

1.5 REPETITION

Objectives

- ❑ Concept of loop
- ❑ Loop invariant
- ❑ Pretest and post-test loops
- ❑ Initialization and updating
- ❑ Counter controlled loops
- ❑ Event controlled loops

Agenda

- Introduction to iterative construct
- Counter controlled loops
 - While loop
 - Do-while loop
 - For loop
- Nesting of loops
- Control of loop execution
- Infinite loops
- Try it yourself exercises
- Event controlled loops
 - Sentinel controlled
 - Flag controlled
- Homework problems

Loops

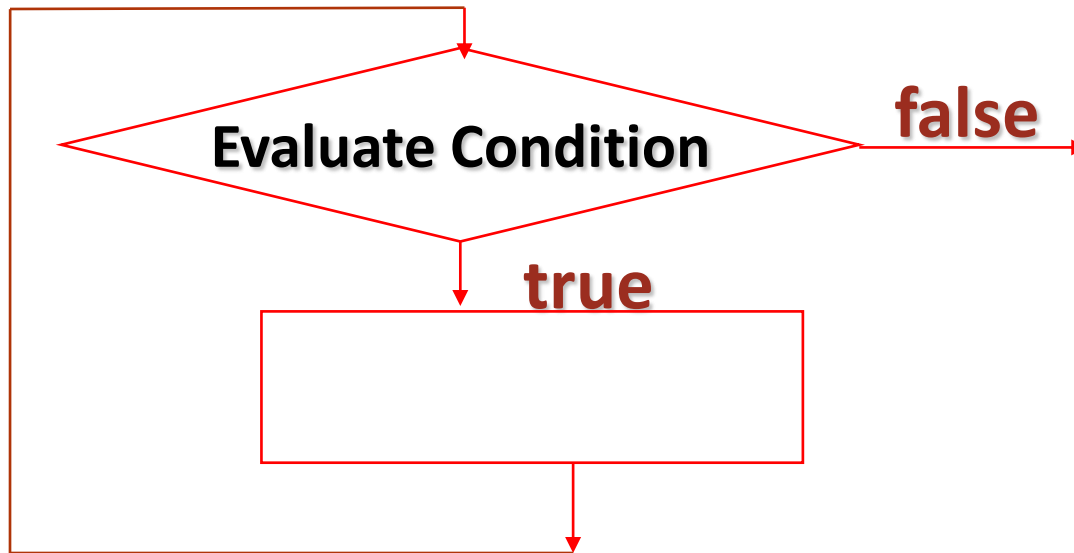
- A loop is a sequence of statements that will be executed repeatedly zero or more times.
- A loop can be executed a set number of times, or as long as some condition is met.
- Each single repetition of the loop is known as an iteration of the loop.
- ✓ *Sometimes you might want to think of a loop as being executed until some condition is met. Of course, looping until one condition is met is the same as looping as long as the opposite of the condition is met. For instance, if you loop until x equals 5, that is the same as looping as long as x does not equal 5.*

Iterative construct: while loop

**while(expression)
statement;**

- ✓ *The statement will be executed as long as the expression remains true, or until a special command is encountered to end the loop.*
- ✓ *Of course, this statement can be a compound statement, and probably it usually will be.*

Looping in a while loop



- ✓ *To repeatedly execute a statement over and over while a given condition is true*
- ✓ *When the condition of the while loop is no longer logically true, the loop terminates and program execution resumes at the next statement following the loop*

Example #1

```
int x = 3;  
while (x > 0)  
{  
    printf("Hello World!\n");  
    x = x - 1;  
}
```

- ✓ The **loop condition** is written first,
followed by the **body of the loop**
- ✓ The loop condition is evaluated first, and
if it is true,
the loop body is executed
- ✓ After the execution of the loop body,
the condition in the while is evaluated again.
- ✓ This repeats until the condition becomes false.

- ❑ *This "while" loop will undergo three iterations.*
 - ✓ *During each, the phrase "HelloWorld!" will be printed on a separate line.*

- ❑ *Why does it execute three times?*
 - ✓ *The variable "x" is initialized above the loop with the value of 3.*
 - ✓ *The loop will repeat as long as the expression "x > 0" is true.*
 - ✓ *At the end of each loop iteration, "x" is decreased by 1.*
 - ✓ *After three iterations, "x" will have the value of 0, and the expression will no longer be true, so the loop will end.*

- ❑ *Note that there is **no semicolon** after the right parenthesis ending the expression that "while" is checking.*
 - ✓ *If there were, it would mean that the program would **repeat the null statement** (statements that do nothing) until the condition were not true.*
 - ✓ *The condition starts off true, it will stay true, and will loop without stopping...*

Example #2

```
int x, y;  
printf("Enter two numbers: ");  
scanf("%d %d", &x, &y);  
while (y != 0)  
{   printf("%d / %d = %d\n", x, y, x/y);  
    printf("Enter two numbers: ");  
    scanf("%d %d", &x, &y);  
}
```

- ✓ This code repeatedly asks the user to enter two integers.
- ✓ As long as the second number is not zero, the program prints the result of dividing the first number by the second.
- ✓ If the second number is 0, the program ends.

Example #3

```
int i = 0;
```

```
int loop_count = 5;
```

```
printf("Case1:\n");
```

```
while (i<loop_count) {  
    printf("%d\n",i); i++; }
```

```
printf("Case2:\n");
```

```
i=20;
```

```
while (0) {  
    printf("%d\n",i); i++; }
```

Cases: 1 and 2

- **Case1 (Normal)** : Variable 'i' is initialized to 0 before 'while' loop; iteration is increment of counter variable 'i'; condition is execute loop till 'i' is lesser than value of 'loop_count' variable i.e. 5.
- **Case2 (Always FALSE condition)** : Variables 'i' is initialized before 'while' loop to '20'; iteration is increment of counter variable 'i'; condition is FALSE always as '0' is provided that causes NOT to execute loop statements and loop statement is NOT executed.
- ✓ Here, it is noted that as compared to 'do-while' loop, statements in 'while' loop are NOT even executed once which executed at least once in 'do-while' loop because 'while' loop only executes loop statements only if condition succeeds.

