Repetition and Loop Statements

SDC OSW CSE-3541

Department of Computer Science & Engineering
ITER, Siksha ‘O’ Anusandhan Deemed To Be University
Jagamohan Nagar, Jagamara, Bhubaneswar, Odisha - 751030

Text Book(s)

- **Jeri R. Hanly & Elliot B. Koffman**

  Problem Solving and Program Design in C

  Seventh Edition, Pearson Education

- **Robert Love**

  LINUX

  System Programming

  Second Edition, SPD, O'REILLY

Talk Flow

Introduction

- Till now in our programs so far, the statements in the program body execute only once.
- However, in most cases, we may require a program statement/ group of program statements may be repeated for many times.
- To display a message "**`CSE-ITER`**" , the code fragment

```
printf("CSE-ITER");
```

- To display the message "**`CSE-ITER`**" for 1000 or more times, the code fragment ???
- 1000 or more **`printf`** statements one after other. But it is time consuming and monotonous work. So, an alternate tool is required to optimize the number of **`printf`**s.
- The alternate tool is a kind of program control structure called repetition/ iteration/ looping.
- The repetition of steps in a program is called a loop.

Repetition in Programs

- An ability to specify repetition of a statement or a group of statements.
- **For example**, a company that has seven employees will want to repeat the gross pay and net pay computations in its payroll program seven times, once for each employee.
- **Observation:** a counter initialization, Repetition for how many times and updating of counter value.
- The loop body contains the statement(s) to be repeated.
- Three C loop control statements: **`while`** , **`for`** , and **`do`** - **`while`**.

State, whether looping required or not

✓ Calculate the sum of the test scores of a class of 60 students.

× Determine a number is even or odd

✓ Process a data file of Celsius temperatures. Count how many are above 100° C.

while Loop

```
SYNTAX:
           intialization;
           while (loop repetition condition){
                      statement ;
                      update;
          }
```

INTERPRETATION: The loop repetition condition (a condition to control the loop process) is tested; if it is **true**, the statement (loop body) is executed, and the loop repetition condition is retested. The statement is repeated as long as (**`while`**) the loop repetition condition is **true**. When this condition is tested and found to be **false**, the **`while`**loop is exited and the next program statement after the **`while`** statement is executed.
Note: If loop repetition condition evaluates to false the first time it is tested, statement is not executed.

```
Example:  /* Display N asterisks. */
               count_star = 0;
               while (count_star < N) {
                    printf("*");
                    count_star = count_star + 1;
               }
```

Test Yourself

1. Predict the output of this program fragment:

```
i = 0;
while (i <= 5) {
   printf("%d  %d\n", i, 10 - i);
   i = i + 1;
}
```

2. What is displayed by this program fragment for an input of 8?

```
    scanf("%d", &n);
    ev = 0;
    while (ev < n) {
         printf("%3d", ev);
         ev = ev + 2;
    }
   printf("\n");
```

3. Write a program fragment that produces this output:

```
0     1
1     2
2     4
3     8
4    16
5    32
6    64
```

Counting Loops and the while Statement

counter-controlled loop (counting loop)	:	a loop whose required number of iterations can be determined before loop execution begins
loop repetition condition	:	the condition that controls loop repetition
loop control variable	:	the variable whose value controls loop repetition
general format: a counter-controlled loop	:	**Pseudocode** (below)

Pseudocode

```
Set loop control variable to an initial value of 0.
while loop control variable < final value
        ................
        Increase loop control variable by 1.
```

Example: Multiplying a List of Numbers

Construct a loop to compute the product of a list of numbers as long as the product remains less than 10,000.

Example: Multiplying a List of Numbers

Construct a loop to compute the product of a list of numbers as long as the product remains less than 10,000.

```
/* Multiply data while product remains less than 10000 */
product = 1;
while (product < 10000) {
   printf("%d\n", product); /* Display product so far */
   printf("Enter next item> ");
   scanf("%d", &item);
   product = product * item; /* Update product */
}
```

Compound Assignment Operators

We have seen several instances of assignment statements of the form

$\boxed{\texttt{variable = variable op expression;}}$ where op is a C arithmetic operator.

These include increments and decrements of loop counters

```
count_emp = count_emp + 1;
time = time - 1;
```

C provides special assignment operators that enable a more concise notation for statements of this type.

For the operations + , - , * , / , and % , C defines the compound $\boxed{\text{op} =}$ assignment operators += , -= , *= , /= , and %= .

