Operating Systems Lab (C+Unix)

Enrico Bini

University of Turin

Outline

Outline

Operators with conditions

- Comparison operators
  - == "equal to" (**WARNING**: not =)
  - != "different than"
  - <, <=, >, >=
- Logical operators
  - !, logic NOT
  - &&, logic AND
  - ||, logic OR
- `boolean` type does not exist
- Example of operations among conditions

```
int cond;
cond = x >= 3;
cond = cond && x <= 10;
if (cond) {...}
```

more readable than

```
if (x >= 3 && x <= 10) {...}
```

especially when the condition is long and is broken over many lines

Operators on numbers

- Arithmetic operators
  - `*` multiplication
  - `/` division (integer if both operands are integer)
    - ★ at the end of the next code, what is the value of `x`?

```
int a = 15, b = 6, x;
x = a/b*b;
```

  - `%` remainder of integer division `15 % 6 = ???`
  - `+`, `-` sum and subtracion
- Bit-wise operators: useful to get/set bits of a representation
  - `~`, binary NOT
  - `&`, binary AND
  - `|`, binary OR
  - `^`, binary XOR

```
if (x & 0x80) {
    /* the MS bit of the LS byte is 1  */
}
```

```
x = x ^ 0xFF /* flip the LS byte */
```

The shift operator

- >>, << shift operator (fast way to divide or multiply by 2)
- the shift operator **must be applied to unsigned numbers**
- if applied to signed numbers the result depends on the architecture
  - `(~1 << 4) & 0xFF = ???`

Operators for assignment

- ++, -- increment and decrement
  - the value of a++ is a and then a is incremented
  - when evaluating ++a, the value of a is first incremented. Hence the value of the expression is a+1
- =, assignment (yes, assignment in C are expressions), the returned expression is the value being assigned

```
if (x = 0) {
    /* never taken, x = 0 is always false */
}
```

- *=, /=, %=, +=, -=, <<=, >>=, &=, ^=, |=, compact assignment
  - <expr1> = <expr1> <operator> <expr2> can be written as <expr1> <operator>= <expr2>

Outline

Control constructs: basics

(similar to Java)

- The available constructs to control for the execution flow are:
  - `if ( expr ) <instr>`
  - `if ( expr ) <instr> else <instr>`
  - `while ( expr ) <instr>`
  - `for ( expr ; expr ; expr ) <instr>`
  - `do <instr> while ( expr ) ;`
  - `switch ( )  case:  ....`
  - `break ;`
  - `continue ;`
  - `return [ expr ] ;`
- Above `<instr>` stands for
  - an expression terminated by ";"
  - a control construct
  - a block with curly braces: from "{" to "}"
- The syntax `[...]` denotes an optional argument

"if" control

```
if (<cond-expr>) {
    /* block TRUE */
    ...
}
```

```
if (<cond-expr>) {
    /* block TRUE */
    ...
} else {
    /* block FALSE */
    ...
}
```

- "block TRUE" is executed if <cond-expr> **is not zero**
- "block FALSE", if present, executed if <cond-expr> **is zero**

while loop

```
while (<cond-expr>) {
    /* body of the loop */
    ...
}
```

- body of the loop repeated until <cond-expr> becomes **zero** (which represent "**false**")
- if <cond-expr> is zero the loop is never executed
- if <cond-expr> is always non-zero (not necessarily 1), it loops forever

```
while (1) {
    /* forever-loop */
    ...
    break;
    ...
}
```

do-while loop

```
do {
    /* body of the loop */
    ...
} while (<cond-expr>);
```

for loop

```
for (<expr1>; <expr2>; <expr3>) {
    /* body of the loop */
    ...
}
```

- more natural for looping a known number of times
- <expr1> is evaluated the before the first execution of the for
- <expr2> is evaluate at the beginning of every loop. If zero, then exit the for
- <expr3> is evaluated at the end of every loop
- Classic example (n-times loop)

```
for (i=0; i<n; i++) {
    /* body of the loop */
    ...
}
```

switch construct

```
switch (letter) {
case 'A':
case 'a':
        /* when letter is 'a', or 'A' */
        break;
case 'M':
case 'm':
        /* when letter is 'm', or 'M' */
case 'K':
case 'k':
        /* when letter 'm', 'M', 'k, or 'K' */
        break;
default:
        /* executed otherwise */
        break;
}
```