Flow of Control

Fall 2014

Jinkyu Jeong
(jinkyu@skku.edu)
Flow of Control

- C is a sequential language
  - Statements in a program are executed one after another

- To change flow of control, use
  - Choice instructions: if, switch
  - Iterative instructions: while, for
  - OR recursion
  - You may need operators
## Operators for them

<table>
<thead>
<tr>
<th>Type of Operators</th>
<th>Informal Description</th>
<th>Operator Mnemonic</th>
</tr>
</thead>
</table>
| Relational Operators   | Less than
Greater than
Less than or equal to
Greater than or equal to | <
>
<=
>= |
| Equality Operators     | Equal to
Not equal to                                         | ==
!= |
| Logical Operators      | (unary) negation
Logical and
Logical or             | !
&&
|| |
# Operators

<table>
<thead>
<tr>
<th>Operators</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>() ++ (postfix) -- (postfix)</td>
<td>left to right</td>
</tr>
<tr>
<td>+ (unary) - (unary) ++ (prefix) -- (prefix)</td>
<td>right to left</td>
</tr>
<tr>
<td>* / %</td>
<td>left to right</td>
</tr>
<tr>
<td>+ -</td>
<td>left to right</td>
</tr>
<tr>
<td>&lt; &lt;= &gt; &gt;=</td>
<td>left to right</td>
</tr>
<tr>
<td>== !=</td>
<td>left to right</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>left to right</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>right to left</td>
</tr>
<tr>
<td>= += -= *= /= etc.</td>
<td>right to left</td>
</tr>
<tr>
<td>, (comma operator)</td>
<td>left to right</td>
</tr>
</tbody>
</table>
True/False Representation in C

**True**
- int value 1
- a = 2 > 1; /* a == 1 */

**False**
- int value 0
- a = 1 < 2; /* a == 0 */

**Zero values**
- 0, 0.0, '\0', NULL are deemed false
- if(0) {}, while(0) {} 

**Non-zero values**
- 1, 1.0, 2, 3, ... are deemed true
- if(1) {}, while(2) {}
Relational Operators

- >, <, >=, >=

- Common mistakes
  - a => b, a =< b /* out of order */
  - a <= b /* space not allowed */
  - a >> b /* different operator */
  - a < b < c /* equals 1 < c or 0 < c */
## Relational Operators Examples

### Declarations and initializations

<table>
<thead>
<tr>
<th>Type</th>
<th>Declaration</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>c = 'w'</td>
</tr>
<tr>
<td>int</td>
<td>i = 1, j = 2, k = -7</td>
</tr>
<tr>
<td>double</td>
<td>x = 7e+33, y = 0.001</td>
</tr>
</tbody>
</table>

### Expressions and Equivalent Expressions

<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>
### Equality Operators

- **Equality expression ::=**
  
  expression == expression | expression != expression

- **Examples**
  
  c == 'A'
  
  k != 2
  
  x + y == 3 * z - 7

- **Common mistakes**
  
  a = b instead of a == b
  
  a = = b
  
  a =! b
### Equality Operators Examples

**Declarations and initializations**

```cpp
int i = 1, j = 2, k = 3;
```

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

- **Logical expressions**
  - Negative:  \(! expr\)
  - Or:  \( expr \lor expr \)
  - And:  \( expr \land expr \)

- **Examples**
  - \(!a\)
  - \(! (x + 7.3)\)
  - \(! (a < b \lor c < d)\)
  - \(a \land b\)
  - \(a \lor b\)

- **Common mistakes**
  - \(!a\)
  - \(a\land\)
  - \(a \& b\)
  - \& b  --- this is serious
Logical Operators Examples

<table>
<thead>
<tr>
<th>Declarations and initializations</th>
</tr>
</thead>
<tbody>
<tr>
<td>char    c = ‘A’;</td>
</tr>
<tr>
<td>int     i = 7, j = 7;</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>! 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 Operators Examples

