ICS-103

Computer Programming in C

Chapter 5:

Repetition and Loop Statements in C

Dr. Tarek Ahmed Helmy El-Basuny

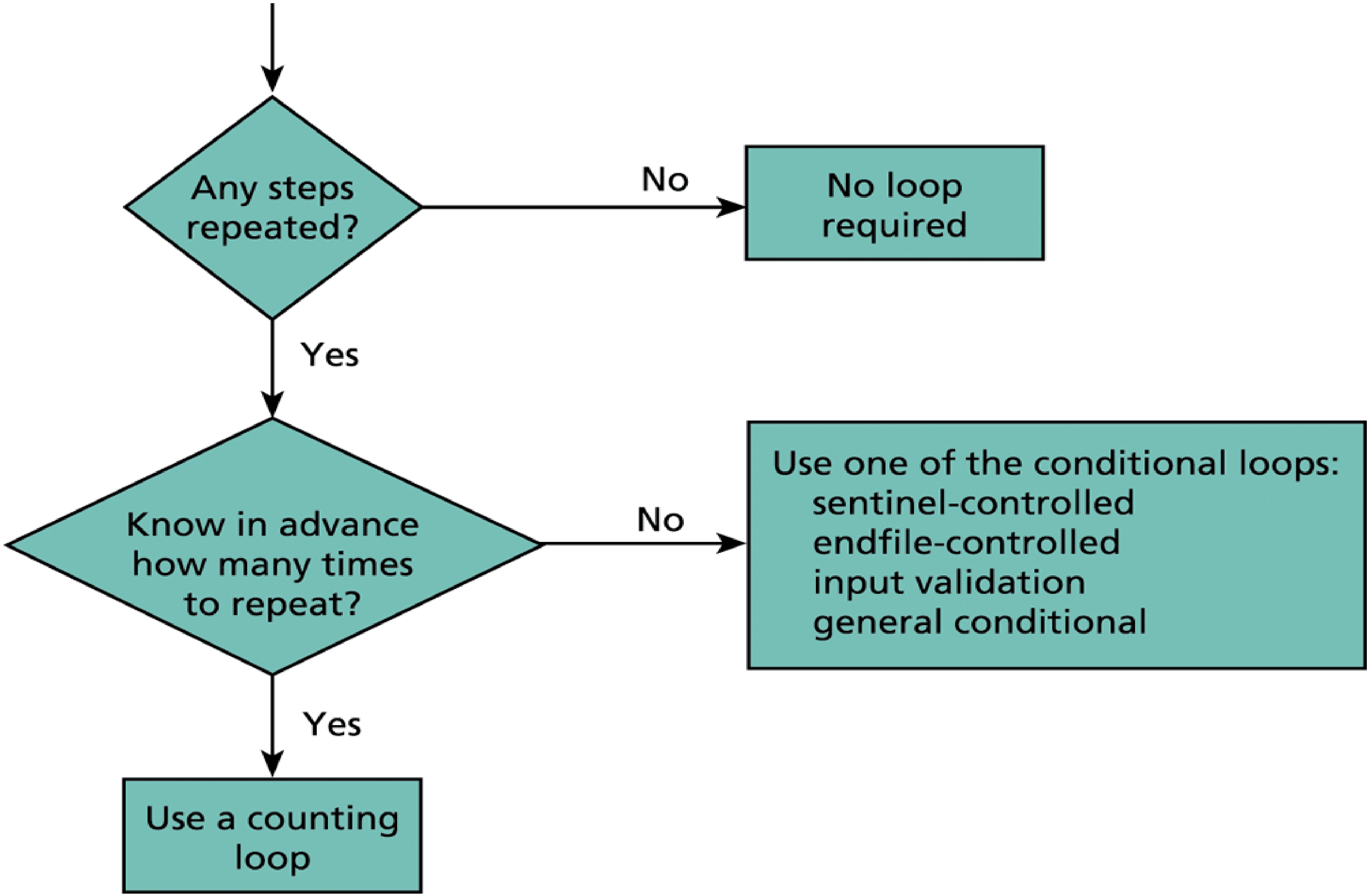
Outline of Ch. 05 Topics

- Repetition in C Programs
- Counting/Unconditional Loops
  - The while statement
    - Code examples of using while statement
  - The for statement
    - Code examples of using for statement
- Conditional Loops
    - Code examples of using conditional loops
- Nested Loops
    - Code examples of using nested loops
- The do-while statement
    - Code examples of using do-while statement
- How to debug and test programs
- Common Programming Errors

Repetition in Programs

- Loop structure
  - A control structure that repeats a group of statements in a program
- **Three loop control structures in C:**
  - The **while** statement
  - The **for** statement
  - The **do-while** statement
- Loop body
  - The statements that are repeated inside the loop
- Three questions to raise to implement loop structure:
  1. Are there any statements need to be repeated in the problem?
  2. If the answer to question 1 is yes,
     - Is the number of repetitions known in advance?
  3. If the answer to question 2 is no,
     - Then how long to keep repeating the steps?
- Based on the answers of the above questions, we can decide which loop structure we can use?

Flowchart of Loop Choice



Counting/Unconditional Loop: while Statement

- Counting loop:
  - ➲ A loop that can be controlled by a counter variable
- The number of iterations (repetitions) **can be determined before loop execution begins.**
- General format of a counting loop:

```
Set loop control variable to an initial value;
while (loop control variable < final value)  {
   Do something multiple times ;
   Increase loop control variable by 1;
}
```

The while Statement

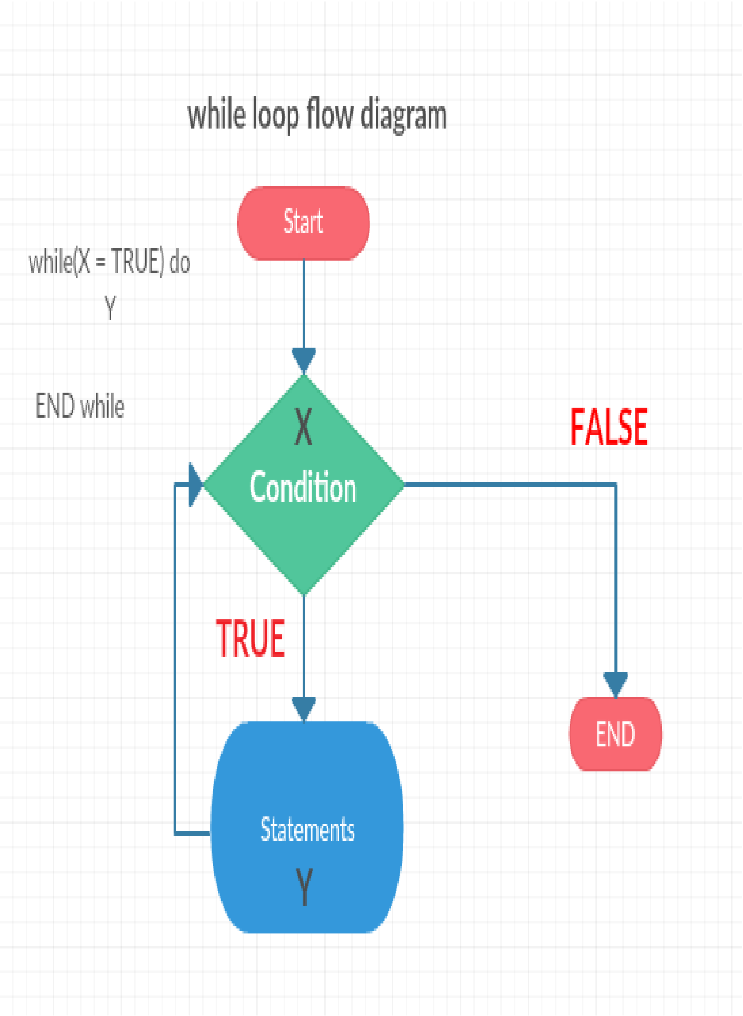
- Syntax:

Loop Repetition Condition

```
while (condition) {
   statement_1 ;
   statement_2 ;
   . . .
   statement_N ;
}
```

