Chapter 4. Flow of Control

Byoung-Tak Zhang
TA: Hanock Kwak

Biointelligence Laboratory
School of Computer Science and Engineering
Seoul National University

http://bi.snu.ac.kr
Flow of Control

- **Sequential flow of control**
  - Statement in a program are normally executed one after another.

- Often it is desirable to alter the sequential flow of control to provide for
  - a choice of action
    - *if, if-else, switch*
  - or a repetition of action
    - *while, for, do-while*
True and False

- True
  - Any non-zero integers are treated as true.

- False
  - Zero means false.

- Logical and Relational Operations
  - Results of the logical and relational operations are one(true) or zero(false)

```
printf("%d", 2 == 2); // equal, 1
printf("%d", !0); // negation, 1
```
Relational Operator

- $expr < expr$
- $expr > expr$
- $expr <= expr$
- $expr >= expr$

- $a < b$
  - If $a$ is less than $b$, then 1 (true).
  - If $a$ is not less than $b$, then 0 (false).

<Examples>

- $a < 3$
- $a > b$
- $-1.3 >= (2.0 * x + 3.3)$
## Declarations and Initializations

```plaintext
char   c = 'w';
int    i = 1, j = 2, k = -7;
double x = 7e+33, y = 0.001
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a' + 1 &lt; c</td>
<td>('a' + 1) &lt; c</td>
<td>1</td>
</tr>
<tr>
<td>- i - 5 * j &gt;= k + 1</td>
<td>(((- i) - (5 * j)) &gt;= (k +1)</td>
<td>0</td>
</tr>
<tr>
<td>3 &lt; j &lt; 5</td>
<td>(3 &lt; j) &lt; 5</td>
<td>1</td>
</tr>
<tr>
<td>x - 3.333 &lt;= x + y</td>
<td>(x - 3.333) &lt;= (x + y)</td>
<td>1</td>
</tr>
<tr>
<td>x &lt; x + y</td>
<td>x &lt; (x + y)</td>
<td>0</td>
</tr>
</tbody>
</table>
Relational Operator

- \( \text{expr} == \text{expr} \)
- \( \text{expr} != \text{expr} \)
- \( a != b \)
  - If \( a \) is not equal to \( b \), then 1 (true).
  - If \( a \) is equal to \( b \), then 0 (false).

<Examples>

- \( c == 'A' \)
- \( k != -2 \)
- \( 3 + z == x + y/4 \)
Relational Operator

!! A common programming error

if (a = 1)
...
if (a == 1)
...

<table>
<thead>
<tr>
<th>Declarations and Initializations</th>
</tr>
</thead>
<tbody>
<tr>
<td>int i = 1, j=2, k=3;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i == j</td>
<td>j == i</td>
<td>0</td>
</tr>
<tr>
<td>i != j</td>
<td>j != i</td>
<td>1</td>
</tr>
<tr>
<td>i + j + k == - 2 * - k</td>
<td>((i + j) + k) == ((- 2) * (- k))</td>
<td>1</td>
</tr>
</tbody>
</table>
Negation

- `! expr`
  - **negation**
  - If `expr` has value zero, `! expr` is 1 (true).
  - If `expr` has non-zero value, `! expr` is 0 (false).

<Examples>

- `! a`
- `! (x + 7.7)`
- `! (a < b || c < d)`
### Declarations and Initializations

```c
char c = 'A';
int i = 7, j = 7;
double x = 0.0, y = 2.3;
```

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>! c</td>
<td>! c</td>
<td>0</td>
</tr>
<tr>
<td>! (i - j)</td>
<td>! (i - j)</td>
<td>1</td>
</tr>
<tr>
<td>! i - j</td>
<td>(! i) - j</td>
<td>-7</td>
</tr>
<tr>
<td>! ! (x + y)</td>
<td>! (! (x + y))</td>
<td>1</td>
</tr>
<tr>
<td>! x * ! ! y</td>
<td>(! x) * (!(! y))</td>
<td>1</td>
</tr>
</tbody>
</table>
Logical Operator

- `expr || expr` (logical or)
- `expr && expr` (logical and)

- `a || b`
  - If `a` or `b` is 1, then 1 (true).
  - If `a` and `b` are both 0, then 0 (false).

- `a && b`
  - If `a` and `b` are both 1, then 1 (true).
  - If `a` or `b` is 0, then 0 (false)

Examples:

```
a && b
a || b
10 < a && a < 100
```
Logical Operator

- **&&** has higher precedence than **||**.
- Both of **&&** and **||** are of lower precedence than all unary, arithmetic, equality, and relational operators.

**Short-circuit Evaluation**
- In evaluating the expr.s that are the operands of **&&** and **||**, the evaluation process stops as soon as the outcome true or false is known.
  - **expr1 && expr2**, if expr1 has value zero, then expr2 is not evaluated
  - **expr1 || expr2**, if expr1 has nonzero value, then expr2 is not evaluated
# Logical Operator

<table>
<thead>
<tr>
<th>Declarations and Initializations</th>
</tr>
</thead>
<tbody>
<tr>
<td>char c = 'B';</td>
</tr>
<tr>
<td>int i = 3, j = 3, k = 3;</td>
</tr>
<tr>
<td>double x = 0.0, y = 2.3;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>i &amp;&amp; j &amp;&amp; k</td>
<td>(i &amp;&amp; j) &amp;&amp; k</td>
<td>1</td>
</tr>
<tr>
<td>x</td>
<td></td>
<td>i &amp;&amp; j - 3</td>
</tr>
<tr>
<td>i &lt; j &amp;&amp; x &lt; y</td>
<td>(i &lt; j) &amp;&amp; (x &lt; y)</td>
<td>0</td>
</tr>
<tr>
<td>i &lt; j</td>
<td></td>
<td>x &lt; y</td>
</tr>
<tr>
<td>A' &lt;= c &amp;&amp; c &lt;= 'Z'</td>
<td>('A' &lt;= c) &amp;&amp; (c &lt;= 'Z')</td>
<td>1</td>
</tr>
<tr>
<td>c - 1 == 'A'</td>
<td></td>
<td>c + 1 == 'Z'</td>
</tr>
</tbody>
</table>
Statement

- Statement
  - A statement is the smallest standalone element of a C source code.

- Expression Statement
  - An expression followed by `;`

- Empty statement
  - Written as a single semicolon
  - Useful where a statement is needed syntactically

