OPERATORS AND EXPRESSIONS

PERFORMING COMPUTATIONS

C provides operators that can be applied to calculate expressions:

example: tax is 8.5% of the total sale

expression: tax = 0.085 * totalSale

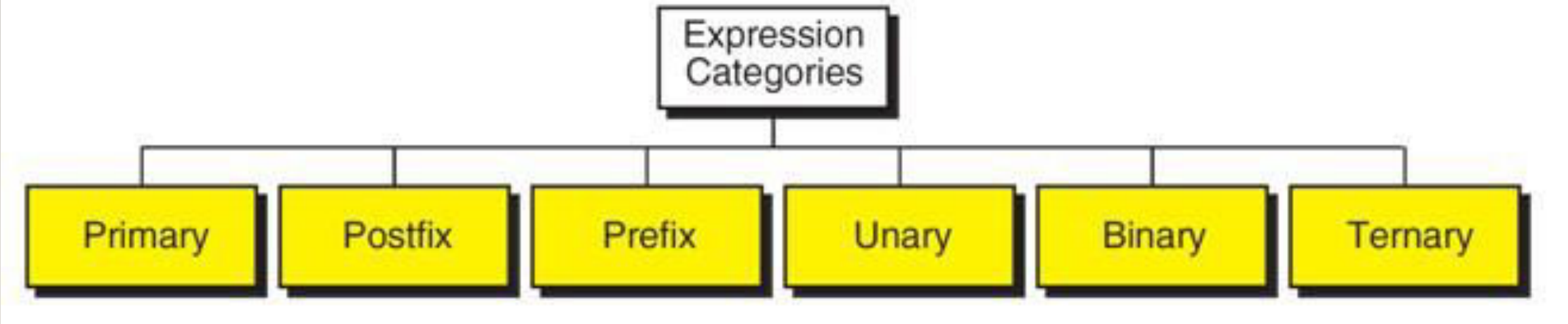
Need to specify what operations are legal, how operators evaluated, etc.

C operations are referred to as Expressions

EXPRESSIONS

- An expression is a sequence of operands and operators that reduces to a single value.
- Expressions can be simple or complex.
- An operator is a syntactical token that requires an action be taken.
- An operand is an object on which an operation is performed; it receives an operator's action.

EXPRESSIONS



EXPRESSIONS IN C

- Example: `total * TAXRATE`
- Consists of a sequence of operators and operands
  - operands: total, TAXRATE
    - can be expressions themselves
  - operator: *
    - operation calculating a result based on the operand(s)

SOME EXPRESSION FORMATS

- primary
  - expressions that directly map to a value
- unary
  - prefix: *operator operand*
  - postfix: *operand operator*
- binary
  - *operand1 operator operand2*
- ternary (operator in two parts)

TYPES FOR OPERANDS

- Operators only apply to certain types
  - e.g., addition applies to numbers
- Each operand must be of the appropriate type
- Operators calculate a value
- Value can then be used as operands to other operators

PRIMARY EXPRESSIONS

- identifiers
  - variable
  - defined constant
- literal constants
  - operands can be values of the appropriate type
  - operator can have constant (e.g., 5, -34) as operand

BINARY EXPRESSIONS

expr1 op expr2

arithmetic operators

+ - binary addition (e.g., 5 + 3 result: 8)

- - binary subtraction (e.g., 5 - 3 result: 2)

* - multiplication (e.g., 5 * 3 result: 15)

/ - division (e.g., 5 / 3 result: 1)

% - remainder (e.g., 5 % 3 result: 2)

arithmetic operators apply to number types, result type depends on operand types

ARITHMETIC OPERATORS

Two int operands -> int result

Two float operands -> float result

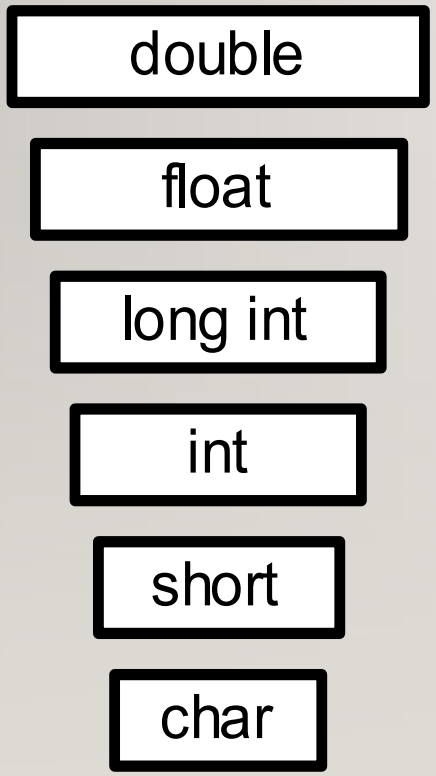
/ - division operator, is whole number division for ints, floating point division for other operands

% - remainder operator, only works for int values

Mixed operands (float/int) - result produced is based on a "promotion" hierarchy, result cast to higher type in hierarchy

e.g., 5 / 3.0 produces 1.666666

PROMOTION HIERARCHY



- Operands of the lower type can be implicitly converted to the higher type (in C)
- Occurs when an operator expects the higher type (adding together two floats when one is an int)

OPERATORS IN C

C language supports a lot of operators to be used in expressions.
These operators can be categorized into the following major groups:

1) Arithmetic operators
2) Relational Operators
3) Equality Operators
4) Logical Operators
5) Unary Operators
6) Conditional Operators
7) Bitwise Operators
8) Assignment operators
9) Comma Operator
10) Sizeof Operator

Arithmetic operators

- Assume the values a=9 and b=3

OPERATION	OPERATOR	SYNTAX	COMMENT	RESULT
Addition	+	a + b	result = a + b	12
Subtraction	-	a - b	result = a – b	6
Multiply	*	a * b	result = a * b	27
Divide	/	a / b	result = a / b	3
Modulus	%	a % b	result = a % b	0

```c
/* This program demonstrates binary expressions.
        Written by:
        Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int   a = 17;
   int   b = 5;
   float x = 17.67;
   float y = 5.1;

// Statements
   printf("Integral calculations\n");
   printf("%d + %d  = %d\n", a, b, a + b);
```

```
25      printf("%f - %f   = %f\n", x, y, x - y);
26      printf("%f * %f   = %f\n", x, y, x * y);
27      printf("%f / %f   = %f\n", x, y, x / y);
28      return 0;
29  }   // main
```

```
Results:
Integral calculations
17 + 5 = 22
17 - 5 = 12
17 * 5 = 85
17 / 5 = 3
17 % 5 = 2

Floating-point calculations
17.670000 + 5.100000  = 22.770000
17.670000 - 5.100000  = 12.570000
17.670000 * 5.100000  = 90.116997
17.670000 / 5.100000  = 3.464706
```

Relational Operators

- Thease operators compares two values so also called Comparison operators.
- Relational operators return true or false value, depending on the conditional relationship between

OPERATOR	MEANING	EXAMPLE
<	LESS THAN	3 < 5 GIVES 1
>	GREATER THAN	7 > 9 GIVES 0
<=	LESS THAN OR EQUAL TO	100 <= 100 GIVES 1
>=	GREATER THAN EQUAL TO	50 >=100 GIVES 0

Equality Operators

- C language supports two kinds of equality operators to compare their operands for strict equality or inequality. They are equal to (==) and not equal to (!=) operator.

OPERATOR	MEANING
==	RETURNS 1 IF BOTH OPERANDS ARE EQUAL, 0 OTHERWISE
!=	RETURNS 1 IF OPERANDS DO NOT HAVE THE SAME VALUE, 0 OTHERWISE

Logical Operators

- C language supports three logical operators.

  They are 1) Logical AND (&&)

  2) Logical OR (||)

  3) Logical NOT (!)

- In case of arithmetic expressions, the logical expressions are evaluated from left to right.

A	B	A &&B
0	0	0
0	1	0
1	0	0
1	1	1

A	B	A || B
0	0	0
0	1	1
1	0	1
1	1	1

A	!A
0	1
1	0

UNARY EXPRESSIONS

- Postfix

  fName(*arglist*) - function call

  subprogram is called (invoked) which performs some set of computations and returns a value (more later)

- Prefix

  sizeof (*expr*) - expr can be any primary expression or the name of a type, operator returns num bytes used to represent type (int)

  + *expr* - unary plus (applies to numbers, no effect)

  - *expr* - unary minus (inverts sign of numbers)

Unary Operators

- Unary operators act on single operands.
- C language supports three unary operators.

  They are :

  1) Unary minus(-)
  2) Increment (++)
  3) Decrement(--)

- When an operand is preceded by a minus sign, the unary operator negates its value.

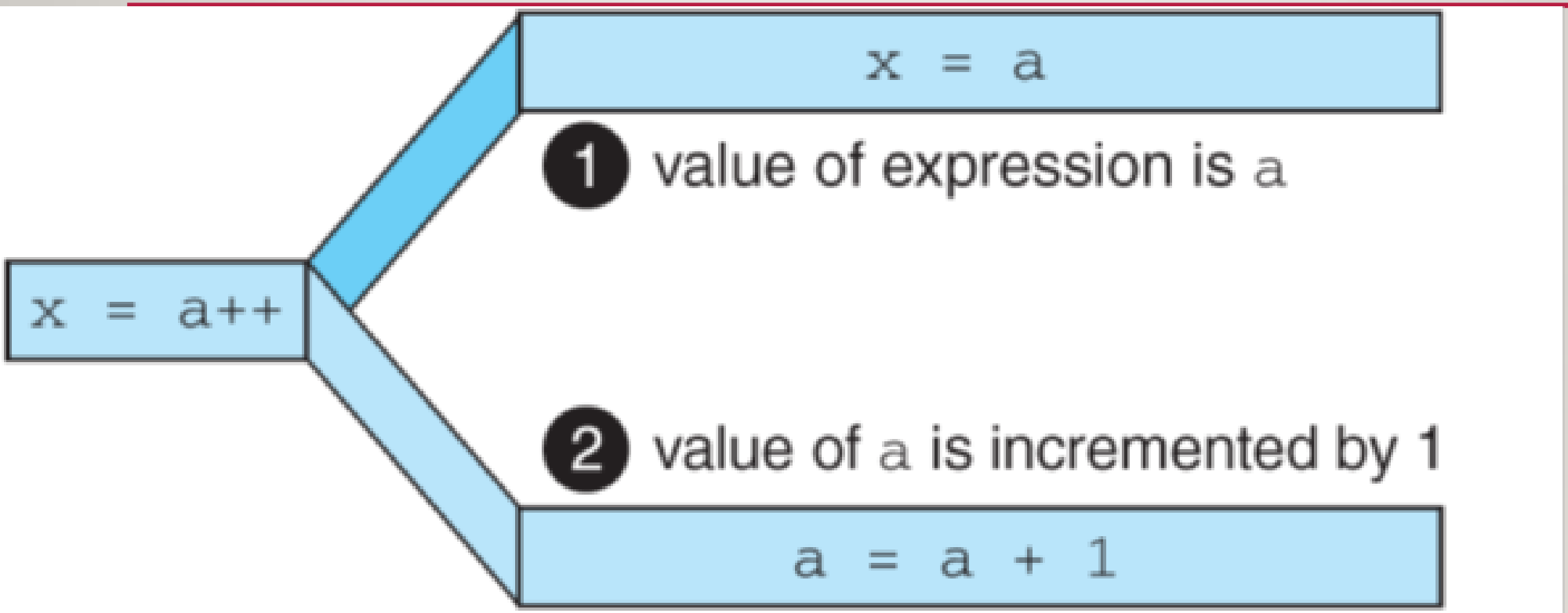
  Ex: int x=5; int y= -x; then y store the value -5.

- The increment operator increases the value of its operand by 1.

  Ex: int x=3,y;

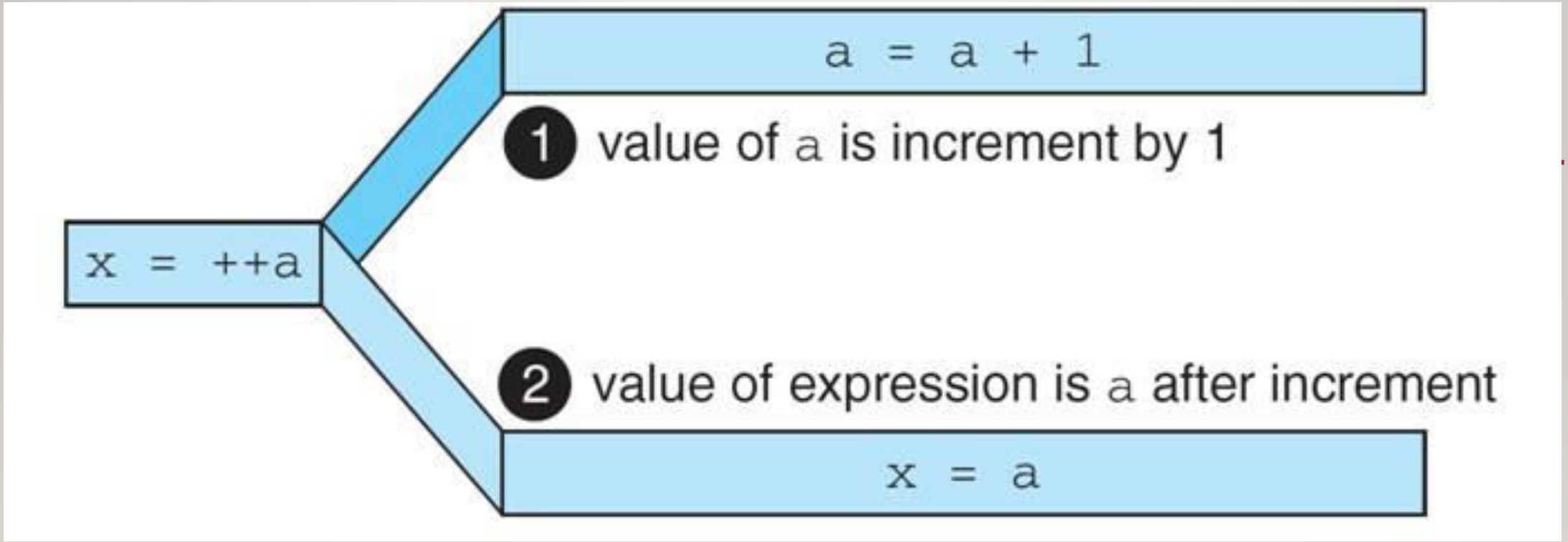
```
printf("x=%d",x++);     x=3
y=x;                    y=4;
printf("y=%d",++y);     y=5;
```

UNARY EXPRESSIONS



```c
/* Example of postfix increment.
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int a;

// Statements
   a = 4;
   printf("value of a    : %2d\n", a);
   printf("value of a++  : %2d\n",   a++);
   printf("new value of a: %2d\n\n", a);
   return 0;
}  // main
```

```
Results:
value of a    :  4
value of a++  :  4
new value of a:  5
```


a = a + 1
1 value of a is increment by 1
x = ++a
2 value of expression is a after increment
x = a

```c
/* Example of postfix increment.
      Written by:
      Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int a;

// Statements
   a = 4;
   printf("value of a    : %2d\n", a);
   printf("value of a++  : %2d\n",   a++);
   printf("new value of a: %2d\n\n", a);
   return 0;
}  // main
```

```
Results:
value of a    :  4
value of ++a  :  5
new value of a:  5
```

UNARY PLUS AND MINUS

Expression	Contents of a Before *and* After Expression	Expression Value
+a	3	+3
-a	3	-3
+a	-5	-5
-a	-5	+5

CAST OPERATOR (UNARY PREFIX OP)

- (*TypeName*) *expression*
  - conversion forces expression value from current type to the provided type
    - example: (float) 5 results in 5.0
  - especially useful for division
  - example:
    - totalScore, totalGrades (ints)
    - totalScore / totalGrades produces integer
    - (float) totalScore / totalGrades casts totalScore to a floating point number, then division occurs

```
#include <stdio.h>
int main()
{
    char ch = 'g';
    int N = (int)(ch);
    printf("%d", N);
    return 0;
}
```

Operators in C

Unary Operators

- The decrement operator decreases the value of its operand by 1.

```
Ex:  int  x=3,y;
     printf("x=%d",x--);      x=3
     y=x;                     y=2;
     printf("y=%d",--y);      y=1;
```

Conditional Operator or Ternary Operator

- The conditional operator (?:) is just like an if .. else statement.
- The syntax of the conditional operator is

```
        exp1 ? exp2 : exp3
Ex:       a=10   b=5
     large = ( a > b) ? a : b
     large=10.
```

Operators in C

Bitwise Operators

- Bitwise operators perform operations at bit level.

They are : 1) Bitwise AND(&)

2) Bitwise OR (|)

3) Bitwise XOR (^)

4) Bitwise Shift (<< and >>)

5) Bitwise NOT (~)

- The bitwise AND operator (&) is a small version of the boolean AND (&&) as it performs operation on bits instead of bytes, chars, integers, etc.

A	B	A & B
0	0	0
0	1	0
1	0	0
1	1	1

Ex: x=2 y=3 x&y

0 0 1 0

0 0 1 1

x&y= 0 0 1 0

Operators in C

- The bitwise OR operator (|) is a small version of the boolean OR (||) as it performs operation on bits instead of bytes, chars, integers, etc.

A	B	A | B
0	0	0
0	1	1
1	0	1
1	1	1

Ex: x=2 y=3 x|y

0 0 1 0

0 0 1 1

---

x|y= 0 0 1 1

- The bitwise XOR operator (^) performs operation on individual bits of the operands.

A	B	A ^ B
0	0	0
0	1	1
1	0	1
1	1	0

Ex: x=2 y=3 x^y

0 0 1 0

0 0 1 1

---

x^y= 0 0 0 1

Operators in C

- In bitwise Shift operations, the digits are moved, or *shifted*, to the left or right.
- The CPU registers have a fixed number of available bits for storing numerals, so when we perform shift operations; some bits will be "shifted out" of the register at one end, while the same number of bits are "shifted in" from the other end.

Ex:

In a left arithmetic shift,the right side end filled with 0's.

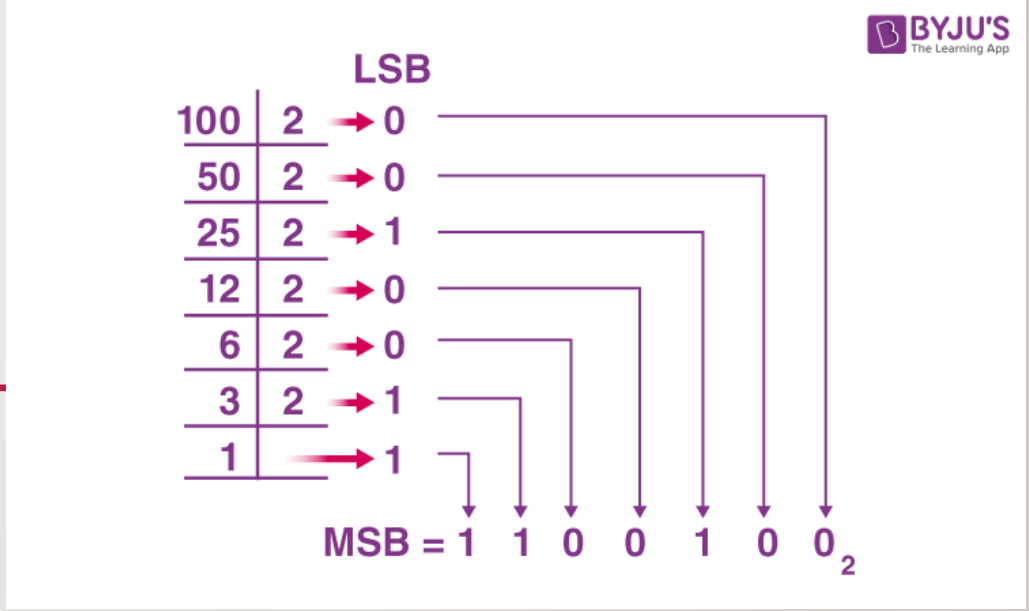
int x = 11000101;

Then x << 2 = 00010100

In a right arithmetic shift,the left side end filled with 0's.

int x = 11000101;

Then x >> 2 = 00110001




```
1.#include <stdio.h>
2.int main ()
3.{
4.// declare local variable
5.int num;
6.printf (" Enter a positive number: ");
7.scanf (" %d", &num);
8.// use left shift operator to shift the bits
9.num = (num << 2); // It shifts two bits at the left side
10.printf (" \n After shifting the binary bits to the left side. ")
;
11.printf (" \n The new value of the variable num = %d", nu
m);
12.return 0;
13.}
```

- 11001 -25
- 1100100 -100

```
Enter a positive number: 25
After shifting the binary bits to the left side.
The new value of the variable num = 100
```

Operators in C

- The bitwise NOT (~), or complement, is a unary operation that performs logical negation on each bit of the operand.

- By performing negation of each bit, it actually produces the 1's complement of the given binary value.

Ex: int x=4,y;

y=(~x); Hear x is 0 1 0 0

y=11 ~x is 1 0 1 1

ASSIGNMENT EXPRESSION

varName = expr

- value of expr determined
- result of expression is the value of expr
- as a "side effect" the value calculated is also stored in the named variable (the current value stored in the variable is replaced)
- example:
  - current value of total is 5
  - total = 3
  - replaces the value 5 with the value 3 in the memory location corresponding to total

ASSIGNMENT EXPRESSION (CONT)

- *varName = expr*
  - expr value must be of the proper type to be stored in the variable (error otherwise)
  - the value may be converted automatically using the promotion hierarchy
    - example:
    - float f;
    - f = 3; /* 3 cast from an integer to a float value */

SHORTHAND ASSIGNMENTS

- += -= *= /= %=
  - example:
    - x += 3
  - is a shorthand for
    - x = x + 3
  - assuming x is 7 before x += 3 is calculated, the right-hand side x + 3 is calculated, producing 10, this value is then stored in x, and the result of the expression is 10
  - thus, these operators also have side effects

Operators in C

Assignment operators

- The assignment operator is responsible for assigning values to the variables.
- The equal sign (=) is the fundamental assignment operator.
- C supports other assignment operators that provide shorthand (Compact)ways to represent common variable assignments.

OPERATOR	SYNTAX	EQUIVALENT TO
/=	variable /= expression	variable = variable / expression
\=	variable \= expression	variable = variable \ expression
*=	variable *= expression	variable = variable * expression
+=	variable += expression	variable = variable + expression
-=	variable -= expression	variable = variable - expression
&=	variable &= expression	variable = variable & expression
^=	variable ^= expression	variable = variable ^ expression
<<=	variable <<= amount	variable = variable << amount
>>=	variable >>= amount	variable = variable >> amount

```c
/* Demonstrate examples of compound assignments.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int x;
   int y;

// Statements
   x = 10;
   y = 5;

```

```
17      printf("x: %2d  |  y: %2d ",     x, y);
18      printf("  |  x *= y + 2: %2d ", x *= y + 2);
19      printf("  |  x is now: %2d\n",  x);
20
21      x = 10;
22      printf("x: %2d  |  y: %2d ",     x, y);
23      printf("  |  x /= y + 1: %2d ", x /= y + 1);
24      printf("  |  x is now: %2d\n",  x);
25
26      x = 10;
27      printf("x: %2d  |  y: %2d ",      x, y);
28      printf("  |  x %%= y - 3: %2d ", x %= y - 3);
29      printf("  |  x is now: %2d\n", x);
30
31      return 0;
32  }   // main
```

```
Results:
x: 10  |  y:  5  |  x *= y + 2: 70  |  x is now: 70
x: 10  |  y:  5  |  x /= y + 1:  1  |  x is now:  1
x: 10  |  y:  5  |  x %= y - 3:  0  |  x is now:  0
```

Operators in C

Comma operator

- The Comma operator in C takes two operands. It works by evaluating the first and discarding its value, and then evaluates the second and returns the value as the result of the expression.
- Comma separated operands when chained together are evaluated in left-to-right sequence with the right-most value yielding the result of the expression.
- Among all the operators, the comma operator has the lowest precedence.