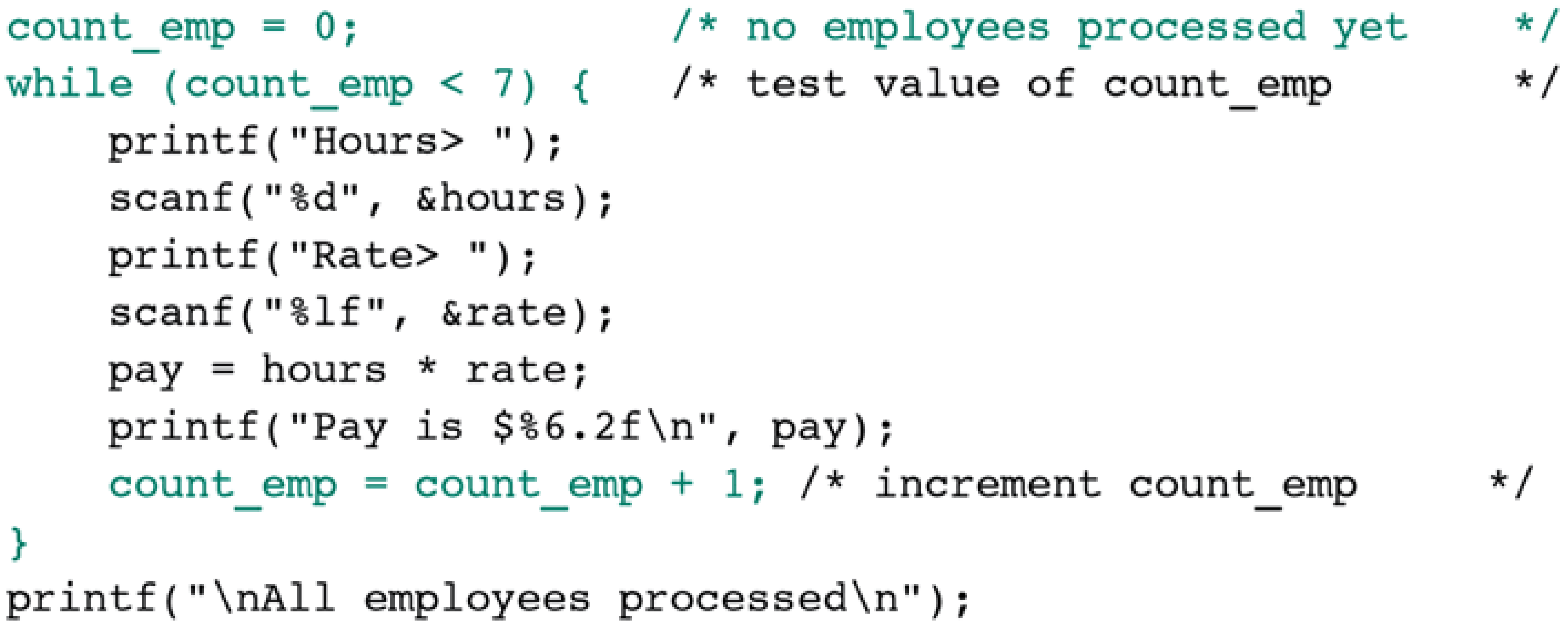
Loop Body:
Can be one statement,
or compound statement

- As long as the condition is true, the loop body is executed.
- Re-test the condition after each iteration.
- The loop terminates when the condition becomes false.



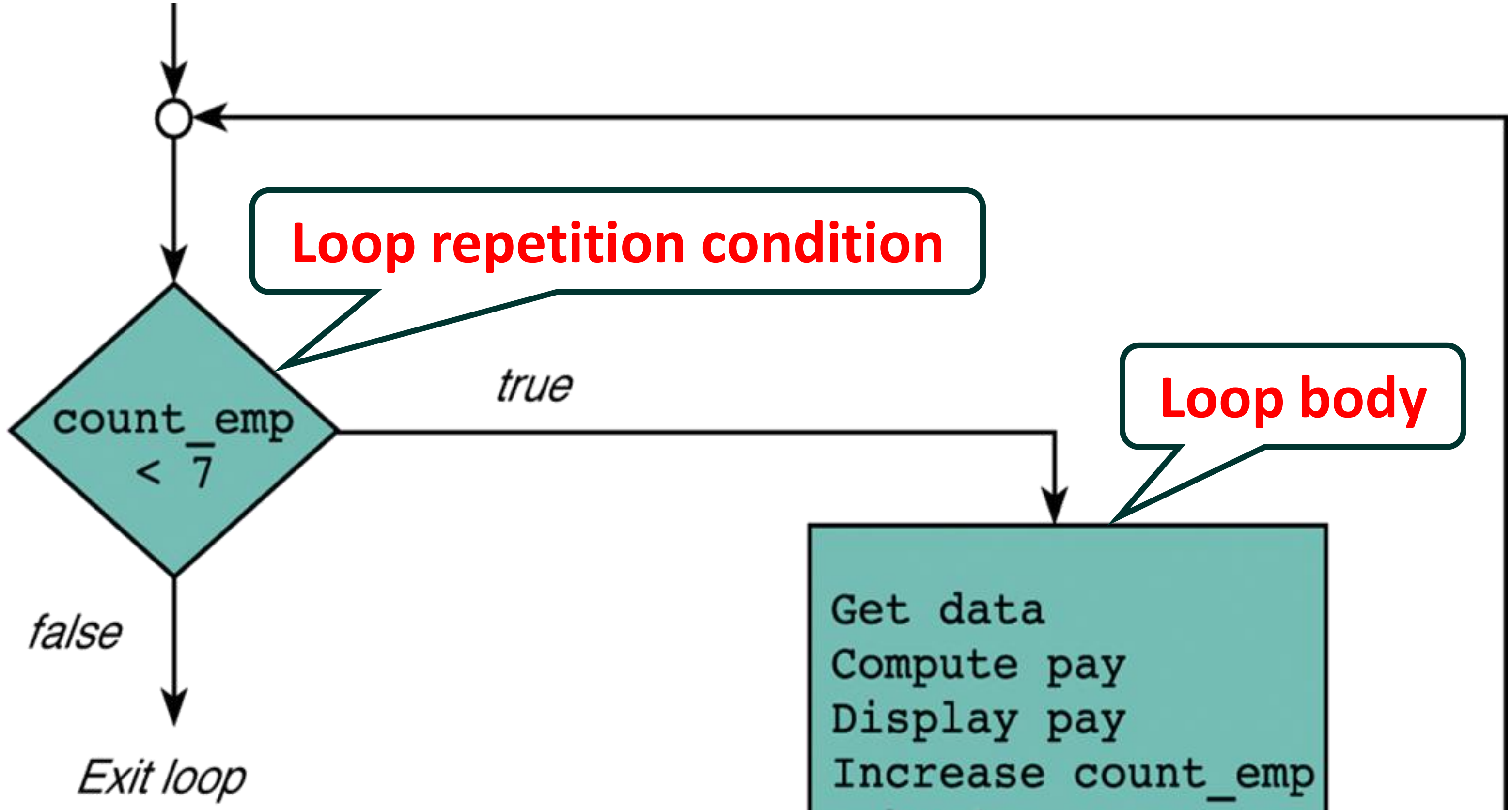
Example of a while Loop

- Compute and display the total payment for 7 employees.
  - Initialization: `count_emp = 0;`
  - Testing: `(count_emp < 7)`
  - Do the calculation;
  - Updating: `count_emp = count_emp + 1;`



