Loops

Acknowledgement

The contents (figures, concepts, graphics, texts etc.) of the slides are gathered and utilized from the books mentioned and the corresponding PPTs available online:

Books:

1. **Let Us C,** Yashawant Kanetkar, BPB Publications.
2. **The C Programming Language,** B. W. Kernighan, D. Ritchie, Pearson Education India.

Web References:

1. **Problem Solving through Programming in C,** Anupam Basu, NPTEL Video Lectures. Link: https://nptel.ac.in/courses/106/105/106105171/
2. **Compile and Execute C Online (GNU GCC v7.1.1), Link:** https://www.tutorialspoint.com/compile_c_online.php

Chapter Objective

- To learn about different types of Control Statements

Chapter Topics

- Repetition or Iteration structure - for statement, continue statement, nested loop, while loop

- The programs that we have developed so far used either
  - a sequential – the calculations were carried out in a fixed order

  or
  - a decision control instruction- an appropriate set of instructions were executed depending upon the outcome of the condition being tested (or a logical decision being taken).
- The versatility of the computer lies in its ability to perform a set of instructions repeatedly.
- This involves repeating some portion of the program either a specified number of times or until a particular condition is being satisfied .
- This is done by loops

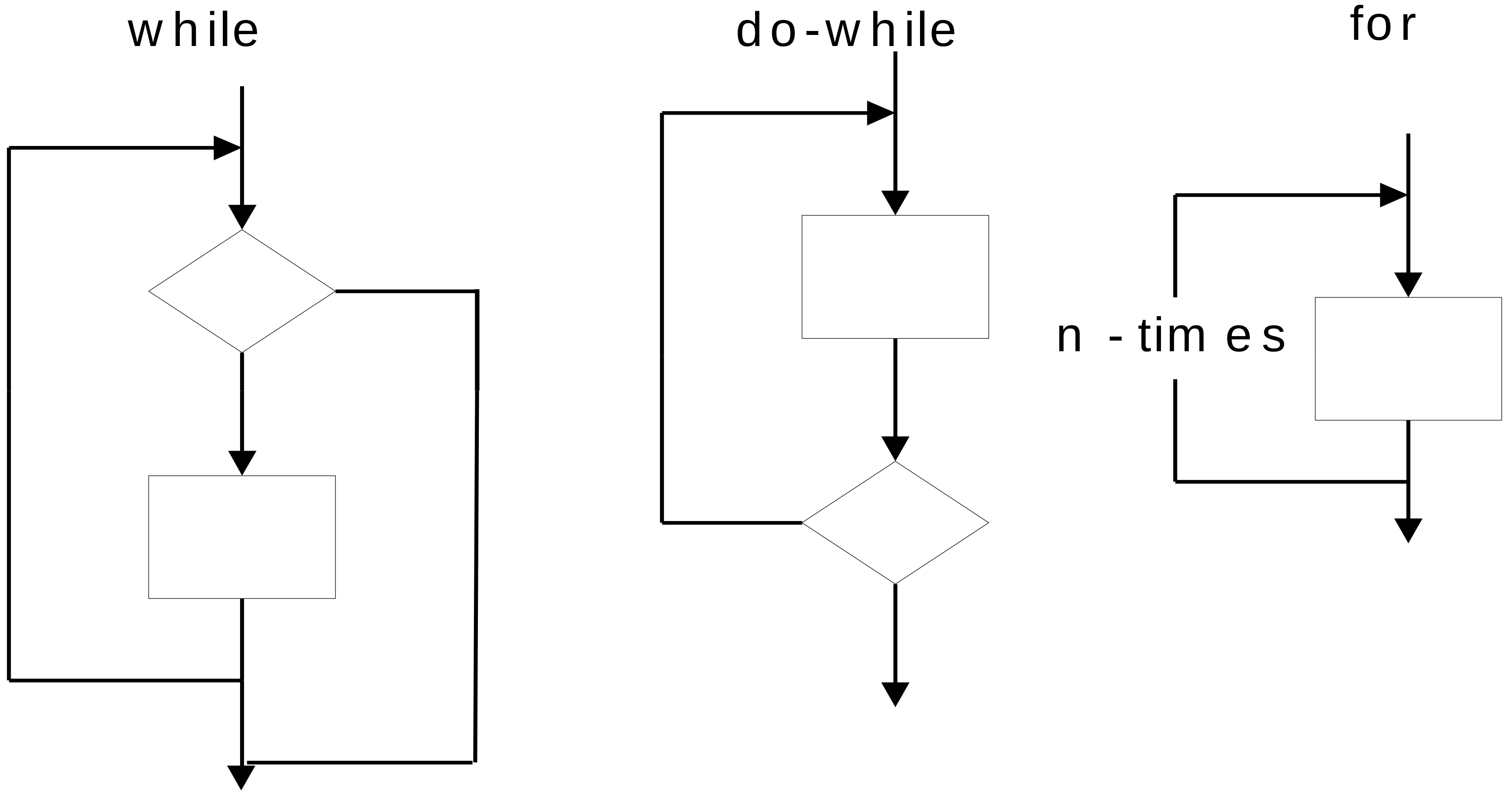
There are three methods by way of which we can repeat a part of a program.

