Chapter Four

Boolean Expressions and Control Statements

1

Alternation & Repetition ?

- *How* to write C programs for the following two problems?
- Given a score, print "pass" if it is *greater than or equal to* 60; otherwise, print "fail"
- Print the numbers that is *greater than* 0 *and less than* 1000, and print each number on a separate line

2

Control Statements

- *Conditional* statements
  - The if statements
  - The switch statements
- *Iterative* statements
  - The for statements
  - The while statements
  - The do-while statements

3

Boolean Expressions

- A *Boolean expression* produces *Boolean values* – TRUE *and* FALSE
- In C, TRUE is denoted by *non-zero integers* (usually 1) and FALSE is denoted by 0
- A *Boolean expression* consists of *relational operators* or *logical operators*

4

Relational Operators

- The *relational operators* are used to compare two values
- Test for ordering relationship
  - `>` Greater than
  - `<` Less than
  - `>=` Greater than or equal to
  - `<=` Less than or equal to
- Test for equality and inequality
  - `==` Equal
  - `!=` Not equal

5

Examples

```
4 > 2     → TRUE
4 < 2     → FALSE
3 >= 2    → TRUE
3 <= 2    → FALSE
3 == 3    → TRUE
3 != 3    → FALSE
```

6

Logical Operators

- The *logical operators* perform logical operations
  - `!` Logical not
    (TRUE if the operand is FALSE)
  - `&&` Logical and
    (TRUE if both operands are TRUE)
  - `||` Logical or
    (TRUE if either or both operands are TRUE)

7

Examples

```
!(4 > 2)              → FALSE
!(4 < 2)              → TRUE
3 >= 2 && 3 <= 2      → FALSE
3 >= 2 && 2 <= 3      → TRUE
3 == 2 || 3 != 3      → FALSE
3 == 3 || 3 != 3      → TRUE
```

8

The if Statements

- The if statement is used to express *alternations*
  if (*bool-expr*) $stmt_1$ else $stmt_2$
  where the else clause is optional
- The *bool-expr* is evaluated. If it is TRUE, $stmt_1$ is executed. If it is FALSE and if there is an else clause, $stmt_2$ is executed instead

9

An Example

```
#include <stdio.h>

main( )
{
   int score;

   printf("score = ?"); scanf("%d", &score);
   if (score >= 60)
      printf("Pass!\n");
   else
      printf("Fail!\n");
}
```

10

Four if-Statement Forms

- The single-line if statements
- The multi-line if statements
- The if-else statements
- The cascading if statements

11

The Single-Line if Statements

- Syntax:
  if (*bool-expr*) *stmt*;
- This form is used only for those if statements in which there is *no else clause* and in which the *then clause* is a *single* statement *short* enough to fit on the same line as the if

12

An Example

```
#include <stdio.h>

main( )
{
   int score;

   printf("score = ?");
   scanf("%d", &score);
   if (score > 100) score = 100;
}
```

13

The Multi-Line if Statements

- Syntax:

  if (*bool-expr*) {
  stmts;
  }

- This form is used for those if statements in which there is *no else clause* and in which the *then clause* consists of *multiple* statements or a *single* statement *too long* to fit on a single line

14

An Example

```
#include <stdio.h>

main( )
{
   int score;

   printf("score = ?"); scanf("%d", &score);
   if (score > 100) {
      score = 100;
      printf("Warning: the input score is set to 100!\n");
   }
}
```

15

The if-else Statements

- Syntax:

  if (*bool-expr*) {
  $stmts_T$;
  } else {
  $stmts_F$;
  }

- This form is used for those if statements that have *else clause*. The *then clause* and the *else clause* are always enclosed in a block

16

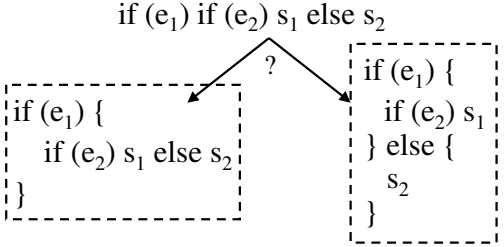
An Example

```
#include <stdio.h>

main( )
{
   int score;

   printf("score = ?"); scanf("%d", &score);
   if (score >= 60) {
      printf("Pass!\n");
   } else {
      printf("Fail!\n");
   }
}
```

17

The Dangling-Else Problem

- The *optional* else clause may create ambiguity



18

The Cascading if Statements

- Syntax:

```
if ( bool-expr1 ) {
   stmts1;
} else if ( bool-exprᵢ ) {
   stmtsᵢ;
} else {
   stmtsnone;
}
```

may appear any number of times

- This form is used for those if statements in which the number of alternatives is larger than two