Ex:

```
int a=2, b=3, x=0;
x = (++a, b+=a);
```

Now, the value of x = 6.

Operators in C

Sizeof Operator

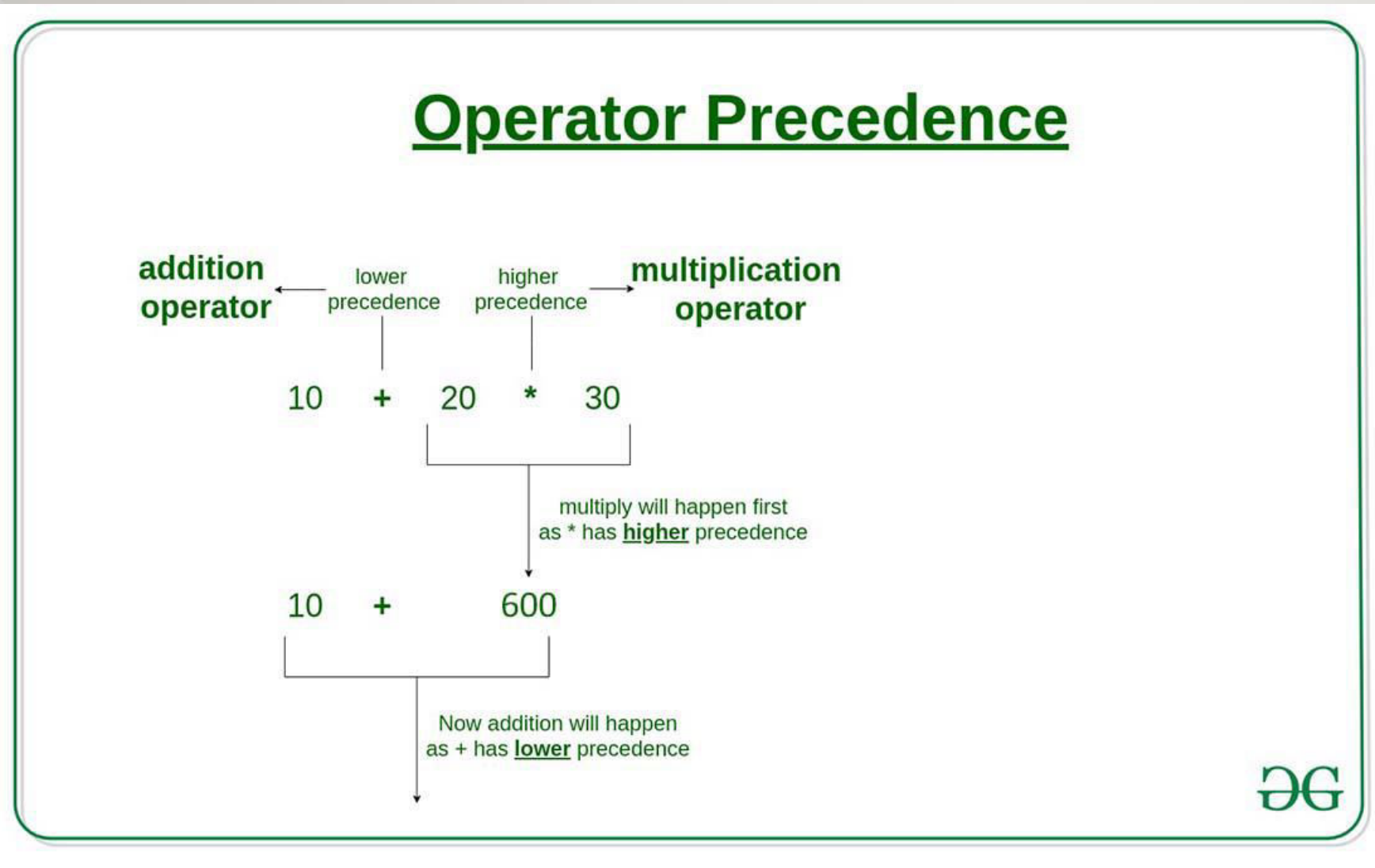
- Sizeof operator used to calculate the sizes of data types.
- It can be applied to all data types.
- The operator returns the size of the variable, data type or expression in bytes.

Ex:

→ sizeof(char) returns 1 byte, that is the size of a character data type. If we have,

```
int a = 10;
unsigned int result;
result = sizeof(a);
```

then result = 2 bytes.



- **Operator precedence determines which operation is performed first in an expression with more than one operators with different precedence.**

OPERATOR PRECEDENCE

Operator	Description	Associativity
() [ ] . -> ++ --	Parentheses: grouping or function call Brackets (array subscript) Member selection via object name Member selection via pointer Postfix increment/decrement	left-to-right
++ -- + - ! ~ (*type*) * & sizeof	Prefix increment/decrement Unary plus/minus Logical negation/bitwise complement Cast (convert value to temporary value of *type*) Dereference Address (of operand) Determine size in bytes on this implementation	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right

== !=	Relational is equal to/is not equal to	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
|	Bitwise inclusive OR	left-to-right
&&	Logical AND	left-to-right
||	Logical OR	left-to-right
? :	Ternary conditional	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left
,	Comma (separate expressions)	left-to-right

OPERATOR PRECEDENCE

18: Identifier, Constant, Parenthesized Expression

17: Function call

16: Postfix increment/decrement,

15: Prefix increment/decrement, sizeof, unary +, unary -

14: (Type) - cast operator

13: * / %

12: + -

2: = += -= *= /= %=

USING OPERATOR PRECEDENCE

- Example:
  - total = totalSale + TAXRATE * totalSale
  - * has highest precedence, so TAXRATE * totalSale is calculated first
  - + has next highest precedence, so the result of TAXRATE * totalSale is added to totalSale
  - = has the lowest precedence, so the value calculated in the previous operations is stored in total

PARENTHESIZED EXPRESSIONS

- What if the precedence order is not what you want?
  - example: sale1 + sale2 * TAXRATE
  - in this case the multiplication would happen first, but you might want the addition first
  - answer: parenthesize the expressions you want to happen first
  - result: (sale1 + sale2) * TAXRATE
  - parenthesized expressions have highest precedence

PROGRAMMING TIP: PARENTHESIZING

- Does not hurt to include parentheses for all expressions
- Thus you can guarantee the order of evaluation

  Example:

  total = totalSale + TAXRATE * totalSale

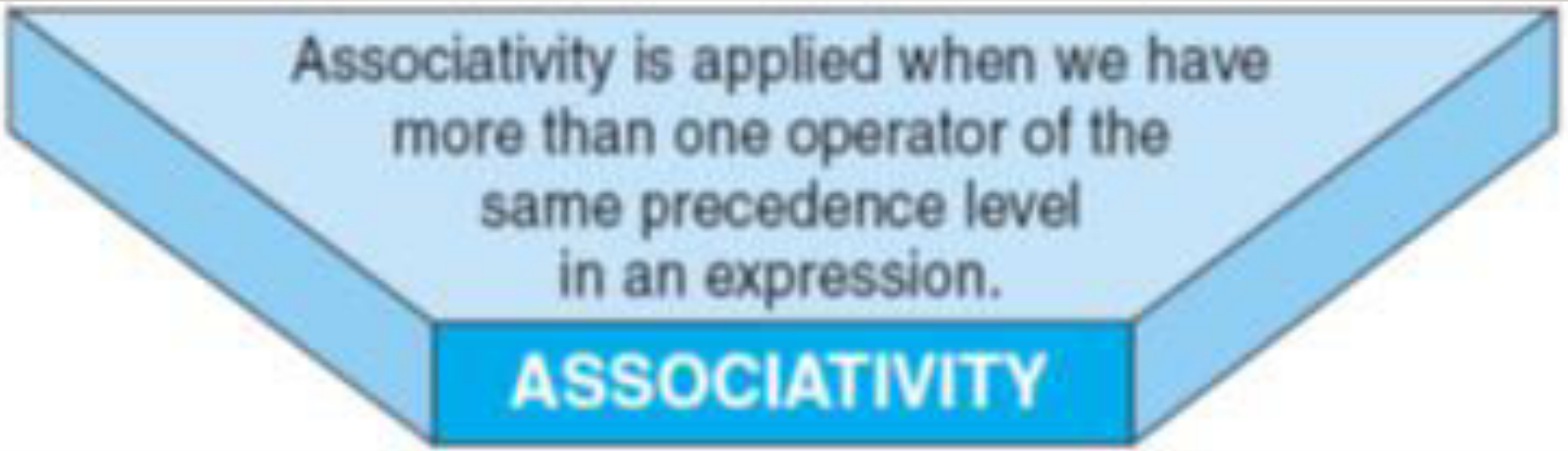
  becomes

  total = (totalSale + (TAXRATE * totalSale))

```
 3          Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9  // Local Declarations
10     int a = 10;
11     int b = 20;
12     int c = 30;
13
14  // Statements
15     printf ("a *  b + c  is: %d\n", a *  b + c);
16     printf ("a * (b + c) is: %d\n", a * (b + c));
17     return 0;
18  }  // main
```

```
Results:
a *  b + c  is: 230
a * (b + c) is: 500
```

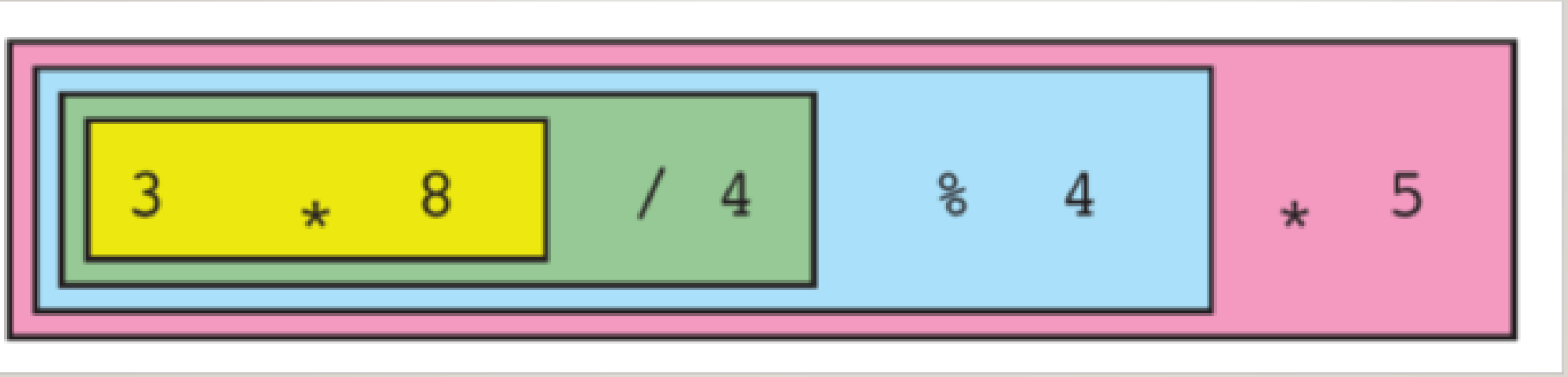
```
#include <stdio.h>

main() {

   int a = 20;
   int b = 10;
   int c = 15;
   int d = 5;
   int e;
    e = (a + b) * c / d;      // ( 30 * 15 ) / 5
   printf("Value of (a + b) * c / d is : %d\n",  e );
   e = ((a + b) * c) / d;    // (30 * 15 ) / 5
   printf("Value of ((a + b) * c) / d is  : %d\n" ,  e );
   e = (a + b) * (c / d);   // (30) * (15/5)
   printf("Value of (a + b) * (c / d) is  : %d\n",  e );
   e = a + (b * c) / d;     //  20 + (150/5)
   printf("Value of a + (b * c) / d is  : %d\n" ,  e );
     return 0;
}
```


Associativity is applied when we have more than one operator of the same precedence level in an expression.
ASSOCIATIVITY

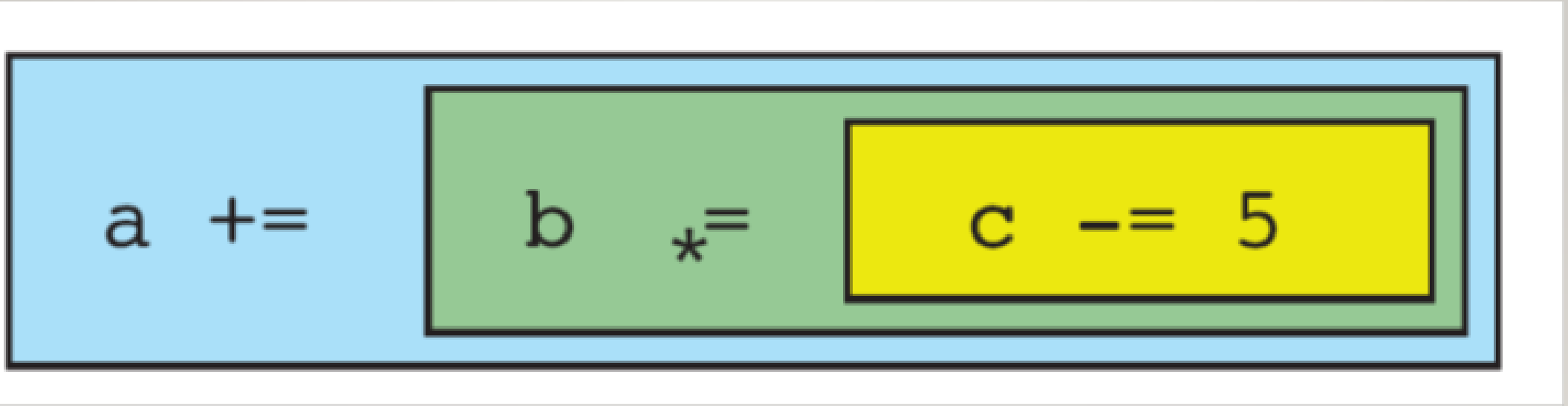
ASSOCIATIVITY

- Rules determining how to evaluate expressions containing more than one operator with the same precedence
  - example: is 5 - 4 - 3 ((5 - 4) - 3) or (5 - (4 - 3))
  - its important because the values may differ (e.g., -2 or 4 for the above possibilities)
- Left associativity: operators are evaluated left to right
- Right associativity: evaluated right to left

LEFT TO RIGHT ASSOCIATIVITY



RIGHT TO LEFT ASSOCIATIVITY



OPERATOR ASSOCIATIVITY

18: Identifier, Constant, Parenthesized Expression

17: Function call *(LEFT)*

16: Postfix increment/decrement *(LEFT)*

15: Prefix increment/decrement, sizeof, unary +, unary - *(RIGHT)*

14: (Type) - cast operator *(RIGHT)*

13: * / % *(LEFT)*

12: + - *(LEFT)*

2: = += -= *= /= %= *(RIGHT)*

USING OPERATOR ASSOCIATIVITY

- Evaluate operators by precedence
- When evaluating operators with the same precedence, consult associativity rules

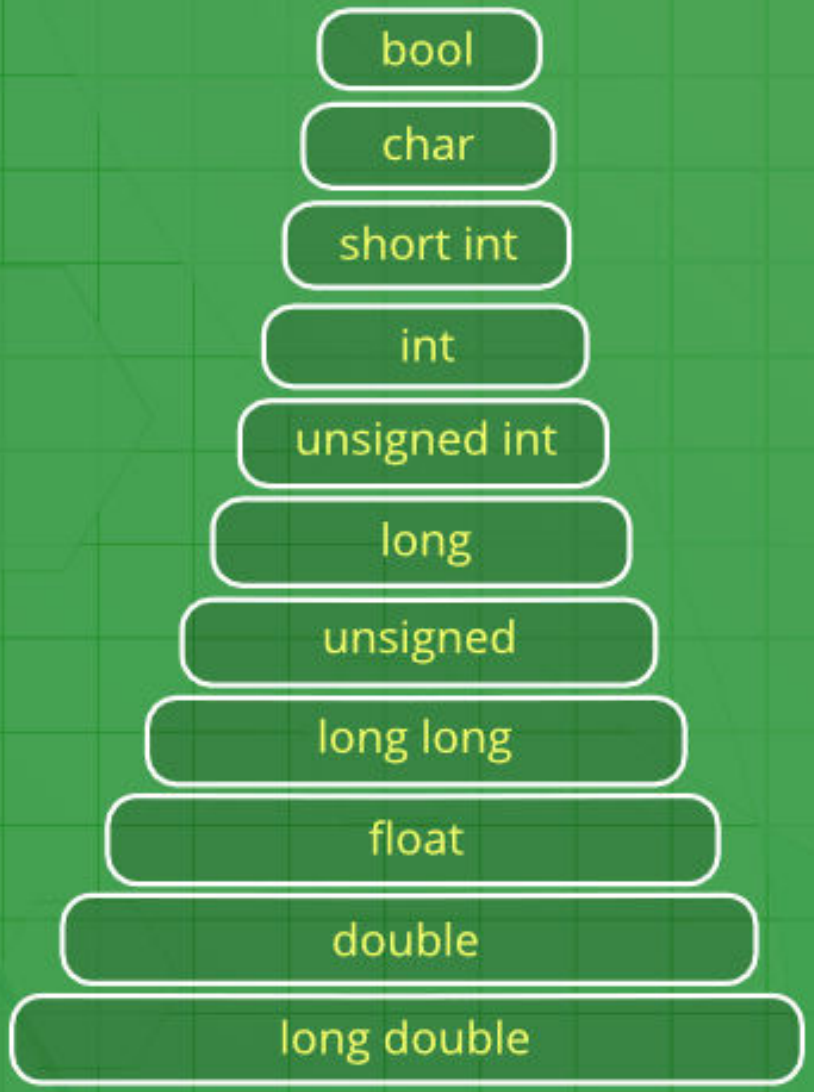
  example: 5 - 4 - 3, two - operators, so associativity (left) is used

  5 - 4 - 3 evaluates to ((5 - 4) - 3) or -2
- Again, use parentheses to make sure things are evaluated the way you expect

TYPE CONVERSION

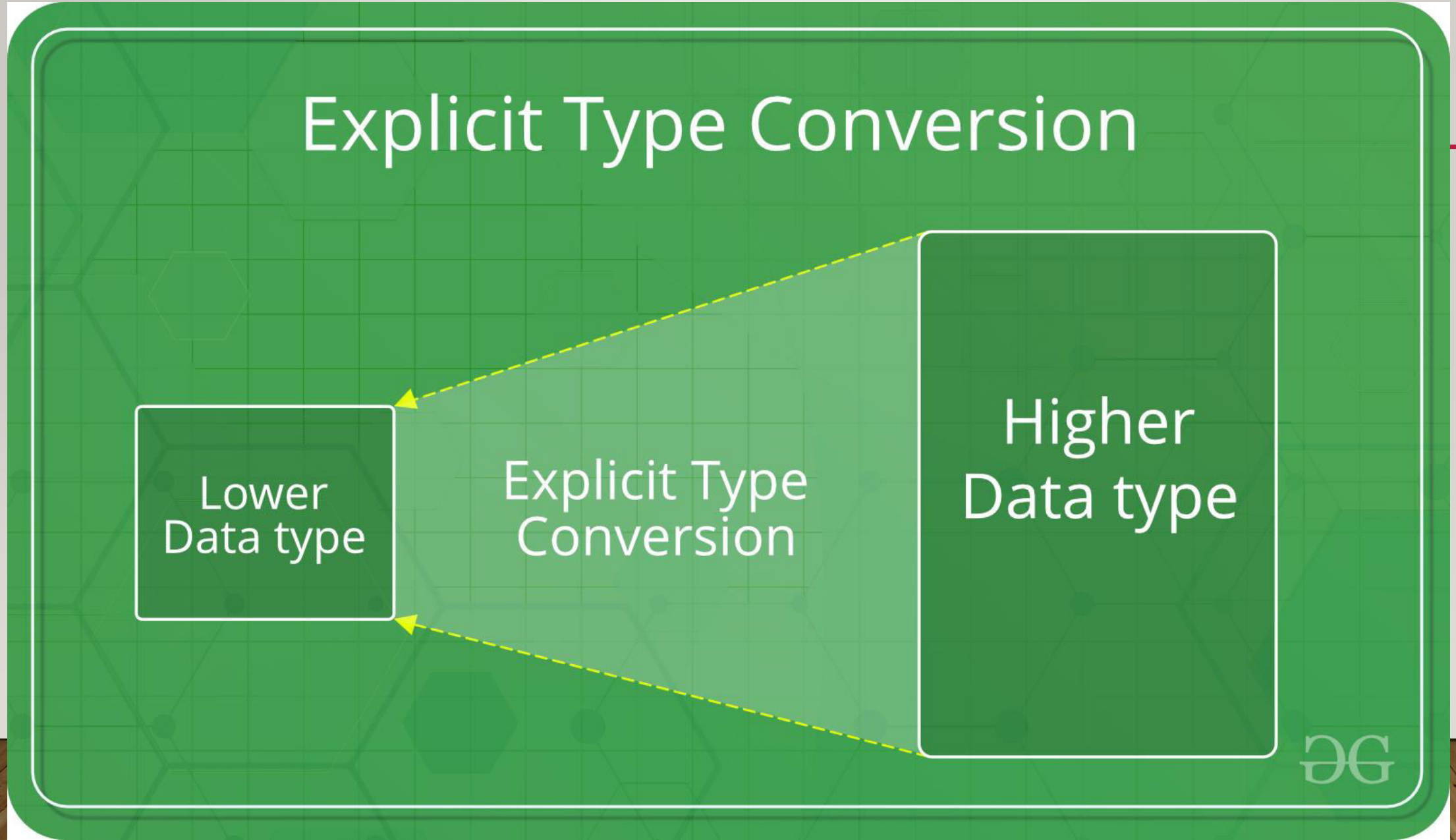
- Type conversion in C is the process of converting one data type to another.
- The type conversion is only performed to those data types where conversion is possible.
- Type conversion is performed by a compiler.
- In type conversion, the destination data type can't be smaller than the source data type.
- Type conversion is done at compile time

Implicit Type Conversion



```
#include <stdio.h>
int main()
{
    int x = 10; // integer x
    char y = 'a'; // character c
      // y implicitly converted to int. ASCII
    // value of 'a' is 97
    x = x + y;
      // x is implicitly converted to float
    float z = x + 1.0;
    printf("x = %d, z = %f", x, z);
    return 0;
}
```

(type) expression



```
#include<stdio.h>

int main()
{
    double x = 1.2;
        // Explicit conversion from double to int
    int sum = (int)x + 1;

    printf("sum = %d", sum);

    return 0;
}
```

sum = 2

SELECTION - MAKING DECISION

LOGICAL DATA AND OPERATORS

A piece of data is called logical if it conveys the idea of true or false. In real life, logical data (true or false) are created in answer to a question that needs a yes–no answer. In computer science, we do not use yes or no, we use true or false.