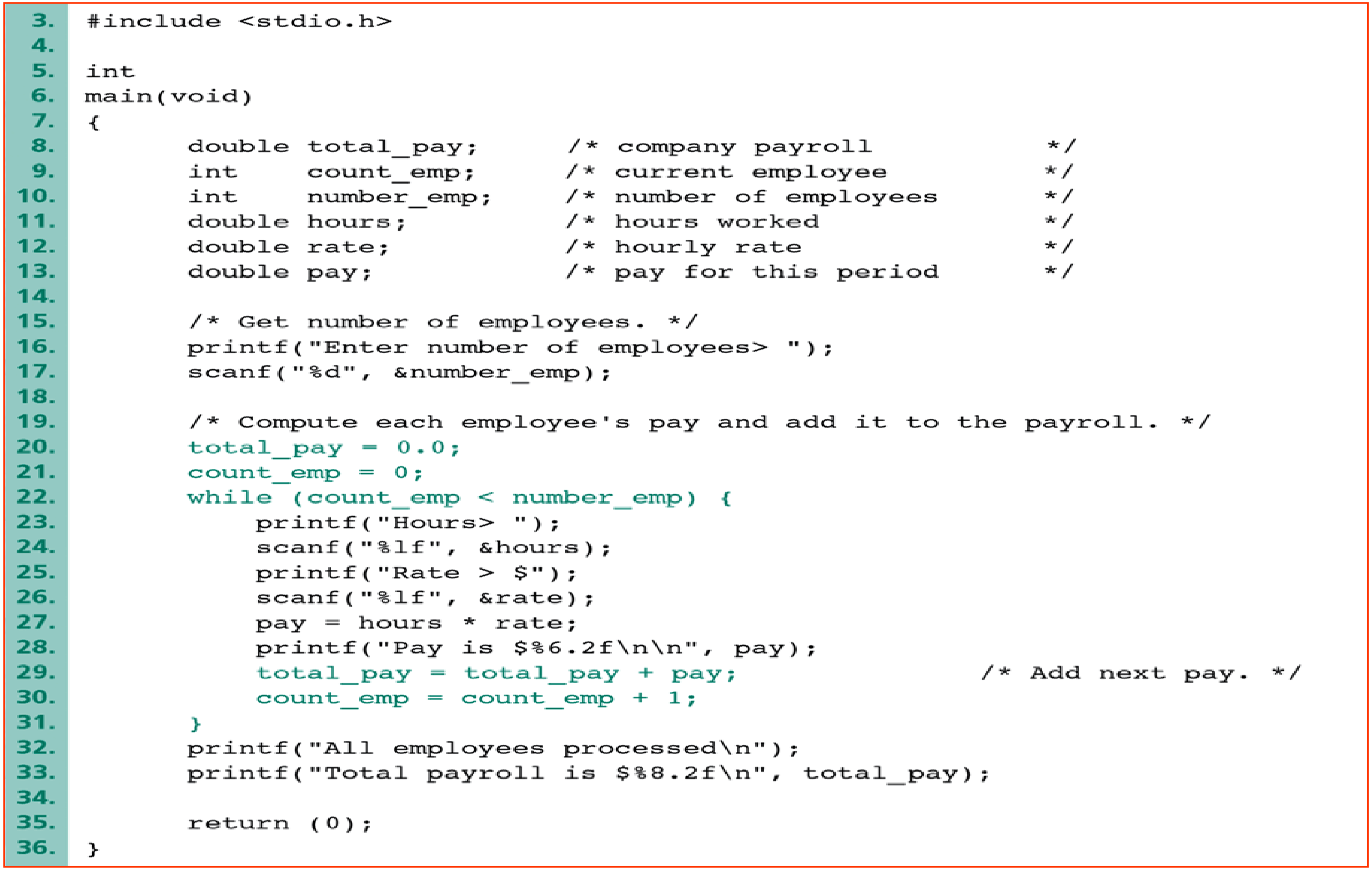
```
count_emp = 0;                  /* no employees processed yet     */
while (count_emp < 7) {         /* test value of count_emp        */
    printf("Hours> ");
    scanf("%d", &hours);
    printf("Rate> ");
    scanf("%lf", &rate);
    pay = hours * rate;
    printf("Pay is $%6.2f\n", pay);
    count_emp = count_emp + 1; /* increment count_emp          */
}
printf("\nAll employees processed\n");
```

Flowchart of a while Loop



If **count_emp** is not updated, the loop will execute forever. Such a loop is called **infinite loop**.

Total Payroll of a Company



```
#include <stdio.h>

int
main(void)
{
      double total_pay;      /* company payroll            */
      int    count_emp;      /* current employee           */
      int    number_emp;     /* number of employees        */
      double hours;          /* hours worked               */
      double rate;           /* hourly rate                */
      double pay;            /* pay for this period        */

      /* Get number of employees. */
      printf("Enter number of employees> ");
      scanf("%d", &number_emp);

      /* Compute each employee's pay and add it to the payroll. */
      total_pay = 0.0;
      count_emp = 0;
      while (count_emp < number_emp) {
          printf("Hours> ");
          scanf("%lf", &hours);
          printf("Rate > $");
          scanf("%lf", &rate);
          pay = hours * rate;
          printf("Pay is $%6.2f\n\n", pay);
          total_pay = total_pay + pay;                /* Add next pay. */
          count_emp = count_emp + 1;
      }
      printf("All employees processed\n");
      printf("Total payroll is $%8.2f\n", total_pay);

      return (0);
}
```

Sample Run

```
Enter number of employees> 3
Hours> 50
Rate> $5.25
Pay is $262.50

Hours> 6
Rate> $5.0
Pay is $ 30.00

Hours> 15
Rate> $7.0
Pay is $105.00

All employees processed
Total payroll is $ 397.50
```

Example of a while Loop

- Just like relational operators (<, >, >=, <=, ! =, ==), we can also use logical operators in while loop.
- In this example we are testing multiple conditions **using logical operator** inside while loop.

```
#include <stdio.h>
int main() {
int i=1, j=1;
while (i <= 4 || j <= 3) {
printf("%d %d\n",i, j);
i++;
j++;
}
return 0;
}
```

Example of a while Loop

- Just like relational operators (<, >, >=, <=, ! =, ==), we can also use logical operators in while loop.
- In this example we are testing multiple conditions using **logical or** operator inside while loop.

```
#include <stdio.h>
int main() {
int i=1, j=1;
while (i <= 4 || j <= 3) {
printf("%d %d\n",i, j);
i++;
j++;
}
return 0;
}
```

Output:

```
1 1
2 2
3 3
4 4
```

Example of a while Loop

- You can make use of **break** to come out of while loop at any time.
- In this example we are testing **how to use break to terminate the while loop**.

```
 #include <stdio.h>
 main() {
         int i = 10;
         while ( i > 0 ) {
 printf("Hello %d\n", i );
         i = i -1;
                         if( i == 6 ) {
                         break;
                                       }
                                  }
 }
```

Example of a while Loop

- You can make use of **break** to come out of while loop at any time.
- In this example we are testing **how to use break to terminate the while loop**.

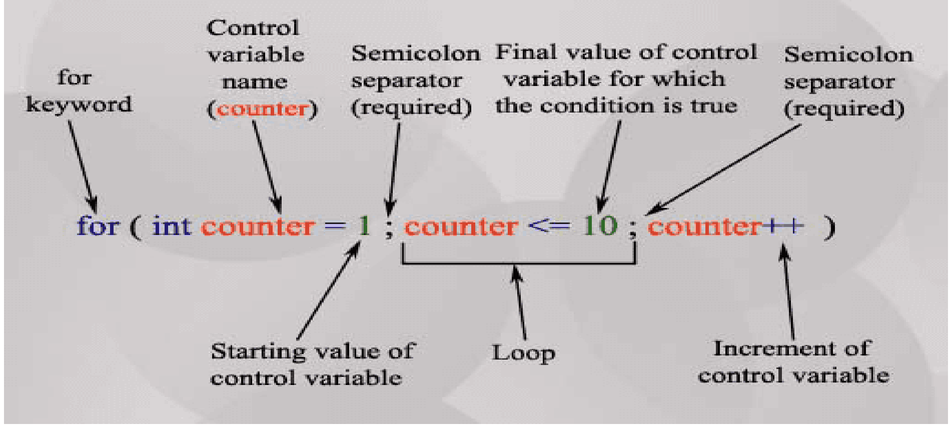
```
#include <stdio.h>
main() {
    int i = 10;
    while ( i > 0 ) {
printf("Hello %d\n", i );
    i = i -1;
            if( i == 6 ) {
            break;
                  }
              }
}
```

```
Output:
Hello 10
Hello 9
Hello 8
Hello 7
```

Counting Loop: The for Statement

- Format of for loop:

