Jin-Soo Kim
(*jinsoo.kim@snu.ac.kr*)

Systems Software &
Architecture Lab.

Seoul National University

Spring 2023

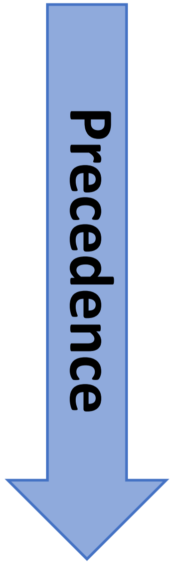
Flow of Control

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`
  - a repetition of action: `while`, `for`, `do-while`

Relational, Equality, Logical Operators


Precedence

Operator	Associativity
( ) ++ (*postfix*) -- (*postfix*)	Left to right
+ (*unary*) - (*unary*) ++ (*prefix*) -- (*prefix*) !	Right to left
* / %	Left to right
+ -	Left to right
< <= > >=	Left to right
== !=	Left to right
&&	Left to right
||	Left to right
?:	Right to left
= += -= *= /= %= *etc.*	Right to left
, (*comma operator*)	Left to right

true: ***nonzero*** **value**

false: ***zero*** **value**

Equality Operators and Expressions (1)

- *expr* == *expr*
  expr != *expr*

Examples

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

Wrong examples

```
a = b               /* assignment */
a = = b - 1         /* space not allowed */
(x + y) =! 44       /* (x + y) = (!44) */
```

- `a == b`
  - The result is either ***true*** (1) or ***false*** (0)
  - It is implemented as `a - b == 0`

Equality Operators and Expressions (2)

Declarations and initializations		
`int i = 1, j = 2, k = 3;`		
Expression	**Equivalent expression**	**Value**
`i == j`	`j == i`	0
`i != j`	`j != i`	1
`i + j + k == - 2 * - k`	`((i + j) + k) == ((-2) * (-k))`	1

- A common programming error

  `if (a == 1) {    ...    }` vs. `if (a = 1)  {    ...    }`

Relational Operators and Expressions (1)

- *expr* < *expr*   *expr* > *expr*
  expr <= *expr*   *expr* >= *expr*

Examples

```
a < 3
a > b
-1.3 >= (2.0*x + 3.3)
```

Wrong examples

```
a =< b                /* out of order */
a < = b               /* space not allowed */
a >> b                /* shift expression */
```

- `a < b`
  - if `a` is less than `b`, then the expression has the `int` value 1 (***true***)
  - Otherwise, the expression has the `int` value 0 (***false***)
  - On many machines, it is implemented as `a - b < 0`

Relational Operators and Expressions (2)

Declarations and initializations

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

Expression	Equivalent expression	Value
`'a' + 1 < c`	`('a' + 1) < c`	1
`- i – 5 * j >= k + 1`	`((-i) – (5 * j)) >= (k + 1)`	0
`3 < j < 5`	`(3 < j) < 5`	1
`x – 3.333 <= x + y`	`(x – 3.333) <= (x + y)`	1

Logical Operators and Expressions (1)

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

Examples

```
a && b
a || b
(a < b) && c
3 && (-2 * a + 7)
```

Wrong examples

```
a &&              /* missing operand */
a | | b           /* space not allowed */
a & b             /* bitwise operator */
&b                /* the address of b */
```

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

Logical Operators and Expressions (2)

<table>
<tr><th colspan="3">Declarations and initializations</th></tr>
<tr><td colspan="3"><pre>char c = 'B';
int i = 3, j = 3, k = 3;
double x = 0.0, y = 2.3;</pre></td></tr>
<tr><th>Expression</th><th>Equivalent expression</th><th>Value</th></tr>
<tr><td>i && j && k</td><td>(i && j) && k</td><td>1</td></tr>
<tr><td>x || i && j - 3</td><td>x || (i && (j - 3))</td><td>0</td></tr>
<tr><td>i < j && x < y</td><td>(i < j) && (x < y)</td><td>0</td></tr>
<tr><td>i < j || x < y</td><td>(i < j) || (x < y)</td><td>1</td></tr>
<tr><td>'A' <= c && c <= 'Z'</td><td>('A' <= c) && (c <= 'Z')</td><td>1</td></tr>
<tr><td>c-1 == 'A' || c+1 == 'Z'</td><td>((c-1) == 'A') || (c+1) == 'Z')</td><td>1</td></tr>
</table>