A statement of the form $\boxed{\texttt{variable op = expression;}}$

is an alternative way of writing the statement

```
variable = variable op (expression);
```

Example: Compound Assignment Operators

Statement with Simple Assignment Operator	Equivalent Statement with Compound Assignment Operator
`count_emp = count_emp + 1;`	`count_emp += 1;`
`time = time - 1;`	`time -= 1;`
`total_time = total_time + times;`	`total_time += time;`
`product = product * item;`	`product *= item;`
`n = n * (x + 1);`	`n *= x + 1;`

Where possible, write equivalents for the following statements using compound assignment operators:

```
s=s/5;
q=q*n+4;
z=z-x*y;
t=t+(u%v);
```

for Loop

```
SYNTAX:
  for (initialization expression;loop repetition condition;
                                          update expression){
         statement ;
  }
```

INTERPRETATION: First, the **`initialization expression`** is executed. Then, the **`loop repetition condition`** is tested. If it is **true**, the statement is executed, and the **`update expression`** is evaluated. Then the **`loop repetition condition`** is retested. The statement is repeated as long as the **`loop repetition condition`** is **true**. When this condition is tested and found to be **false**, the for loop is exited, and the next program statement after the **`for`** statement is executed.

```
Example:  /* Display N asterisks. */
      for(count_star = 0; count_star < N; count_star += 1){
                  printf("*");
      }
```

Caution: Although C permits the use of fractional values for counting loop control variables of type **`double`**, we strongly discourage this practice. Counting loops with type **`double`** control variables will not always execute the same number of times on different computers.

Example: The for statement

- Processing total pay for all employees in an organization:

```
 for ( count_emp = 0;                /* initialization */
       count_emp < number_emp;       /* loop repetition condition */
       count_emp += 1) {             /* update */

           statement(s);               /* loop body: total pay
                                                 computation*/
 }
```

- To display the value of **i** from 0 to a desired **max** value:

```
 /* Display nonnegative numbers < max */
for (i = 0; i < max; i=i+1)
       printf("%d\n", i);
```

- To Compute *n*! (factorial of *n*), where n is greater than or equal to zero:

```
 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;
}
```

Test Yourself

1. Trace the execution of the loop that follows for $n = 8$. Show values of odd and sum after the update of the loop counter for each iteration.

```
sum = 0;
for ( odd= 1;odd< n;odd+= 2)
         sum =sum + odd;
printf("Sum of positive odd
    numbers less than %d is %
    d.\n", n,sum);
```

2. Trace the following program fragment:

```
j = 10;
for (i =1; i <= 5;i=i+1) {
   printf("%d %d\n", i, j);
   j -= 2;
}
```

3. Trace output of the following program fragment:

```
for(num = 0;num < 26;++num)
{
   square = num * num;
   printf("%5d %5d\n", num,
       square);
}
```

4. For values of volts equal to 20 , 10 , 0 , -10 , -20 , computes value of **current** and displays **volts** and **current**. Assume **resistance=2**Ω

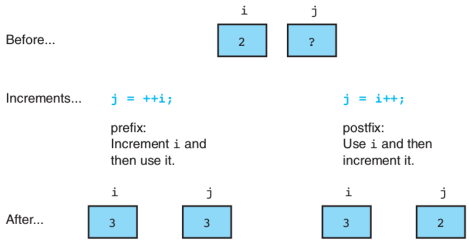
```
for(volts=20;volts>= -20;
    volts -= 10)
{
  current=volts/resistance;
  printf("%d %d\n", volts,
      current);
}
```

Increment and Decrement Operators

Increment operator: ++

- The increment operator ++ takes a single variable as its operand. The value of its operand is incremented by one.
- The value of the expression in which the ++ operator is used depends on the position of the operator.
- **Prefix increment:** When the ++ is placed immediately in front of its operand (i.e. $\boxed{\text{++i}}$), the value of the expression is the variable's value after incrementing.
- **Postfix increment:** When the ++ comes immediately after the operand (i.e. $\boxed{\text{i++}}$), the expression's value is the value of the variable before it is incremented.

An Example:



Increment and Decrement Operators

Decrement operator: - -

- This operator - - takes a single variable as its operand. The value of its operand is decremented by one.
- The value of the expression in which the - - operator is used depends on the position of the operator.
- **Prefix decrement:** When the - - is placed immediately in front of its operand (i.e. $\boxed{\text{- -i}}$), the value of the expression is the variable's value after decrementing.
- **Postfix decrement:** When the - - comes immediately after the operand (i.e. $\boxed{\text{i- -}}$), the expression's value is the value of the variable before it is decremented.

Example:

- Let the initial value of **n** is **4**. Output ???

```
printf("%d", --n); /* 3 */
printf("%d", n);   /* 3 */
```

- Let the initial value of **n** is **4**. Output ???

```
printf("%d", n--); /* 4 */
printf("%d", n);   /* 3 */
```

Note: Try to avoid using the increment and decrement operators in complex expressions in which the variables to which they are applied appear more than once. C compilers are expected to exploit the commutativity and associativity of various operators in order to produce efficient code.

