# UNIT-2

# Topics to be covered in UNIT-2

- C Tokens
- Expressions
- Decision control structures
- Repetitive control Structures
- Unconditional statements

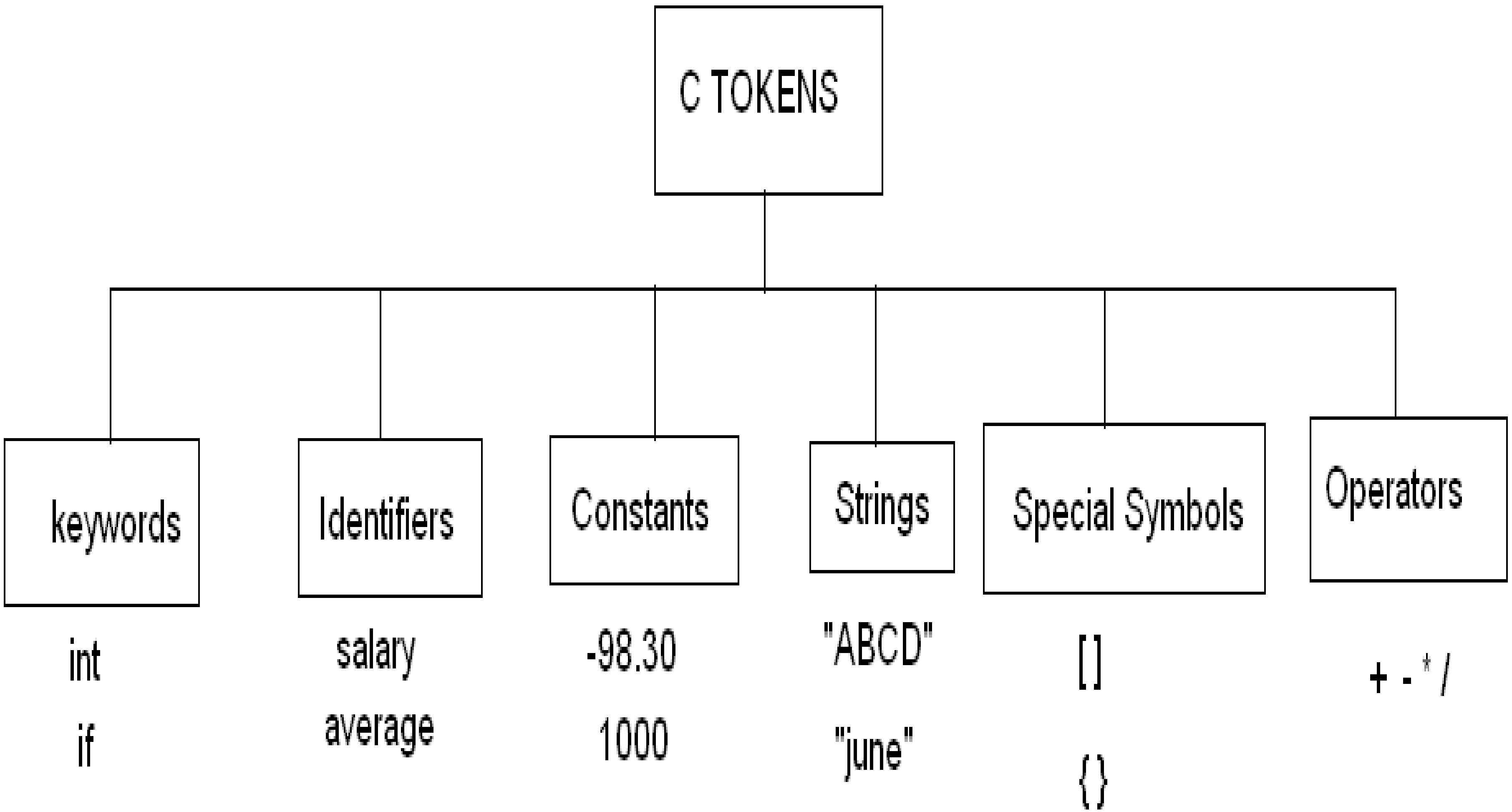
## C Tokens

- Identifiers
- Keywords
- Constants
- variables
- Operators

## C Tokens:

In a passage of text, individual words and punctuation marks are called as tokens.

The compiler splits the program into individual units, are known as C tokens. C has six types of tokens.



# Key Words

- C word is classified as either **keywords** or **identifiers**.
- Keywords have fixed meanings, these meanings cannot be changed.
- Keywords must be in **lowercase**.

# Key Words in C

| | | | | |
|---|---|---|---|---|
| auto | break | case | char | const |
| continue | default | do | double | else |
| enum | extern | float | for | goto |
| if | int | long | register | return |
| short | signed | sizeof | static | struct |
| switch | typedef | union | unsigned | void |
| volatile | while | | | |

# Character Set

- Characters are used to form words, numbers and expressions.
- Characters are categorized as
  - Letters
  - Digits
  - Special characters
  - White spaces.

**Letters**: (Upper Case and Lower Case)

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

a b c d e f g h i j k l m n o p q r s t u v w x y z

**Digits:** 0 1 2 3 4 5 6 7 8 9

**Special Characters:**

’ " ( ) * + - / : = ! & $ ; < > % ? , . ˆ # @ ˜ ‘ { } [ ] \ |

**White Spaces:** Blank Space, Horizontal Space, Carriage Return, New Line.

# Identifiers

- Identifiers are **names** given to various programming elements such as variables, constants, and functions.
- It should **start** with an **alphabet or underscore**, followed by the combinations of alphabets and digits.
- No special character is allowed except underscore.
- An Identifier can be of arbitrarily long. Some implementation of C recognizes only the first 8 characters and some other recognize first 32 Characters.

**The following are the rules for writing identifiers in C:**

- First character must be alphabetic character or underscore.
- Must consist only of alphabetic characters, digits, or underscore.
- Should not contain any special character, or white spaces.
- Should not be C keywords.
- Case matters (that is, upper and lowercase letters). Thus, the names **count** and **Count** refer to two different identifiers.

| Identifier | Legality |
| --- | --- |
| Percent | Legal |
| y2x5__fg7h | Legal |
| annual profit | Illegal: Contains White space |
| _1990_tax | Legal but not advised |
| savings#account | Illegal: Contains the illegal character # |
| double | Illegal: It is s a C keyword |
| 9winter | Illegal: First character is a digit |

**Examples of legal and illegal C identifiers**

# Constants

- Constants are data values that cannot be changed during the program execution.
- Like variables, constants have a type.

**Types of constants:**

- **<u>Boolean constants:</u>** A Boolean data type can take only two values **true** and **false**.
- **<u>Numeric constants.</u>**
  - integer constant
  - real constants
- **<u>Character constants</u>**
  - Single character constants
  - string constants
- **<u>Coding Constants</u>**
  - Literal constants
  - Defined constants
  - Memory constants

# Numeric Constants

**integer constant:** It is a sequence of digits that consists numbers from 0to 9.

**Example:** 23 -678 0 +78

**Rules:**

1. integer constant have at least one digit.
2. No decimal points.
3. No commas or blanks are allowed.
4. The allowable range for integer constant is **-32768** to **32767.**

To store the larger integer constants on 16 bit machine use the qualifiers such as U,L,UL.

| **Type** | **Representation** | **Value** |
|---|---|---|
| int | +245 | 245 |
| int | -678 | -678 |
| unsigned int | 65342u / 65342U | 65342 |
| unsigned long int | 99999UL | 99999 |
| long int | 999999L | 999999 |

**Real constants:** The numbers containing fractional parts like 3.14

**Example:** 1.9099 -0.89 +3.14

Real constants are also expressed in exponential notation.

**Mantissa e exponent**

- A number is written as the combination of the mantissa, which is followed by the prefix **e** or **E**, and the exponent.

  **Example:**

| | | |
|---|---|---|
| 87000000 | = | 8.7e7 |
| - 550 | = | -5.5e2 |
| 0.00000000031 | = | 3.1e-10. |

# Examples of real constants

| Type | Representation | Value |
|---|---|---|
| double | 0. | 0.0 |
| double | 0.0 | .0 |
| float | -2.0f | -2.0 |
| long double | 3.14159276544L | 3.14159276544 |

# Single character constants

- A **single character constants** are enclosed in **single quotes**.

  Example: ‘1’ ‘X’ ‘%’ ‘ ‘

- Character constants have integer values called **ASCII** values.

  char ch=‘A’;

  **printf(“%d”,ch);** **Output: 65**

  similarly **printf(“%c”,65)** **Output: A**

# string Constants

- **String** is a collection of characters or sequence of characters enclosed in **double quotes**.
- The characters may be letters, numbers, special characters and blank space.

  Example: “snist” “2016” “A”.

# Backslash \escape characters

- **Backslash characters** are used in **output** functions.
- These backslash characters are preceded with the \ symbol.

| constant | meaning |
|---|---|
| **‘\a’** | Alert(bell) |
| **‘\b’** | Back space |
| **‘\f’** | Form feed |
| **‘\n’** | New line |
| **‘\r’** | Carriage return |
| **‘\v’** | Vertical tab |
| **‘\t’** | Horizontal tab |
| ‘\’’ | Single quote |
| ‘\”’ | Double quotes |
| **‘\?’** | Question mark |
| ‘\\’ | Backslash |
| **‘\0’** | null |

## Coding Constants: Different ways to create constants.

**Literal constants:**

A literal is an unnamed constant used to specify data.

**Example: a = b + 5;**

**Defined constants:**

By using the preprocessor command you can create a constant.

**Example: #define PI 3.14**

**Memory constants:**

Memory constants use a C type qualifier, const, to indicate that the data can not be changed.

Its format is: const type identifier = value;

**Example:** const float PI = 3.14159;

```
#include<stdio.h>
void main()
{
    const int a=10;
    int b=20;
    a=a+1;
    b=b+1;
    printf("a=%d  b=%d",a,b);
}
```

# Operators

- C supports a rich set of operators.
- An **operator** is a symbol that tells the computer to perform mathematical or logical operations.
- Operators are used in C to **operate on data and variables**.

expression

| X=Y+Z |
|---|

Operators: **=, +** Operands: **x, y, z**

- **Unary operators** are used on a **single operand** (**- -, +, ++, --**)
- **Binary operators** are used to apply in between **two operands** (**+, -, /,*, %**)
- Conditional (or **ternary**) operator can be applied on **three operands**. ( **?:** )

# Types of Operators

C operators can be classified into a number of categories. They include:

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Assignment Operator
- Increment and Decrement Operators
- Conditional Operators
- Bitwise Operators
- Special Operators

# Arithmetic Operators

| C Operation | Binary Operator | C Expression |
|---|---|---|
| Addition | + | a + b |
| Subtraction | - | a - b |
| Multiplication | * | a * b |
| Division(second operand must be nonzero) | / | a / b |
| Modulus (Remainder both operands must be integer and second operand must be non zero) | % | a % b |

**Syntax:** **operand1 arithmetic_operator operand2**

**Examples:**

**10 + 10 = 20** **(addition on integer numbers)**
**10.0 + 10.0 = 20.0** **(addition on real numbers)**
**10 + 10.0 = 20.0** **(mixed mode)**
**14 / 3 = 4** **(ignores fractional part)**

# Relational Operators

- Relational operators are used to compare the relationship between two operands.

  **Syntax:** exp1 **relational_operator** exp2

- The value of a relational expression is either one or zero.
- It is **one** if the specified relation is **true** and **zero** if the relation is **false.**
- Relational operators are used by **if , while** and **for** statements.

| C operation | Relational Operator | C expression |
|---|---|---|
| greater than | > | x > y |
| less than | < | x < y |
| greater than or equal to | >= | x >= y |
| less than or equal to | <= | x <= y |
| Equality | == | x == y |
| not equal | != | x != y |

# Logical Operators

- Logical operators used to test more than one condition and make decision. Yields a value either one or zero.
- **Syntax:** operand1 **logical_operator** operand2 **or** **logical_operator** operand
- **Example:** (x<y) && (x= = 8)

| Operator | Meaning |
|---|---|
| && | Logical AND (true only if both the operands are true) |
| \|\| | Logical OR (true if either one operand is true) |
| ! | Logical NOT (negate the operand) |

| A | B | A && B | A \|\| B |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 |

| A | ! A |
|---|---|
| 0 | 1 |
| 1 | 0 |

# Assignment Operators

- Assignment operators are used to assign the result of an expression to a variable.
- Assignment Operator is **=**

**Syntax:** **variable = expression;**

- **Types of assignment:**
  - Single Assignment Ex: a = 10;
  - Multiple Assignment Ex: a=b=c=0;
  - Compound Assignment Ex: c = a + b;

| Operator | Example | Equivalent Statement |
|---|---|---|
| += | c += 7 | c = c + 7 |
| -= | c -= 8 | c = c – 8 |
| *= | c *= 10 | c = c * 10 |
| /= | c /= 5 | c = c / 5 |
| %= | c %= 5 | c = c % 5 |

# Increment and Decrement Operators

- We can add or subtract 1 to or from variables by using **increment (++)** and **decrement (--)** operators.
- The operator ++ **adds 1** to the operand and the operator **– – subtracts 1**.
- They can apply in two ways: **postfix** and **prefix.**
- **Syntax:** **increment or decrement_operator operand**

  **operand increment or decrement_operator**
- **Prefix form:** Variable is changed before expression is evaluated
- **Postfix form:** Variable is changed after expression is evaluated.

| Operator | Example | Meaning | Equivalent Statements |
|---|---|---|---|
| ++ | i++ | postfix | i=i+1; i+=1; |
| ++ | ++i | prefix | i=i+1; i+=1; |
| -- | i-- | postfix | i=i-1; i -=1; |
| -- | --i | prefix | i=i-1; i-=1; |

# **Conditional (ternary)Operators ( ?: )**

- C's only conditional (or **ternary**) operator requires three operands.

  **Syntax:** **conditional_expression? expression1: expression2;**
- The conditional_expression is any expression that results in a true (nonzero) or false (zero).
- If the result is true then expression1 executes, otherwise expression2 executes.

**Example:**

```
a=1;
b=2;
x = (a<b)?a:b;
```

This is like

```
if(a<b)
    x=a;
else
    x=b;
```

# Bitwise Operators

- C has a special operator known as Bitwise operator for manipulation of data at bit level.
- Bitwise operator may not be applied for float and double.
- Manipulates the data which is in binary form.
- **Syntax:** **operand1 bitwise_operator operand2**

| Bitwise Operators | Meaning |
|---|---|
| & | Bitwise AND |
| \| | Bitwise OR |
| ^ | Exclusive OR |
| << | Shift left |
| >> | Shift right |
| ~ | One's compliment |

<table>
<tr><th>A</th><th>B</th><th>A&B</th><th>A|B</th><th>A^B</th></tr>
<tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>0</td><td>1</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>0</td><td>0</td><td>1</td><td>1</td></tr>
<tr><td>1</td><td>1</td><td>1</td><td>1</td><td>0</td></tr>
</table>

- **Examples:**

<table>
<tr><td>&</td><td>Bitwise AND</td><td>0110 & 0011</td><td>→</td><td>0010</td></tr>
<tr><td>|</td><td>Bitwise OR</td><td>0110 | 0011</td><td>→</td><td>0111</td></tr>
<tr><td>^</td><td>Bitwise XOR</td><td>0110 ^ 0011</td><td>→</td><td>0101</td></tr>
<tr><td><<</td><td>Left shift</td><td>01101110 << 2</td><td>→</td><td>10111000</td></tr>
<tr><td>>></td><td>Right shift</td><td>01101110 >> 3</td><td>→</td><td>00001101</td></tr>
<tr><td>~</td><td>One's complement</td><td>~0011</td><td>→</td><td>1100</td></tr>
</table>

- Don't confuse bitwise & | with logical && ||

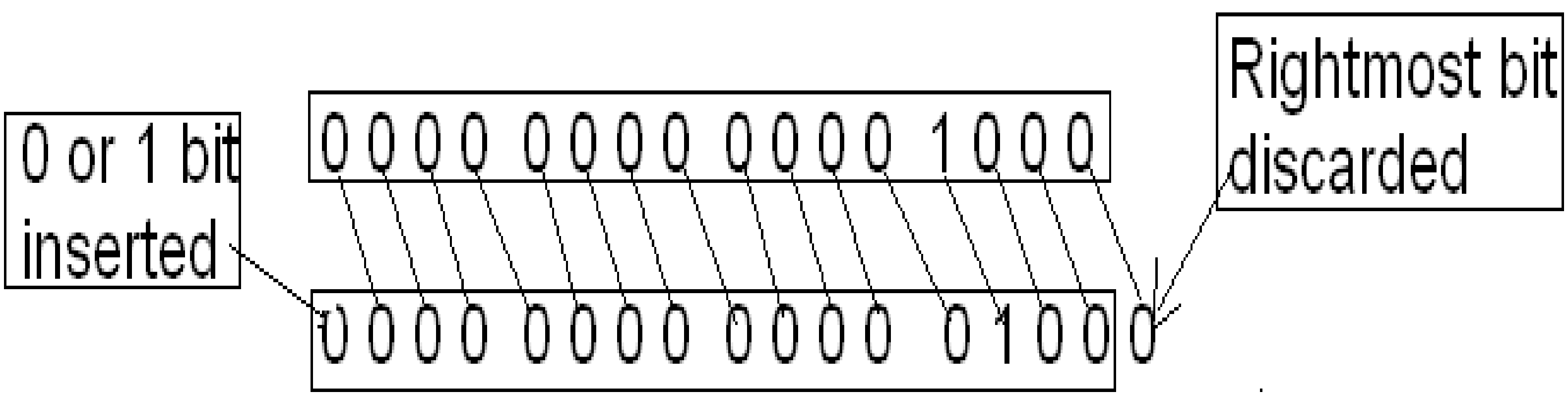
- **>>** is a binary operator that requires two integral operands. the first one is value to be shifted, the second one specifies number of bits to be shifted.
- The general form is as follows:

**variable >> expression;**

- When bits are shifted right, the bits at the rightmost end are deleted.
- Shift right operator divides by a power of 2. I.e. a>>n results in $a/2^n$, where **n** is number of bits to be shifted.