19

An Example

```
#include <stdio.h>
main( )
{
   int n;

   printf("n = ?"); scanf("%d", &n);
   if (n > 0) {
      printf("That number is positive.\n");
   } else if (n > 0) {
      printf("That number is zero.\n");
   } else {
      printf("That number is negative.\n");
   }
}
```

20

Short-Circuit Evaluation

- The evaluation of a Boolean expression stops as soon as its answer is known. This style of evaluation is called *short-circuit evaluation*
- If the left operand of **&&** is FALSE, there is no need to evaluate the right operand
- If the left operand of **||** is TRUE, there is no need to evaluate the right operand

21

Examples

```
if ( (x != 0) && (y % x == 0) ) { … }

x = y = 3;
if ( (x == 2) && (y++ == 4) ) {
   printf("%d, %d\n", x, y);
} else if ( (x == 2) || (y++ == 5) ) {
   printf("%d, %d\n", x, y);
} else {
   printf("%d, %d\n", x, y);
}
```

22

The ?: Operator

- The *conditional expression*
  (*bool-expr*) **?** $expr_1$ **:** $expr_2$
  evaluates *alternative* expressions
- The *bool-expr* is first evaluated. If it is TRUE, $expr_1$ is evaluated. If it is FALSE, $expr_2$ is evaluated instead

```
max = (x > y) ? x : y;
```

23

The ?: Operator

- The statement
  var **=** (*bool-expr*) **?** $expr_1$ **:** $expr_2$**;**
  is a *shorthand* of the following statement

```
if ( bool-expr ){
   var = expr1;
} else {
   var = expr2;
}
```

24

Examples

```
if ( x > y ){
   max = x;
} else {
   max = y;
}
```

```
max = (x > y) ? x : y;
```

```
printf("%d item%s found\n", n, (n > 1) ? "s" : "");
```

25

Common Pitfalls

- When testing for equality, be sure to use the '==' operator instead of the '=' operator. This error is *extremely common*.

```
if (x == 0) { … }

if (x  =  0) { … }     /* always FALSE */
```

26

Common Pitfalls

- Be careful when using *logical* operators because human languages can be somewhat *fuzzy* in expressing logic

"x is not equal to either 2 or 3"

```
if ( x != 2 || x != 3 ) { … }
if ( !(x == 2 || x == 3) ) { … }
if ( x != 2 && x != 3 ) { … }
```

27

Common Pitfalls

- To test whether a number is *in a particular range*, it is not sufficient to combine relational operators, as is conventional in mathematics

$$0 < x < 10$$

- The two part of the condition must be written explicitly using &&

$$(0 < x) \;\&\&\; (x < 10)$$

28

Common Pitfalls

- Be careful to *avoid redundancy* when using Boolean data

```
if ( done == TRUE ) { … }        if ( done ) { … }

if ( itemsLeft == 0 ) {
   done = TRUE;
} else {
   done = FALSE;
}

done = ( itemsLeft == 0 )
          ? TRUE : FALSE;

done = ( itemsLeft == 0 ) ;
```

29

The Switch Statements

- The *switch* statements are more *efficient* and more *readable* alternative statements than the *cascading if* statements in the cases as the right side

var = *expr*;
if (*var* == c_1) {
  $stmts_1$;
} else if (*var* == c_i) {
  $stmts_i$;
} else {
  $stmts_{none}$;
}

30

The Switch Statements

- The *switch* statement

  switch (*expr*) {
    case c_1: $stmt_1$; break;
    case c_i: $stmt_i$; break;
    default: $stmt_{none}$; break;
  }

  may appear any number of times

  evaluates *expr* first and then *directly* executes $stmt_i$ if *expr*'s value equals to *integral constant* c_i; it *directly* executes $stmt_{none}$ otherwise

31

An Example

```
#include <stdio.h>
main( )
{
   int cardRank;

   printf("cardRank = ?"); scanf("%d", &cardRank);
   switch (cardRank) {
      case 1: printf("Ace\n"); break;
      case 11: printf("Jack\n"); break;
      case 12: printf("Queen\n"); break;
      case 13: printf("King\n"); break;
      default: printf("%d\n", cardRank); break;
   }
}
```

32

The Break Statements

- The execution of the *break* statement in a switch statement *terminates* the execution of the *switch* statement
- It is a *good* programming practice to include a *break statement* at the end of every clause
- It is a *good* programming practice to include a *default clause*

33

An Example

```
#include <stdio.h>
main( )
{
   int n;

   printf("n (1~4) = ?"); scanf("%d", &n);
   switch (n) {
      case 1:
      case 3: printf("odd number\n"); break;
      case 2:
      case 4: printf("even number \n"); break;
      default: printf("illegal input%d\n", n); break;
   }
}
```