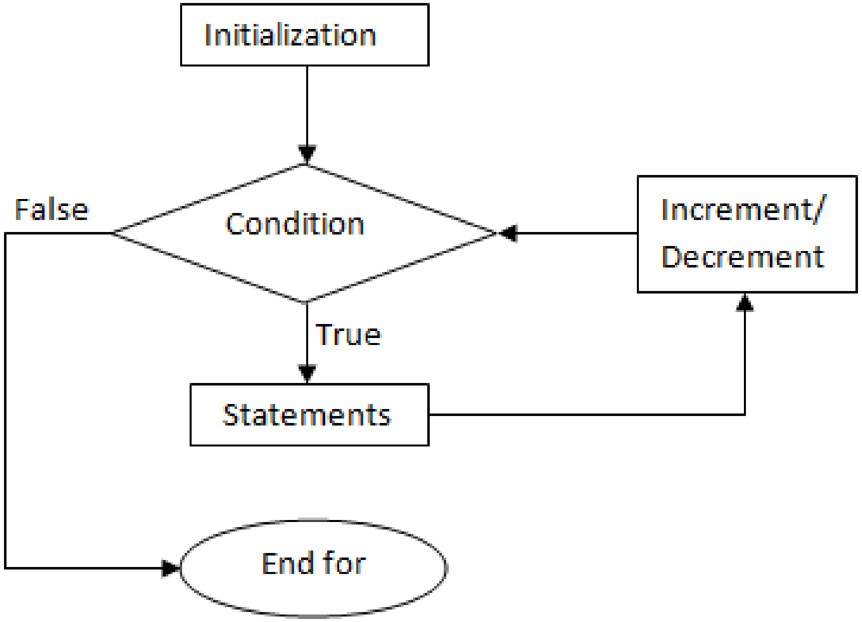
```
for (initialization expression;
     loop repetition condition;
     update expression)
     Statement;/*Can be Compound*/
```



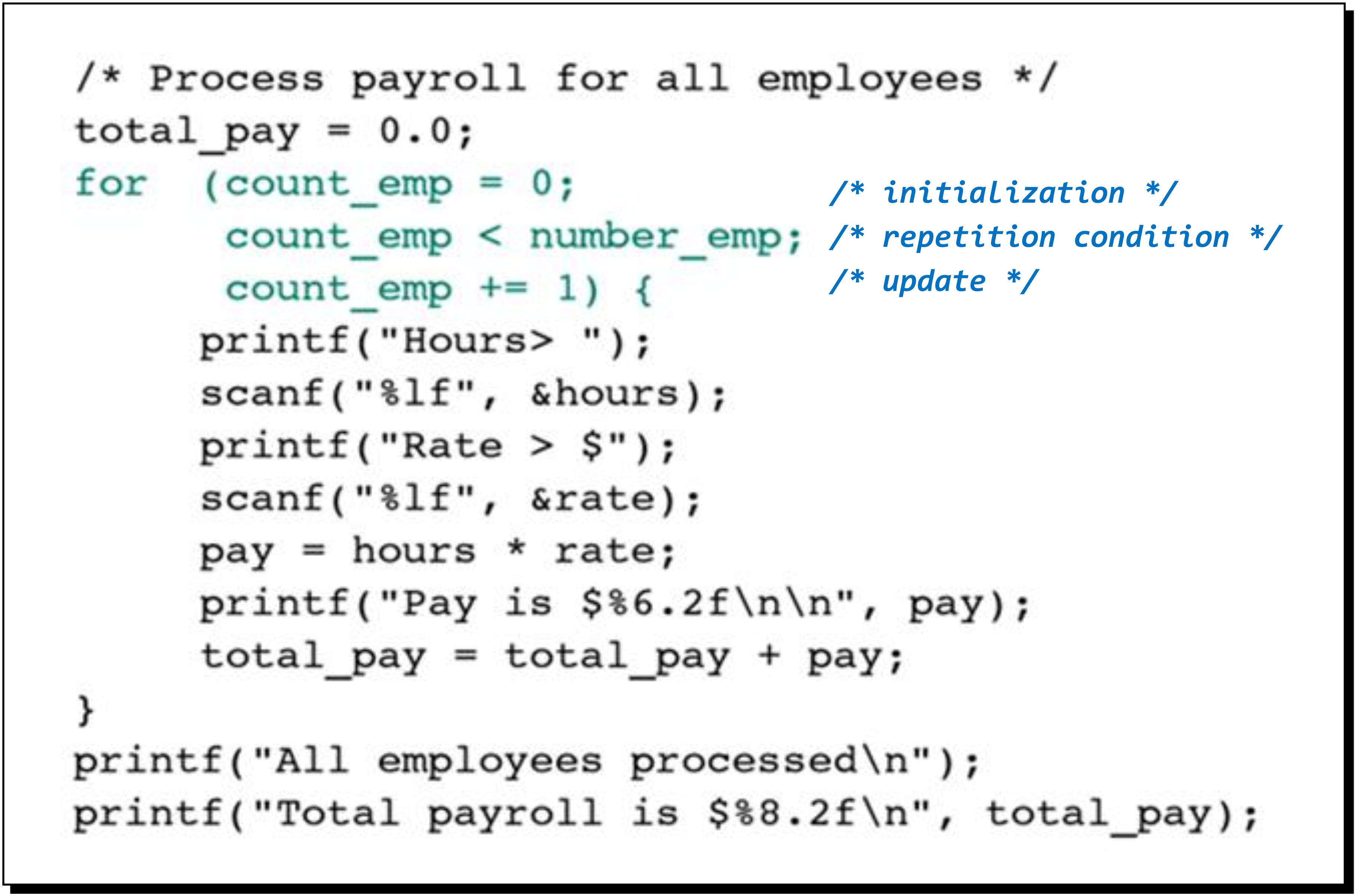
```
for ( int counter = 1 ; counter <= 10 ; counter++ )
```

for keyword
Control variable name (counter)
Semicolon separator (required)
Final value of control variable for which the condition is true
Semicolon separator (required)
Starting value of control variable
Loop
Increment of control variable

- First, the initialization expression is executed.
- Then, the loop repetition condition is tested.
  - If true, the statement is executed, the update expression is computed, and the repetition condition is re-tested.
- Repeat as long as the repetition condition is true.



Accumulating a Sum: total_pay



```
/* Process payroll for all employees */
total_pay = 0.0;
for  (count_emp = 0;                   /* initialization */
      count_emp < number_emp;          /* repetition condition */
      count_emp += 1) {                /* update */
     printf("Hours> ");
     scanf("%lf", &hours);
     printf("Rate > $");
     scanf("%lf", &rate);
     pay = hours * rate;
     printf("Pay is $%6.2f\n\n", pay);
     total_pay = total_pay + pay;
}
printf("All employees processed\n");
printf("Total payroll is $%8.2f\n", total_pay);
```

Example of a for Loop

- **Calculate the sum of first n natural numbers.**

```
#include <stdio.h>
main() {
        int num, count, sum = 0;
        printf("Enter a positive integer: ");
        scanf("%d", &num);
// for loop terminates when count is less than or equal the num
for(count = 1; count <= num; ++count)
        {
        sum += count;
        }
printf("Sum = %d", sum);
return 0;
}
```

Example of a for Loop

- Calculate the sum of first n natural numbers.

```
#include <stdio.h>
main() {
        int num, count, sum = 0;
        printf("Enter a positive integer: ");
        scanf("%d", &num);
// for loop terminates when n is less than count
for(count = 1; count <= num; ++count)
        {
        sum += count;
        }
printf("Sum = %d", sum);
return 0;
}
```

Output:

```
Enter a positive integer: 10
Sum = 55
```

Example of a for Loop

- The program takes an integer input from the user and generates the multiplication table up to 10.

```
#include <stdio.h>
int main() {
int n, i;
printf("Enter an integer: ");
scanf("%d",&n);
      for(i=1; i<=10; ++i) {
      printf("%d * %d = %d \n", n, i, n*i);
                              }
return 0;
          }
```

Example of a for Loop

- The program takes an integer input from the user and generates the multiplication table up to 10.

```
 #include <stdio.h>
 int main() {
 int n, i;
 printf("Enter an integer: ");
 scanf("%d",&n);
        for(i=1; i<=10; ++i) {
        printf("%d * %d = %d \n", n, i, n*i);
                                }
 return 0;
            }
```

Output:

```
Enter an integer: 9
     9 * 1 = 9
     9 * 2 = 18
     9 * 3 = 27
     9 * 4 = 36
     9 * 5 = 45
     9 * 6 = 54
     9 * 7 = 63
     9 * 8 = 72
     9 * 9 = 81
    9 * 10 = 90
```

Compound Assignment Operators

- The compound assignment operators enable you to abbreviate assignment statements.
  - For example, the statement value = value + 3 can be written as value += 3.
- The += operator adds the value of the right operand to the value of the left operand and stores the result in the left operand's variable.

