Variables and Operators

Chapter 12
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
Data Types

• C has three basic data types

\begin{itemize}
\item \textbf{int} \hspace{1cm} \text{integer (at least 16 bits)}
\item \textbf{Double} \hspace{1cm} \text{floating point (at least 32 bits)}
\item \textbf{char} \hspace{1cm} \text{character (at least 8 bits)}
\end{itemize}

• 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
Variable Names

• Any combination of letters, numbers, and underscore (_)
  • Case matters
    - "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 used
Examples

Legal

i
wordsPerSecond
words_per_second
_green
aReally_longName_moreThan31chars
aReally_longName_moreThan31characters

Illegal

10sdigit
ten'sdigit
done?
double
Literals

**Integer**

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

**Floating point**

- 6.023
- 6.023e23 /* 6.023 \times 10^{23} */
- 5E12 /* 5.0 \times 10^{12} */
Literals

Character

'c'
'
' /* newline */
'\xA' /* ASCII 10 (0xA) */
Scope: Global and Local

Where is the variable accessible?

- **Global**: accessed anywhere in program
- **Local**: only accessible in a particular region
Scope: Global and Local

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
Structure of a C program

- One `main()` function
- A bunch of other functions

```
/**************************
main()
{
}
/**************************
functionA()
{
}
/**************************
functionB()
{
}
/**************************
```
#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** which denote the work to be done by the program.
Operators

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 & z + 3 || 9 - w-- % 6
Statement

• Expresses a complete unit of work
  - executed in sequential order

• Simple statement ends with semicolon

```plaintext
z = x * y; /* assign product to z */
y = y + 1; /* after multiplication */
; /* null statement */
```
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 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 (+)
Operators

• (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 Operator

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 * 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>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 + -.
Examples

• Do examples with the compiler
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...)
Arithmetic Expressions

Modulo -- result is remainder.

\[ x \% 3 \]

if \( x \) is int and \( x=5 \), result is 2.
Bitwise Operators

- 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.

<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>
Examples

• Do examples with the compiler
Logical Operators

- Treats entire variable (or value) as TRUE (non-zero) or FALSE (zero).
- Result is 1 (TRUE) or 0 (FALSE).

<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>Logical NOT</td>
<td>!X</td>
<td>4</td>
<td>R-to-L</td>
</tr>
<tr>
<td><code>&amp;&amp;</code></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>
Examples

• Do examples with the compiler
Relational Operators

- Result is 1 (TRUE) or 0 (FALSE).

- Note: Don't confuse equality (==) with assignment (=).
Examples

• Do examples with the compiler
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>Post increment</td>
<td>X++</td>
<td>2</td>
<td>R-to-L</td>
</tr>
<tr>
<td>--</td>
<td>Post decrement</td>
<td>X--</td>
<td>2</td>
<td>R-to-L</td>
</tr>
<tr>
<td>++</td>
<td>Pre increment</td>
<td>++X</td>
<td>3</td>
<td>R-to-L</td>
</tr>
<tr>
<td>--</td>
<td>Pre decrement</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 --

\[
x = 4;
\]
\[
y = x++;
\]
• Results: \( x = 5, y = 4 \)
  (because \( x \) is incremented after assignment)

\[
x = 4;
\]
\[
y = ++x;
\]
• Results: \( x = 5, y = 5 \)
  (because \( x \) is incremented before assignment)
Practice with Precedence

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

\[
x = a \times b + c \times d / 2; \quad /* x = 8 */
\]

same as:

\[
x = (a \times b) + ((c \times d) / 2);
\]

• For long or confusing expressions, use parentheses, because reader (or you!) 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.
### 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

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

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

<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>Conditional</td>
<td>( X \ ? \ Y : Z )</td>
<td>16</td>
<td>L-to-R</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccc}
Y & \text{downward} & Z & \text{downward} \\
1 & \text{upward} & 0 & \text{upward} \\
\end{array}
\]
Special Operator: Conditional

```c
main ()
{
    int x, y, z;
    int result;

    x = 0;
    y = 5;
    z = 10;

    result = x ? y : z;
    printf ("%d\n", result);
}
```
Variable allocation

• **Global** variables – allocated on the HEAP
• **Local** variables – allocated in the activation record
  - For each block
  - For each function
Memory map

- **Global data section**
  - All global variables stored here (actually all static variables)
  - 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`
Sample program

Network rate calculation program

```c
#include <stdio.h>

int main()
{
    int amount;    /* The number of bytes to be transferred */
    int rate;      /* The average network transfer rate */
    int time;      /* The time, in seconds, for the transfer */
    int hours;     /* The number of hours for the transfer */
    int minutes;   /* The number of mins for the transfer */
    int seconds;   /* The number of secs for the transfer */

    /* Get input: number of bytes and network transfer rate */
    printf("How many bytes of data to be transferred? ");
    scanf("%d", &amount);

    printf("What is the transfer rate (in bytes/sec)? ");
    scanf("%d", &rate);

    /* Calculate total time in seconds */
    time = amount / rate;

    /* Convert time into hours, minutes, seconds */
    hours = time / 3600;  /* 3600 seconds in an hour */
    minutes = (time % 3600) / 60; /* 60 seconds in a minute */
    seconds = ((time % 3600) % 60); /* remainder is seconds */

    /* Output results */
    printf("Time: %dh %dm %ds\n", hours, minutes, seconds);
}
```

Figure 12.4  A C program that performs a simple network rate calculation
Local variable storage

• Local variables are stored in an *activation record*, also known as a *stack frame*.

• Symbol table “offsets” give 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 \( \leq 0 \).
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 (offset wrt frame ptr)
  - 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>
register variables

• 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.

• Programmer can try to suggest to the compiler which variables, using the modifier `register`
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>
Example: Code generation (1/3)

; main
; initialize variables
    AND R0, R0, #0
    ADD R0, R0, #5 ; inLocal = 5
    STR R0, R5, #0 ; (offset = 0)

    AND R0, R0, #0
    ADD R0, R0, #3 ; inGlobal = 3
    STR R0, R4, #0 ; (offset = 0)
Example: Code generation (2/3)

; first statement:
; outLocalA = inLocal++ & ~inGlobal;
LDR R0, R5, #0 ; get inLocal
ADD R1, R0, #1 ; increment
STR R1, R5, #0 ; store

LDR R1, R4, #0 ; get inGlobal
NOT R1, R1 ; ~inGlobal
AND R2, R0, R1 ; inLocal & ~inGlobal
STR R2, R5, #-1 ; store in outLocalA
; (offset = -1)
Example: Code generation (3/3)

; next statement:
; outLocalB = (inLocal + inGlobal)
; outLocalB = (inLocal + inGlobal) - (inLocal - inGlobal);

LDR R0, R5, #0 ; inLocal
LDR R1, R4, #0 ; inGlobal
ADD R0, R0, R1 ; R0 is sum
LDR R2, R5, #0 ; inLocal
LDR R3, R4, #0 ; inGlobal
NOT R3, R3
ADD R3, R3, #1
ADD R2, R2, R3 ; R2 is difference
NOT R2, R2 ; negate
ADD R2, R2, #1
ADD R0, R0, R2 ; R0 = R0 - R2
STR R0, R5, #-2 ; outLocalB (offset = -2)