Example #3 Contd...

```
printf("Case3:\n");
```

```
i=0;
```

```
while (i++<5) {  
    printf("%d\n",i); }
```

```
printf("Case4:\n");
```

```
i=3;
```

```
while (i < 5 && i >=2) {  
    printf("%d\n",i); i++; }
```

Cases:3 and 4

- **Case3 (Iteration in condition check expression)** : *Variable 'i' is initialized to 0 before 'while' loop; here note that iteration and condition is provided in same expression. Here, observe the condition is execute loop till 'i' is lesser than '5' and loop iterates 5 times.*
- ✓ *Unlike 'do-while' loop, here condition is checked first then 'while' loop executes statements.*
- **Case4 (Using logical AND condition)** : *Variable 'i' is initialized before 'while' loop to '3'; iteration is increment of counter variable 'i'; condition is execute loop when 'i' is lesser than '5' AND 'i' is greater or equal to '2'.*

./a.out

- Case1: 0 1 2 3 4
- Case2:
- Case3: 1 2 3 4 5
- Case4: 3 4 #

Find any differences ?



```
int x = 3;
while (x-- > 0)
    printf("Hello World!\n");
```

- ✓ Here, since the decrement operator is placed after the variable, the old value of the variable is used to compare against 0.
- ✓ The first time through, this value is 3, then 2, then 1.
- ✓ The fourth time, it is 0 so we exit the loop.

```
int x = 3;
while (--x >= 0)
    printf("Hello World!\n");
```

- ✓ Now, the decrement operator is placed before the variable, so the value of "x" is decreased and then its value is used.
- ✓ The first time through, this value is 2, then 1, then 0.
- ✓ The fourth time, it is -1 so we exit the loop.



Write a program....

- Ask the user to enter a number, and if it is positive, sum the digits.
- The user enters a number, and the value is stored in "x".
- Store sum in the variable "sum_digits", which is initialized to zero.

❖ *Hint:*

- ✓ *As long as "x" is greater than zero, we mod it by 10, which gets the right-most digit of the number, and we add this digit to "sum_digits".*
- ✓ *Then we divide "x" by 10. Remember, when we do integer division, the fractional part is cut off, so in effect, we are removing the right-most digit of "x" (which we have already added to the sum).*

Solution

```
int x, digit, sum_digits;
printf("Enter a positive integer: ");
scanf("%d", &x);
sum_digits = 0;
while (x > 0)
{
    digit = x % 10;
    sum_digits = sum_digits + digit;
    x = x / 10;
}
printf("The sum of the digits is %d!\n", sum_digits);
```

Iterative construct: do - while loop

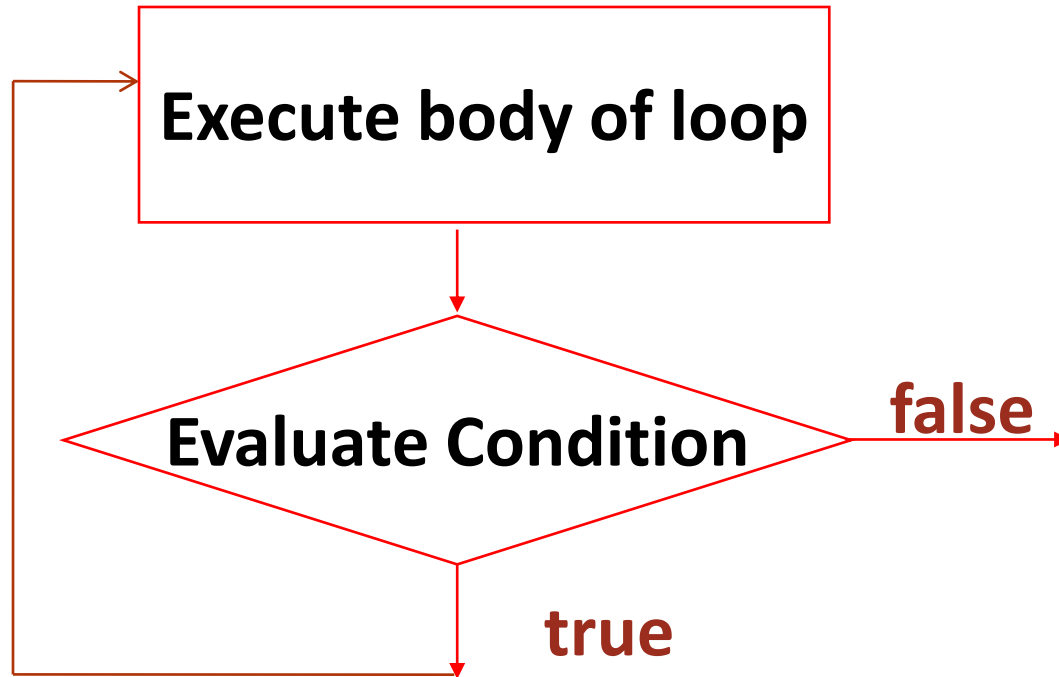
do

statement

while(condition);

✓ *It is similar to the "while" statement, but the condition is checked at the end.*

Looping in a do-while loop



✓ *Statements inside the statement block are executed once, and then expression is evaluated in order to determine whether the looping is to continue*

Example #1

```
int i;  
i = 0;  
do{  
    printf("The value of i is now %d\n",i);  
    i = i + 1;  
} while (i < 5);
```

- ✓ *The body of the loop comes first, followed by the loop condition at the end*
- ✓ *The loop is entered into straightaway, and after the first execution of the loop body, the loop condition is evaluated*
- ✓ ***The body of the loop is guaranteed to execute at least once***
- ✓ *Subsequent executions of loop body would be subject to loop condition evaluating to true*

Example #2

```
int x = 3;
do
{
    printf("Hello World!\n");
    x = x - 1;
} while (x > 0);
```

- ✓ Notice now that there is a semicolon after the right parenthesis ending the expression while is checking.
- ✓ If you forget it, you will get a compiler error when you try to compile the program.