Compound assignment operator	Sample expression	Explanation	Assigns
***Assume:* c = 4, d = "He"**			
+=	c += 7	c = c + 7	11 to c
-=	c -= 3	c = c - 3	1 to c
*=	c *= 4	c = c * 4	16 to c
/=	c /= 2	c = c / 2	2 to c
\= or %=(reminder)	c \= 3	c = c \ 3	1 to c
^= (exponent)	c ^= 2	c = c ^ 2	16 to c
&= (concatenate)	d &= "llo"	d = d & "llo"	"Hello" to d

Compound Assignment Operators

- The following table lists the assignment operators supported by the C language.

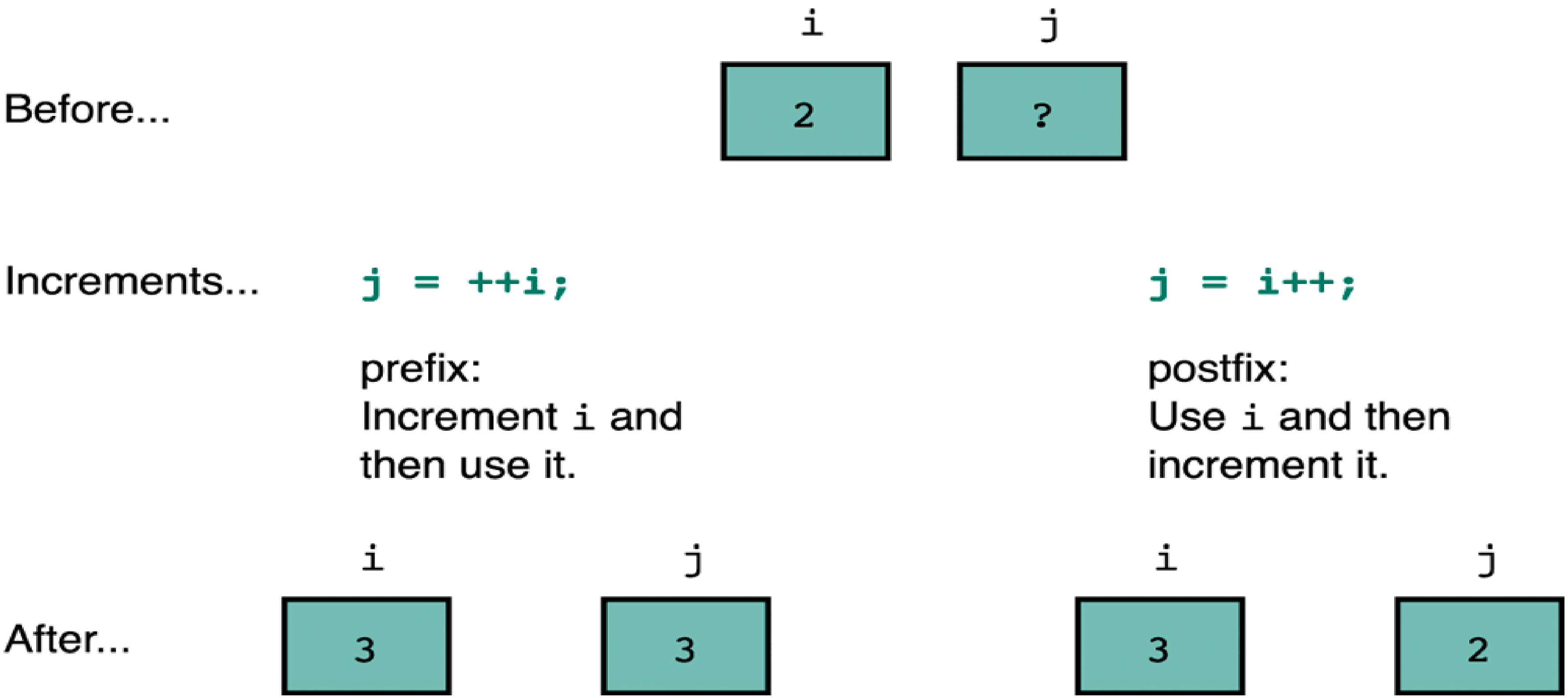
Operator	Description	Example
=	**Assignment operator**. Assigns values from right side operands to left side operand.	C = A + B will assign the value of A + B to C
+=	**Add AND assignment operator**. It adds the right operand to the left operand and assign the result to the left operand.	C += A is equivalent to C = C + A
−=	**Subtract AND assignment operator**. It subtracts the right operand from the left operand and assigns the result to the left operand.	C −= A is equivalent to C = C − A
*=	**Multiply AND assignment operator**. It multiplies the right operand with the left operand and assigns the result to the left operand.	C *= A is equivalent to C = C * A
/=	**Divide AND assignment operator**. It divides the left operand with the right operand and assigns the result to the left operand.	C /= A is equivalent to C = C / A
\=	**Divide AND assignment operator:** Divides the value of a variable or property on its left by the value on its right, and assigns the **integer** result to the variable or property on its left.	C \= A is equivalent to C = C / A
%=	**Remainder Operator**: Compute the remainder after division.	C %= A is equivalent to C = C % A
<<=	**Left shift AND assignment operator**.	C <<= 2 is same as C = C << 2
>>=	**Right shift AND assignment operator**.	C >>= 2 is same as C = C >> 2
&=	**Bitwise AND assignment operator**.	C &= 2 is same as C = C & 2
^=	**Bitwise exclusive OR and assignment operator**.	C ^= 2 is same as C = C ^ 2
|=	**Bitwise inclusive OR and assignment operator**.	C |= 2 is same as C = C | 2

Examples: Compound Assignment Operators

- variable op= expression; *is equivalent* *to* variable = variable op (expression);
- expression1 += expression2 *is equivalent* expression1 = expression1 + expression2

Statement with Simple Assignment Operator	Equivalent with Compound Assignment Operator
`count_emp = count_emp + 1;`	`count_emp += 1;`
`time = time - 1;`	`time -= 1;`
`product = product * item;`	`product *= item;`
`total = total / number;`	`total /= number;`
`n = n % (x+1);`	`n %= x+1;`

Prefix and Postfix Increment & Decrement



Before...

i j

2 ?

Increments... `j = ++i;` `j = i++;`

prefix:
Increment i and
then use it.

postfix:
Use i and then
increment it.

After...