(a)Using a **for** statement

(b) Using a **while** statement

(c) Using a **do-while** statement

C Control Structure Looping



Operators (contd…)

- A unary operator has one operand
  - Post and Pre increment and decrement
    - a++,b--,++a,--b
      - » a = a++;
- A binary operator has two operands
    - a = b+c
- A ternary operator has three operands
    - ***boolean-expr*** ? ***expression-1*** : ***expression-2***

The ternary operator

- ***boolean-expr*** `?` ***expression-1*** `:` ***expression-2***
- This is like `if-then-else` for values rather than for statements
- If the ***boolean-expr*** evaluates to `true`, the result is ***expression-1***, else it is ***expression-2***
- Example: `max = a > b ? a : b ;` sets the variable `max` to the larger of `a` and `b`
- ***expression-1*** and ***expression-2*** need not be the same type, but either result must be useable (not a “void” function)
- *The ternary operator is right associative!*
  - To avoid confusion, use parentheses if your expression has more than one ternary operator

Increment and Decrement Operators

- AKA unary operators
- The increment operator ++ adds 1 to its operand, and the decrement operator - - subtracts 1.
- If either is used as a prefix operator, the expression increments or decrements the operand before its value is used.
- If either is used as a postfix operator, the increment and decrement operation will be performed after its value has been used.

```
main( )
{
int a=5;
printf("a=%d”,a++);
printf("\n a=%d",a);
}
```

```
a=5
a=6
```

```
main()
{
int a=5;
printf("a=%d",++a);

}
```

```
a=6
```

```
main()
{
int a=5;
printf("a=%d“,a--);
printf(“\n a=%d”,a);
}
```

```
a=5
a=4
```

```
main()
{
int a=5;
printf("a=%d”,--a);
}
```

```
a=4
```

```
main()
{
int a=5;
printf("a=%d a=%d",a,++a,a++);
}
```

```
a=7 ++a=7  a++=5
```

```
main()
{
int a=5;
printf("a=%d",a++);
printf("a=%d",++a);
printf("a=%d",a++);
}
```

```
a=5
a=7
a=7
```

```
main( )
{
int a=5;
printf("a=%d",a--);
printf("a=%d“,--a);
printf("a=%d",a--);
}
```

```
a=5
a=3
a=3
```

The while Statement

- Structure

```
        while(expression)
                statement;
or
            while(expression)
            {
                statement_1;
                statement_2;
            }
```

The while Statement Example

```
 while (a < b)
{
printf("%d\n",
a);
a = a + 1;
}
```

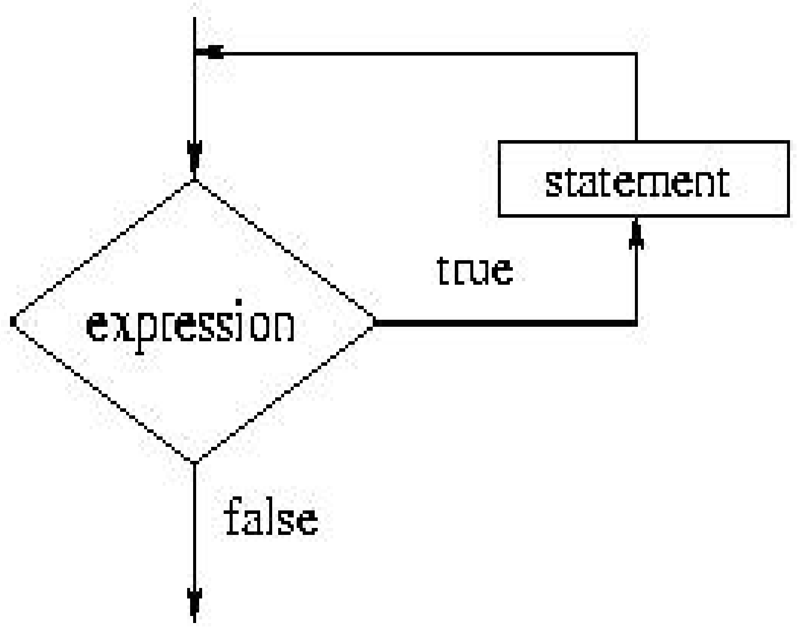
* **Operation: *expression* is evaluated and if *TRUE* then *statement* (or *statement_1 and statement_2)* is executed. The evaluation and executions sequence is repeated until the expression evaluates to be *FALSE*. If the expression is initially *FALSE* then *statement* is not executed at all.**