Example #3

```
do
{
    printf("Enter a number from 1 to 100:");
    scanf("%d", &x);
} while ((x < 1) || (x > 100));
```

- ✓ *If the user does not enter a number in the correct range, the “while” condition will be met, and the loop will undergo another iteration, prompting the user again.*
- ✓ *Only after the user enters a number from 1 to 100 will the expression be false and the loop end.*

Example #4

```
int i = 0;
int loop_count = 5;
printf("Case1:\n");
do {
    printf("%d\n",i); i++;
} while (i<loop_count);

printf("Case2:\n");
i=20;
do {
    printf("%d\n",i); i++;
} while (0);
```

Cases: 1 and 2

- **Case1 (Normal)** : *Variable 'i' is initialized to 0 before 'do-while' loop; iteration is increment of counter variable 'i'; condition is to execute loop till 'i' is lesser than value of 'loop_count' variable i.e. 5.*
- **Case2 (Always FALSE condition)** : *Variables 'i' is initialized before 'do-while' loop to '20'; iteration is increment of counter variable 'i'; condition is FALSE always as '0' is provided that causes NOT to execute loop statements.*
- ✓ *But, it is noted here in output that loop statement is executed once because do-while loop always executes its loop statements at least once even if condition fails at first iteration.*

Example #4 Contd...

```
printf("Case3:\n");
```

```
i=0;
```

```
do {
```

```
    printf("%d\n",i);
```

```
} while (i++<5);
```

```
printf("Case4:\n");
```

```
i=3;
```

```
do {
```

```
    printf("%d\n",i); i++;
```

```
} while (i < 5 && i >=2);
```

Cases: 3 and 4

- **Case3 (Iteration in condition check expression)** : *Variable 'i' is initialized to 0 before 'do-while' loop; here note that iteration and condition is provided in same expression.*
 - ✓ *Here, observe the condition is to execute loop till 'i' is lesser than '5', but in output 5 is also printed that is because, here iteration is being done at condition check expression, hence on each iteration 'do-while' loop executes statements ahead of condition check.*
- **Case4 (Using logical AND condition)** : *Variable 'i' is initialized before 'do-while' loop to '3'; iteration is increment of counter variable 'i'; condition is execute loop when 'i' is lesser than '5' AND 'i' is greater or equal to '2'.*

./a.out

- Case1: 0 1 2 3 4
- Case2: 20
- Case3: 0 1 2 3 4 5
- Case4: 3 4 #

Example #5: Programs with Menus

A)dd part to catalog
R)emove part from catalog
F)ind part in catalog
Q)uit

Select option: A

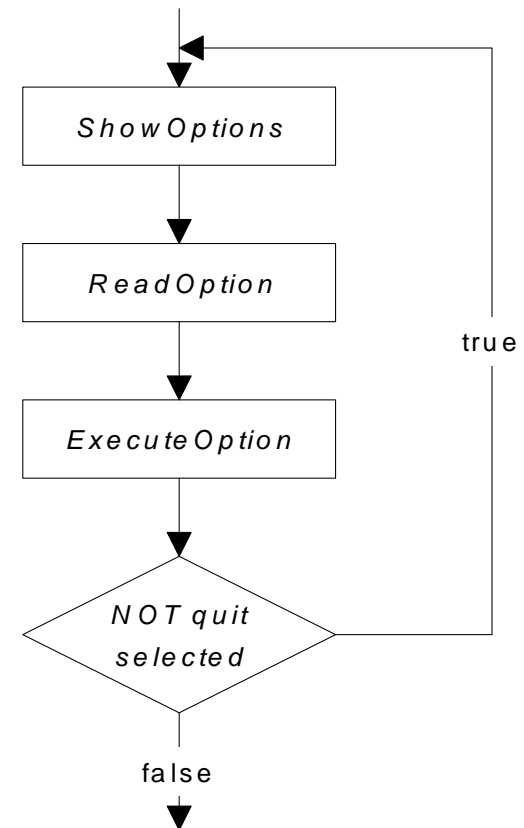
<interaction to add a part>

A)dd part to catalog
R)emove part from catalog
F)ind part in catalog
Q)uit

Select option: *<next option>*

Menu Loop

```
do {  
    showOptions();  
    printf("Select option:");  
    scanf(" %c",&optn);  
    execOption(optn);  
    while (!( (optn == 'Q') || (optn == 'q')));  
}
```



Menu Options

```
void showOptions() {  
    printf("A)dd part to catalog\n");  
    printf("R)emove part from catalog\n");  
    printf("F)ind part in catalog\n");  
    printf("Q)uit\n");  
}
```

Executing Options

```
void execOption( char option ) {  
    switch (option) {  
        case 'A': case 'a': addPart(); break;  
        case 'R': case 'r': delPart(); break;  
        case 'F': case 'f': fndPart(); break;  
        case 'Q': case 'q': break;  
        default: printf("Unknown option  
            %c\n",option); break;  
    }  
}
```

Iterative construct: for loop

```
for (expr1; expr2; expr3)
{
    statement1;
    statement2; . . .
}
```

- ✓ *The for loop construct is by far the **most powerful and compact** of all the loop constructs provided by C.*
- ✓ *This loop keeps **all loop control statements on top of the loop**, thus making it visible to the programmer.*
- ✓ *This loop works well where **the number of iterations of the loop is known before the loop is entered into**.*

for (initialization; condition; update)

- ✓ The **initialization** is usually an assignment of a variable to some starting value, and the **update** is often an assignment which changes this variable.
- ✓ The statement will be executed as long as the **condition**, which is an expression, is true.
- ✓ All three fields are optional.
- ✓ If the initialization or update are left out, they are considered null statements (statements that do nothing).
- ✓ If the condition is left out, it is considered to be always true, and the loop will continue until a statement is reached to break out of the loop.