i j i j

3 3 3 2

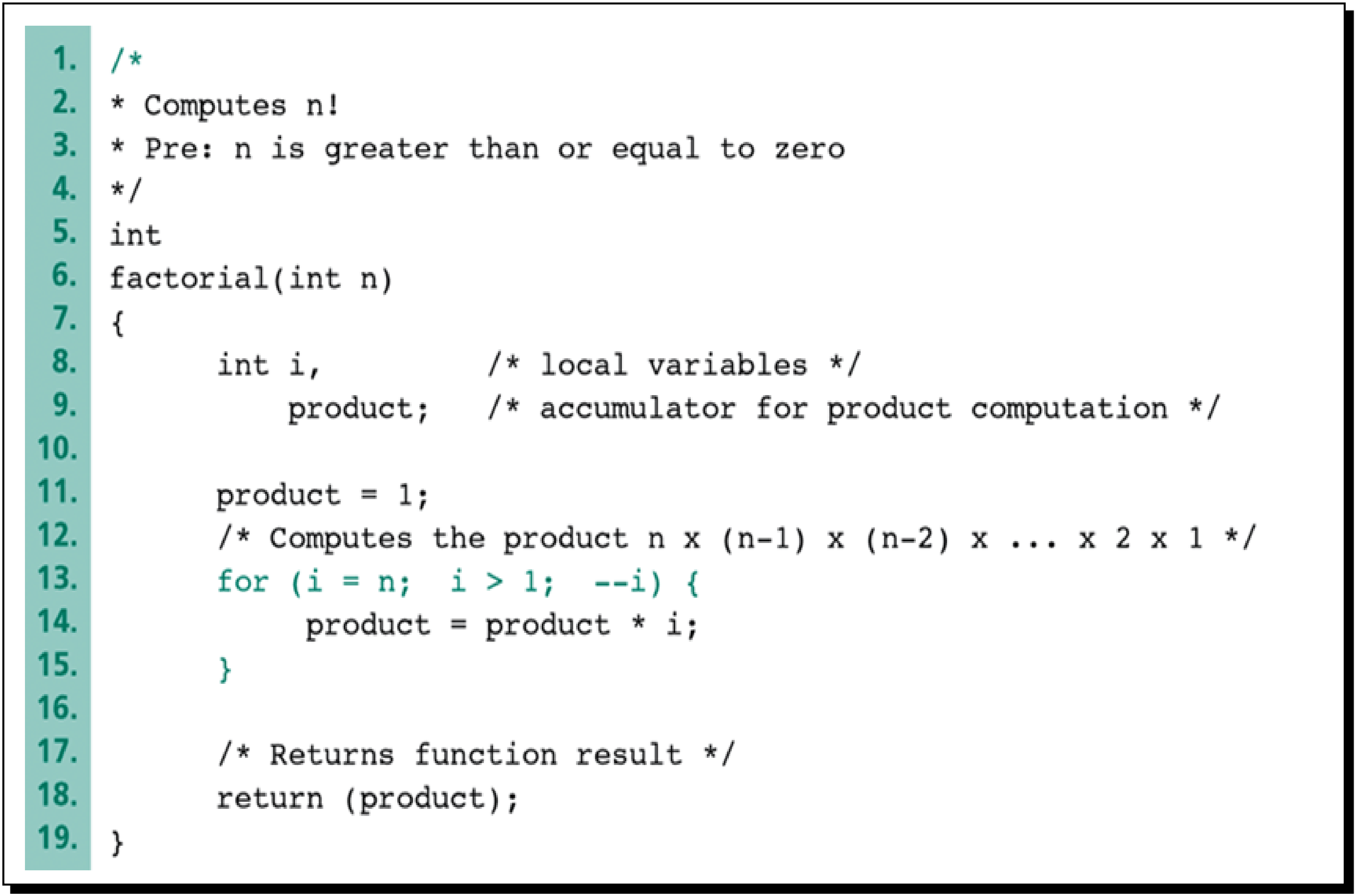
- ❑ C also provides the Decrement operator -- that can be used in either the prefix or postfix position. ***i.e. let***

x = 3; **// x is initialized to 3**

y = x--; ***// y is assigned 3 but x is now 2,***

y = --x; ***// y is assigned 1 and x is now 1***

Function to Compute the Factorial

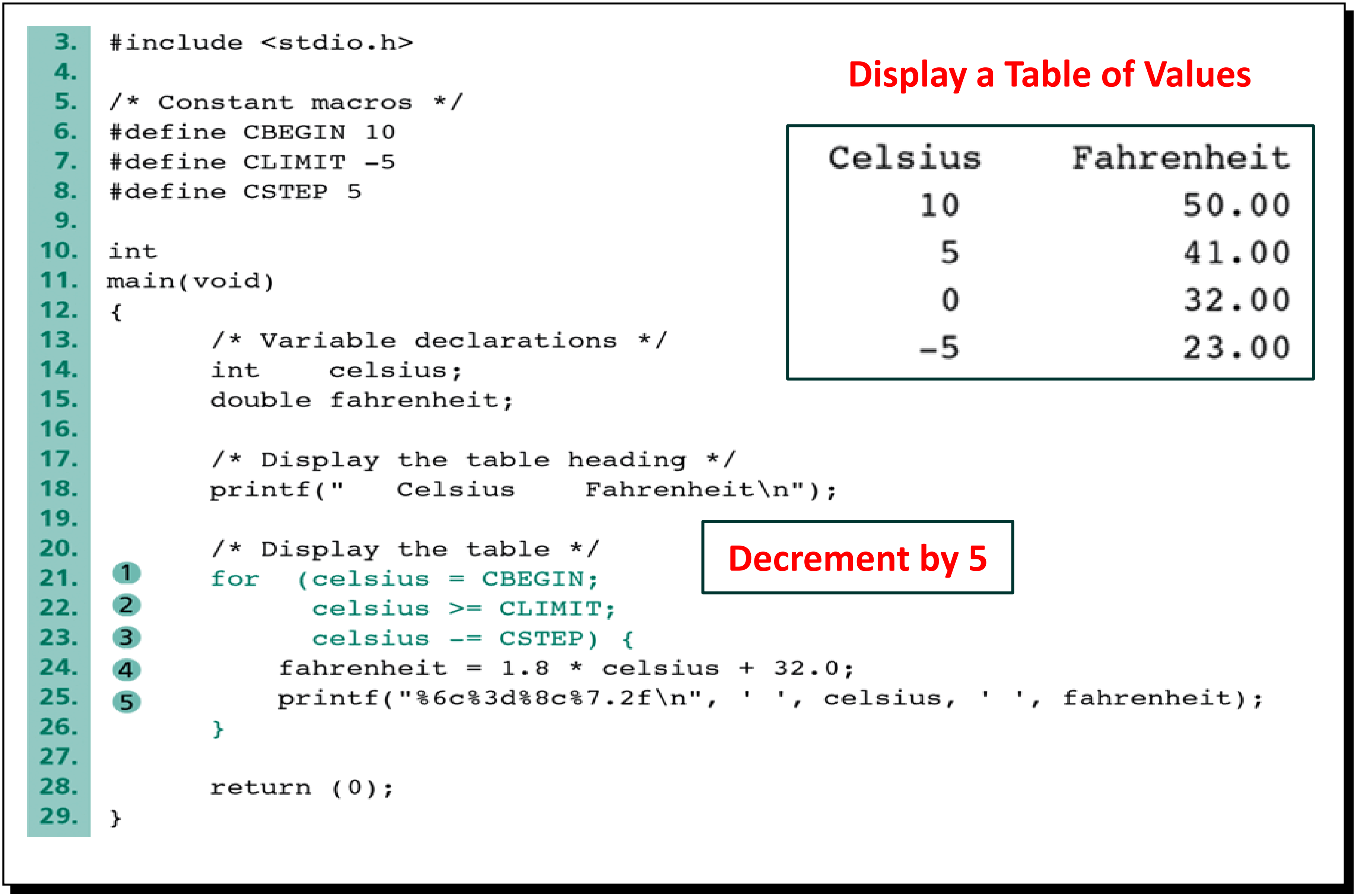


```
/*
* Computes n!
* Pre: n is greater than or equal to zero
*/
int
factorial(int n)
{
      int i,          /* local variables */
          product;    /* accumulator for product computation */

      product = 1;
      /* Computes the product n x (n-1) x (n-2) x ... x 2 x 1 */
      for (i = n;  i > 1;  --i) {
           product = product * i;
      }

      /* Returns function result */
      return (product);
}
```

Conversion of Celsius to Fahrenheit



```c
#include <stdio.h>

/* Constant macros */
#define CBEGIN 10
#define CLIMIT -5
#define CSTEP 5

int
main(void)
{
      /* Variable declarations */
      int    celsius;
      double fahrenheit;

      /* Display the table heading */
      printf("   Celsius    Fahrenheit\n");

      /* Display the table */
      for  (celsius = CBEGIN;
            celsius >= CLIMIT;
            celsius -= CSTEP) {
          fahrenheit = 1.8 * celsius + 32.0;
          printf("%6c%3d%8c%7.2f\n", ' ', celsius, ' ', fahrenheit);
      }

      return (0);
}
```

Display a Table of Values

```
   Celsius    Fahrenheit
        10         50.00
         5         41.00
         0         32.00
        -5         23.00
```

Decrement by 5

Conditional Loops

- **Conditional Loops:** means it is not possible to determine the exact number of loop repetitions before loop execution begins. Or
- The loop body executed repeatedly as long as the logical condition is true.
- **Example of a conditional loop: input validation**

```
printf("Enter number of students> ");
scanf("%d", &num_students);
while (num_students < 0) {
  printf("Invalid negative number; try again> ");
  scanf("%d", &num_students);
}
```

- This loop will be repeated as fare as the user did not input a positive number.
- That means the condition of terminating the above loop is the invalid (positive) input.

Sentinel Controlled Loops

- In many programs, we may need to input a list of data values.
  - Often, we don't know the length of the list.
- We ask the user to enter a unique data value, called a sentinel value, after the last data item.
- Sentinel Value
  - An end marker that follows the last value in a list of data
  - For readability, we used `#define` to name the `SENTINEL`
- The loop repetition condition terminates when the sentinel value is read.