```
a = b;          /* assignment statement */
a + b + c;      /* legal, but no useful work gets done */
;               /* empty statement */
printf("%d\n", a);  /* a function call */
```
Statement

- **Compound Statement**
  - a series of declarations and statements surrounded by braces
    - block
  - for grouping statements into an executable unit
  - It is itself a statement, thus it can be placed wherever a statement is placed.

```c
{ 
    a = 1; 
    { /* nested */ 
        b = 2; 
        c = 3; 
    } 
}
```
if and if-else Statement

if ( expr )
statement

- If expr is nonzero, then statement is executed; otherwise, statement is skipped and control passes to the next statement.

```c
if (j < k) {
    min = j;
    printf("j is smaller than k\n");
}
```
if ( `expr`) 
  statement1
else
  statement2

- If `expr` is nonzero, then `statement1` is executed and then skip `statement2`; otherwise, `statement1` is skipped and then `statement2` is executed.
if and if-else Statement

if (a == 1)
    if (b == 2) /* if statement is itself a statement */
        printf("***\n");

else
    printf("###\n");

dangling else problem (An else attaches to the nearest if.)

if (a == 1)
    if (b == 2)
        printf("***\n");
else
    printf("###\n");

if (a == 1)
    if (b == 2)
        printf("***\n");
else
    printf("###\n");
if and if-else Statement

- **Complex Example**

```java
if (c == ' ')
    ++blank_cnt;
else if (c >= '0' && c <= '9' )
    ++digit_cnt;
else if (c >= 'a' && c <= 'z' || c >= 'a' && c <= 'z' )
    ++letter_cnt;
else if (c == '\n')
    ++nl_cnt;
else
    ++other_cnt;
```
while Statement

```
while ( expr )
  statement
```

- First `expr` is evaluated. If it is nonzero, then `statement` is executed and control is passed back to `expr`. This repetition continues until `expr` is zero.

```c
// print i and increment it until it is less than 10
while ( i < 10 )
  printf("%d", i++);
```
for Statement

```
for ( expr1 ; expr2 ; expr3 )
    statement
```

```
expr1 ;
while ( expr2 )
{
    statement
    expr3;
}
```

- First, `expr1` (initialization) is evaluated.
- `expr2` is evaluated. If it is nonzero, then `statement` is executed, `expr3` is evaluated, and control is passed back to `expr2`.
  - `expr2` is a logical expression controlling the iteration.
  - This process continues until `expr2` is zero.
// print seven factorial value
int i, factorial = 1;

for ( i = 2; i <= 6; i++ )
    factorial = factorial * i;

printf("7! is %d", factorial);
// print all cases of rolling two dices
int i, j;
for (i = 1; i <= 6; i++)
{
    for (j = 1; j <= 6; j++)
        printf("(%d, %d)\n", i, j);
    printf("\n");
}
Comma Operator

`expr1 , expr2`

- `expr1` is evaluated, and then `expr2`.

```c
for (sum = 0, i =1; i <= 10; ++i)
    sum += i;
```
do-while Statement

do
    statement
while ( expr )

- First **statement** is executed and **expr** is evaluated. If the value of **expr** is nonzero, then control is passed back to **statement**. When **expr** is zero, it finishes the control.
do-while Statement

// try to get positive integer
do {
    printf("Input a positive integer: ");
    scanf("%d", &n);
    if (n < 0) {
        printf("\nERROR: Do it again!\n\n");
    }
} while (n < 0);
break

break;

- causes an exit from the inner most enclosing loop or **switch** statement

```c
while (1) {
    scanf("%lf", &x);
    if (x < 0.0)
        break;  /* exit loop if x is negative */
    printf("%f\n", sqrt(x));
}
/* break jumps to here */
```
continue

continue;

- causes the current iteration of a loop to stop and causes the next iteration of the loop to begin immediately

```c
/* print all even integer less than 100 */
for (i = 0; i < 100; i++)
{
    if (i % 2 == 1) // if i is odd
        continue;    // go to next iteration
    printf("%d\n", i);
}
```
switch Statement

- A multiway conditional statement generalizing the if-else statement

```c
switch (c) {
    case 'a':
        printf("a\n");
        break;
    case 'b':
    case 'B':
        printf("b or B\n");
        break;
    default:
        printf("something else");
}
```

1. Evaluate the switch expression.
2. Go to the case label having a constant value that matches the value of the expression in (1), or, if a match is not found, go to the default label, or, if there is no default label, terminate the switch.
3. Terminate the switch when a break statement is encountered, or terminate the switch by “falling off the end”.
Conditional Operator

`expr1 ? expr2 : expr3;`

- `expr1` is evaluated.
  - If `expr1` is nonzero(true), then `expr2` is evaluated, and that is the value of the conditional expression as a whole.
  - If `expr1` is zero(false), then `expr3` is evaluated, and that is the value of the conditional expression as a whole.

```c
// x is 100 if y > 10, otherwise -5.
x = (y > 10) ? 100 : -5;
```