The first part :

- **Expression 1:** is executed before the loop is entered
- ✓ This is usually the initialization of the loop variable

The second part :

- **Expression 2:** is a test, is evaluated immediately after expression1, and then later is evaluated again after each successful looping
- ✓ The loop is terminated when this test returns a false

The third part :

- **Expression 3:** is a statement to be run every time the loop body is completed
- ✓ It is not evaluated when the for statement is first encountered.
However, expression3 is evaluated after each looping and before the statement goes back to test expression2 again.
- ✓ This is usually an increment of the loop counter

for loop....

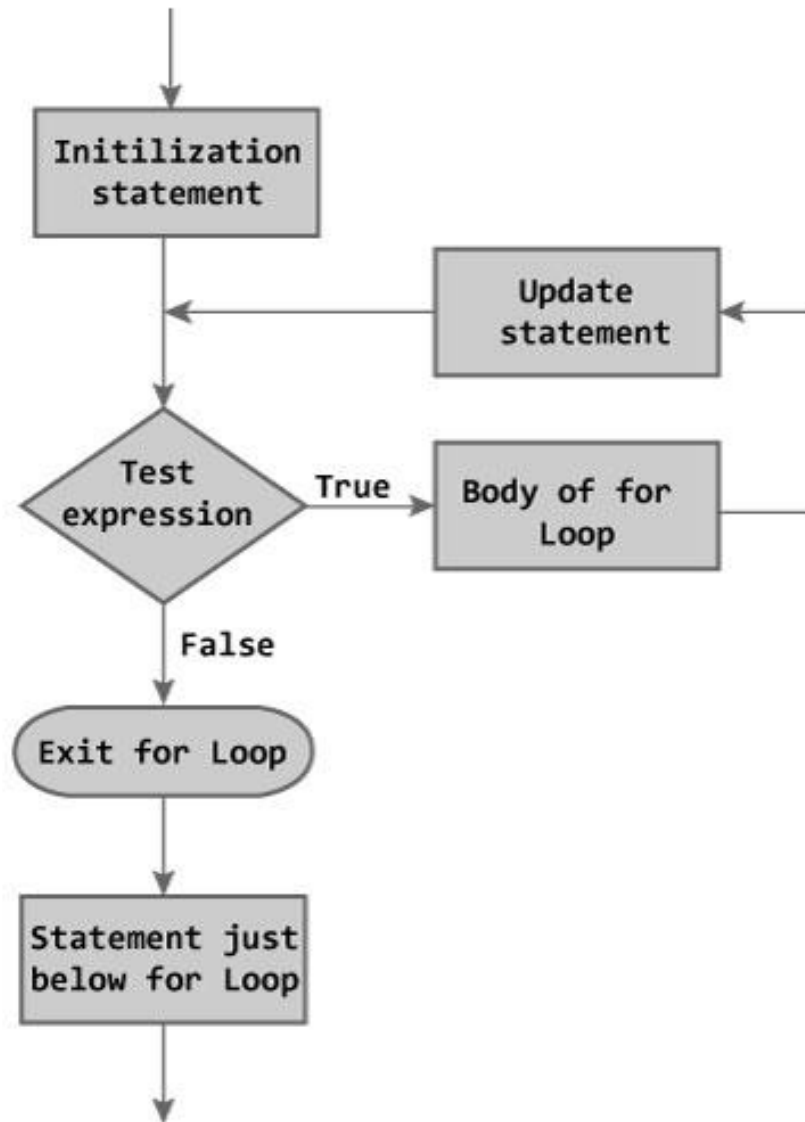


Figure: Flowchart of for Loop

Example #1

```
int n, count, sum=0;
printf("Enter the value of n. \n");
scanf("%d",&n);
for(count=1;count<=n;++count) //for loop
    terminates if count>n
{
    sum+=count; // this statement is
    equivalent to sum=sum+count
}
printf("Sum=%d",sum); return 0;
```

- ✓ *In this program, the user is asked to enter the value of n.*
- ✓ *Suppose you entered 19 then, count is initialized to 1 at first.*
- ✓ *Then, the test expression in the for loop, i.e., $(\text{count} \leq n)$ becomes true.*
- ✓ *So, the code in the body of for loop is executed which makes sum to 1.*
- ✓ *Then, the expression $++\text{count}$ is executed and again the test expression is checked, which becomes true.*
- ✓ *Again, the body of for loop is executed which makes sum to 3 and this process continues.*
- ✓ *When count is 20, the test condition becomes false and the for loop is terminated.*

- **Note:** *Initial, test and update expressions are separated by semicolon(;).*

Example #2

```
int i = 0, k = 0; float j = 0;
int loop_count = 5;
printf("Case1:\n");
for (i=0; i < loop_count; i++) {
    printf("%d\n",i); }

printf("Case2:\n");
for (j=5.5; j > 0; j--) {
    printf("%f\n",j); }

printf("Case3:\n");
for (i=2; (i < 5 && i >=2); i++) {
    printf("%d\n",i); }
```

Cases: 1,2 and 3

- **Case1 (Normal)** : Variable 'i' is initialized to 0; condition is to execute loop till 'i' is lesser than value of 'loop_count' variable; iteration is increment of counter variable 'i'
- **Case2 (Using float variable)** : Variable 'j' is float and initialized to 5.5; condition is to execute loop till 'j' is greater than '0'; iteration is decrement of counter variable 'j'.
- **Case3 (Taking logical AND condition)** : Variable 'i' is initialized to 2; condition is to execute loop when 'i' is greater or equal to '2' and lesser than '5'; iteration is increment of counter variable 'i'.