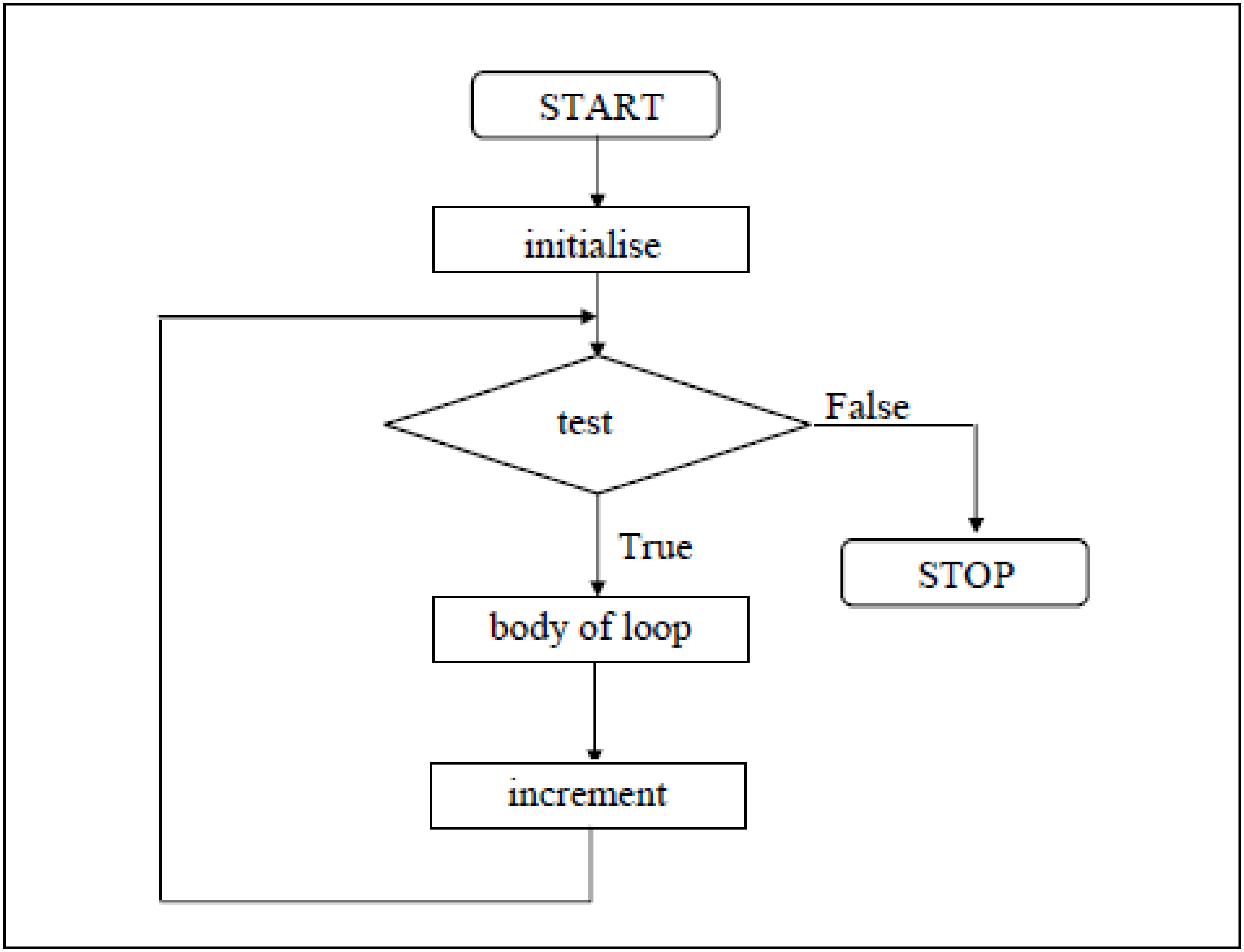
```
void main()
{
 int count=1;
while(count<=5)
 {
  printf("\n count = %d", count);
 count=count+1;
 }
}
```