Topics discussed in this section:

Logical Data in C
Logical Operators
Evaluating Logical Expressions
Comparative Operators

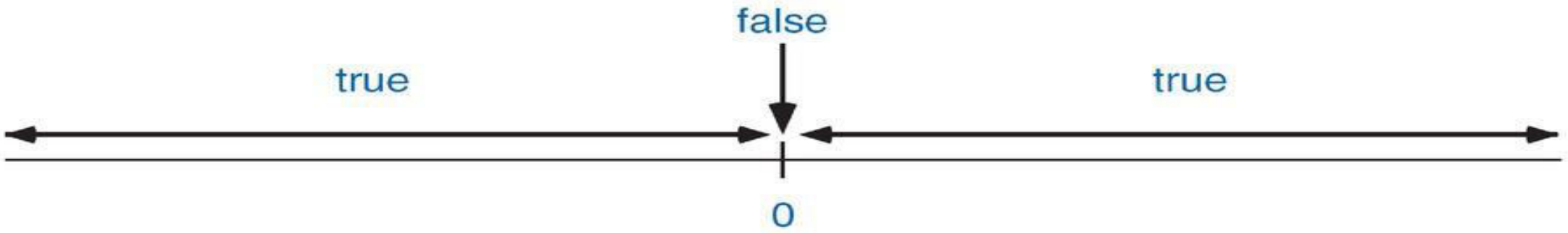


FIGURE 4-1 ***true*** **and** ***false*** **on the Arithmetic Scale**

not

x	!x
false	true
true	false

and

x	y	x&&y
false	false	false
false	true	false
true	false	false
true	true	true

or

x	y	x||y
false	false	false
false	true	true
true	false	true
true	true	true

FIGURE 4-2 Logical Operators Truth Table

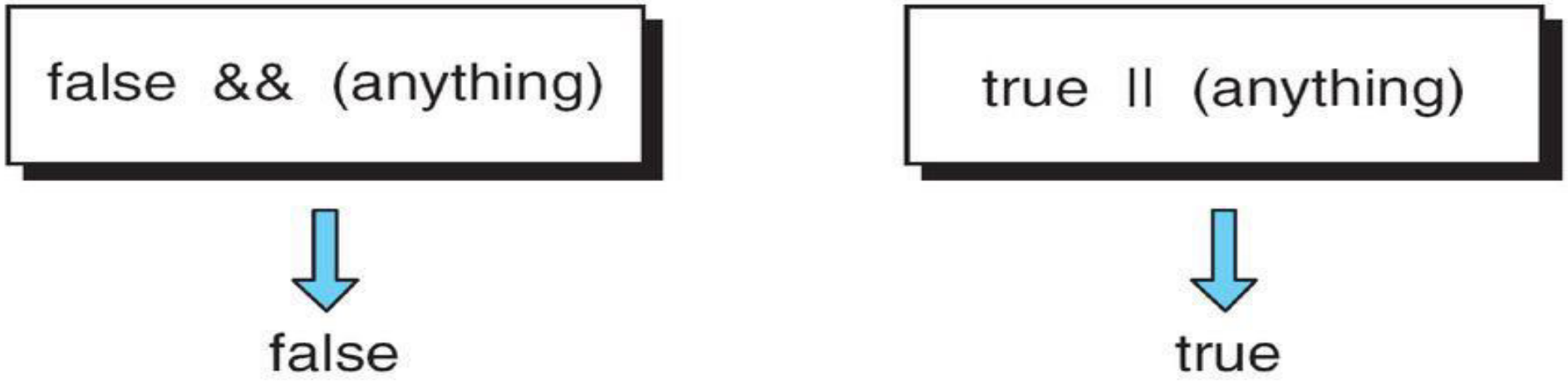


FIGURE 4-3 **Short-circuit Methods for *and* /*or***

PROGRAM 4-1 Logical Expressions

```
 1  /* Demonstrate the results of logical operators.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6  #include <stdbool.h>
 7
 8  int main (void)
 9  {
10  // Local Declarations
11     bool a = true;
12     bool b = true;
13     bool c = false;
14
15  // Statements
16     printf("    %2d AND     %2d: %2d\n", a, b,  a &&  b);
17     printf("    %2d AND     %2d: %2d\n", a, c,  a &&  c);
18     printf("    %2d AND     %2d: %2d\n", c, a,  c &&  a);
```

PROGRAM 4-1 Logical Expressions

```
19      printf("     %2d OR       %2d: %2d\n", a, c,  a ||  c);
20      printf("     %2d OR       %2d: %2d\n", c, a,  c ||  a);
21      printf("     %2d OR       %2d: %2d\n", c, c,  c ||  c);
22      printf("NOT %2d AND NOT %2d: %2d\n", a, c, !a && !c);
23      printf("NOT %2d AND     %2d: %2d\n", a, c, !a &&  c);
24      printf("     %2d AND NOT %2d: %2d\n", a, c,  a && !c);
25      return 0;
26  }   // main
```

```
Results:
      1 AND       1:  1
      1 AND       0:  0
      0 AND       1:  0
      1 OR        0:  1
      0 OR        1:  1
      0 OR        0:  0
NOT   1 AND NOT   0:  0
NOT   1 AND       0:  0
      1 AND NOT   0:  1
```

Type	Operator	Meaning	Precedence
Relational	<	less than	10
	<=	less than or equal	
	>	greater than	
	>=	greater than or equal	
Equality	==	equal	9
	!=	not equal	

FIGURE 4-4 Relational Operators

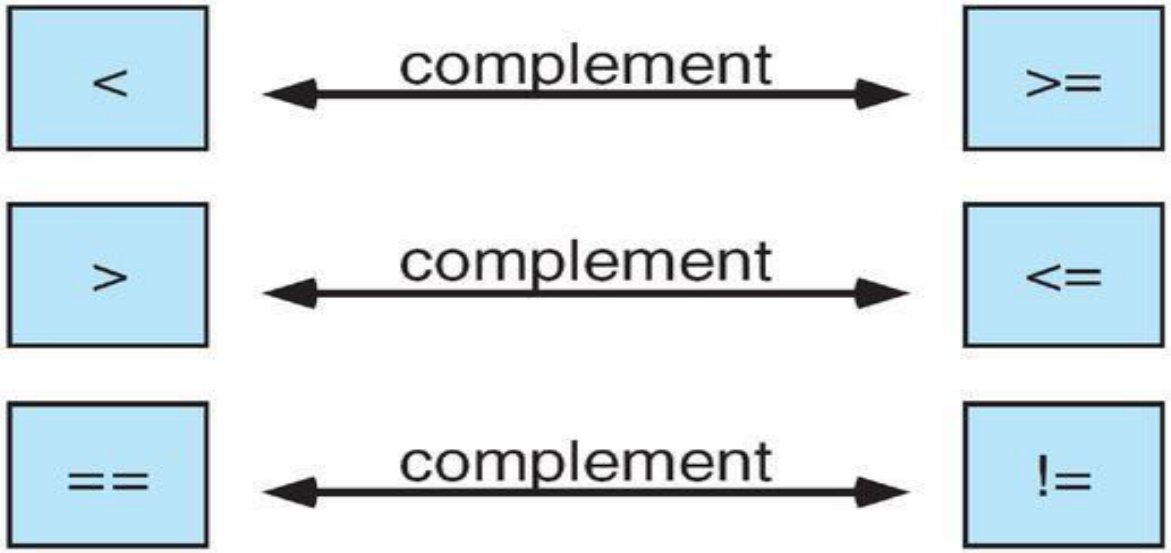


FIGURE 4-5 Comparative Operator Complements

Original Expression	Simplified Expression
`!(x < y)`	`x >= y`
`!(x > y)`	`x <= y`
`!(x != y)`	`x == y`
`!(x <= y)`	`x > y`
`!(x >= y)`	`x < y`
`!(x == y)`	`x != y`

Table 4-1 Examples of Simplifying Operator Complements

PROGRAM 4-2 Comparative Operators

```
 1  /* Demonstrates the results of relational operators.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9  // Local Declarations
10     int a = 5;
11     int b = -3;
12
13  // Statements *
14     printf(" %2d <  %2d is %2d\n", a, b,  a <  b);
15     printf(" %2d == %2d is %2d\n", a, b,  a == b);
16     printf(" %2d != %2d is %2d\n", a, b,  a != b);
17     printf(" %2d >  %2d is %2d\n", a, b,  a >  b);
```

PROGRAM 4-2 Comparative Operators

```
18      printf(" %2d <= %2d is %2d\n", a, b,  a <= b);
19      printf(" %2d >= %2d is %2d\n", a, b,  a >= b);
20      return 0;
21  }   // main
```

```
Results:
5 <  -3 is  0
5 == -3 is  0
5 != -3 is  1
5 >  -3 is  1
5 <= -3 is  0
5 >= -3 is  1
```

4-2 Two-Way Selection

The decision is described to the computer as a conditional statement that can be answered either true or false. If the answer is true, one or more action statements are executed. If the answer is false, then a different action or set of actions is executed.

Topics discussed in this section:

***if*...*else* and Null *else* Statement**
Nested *if* Statements and Dangling *else* Problem
Simplifying *if* Statements
Conditional Expressions

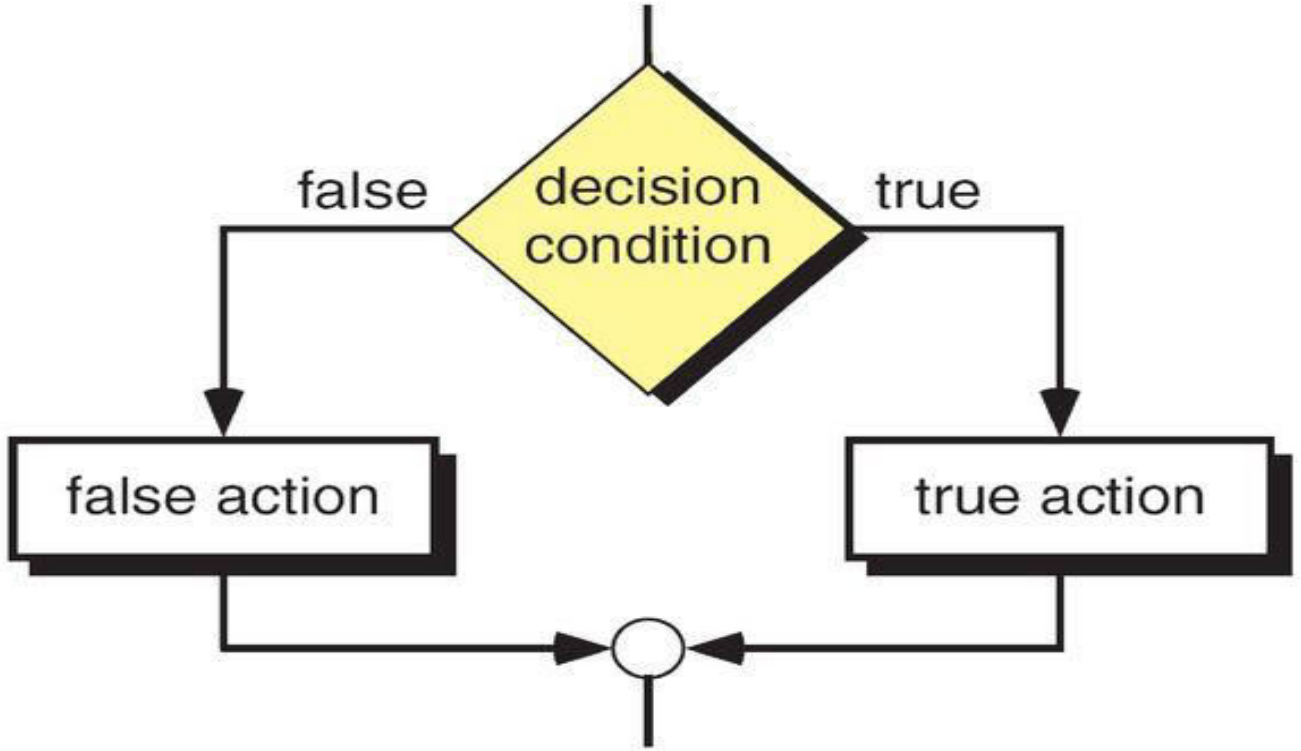
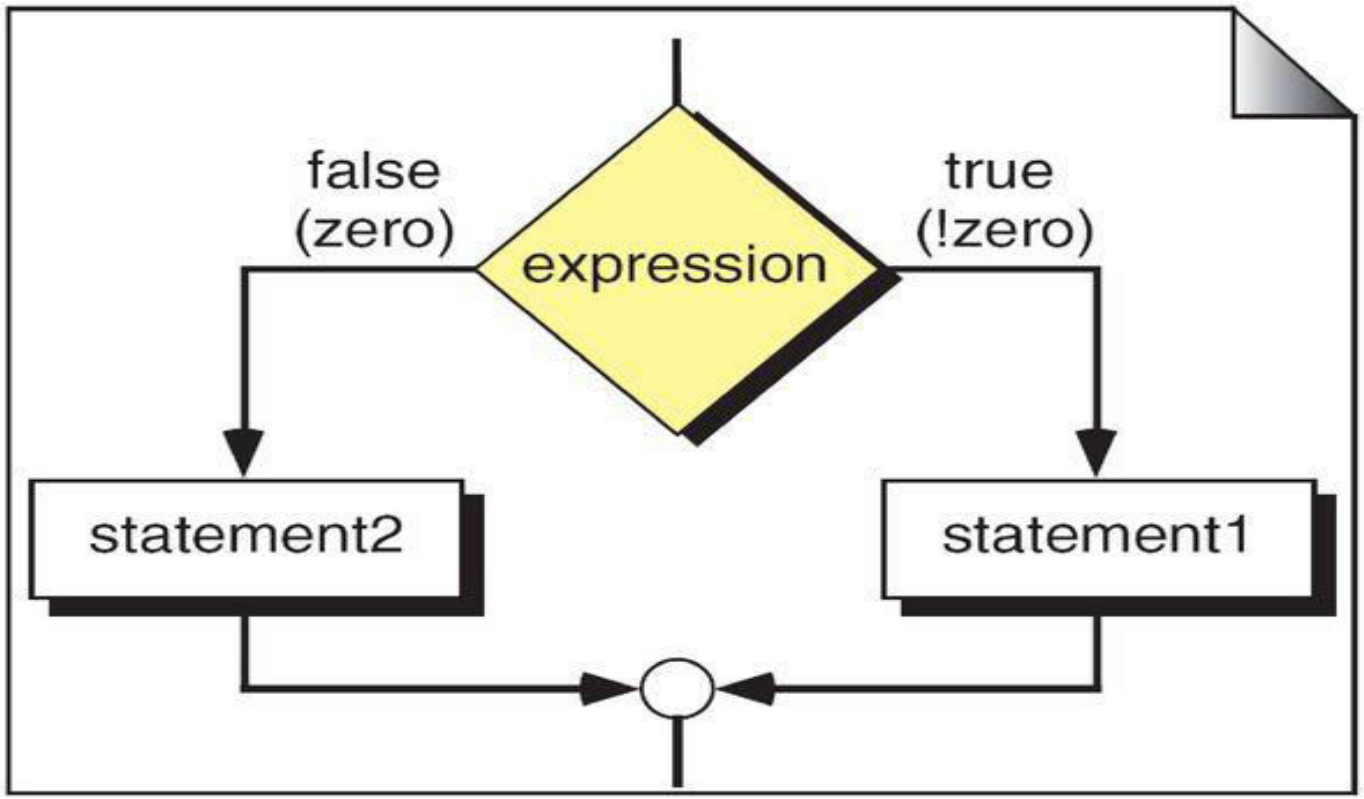


FIGURE 4-6 Two-way Decision Logic



(a) Logical Flow

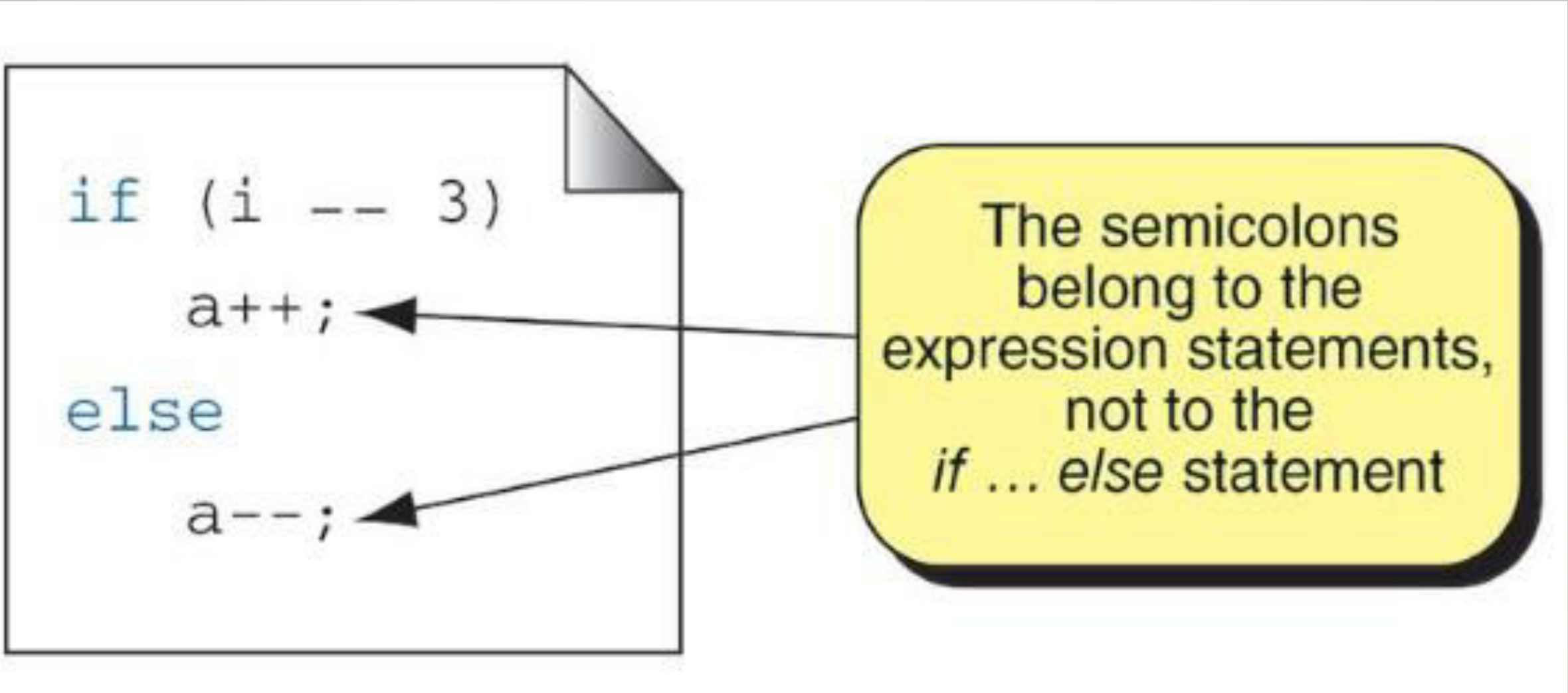
```
if (expression)
     statement1
else
     statement2
```

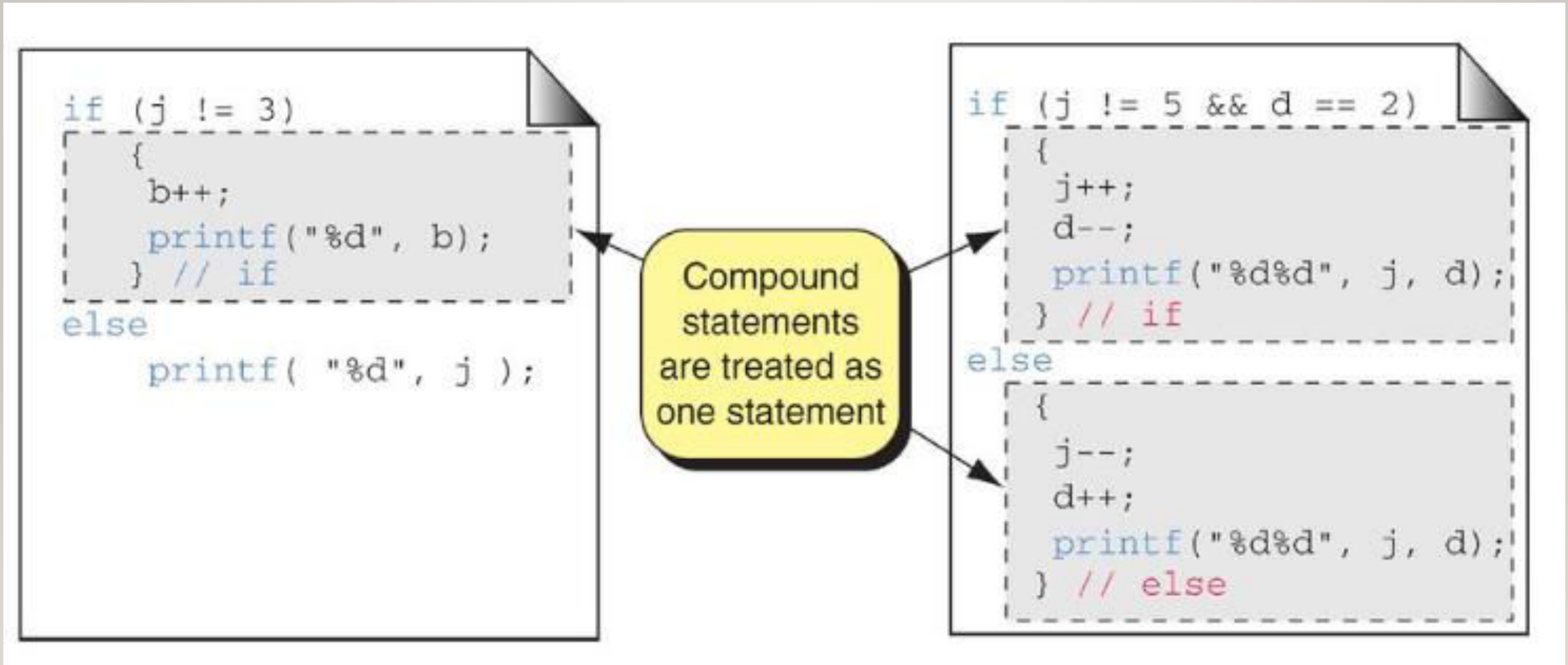
(b) Code

FIGURE 4-7 ***if...else*** **Logic Flow**

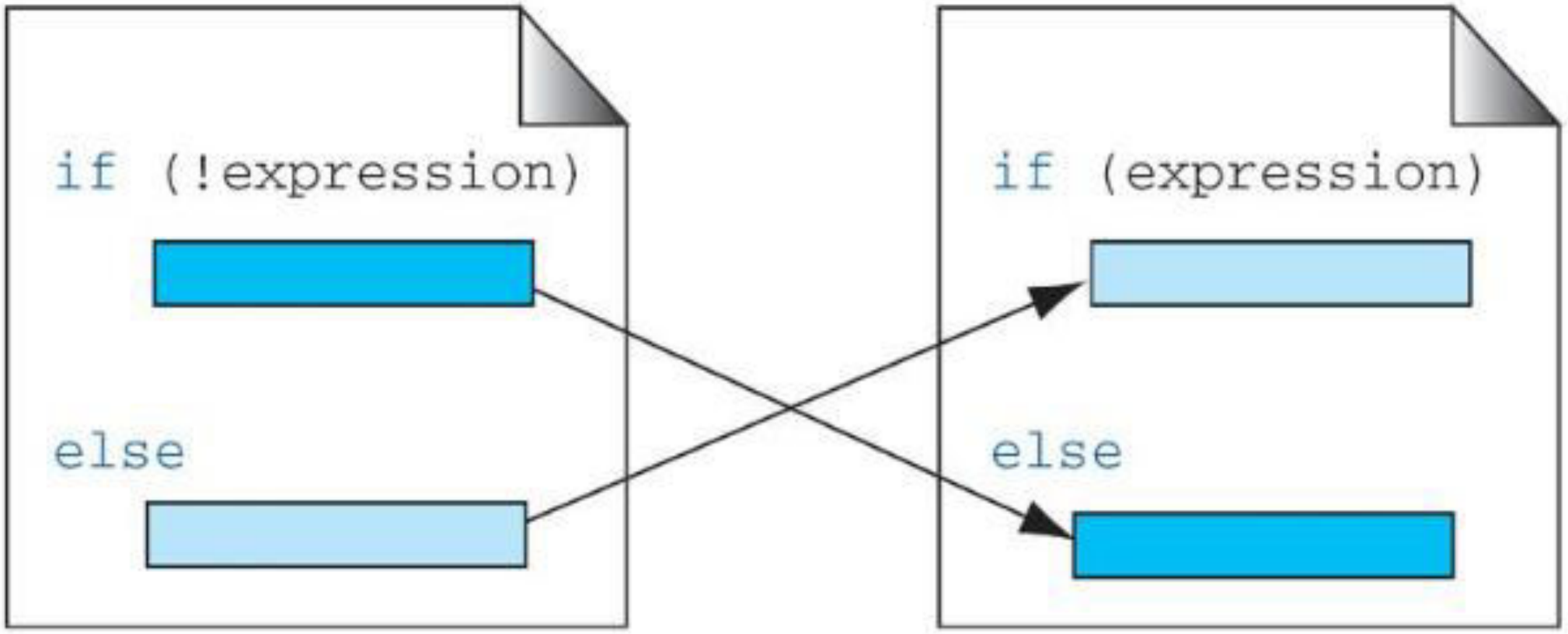
1. The expression must be enclosed in parentheses.
2. No semicolon (;) is needed for an *if...else* statement; statement 1 and statement 2 may have a semicolon as required by their types.
3. The expression can have a side effect.
4. Both the true and the false statements can be any statement (even another *if...else* statement) or they can be a null statement.
5. Both statement 1 and statement 2 must be one and only one statement. Remember, however, that multiple statements can be combined into a compound statement through the use of braces.
6. We can swap the position of statement 1 and statement 2 if we use the complement of the original expression.