34

The For Statements

- The *for* (*loop*) statement

```
for (init; test; step) {
   stmts;
}
```

loop body

(1) first evaluates *init*, and then evaluates *test*;
(2) if *test*'s value is TRUE, it executes *stmts*, evaluates *step* and *test*, and repeats step (2);
(3) if *test*'s value is FALSE, it terminates

35

An Example

```
#include <stdio.h>

main( )
{
   int i, limit;

   printf("limit (1~999) = ?"); scanf("%d", &limit);
   for (i = 1; i <= limit; i++) {
      printf("%d\n", i);              /* 1 <= i <= limit */
   }
}
```

36

An Example

i	limit	i < limit	print(i)	i++
1	6	TRUE	1	2
2	6	TRUE	2	3
3	6	TRUE	3	4
4	6	TRUE	4	5
5	6	TRUE	5	6
6	6	FALSE		

37

Definite Loops

- The *for* statements should be used when the *number of iterations* of the loop is *known before* entering the loop
- The *test* expression is evaluated at each iteration. It is better to evaluate it *once and for all* before entering the loop
- The *index variable* of the *for* statements may step *increasingly* or *decreasingly*, and may step *more than one* each iteration

38

An Example

```
#include <stdio.h>

main( )
{
   int i, lower, upper;

   printf("upper (1~999) = ?"); scanf("%d", &upper);
   lower = upper / 2;
   for (i = upper; i >= lower; i -= 2) {
      printf("%d\n", i);      /* lower <= i <= upper */
   }
}
```

decreasing and > 1

expr

variable

39

Infinite Loops

- The expressions *init*, *test*, and *step* are each *optional*, but *semicolons* must appear
- If *init* is missing, no initialization is done
- If *test* is missing, it is assumed to be TRUE
- If *step* is missing, no action occurs between iterations
- An *infinite loop* can be specified as
  for (; ;) { … }

40

Nested Loops

- If a *for* loop is *nested* inside another *for* loop, the *inner for* loop is executed once for each iteration of the *outer for* loop
- Each *for* loop must have its *own* (*distinct*) *index variables* so that the variables do not interfere with each other

41

An Example

```
#include <stdio.h>
main( )
{
   int i, j;

   for (i = 1; i <= 5; i++) {
      for (j = 1; j <= 5; j++) {
         printf("(%d, %d) ", i, j);
      }
      printf("\n");
   }
}
```

42

An Example

(1, 1) (1, 2) (1, 3) (1, 4) (1, 5)
(2, 1) (2, 2) (2, 3) (2, 4) (2, 5)
(3, 1) (3, 2) (3, 3) (3, 4) (3, 5)
(4, 1) (4, 2) (4, 3) (4, 4) (4, 5)
(5, 1) (5, 2) (5, 3) (5, 4) (5, 5)

43

Common Pitfalls

- Be very careful when *testing floating-point numbers for equality*. Because floating-point numbers are only *approximations*, they might not behave in the same way as real numbers in mathematics
- It is best to use an *integer index variable* and avoid using a floating-point index variable in a loop

44

An Example

```
for (x = 1.0; x <= 2.0; x += 0.1) {
   printf("%lf\n", x);
}

for (i = 10; i <= 20; i++) {
   x = i / 10.0;
   printf("%lf\n", x);
}
```

45

Common Pitfalls

- The *index variable* in a loop is used to control the number of iterations of the loop, and should be updated *only* in *init* and *step* expressions

```
for (i = 1; i <= 10; i++) {
   x = i / 10.0;
   printf("%lf\n", x);
   i++;
}
```

46

An Example

```
#include <stdio.h>
main( )
{
   int i, n, sum;

   printf("n = ?"); scanf("%d", &n);
   sum = 0;                    /* initialization */
   for (i = 1; i <= n; i++) {
         sum += i;             /* sum = ∑j=1,i j */
   }
   printf("%d\n", sum);   /* sum = ∑j=1,n j */
}
```

47

The While Statements

- The *while* (*loop*) statement

```
while (bool-expr) {
   stmts;
}
```

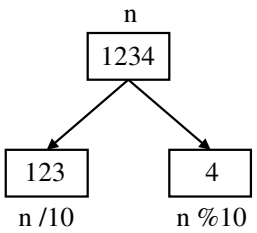
loop body

(1) first evaluates *bool-expr*;
(2) if *bool-expr*'s value is TRUE, it executes *stmts*, evaluates *bool-expr*, and repeats step (2);
(3) if *bool-expr*'s value is FALSE, it terminates

48

An Example

Sum the digits in a number



49

An Example

```
#include <stdio.h>
main( )
{
   int n, dsum;

   printf("n = ?"); scanf("%d", &n);
   dsum = 0;                    /* initialization */
   while (n > 0) {
         dsum += n % 10;
         n /= 10;
   }
   printf("%d\n", dsum);
}
```