Example: Function to Compute 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);
}
```

Function **`factorial`** computes the factorial of an integer represented by the **formal parameter** **`n`**. The loop body executes for decreasing values of **`i`** from **`n`** through **`2`**, and each value of **`i`** is incorporated in the accumulating product. Loop exit occurs when **`i`** is **`1`**.

Increments and Decrements Other Than 1

We have observed **for** statement counting loops that count up by one and down by one. Now let us use a loop that counts down by **five** to display a Celsius-to-Fahrenheit conversion table. the following code segment shows temperature conversions from 10 degrees Celsius to -5 degrees Celsius because of the values of the constant macros named **CBEGIN** and **CLIMIT**.

```
#include <stdio.h>
/* Constant macros */
#define CBEGIN 10
#define CLIMIT -5
#define CSTEP 5
int main(void){
..................
..................
/* celsius=10 */
for (celsius = CBEGIN;celsius >= CLIMIT; celsius -= CSTEP) {
    fahrenheit = 1.8 * celsius + 32.0;
    printf("%6c%3d%8c%7.2f\n", ' ', celsius, ' ', fahrenheit);
}
........
return(0);
}
```

Conditional Loops

In many programming situations, you will not be able to determine the exact number of loop repetitions before loop execution begins. **For example**, if we want to multiply a list of numbers and the number of repetitions depended on the data entered. Although we did not know in advance how many times the loop would execute, we were still able to write a condition to control the loop. Here is another case of this type of repetition. You want to continue prompting the user for a data value as long as the response is unreasonable. Let us consider the following code segment:

```
printf("Enter number of observed values> ");
scanf("%d", &num_obs);     /* initialization */
while (num_obs < 0) {
   printf("Negative number invalid; try again> ");
   scanf("%d", &num_obs); /* update */
}
...........
```

TRY....

There are 9,870 people in a town whose population increases by 10% each year. Write a loop that displays the annual population and determines how many years (`count_years`) it will take for the population to surpass 30,000.

do-while loop

```
SYNTAX:
  do{
        statement;
 }while (loop repetition condition);
```

INTERPRETATION: First, the statement is executed. Then, the **`loop repetition condition`** is tested, and if it is **true**, the statement is repeated and the condition retested. When this condition is tested and found to be false, the loop is exited and the next statement after the **`do-while`** is executed.

Note: If the loop body contains more than one statement, the group of statements must be surrounded by braces.

```
Example: /* Find first even number input */
do{
     status = scanf("%d", &num);
}while(status > 0 && (num % 2) != 0);
```

do-while loop usefulness

do-while loop for interactive input

There are some situations, generally involving interactive input, when we know that a loop must execute at least one time.

```
1. Get a data value.
2. If data value isn't in the acceptable range, go back to step 1.
```

Example: prompts the user to enter one of the letters A through E . After scanf gets a data character, the loop repetition condition tests to see whether letter_choice contains one of the letters requested. If so, the repetition condition is false, and the next statement after the loop executes. If letter_choice contains some other letter, the condition is true and the loop body is repeated.

```
do {
printf("Enter a letter from A through E> ");
scanf("%c", &letter_choice);
} while (letter_choice < 'A' || letter_choice > 'E');
```

Flag-Controlled Loops

Sometimes a loop repetition condition becomes so complex that placing the full expression in its usual spot is awkward. In many cases, the condition may be simplified by using a flag.

flag: A flag is a type int variable used to represent whether or not a certain event has occurred. A flag has one of two values: 1 (true) and 0 (false).

Example: Code fragment to compute the sum of N numbers till the sum is greater than equal to 100 using flag (i.e. status), controlled mechanism.

```
int status=0, sum=0,num;
while(status != 1){
      printf("Enter a number>");
      scanf("%d",&num);
      sum=sum+num;
      if(sum>=100)
        status=1;
}
printf("Sum =%d\n",sum);
```

Self Test

- What does the following code segment display? Try each of these inputs: 345, 82, 6 . Then, describe the action of the code.

```
printf("\nEnter a positive integer> ");
scanf("%d", &num);
do {
      printf("%d", num % 10);
      num /= 10;
  }  while (num > 0);
printf("\n");
```

- Write a do-while loop that repeatedly prompts for and takes input until a value in the range 0 through 15 inclusive is input.

Comparison of Loop Kinds

Kind	When Used	C Implementation Structures
Counting loop	We can determine before loop execution exactly how many loop repetitions will be needed to solve the problem.	**`while, for`**
Sentinel-controlled	loop Input of a list of data of any length ended by a special value	**`while, for`**
Endfile-controlled loop	Input of a single list of data of any length from a data file	**`while, for`**
Input validation loop	Repeated interactive input of a data value until a value within the valid range is entered	**`do-while`**
General conditional loop	Repeated processing of data until a desired condition is met	**`while, for`**

Counter-Controlled Loop

Algorithmic steps