**Example:** a=8;

b=a>>1; // assigns 4 after shift right operation



- **<<** is a binary operator that requires two integral operands. the first one is value to be shifted, the second one specifies number of bits to be shifted.
- The general form is as follows:

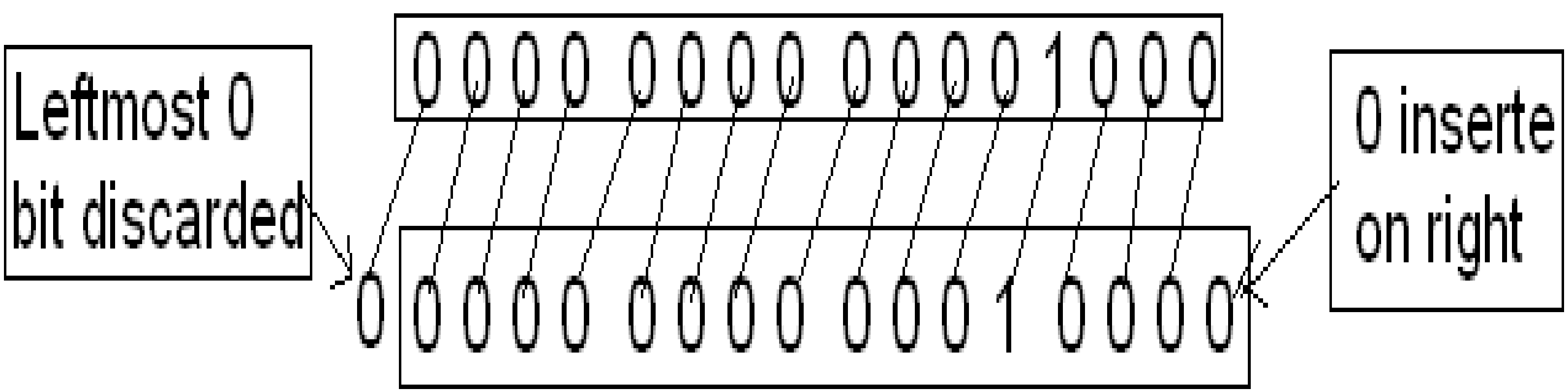
**variable << expression;**

- When bits are shifted left, the bits at the leftmost end are deleted.

**Example:** a=8;

b=a<<1; // assigns 16 after left shift operation

- Shift left operator multiply by a power of 2, a<<n results in $a*2^n$, where **n** is number of bits to be shifted.



# Special Operators

- C supports the following special category of operators.

| | |
|---|---|
| **&** | Address operator |
| ***** | Indirection operator |
| **,** | Comma operator |
| **sizeof()** | Size of operator |
| **.** and **→** | Member selection Operators |

**comma operator :**

- It doesn't operate on data but allows more than one expression to appear on the same line.

  **Example:** int i = 10, j = 20;

  printf (%d %.2f %c", a,f,c);

  j = (i = 12, i + 8); //i is assigned 12 added to 8 produces 20

**sizeof Operator :**

- It is a unary operator (operates on a single value).
- Produces a result that represent the size in bytes.

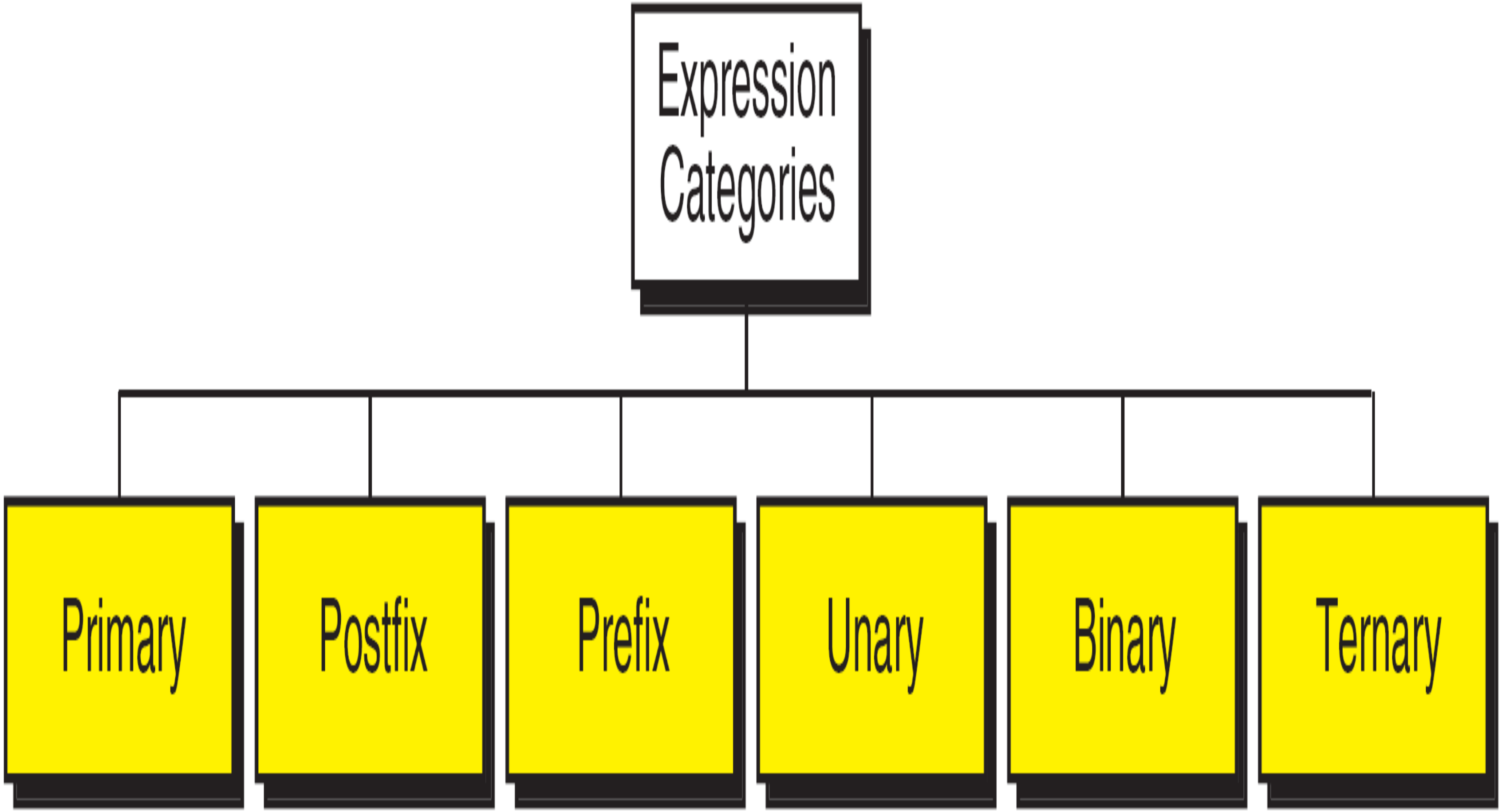
  **Syntax:** **sizeof(datatype);**

  **Example:**

```
int a = 5;
sizeof (a);        //produces 2
sizeof(char);      // produces 1
sizeof(int);       //  produces 2
```

## Expressions

- **Arithmetic expressions**
- **Precedence and Associativity**
- **Evaluating expressions**




**Expression Categories**

- 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.

## Primary Expression:

- The most elementary type of expression is a primary expression.
- It consists of **only one operand with no operator.**
- In C, the operand in the primary expression can be a **name**, a **constant**,or a **parenthesized expression**.
- Name is any identifier for a variable, a function, or any other object in the language.
- The following are examples of primary expressions:

**Example:** a price sum max

- Literal Constants is a piece of data whose value can't change during the execution of the program.

## Primary Expression: (contd…)

- The following are examples of literal constants used in primary expression:
  **Example:** ‘A’ 56 98 12.34

- Any value enclosed in parentheses must be reduced in a single value is called as primary expression.

- The following are example of parentheses expression:
  **Example:** (a*x + b) (a-b*c) (x+90)

**Post fix expression:**

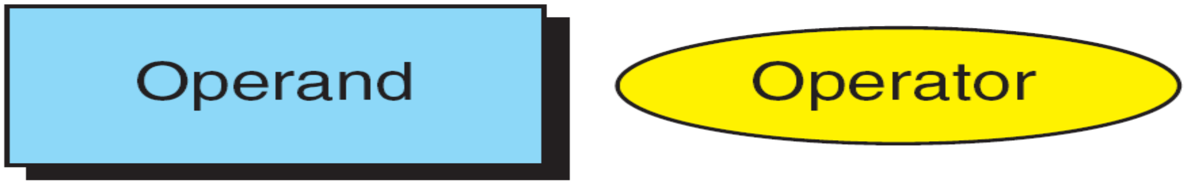
- It is an expression which contains **operand** followed by one **operator**.

  **Example:** a++; a- -;

- The operand in a postfix expression must be a variable.

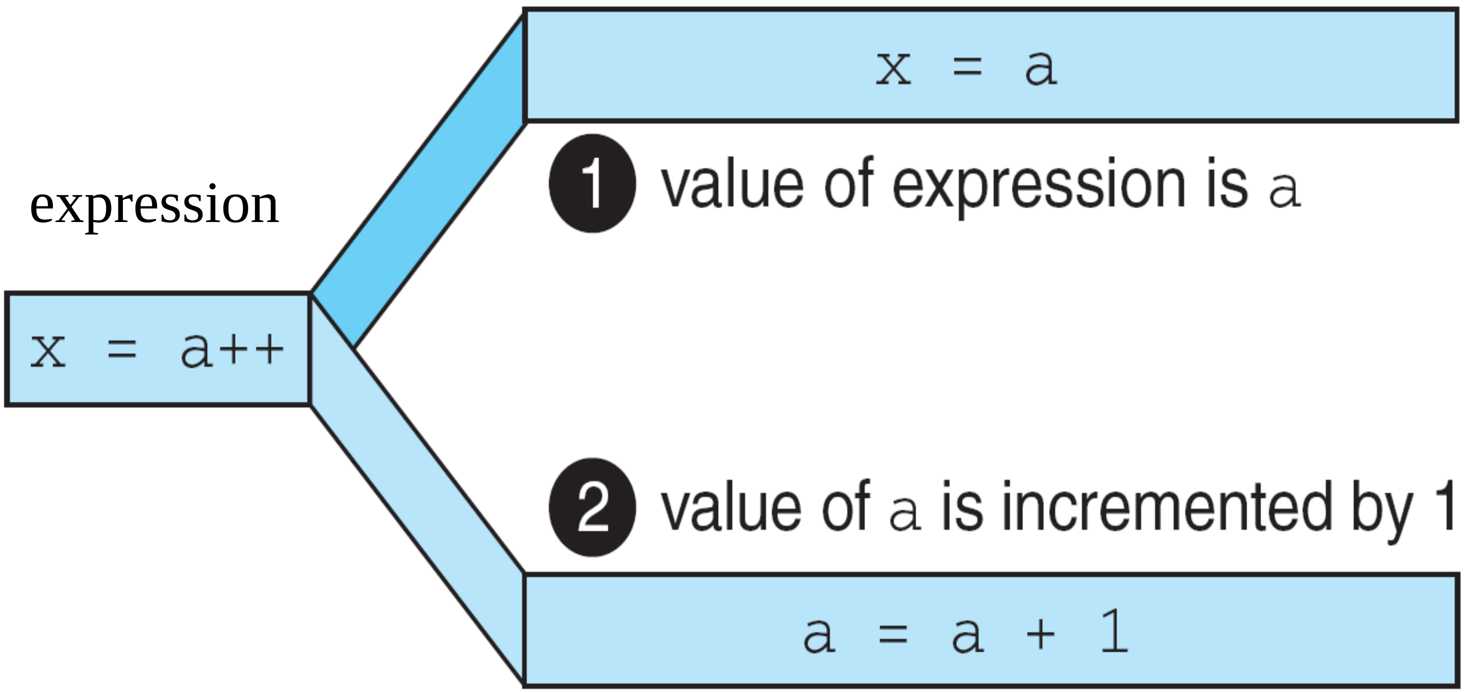
- (a++) has the same effect as (a = a + 1)

- If ++ is after the operand, as in a++, the increment takes place **after** the expression is evaluated.



In the following figure:

1. Value of the variable a is assigned to x
2. Value of the a is incremented by 1.




**Result of Postfix a++**

## Example for Post fix expression

```
#include<stdio.h>
void main()
{
    a=10;
    x=a++;
    printf("x=%d, a=%d",x,a);
}
```

Output:

x=10, a=11

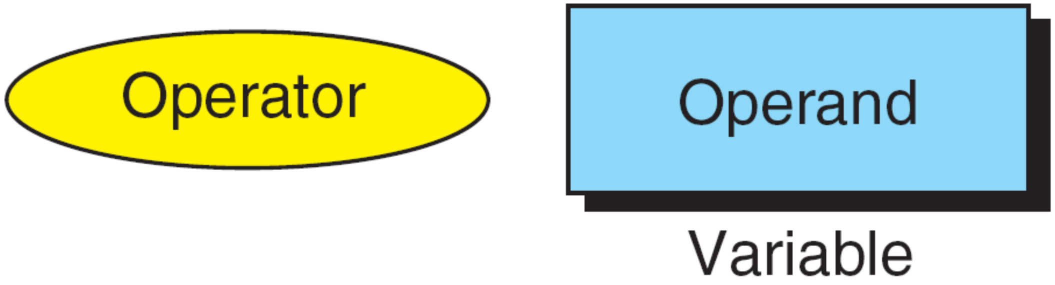
## Pre fix expression:

- It is an expression which contains **operator** followed by an **operand**.

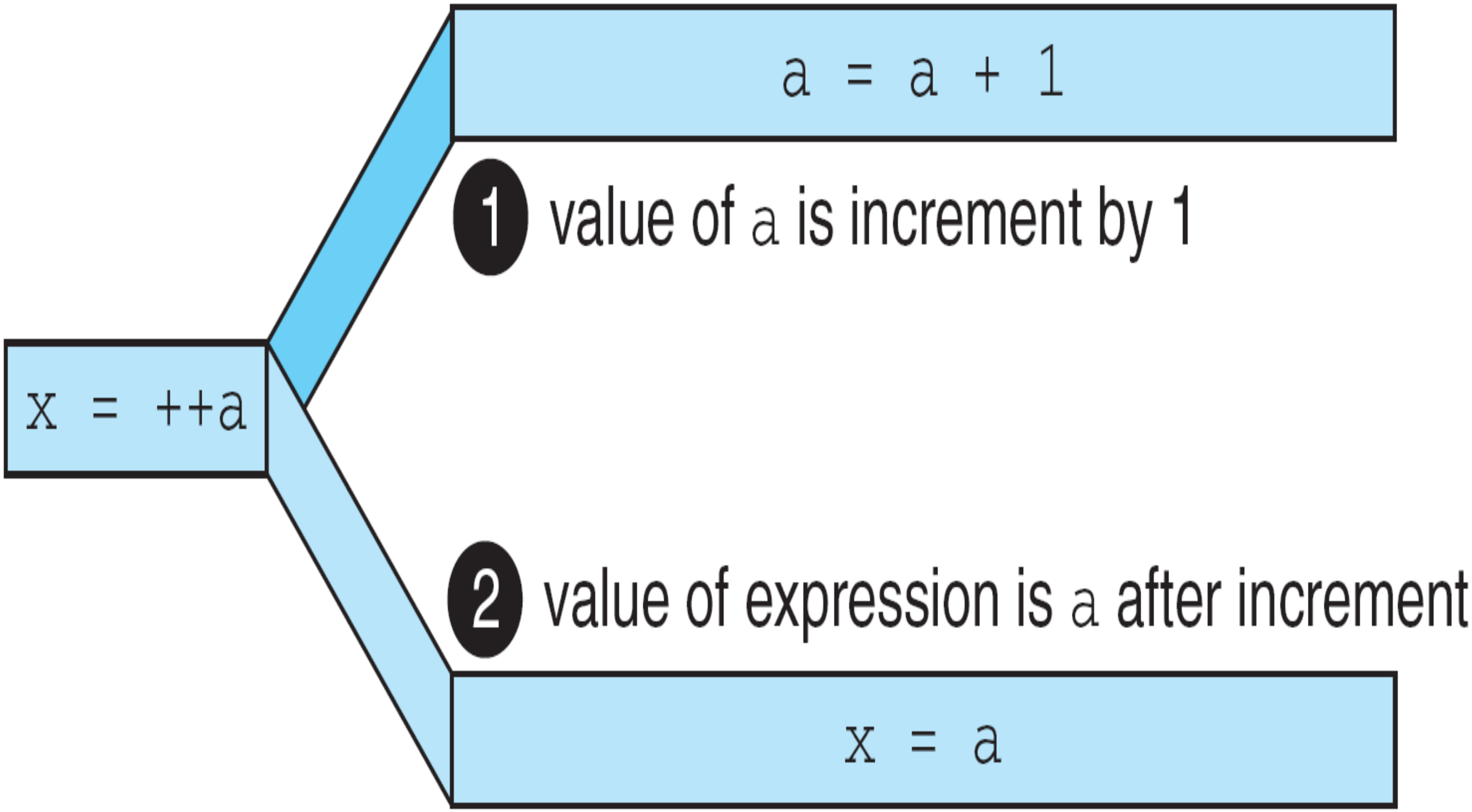
  **Example:** ++a; - -a;

- The operand of a prefix expression must be a variable.
- (++a) has the same effect as (a = a + 1)

- If ++ is before the operand, as in ++a, the increment takes place **before** the expression is evaluated.




