

Chapter #3: FLOW OF CONTROL

C by Dissection, 4th Ed.

Al Kelley and Ira Pohl

Addison-Wesley Publishing Company, Inc.

Copyright(c) 1996, 1997, SungKyunKwan University, All rights reserved.

No part of these slides may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of School of Electrical and Computer Engineering, SungKyunKwan University.

Relational, Equality, and logical operators(1)

Relational, Equality, and Logical operators

Relational operators:	less than :	<
	greater than:	>
	less than or equal:	<=
	greater than or equal:	>=
Equality operators:	equal :	==
	not equal :	!=
Logical operators:	(unary) negation:	!
	logical and :	&&
	logical or :	

Relational, Equality, and logical operators(2)

- Refer to the course textbook (p.78)

Relational Operators and Expressions(1)

< > <= >=

- They each take two expressions as operands and yield either the *int* value 0 or the *int* value 1.

```
a < 3
```

```
a > b
```

```
-1.3 >= (2.0 * x + 3.3)
```

```
a < b < c
```

```
/* syntactically correct, but confusing */
```

```
a =< b
```

```
/* out of order */
```

```
a < = b
```

```
/* space not allowed */
```

```
a >> b
```

```
/* this is a shift expression */
```

Relational Operators and Expressions(2)

- $a - b < 0$ is equivalent to $(a - b) < 0$.
- The following table shows how the value of $e1 - e2$ determines the values of relational expressions.

values of : $e1 - e2$	$e1 < e2$	$e1 > e2$	$e1 \leq e2$	$e1 \geq e2$
positive	0	1	0	1
zero	0	0	1	1
negative	1	0	1	0

Relational Operators and Expressions(3)

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
x < x + y	x < (x+y)	0

Relational Operators and Expressions(4)

```
j = 7;
printf("%d\n", 3 < j < 5);
/* 1 gets printed, not 0 */
```

- By analogy with mathematics, one might expect that the expression is *false* and that *0* is printed.
- However, that is not the case.

```
3 < j < 5          /* (3 < j) < 5 */
```

- In C, the correct way to write "3 < j < 5" is

```
(3 < j) && (j < 5)
```

Relational Operators and Expressions(5)

- In mathematics the relation $x < x + y$ is equivalent to $0 < y$ (y is positive).
- if x is a floating variable with a large value such as $7e+33$ and y is a floating variable with a small value such as 0.001 ,

`x < x + y /* may be false */`

Equality operators and Expressions(1)

- The equality operators == and != are binary operators acting on expressions.

```
c == 'A'
```

```
k != -2
```

```
x + y == 3 * z - 7
```

```
a = b           /* assignment */
```

```
a = = b - 1     /* space not allowed */
```

```
(x + y) =! 44   /* syntax error: equivalent  
to (x + y) = (!44) */
```

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

Equality operators and Expressions(3)

- $a \neq b$ is equivalent to $!(a == b)$.
- $a == b$ and $a = b$

```
if (a = 1)           /* always true */
    .....          /* do something */
if (a == 1)
    .....          /* do something */
```

Logical operators and Expressions(1)

- The logical operator ! is unary, and the logical operators && and || are binary.

Logical negation expressions

!a

!(x + 7.7)

!(a + b || c < d)

a! /* out of order */

a != b /* != is the token for the
"not equal" operator */

- !!5 is equivalent to !(!5). /* !(0) */

Logical operators and Expressions(2)

Declarations and initializations

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

Expression	Equivalent	value
!c	! c	0
!(i - j)	! (i - j)	1
! i - j	(! i) - j	-7
!! (x + y)	! (! (x + y))	1
! x * !! y	(! x) * (!(! y))	1

Logical operators and Expressions(3)

```
a && b
```

```
a || b
```

```
!(a < b) && c
```

```
3 && (-2 * a + 7)
```

```
a && /* one operand missing */
```

```
a | | b /* extra space not allowed */
```

```
&b /* the address of b */
```

- The precedence of && is higher than ||.
- Their associativity are used to compute the value of some logical expressions.