```
1.   Set loop control variable to an initial value of 0.
2.   While loop control variable < final value
               ...............
               Increase loop control variable by 1.
```

Example:

```c
int count=0;
while(count<50){
     printf("count value=%d\n",count);
     count=count+1;
}
```

Sentinel-Controlled Loops

Sentinel value: An end marker that follows the last item in a list of data

A code snippet that calculates the sum of a collection of exam scores of a candidate using a sentinel value

```
#define SENTINEL -99
int sum = 0,           /* output - sum of scores input so far */
           score;      /* input - current score */
printf("Enter first score (or %d to quit)> ", SENTINEL);
scanf("%d", &score);  /* Get first score.*/
while (score != SENTINEL) {
   sum += score;
   printf("Enter next score (%d to quit)> ", SENTINEL);
   scanf("%d", &score); /* Get next score. */
}
printf("\nSum of exam scores is %d\n", sum);
```

Sample run of the above code snippet

```
Enter first score (or -99 to quit)> 55
Enter next score (or -99 to quit)> 33
Enter next score (or -99 to quit)> 77
Enter next score (or -99 to quit)> -99

Sum of exam scores is 165
```

Endfile-Controlled Loop

A data file is always terminated by an endfile character that can be detected by the **scanf** function. Therefore, you can write a batch program that processes a list of data of any length without requiring a special sentinel value at the end of the data.

Example

```
int sum = 0,           /* output - sum of scores input so far */
          score,       /* input - current score */
          input_status; /* status value returned by scanf */
printf("Enter first score (or %d to quit)> ", SENTINEL);
input_status=scanf("%d", &score);   /* Get first score.*/
while (input_status != EOF) {       /* EOF can be replaced as -1 */
   sum += score;
   printf("Enter next score (%d to quit)> ", SENTINEL);
   input_status=scanf("%d", &score); /* Get next score. */
}
printf("\nSum of exam scores is %d\n", sum);
```

Note: When **scanf** is successfully able to fill its argument variables with values from the standard input device, the result value that it returns is the number of data items it actually obtained.

Input Validation Loop

Algorithmic steps

```
1. Get a data value.
2. If data value isn't in the acceptable range,
         go back to first step.
```

Example:

```
do {
   ........
   printf("One more time? (1 to continue/0 to quit)> ");
   scanf("%d", &again);
   .........
} while (again == 1);
```

General Conditional Loop

Algorithmic steps

```
1. Initialize loop control variable.
2. As long as exit condition hasn't been met,
                continue processing.
```

Example:

```
int num=10;
while(num<=100){
     printf("updated num=%d\n",num);
     num=num+2;
}
```

Nested Loops

Loops may be nested just like other control structures. Nested loops consist of an outer loop with one or more inner loops. Each time the outer loop is repeated, the inner loops are reentered, their loop control expressions are reevaluated, and all required iterations are performed.

Example

```
int i, j; /* loop control variables */
for (i = 1;i < 4; ++i) {
   printf("Outer %6d\n", i);
   for(j = 0;j < i;++j) {
       printf("Inner %9d\n", j);
   }  /* end of inner loop */
}/* end of outer loop */
```

Self Test

- ✍ What is displayed by the following program segments, assuming m is 3 and n is 5 ?

```
for(i = 1;i <= n;++i) {
    for(j = 0;j < i;++j) {
         printf("*");
    }
    printf("\n");
}
```

- ✍ Show the output displayed by these nested loops:

```
for(i=0;i<3;++i){
   printf("Outer %4d\n", i);
   for (j = 0; j < 2;++j) {
       printf(" Inner%3d%3d\n", i, j);
   }
   for (k = 2; k > 0; --k) {
       printf(" Inner%3d%3d\n", i, k);
   }
}
```

- ✍ Write a program that displays the multiplication table for numbers 0 to 9 .
- ✍ Design an interactive input loop that scans pairs of integers until it reaches a pair in which the first integer evenly divides the second.

Review Questions

- ✍ A loop that continues to process input data until a special value is entered is called a _________ controlled loop.
- ✍ In an endfile-controlled while loop, the initialization and update expressions typically include calls to the function _________
- ✍ In a typical counter-controlled loop, the number of loop repetitions may not be known until the loop is executing. True or false?
- ✍ During execution of the following program segment, how many lines of asterisks are displayed?

```
for(i = 0;i < 10;++i)
   for(j = 0;j < 5;++j)
       printf("**********\n");
```

- ✍ During execution of the following program segment: (a) How many times does the first call to printf execute? (b) How many times does the second call to printf execute? (c) What is the last value displayed?

```
for(i = 0;i < 7;++i) {
  for(j = 0;j < i;++j)
       printf("%4d", i * j);
  printf("\n");
}
```

THANK YOU