Sentinel Controlled while Loop

```c
#include <stdio.h>
#define SENTINEL -1  /* Marking end of input */

int main(void) {     /* Compute the sum of test scores */
  int sum = 0;       /* Sum of test scores */
  int score;         /* Current input score */

  printf("Enter first score (%d to quit)> ", SENTINEL);
  scanf("%d", &score);
  while (score != SENTINEL) {
    sum += score;
    printf("Enter next score (%d to quit)> ", SENTINEL);
    scanf("%d", &score);
  }
  printf("\nSum of exam scores is %d\n", sum);
  return (0);
}
```

Sentinel Controlled for Loop

```
#include <stdio.h>
#define SENTINEL -1  /* Marking end of input */

int main(void) {     /* Compute the sum of test scores */
  int sum = 0;       /* Sum of test scores */
  int score;         /* Current input score */

  printf("Enter first score (%d to quit)> ", SENTINEL);
  for (scanf("%d", &score); score != SENTINEL;
       scanf("%d", &score)) {
    sum += score;
    printf("Enter next score (%d to quit)> ", SENTINEL);
  }
  printf("\nSum of exam scores is %d\n", sum);
  return (0);
}
```

Infinite Loop on Faulty Input Data

- Reading faulty data or incorrect updating the control variable can result in an infinite loop.

```
scanf("%d", &score); /* read integer */
```

- Suppose the user enters the letter X

```
Enter next score (-1 to quit)> X
```

`scanf` fails to read variable `score` as letter `X.`

- Variable `score` is not modified in the program

`score != SENTINEL` is always true

- Therefore, Infinite Loop

```
#include <stdio.h>
void main()
{ int i=0,sum=0,a;
while(i<=9)
{ scanf(“&d”,&a) ;
sum+=a; }
printf(“%d”, sum);
}
//infinite loop
//never terminate,
// since i = 0 all the time
```

Detecting Faulty Input Data

- `scanf` can detect faulty input as follows:

```
status = scanf("%d", &score);
```

- If `scanf` successfully reads `score` then `status` is `1`.
- If `scanf` fails to read `score` then `status` is `0`.
- We can test `status` to detect faulty input.
- This can be used to terminate the execution of a loop.
- In general, `scanf` can read multiple variables.
- It returns the number of successfully read inputs.

Terminating Loop on Faulty Input

```
  #include <stdio.h>
  #define SENTINEL -1 /* Marking end of input */
  int main(void) {     /* Compute the sum of test scores */
  int sum = 0;       /* Sum of test scores */
  int score;         /* Current input score */
  int status;        /* Input status of scanf */
  printf("Enter first score (%d to quit)> ", SENTINEL);
  status = scanf("%d", &score);
  while (status != 0 && score != SENTINEL) {
    sum += score;
    printf("Enter next score (%d to quit)> ", SENTINEL);
    status = scanf("%d", &score);
  }
  printf("\nSum of exam scores is %d\n", sum);
  return (0);
}
```

Nested for Loops

- ❑ Consist of an outer loop with one or more inner loops.
  - ➲ Each time the outer loop is repeated, the inner loops are fully executed.
- ❑ Example:

outer loop

inner loop

```
void stars(int n) {
  int i, j;
  for (i=1; i<=n; i++) {
    for (j=1; j<=i; j++) {
      printf("*");
    }
    printf("\n");
  }
}
```

```
Outer-Loop {

// body of outer-loop
        Inner-Loop {
        // body of inner-loop
        }
// continue body of outer-loop
}
```

- ❑ What is the output of this code?

Example: Nested for Loops

- Consist of an outer loop with one or more inner loops
- Each time the outer loop is repeated, the inner loops are reentered and executed.
- Example:

```
void stars(int n) {
  int i, j;
  for (i=1; i<=n; i++) {
    for (j=1; j<=i; j++) {
      printf("*");
    }
    printf("\n");
  }
}
```

outer loop

inner loop

```
stars(5);

*
**
***
****
*****
```

Example of a nested while Loop

```
#include <stdio.h>
int main() {
int i=1,j;
    while (i <= 5)
    {
    j=1;
           while (j <= i ) {
           printf("%d ",j); j++;
                           }
    printf("\n"); i++;
    }
return 0;
}
```

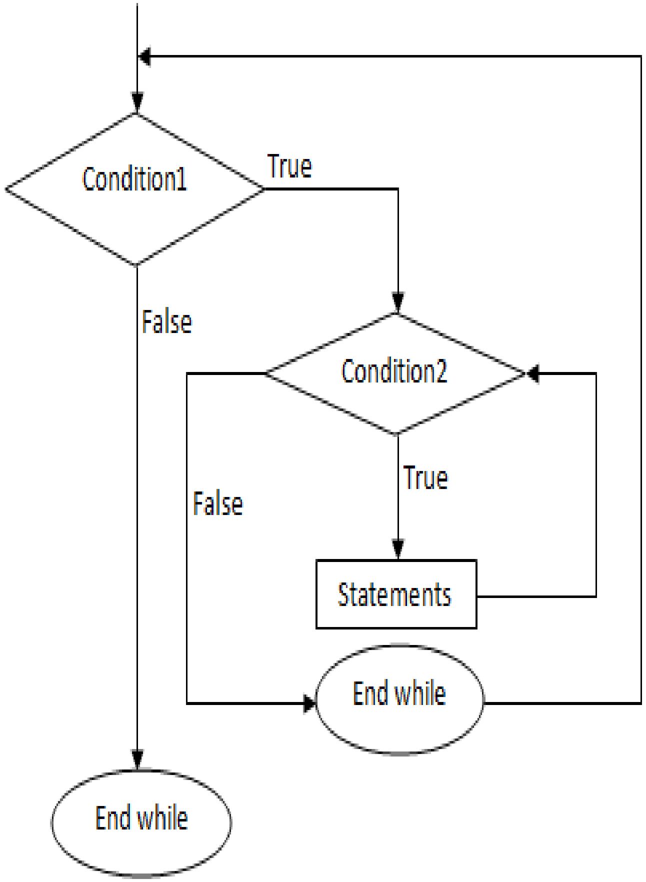


fig: Flowchart for nested while loop

Example of a nested while Loop

```
#include <stdio.h>
int main() {
int i=1,j;
    while (i <= 5)
    {
    j=1;
          while (j <= i ) {
          printf("%d ",j); j++;
                          }
    printf("\n"); i++;
    }
return 0;
}
```

Output:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

Nested if Statement inside Loop

```
/* day1: Sun is 1, Mon is 2, ..., Sat is 7 */
/* days: number of days in month */
void display_month(int day1, int days) {
  int i;
  printf(" Sun Mon Tue Wed Thu Fri Sat\n");
  for (i=1; i<day1; i++)
    printf("    ");          /* spaces before day1 */
  for (i=1; i<=days; i++) {
    printf("%4d", i);        /* print day number */
    if ((day1+i-1)%7 == 0){ /* end of week */
      printf("\n");
    }
  }
  printf("\n\n");
}
```

outer for loop

nested if

Displaying a Month

```
display_month(7, 30); /* function call */
```

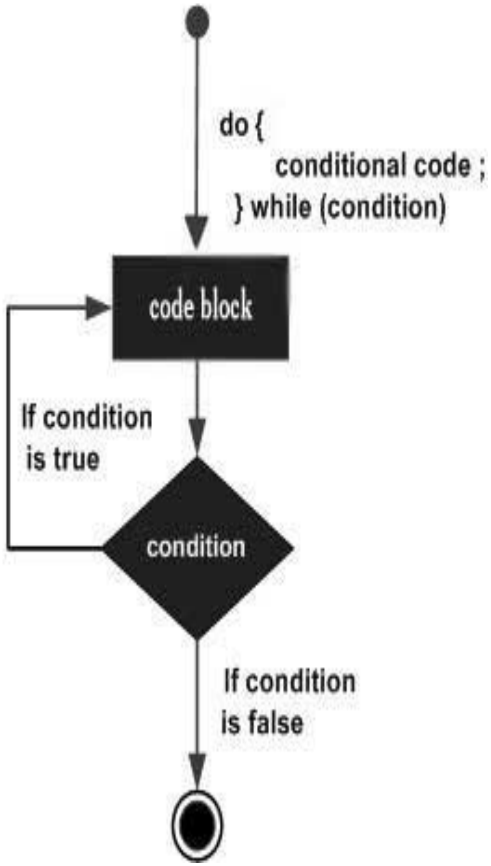
```
Sun Mon Tue Wed Thu Fri Sat
                          1
  2   3   4   5   6   7   8
  9  10  11  12  13  14  15
 16  17  18  19  20  21  22
 23  24  25  26  27  28  29
 30
```