Table 4-2 Syntactical Rules for *if...else* Statements


if (i -- 3)
a++;
else
a--;
The semicolons belong to the expression statements, not to the *if … else* statement


if (j != 3)
{
b++;
printf("%d", b);
} // if
else
printf("%d", j);
Compound statements are treated as one statement
if (j != 5 && d == 2)
{
j++;
d--;
printf("%d%d", j, d);
} // if
else
{
j--;
d++;
printf("%d%d", j, d);
} // else

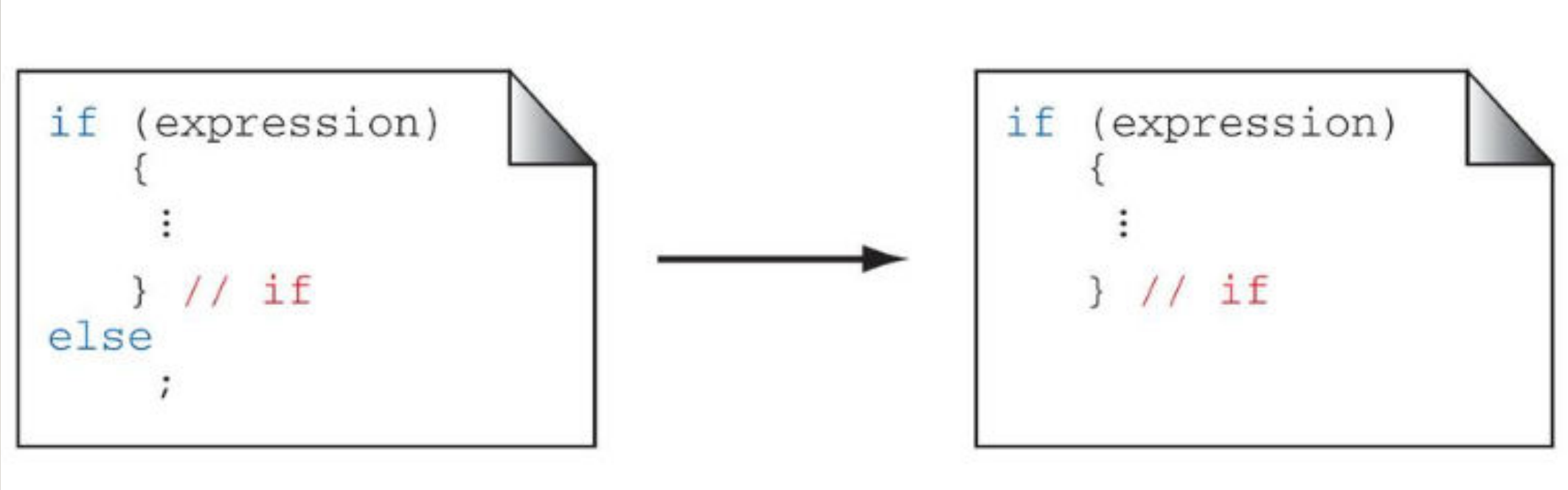
These two statements are the same because the expressions are the complements of each other!



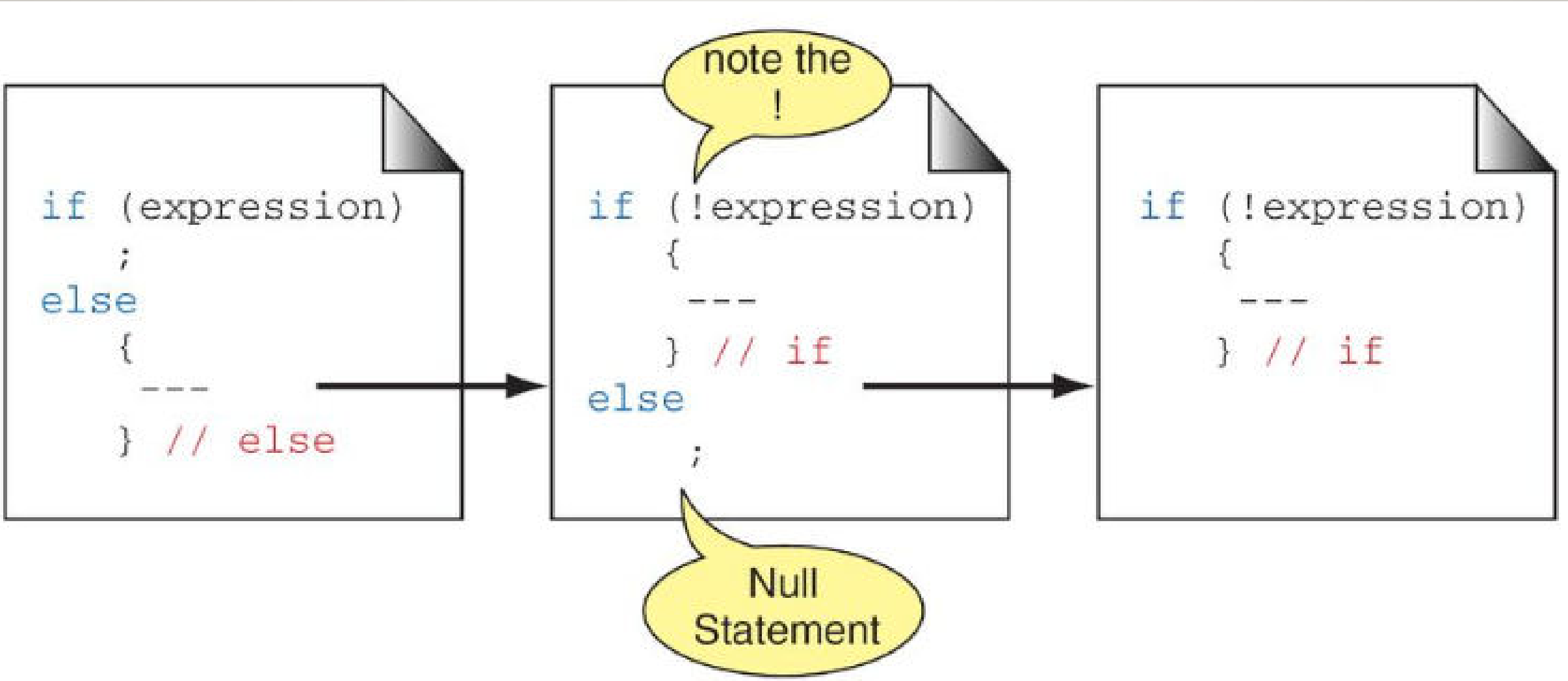
(a) Original

(b) Complemented

NULL ELSE STATEMENT



NULL IF STATEMENT



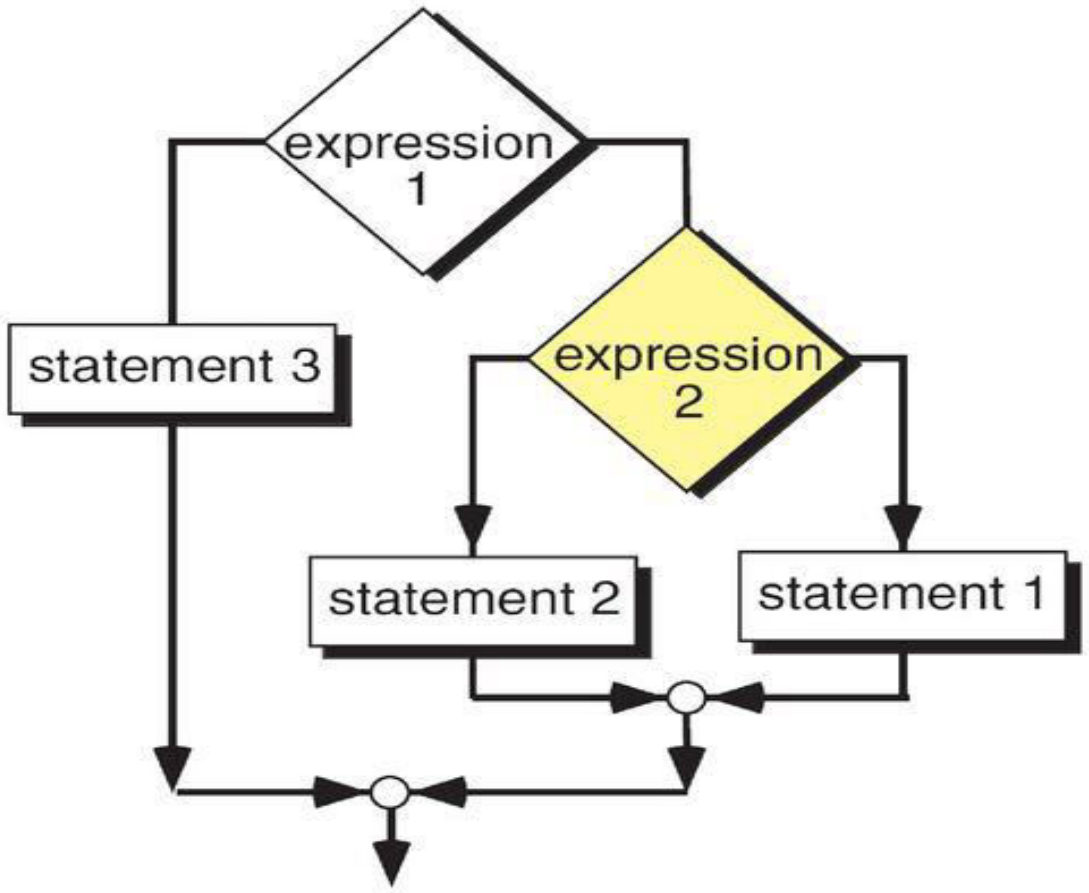
PROGRAM 4-3 Two-way Selection

```
 1  /* Two-way selection.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9     // Local Declarations
10     int a;
11     int b;
12
13     // Statements
14     printf("Please enter two integers: ");
15     scanf ("%d%d", &a, &b);
16
```

PROGRAM 4-3 Two-way Selection

```
17      if (a <= b)
18          printf("%d <= %d\n", a, b);
19      else
20          printf("%d > %d\n", a, b);
21
22      return 0;
23  }   // main
```

```
Results:
Please enter two integers: 10 15
10 <= 15
```



(a) Logic flow

```
if (expression 1)
    if (expression 2)
        statement 1
    else
        statement 2
else
    statement 3
```

(b) Code

FIGURE 4-13 Nested *if* Statements

PROGRAM 4-4 Nested *if* Statements

```
 1  /* Nested if in two-way selection.
 2        Written by:
 3        Date:
 4  */
 5  #include <stdio.h>
 6
 7  int main (void)
 8  {
 9  // Local Declarations
10     int a;
11     int b;
12
13  // Statements
14     printf("Please enter two integers: ");
15     scanf ("%d%d", &a, &b);
16
```

PROGRAM 4-4 Nested *if* Statements

```
17      if (a <= b)
18          if (a < b)
19              printf("%d < %d\n", a, b);
20          else
21              printf("%d == %d\n", a, b);
22      else
23          printf("%d > %d\n", a, b);
24
25      return 0;
26  }   // main
```

```
Results:
Please enter two integers: 10 10
10 == 10
```

Note

***else* is always paired with the most recent unpaired *if*.**

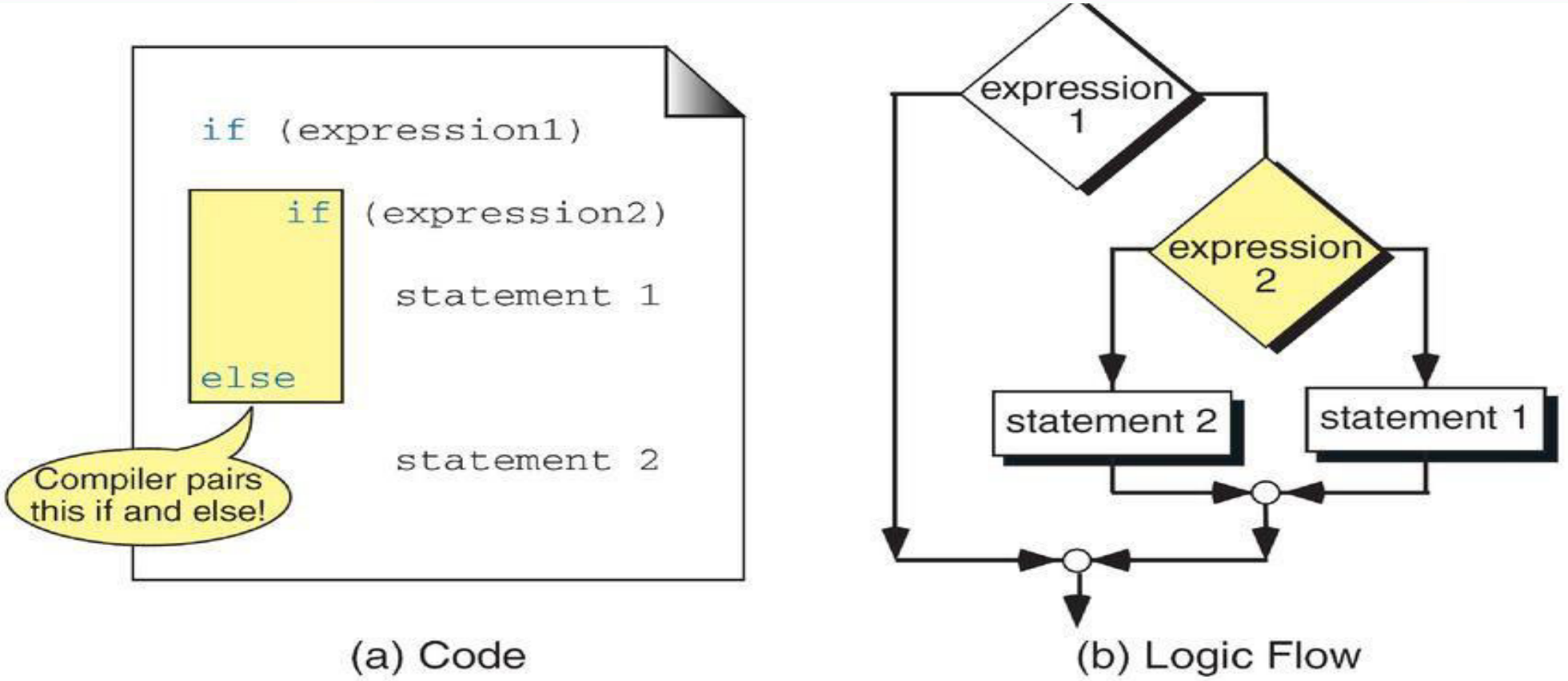
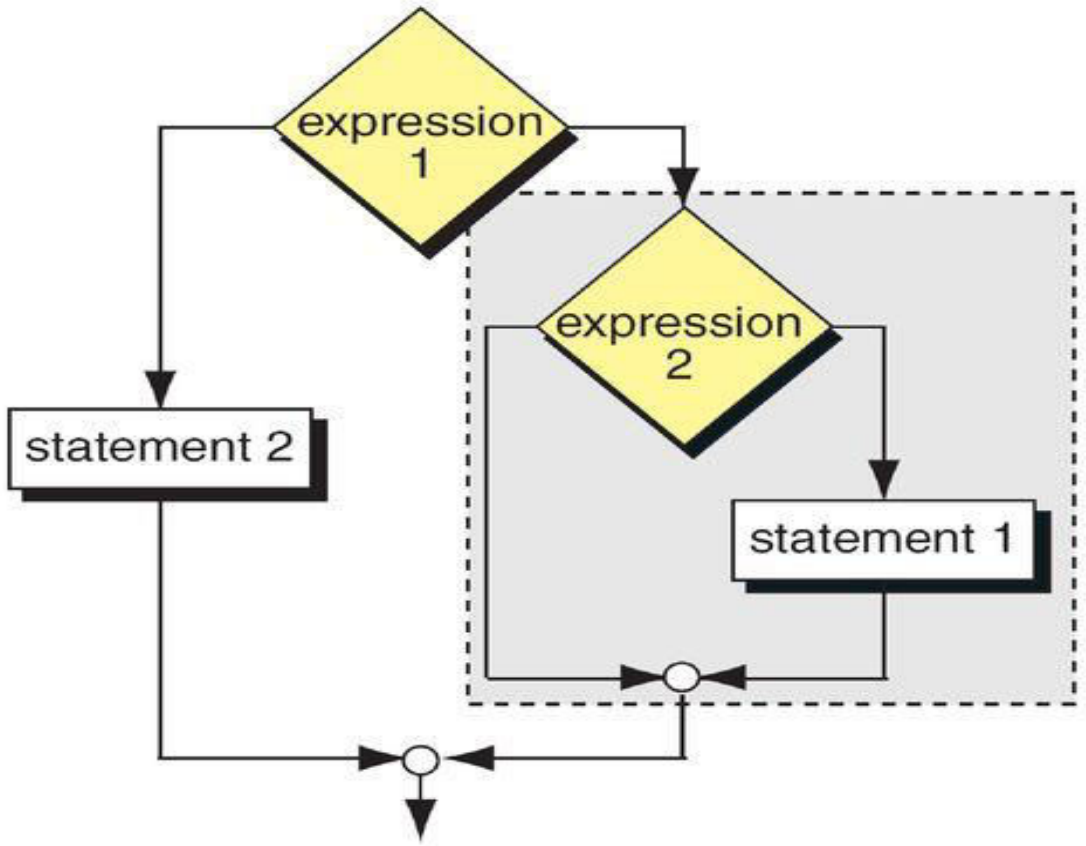


FIGURE 4-14 Dangling *else*



(a) Logic Flow

```
if (expression 1)
    {
      if (expression 2)
           statement 1
    } // if
else
        statement 2
```

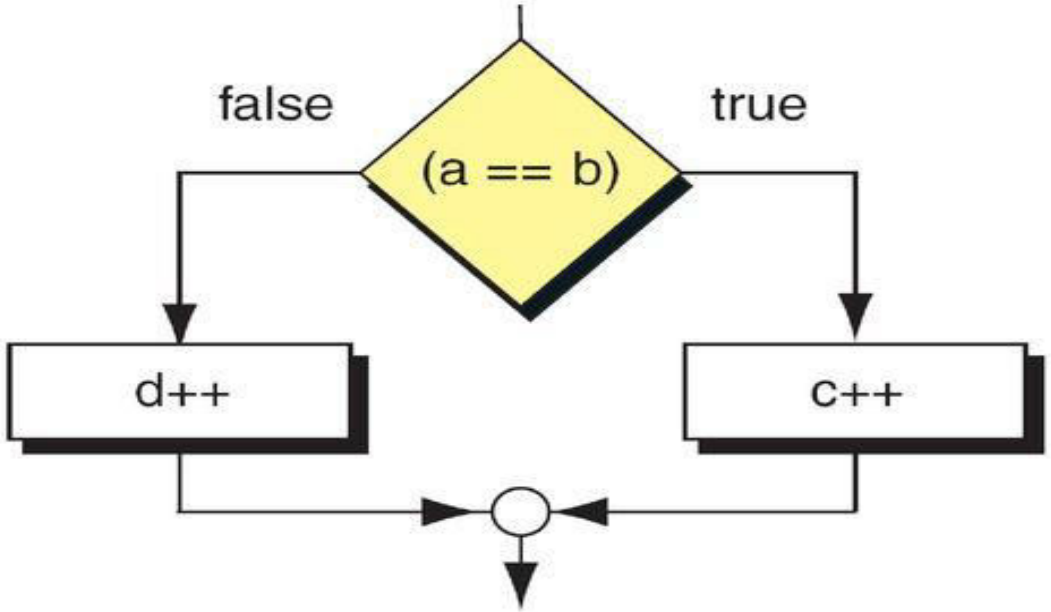
The block closes the if statement

(b) Code

FIGURE 4-15 Dangling *else* Solution

Original Statement	Simplified Statement
`if (a != 0)` `  statement`	`if (a)` `  statement`
`if (a == 0)` `  statement`	`if (!a)` `  statement`

Table 4-3 Simplifying the Condition



(a) Logic Flow

```
a == b ? c++ : d++;
```

(b) Code

FIGURE 4-16 Conditional Expression

Case 1: Total Income 23,000			Case 2: Total Income 18,000		
Income in Bracket	Tax Rate	Tax	Income in Bracket	Tax Rate	Tax
(1) 10,000	2%	200	(1) 10,000	2%	200
(2) 10,000	5%	500	(2) 8,000	5%	400
(3) 3,000	7%	210	(3) none	7%	0
Total Tax		910	Total Tax		600

Table 4-4 **Examples of Marginal Tax Rates**

4-3 Multiway Selection

In addition to two-way selection, most programming languages provide another selection concept known as multiway selection. Multiway selection chooses among several alternatives. C has two different ways to implement multiway selection: the switch statement and else-if construct.