**Prefix Expression**




**Result of prefix ++a**

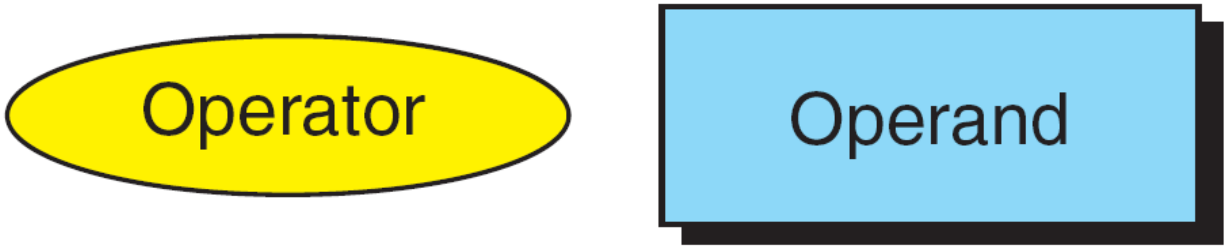
## Example on pre fix expression

```
#include<stdio.h>
void main()
{
      a=10;
      x=++a;
      printf("x=%d, a=%d",x,a);
}
```

Output:

x=11, a=11

**Unary expression:** It is an expression which consists of unary operator followed by the operand




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

**Examples of Unary Plus And Minus Expressions**

## Binary Expressions:

- In binary expression **operator** must be placed in between the **two operands**.

- Both operands of the modulo operator (%) must be integral types.


**Binary Expressions**

The left operand in an **assignment expression** must be a single variable.

| Compound Expression | Equivalent Simple Expression |
|---|---|
| `x *= expression` | `x = x * expression` |
| `x /= expression` | `x = x / expression` |
| `x %= expression` | `x = x % expression` |
| `x += expression` | `x = x + expression` |
| `x -= expression` | `x = x - expression` |

**Expansion of Compound Expressions**

# Demonstration of Compound Assignments (contd…)

```
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
```

# Precedence and Association rules among operators

- **Precedence** is used to determine the order in which different operators in a complex expression are evaluated.

- **Associativity** is used to determine the order in which operators with the same precedence are evaluated in a complex expression.

- Every operator has a precedence.

- The operators which has higher precedence in the expression is evaluated first.

  **Example:** a=8+4*2;

  a=?

# Precedence and Associativity of Operators in C (from higher to lower)

| Category | Operator | Associativity |
|---|---|---|
| Postfix | `() [] -> . ++ --` | Left to right |
| Unary,prefix | `+ - ! ~ ++ -- (type) * & sizeof` | Right to left |
| Multiplicative | `* / %` | Left to right |
| Additive | `+ -` | Left to right |
| Shift | `<< >>` | Left to right |
| Relational | `< <= > >=` | Left to right |
| Equality | `== !=` | Left to right |
| Bitwise AND | `&` | Left to right |
| Bitwise XOR | `^` | Left to right |
| Bitwise OR | `\|` | Left to right |
| Logical AND | `&&` | Left to right |
| Logical OR | `\|\|` | Left to right |
| Conditional | `?:` | Right to left |
| Assignment | `= += -= *= /= %=`<br>`>>= <<= &= ^= \|=` | Right to left |
| Comma | `,` | Left to right |

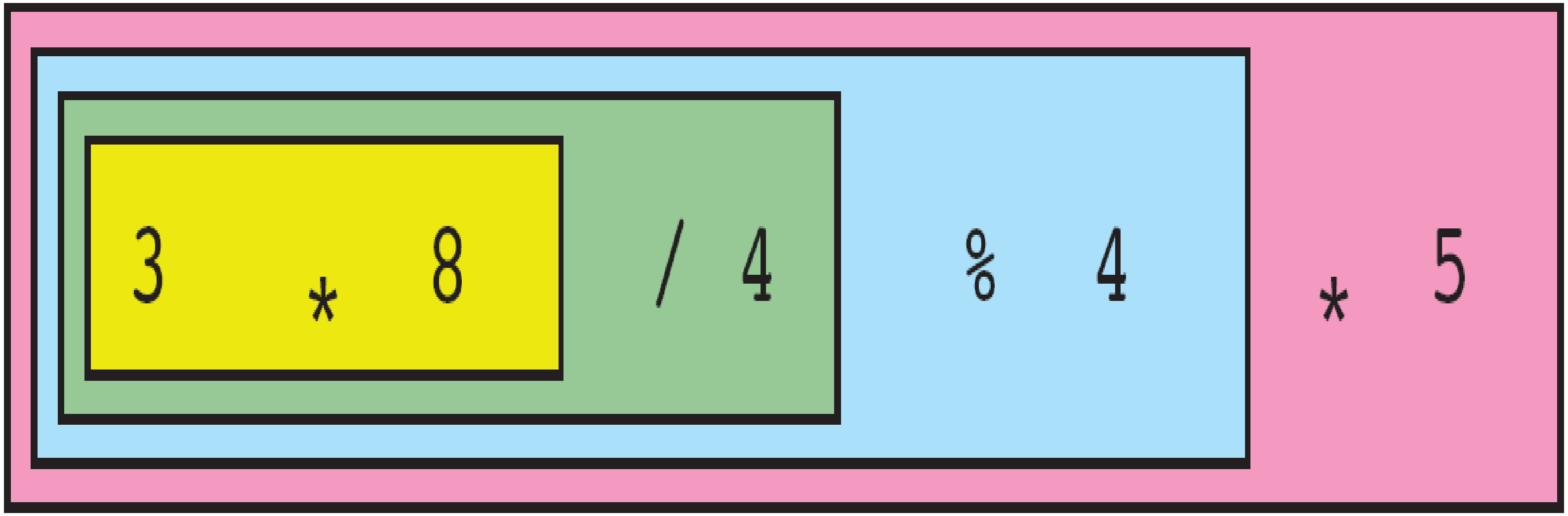
# Example program to illustrate operator precedence

```c
/* Examine the effect of precedence on an expression.
      Written by:

      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int a = 10;
   int b = 20;
   int c = 30;

// Statements
   printf ("a *  b + c  is: %d\n", a *  b + c);
   printf ("a * (b + c) is: %d\n", a * (b + c));
   return 0;
}  // main
```

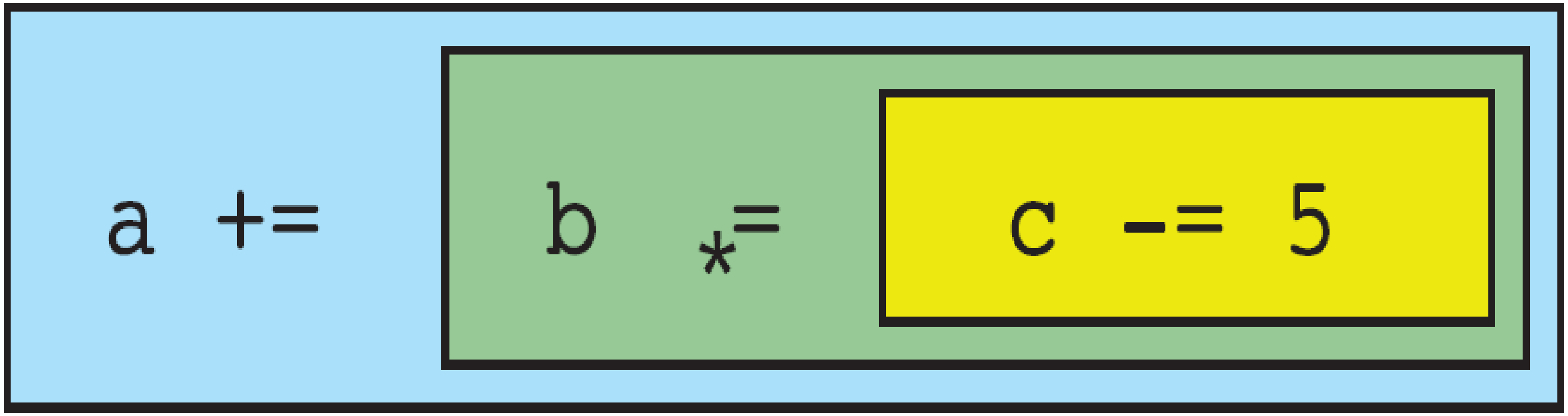
# Example program to illustrate operator precedence (contd…)

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


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




**Left-to-Right Associativity**




**Right-to-Left Associativity**

# Type Conversion

- Up to this point, we have assumed that all of our expressions involved data of the same type.
- But, what happens when we write an expression that involves two different data types, such as multiplying an integer and a floating-point number?
- To perform these evaluations, one of the types must be converted.

- **Type Conversion:** Conversion of one data type to another data type.

- Type conversions are classified into:
    - Implicit Type Conversion
    - Explicit Type Conversion (Cast)

**Implicit Conversion:**

- In implicit type conversion, if the operands of an expression are of different types, the lower data type is automatically converted to the higher data type before the operation evaluation.

- The result of the expression will be of higher data type.
- The final result of an expression is converted to the type of the variable on the LHS of the assignment statement, before assigning the value to it.
- Conversion during assignments:

```
char c = 'a';
int i;
i = c;    /* i  is assigned by the ascii of 'a' */
```

- Arithmetic Conversion: If two operands of a binary operator are not the same type, **implicit** conversion occurs:

```
int i = 5 , j = 1;
float x = 1.0, y;
y = x / i;          /* y = 1.0 /  5.0  */
y = j / i;          /* y = 1 / 5  so  y = 0 */
```

## Explicit Conversion or Type Casting:

- In explicit type conversion, the user has to enforce the compiler to convert one data type to another data type by using typecasting operator.

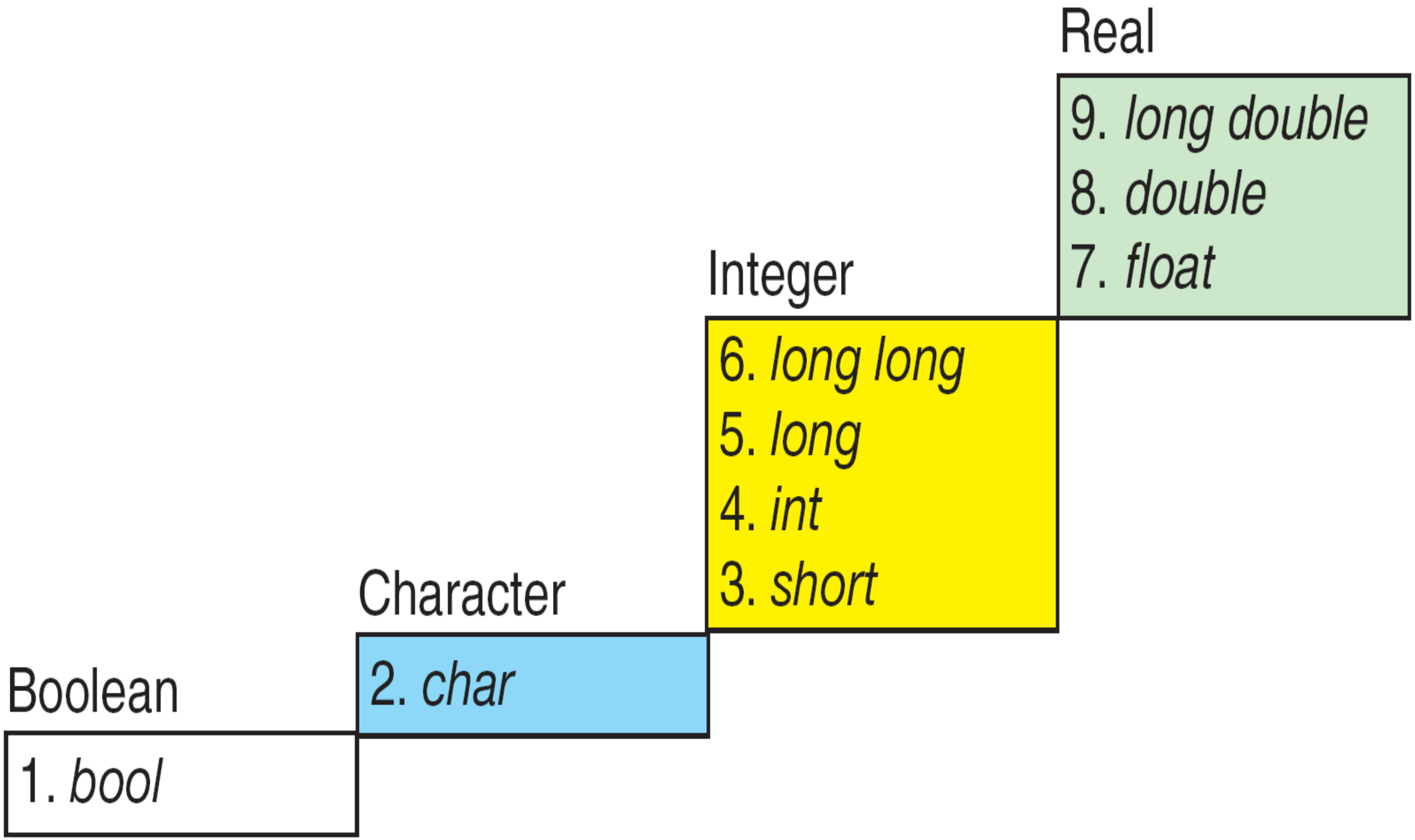
- This method of typecasting is done by prefixing the variable name with the **data type enclosed within parenthesis**.

  **(data type) expression**

- Where (**data type**) can be any valid C data type and expression is any variable, constant or a combination of both.