Logical operators and Expressions(4)

Declarations and initializations

```
char    c = 'B' ;  
int     i = 3, j = 3, k = 3 ;  
double x = 0.0, y = 2.3 ;
```

Expression	Equivalent expression
<code>i && j && k</code>	<code>(i && j) && k</code>
<code>x i && j - 3</code>	<code>x (i && (j - 3))</code>
<code>i < j && x < y</code>	<code>(i < j) && (x < y)</code>
<code>i < j x < y</code>	<code>(i < j) (x < y)</code>
<code>'A' <= c && c <= 'Z'</code>	<code>('A' <= c) && (c <= 'Z')</code>
<code>c - 1 == 'A' c + 1 == 'Z'</code>	<code>((c - 1) == 'A') ((c + 1) == 'z')</code>

Logical operators and Expressions(5)

- In the evaluation of expressions that are the operands of `&&` and `||`, the evaluation process stops as soon as the outcome *true* or *false* is known.

```
int    cnt = 0;
while (++cnt <= 3 && (c = getchar()) != EOF) {
    ..... /* do something */
}
```

- When the expression `++cnt <= 3` is *false*, the next character will not be read.

The Compound statement

- The chief use of the compound statements is to group statements into an executable unit.

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

The Empty statement

- The empty statement is written as a single semicolon.

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

The *if* and the *if-else* statements(1)

```
if (grade >= 90)
    printf("congratulations!\n");
printf("Your grade is %d. \n", grade);
```

- A congratulatory message is printed only when the value of *grade* is greater than or equal to *90*.
- The second *printf()* is always executed.

The *if* and the *if-else* statements(2)

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

- Compound statements should be used to group a series of statements under the control of a single if expression.

The *if* and the *if-else* statements(3)

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

- The code can be written to be more efficient by using a compound statement.

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

The *if* and the *if-else* statements(4)

- If-else statement

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

(Example)

```
if (x < y)
    min = x;
else
    min = y;
```

The *if* and the *if-else* statements(5)

- Consider following example.

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

- The syntax error occurs because the semicolon following the right brace creates an empty statement, and consequently the *else* has nowhere to attach.

The *if* and the *if-else* statements(6)

```
if (a == 1)
    if (b == 2)
        printf("***\n");
else      /* the dangling else problem */
    printf("###\n");
```

- The rule is that an *else* attaches to the nearest *if*.

The *while* statement(1)

- Repetition of action is one reason we rely on computers.
- In C, the *while*, *for*, and *do* statements provide for repetitive action.
- **while statement**

```
while (expr)
    statement
next-statement
```

The *while* statement(2)

- (Example)

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

while ((c = getchar()) != EOF) {
    if (c >= 'a' && <= 'z')
        ++lowercase_letter_cnt;
    ++total_cnt;
}
```

The *while* statement(3)

```
printf("Input an integer:  ");
scanf("%d", &n);
while (--n)
    ..... /* do something */
```

- if a negative integer is assigned to n , then the loop will be infinite.
- The next code will cause blank characters in the input stream to be skipped.

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

The *while* statement(4)

```
/* count blanks, digits, letters,  
           newlines, and others. */  
  
.....  
while ((c = getchar()) != EOF) {  
    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;  
}  
  
.....
```

The *for* statement(1)

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

- is semantically equivalent to

```
expr1;
while (expr2){
    statement
    expr3;
}
next statement
```
- Any or all of the expression in a *for* statement can be missing, but the two semicolons must remain.

The *for* statement(2)

- Examples.

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

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

```
for ( ; ; )                /* infinite loop */
    sum += i++;
```

An Example: Boolean Variables

```
/* Print a table of values
   for some boolean functions. */
```

```
.....
```

```
for(b1 = 0; b1 <= 1; ++b1)
  for(b2 = 0; b2 <= 1; ++b2)
    for(b3 = 0; b3 <= 1; ++b3)
      for(b4 = 0; b4 <= 1; ++b4)
        for(b5 = 0; b5 <= 1; ++b5)
```

```
          printf("%5d%5d%5d...%9d\n",
                ++cnt, b1, b2, b3, b4, b5,
                b1 || b3 || b5, b1 && b2 || b4 && b5,
                b1 + b2 + b3 + b4 + b5 >= 3);
```

```
.....
```

cnt	b1	b2	b3	b4	b5	fct1	fct2	majority
1	0	0	0	0	0	0	0	0
2	0	0	0	0	1	1	0	0
3	0	0	0	1	0	0	0	0
.....								

The comma operator(1)

- The comma operator has the lowest precedence of all the operators in C.
- *expr1* is evaluated first, and then *expr2*.