Topics discussed in this section:

The *switch* Statement
The *else-if*

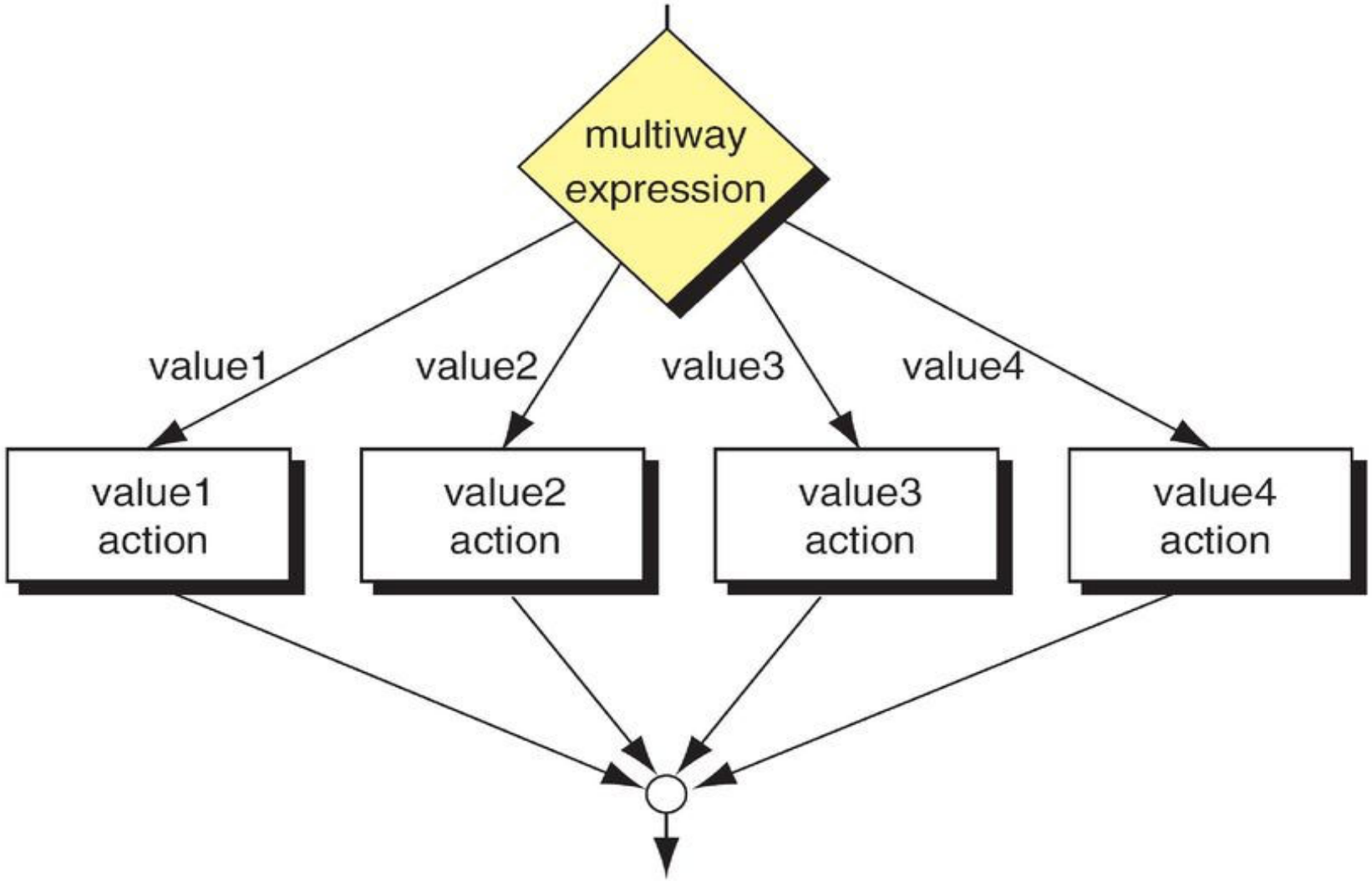


FIGURE 4-19 ***switch*** **Decision Logic**

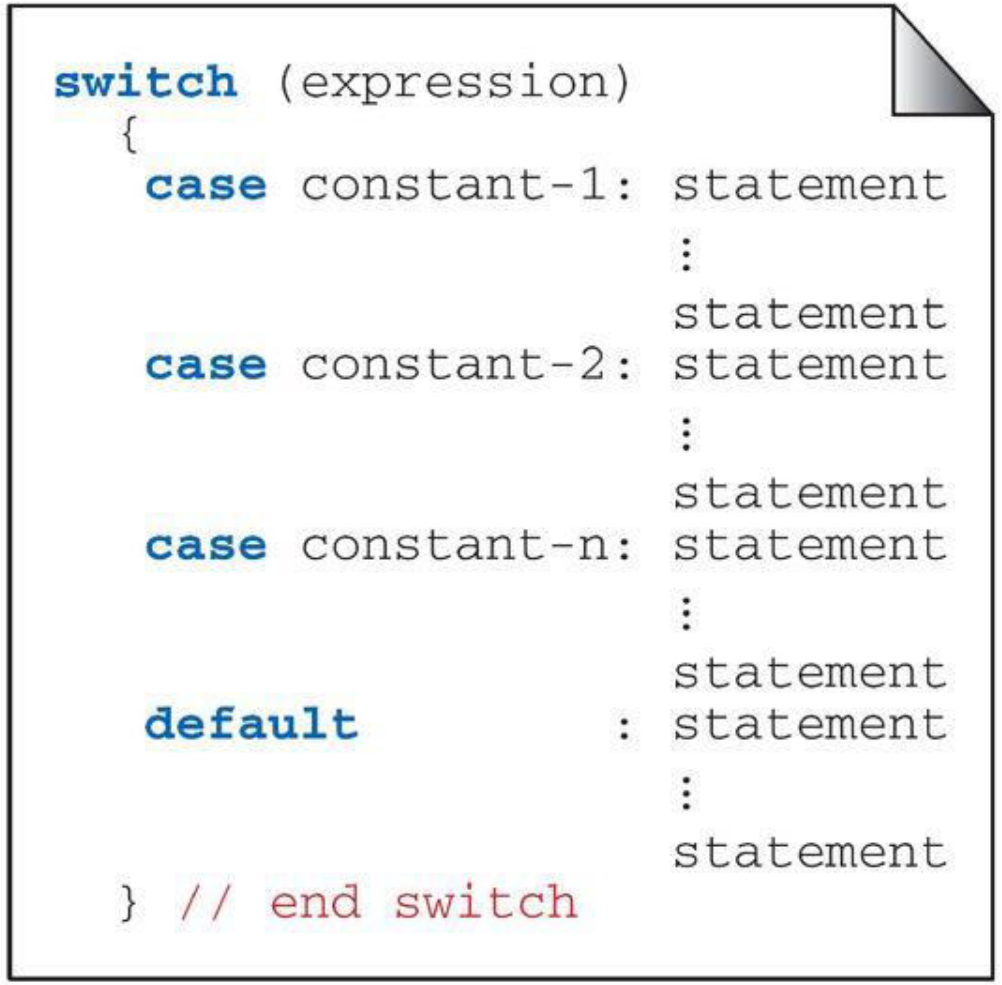


FIGURE 4-20 ***switch*** **Statement Syntax**

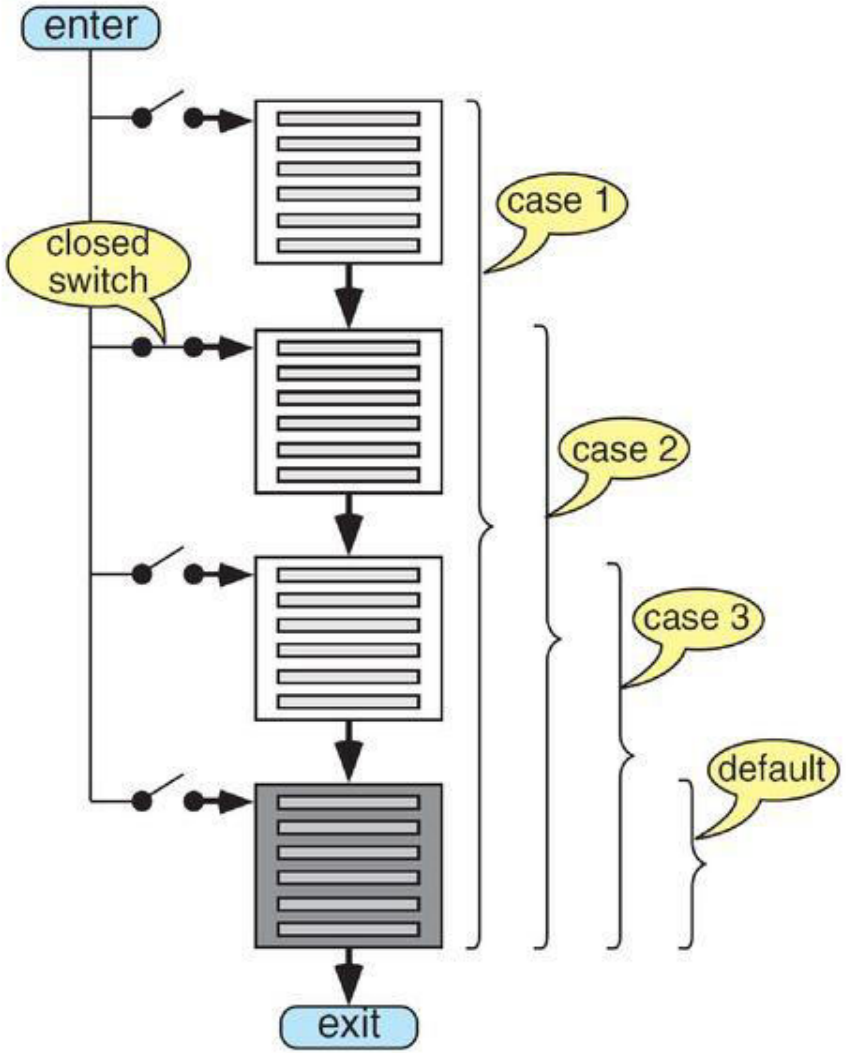
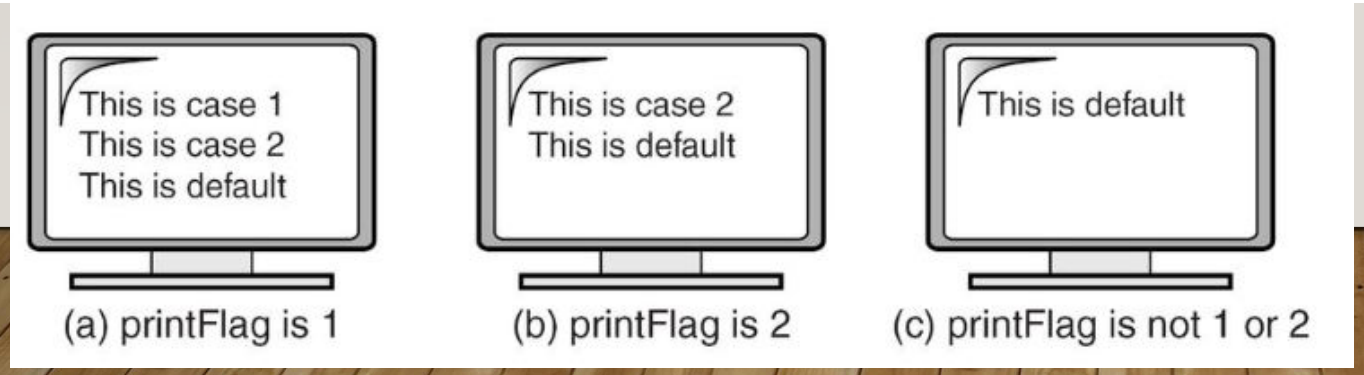
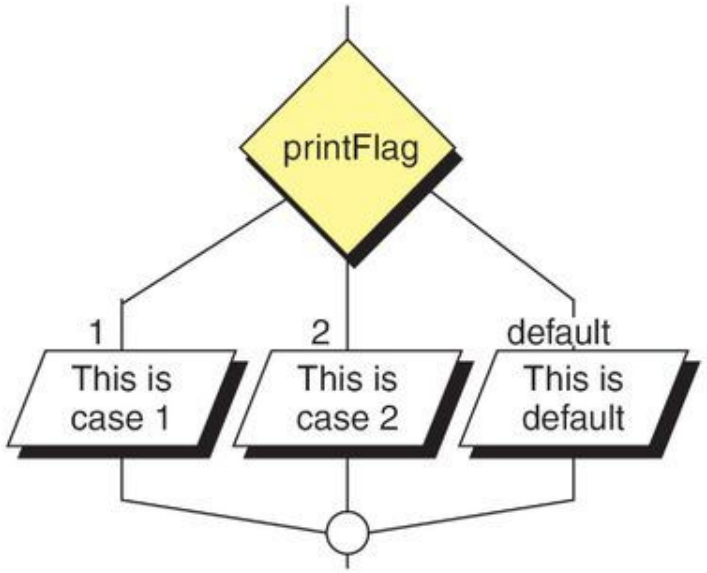


FIGURE 4-21 ***switch*** **Flow**

PROGRAM 4-6 Demonstrate the *switch* Statement

```
1  // Program fragment to demonstrate switch
2     switch (printFlag)
3        {
4         case 1:  printf("This is case 1\n");
5
6         case 2:  printf("This is case 2\n");
7
8         default: printf("This is default\n");
9        } // switch
```





(a) Logic Flow

```
switch (printFlag)
  {
   case 1:
            printf
              ("This is case 1");
            break;
   case 2:
            printf
              ("This is case 2");
            break;
   default:
            printf
              ("This is default");
            break;
  }  // switch
```

(b) Code

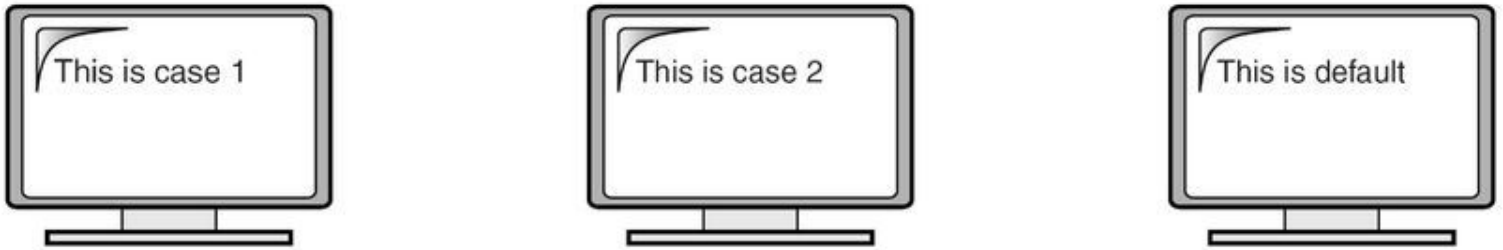


FIGURE 4-23 A *switch* with *break* Statements

PROGRAM 4-7 Multivalued *case* Statements

```
1  /* Program fragment that demonstrates multiple
2     cases for one set of statements
3  */
4  switch (printFlag)
5     {
6      case 1:
7      case 3:  printf("Good Day\n");
8               printf("Odds have it!\n");
9               break;
10     case 2:
11     case 4:  printf("Good Day\n");
12              printf("Evens have it!\n");
13              break;
14     default: printf("Good Day, I'm confused!\n");
15              printf("Bye!\n");
16              break;
17    } // switch
```

1. The control expression that follows the keyword *switch* must be an integral type.
2. Each *case* label is the keyword *case* followed by a constant expression.
3. No two *case* labels can have the same constant expression value.
4. But two *case* labels can be associated with the same set of actions.
5. The *default* label is not required. If the value of the expression does not match with any labeled constant expression, the control transfers outside of the *switch* statement. However, we recommend that all *switch* statements have a *default* label.
6. The *switch* statement can include at most one *default* label. The *default* label may be coded anywhere, but it is traditionally coded last.

Table 4-5 Summary of *switch* Statement Rules

PROGRAM 4-8 Student Grading

```
39      switch (temp)
40          {
41           case 10:
42           case 9 : grade = 'A';
43                    break;
44           case 8 : grade = 'B';
45                    break;
46           case 7 : grade = 'C';
47                    break;
48           case 6 : grade = 'D';
49                    break;
50           default: grade = 'F';
51          } // switch
52      return grade;
53  }  // scoreToGrade
```

```
Results:
Enter the test score (0-100): 89
The grade is: B
```

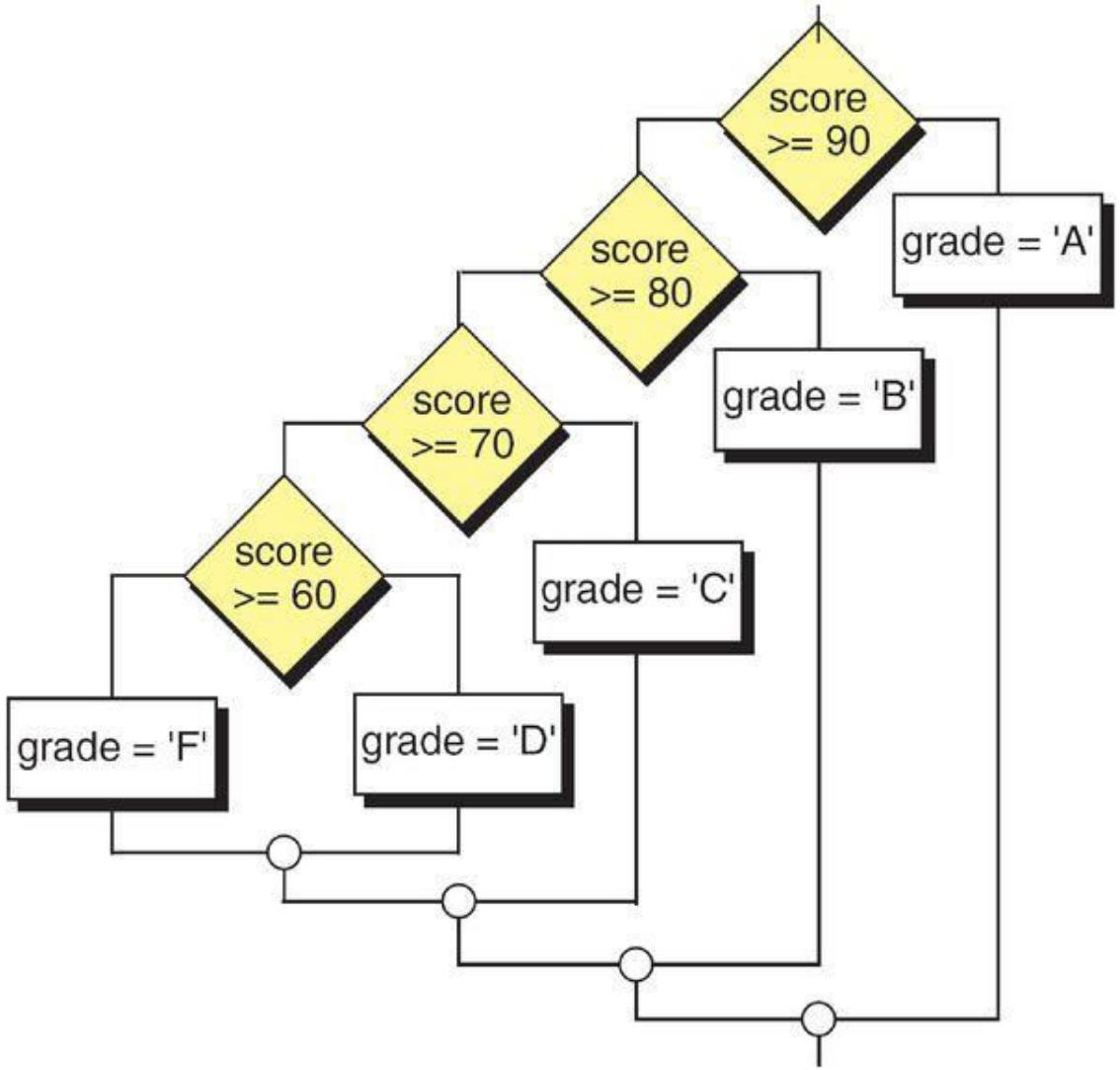


FIGURE 4-24 The *else-if* Logic Design for Program 5-9

Note

The *else-if* is an artificial C construct that is only used when
1. The selection variable is not an integral, and
2. The same variable is being tested in the expressions.

```
if (score >= 90)
    grade = 'A';
else if (score >= 80)
    grade = 'B';
else if (score >= 70)
    grade = 'C';
else if (score >= 60)
    grade = 'D';
else
    grade = 'F';
return grade;
```

```
int day = 4;

switch (day) {
  case 1:
    printf("Monday");
    break;
  case 2:
    printf("Tuesday");
    break;
  case 3:
    printf("Wednesday");
    break;
  case 4:
    printf("Thursday");
    break;
  case 5:
    printf("Friday");
    break;
  case 6:
    printf("Saturday");
    break;
  case 7:
    printf("Sunday");
    break;
}
```

```c
int main()
{
    char x = 'A';
    switch (x) {
    case 'A':
        printf("Choice is A");
        break;
    case 'B':
        printf("Choice is B");
        break;
    case 'C':
        printf("Choice is C");
        break;
    default:
        printf("Choice other than A, B and C");
        break;
    }
    return 0;
}
```

```c
#include <stdio.h>
int main() {
    int number1, number2;
    printf("Enter two integers: ");
    scanf("%d %d", &number1, &number2);

    //checks if the two integers are equal.
    if(number1 == number2) {
        printf("Result: %d = %d",number1,number2);
    }

    //checks if number1 is greater than number2.
    else if (number1 > number2) {
        printf("Result: %d > %d", number1, number2);
    }

    //checks if both test expressions are false
    else {
        printf("Result: %d < %d",number1, number2);
    }

    return 0;
}
```

CONCEPT OF LOOP

The real power of computers is in their ability to repeat an operation or a series of operations many times. This repetition, called looping, is one of the basic structured programming concepts.

Each loop must have an expression that determines if the loop is done. If it is not done, the loop repeats one more time; if it is done, the loop terminates.