  **Example:**

  ```
  int x;
  x=(int)7.5;
  ```

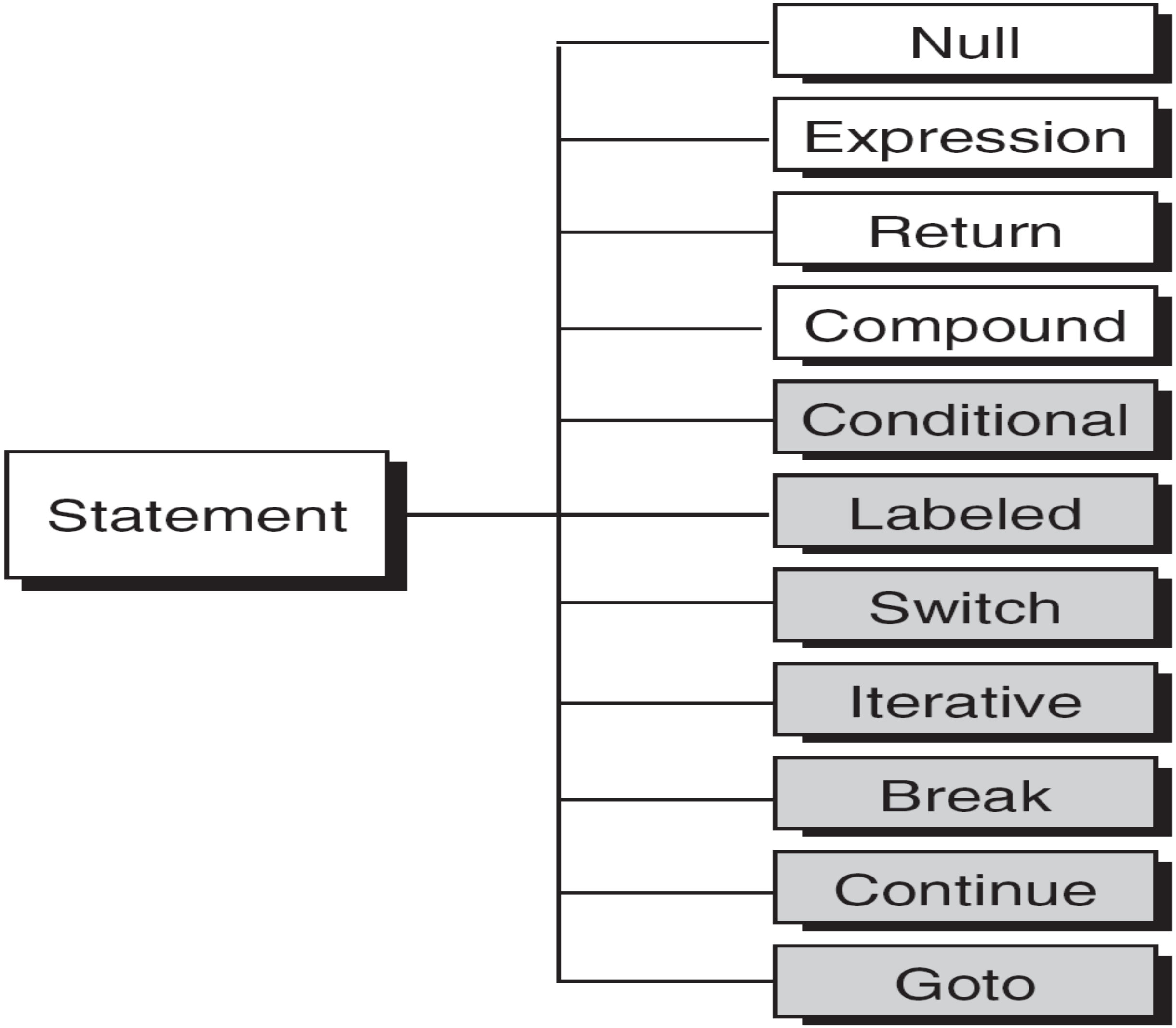



**Conversion Rank (C Promotion Rules)**

```c
//Program to demonstrate type casting
#include<stdio.h>
void main()
{
    char c='z';
    int a=100,b=45;
    double x=100.0, y=45.0;
    double z;

    printf("\n c = %c",c);
    printf("\n a = %d b = %d", a, b);
    printf("\n x = %f y = %f", x, y);
    printf("\n c*a = %d", c*a);

    z=(double)(a/b);
    printf("\n (double)(a/b) = %f", z);

    c=(char)(x/y);
    printf("\n (char)(x/y) = %d", c);
}
```

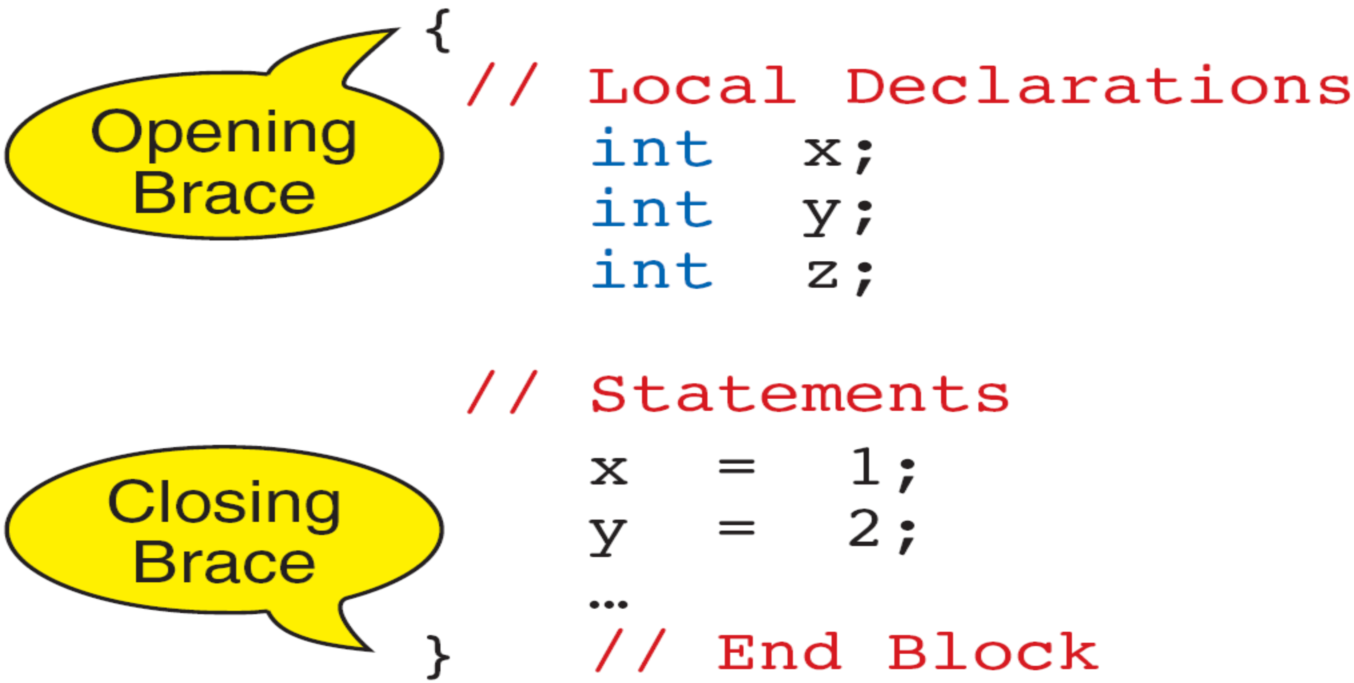
# Statements

- A statement causes an action to be performed by the program.
- It translates directly into one or more executable computer instructions.
- Generally statement is ended with semicolon.
- Most statements need a semicolon at the end; some do not.




**Types of Statements**

- Compound statements are used to group the statements into a single executable unit.
- It consists of one or more individual statements enclosed within the braces { }




**Compound Statement**

## Decision control structures

- if

### Two-way selection

- if else
- nested if
- dangling else

### Multi-way selection

- else if ladder
- switch.

# Conditional Statements

- 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.

**Types of decision control structures:**

- if
- if..else
- nested if…else
- else if ladder
- dangling else
- switch statement

# Conditional Statement: if

The general form of a simple if statement is:

```
if (condition)
{
  statement-block;
}
```

Following are the properties of an if statement:

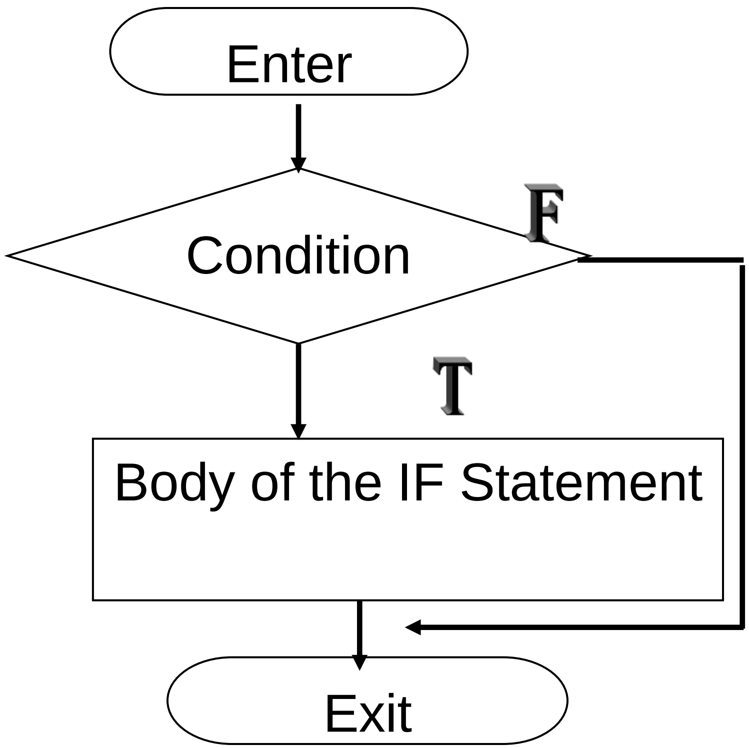
- If the condition is true then the statement-block will be executed.
- If the condition is false it does not do anything ( or the statement is skipped)
- The condition is given in parentheses and must be evaluated as true (nonzero value) or false (zero value).
- If a compound statement is provided, it must be enclosed in opening and closing braces.

# Conditional Statement: if

**Algorithm**

1.Start
2.Declare a,b
3.Read a and b
4.If a>b then
    4.1 print a
5.print If Block
6.stop

**Flow Chart**

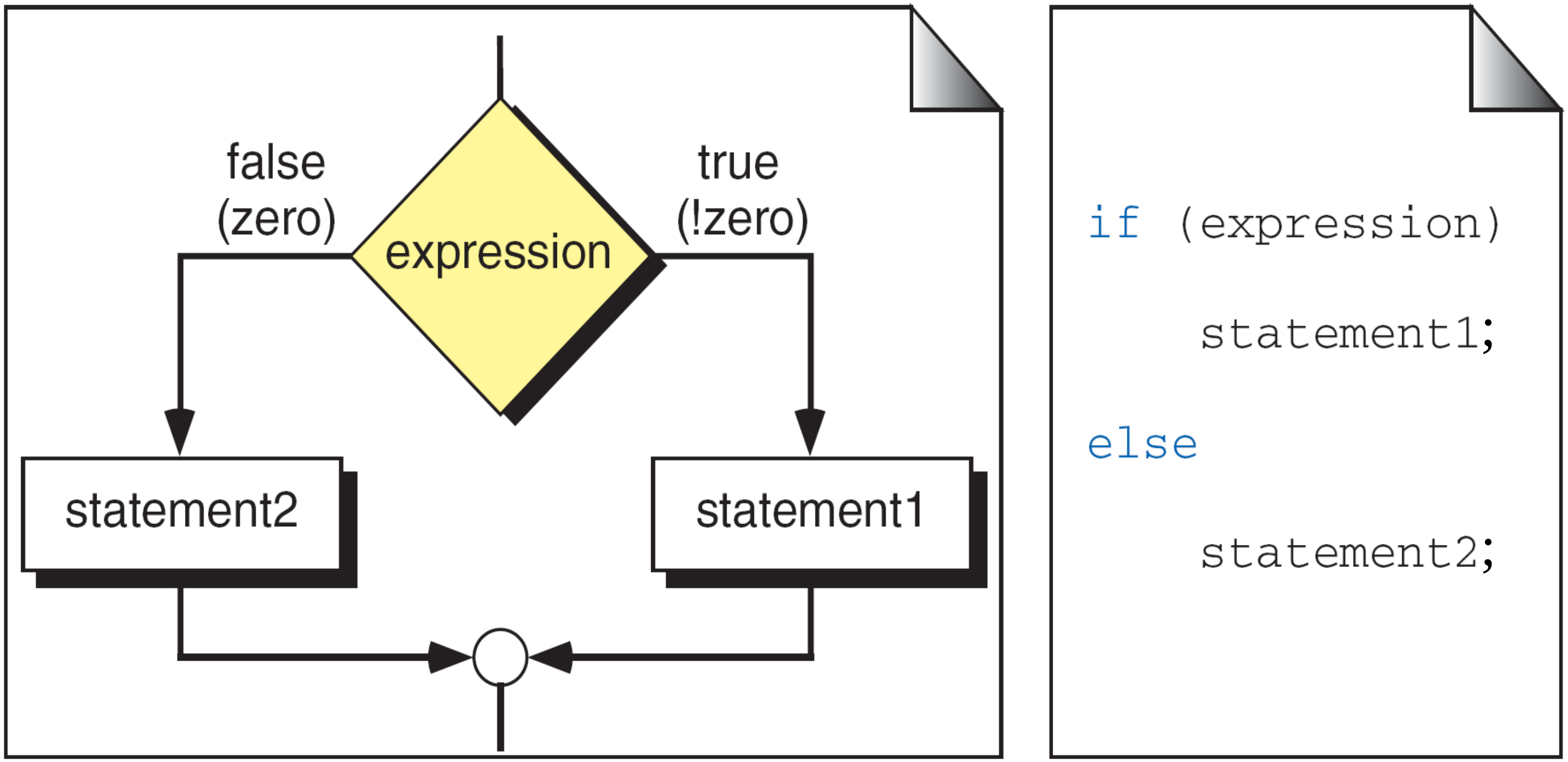



**Program**

**Example:**

```
void main()
{
 int a=10,b=20;
  if(a>b)
 {
      printf(“%d”,a);
  }
      printf(“ IF  BLOCK”);
      }
```

# Conditional Statement: if..else




(a) Logical Flow

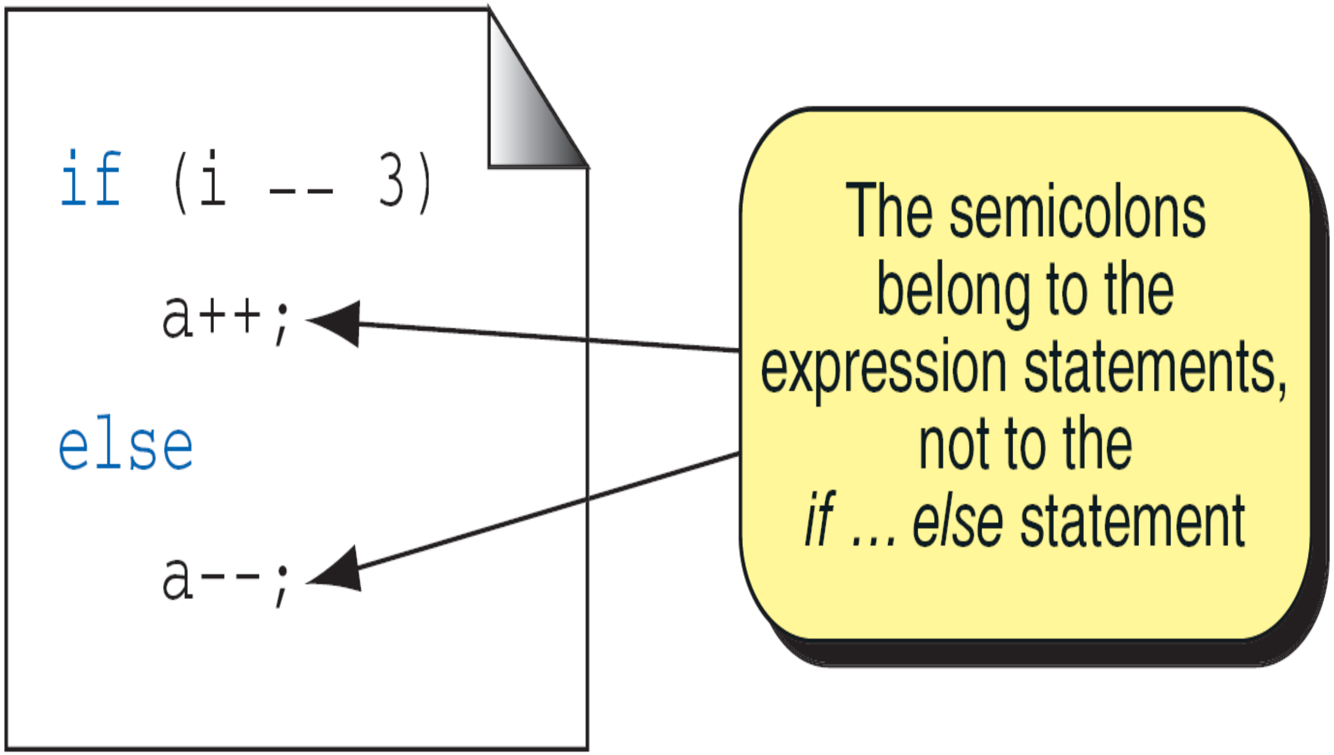
(b) Code

**if...else** Logic Flow

## Syntactical Rules for **if…else** Statements

- The expression or condition which is followed by **if** statement must be enclosed in **parenthesis**.
- No **semicolon** is needed for an **if…else** statement.
- Both the true and false statements can be any statement (even another if…else).
- Multiple statements under **if and else** should be enclosed between curly braces.
- No need to enclose a single statement in curly braces.

