - 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>~x</td>
<td>4</td>
<td>r-to-l</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>left shift</td>
<td>( x &lt;&lt; y )</td>
<td>8</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>right shift</td>
<td>( x &gt;&gt; y )</td>
<td>8</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise AND</td>
<td>( x &amp; y )</td>
<td>11</td>
<td>l-to-r</td>
</tr>
<tr>
<td>^</td>
<td>bitwise XOR</td>
<td>( x ^ y )</td>
<td>12</td>
<td>l-to-r</td>
</tr>
<tr>
<td></td>
<td></td>
<td>bitwise OR</td>
<td>( x</td>
<td>y )</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.
- \( x = y << 1 \) same as \( x = y + y \)
## 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>!x</td>
<td>4</td>
<td>r-to-l</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical AND</td>
<td>x &amp;&amp; y</td>
<td>14</td>
<td>l-to-r</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>logical OR</td>
<td>x</td>
</tr>
</tbody>
</table>

Treats entire variable (or value) as
- TRUE (non-zero), or
- FALSE (zero).

Result is 1 (TRUE) or 0 (FALSE)
- x = 15; y = 0; printf("%d", x || y);

### Bit-wise vs Logical
- 1 & 8 = 0  (000001 AND 001000 = 000000)
- 1 && 8 = 1 (True & True = True)

## 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>x &gt; y</td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
<td>x &gt;= y</td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
<td>x &lt; y</td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
<td>x &lt;= y</td>
<td>9</td>
<td>l-to-r</td>
</tr>
<tr>
<td>==</td>
<td>equal</td>
<td>x == y</td>
<td>10</td>
<td>l-to-r</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
<td>x != y</td>
<td>10</td>
<td>l-to-r</td>
</tr>
</tbody>
</table>

Result is 1 (TRUE) or 0 (FALSE)

### Assignment vs Equality

Don't confuse equality (==) with assignment (=)

```c
int x = 9;
int y = 10;
if (x == y) {
    printf("not executed\n");
}
if (x = y) {
    printf("%d %d", x, y);
}
```

Result: “10 10” is printed. Why?
Compiler will not stop you! (What happens in Java?)

### Special Operators: ++ and --

Changes value of variable before (or after) its value is used in an expression

<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>x++</td>
<td>2</td>
<td>r-to-l</td>
</tr>
<tr>
<td>--</td>
<td>postdecrement</td>
<td>x--</td>
<td>2</td>
<td>r-to-l</td>
</tr>
<tr>
<td>++</td>
<td>preincrement</td>
<td>+++x</td>
<td>3</td>
<td>r-to-l</td>
</tr>
<tr>
<td>--</td>
<td>predecrement</td>
<td>--x</td>
<td>3</td>
<td>r-to-l</td>
</tr>
</tbody>
</table>

Pre: Increment/decrement variable before using its value
Post: Increment/decrement variable after using its value
Using ++ and --

```c
x = 4;
y = x++; 
```

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

```c
x = 4;
y = ++x; 
```

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

Please, don't combine ++ and =. Really. Just don't!

### Special Operators: +=, *=, etc.

Arithmetic and bitwise operators can be combined with assignment operator.

<table>
<thead>
<tr>
<th>Statement</th>
<th>Equivalent assignment</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>x += y;</code></td>
<td><code>x = x + y;</code></td>
</tr>
<tr>
<td><code>x -= y;</code></td>
<td><code>x = x - y;</code></td>
</tr>
<tr>
<td><code>x *= y;</code></td>
<td><code>x = x * y;</code></td>
</tr>
<tr>
<td><code>x /= y;</code></td>
<td><code>x = x / y;</code></td>
</tr>
<tr>
<td><code>x %= y;</code></td>
<td><code>x = x % y;</code></td>
</tr>
<tr>
<td><code>x &amp;= y;</code></td>
<td><code>x = x &amp; y;</code></td>
</tr>
<tr>
<td>`x</td>
<td>= y;`</td>
</tr>
<tr>
<td><code>x ^= y;</code></td>
<td><code>x = x ^ y;</code></td>
</tr>
<tr>
<td><code>x &lt;&lt;= y;</code></td>
<td><code>x = x &lt;&lt; y;</code></td>
</tr>
<tr>
<td><code>x &gt;&gt;= y;</code></td>
<td><code>x = x &gt;&gt; y;</code></td>
</tr>
</tbody>
</table>

All have same precedence and associativity as = and associate right-to-left.

### Special Operator: Conditional

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Operation</th>
<th>Usage</th>
<th>Precedence</th>
<th>Assoc</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>?:</code></td>
<td>conditional</td>
<td><code>x?y:z</code></td>
<td>16</td>
<td>l-to-r</td>
</tr>
</tbody>
</table>

\(x \, ? \, y \, : \, z\)
- If \(x\) is non-zero, result is \(y\)
- If \(x\) is zero, result is \(z\)

Seems useful, but don’t use it
- A normal "if" is almost always more clear
- You don’t need to use every language feature
- Really, don’t use it (you don’t have to show how clever you are)

### Practice with Precedence

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

```c
x = a * b + c * d / 2; /* x = 8 */
```

same as:

```c
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 all the arithmetic operations on the right-hand side are evaluated first
 Practice with Operators
In preparation for our dis-assembler (HW#8):
int opcode(int ir)
{
    /* code to extract bits 15 through 12 of ir */
}

int get_field(int bits, int hi_bit, int lo_bit)
{
    /* code to extract hi_bit through lo_bit of bits */
}

For example, body of opcode function is now just
• get_field(ir, 15, 12);
What about a “signed-extended” version?

 Practice with Operators (Solution 1)
int opcode(int ir)
{
    ir = ir >> 12;
    ir = ir & 0xf;
    return ir;
}

OR
int opcode(int ir)
{
    ir = ir & 0xf000;
    ir = ir >> 12;
    return ir;
}

 Practice with Operators (Solution 2)
int get_field(int bits, int hi_bit, int lo_bit)
{
    int inv_mask = -0 << (hi_bit+1)
    int mask = ~inv_mask;
    bits = bits & mask; // Mask off high-order bits
    bits = bits >> lo_bit; // Shift away low-order bits
    return bits;
}

OR
int get_field(int bits, int hi_bit, int lo_bit)
{
    bits = ~(~0 << (hi_bit+1)) & bits; // Mask high bits
    bits = bits >> lo_bit; // Shift away low-order bits
    return bits;
}

 Sign Extended Version
int get_sext_field(int bits, int hi_bit, int lo_bit)
{
    int most_significant_bit = bits & (1 << hi_bit);
    if (most_significant_bit != 0) {
        bits = (~0 << hi_bit) | bits; // One extend
    } else { // Zero extend
        bits = (~(~0 << (hi_bit+1)) & bits;
    }
    bits = bits >> lo_bit; // Shift away low-order bits
    return bits;
}