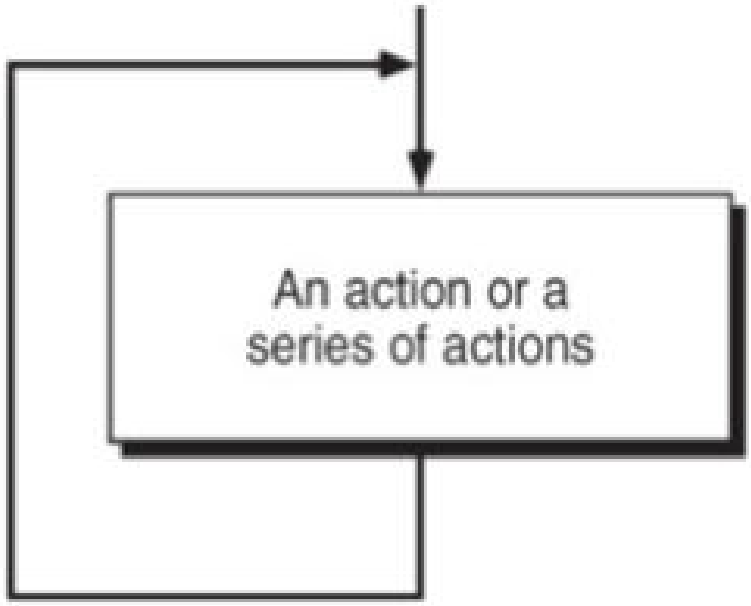
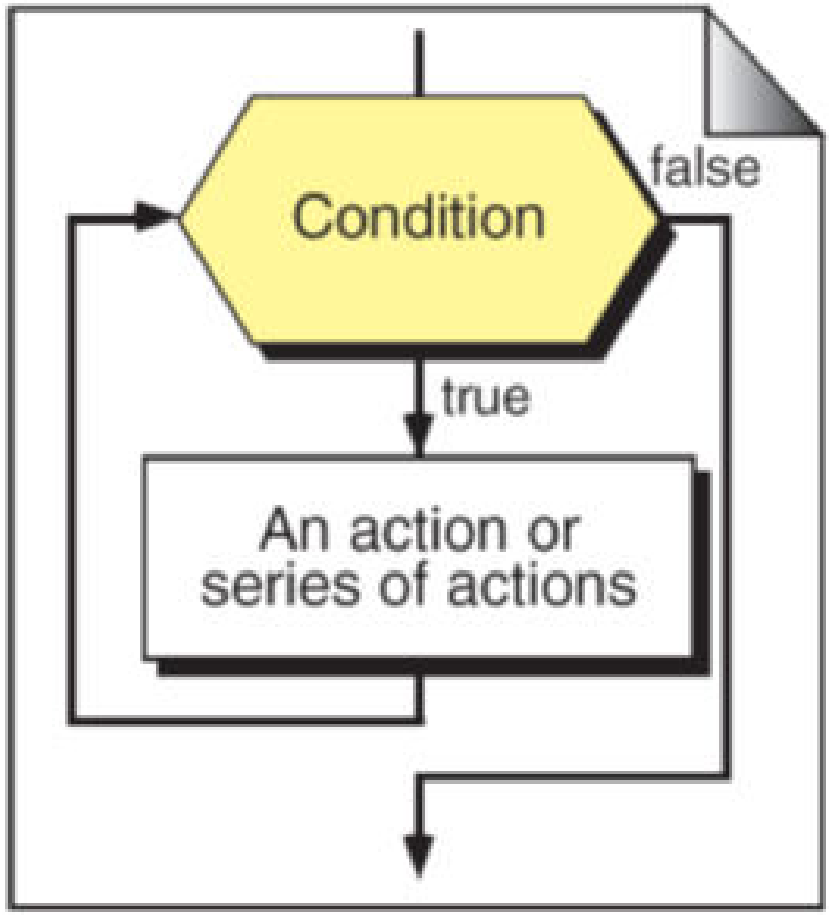


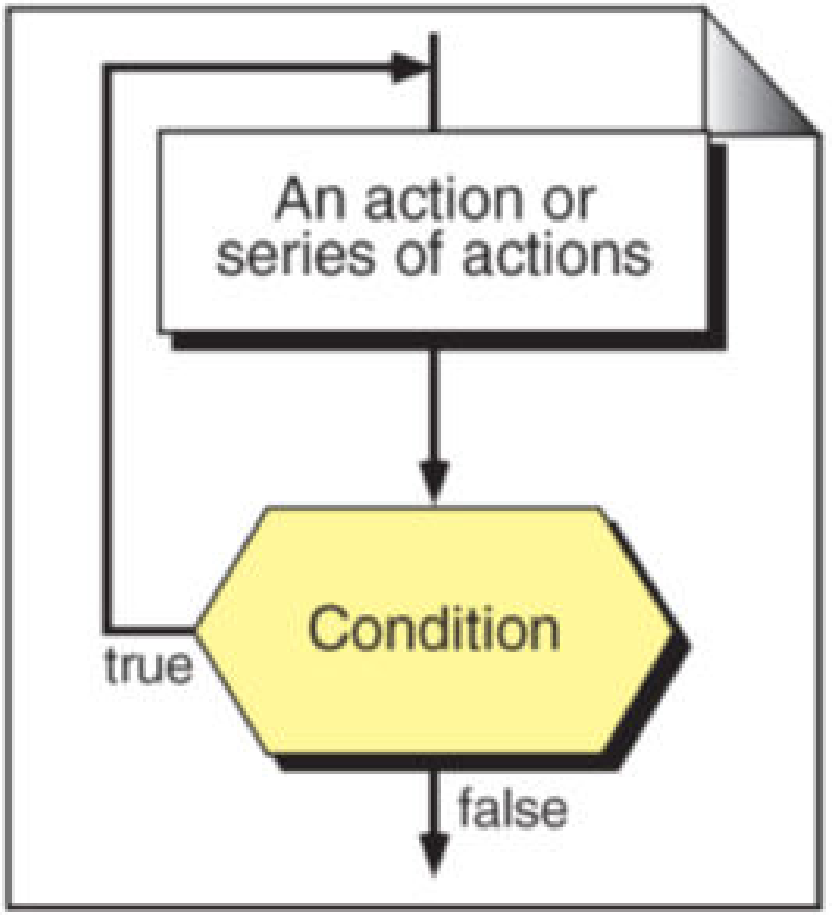
FIGURE 6-1
Concept of a Loop

PRETEST AND POST-TEST LOOPS

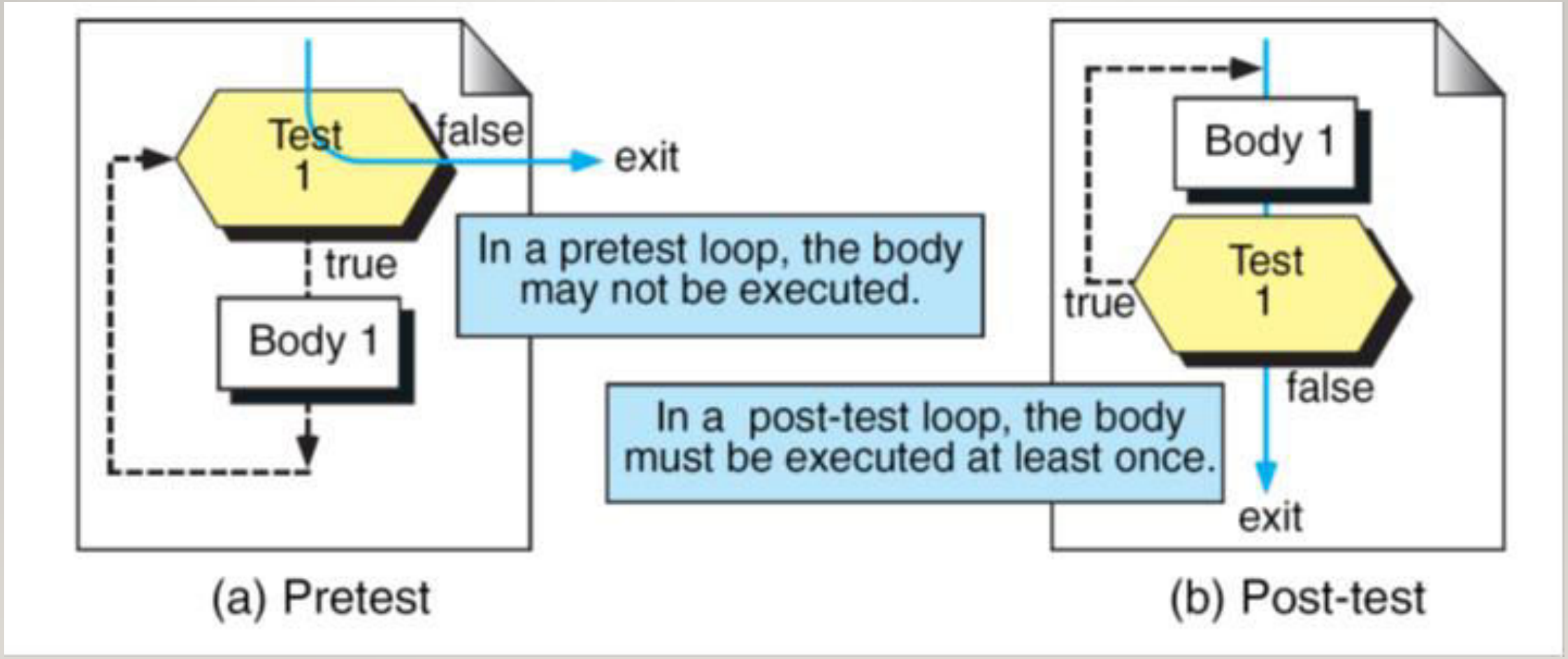
- We need to test for the end of a loop, but where should we check it
  - before or after each iteration
  - We can have either a pre- or a post-test terminating condition.
- In a pretest loop, the condition is checked at the beginning of each iteration.
- In a post-test loop , the condition is checked at the end of each iteration.



(a) Pretest Loop

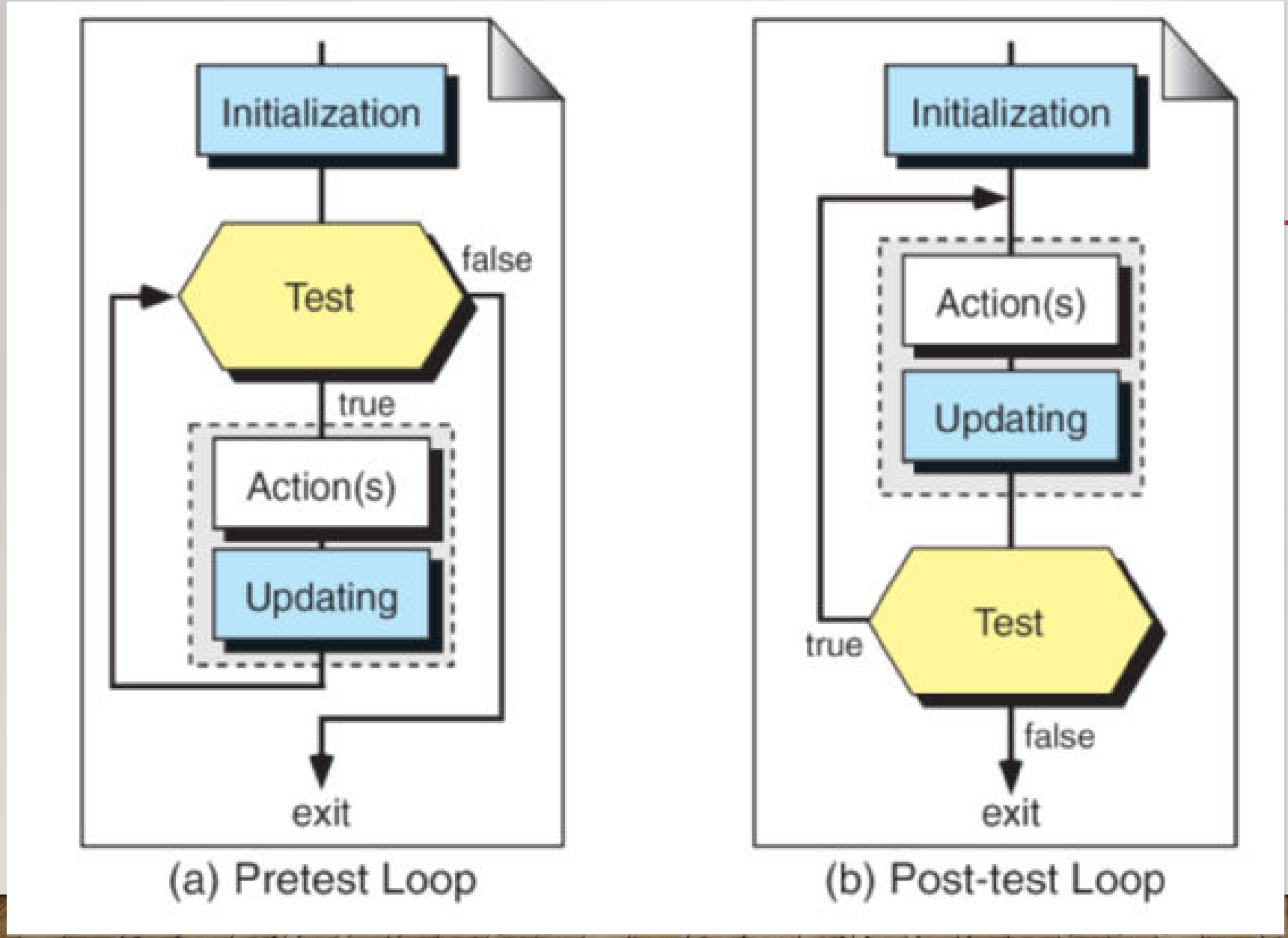


(b) Post-test Loop


Test
1
false
exit
true
Body 1
In a pretest loop, the body
may not be executed.
Body 1
Test
1
true
false
In a post-test loop, the body
must be executed at least once.
exit
(a) Pretest
(b) Post-test

6-3 Initialization and Updating

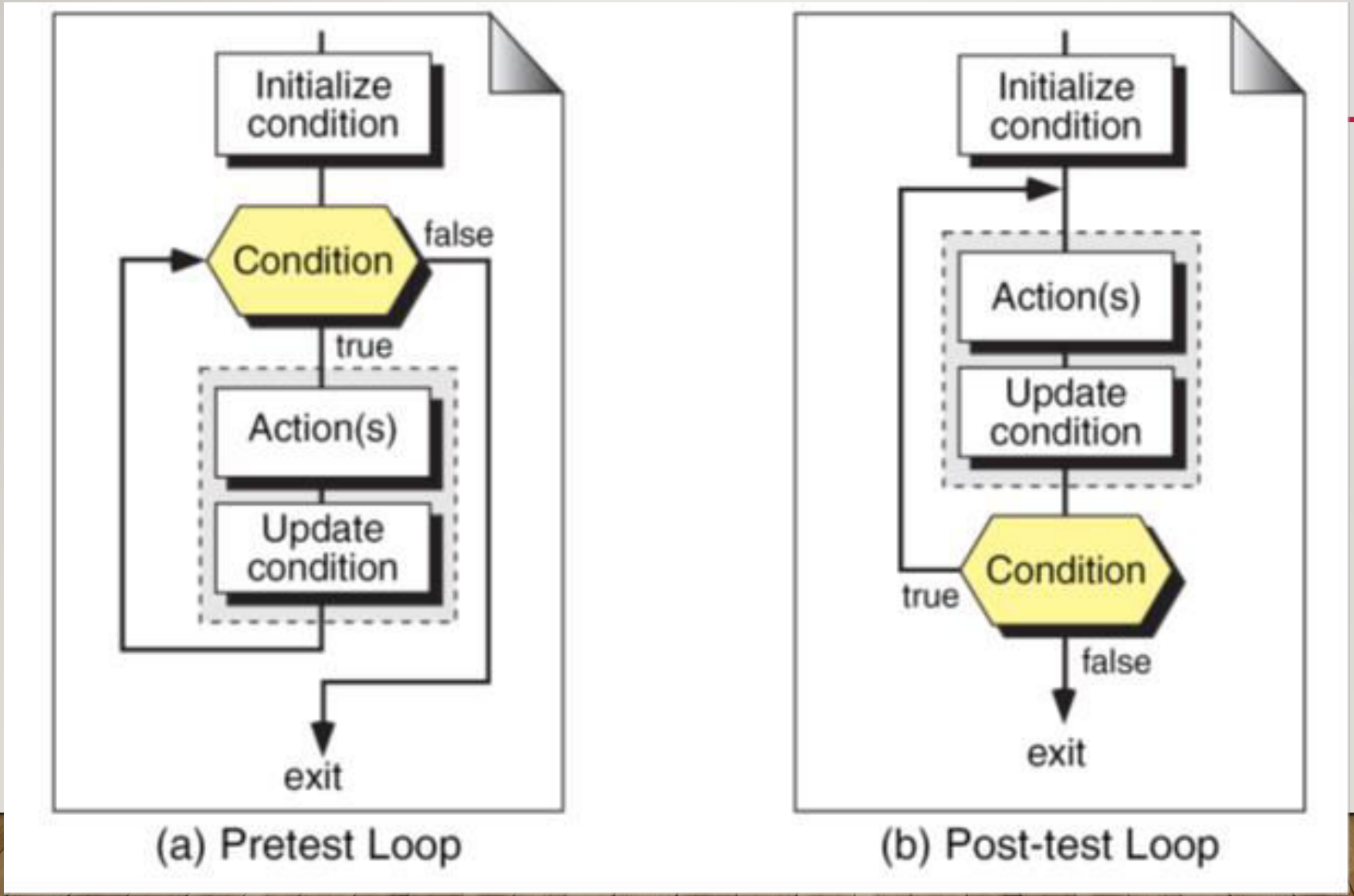
In addition to the loop control expression, two other processes, initialization and updating, are associated with almost all loops.


Initialization
Test
false
true
Action(s)
Updating
exit
(a) Pretest Loop
Initialization
Action(s)
Updating
Test
true
false
exit
(b) Post-test Loop

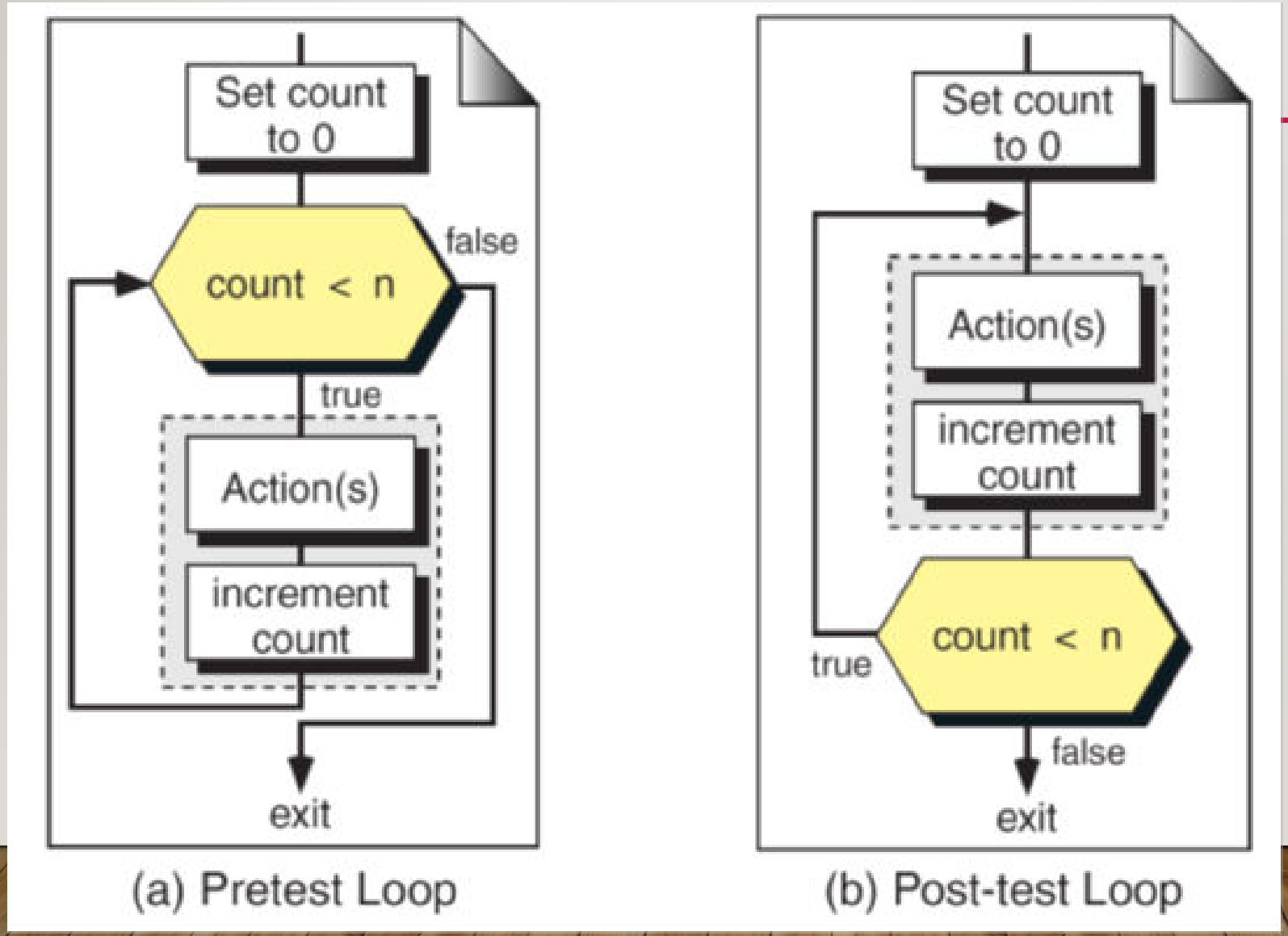
6-4 Event- and Counter-Controlled Loops

All the possible expressions that can be used in a loop limit test can be summarized into two general categories: event-controlled loops and counter-controlled loops.

EVENT CONTROLLED



COUNTER CONTROLLED

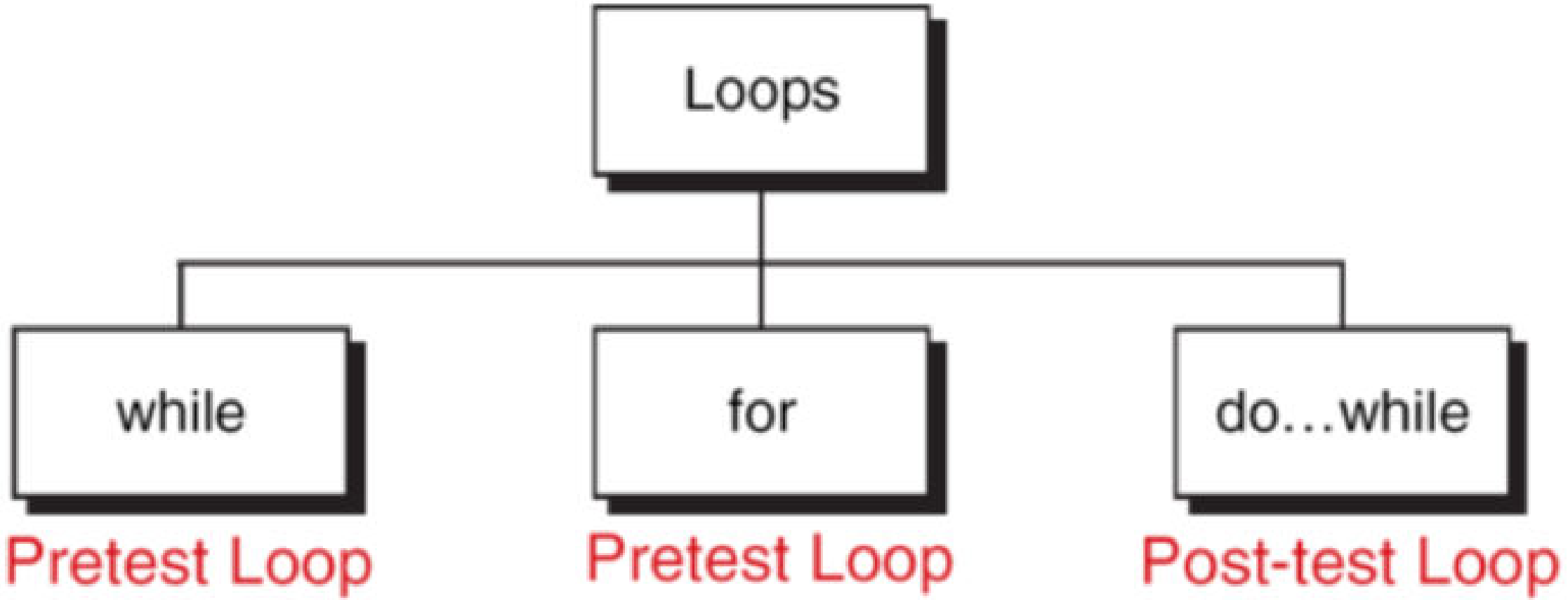


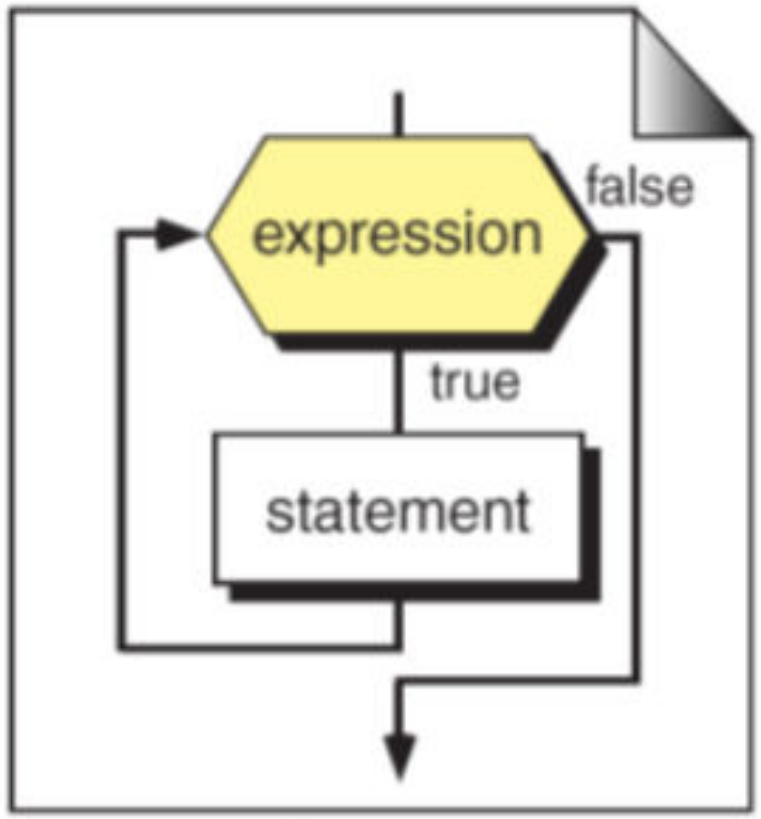
LOOPS IN C

C has three loop statements: the while, the for, and the do…while. The first two are pretest loops, and the the third is a post-test loop. We can use all of them for event-controlled and counter-controlled loops.