# Conditional Statement: if..else




**A Simple if...else Statement**

# Example for Conditional Statement: if..else

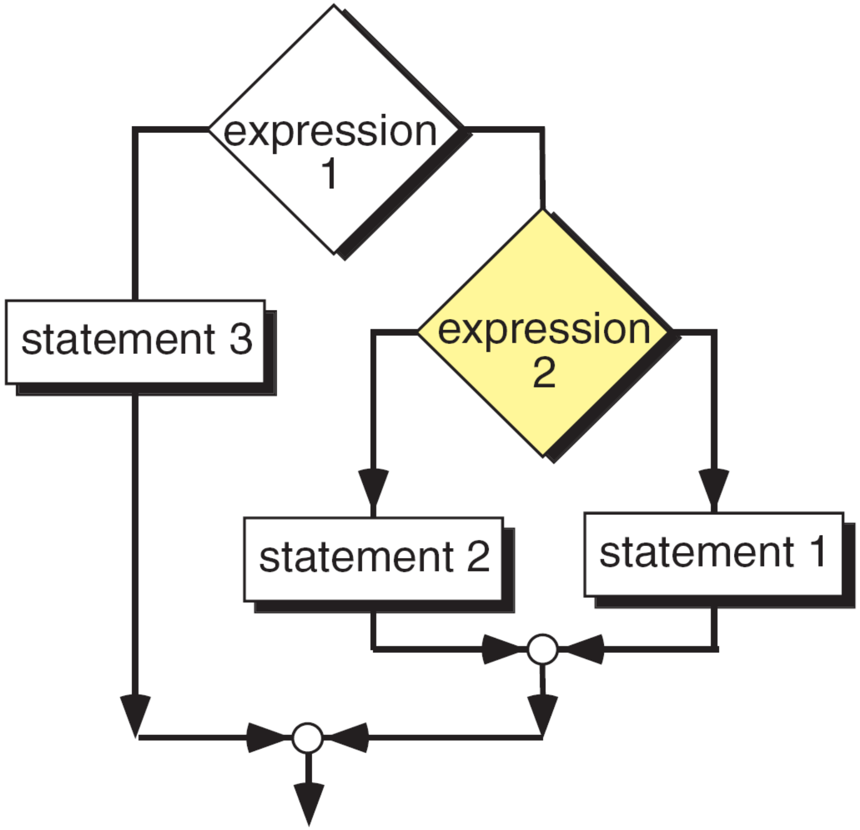
```
/* Two-way selection.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
   // Local Declarations
   int a;
   int b;

   // Statements
   printf("Please enter two integers: ");
   scanf ("%d%d", &a, &b);

```

# Example for Decision Control Statement: if..else

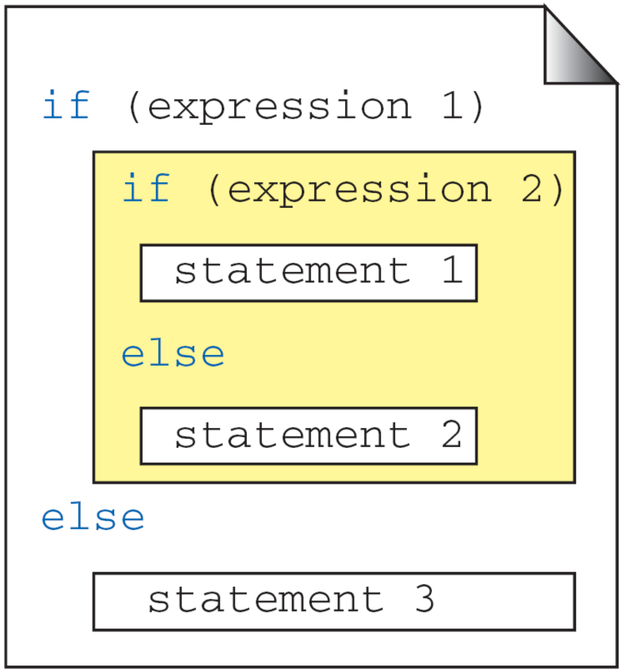
```c
    if (a <= b)
        printf("%d <= %d\n", a, b);
    else
        printf("%d > %d\n", a, b);

    return 0;
} // main
```

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

# Conditional Statement: nested if…else




(a) Logic flow




(b) Code

- Nested if…else means within the **if…else** you can include **another if…else** either in **if block or else block**.

Nested **if…else** Statements

# Conditional Statement: nested if…else

```c
/* Nested if in two-way selection.
      Written by:
      Date:
*/
#include <stdio.h>

int main (void)
{
// Local Declarations
   int a;
   int b;

// Statements
   printf("Please enter two integers: ");
   scanf ("%d%d", &a, &b);

```

# Conditional Statement: nested if…else

```c
    if (a <= b)
        if (a < b)
            printf("%d < %d\n", a, b);
        else
            printf("%d == %d\n", a, b);
    else
        printf("%d > %d\n", a, b);

    return 0;
}   // main
```

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

# Conditional Statement: else if

```
if (condition1)
        statements1;
else if (condition2)
                statements2;
        else if (condition3)
                        statements3;
                else if (condition4)
                                statements4;
                        ……
                        else if(conditionn)
                                        statementsn;
                                else
                                                default_statement;
statement x;
```

- The conditions are evaluated from the top to down.
- As soon as a true condition is found the statement associated with it is executed and the control is transferred to the statementx by skipping the rest of the ladder.
- When all **n** conditions become false,final else containing default_statement that will be executed

# Example program for nested if…else

```
main()
{
float m1,m2,m3,m4,m5,m6,per;
printf("Enter marks\n");
scanf("%f%f%f%f%f%f",&m1,&m2,&m3,&m4,&m5,&m6);
per=(m1+m2+m3+m4+m5+m6)/6;
if(per>=70)
        printf("\nDistinction");
else {
        if(per<70 && per>=60)
                printf("\nFirst Class");
        else {
                if(per<60 && per>=50)
                        printf("\nSecond Class");
                else {
                        if(per<50 && per>=40)
                                printf("\nThird Class");
                        else
                                printf("\nFail");
                }//else
        }//else
}//else
}//main
```

# Example program for if

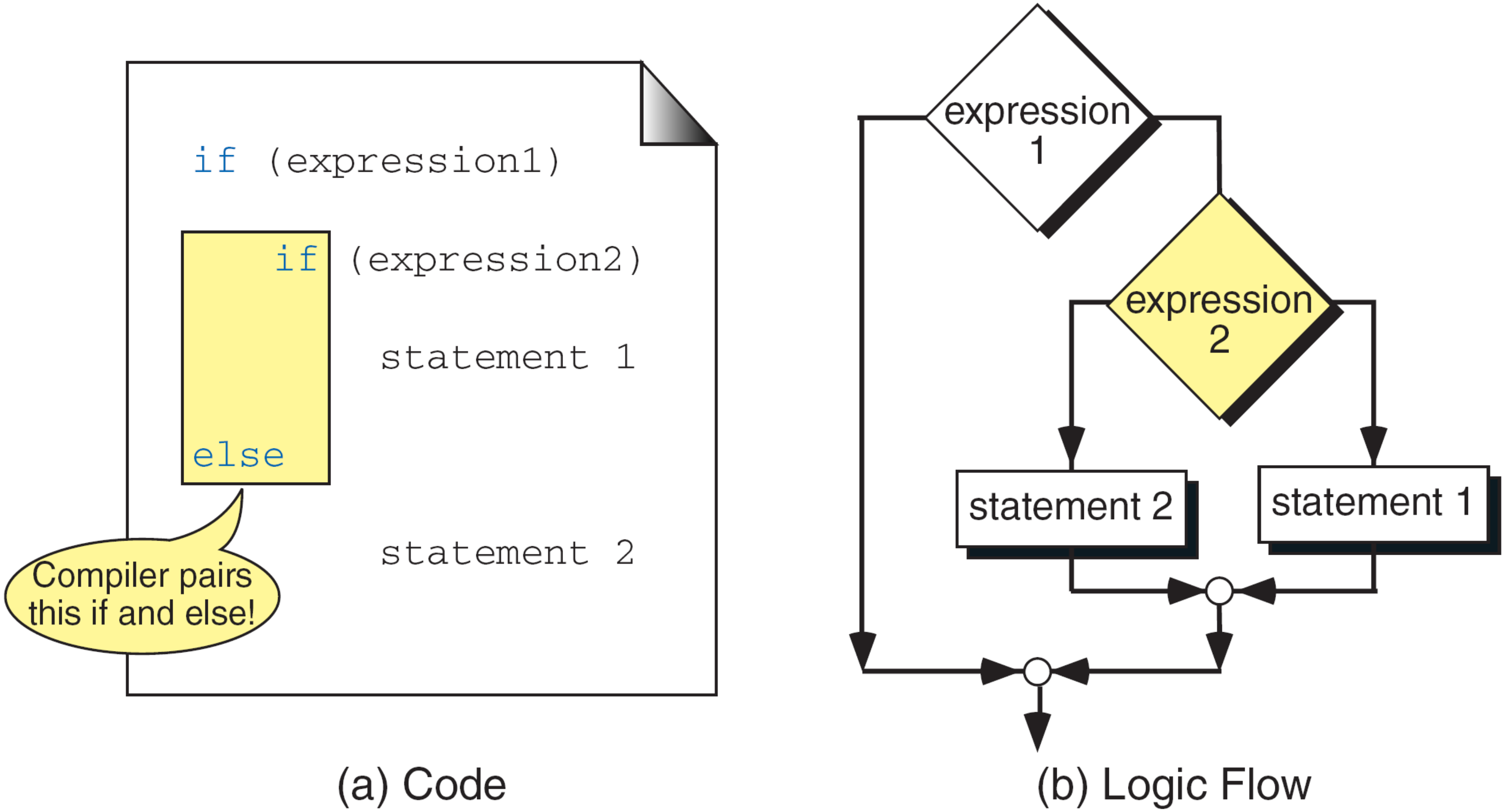
```
main()
{
        float m1,m2,m3,m4;
        float per;
        printf("enter marks\n");
        scanf("%f%f%f%f",&m1,&m2,&m3,&m4);
        per=(m1+m2+m3+m4)/4;
        if(per>=70)
                printf("\nDistinction");
        if(per<70 && per>=60)
                printf("\nFirst Class");
        if(per<60 && per>=50)
                printf("\nSecond Class");
         if(per<50 && per>=40)
                printf("\nThird Class");
        else
                printf("\nFail");
}//main
```

## Example program for else if ladder

```
main()
{
        float m1,m2,m3,m4;
        float per;
        printf("enter marks\n");
        scanf("%f%f%f%f",&m1,&m2,&m3,&m4);
        per=(m1+m2+m3+m4)/4;
        if(per>=70)
                printf("\nDistinction");
        else  if(per<70 && per>=60)
                printf("\nFirst Class");
        else if(per<60 && per>=50)
                printf("\nSecond Class");
        else  if(per<50 && per>=40)
                printf("\nThird Class");
        else
                printf("\nFail");
}//main
```

# Dangling else

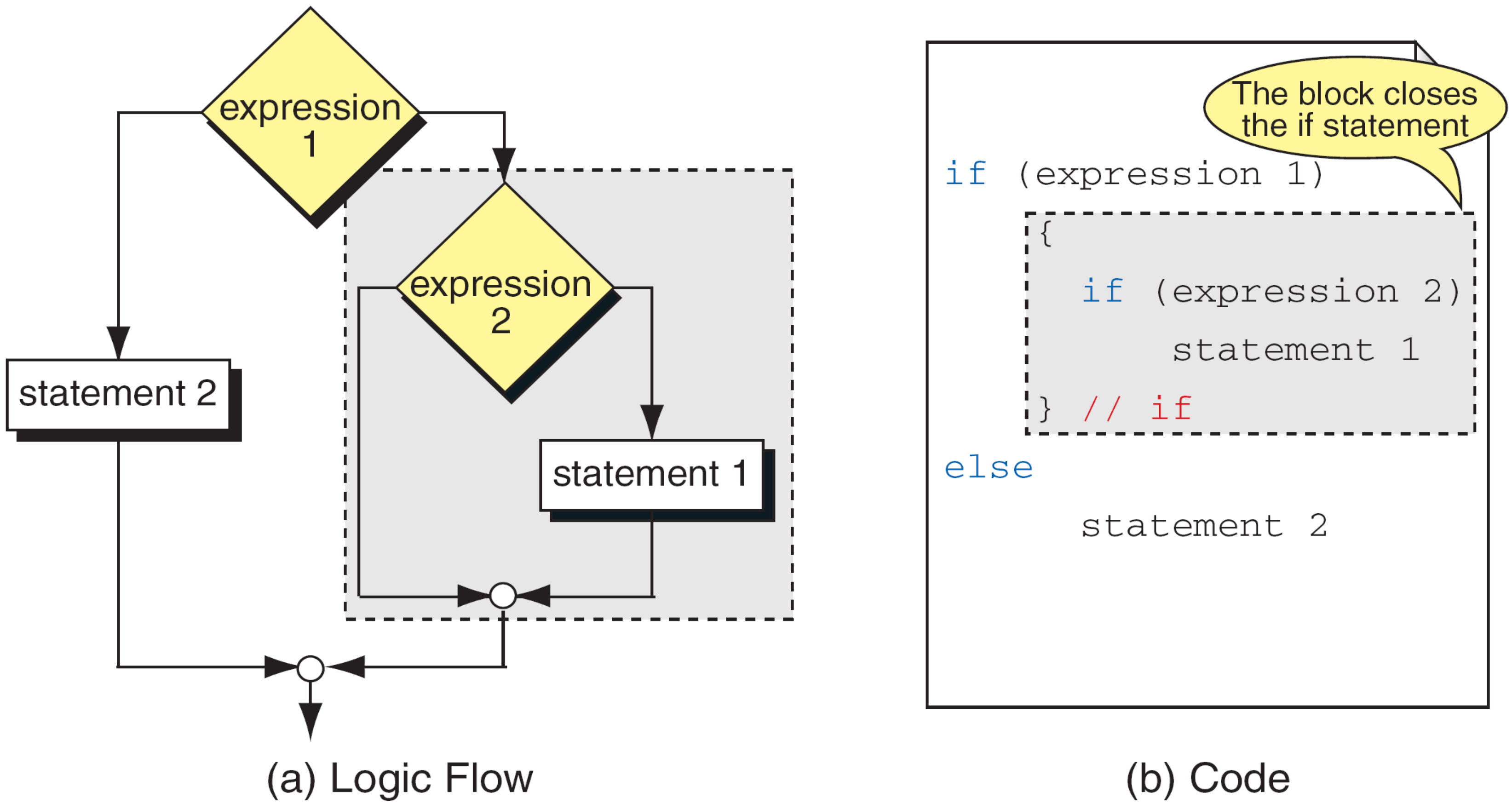
- **else** is always paired with the most recent unpaired **if**.




**Dangling *else***

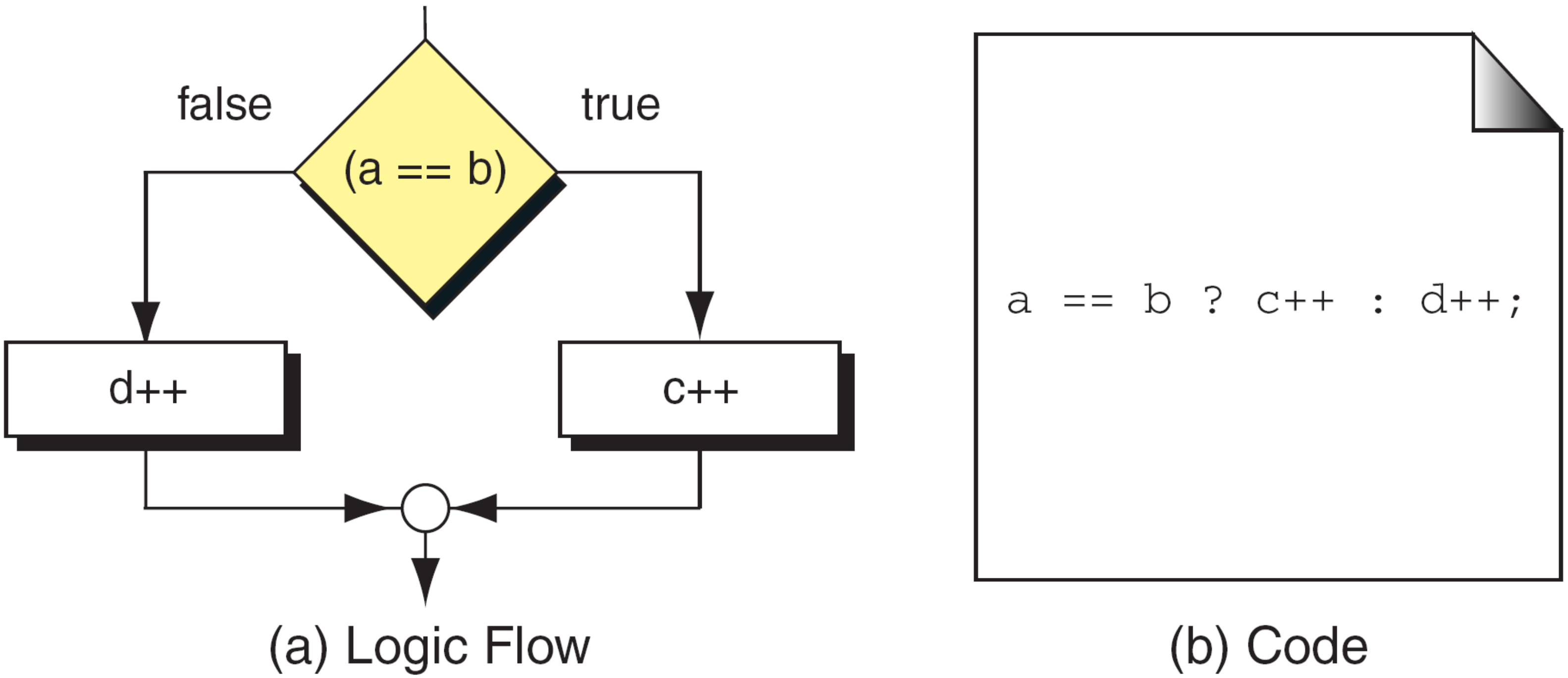
# Dangling else (contd…)

- To avoid **dangling else** problem place the **inner if statement** with in the **curly braces**.