Example #2 Contd...

```
printf("Case4:\n");
```

```
for (i=0; (i != 5); i++) {  
    printf("%d\n",i); }
```

```
printf("Case5:\n");
```

```
/* Blank loop */ for (i=0; i < loop_count; i++) ;
```

```
printf("Case6:\n");
```

```
for (i=0, k=0; (i < 5 && k < 3); i++, k++) {  
printf("%d\n",i); }
```

```
printf("Case7:\n");
```

```
i=5;
```

```
for (; 0; i++) { printf("%d\n",i); }
```


Case : 4,5,6 and 7

- **Case4 (Using logical NOT EQUAL condition)** : Variable 'i' is initialized to 0; condition is to execute loop till 'i' is NOT equal to '5'; iteration is increment of counter variable 'i'.
- **Case5 (Blank Loop)** : This example shows that loop can execute even if there is no statement in the block for execution on each iteration.
- **Case6 (Multiple variables and conditions)** : Variables 'i' and 'k' are initialized to 0; condition is to execute loop when 'i' is lesser than '5' and 'k' is lesser than '3'; iteration is increment of counter variables 'i' and 'k'.
- **Case7 (No initialization in for loop and Always FALSE condition)** : Variable 'i' is initialized before for loop to '5'; condition is FALSE always as '0' is provided that causes NOT to execute loop statement; iteration is increment of counter variable 'i'.

./a.out

- Case1: 0 1 2 3 4
- Case2: 5.500000 4.500000 3.500000 2.500000
1.500000 0.500000
- Case3: 2 3 4
- Case4: 0 1 2 3 4
- Case5:
- Case6: 0 1 2
- Case7:

What is the o/p?



1.

```
int i;  
for (i=0; i<16; i++)  
    printf(“%X %oX %d\n”, i, i, i);
```

2.

```
for (i=0; i<8; i++)  
sum += i;
```

3.

```
for (i=0; i<8; i++);  
sum += i;
```

Conversion

✓ *Anything that can be done with a “for” statement can also be done with an equivalent “while” statement.*

✓ **for (initialization; condition; update)
statement;**

• The conversion is simply:

```
initialization;  
while (condition)  
{  
    statement;  
    update;  
}
```

Comma operator to combine multiple expressions in for loop

```
for (i=0, j=10; i!=j; i++, j--)  
{  
    /* statement block */  
}
```

Expr1: integer variables ***i*** and ***j*** are ***initialized***, respectively, with 0 and 10 when the for statement is first encountered.

Expr2: relational expressions ***i!=j*** is evaluated and tested. If it evaluates to zero (false), the loop is terminated.

Expr3: After each iteration of the loop, ***i*** is ***increased by 1*** and ***j*** is ***reduced by 1***.

Then the expression ***i!=j*** is evaluated to determine whether or not to execute the loop again.

Examples

```
int i, j;  
for (i=0, j=8; i<8; i++, j--)  
    printf(“%d + %d = %d\n”, i, j, i+j);
```

```
int power, result;  
for (power = 1, result = 2; power <= 10; power++, result = result * 2)  
{  
    printf(“2 to the power of %d equals %d.\n”, power, result);  
}
```

- ✓ *So, here we are initializing "power" to be 1 and "result" to be 2. At the end of each iteration of the loop, we increase "power" by 1 and multiply "result" by 2.*
- ✓ *We continue as long as "power" is less than or equal to 10.*

Nesting of loops

- ✓ *Sometimes, within compound statements that are part of loops, there will be additional loops.*
- **Loops within loops are referred to as nested loops.**
- ✓ *Sometimes you may hear people refer to outer loops (the loop encountered first) and inner loops (any loops embedded within an outer loop).*

Nested loops While loop

initialize outer loop

```
while ( outer loop condition )
```

```
{ ...
```

initialize inner loop

```
while ( inner loop condition )
```

```
{
```

inner loop processing and update

```
}
```

```
...  
}
```


Nested loops for loop

```
for (i=1; i<=3; i++) /* outer loop */  
{  
    printf("Start of iteration %d of the outer loop.\n", i);  
  
    for (j=1; j<=4; j++) /* inner loop */  
        printf(" Iteration %d of inner loop.\n", j);  
  
    printf("End of iteration %d of outer loop.\n", i);  
}
```

Example #1

```
int x, sum_digits, digit, temp;
for (x = 1; x <= 1000; x++)
{
    temp = x;
    sum_digits = 0;
    while (temp > 0)
    {
        digit = temp % 10;
        sum_digits = sum_digits + digit;
        temp = temp / 10;
    }
    if (sum_digits == 5)
        printf("%d\n", x);
}
```

Example #2

```
int row, col;
printf("\t0\t1\t2\t3\t4\t5\t6\t7\t8\t9\n");
  for (row = 0; row <= 9; row++)
  {
    printf("%d", row);
    for (col = 0; col <= 9; col++)
    {
      printf("\t%d", row*col);
    }
    printf("\n");
  }
```

- ✓ *Using the '\t' symbol within a string passed to "printf" causes a tab to be printed, and the computer skips to the start of the next 8 character column.*
- ✓ *The '\t' symbol is a special way of representing the tab character, in the same way that the '\n' symbol is a special way of representing the newline character.*
- ✓ *The first "printf" in this program skips the first column (since no characters appear to the left of the first tab), then prints out column headers 0 through 0 in the next 9 columns.*