<u>Topics discussed in this section:</u>

The *while* Loop
The *for* Loop
The *do…while* Loop


Loops
while
for
do…while
Pretest Loop
Pretest Loop
Post-test Loop



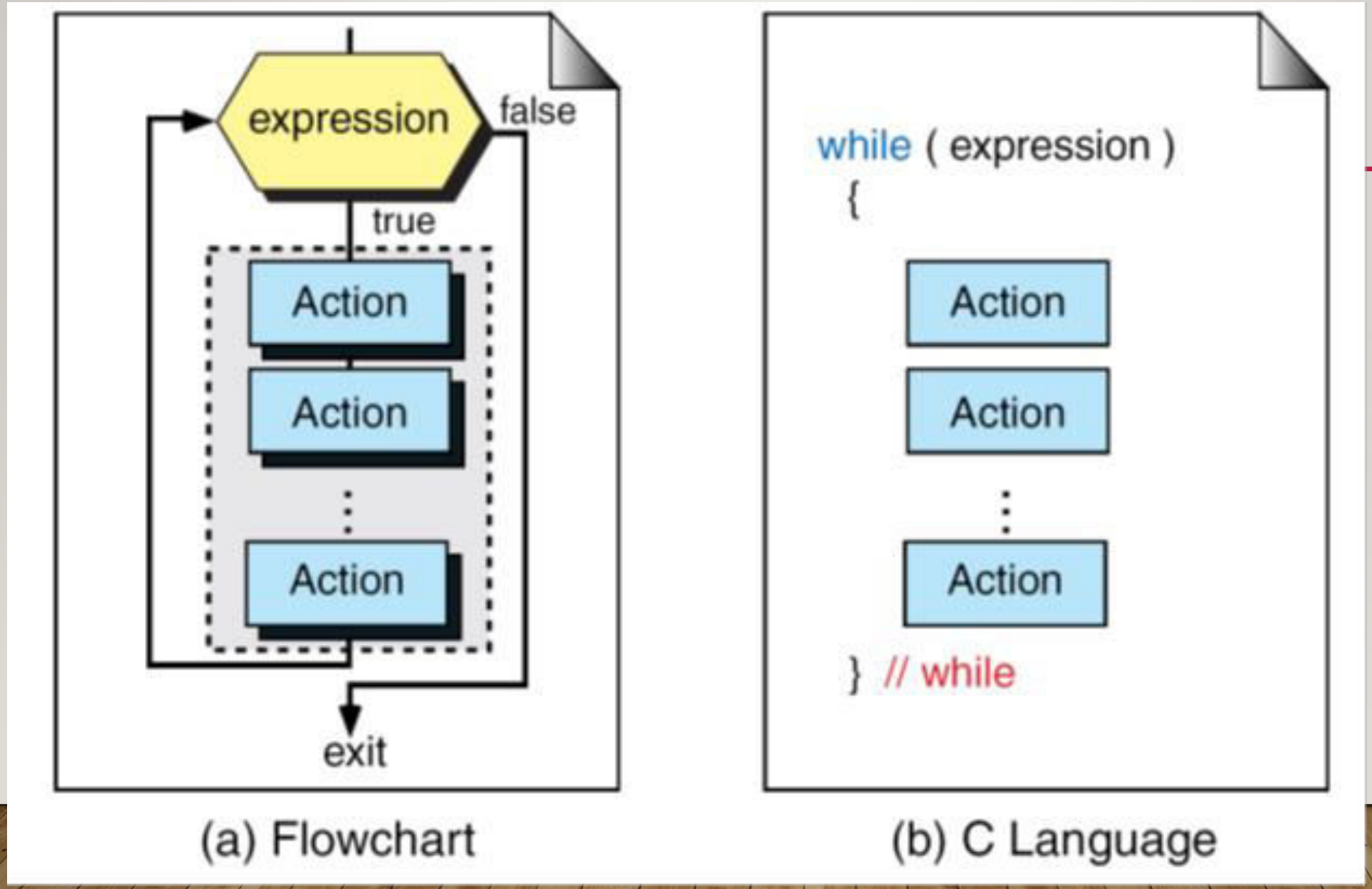
(a) Flowchart

```
while (expression)

    statement
```

(b) Sample Code

IGURE 6-10 The while Statement

COMPOUND WHILE STATEMENT



```
1  while (true)
2      {
3       temp = getTemperature();
4       if (temp < 68)
5            turnOnHeater();
6       else if (temp > 78)
7            turnOnAirCond();
8       else
9            {
10           turnOffHeater();
11           turnOffAirCond();
12           } // else
13     } // while true
```

```c
/* Simple while loop that prints numbers 10 per line.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int num;
   int lineCount;

// Statements
   printf ("Enter an integer between 1 and 100: ");
   scanf  ("%d", &num);          // Initialization

   // Test number
   if (num > 100)
       num = 100;

```

```
21      lineCount = 0;
22      while (num > 0)
23          {
24           if (lineCount < 10)
25                lineCount++;
26           else
27               {
28                printf("\n");
29                lineCount = 1;
30               } // else
31           printf("%4d", num--);          // num-- updates loop
32          } // while
33      return 0;
34  }   // main
```

Results:

```
Enter an integer between 1 and 100: 15
  15   14   13   12   11   10    9    8    7    6
   5    4    3    2    1
```

```c
/* Add a list of integers from the standard input unit
   Written by:
   Date:
*/
#include <stdio.h>
int main (void)
{
// Local Declarations
   int x;
   int sum = 0;

// Statements
   printf("Enter your numbers: <EOF> to stop.\n");
   while (scanf("%d", &x) != EOF)
      sum += x;
   printf ("\nThe total is: %d\n", sum);
   return  0;
}  // main
```

```
Results:
Enter your numbers: <EOF> to stop
15
22
3^d
The total is: 40
```

```c
#include <stdio.h>

int main() {

  char op;
  double first, second;
  printf("Enter an operator (+, -, *, /): ");
  scanf("%c", &op);
  printf("Enter two operands: ");
  scanf("%lf %lf", &first, &second);

  switch (op) {
    case '+':
      printf("%.1lf + %.1lf = %.1lf", first, second, first + second);
      break;
```

```
 case '-':
      printf("%.1lf - %.1lf = %.1lf", first, second, first - second);
      break;
    case '*':
      printf("%.1lf * %.1lf = %.1lf", first, second, first * second);
      break;
    case '/':
      printf("%.1lf / %.1lf = %.1lf", first, second, first / second);
      break;
    // operator doesn't match any case constant
    default:
      printf("Error! operator is not correct");
  }

  return 0;
}
```

QUADRATIC EQUATION

The standard form of a quadratic equation is:

$$ax^2 + bx + c = 0, \text{ where}$$
$$a, b \text{ and } c \text{ are real numbers and}$$
$$a \neq 0$$

term $b^2 - 4ac$ is known as the **discriminant** of a quadratic equation.

If the discriminant > 0,

$$root1 = \frac{-b + \sqrt{(b^2 - 4ac)}}{2a}$$

$$root2 = \frac{-b - \sqrt{(b^2 - 4ac)}}{2a}$$

If the discriminant = 0,

$$root1 = root2 = \frac{-b}{2a}$$

If the discriminant < 0,

$$root1 = \frac{-b}{2a} + \frac{i\sqrt{-(b^2 - 4ac)}}{2a}$$

$$root2 = \frac{-b}{2a} - \frac{i\sqrt{-(b^2 - 4ac)}}{2a}$$

$x^2 - 7x + 10 = 0$ are $x = 2$ and $x = 5$ because they satisfy the equation. i.e. when each of them is substituted in the given equation we get 0.

- when $x = 2$, $2^2 - 7(2) + 10 = 4 - 14 + 10 = 0$.
- when $x = 5$, $5^2 - 7(5) + 10 = 25 - 35 + 10 = 0$.

The roots of a quadratic equation $ax^2 + bx + c = 0$ are found using

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

→ Quadratic Formula

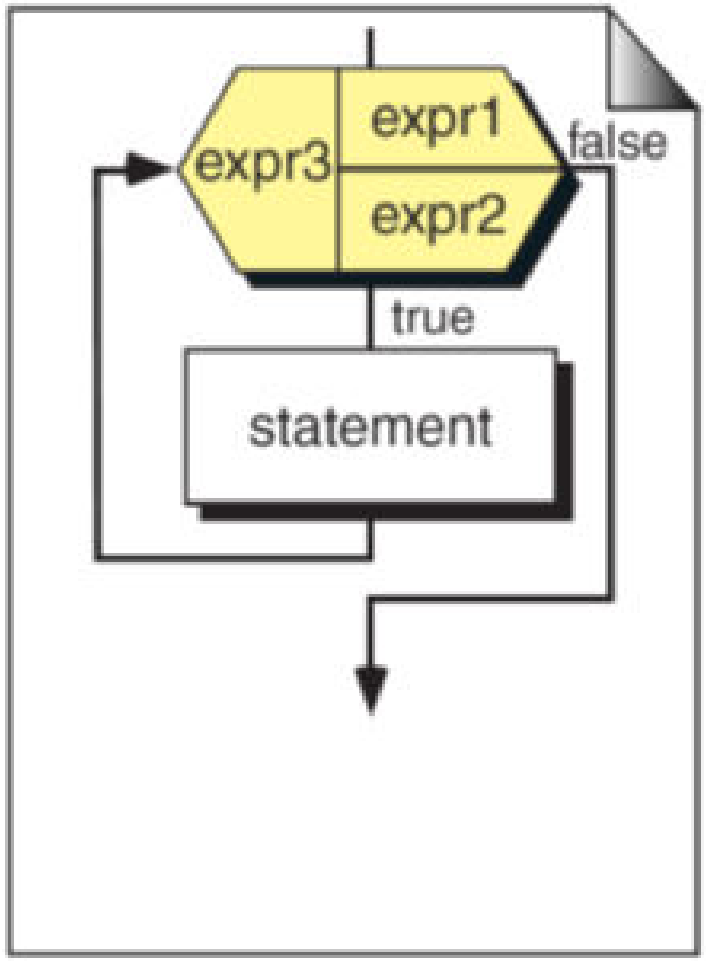
```
#include <stdio.h>
#include <math.h>

void main()
{
  int a,b,c,d;
  float x1,x2,real,img;

  printf("Input the value of a,b & c : ");
  scanf("%d%d%d",&a,&b,&c);
  if(a==0 &&b==0)
  {
        printf(" invalid inputs\n");
  }
   else if(a==0)
  {
        printf(" Leniar equation\n");
    x1=-c/b;
        printf(" linear equation is: %f",x1);

  }
else
{
        d=b*b-4*a*c;
       if(d==0)
        {
                printf("Both roots are equal.\n");
                x1=-b/(2.0*a);
                   x2=x1;
                printf("First  Root Root1= %f\n",x1);
                printf("Second Root Root2= %f\n",x2);
        }
       else if(d>0)
       {
        printf("Both roots are real and diff-2\n");
        x1=(-b+sqrt(d))/(2*a);
        x2=(-b-sqrt(d))/(2*a);
        printf("First  Root Root1= %f\n",x1);
        printf("Second Root root2= %f\n",x2);
       }
```

```
        else
    {
        printf("Root are imaginary\n");
        real=-b/(2*a);
         img= sqrt(fabs(d))/(2*a);
        printf("root 1:%f+%fi \n root 2:%f-%fi",real,img,real,img);
        }
    }

}
```

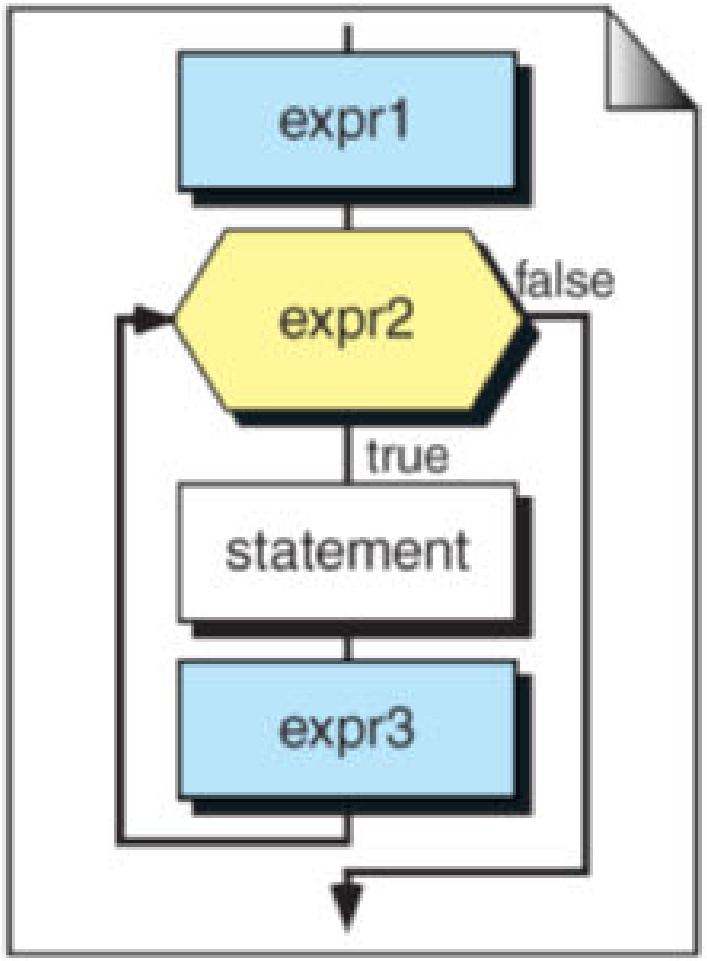
```c
#include <math.h>
#include <stdio.h>
int main() {
    double a, b, c, discriminant, root1, root2, realPart, imagPart;
    printf("Enter coefficients a, b and c: ");
    scanf("%lf %lf %lf", &a, &b, &c);

    discriminant = b * b - 4 * a * c;

    // condition for real and different roots
    if (discriminant > 0) {
        root1 = (-b + sqrt(discriminant)) / (2 * a);
        root2 = (-b - sqrt(discriminant)) / (2 * a);
        printf("root1 = %.2lf and root2 = %.2lf", root1, root2);
    }
```

```
    // condition for real and equal roots
    else if (discriminant == 0) {
        root1 = root2 = -b / (2 * a);
        printf("root1 = root2 = %.2lf;", root1);
    }

    // if roots are not real
    else {
        realPart = -b / (2 * a);
        imagPart = sqrt(-discriminant) / (2 * a);
        printf("root1 = %.2lf+%.2lfi and root2 = %.2f-%.2fi", realPart, imagPart, realPar
    }

    return 0;
}
```

FOR STATEMENT



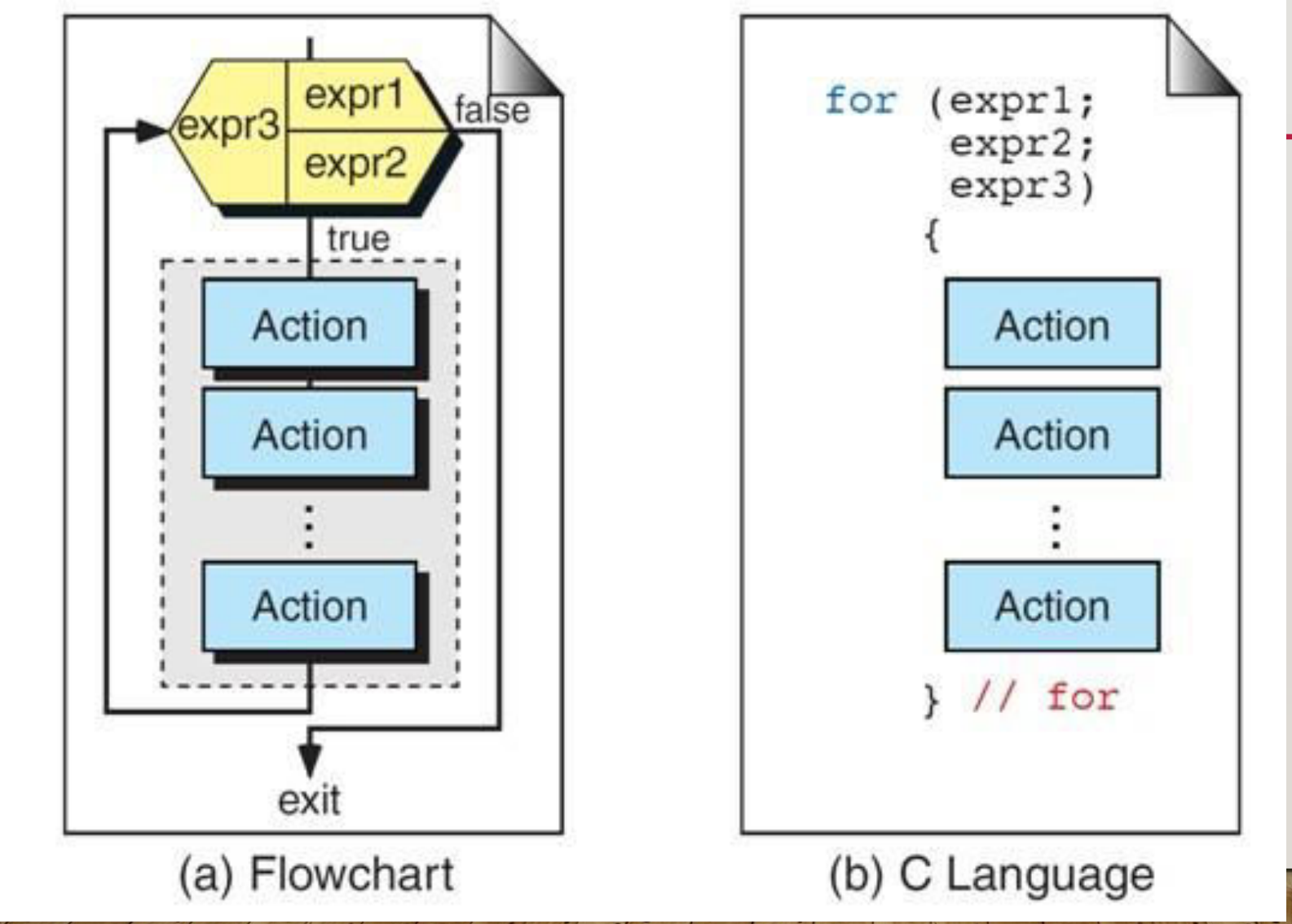
(a) Flowchart



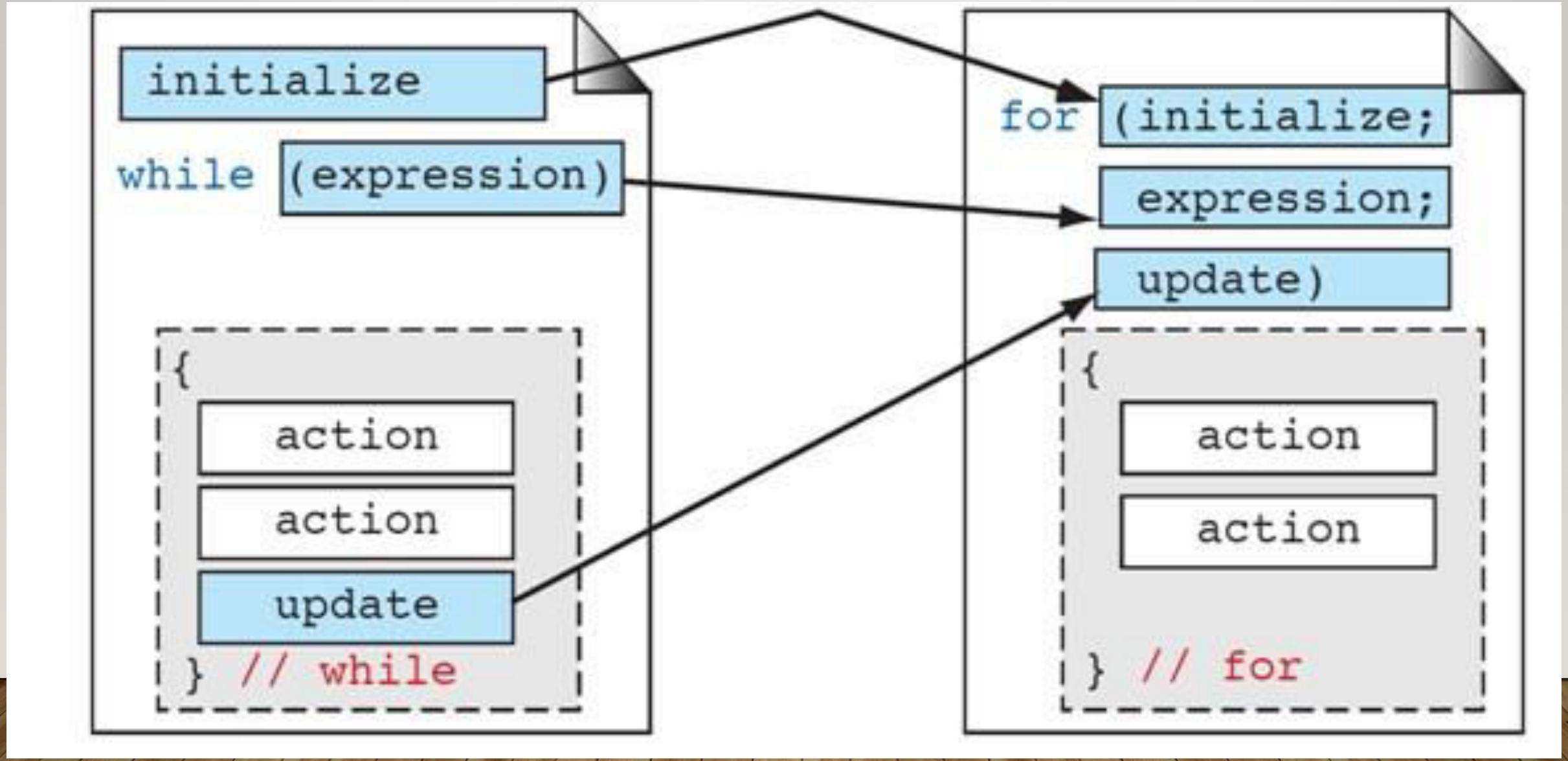
(b) Expanded Flowchart

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

COMPOUND FOR STATEMENT



WHILE V/S FOR


initialize
while (expression)
{
action
action
update
} // while
for (initialize;
expression;
update)
{
action
action
} // for

