Chapter 12
Variables and Operators
Highlights (1)

\[ \text{area} = \pi \times r \times \text{width} + \text{width} \times \text{height} \]
Highlights (2)

area = 3.14 * r * r + width * height

• Precedence: 우선 순위
  area = 3.14 * r * r + (width * height)

• Associativity: 결합 법칙
  area = ((3.14 * r) * r) + width * height

• also need to distinguish value (from evaluating an expression on the right-hand side) from action (assignment of a value to a variable on the left-hand side)
Basic C Elements

Variable
- holds a value upon which a program acts
- has a type (integer, floating-point, character, etc)
- is an abstraction (i.e., symbolic name) of a memory location but its value is moved to a register when subject to an operation and moved back to memory when registers are full

Operators
- predefined actions performed on data values
- combined with variables and constants (literals) to form expressions
- is an abstraction of arithmetic / logic 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 ISA

- `int` is supposed to be "natural" integer size; for LC-3, that's 16 bits -- 32 bits or 64 bits for most modern processors
Variable Declaration

Syntax

- `int <variable name list>;`
- `double <variable name list>;`
- `char <variable name list>;`

Examples

- `int score;`
- `int score, sum;`
- `double area, volumn;`
- `char InputChar;`
- `int sum = 0;`
Variable Name (i.e., \textit{identifier})

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 meaningful
Examples

Legal

i
wordsPerSecond
words_per_second
_green
aReally_longName_moreThan31chars
aReally_longName_moreThan31characters

Illegal

10sdigit
ten'sdigit
done?
double

reserved keyword

same identifier
**Literals (i.e., constants)**

**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 (more on Page 567 */
'\xA'  /* ASCII 10 - hexadecimal */
'\012' /* ASCII 10 - octal */
```
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
Example

```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 values (ones stored in variables, literals/constants and, return values from functions) using the *operators* provided by the high-level language.

Variables, constants, and operators combine to form an *expression*, which yields a *value*.

Each operator may correspond to many machine instructions.

- Example: The multiply operator (*) typically requires multiple LC-3 ADD instructions.
Expression

Any combination of variables, literals/consts, operators, and function calls (i.e., their return values) that yield a value

- every expression has a type (integer, floating-point, character, boolean) derived from the types of its components (according to rules defined in the C programming language)

Examples:

- \(3.14\)
- \(\text{width}\)
- \(\text{width} \times \text{height}\)
- \(3.14 \times r \times r + \text{width} \times \text{height}\)
- \(\text{counter} \geq \text{STOP}\)
- \(x + \sqrt{y}\)
- \(x \& z + 3 \mid\mid 9 - w - w - w \mod 6\)
Statement

Expresses a complete unit of work
  • executed in sequential order

Simple statement ends with semicolon

z = x * y;
y = y + 1;
;  /* null statement */

Compound statement groups simple statements using braces (why needed?).
  • 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 Statement (action)
Changes the value of a variable.

\[
x = x + 4;
\]

1. Evaluate the expression on the right-hand side.
2. Set value of the left-hand side variable to result.
**Assignment Operator** (value) – unique in C

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

For assignment, the result is the value assigned.

- the value of the right-hand side
- the value of expression \( y = x + 3 \) is the current value of variable \( x \) plus 3

Assignment associates right to left.

\[ y = x = 3; \text{ is equivalent to } y = (x = 3); \]

\( y \) 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>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 +  -.

Operate on values -- neither operand is changed.
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 1.6666666...)

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>(~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.
# 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): boolean type

Note: Don't confuse equality (==) with assignment (=).
### 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 the operands as boolean type: TRUE (non-zero) or FALSE (zero).

Result is 1 (TRUE) or 0 (FALSE): boolean type
Special Operators: ++ and -- (unique in C)

Changes value of variable after (or before) 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 `--`

```plaintext
x = 4;
y = x++;  
Results: x = 5, y = 4  
(because x is incremented after using its value)
```

```plaintext
x = 4;  
y = ++x;  
Results: x = 5, y = 5  
(because x is incremented before using its value)
```
### Special Operators: +=, *=, etc. (unique in C)

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>?:</td>
<td>conditional</td>
<td>x?y:z</td>
<td>16</td>
<td>l-to-r</td>
</tr>
</tbody>
</table>

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

Like a MUX, with $x$ as the select signal.