### Declarations and initializations

```
char    c = 'B';
int     i = 3, j = 3, k = 3;
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><code>i &amp;&amp; j &amp;&amp; k</code></td>
<td><code>(i &amp;&amp; j) &amp;&amp; k</code></td>
<td>1</td>
</tr>
<tr>
<td>`x</td>
<td></td>
<td>i &amp;&amp; j - 3`</td>
</tr>
<tr>
<td><code>i &lt; j &amp;&amp; x &lt; y</code></td>
<td><code>(i &lt; j) &amp;&amp; (x &lt; y)</code></td>
<td>0</td>
</tr>
<tr>
<td>`i &lt; j</td>
<td></td>
<td>x &lt; y`</td>
</tr>
<tr>
<td><code>'A' &lt;= c &amp;&amp; c &lt;= 'Z'</code></td>
<td><code>('A' &lt;= c) &amp;&amp; (c &lt;= 'Z')</code></td>
<td>1</td>
</tr>
<tr>
<td>`c-1 == 'A'</td>
<td></td>
<td>c+1 == 'Z'`</td>
</tr>
</tbody>
</table>
Short-circuit

- The evaluation stops as soon as the outcome is known
- `expr1 && expr2`
  - If `expr1` is evaluated to be `false`, `expr2` needs not be evaluated
- `expr1 || expr2`
  - If `expr1` is evaluated to be `true`, `expr2` needs not be evaluated
**Compound Statement**

- A compound statement is a series of declarations and statements surrounded by braces `{ }`

```c
{
    int a, b, c;
    a += b += c;
    printf("a = %d, b = %d, c = %d\n", a, b, c);
}
```

- A compound is usually called “block”
- **Expression statements**
  
  ```c
  a + b + c;
  ; /* empty statement */
  ```
If Statement (1)

- `if (expr) (then) statement | block`
- `if (expr) (then) statement | block else statement | block`

```c
if ( y != 0.0 )
    x /= y;
if ( c == ' ' ) {
    ++blank_cnt;
    printf("found another blank\n");
}
if b == a // parentheses missing
    area = a * a;
```

- Statement can be an empty one
- Same for else statement
if Statement (2)

```c
if ( c >= 'a' && c <= 'z')
    ++lc_cnt;
else {
    ++other_cnt;
    printf("%c is not a lowercase letter\n", c);
}

if ( c >= 'a' && c <= 'z')
    ++lc_cnt;
else if ( c >= 'A' && c <= 'Z')
    ++uc_cnt;
else {
    ++other_cnt;
    printf("%c is not a letter\n", c);
}

if ( i != j ) {
    i += 1;
    j += 2;
};
else
    i -= j; // syntax error
```
if Statement (3)

- else attaches to the nearest if

```c
if ( a == 1 )
    if ( b == 2 )
        printf("***\n");

if ( a == 1 )
    if ( b == 2 )
        printf("***\n");
    else
        printf("###\n");
/* equals */
if ( a == 1 )
    if ( b == 2 )
        printf("***\n");
    else
        printf("###\n");
```
Iterative Statements

- **while, for, and do statements**
  - Provide iterative action

- **goto, break, continue, and return statements cause an unconditional transfer**
  - SE people hate these (except return)
while Statement

- **while (expr) statement | block**
  - Repeat the statement or block while expr is true
  - Statement can be empty

```c
while ( i++ < n )
    factorial *= i;

while ( (c = getchar()) == ' ' )
    ; // skip blank characters in the input stream

while ( ++i < LIMIT ) do {
    // syntax error: do is not allowed
    j = 2 * i + 3;
    printf("%d\n", j);
}
```
#include <stdio.h>

int main(void) {
    int blank_cnt = 0, c, digit_cnt = 0,
    letter_cnt = 0, nl_cnt = 0, other_cnt = 0;

    while ((c = getchar()) != EOF)          /* braces not necessary */
        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 == '
')
            ++nl_cnt;
        else
            ++other_cnt;

    printf("%10s%10s%10s%10s%10s%10s\n\n", "blanks", "digits", "letters", "lines", "others", "total");
    printf("%10d%10d%10d%10d%10d%10d\n\n", blank_cnt, digit_cnt, letter_cnt, nl_cnt, other_cnt,
        blank_cnt + digit_cnt + letter_cnt + nl_cnt + other_cnt);
    return 0;
}
for statements

- for ( expr1; expr2; expr3 ) statement | block
  1. evaluate expr1 first
  2. if expr2 is true, execute statement or block
  3. evaluate expr3
  4. continues step 2

for (i = 1; i <= n; ++i)  
  sum += i;

for (i = 1; i <= n; ++i) {
  sum += i;
  sqr += i * i;
}

- Any expr can be missing, but two semicolons must remain

for (;;)  /* infinite loop */
  sum += 1;
Comma operator

- **expr, expr**
  - Have the value and type of its right operand