Output

The do-while Statement

- As we have seen, both for and while statements evaluate the loop condition before the execution of the **loop body**.
- But, the do-while statement **evaluates** the loop condition after the **execution of the loop body**.
- That means, in do-while statement, the **loop body** executes at least one time before **evaluating** the loop condition.
- Syntax of do-while:

```
Do {
 Loop body statement; /* Can be compound */
}
while (loop repetition condition);
```



Example: Using do-while Loop

```
#include <conio.h>

int main() {

 char ch;    /* Variable Declarations */

      /* do… while statement*/
  do {

    printf("Repeat again [y/n]? ");

    ch = getch(); /*in conio.h and used to read only char from keyboard*/

    printf("%c\n", ch); /* display character */

  } while (ch=='y'|| ch=='Y');  // condition

} /* as fare as the input is y or Y, the loop will be repeated */
```

Example of Using do-while Loop

```
#include <stdio.h>

int main()

{

    int j=0;

            do {

            printf("Value of variable j is: %d\n", j);

            j++;

            }

            while (j<=3);

            return 0;

            }
```

```
Value of variable j is: 0
Value of variable j is: 1
Value of variable j is: 2
Value of variable j is: 3
```

Example While vs. do..while loop in C

Using while loop:

```
#include <stdio.h>
int main() {
int i=0;
while(i==1) {
printf("while vs. do-while");
}
printf("Out of loop");
}
```

Same example using do-while loop

```
#include <stdio.h>
int main() {
int i=0;
do {
printf("while vs. do-while\n");
}
while(i==1);
printf("Out of loop");
}
```

Example While vs. do..while loop in C

Using while loop:

```
#include <stdio.h>
int main() {
int i=0;
while(i==1) {
printf("while vs. do-while");
}
printf("Out of loop");
}
```

```
Out of loop
```

Same example using do-while loop

```
#include <stdio.h>
int main() {
int i=0;
do {
printf("while vs. do-while\n");
}
while(i==1);
printf("Out of loop");
}
```

```
while vs. do-while
Out of loop
```

Example: Using do-while

```c
// Program to add numbers until the user enters zero
#include <stdio.h>
int main(){
double number, sum = 0;     /* Variable Declarations */
// loop body is executed at least once
do    {
printf("Enter a number: ");
     scanf("%lf", &number);
    sum += number;
   }
 while(number != 0.0);
  printf("Sum = %.2lf",sum);
  return 0;
}
```

Output:

```
Enter a number: 1.5
Enter a number: 2.4
Enter a number: -3.4
Enter a number: 4.2
Enter a number: 0
Sum = 4.70
```

Using do-while to Validate Input

```
/* get integer value between min and max */
int get_int (int min, int max) {
  int  inval;         /* input value between min and max */
  int  status;        /* returned by scanf */
  int  error;         /* error flag for bad input */
  char ch;            /* character input to skip */
  do {
    printf("Enter integer from %d to %d> ", min, max);
    status = scanf("%d", &inval);
    error = 1;              /* set error flag */
    if (status == 0)        /* faulty input */
      printf("Invalid character %c\n", getchar());
    else if (inval < min || inval > max)
      printf("Number %d is out of range\n", inval);
    else error = 0;         /* clear error flag */
    do ch = getchar();  /* used to read from keyboard */
    while (ch != '\n');   /* skip to end-of-line */
  } while (error);
  return inval;   }
```

Common Programming Errors (1/3)

- A loop executes one more time or one less time than the expected.
- Example:

```
for (count = 0; count <= n; ++count)
  sum += count;
```

Executes n + 1 times

```
for (count = 1; count < n; ++count)
  sum += count;
```

Executes n – 1 times

- Checking loop boundaries
  - Be carful while setting the initial and final values of the loop control variable.

Common Programming Errors (2/3)

- Do not confuse `if` and `while` statements
  - `if` statement implements **a decision step** (choose one option)
  - `while` statement implements **a loop** (repetition)
- `In for` loop: remember to end the initialization step and the loop repetition condition with semicolon (;)
- Remember to use braces `{` and `}` around a loop body consisting of multiple statements.
- Remember to provide a prompt for the user, when using a sentinel-controlled loop.
- Make sure the sentinel value cannot be confused with a normal data input.

Common Programming Errors (3/3)

- Use `do-while` only when there is no possibility of zero loop iterations.
- Do not use increment, decrement, or compound assignment as sub-expressions in complex expressions.

```
a *= b + c;        /* a = a*(b+c); */
```

- There is no shorter way to write: `a = a*b + c;`
- Be sure that the operand of an increment/decrement operator is a variable.

```
z = ++j * k--;  /* ++j; z=j*k; k--; */
```

How to Debug and Test a Program?

- ❑ We have discussed before how to write, compile and execute C Programs.
- ❑ We have explained before three types of errors; syntax error, runtime error, and logical error.
- ❑ **Today, we need to learn how to debug C program by using a debugger.**
- ❑ **Debugger:** is a program that can run *your program* one line at a time to observe the effect of each C statement on the variables.
- ❑ **Debugging** is the process of locating and removing **program's** errors or abnormalities, which is handled by software **programmers.** **This can be done by:**

1. Using a debugger program (i.e. gdb) as following:

- ➲ **Select the Debug option** while compiling the program.
- ➲ Launch the C debugger (i.e. gdb)
- ➲ Execute program one statement at a time,
- ➲ Watch and print the value of variables at runtime,
- ➲ Set **breakpoints** at selected statements, where you suspect errors.
  - ▪ The debugger will stop at the break point, and you can examine the values of variables to determine whether the program segment has executed correctly or not.

2. Using extra printf statements without a debugger

➲Insert ***extra printf statements*** that display intermediate results at critical points in C program.

Steps to Debug a C Program using a gdb Debugger

- **Step 1**. Compile the C program with debugging option –g
  - $ cc -g factorial.c   /* file name is factorial.c */
- This allows the compiler to collect the debugging information.
- **Step 2**. Launch the C debugger (gdb) as shown below.
  - $ gdb a.out   /* creates a.out file which will be used for debugging */
- **Step 3**. Set up a break point inside C program using.
  - break line_number; i.e. break 8;
- **Step 4**. Execute the C program in gdb debugger.
  - run
- **Step 5**. Printing the variable values inside gdb debugger.
  - Exemples: print i
- **Step 6**. you can use continue, next, and stepping over in gdb commands.
  - There are three kind of gdb operations you can choose when the program stops at a break point.
    - c or continue: Debugger will continue executing until the next break point.
    - n or next: Debugger will execute the next line as single instruction.
    - s or step: Same as next, but does not treats function as a single instruction, instead goes into the function and executes it line by line.

Is this Program correct?

```c
/* C program that calculates and prints the factorial of a number.
 *  However this C program contains some errors in it for debugging purpose*/
# include <stdio.h>
int main()
{
int i, num, j;
printf ("Enter the number: ");
scanf ("%d", &num );
for (i=1; i<num; i++)
j=j*i;
printf("The factorial of %d is %d\n",num,j);
}
```

- What is the problem in the above program?
- Can we apply the steps to debug this program?

Example: Debugging using gdb

- Step 1. Compile the C program with debugging option –g
  - cc -g factorial.c
- Launch the C debugger (gdb) as shown below.
  - gdb a.out
- Step 3. Set up a break point inside C program.
  - Places break point in the C program, where you suspect errors.
  - i.e. break 8 // While executing the program, the debugger will stop at the break point, and gives you the prompt to debug.
- Step 4. Execute the C program in gdb debugger
  - Run // it would execute until the first break point, and give you the prompt for debugging.
- Step 5. Printing the variable values inside gdb debugger
  - print i or print j or print num
- As you will see, in the factorial.c,
  - we have not initialized the variable j.
  - So, it gets garbage value resulting in a big numbers as factorial values.

The End!!

Thank you

Any Questions?