Dangling **else** Solution

- A simple **if…else** can be represented using the conditional (ternary) expression.



# Conditional Statement: switch

- It is a **multi-way** conditional statement generalizing the **if…else** statement.
- It is a conditional control statement that allows some particular **group of statements to be chosen** from several available groups.
- A switch statement allows a single variable to be compared with several possible **case** labels, which are represented by constant values.
- If the variable matches with one of the constants, then an execution jump is made to that point.
- A case label cannot appear more than once and there can only be one default expression.

# Conditional Statement: switch

- Note: **switch** statement does not allow less than ( < ), greater than ( > ).
- ONLY the equality operator (==) is used with a switch statement.
- The control variable must be integral (int or char) only.
- When the switch statement is encountered, the control variable is evaluated.
- Then, if that evaluated value is equal to any of the values specified in a **case** clause, the statements immediately following the colon (“:”) begin to run.
- **Default case** is optional and if specified, default statements will be executed, if there is no match for the case labels.
- Once the program flow enters a **case** label, the statements associated with **case** have been executed, the program flow continues with the statement for the next case. (if there is no **break** statement after case label.)

# switch

**General format of switch:**

```
switch (expression)
   {
    case constant-1:  statement
                      :
                      statement
    case constant-2:  statement
                      :
                      statement
    case constant-n:  statement
                      :
                      statement
    default        :  statement
                      :
                      statement
   }
```

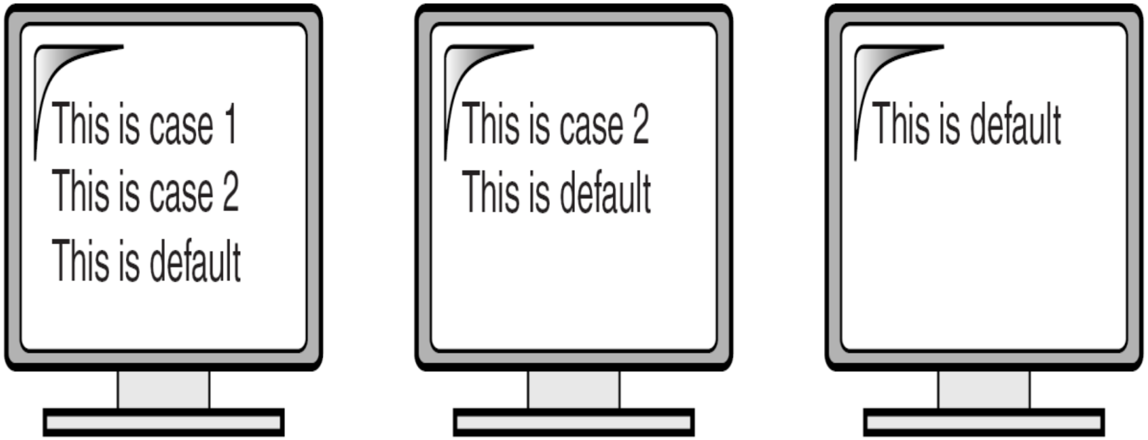
# switch

- The following results are possible, depending on the value of printFlag.
- If printFlag is 1, then all three printf statements are executed.
- If printFlag is 2, then the first print statement is skipped and the last two are executed.
- Finally, if printFlag is neither 1 nor 2, then only the statement defined by the default is executed.

# switch

**Example1 for switch statement:**

```
switch (printFlag)
    {
     case 1:  printf("This is case 1\n");

     case 2:  printf("This is case 2\n");

     default: printf("This is default\n");
    }
```




(a) printFlag is 1 (b) printFlag is 2 (c) printFlag is not 1 or 2

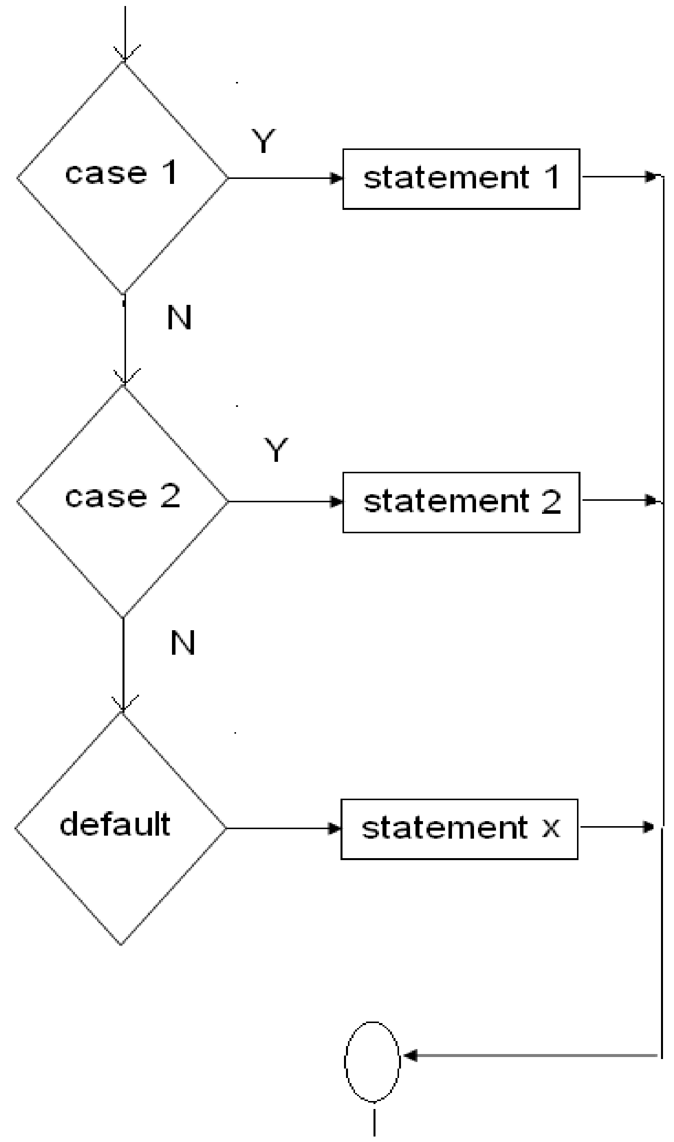
# Decision Control Statement: switch

- you can put the cases in any order, need not be in ascending or descending order
- You are also allowed to use char values in case
- You can execute common set of statements for multiple case
- If there are multiple statements to be executed in each case,thereis no need to enclose them within a pair of braces.
- Every statement in a switch must belong to some case. If statement doesn't belong to any case, the compiler won't report any error. The statement would never get executed.
- If you want to execute only one case-label, C provides break statement.
- It causes the program to jump out of the switch statement, that is go to the closing braces (}) and continues the remaining code of the program.
- If we add break to the last statement of the case, the general form of switch case is as follows:

# Decision Control Statement: switch

**General format of switch:**

```
switch (expression)
  {
   case constant-1: statement
                        :
                      break;
   case constant-2: statement
                        :
                      break;
   case constant-n: statement
                        :
                      break;
   default        : statement
                        :
                      break;
  }
```



# Decision Control Statement: switch

## Example2 for switch statement:

```c
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
```


(a) printFlag is 1    (b) printFlag is 2    (c) printFlag is not 1 or 2

# switch versus if- else ladder

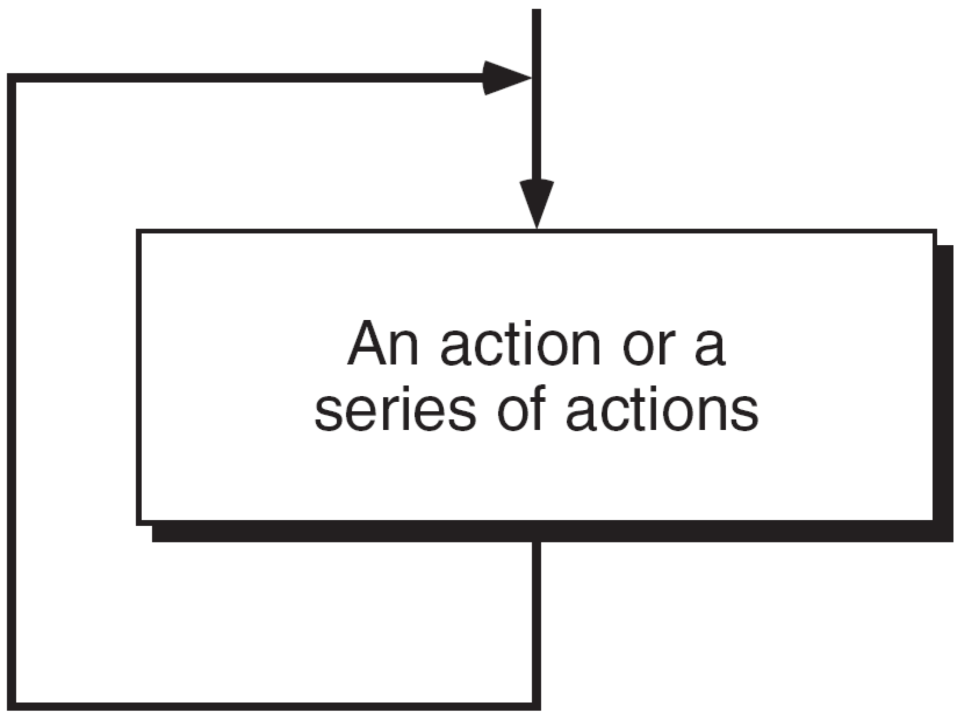
| Switch | ELSE -IF |
| --- | --- |
| 1. A float expressions can not be tested | 1. A float expressions can be tested |
| 2. Cases can never have variable expressions | 2. If-else can have variable expressions |
| 3. Multiple cases can not use same expressions. | 3. There is no such limitation |
| 4. Use of break statement is essential | 4. There is no need of break statement |
| 5. As per the value of switch ,the control jumps to corresponding case | 5. The control goes through the every else if statement until it finds true value of statement or end of else if ladder |
| 6. Faster | 6. slower |

# Repetitive control structures

- pre-test and post-test loops
- initialization and updation
- while
- do while
- for loop
- nested loops.

# Concept of a 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.


An action or a
series of actions

# 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.

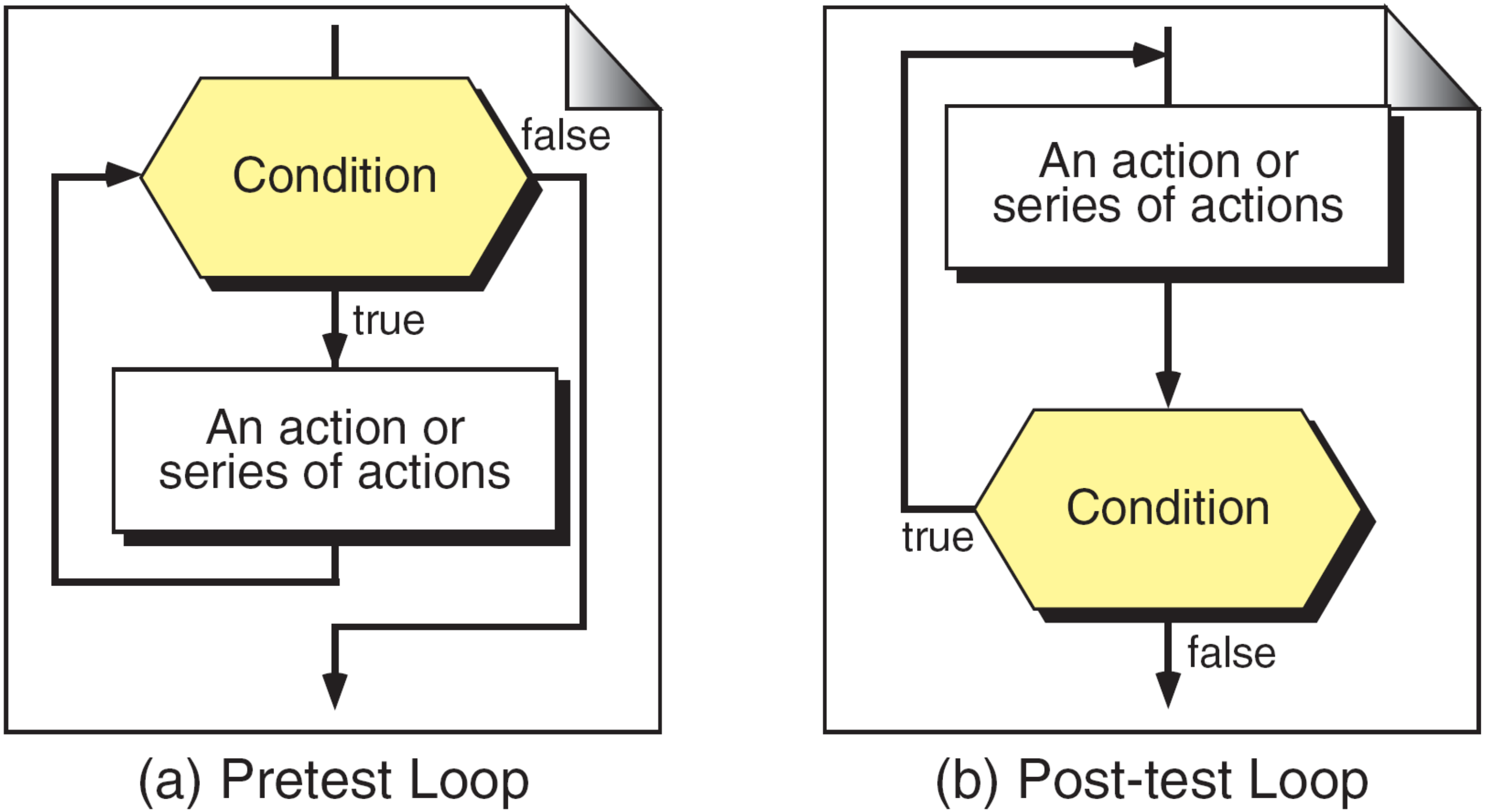
# Note

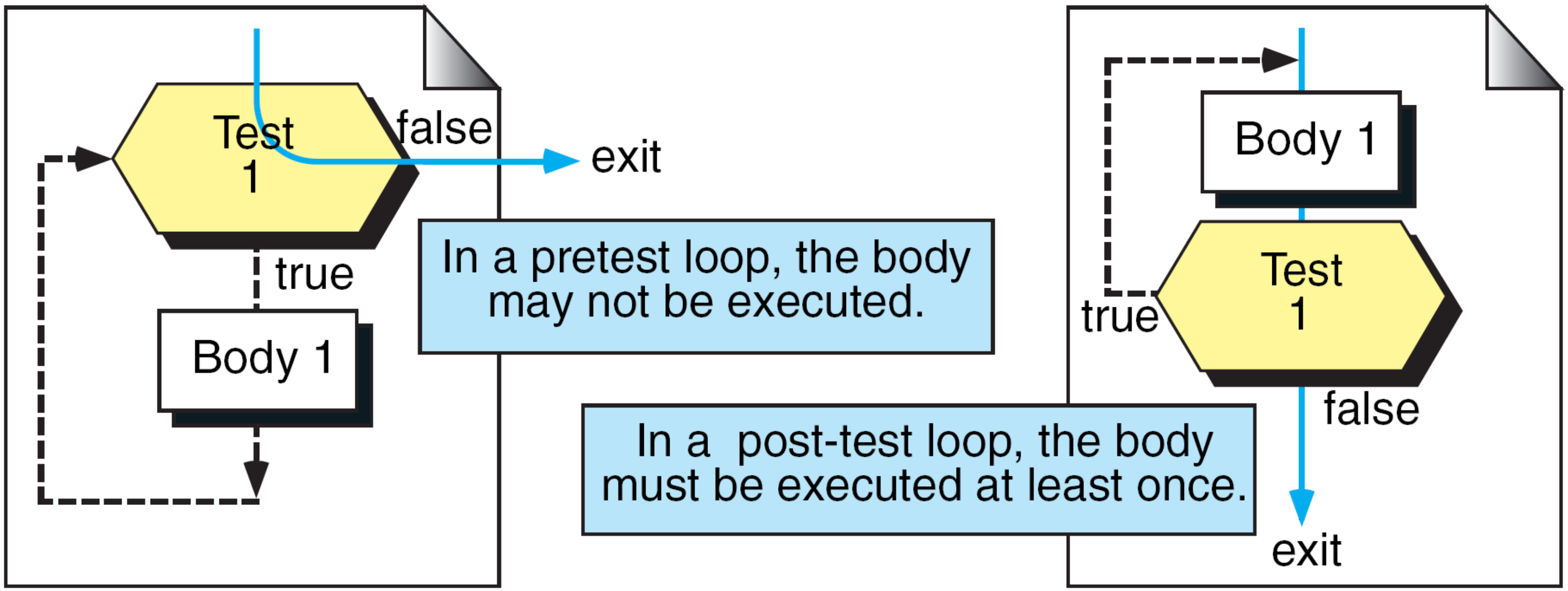
Pretest Loop

In each iteration, the control expression is tested first. If it is true, the loop continues; otherwise, the loop is terminated.