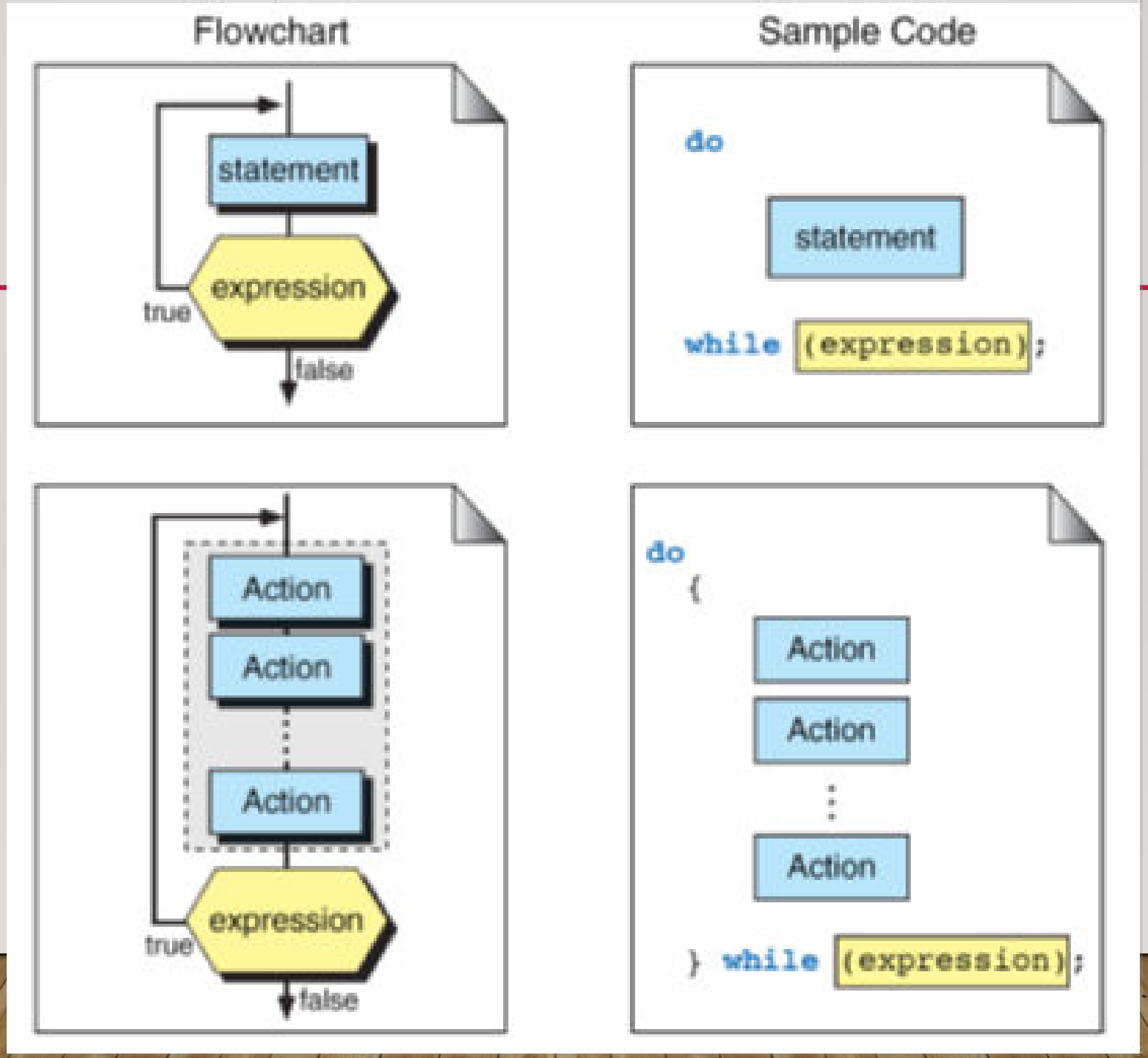
```
 4  */
 5  #include <stdio.h>
 6  int main (void)
 7  {
 8  // Local Declarations
 9     int limit;
10
11  // Statements
12     printf ("\nPlease enter the limit: ");
13     scanf ("%d", &limit);
14     for (int i = 1; i <= limit; i++)
15        printf("\t%d\n", i);
16     return 0;
17  }  // main
```

```
Results:
Please enter the limit: 3
 1
 2
 3
```

```c
#include <stdio.h>

int main (void)
{
// Statements
   for (int i = 1; i <= 3; i++)
       {
        printf("Row %d: ", i);
        for (int j = 1; j<= 5; j++)
             printf("%3d", j);
        printf("\n");
       } // for i
   return 0;
}   // main
```

```
Results:
Row 1:   1  2  3  4  5
Row 2:   1  2  3  4  5
Row 3:   1  2  3  4  5
```

do-while


Flowchart
Sample Code
statement
expression
true
false
do
statement
while (expression);
Action
Action
Action
expression
true
false
do
{
Action
Action
Action
} while (expression);

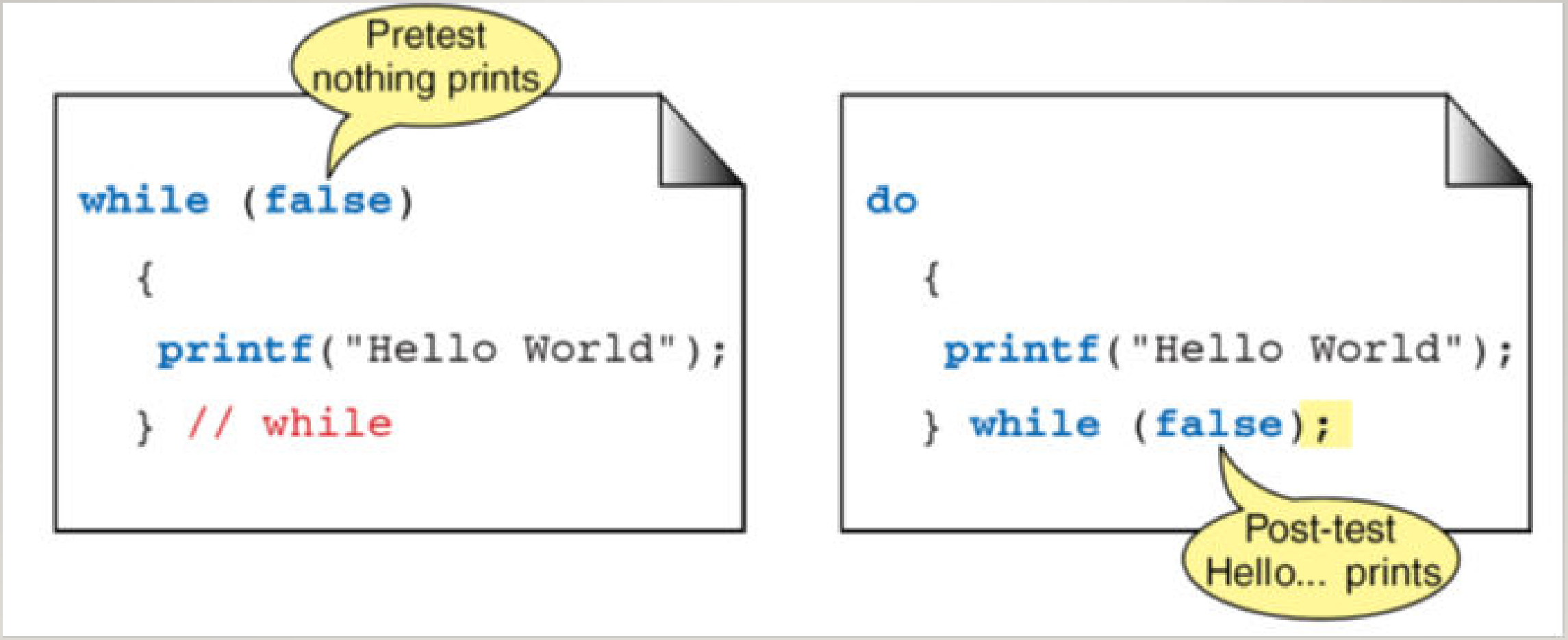
```
1  /* Demonstrate while and do...while loops.
2        Written by:
3        Date:
4  */
5  #include <stdio.h>
6
7  int main (void)
8  {
9  // Local Declarations
10    int loopCount;
11
12 // Statements
```

```
13      loopCount = 5;
14      printf("while loop      : ");
15      while (loopCount > 0)
16          printf ("%3d", loopCount--);
17      printf("\n\n");
18
```

```
19      loopCount = 5;
20      printf("do...while loop: ");
21      do
22          printf ("%3d", loopCount--);
23      while (loopCount > 0);
24      printf("\n");
25      return 0;
26  }   // main
```

Results

```
while loop      :    5   4   3   2   1

do...while loop:    5   4   3   2   1
```


Pretest
nothing prints
while (false)
{
printf("Hello World");
} // while
do
{
printf("Hello World");
} while (false);
Post-test
Hello... prints

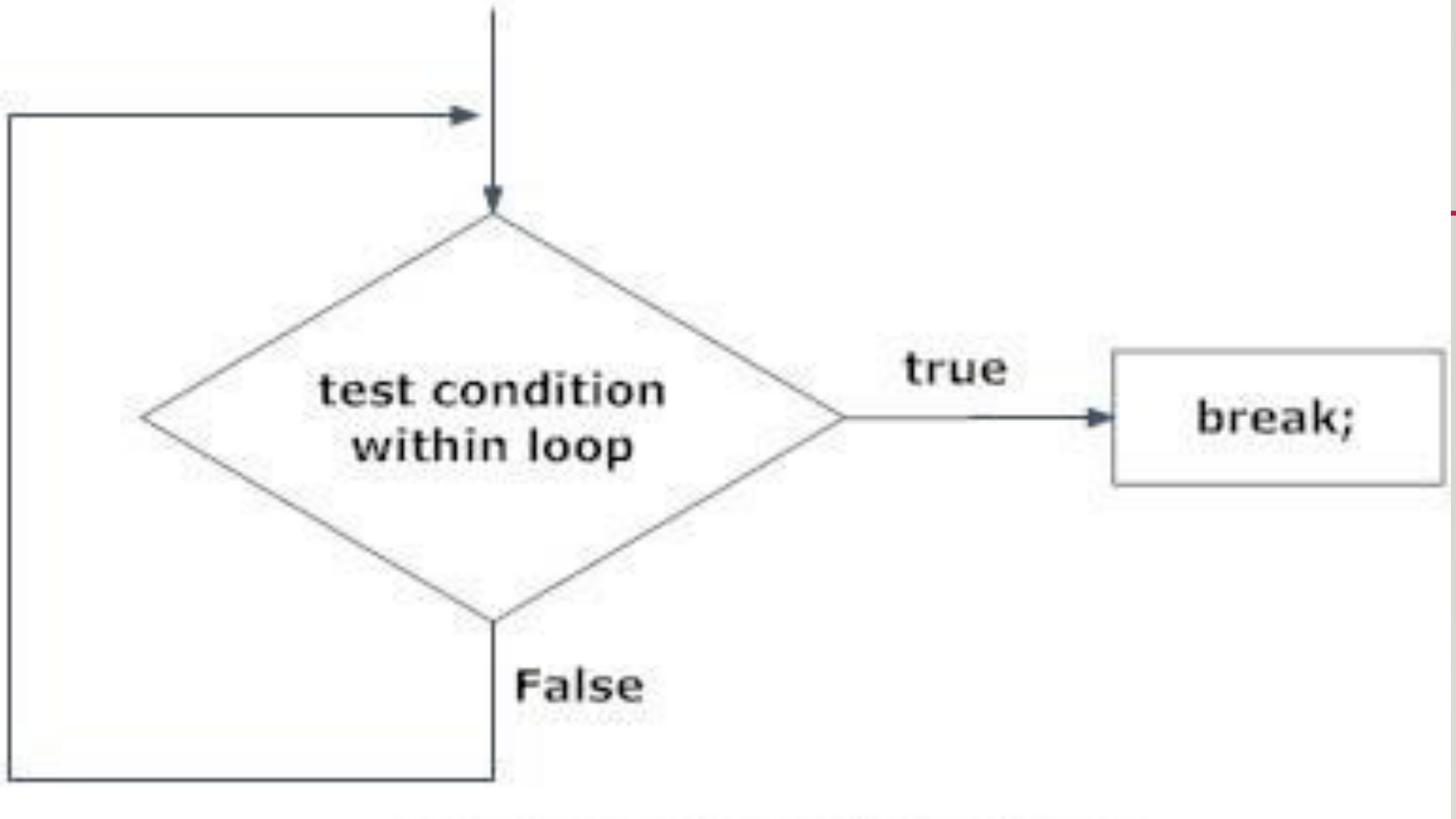
```c
/* Add a list of integers from the standard input unit
        Written by:
        Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
    int x;
    int sum = 0;
    int testEOF;

    // Statements
    printf("Enter your numbers: <EOF> to stop.\n");
    do
        {
         testEOF = scanf("%d", &x);
```

```
19          if (testEOF != EOF)
20              sum += x;
21          } while (testEOF != EOF);
22      printf ("\nTotal: %d\n", sum);
23      return  0;
24  }   // main
```

```
Results:
Run 1:
    Enter your numbers: <EOF> to stop.
    10 15 20 25 ^d
    Total: 70

Run 2:
    Enter your numbers: <EOF> to stop.
    ^d
    Total: 0
```

BREAK AND CONTINUE STATEMENTS

- **`break`**
  - Causes immediate exit from a **`while`**, **`for`**, **`do`**/**`while`** or **`switch`** structure
  - Program execution continues with the first statement after the structure
  - Common uses of the **`break`** statement
    - Escape early from a loop
    - Skip the remainder of a **`switch`** structure

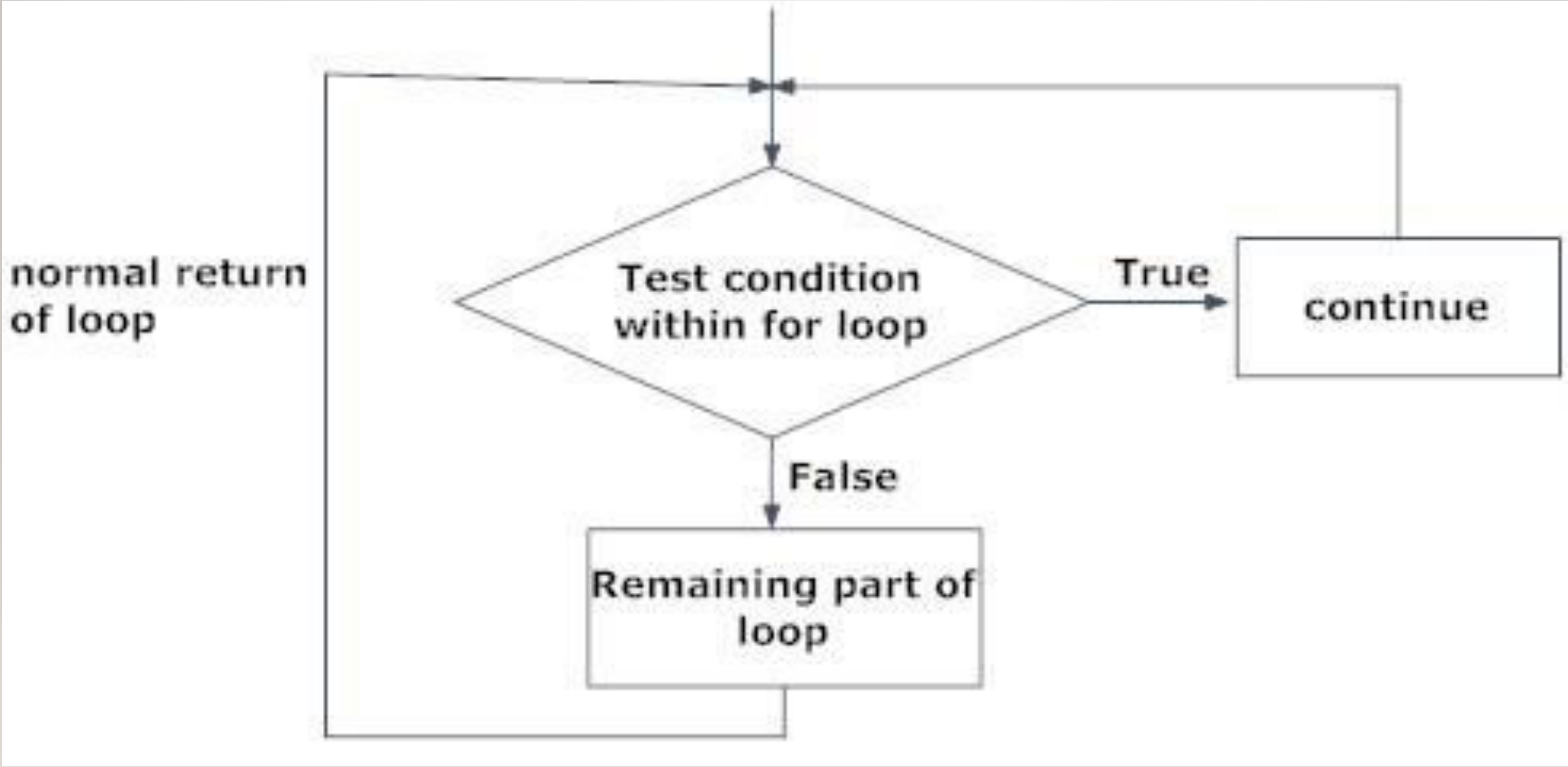

test condition
within loop
true
break;
False

```c
#include<stdio.h>
void main(){
    int num, sum=0;
    int i,n;
    printf("Enter Number of inputs\n");
    scanf("%d",&n);
    for(i=1;i<=n;++i){
        printf("Enter num%d: ",i);
        scanf("%d",&num);
        if(num==0) {
            break;      /*this breaks loop if num == 0 */
            printf("Loop Breaked\n");
        }
        sum=sum+num;
    }
    printf("Total is %d",sum);
}
```

```
Enter Number of inputs
5
Enter num1: 5
Enter num2: 10
Enter num3: 0
Loop Breaked
Total is 15
```

BREAK AND CONTINUE STATEMENTS

- **`continue`**
  - Skips the remaining statements in the body of a **`while`**, **`for`** or **`do/while`** structure
    - Proceeds with the next iteration of the loop
  - **`while`** and **`do/while`**
    - Loop-continuation test is evaluated immediately after the **`continue`** statement is executed
  - **`for`** structure
    - Increment expression is executed, then the loop-continuation test is evaluated


normal return
of loop
Test condition
within for loop
True
continue
False
Remaining part of
loop

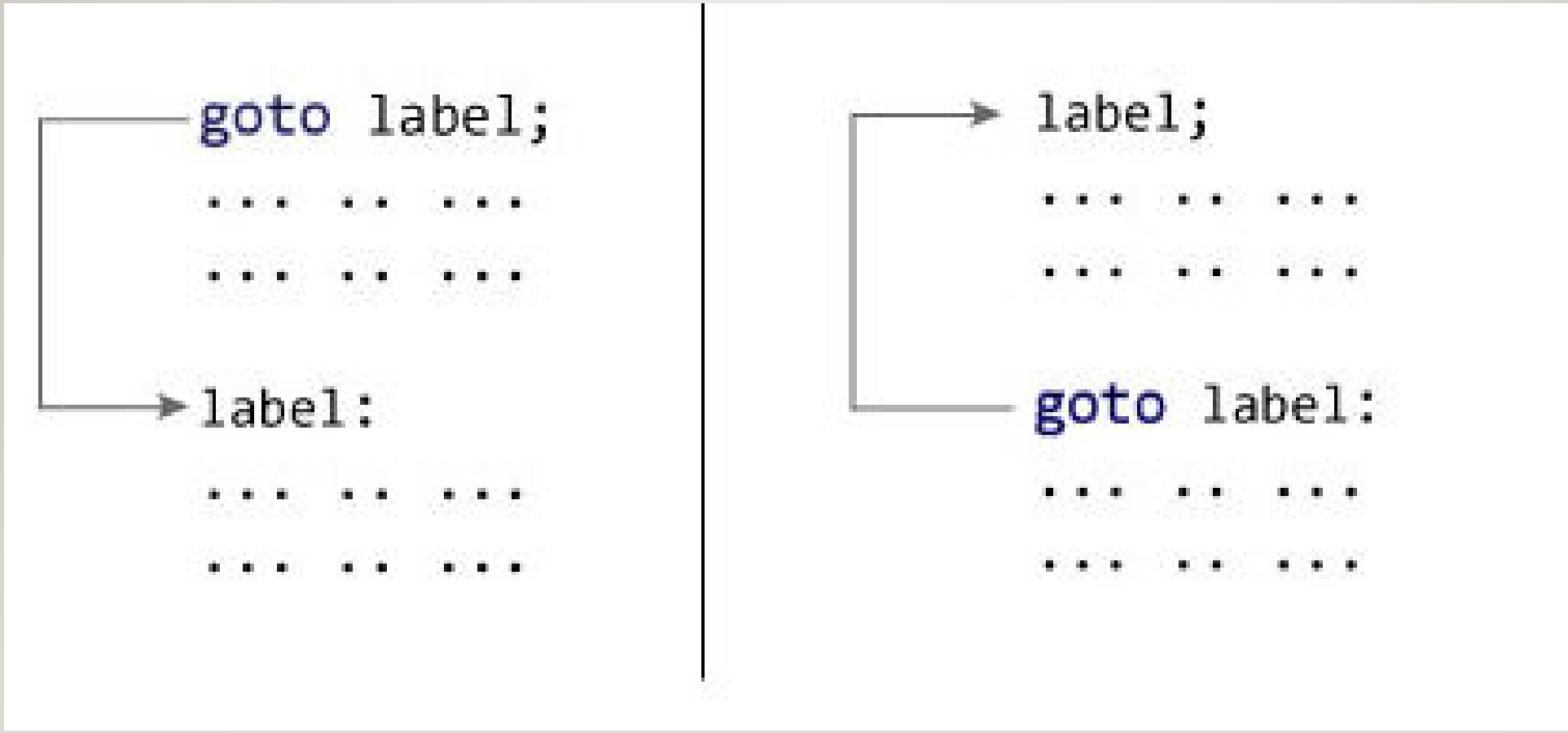
```c
#include<stdio.h>
void main(){
    int i, n=20;
     for(i=1;i<=n;++i){
        if(i % 5 == 0) {
            printf("pass\n");
            continue;      /*this continue the execution of loop if i %
        }
        printf("%d\n",i);
    }
}
```

```
1
2
3
4
pass
6
7
8
9
pass
11
12
13
14
pass
16
17
18
19
pass
```

GOTO STATEMENT

- **goto** statement is used for altering the normal sequence of program execution by transferring control to some other part of the program.

```
goto label;
.............
.............
.............
label:
statement;
```


goto label;
... .
... .
label:
... .
... .
label;
... .
... .
goto label:
... .
... .

```
#include<stdio.h>
void main()
{
    int i=1;
    count:              //This is Label

    printf("%d\n",i);
    i++;
    if(i<=10) {
        goto count;     //This jumps to label "count:"
    }
}
```

```
1
2
3
4
5
6
7
8
9
10
```

```
#include<stdio.h>
#include<stdbool.h>
int main()
{
        //local declarations
        bool b=true;
        char c='A';
        float d=245.3;
        int i=3650;
        short s=78;

        //statements
        printf("bool+char is char:%c\n",b+c);
        printf("int * short is int: %d\n",i*s);
        printf("float * char is float: %f\n",d*c);
        c=c+b;
        d=d+c;
        b=false;
        b=-d;

        printf("\n After execution...\n");
        printf("char +true:  %c\n",c);
        printf("float +char :%f\n",d);
        printf("bool =-float:%f\n",b);

        return 0;
}
```

Output

bool+char is char:B

int * short is int: 284700

float * char is float: 15944.500000

After execution...

char +true: B

float +char :311.299988

bool =-float:311.299988

```
#include<stdio.h>
#define TAX_RATE 8.50
int main()
{
        //local declarations
        int quantity;
        float discountrate;
        float discountamt;
        float unitprice;
        float subtotal;
        float subtaxable;
        float taxam;
        float total;

        //statements
        printf("enter the number of items sold:\n");
        scanf("%d",&quantity);
        printf("enter the unit price:\n");
        scanf("%f",&unitprice);

        printf("enter the discount rate(percent):\n");
        scanf("%f",&discountrate);

        subtotal=quantity*unitprice;
        discountamt=subtotal*discountrate/100;
        subtaxable=subtotal-discountamt;
        taxam=subtaxable*TAX_RATE/100;
        total=subtaxable+taxam;

        printf("\n quantity sold:       %6d\n",quantity);
        printf("unit price of items :  %9.2f\n",unitprice);
        printf("                      ----------\n");

        printf("subtotal :            %9.2f\n",subtotal);
        printf("discount :            %-9.2f\n",discountamt);
```

```
        printf("discount total:       %9.2f\n",subtaxable);
        printf("sales tax:           %+9.2f\n",taxam);
        printf("Totsl sales:         %9.2f\n",total);

        return 0;
}
```

```
Output
enter the number of items sold:
5
enter the unit price:
10
enter the discount rate(percent):
10
quantity sold:          5
unit price of items :      10.00
                    ------------
subtotal :              50.00
discount :          5.00
discount total:          45.00
sales tax:             +3.83
Totsl sales:            48.83
```