![MUX Diagram](attachment:MUX.png)
Practice with Precedence

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

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

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

For long or confusing expressions, use parentheses, because reader might not have memorized precedence table.
## Summary: Operator Precedence and Associativity

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Associativity</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (highest)</td>
<td>l(eft) to r(ight)</td>
<td>( ) (function call) [ ] (array index) . - &gt;</td>
</tr>
<tr>
<td>2</td>
<td>r(ight) to l(eft)</td>
<td>++ - - (postfix versions)</td>
</tr>
<tr>
<td>3</td>
<td>r to l</td>
<td>++ - - (prefix versions)</td>
</tr>
<tr>
<td>4</td>
<td>r to l</td>
<td>* (indirection) &amp; (address of)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>+ (unary) - (unary) ~ (logical NOT) ! (bitwise NOT) sizeof</td>
</tr>
<tr>
<td>5</td>
<td>r to l</td>
<td>(type) (type cast)</td>
</tr>
<tr>
<td>6</td>
<td>l to r</td>
<td>* (multiplication) / %</td>
</tr>
<tr>
<td>7</td>
<td>l to r</td>
<td>+ (addition) - (subtraction)</td>
</tr>
<tr>
<td>8</td>
<td>l to r</td>
<td>&lt;&lt; &gt;&gt; (shifts)</td>
</tr>
<tr>
<td>9</td>
<td>l to r</td>
<td>&lt; &gt; &lt;= &gt;= (relational operators)</td>
</tr>
<tr>
<td>10</td>
<td>l to r</td>
<td>== != (equality/inequality operators)</td>
</tr>
<tr>
<td>11</td>
<td>l to r</td>
<td>&amp; (bitwise AND)</td>
</tr>
<tr>
<td>12</td>
<td>l to r</td>
<td>^ (bitwise exclusive OR)</td>
</tr>
<tr>
<td>13</td>
<td>l to r</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>l to r</td>
<td>&amp;&amp; (logical AND)</td>
</tr>
<tr>
<td>15</td>
<td>l to r</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>l to r</td>
<td>?: (conditional expression)</td>
</tr>
<tr>
<td>17 (lowest)</td>
<td>r to l</td>
<td>= += -= *= etc. (assignment operators)</td>
</tr>
</tbody>
</table>
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

<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>
Local Variable Storage

Local variables are stored in user stack called 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 function is called or a block is entered.
- Because stack grows downward, **R5** contains the highest address of the frame, and variable offsets are $\leq 0$. 

```c
#include <stdio.h>

int main()
{
    int amount;
    int rate;
    int time;
    int hours;
    int minutes;
    int seconds;
    ...
```

<table>
<thead>
<tr>
<th>seconds</th>
<th>minutes</th>
<th>hours</th>
<th>time</th>
<th>rate</th>
<th>amount</th>
</tr>
</thead>
</table>

R5
Allocating Space for Variables

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`
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. Why?
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 (R4)</td>
<td>global</td>
</tr>
<tr>
<td>inLocal</td>
<td>int</td>
<td>0 (R5)</td>
<td>local:main</td>
</tr>
<tr>
<td>outLocalA</td>
<td>int</td>
<td>-1 (R5)</td>
<td>local:main</td>
</tr>
<tr>
<td>outLocalB</td>
<td>int</td>
<td>-2 (R5)</td>
<td>local:main</td>
</tr>
</tbody>
</table>

![Symbol Table Diagram]
Example: Code Generation

; 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 (continued)

; first statement:

; outLocalA = inLocal & ~inGlobal;
LDR R0, R5, #0 ; get inLocal
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 (continued)

; 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 contains (inLocal + inGlobal)
LDR R2, R5, #0 ; inLocal
LDR R3, R4, #0 ; inGlobal
NOT R3, R3
ADD R3, R3, #1
ADD R2, R2, R3 ; R2 contains (inLocal - inGlobal)
NOT R2, R2 ; negate
ADD R2, R2, #1
ADD R0, R0, R2 ; (inLocal + inGlobal) - (inLocal - inGlobal)
STR R0, R5, #-2 ; outLocalB (offset = -2)
꼭 기억해야 할 것

- variable
  - type
  - scope
- operator
  - precedence
  - associativity
- literal/constant
- expression
  - yields a value after being evaluated
  - has a type
- statement