```plaintext
for (sum = 0, i = 1; i <= n; sum += i, ++i)
```

### Declarations and initializations

<table>
<thead>
<tr>
<th>int</th>
<th>i, j, k = 3;</th>
</tr>
</thead>
<tbody>
<tr>
<td>double</td>
<td>x = 3.3;</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i = 1, j = 2, ++ k + 1</code></td>
<td><code>((i=1, (j=2)), ((++k)+1)</code></td>
<td>5</td>
</tr>
<tr>
<td><code>k != 1, ++x * 2.0 + 1</code></td>
<td><code>((k != 1), (((+ x) * 2.0) + 1)</code></td>
<td>9.6</td>
</tr>
</tbody>
</table>
do statement

A variant of while statement

- do statement | block while (expr);
- do { statements } while (expr);
- The block is executed first, and then the expr is evaluated
- You should be able to convert do statement to while statement, and vice versa

```c
i = 0;
sum = 0;
/* sum a series of integer inputs until 0 is input */
do {
    sum += i;
    scanf("%d", &i);
} while ( i > 0 );
```
Control Expression Tip!

- Use a relational expression rather than an equality expression

/* A test that fails. */

#include <stdio.h>
int main(void)
{
    int cnt = 0;
    double sum = 0.0, x;

    for ( x = 0.0; x != 9.9; x += 0.1) {
        /* trouble ! */
        sum += x;
        printf("cnt = %5d\n", ++cnt);
    }
    printf("sum = %f\n", sum);
    return 0;
}
### goto statement

- **Jump to a label**

  ```
goto label;
  ...
  label: /* label is an identifier */
  ```

- **It is considered harmful, but .....**

  ```
goto error;
  ...
  error: {
    printf("An error has occurred -bye!\n");
    exit(1);
  }

while ( scanf("%lf", &x) == 1 ) {
  if ( x < 0.0 )
    goto negative_alert;
    printf("%f %f\n", sqrt(x), sqrt(2*x));
}

negative_alert: printf("Negative value encountered!\n");```
### An exit from a loop

```c
while ( 1 ) {
    scanf("%lf", &x);
    if ( x < 0.0 )
        break;
    /* no square root if number is negative, exit loop */
    printf("%f
", sqrt(x));
}
/* break jumps to here */
```

```c
for ( ; ; ) {
    scanf("%lf", &x);
    if ( x < 0.0 )
        break;
    /* no square root if number is negative, exit loop */
    printf("%f
", sqrt(x));
}
/* break jumps to here */
```
continue statement

- Stop the current iteration and goto the next iteration

```c
for ( i = 0; i < TOTAL; ++i ) {
    c = getchar();
    if ( c >= '0' && c <= '9' )
        continue;
    /* processing other characters */
    /* continue transfers control to here to begin next iteration */
}

while ( i < TOTAL ) {
    ++i;
    c = getchar();
    if ( c >= '0' && c <= '9' )
        continue;
    /* processing other characters */
    /* continue transfers control to here to begin next iteration */
}
switch statement

- switch (expr1)  /* must be integral */
  - Goto the matched case label

```c
switch (c) {
    case 'a':
        ++a_cnt;
        break;
    case 'b':
    case 'B':
        ++b_cnt;
        break;
default:
    ++other_cnt;
}
```
Conditional Operators

- `expr1 ? expr2 : expr3`
  - If `expr1` is true, evaluate `expr2`
  - Otherwise, evaluate `expr3`

  \[
  a = \begin{cases} 
  b > c \ ? \ d : e \ & \text{if } (b > c) \\
  & \text{else}
  \end{cases} 
  \]

Declarations and initializations

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td><code>a = 'a', b = 'b'; // a has decimal value 97</code></td>
<td></td>
</tr>
<tr>
<td>int</td>
<td><code>i = 1, j = 2;</code></td>
<td></td>
</tr>
<tr>
<td>double</td>
<td><code>x = 7.07</code></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Expression</th>
<th>Equivalent expression</th>
<th>Value</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>i == j ? a - 1 : b + 1</code></td>
<td><code>(i == j) ? (a - 1) : (b + 1)</code></td>
<td>99</td>
<td>int</td>
</tr>
<tr>
<td><code>j % 3 == 0 ? i + 4 : x</code></td>
<td><code>((j % 3) == 0) ? (i + 4) : x</code></td>
<td>7.07</td>
<td>double</td>
</tr>
<tr>
<td><code>j % 3 ? i + 4 : x</code></td>
<td><code>(j % 3) ? (i + 4) : x</code></td>
<td>5.0</td>
<td>double</td>
</tr>
</tbody>
</table>