- ✓ *We then loop through 9 rows of the table.*
- ✓ *At the beginning of each row, we print the row number.*
- ✓ *Then, we loop through 9 columns, and for each, we tab over to the column and print the product of the row number and column number.*
- ✓ *After the inner “for” loop (at the end of each iteration of the outer “for” loop), we print a ‘\n’ which causes the program to start the next line.*

Example #3

* * * * *

* * * * *

* * * * *

* * * *

* * *

*

```
for (I = 0; I <= 5; I++) {  
    for (J = 0; J < I; J++)  
        printf(" ");  
    for (J = 0; J < (11 - 2 * I); J++)  
        printf("*");  
    printf("\n");  
}
```

✓ *Note 2 (sequential) inner loops*

Trace the nested loop given



```
printf("Max N! to print: ");
scanf("%d",&N);
for (I = 1; I <= N; I++) {
    fact = 1;
    for (J = 2; J <= I; J++)
        fact *= J;
    printf("%d! = %d\n",I,fact);
}
```

Tracing.....

Stmt	N	I	J	fact	output
1	4				
2		1			
3				1	
4			2		
6					1! = 1
2		2			
3				1	
4			2		
5				2	
4			3		
6					2! = 2
2		3			
3				1	
4			2		
5				2	

Stmt	N	I	J	fact	output
4			3		
5				6	
4			4		
6					3! = 6
2		4			
3				1	
4			2		
5				2	
4			3		
5				6	
4			4		
5				24	
4			5		
6					4! = 24
2		5			

Control of loop execution:

Break Statement

- A loop construct, whether while, or do-while, or a for loop continues to **iteratively execute until the loop condition evaluates to false**
- There may be situations where it may be necessary **to exit from a loop even before the loop condition is reevaluated after an iteration**
- The **break statement** is used **to exit early** from all loop constructs (while, do-while, and for)

Example #1

```
int x, y;
```

```
while (1)
```

```
{
```

```
    printf("Enter two numbers: ");
```

```
    scanf("%d %d", &x, &y);
```

```
    if (y == 0) break;
```

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

```
}
```

```
while( condition check )
```

```
{
```

```
    statement-1;
```

```
    statement-2;
```

```
    if( some condition)
```

```
    {
```

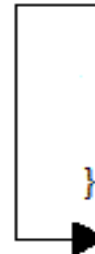
```
        break;
```

```
    }
```

```
    statement-3;
```

```
    statement-4;
```

```
}
```



Jumps out of the loop, no matter how many cycles are left, loop is exited.

- ✓ *When you use "while(1)" (or any other non-zero constant), the loop should continue until some special statement inside the loop stops it.*
 - ✓ *Remember, when a non-zero constant is used as a boolean expression, it is interpreted as true.*
- ✓ *When the "**break**" statement is reached, the computer jumps to the first statement after the "while" loop, regardless of whether or not the condition (expression) being checked by the loop is true.*
- ✓ *The program goes into the loop no matter what and asks the user to enter the two numbers at the start of each iteration of the loop.*
- ✓ *When the user enters 0 as the second number, the loop is exited, and the program ends.*

Example #2

✓ *Break* - terminates loop, execution continues with the first statement following the loop- for and while loops

```
sum = 0;
for (k=1; k<=5; k++)
{
    scanf("%lf",&x);
    if (x > 10.0)
        break;
    sum +=x;
}
printf("Sum = %f \n",sum);
```

```
sum = 0;
k=1;
while (k<=5)
{
    scanf("%lf",&x);
    if (x > 10.0)
        break;
    sum +=x;
    k++;
}
printf("Sum = %f \n",sum);
```

Break statement....

- *If you encounter a “break” statement in the middle of a nested loop, the control of the program jumps to the first statement after the innermost loop surrounding the “break” statement.*
- ✓ *For instance, let’s say in the Example #2 of nested loops, you only want to print out half of the multiplication table, that below the diagonal line from the top left to the bottom right.*
- ✓ *There is no need to print the result of $2*7$ if you are going to print the result of $7*2$ anyway!*

Multiplication table program revisited



- ✓ *There is no need to print the result of $2*7$ if you are going to print the result of $7*2$ anyway!*
- ✓ *Can you can make this adjustment by adding one “if” statement to our multiplication program:?*

Multiplication table program revisited

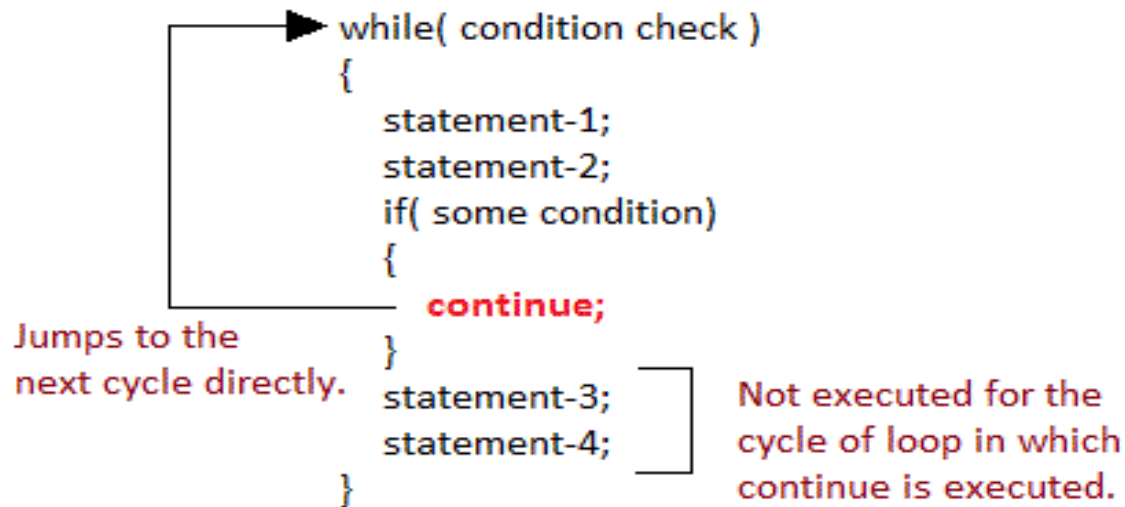
```
int row, col;
printf("\t0\t1\t2\t3\t4\t5\t6\t7\t8\t9\n");
  for (row = 0; row <= 9; row++)
  {
    printf("%d", row);
    for (col = 0; col <= 9; col++)
    {
      printf("\t%d", row*col);
      if (col == row)
        break;
    }
    printf("\n");
  }
```

✓ Now, for each row, once the column equals the row, we skip the rest of the inner “for” loop and jump to the line that prints the newline character. We then move on to the next row!