Post-test Loop

In each iteration, the loop action(s) are executed. Then the control expression is tested. If it is true, a new iteration is started; otherwise, the loop terminates.


Condition
false
true
An action or series of actions
(a) Pretest Loop
An action or series of actions
Condition
true
false
(b) Post-test Loop



(a) Pretest

(b) Post-test

# Initialization and Updating

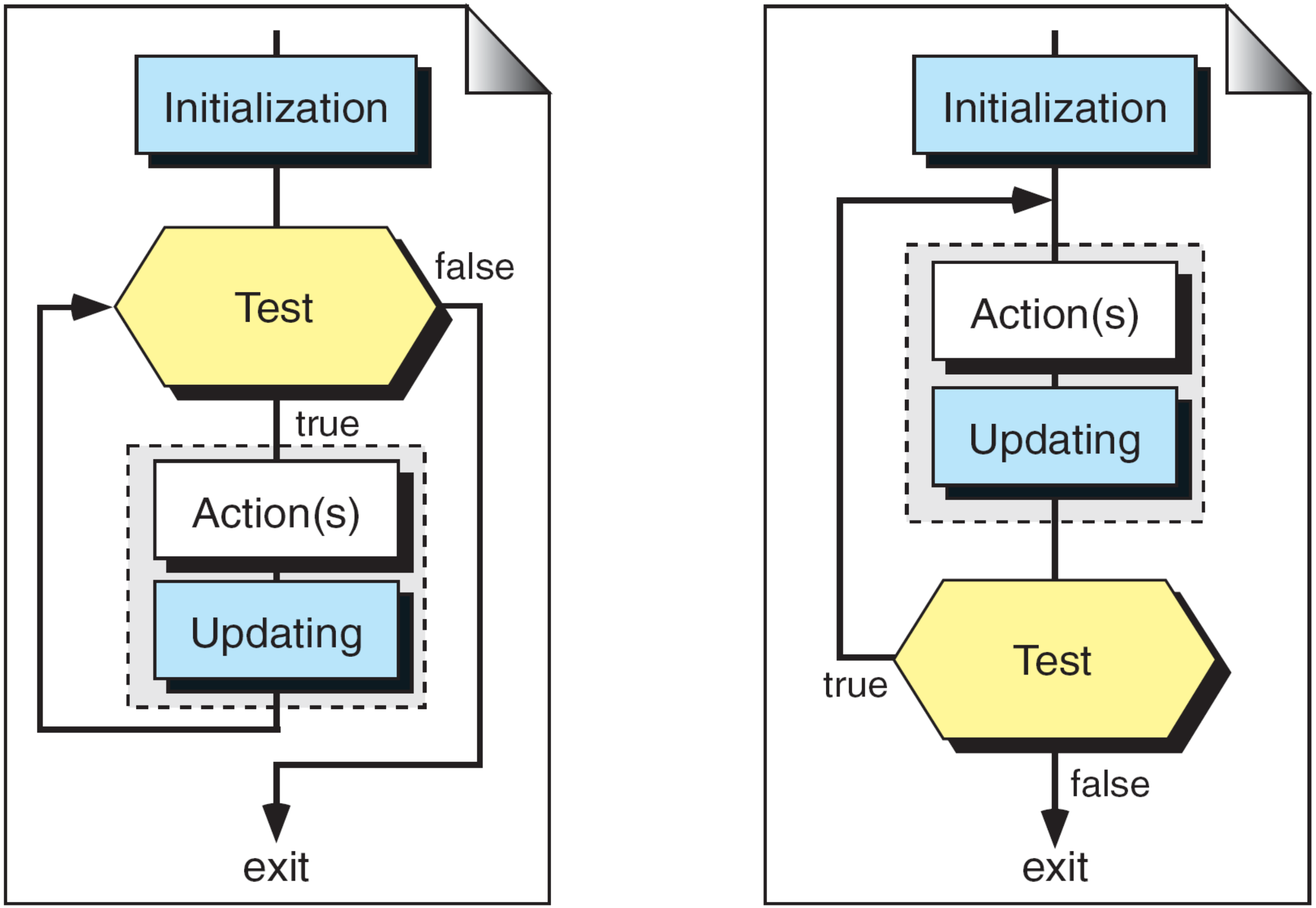
- In addition to the loop control expression, two other processes, initialization and updating, are associated with almost all loops.

  - Loop Initialization
  - Loop Update

- Control expression is used to decide whether the loop should be executed or terminated.

- Initialization is place where you can assign some value to a variable.

- Variable's value can be updated by incrementing a value by some



(a) Pretest Loop (b) Post-test Loop

| Pretest Loop | | Post-test Loop | |
|---|---|---|---|
| Initialization: | 1 | Initialization: | 1 |
| Number of tests: | $n + 1$ | Number of tests: | n |
| Action executed: | n | Action executed: | n |
| Updating executed: | n | Updating executed: | n |
| Minimum iterations: | 0 | Minimum iterations: | 1 |

# Loops in C

- C has three loop statements: the while, the for, and the do…while. The first two are pretest loops, and the third is a post-test loop.

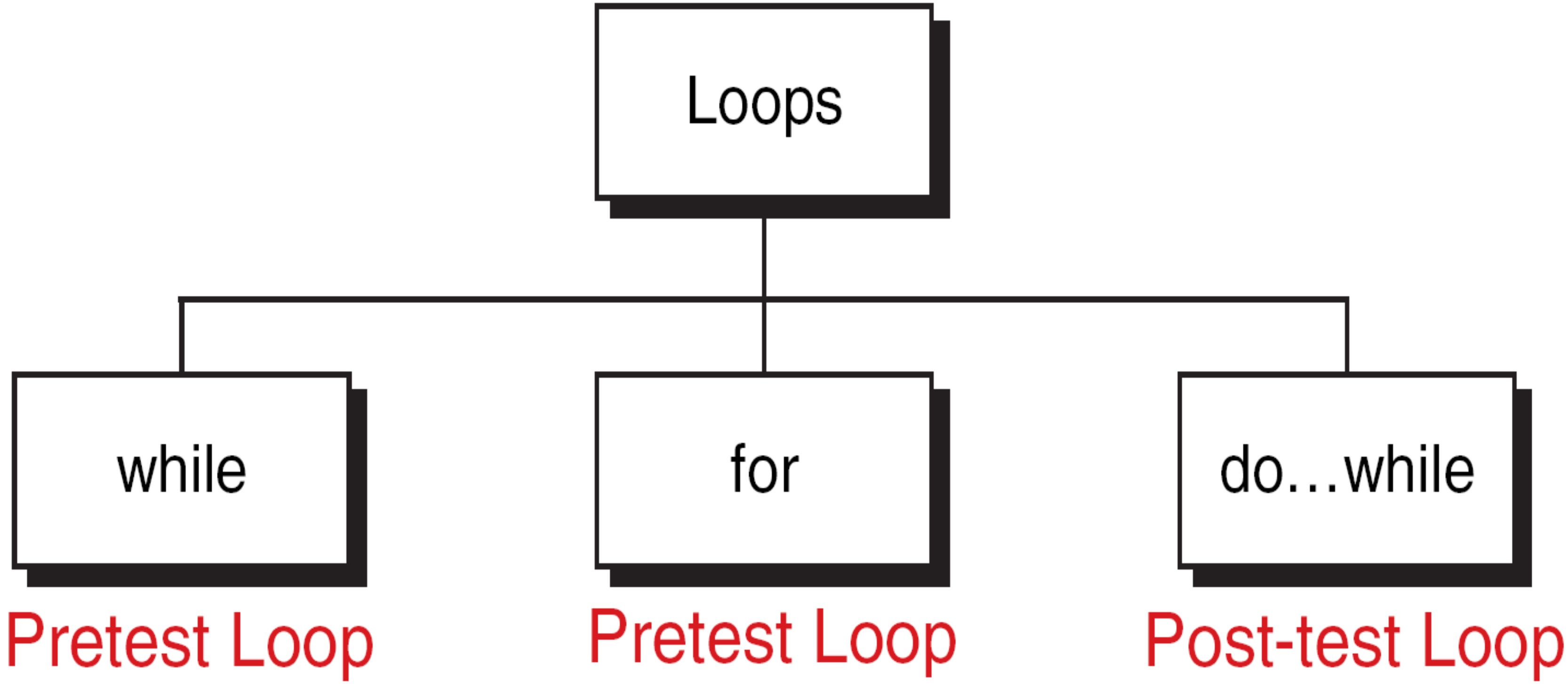
- We can use all of them for event-controlled and counter-controlled loops.

- Before a loop start, the loop control variable must be initialized; this should be done before the first execution of loop body.

- Test for the specified condition for execution of the loop, known as loop control expression.

- Executing the body of the loop, known as actions.

- Updating the loop control variable for performing next condition checking.



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

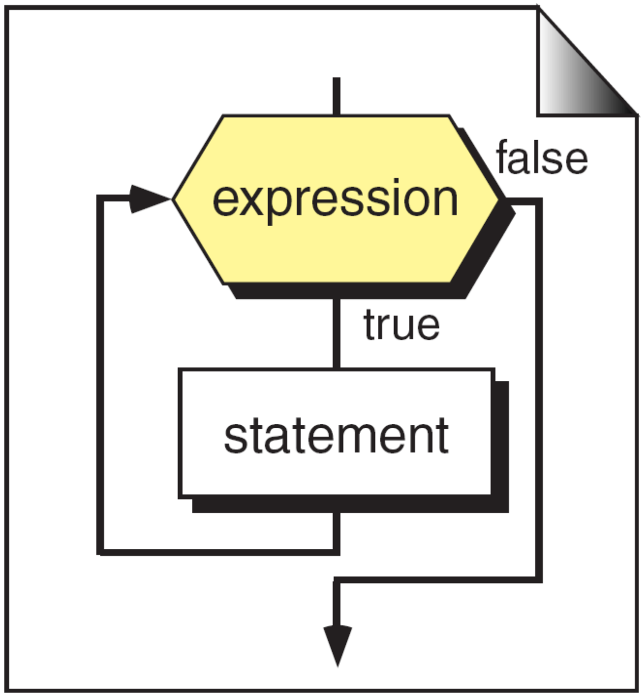
# while

- The "while" loop is a generalized looping structure that employs a variable or expression for testing the condition.

- It is a repetition statement that allows an action to be repeated while some conditions remain true.

- The body of while statement can be a single statement or compound statements.

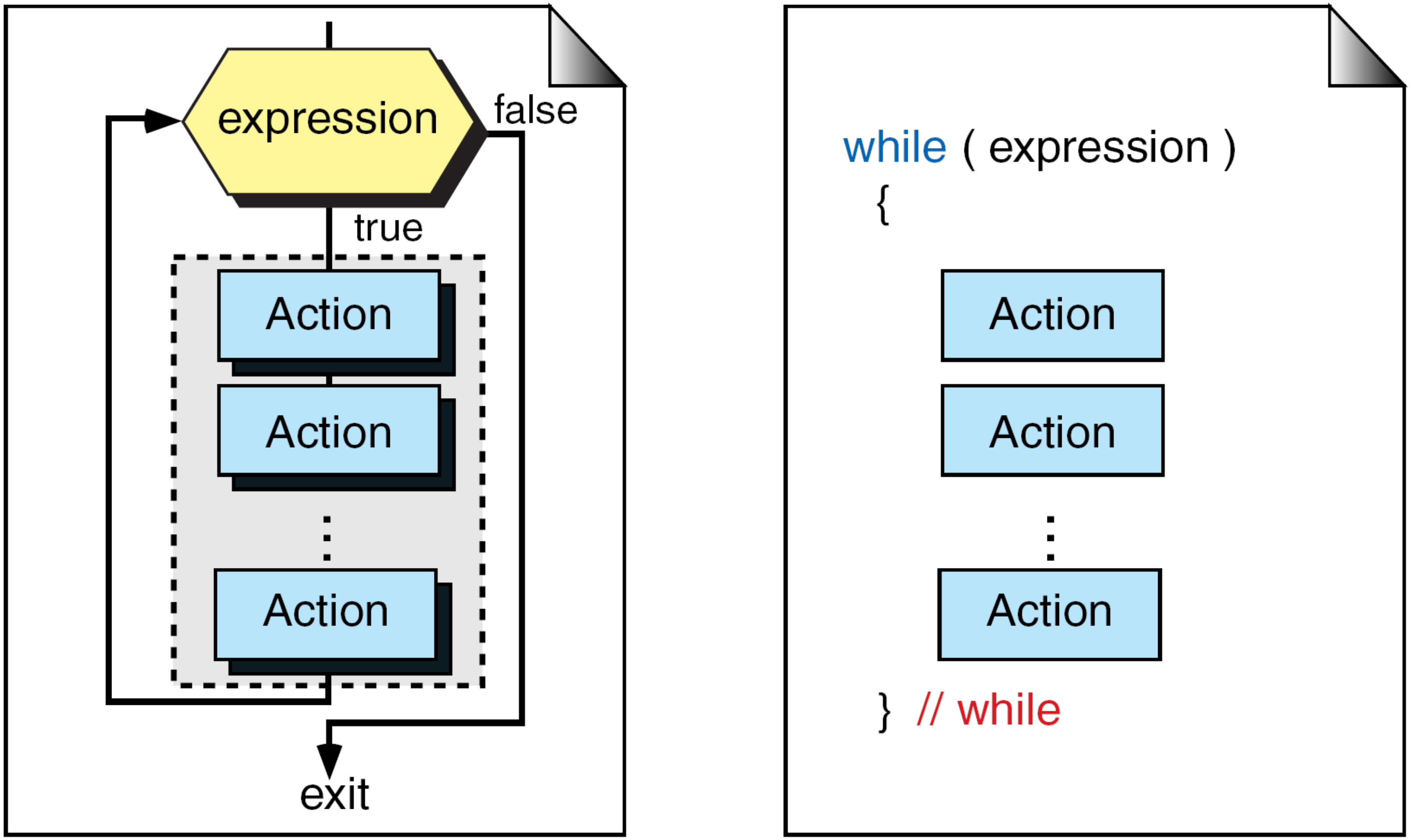
- It doesn’t perform even a single operation if condition fails.




(a) Flowchart

```
while (expression)

    statement
```

(b) Sample Code




(a) Flowchart (b) C Language

## Example 1: To print 1 to 10 natural numbers

```
#include<stdio.h>
main()
{
    int i;
    i=1;
    while (i<=10)
    {
        printf("%d\n",i);
        i++;
    }
}
```

Example 2: To print the reverse of the given number.

```
void main()
{
    int n, rem, rev = 0;
    printf("\n Enter a positive number: ");
    scanf("%d",&n);
    while(n != 0)
    {
        rem = n%10;
        rev = rev*10+rem;
        n = n/10;
    }
    printf("The reverese of %d is %d",n,rev);
}
```

# do while

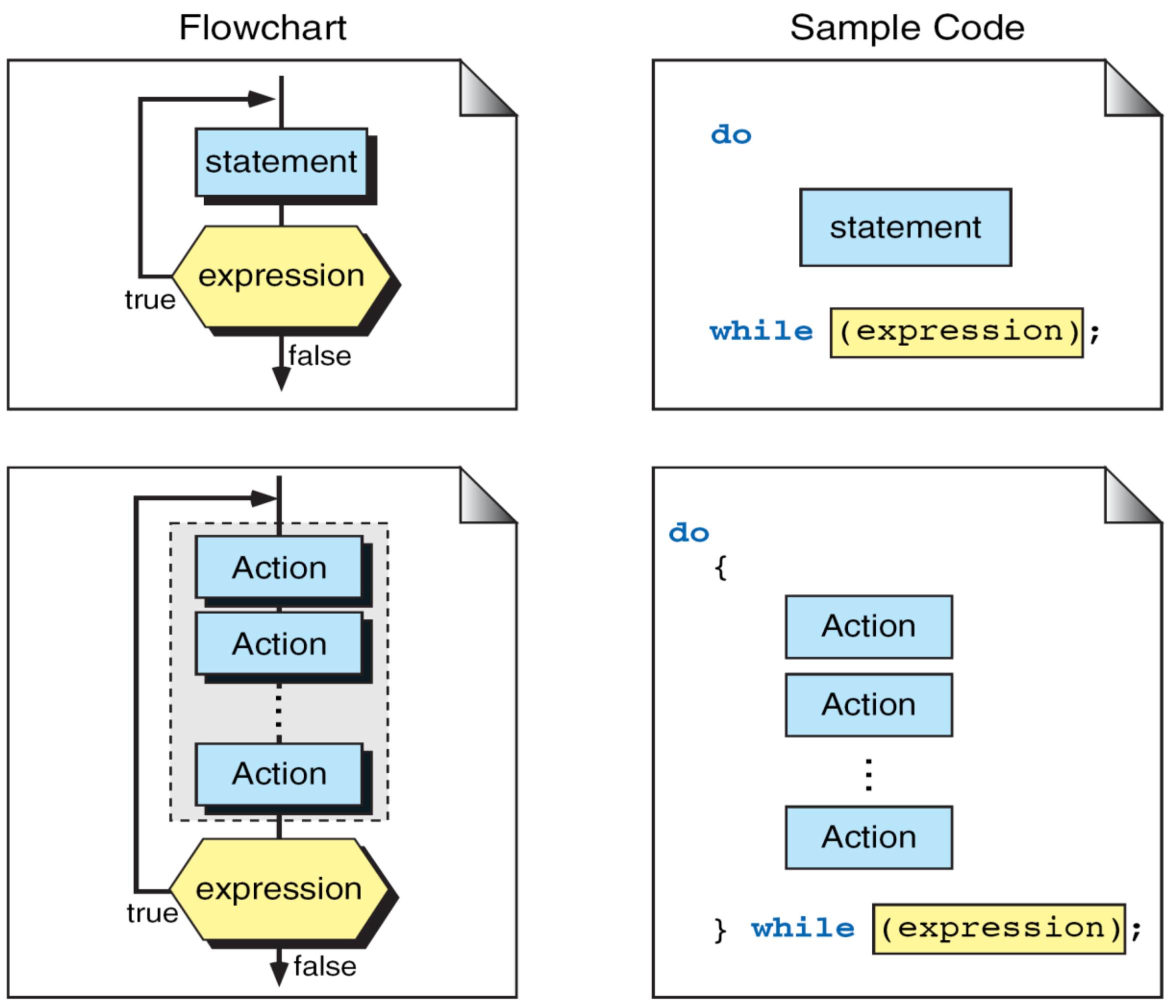
- The “do while" loop is a repetition statement that allows an action to be done at least once and then condition is tested.