Logical Operators and Expressions (3)

- Short-circuit evaluation
  - In evaluating the expressions that are the operands of && and ||, the evaluation process stops as soon as the outcome true or false is known

- *expr1* && *expr2*
  - Stops if *expr1* has value zero (false)

- *expr1* || *expr2*
  - Stops if *expr1* has nonzero value (true)

Logical Operators and Expressions (4)

- `!` *expr* (unary negation)

Examples

```
!a
!(x + 7.7)
!(a < b || c < d)
!!c
```

Wrong examples

```
a!                  /* out of order */
a != b              /* "not equal" operator */
```

- `!` *expr*
  - if *expr* has value zero, `!`*expr* has the `int` value 1 (***true***)
  - if *expr* has nonzero value, `!`*expr* has the `int` value 0 (***false***)
  - `!!5` ⇔ `!(!5)` has the value 1

Logical Operators and Expressions (5)

Declarations and initializations

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

Expression	Equivalent expression	Value
`!c`		0
`!(i – j)`		1
`!i - j`	`(!i) - j`	-7
`!!(x + y)`	`!(!(x + y))`	1
`!x * !!y`	`(!x) * (!(!y))`	1

Compound Statement

- "Block"
  - A series of declarations and statements surrounded by braces
  - For grouping statements into an executable unit
  - It is itself a statement, thus it can be placed wherever a statement is placed (no semicolon needed at the end)

```
{
    a = 1;
    {
        b = 2;
        c = 3;
    }
}
```

Empty Statement

- Expression statement
  - An expression followed by semicolon (;)
- 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 */
while(1);           /* infinite loop */
```

`if` Statement

- `if` (*expr*)
  statement
  - If *expr* is nonzero (***true***), then *statement* is executed
  - Otherwise, *statement* is skipped, and control passes to the next statement

```
if (j < k)
{
    min = j;
    printf("j is smaller than k\n");
}
```

if-else Statement

- `if` (*expr*)
      statement1
  `else`
      statement2

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

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

Nested `if` Statements (1)

- `if` (*expr1*)
      `if` (*expr2*)
          statement

```
if (a == 1)
  if (b == 2)     /* if statement is itself a statement */
    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");
```

Nested `if` Statements (2)

```c
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++;
```

⇔

```c
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 (***true***), then *statement* is executed, and control is passed back to *expr*. This repetition continues until *expr* is zero (***false***).
  - The loop body gets executed zero or more times

```
/* This code causes blank characters
   in the input stream to be skipped */

while ((c = getchar()) == ' ')
     ;                                 /* empty statement */
```

for Statement (1)

- for (*expr1*; *expr2*; *expr3*)
      statement

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

  - First, *expr1* (*initialization*) is evaluated.
  - *expr2* is evaluated. If it is nonzero (***true***), 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 (***false***).

for Statement (2)

```c
sum = 0;
for (i = 1; i <= 10; i++)
    sum += i;
```

⇔

```c
sum = 0;
i = 1;
for ( ; i <= 10; i++)
    sum += i;
```

⇔

```c
sum = 0;
i = 1;
for ( ; i <= 10; )
    sum += i++;
```

- What's wrong?

```c
sum = 0;
i = 1;
for ( ; ; )
    sum += i++;
```

- Nested for statements

```c
for (i = 0; i < 4; i++)
    for (j = 0; j < 5; j++)
        for (k = 0; k < 2; k++)
            printf("(%d, %d, %d)\n", i, j, k);
```

Comma Operator

- *expr1* , *expr2*
  - *expr1* is evaluated, and then *expr2*
  - `a = 2, b = a + 1;`

```
for (sum = 0, i = 1; i <= 10; i++)
    sum += i;
```

=

```
for (sum = 0, i = 1; i <= 10; sum += i, i++)
    ;
```

≠

```
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 (***true***), then control is passed back to *statement*.
  - When *expr* is zero (***false***), control passes to the next statement