50

Indefinite Loops

- The *while* statements should be used when the *number of iterations* of the loop is still *unknown before* entering the loop
- The *bool-expr* is evaluated at the *beginning* of each iteration. If *bool-expr* is FALSE initially, the body of the loop is *not executed at all*

51

Infinite Loops

- Think carefully about the *bool-expr* in a *while* statement so that you are sure the loop will *eventually terminate*

```
while (n > 0) {
   dsum += n % 10;
   n /= 10;
}
```

```
while (n >= 0) {
   dsum += n % 10;
   n /= 10;
}
```

- An *infinite loop* can be specified as
  `while ( 1 ) { … }`

52

The loop-and-a-half Problem

- A *loop* problem is called a *loop-and-a-half* problem if the *test* for the termination of the loop is most naturally somewhere in the *middle* of the loop
- The execution of a *break* statement within a loop *immediately terminates* the innermost enclosing loop

53

An Example

```
#include <stdio.h>
main( )
{
   int n, sum;

   sum = 0;                    /* initialization */
   printf("n (0 to stop) = ?"); scanf("%d", &n);
   while (n) {
         sum += n;             /* sum = ∑ n */
         printf("n (0 to stop) = ?"); scanf("%d", &n);
   }
   printf("%d\n", sum);   /* sum = ∑ n */
}
```

55

An Example

```
#include <stdio.h>
main( )
{
   int n, sum;

   sum = 0;                    /* initialization */
   while (1) {
      printf("n (0 to stop) = ?"); scanf("%d", &n);
      if (n == 0) break;
      sum += n;                /* sum = ∑ n */
   }
   printf("%d\n", sum);   /* sum = ∑ n */
}
```

54

Relationship: for & while

The *for* statement

```
   for (init; test; step) {
      statements;
   }
```

is identical in operation to the *while* statement

```
   init;
   while (test) {
      statements;
      step;
   }
```

56

The Do-While Statements

- The *do-while* (*loop*) statement

```
do {
    stmts;
} while (bool-expr)
```

loop body

(1) it first executes *stmts* , then evaluates *bool-expr*;
(2) if *bool-expr*'s value is TRUE, it repeats step (2);
(3) if *bool-expr*'s value is FALSE, it terminates

57

Indefinite Loops

- The *do-while* statements should be used when the *number of iterations* of the loop is still *unknown before* entering the loop
- The *bool-expr* is evaluated at the *end* of each iteration. The body of the loop is *executed at least once*

58

An Example

Remove adjacent duplicates

1 1	2 2 2	3	1	4 4	5 5 5 5	6	7 7

1 2 3 1 4 5 6 7

59

An Example

```
#include <stdio.h>
main( )
{
   int x, next;

   printf("x = ?"); scanf("%d", &x);
   while (x != 0) {
        printf("%d\n", x);
          do {
            printf("x = ?"); scanf("%d", &next);
        } while (next == x);
        x = next;
   }
}
```

60

Relationship: do-while & while

The *do-while* statement

```
do {
    statements;
}while (bool-expr)
```

is identical in operation to the *while* statement

```
statements;
while (bool-expr) {
    statements;
}
```

61

Numeric Data Types

- Integer numbers
  - (unsigned) char (1 byte)
  - (unsigned) short (2 bytes)
  - (unsigned) int (4 bytes)
  - (unsigned) long (4 or 8 bytes)
- Floating-point numbers
  - float (4 bytes)
  - double (8 bytes)

62

Formatted Output

- The *format code %-w.pl*X
- d, i: decimal number
- o: unsigned octal number
- x, X: unsigned hexadecimal number
- u: unsigned decimal number
- c: single character
- s: string

63

Formatted Output

- The *format code %-w.pl*X
- f: floating-point; [-]m.dddddd
- e, E: floating-point; [-]m.dddddd e± xx
- g, G: floating-point; use %e or %E if the exponent is less than -4 or greater than or equal to the precision; otherwise, use %f
- p: pointer
- %: no argument is converted; print a %

64

Formatted Output

- The *format code %-w.pl*X
- Alignment:-, left, otherwise, right
- Width: the minimum number of characters; it will be padded if necessaey
- Precision: the number of digits after the decimal point of a floating-point, the minimum number of digits for an integer, or the maximum number of characters for a string
- l/h: *l* represents long, *h* represents short

65

Examples

```
:%lf:          :12.123456:
:%12lf:        :    12.123456:
:%-12lf:       :12.123456    :
:%.2lf:        :12.12:
:%.10lf:       :12.1234563333:
:%15.10lf:     :   12.1234563333:
:%-15.10lf:    :12.1234563333   :
```

66