- On reaching do statement, the program proceeds to evaluate the body of the loop first.

- At the end of the loop, condition statement is evaluated.

- If the condition is true, it evaluates the body of the loop once again.

- This process continues up to the condition becomes false.


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

Example 3: To print Fibonacci sequence for the given number.

```
#include<stdio.h>
main()
{
    int a=0,b=1,c,i;
    i=1;
    printf("%d%d",a,b);
    do
    {
        c=a+b;
        i++;
        printf("%3d",c);
        a=b;
        b=c;
    }while(i<=10);
}
```

Example 4: To print multiplication table for 5.

```
#include <stdio.h>
 void main()
{
      int  i = 1, n=5;
      do
      {
            printf(" %d * %d = %d ", n, i, n*i);
            i = i + 1;
      } while ( i<= 5);
}
```

# Comparison between while and do while

Pretest
nothing prints

```
while (false)
  {
   printf("Hello World");
  } // while
```

```
do
  {
   printf("Hello World");
  } while (false);
```

Post-test
Hello... prints

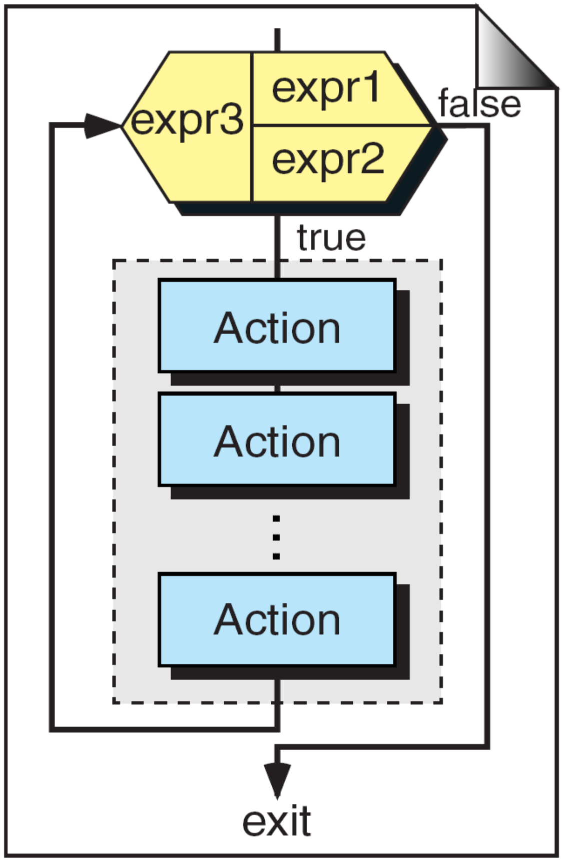
# for

- A for loop is used when a loop is to be executed a known number of times.

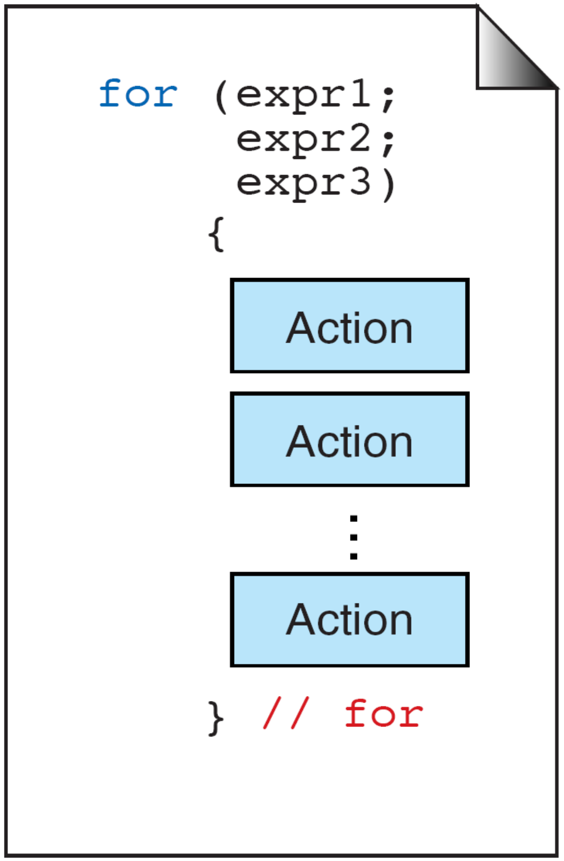
- We can do the same thing with a while loop, but the for loop is easier to read and more natural for counting loops.

General form of the for is:

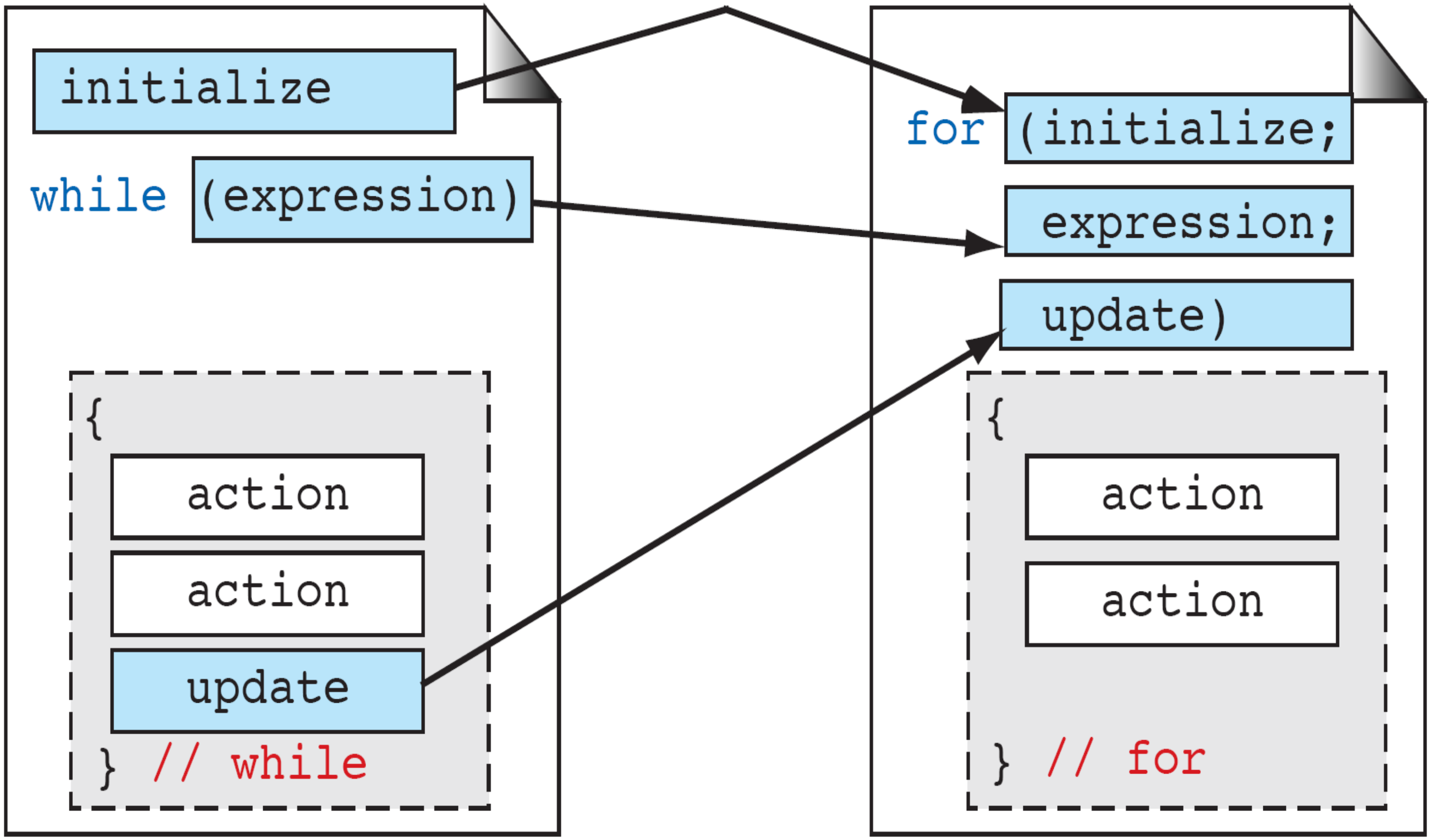
```
for( initialization; test-condition; updation)
{
    Body of the loop
}
```



(a) Flowchart



(b) C Language


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

We can write the for loop in the following ways:

**Option 1:**

```
for (k= 1; k< = 10 ;)
{
        printf("%d", k);
        k = k + 1;
}
```

Here the increment is done within the body of the for loop and not in the for statement. Note that the semicolon after the condition is necessary.

**Option 2:**

```
    int k = 1;
    for (; k< = 10; k++)
    {
        printf("%d",  k);
    }
```

Here the initialization is done in the declaration statement itself, but still the semicolon before the condition is necessary.

**Option 3:**

```
int k = 1;
for (; k< = 10;)
{
      printf("%d",  k);
      k++;
}
```

Here neither initialization nor incrementation is done in the for statement , but still two semicolons are necessary.

For loop with no body

A statement, as defined by the C syntax, may be empty. This means that the body of the for may also be empty. This fact can be used to improve the efficiency of certain algorithms as well as to create time delay loops.

The following statement shows how to create a time delay loop using a for loop:

```
for (t = 0; t < SOME _VALUE; t++);
```

- The operator comma , is used to separate the more than one expressions.

- A pair of expressions separated by a comma is evaluated left to right, and the type and value of the result are the type and value of the right operand.

- Thus, in a for statement, it is possible to place multiple expressions in the various parts.


Multiple initializations and updations in the for loop

The initialization and updation expression of the for loop can contain more than one statement separated by a comma.

for(i=1,j=n;i<10;i++,j--)

## The infinite loop

One of the most interesting uses of the for loop is the creation of the infinite loop. Since none of the three expressions that form the for loop are required, it is possible to make an endless loop by leaving the conditional expression empty.

For example:

```
for (;  ;)
        printf("The loop will run forever\n");
```

Actually the for (;  ;) construct does not necessarily create an infinite loop because C's break statement, when encountered anywhere inside the body of a loop, causes immediate termination of the loop.

Program control then picks up the code following the loop, as shown here:

```
for (; ;)
{
    ch = getchar( );          /* get a character */
    if (ch = = 'A')
    break ;
}
printf ("you typed an A");
```

This loop will run until A is typed at the keyboard.

# Nested Loops:

A loop inside another loop is called a nested loop. We can have any number of nested loops as required.

```
while(condition)
 {
     while(condition)
     {
     statement(s);
      }
  statement(s);
 }
```

```
do
{
 statement(s);
     do
     {
     statement(s);
      }while( condition );
 }while( condition );
```

```
for (initialization; condition; increment/decrement)
{
 statement(s);
  for (initialization; condition; increment/decrement)
 {   statement(s);   }
}
```

## Unconditional statements:

- break,
- continue
- goto statements with examples

# break

- When a break statement is enclosed inside a block or loop, the loop is immediately exited and program continues with the next statement immediately following the loop.

- When loop are nested break only exit from the inner loop containing it.

```
while (expr)
  {
   ...
   break;
   ...
  } // while
```

```
do
  {
   ...
   break;
   ...
  } while (expr);
```

```
for (expr1; expr2; expr3)
  {
   ...
   break;
   ...
  } // for
```

Example 6: Program to demonstrate break statement.

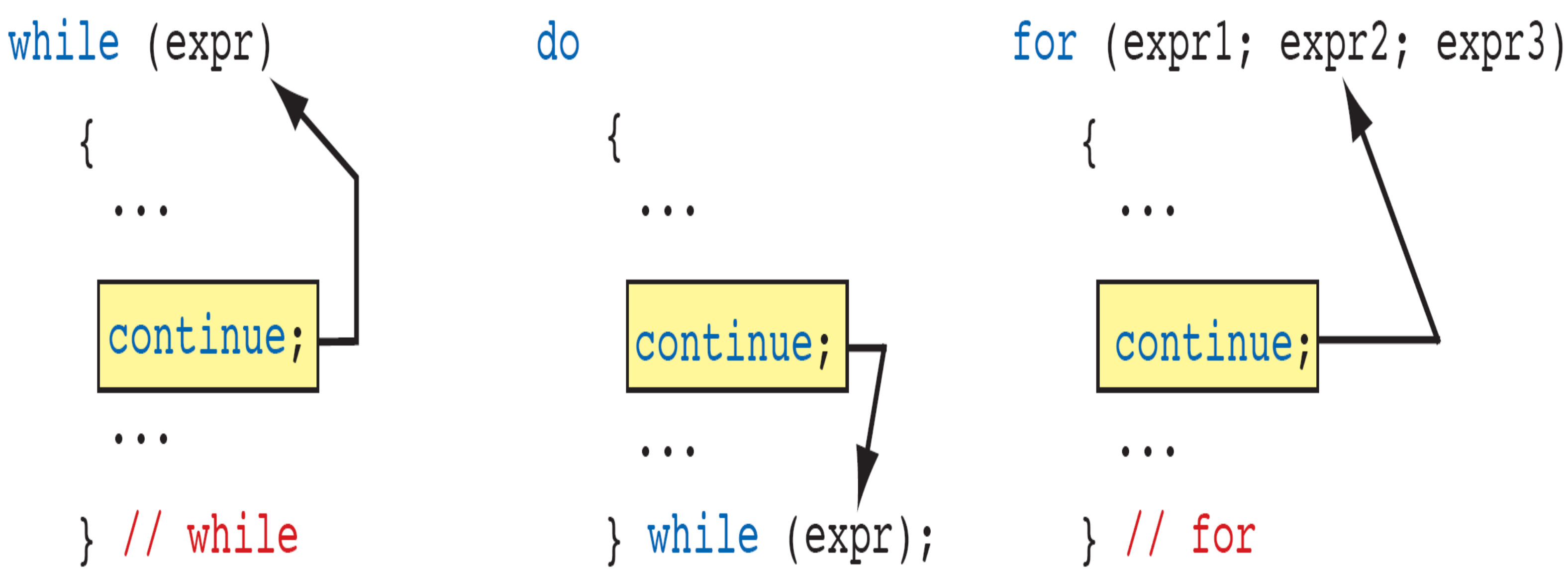
```
#include<stdio.h>
main()
{
    int i;
    i=1;
    while(i<=10)
    {
        if(i==8)
            break;
        printf("%d\t",i);
        i=i+1;
    }
    printf("\n Thanking You");
}
```

# continue

- When a continue statement is enclosed inside a block or loop, the loop is to be continued with the next iteration.

- The continue statement tells the compiler, skip the following statements and continue with the next iteration.

- The format of the continue statement is:



Example 5: Program to demonstrate continue statement.

```
#include<stdio.h>

 main()
{
      int i;
      for(i=1;i<=5;i++)
      {
            if(i = = 3)
                  continue;
            printf(" %d",i);
      }
}
```

# goto

**goto** :is a jumping statement in c language, which transfer the program’s control from one statement to another statement (where label is defined).

**Defining a label :**

**Label_name:** **(**label name is valid identifier**)**

**Transferring the control using ‘goto’:**

**goto label_name;**

**Transferring the control from down to** top

```
#include <stdio.h>
int main()
{
 int i=1;
repeat: printf("%d\n“,i);
             i++;
 if(i<=10)
 goto repeat;
 return 0;
}
```

# goto

**Transferring the control from top to Down**

```
#include <stdio.h>
int main()
{
	int n;
	printf( “enter n”);
	scanf(“%d”,&n);
	if(n<0)
		goto  end;
	printf(“number=%d”,n);
	end: printf(“end”);
 return 0;
}
```