```
do {
    printf("Input a positive integer: ");
    scanf("%d", &n);
    if (error = (n <= 0))
        printf("\nERROR: Do it again!\n\n");
} while (error);
```

break Statement

- break;
  - Causes an exit from the innermost enclosing loop or switch statement

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

/* break jumps to here */
```

continue Statement

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

```
for (i = 0; i < 10; i++)
{
    c = getchar();
    if (c >= '0' && c <= '9')
        continue;               /* ignore digits */
    nondigits++;

    /* continue transfers control to here */
}
```

switch Statement

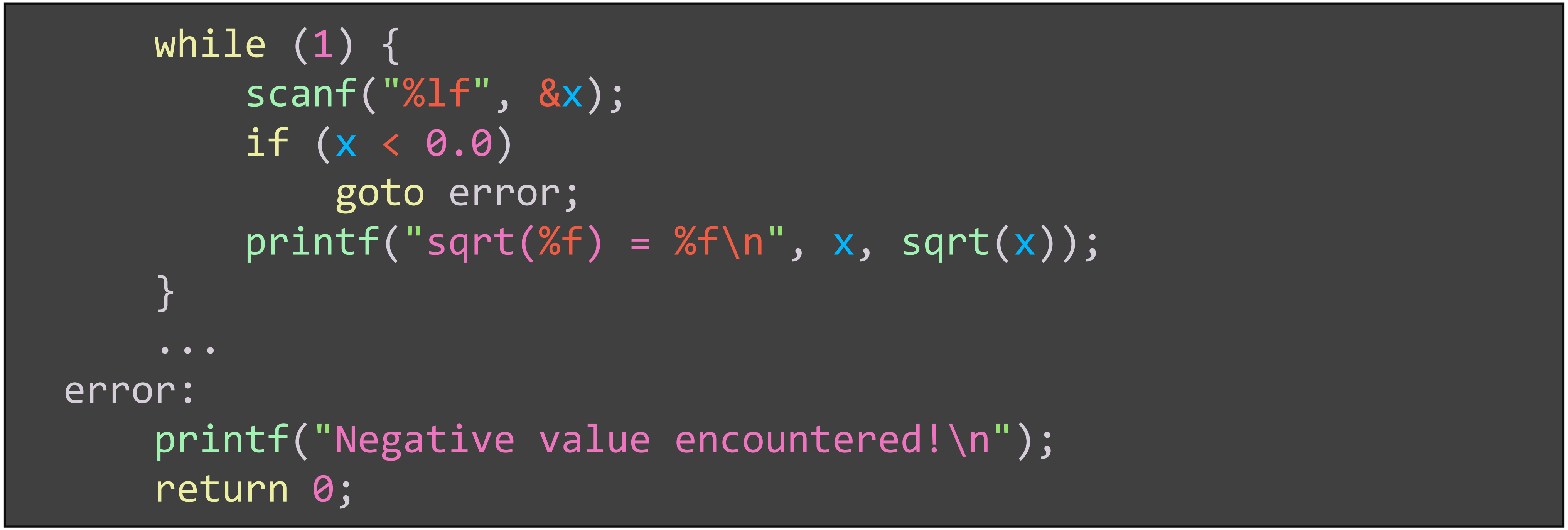
- A multiway conditional statement generalizing the if-else statement

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

- Evaluate the switch expression c (c should be of integral type)
- Go to the case label having a constant value that matches the value of c
- If a match is not found, go to the default label. If there is no default label, terminate the switch statement
- Terminate the switch when a break statement is encountered, or terminate the switch by "falling off the end"

goto Statement

- goto *label*;
  - Causes an unconditional jump to a labeled statement somewhere in the current function
  - "Go To Statement Considered Harmful" (E. Dijkstra, CACM, 1968)



```
    while (1) {
        scanf("%lf", &x);
        if (x < 0.0)
            goto error;
        printf("sqrt(%f) = %f\n", x, sqrt(x));
    }
    ...
error:
    printf("Negative value encountered!\n");
    return 0;
```

Conditional Operator

- *expr1* ? *expr2* : *expr3*
  - First, *expr1* is evaluated
  - If it 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
if (y < z)
    x = y;
else
    x = z;
```

⇔

```c
x = (y < z)? y : z;
```