Control of loop execution:

Continue Statement


- The continue statement causes **all subsequent instructions in the loop body** (coming after the continue statement) **to be skipped**
- **Control passes back to the top of the loop** where the loop condition is evaluated again
 - ✓ *In case of a continue statement in a for loop construct, control passes to the reinitialization part of the loop, after which the loop condition is evaluated again.*



Example #1

✓ *Continue* forces next iteration of the loop, skipping any remaining statements in the loop- for and while loops

```
sum = 0;
for (k=1; k<=5; k++)
{
    scanf("%lf",&x);
    if (x > 10.0)
        continue;
    sum +=x;
}
printf("Sum = %f \n",sum);
```



```
sum = 0;
k=1;
while (k<=5)
{
    scanf("%lf",&x);
    if (x > 10.0)
        continue;
    sum +=x;
    k++;
}
printf("Sum = %f \n",sum);
```

Example #2

```
int x;  
for (x = 1; x <= 20; x++)  
{  
    if (x % 5 == 0)  
        continue;  
    printf(“%d\n”, x);  
}
```

✓ When x is not divisible by 5, the condition of the “if” statement is false, so we reach the “printf” statement and print out the number.

✓ When “ x ” is 5, 10, or 15, the condition of the “if” statement is true, so we “continue”, or skip, to the end of the current iteration of the “for” loop, still do the update “ $x++$ ”, and start the next iteration of the loop.

✓ When “ x ” is 20, we skip to the end of the current iteration of the “for” loop, still do the update “ $x++$ ”, but now x will be 21, so the condition of the “for” loop is no longer met, and we end the loop.

✓ The first thing to note here is the test for divisibility by 5.

✓ Remember that the “%” is the modulus operator; it returns the remainder when the first operand is divided by the second operand.

✓ ‘

✓ If the remainder when “ x ” is divided by 5 is 0, then “ x ” is divisible by 5!

Example #3

```
int c;  
printf("Enter a character:\n(enter x to exit)\n");  
while {  
    c = getch();  
    if (c == 'x ' )  
        break;  
}  
printf("Break the infinite while loop. Bye!\n");
```

Enter a character:

(enter x to exit)

H

I

x

Break the infinite while loop. Bye!

Rewrite the code using continue statement

```
for (I = 0; I < 100; I++)  
{  
    if (!(I % 2) == 1)  
        printf("%d is even",I);  
}
```



```
for (I = 0; I < 100; I++)  
{  
    if ((I % 2) == 1)  
        continue;  
    printf("%d is even",I);  
}
```

Infinite loops

```
for ( ; ; )  
{  
    statement1;  
    statement2;  
    ..  
    .  
}
```



```
while {  
    statement1;  
    statement2;  
    ..  
    .  
}
```

Example #1

- ✓ Now consider the following silly program:

```
while (1)
{
    printf("Hello World!\n");
}
```

- ✓ *It prints "HelloWold!" forever!*
- ✓ *This is called an infinite loop. Here, we created one on purpose, but normally, they are created by accident.*

✓ *What do you do when you are caught in an infinite loop?*

You press **Ctrl-C** on your keyboard.

Pressing Ctrl-C will halt the execution of your C program!

✓ *What happens if you use "while(0)" in your program?*

The loop is skipped no matter what. It is useless, but valid.

Here are five ways to exit a loop:

- ✓ *The condition the loop depends on is not met at the time of the check.*
- ✓ *A "break" statement is encountered.*
- ✓ *A statement that ends the current function, such as "return", is encountered.*
- ✓ *The program crashes.*
- ✓ *The user presses Ctrl-C.*

Loops.....

- ❖ Read exactly 100 blood pressures from a file.
- ❖ Keep reading until a special (impossible) value is read.
- ❖ Read all the blood pressures from a file no matter how many are there.
- ❖ Read blood pressures until a dangerously high BP (200 or more) is read.

Event controlled loops

- ✓ *read until input ends*
- ✓ *read until a number encountered*
- ✓ *search through data until item found*

- **Sentinel controlled** Keep processing data until a special value which is not a possible data value is entered to indicate that processing should stop
- **End-of-file controlled** Keep processing data as long as there is more data in the file (*you will see it in UNIT 2: files*)
- **Flag controlled** Keep processing data until the value of a flag changes in the loop body

Example #1

✓ *Sentinel is negative blood pressure.*

```
int thisBP;  
int total;  
int count;  
count = 1;  
total = 0;  
  
scanf(“%d”, &thisBP);  
  
while (thisBP > 0)           // Test expression  
{  
    total = total + thisBP;  
    count++;                // Update  
}  
  
printf(“The total = %d”, total);
```

Example #2

- ✓ The **value -999** is sometimes referred to as a ***sentinel value***
- ✓ The value serves as the “guardian” for the termination of the loop
- ✓ Often a good idea to make the sentinel a constant

```
#define STOPNUMBER -999  
while (number != STOPNUMBER)  
...
```

Example#3

✓ **done** is the flag, it controls the looping

```
total = 0;
done = 0; /* done is set to 0 state */
do {
    scanf("%d",&num);
    if (num < 0) done = 1; /* done 1 */
} while ((num != 0) && (!done));
```

Loop Testing and Debugging

- Test data should test all sections of the program
- Beware of infinite loops -- the program doesn't stop
- Check loop termination condition
- Use algorithm walk-through to verify that appropriate conditions occur in the right places
- Trace execution of loop by hand with code walk-through
- Use a debugger (if available) to run program in “slow motion” or use debug output statements

Let us try.....





```
int i, sum;
sum = 0;
for (i=1; i<8; i++)
{
    if ((i==3) || (i==5))
        continue;
    sum += i;
}
printf("The sum of 1, 2, 4, 6, and 7 is: %d\n", sum);
}
```

Output?????????