```
expr1, expr2
```

- This can be used to compute the sum of the integers from 1 to n.

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

- Another form

```
for (sum=0, i=1; i<=n; sum += i, ++i); /* Yes */
for (sum=0, i=1; i<=n; ++i, sum += i); /* No */
```

The comma operator(2)

Comma operator returns the type and value of the right operand

Declarations and initializations

```
int    i, j, k = 3;  
double x = 3.3;
```

Expression	Equivalent expression	value
<code>i=1, j=2, ++k+1</code>	<code>((i=1), (j=2)), ((++k)+1)</code>	5
<code>k != 1, ++x*2.0+1</code>	<code>(k!=1), (((++x)*2.0)+1)</code>	9.6

The *do* statement

- The *do* statement can be considered a variant of the *while* statement.

```
do {  
    sum += i;  
    scanf("%d", &i);  
} while (i > 0);
```

- First statement is executed and *expr* is evaluated.

```
do  
    statement  
while (expr);  
next statement
```

The *goto* statement(1)

- It causes an unconditional jump to a labeled statement somewhere in the current function.
- It can undermine all the useful structure provided by other flow of control mechanisms(*for, while, do, if, switch*).

The *goto* statement(2)

```
bye: exit(1);
L444: a = b + c;
bug1: bug2: bug3: printf("bug found\n");
                        /* multiple labels */
333:  a = b + c;
                        /* 333 is not an identifier */

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

The *goto* statement(3)

- Both the *goto* statement and its labeled statement must be in the body of the same function.

```
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");
```

The *goto* statement(4)

- In general, the *goto* should be avoided.
- A programmer who modifies a program by adding *goto*'s to additional code fragments makes the program incomprehensible.

The *break* and *continue* statements(1)

- The *break* statement causes an exit from the innermost enclosing loop or *switch* statement.

```
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 */
```

The *break* and *continue* statements(1)

- The *continue* statement causes the current iteration of a loop to stop and causes the next iteration of the loop to begin immediately.

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

The *break* and *continue* statements (2)

- The *continue* statement may only occur inside *for*, *while*, and *do* loops.
- *continue* transfers control to the end of the current iteration, whereas *break* would terminate the loop.

The *break* and *continue* statements(3)

```
for (expr1; expr2; expr3) {  
    .....  
    continue;  
    .....  
}
```

- is equivalent to

```
expr1;  
while (expr2) {  
    .....  
    goto next;  
    .....  
next:  
    expr3;  
}
```

The *switch* statement(1)

- The *switch* is a multiway conditional statement generalizing the *if-else* statement.

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

The *switch* statement(2)

- The controlling expression in the parentheses following *switch* must be of integral type.
- After the expression is evaluated, control jumps to the appropriate *case* label.
- If there is no *break* statement, then execution "fall through" to the next statement in the succeeding *case*.

The conditional operator(1)

`expr1 ? expr2 : expr3`

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

```
x = (y < z)? y: z;  
/*  if (y < z) x = y;  
    else x = z;    */
```

The conditional operator(2)

Declarations and initializations

```
char    a = 'a', b = 'b' ;    /* a has decimal value 97 */  
int     i = 1, j = 2;  
double x = 7.07;
```

Expression	Equivalent	value	Type
$i==j ? a-1 : b+1$	$(i==j) ? (a-1) : (b+1)$	99	int
$j\%3==0 ? i+4 : x$	$((j\%3) == 0) ? (i+4) : x$	7.07	double
$j\%3 ? i+4 : x$	$(j\%3) ? (i+4) : x$	5.0	double