The sum of 1, 2, 4, 6, and 7 is: 20



```
i = n;  
while (i--)  
    statement;
```

If you use this method,
make sure that n is greater
than zero,

Or

make the test `i-- > 0`

```
while (a = 6)  
    statement
```

6 is assigned to a

– the expression `a = 6` is tested —
it has the value 6 which is non-zero
and

therefore true

– the loop will be executed forever
as `a = 6` is always true

Write this code using while construct



```
for( i=5; i<10; i++)  
{  
    printf("AAA %d \n", i);  
    if (i % 2==0) continue;  
    pritntf("BBB %d \n", i);  
}
```

```
i=5;  
while(i<10) {  
    printf("AAA %d \n", i);  
    if (i % 2==0) {  
        i++;  
        continue;  
    }  
    pritntf("BBB %d \n", i);  
    i++;  
}
```

Write a program to compute the following



$$\ln 2 = \frac{1}{1} - \frac{1}{2} + \frac{1}{3} - \frac{1}{4} + \frac{1}{5} - \frac{1}{6} + \frac{1}{7} - \dots \pm \frac{1}{n}$$

```
Enter n
```

```
ln2=0;
```

```
for(i=1; i<=n; i++)
```

```
    if ( i % 2 == 0)
```

```
        ln2 = ln2 - 1.0 / i;
```

```
    else
```

```
        ln2 = ln2 + 1.0 / i;
```

Write a program for the given summation



$$\sum_{i=0}^m x^i = x^0 + x^1 + x^2 + x^3 + x^4 + \dots + x^m$$

Enter x and m

```
total=0;
```

```
for(i=0; i<=m; i++)
```

```
    total = total + pow(x, i);
```

```
print total
```

Enter x and m

```
total=0; sofarx=1;
```

```
for(i=0; i<=m; i++) {
```

```
    total = total +sofarx;
```

```
    sofarx = sofarx * x;
```

```
}
```

```
print total
```

Find the maximum score using for loop



```
printf("Number of students: ");
scanf("%d",&NumStudents);
for (I = 0; I < NumStudents; I++) {
    printf("Enter student score %d: ");
    scanf("%d",&score);
    if (score > max)
        max = score;
}

/* max is highest score entered */
```

What is the o/p?



```
main() {  
    char t;  
    while((t = getchar()) != '!') {  
        if (t >= 'A' && t <= 'Z')  
            printf("%c\n", (char)(t + 'a' - 'A'));  
        else  
            printf( "%c\n", (char)t);  
    }  
}
```



Try it Yourself.....

Home work exercises

Write C program for the following

- Suppose a man (say, A) stands at $(0, 0)$ and waits for user to give him the direction and distance to go.
- User may enter N E W S for north, east, west, south, and any value for distance. When user enters 0 as direction, stop and print out the location where the man stopped.


```
float x=0, y=0;
char direction;
float distance;
while (1) {
    printf("Please input the direction as N,S,E,W (0 to exit): ");
    scanf("%c", &direction);    fflush(stdin);
    if (direction=='0') { /*stop input, get out of the loop */
        break;
    }
    if (direction!='N' && direction!='S' && direction!='E' &&
        direction!='W') {
        printf("Invalid direction, re-enter \n");
        continue;
    }
    printf("Please input the mile in %c direction: ", direction);
    scanf ("%f", &distance);    fflush(stdin);
```

```
if (direction == 'N') {           /*in north, compute the y*/
    y = y + distance;
} else if (direction == 'E') {   /*in east, compute the x*/
    x = x + distance;
} else if (direction == 'W') {   /*in west, compute the x*/
    x = x - distance;
} else if (direction == 'S') {   /*in south, compute the y*/
    y = y - distance;
}
}
printf("\nCurrent position of A: (%4.2f, %4.2f)\n", x, y);
/* output A's location */
```

Write C program to compute e^x

The program should reads the value of x and n from the keyboard and then approximately computes the value of e^x using the following formula:

$$e^x = 1 + \frac{x}{1!} + \frac{x^2}{2!} + \frac{x^3}{3!} + \dots + \frac{x^n}{n!}$$

Then compare your approximate result to the one returned by `exp(x)` in C library, and print out whether your approximation is higher or lower.

Write C program to print the pattern after getting the size from the user

Size: 5

**

*

Size: 3

**

*

Size: 0

```
do {  
    printf("Size:");  
    scanf("%d",&Size);  
    if (Size > 0) {  
        for (I = Size; I >= 1; I--) {  
            for (J = 1; J <= I; J++)  
                printf("*");  
            printf("\n");  
        }  
    }  
} while (Size > 0);
```

Summary

- ✓ *Introduction to iterative construct*
- ✓ *Counter controlled loops*
 - ✓ *While loop*
 - ✓ *Do-while loop*
 - ✓ *For loop*
- ✓ *Nesting of loops*
- ✓ *Control of loop execution*
- ✓ *Infinite loops*
- ✓ *Event controlled loops*
 - ✓ *Sentinel controlled*
 - ✓ *Flag controlled*