Flowchart of a while Loop

- The syntax of a while loop is as follows:

```
while(expression)
  statement
```



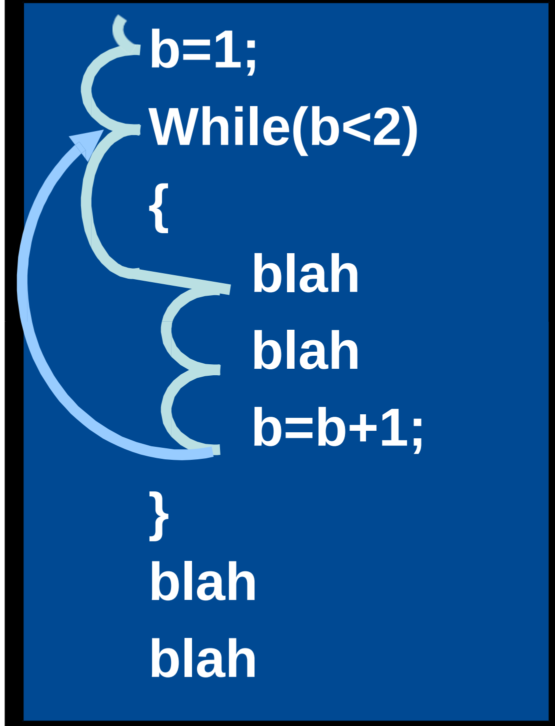

START
initialise
test
False
True
STOP
body of loop
increment

Understanding a while loop

```
b=1;
While(b<2)
{
        blah
        blah
        b=b+1;
}
blah
blah
```

When the computer reaches while statement, it tests to see if the Boolean expression is true. If true, it jumps into the block of code immediately below the while statement

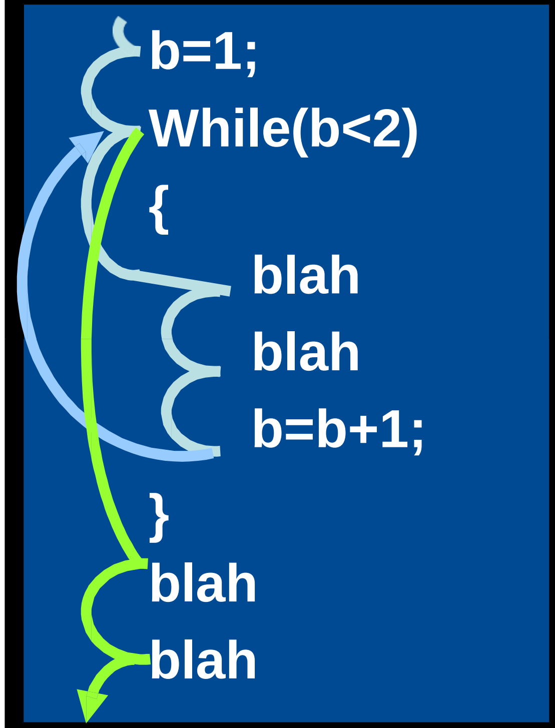
Understanding a while loop



```
b=1;
While(b<2)
{
      blah
      blah
      b=b+1;
}
blah
blah
```

When the computer reaches last line in the while loop's block of code, it jumps back up to the while statement. It re-tests the Boolean expression. If still true, it enters the block of code again.

Understanding a while loop



```
b=1;
While(b<2)
{
        blah
        blah
        b=b+1;
}
blah
blah
```

When the Boolean expression becomes false , the computer skips the while loop statement’s block of code and continues with the remainder of the program.

The general form of **while** is as shown below:

```
initialise loop counter ;
while ( test loop counter using a condition )
{
  do this ;
  and this ;
   increment loop counter ;
}
```

- The statements within the **while** loop would keep on getting executed till the condition being tested remains true.
- When the condition becomes false, the control passes to the first statement that follows the body of the **while** loop.
- In place of the condition there can be any other valid expression. So long as the expression evaluates to a non-zero value the statements within the loop would get executed.
- The condition being tested may use relational or logical operators as shown in the following examples:

```
while ( i <= 10 )
while ( i >= 10 && j <= 15)
```

- As a rule the while must test a condition that will eventually become false, otherwise the loop would be executed forever.

```
main( )
{ int i = 1 ;
 while ( i<=10)
 {
  printf ( “%d\n", i ) ;
  }
}
```

The correct form:

```
main( )
{ int i = 1 ;
 while ( i<=10)
 {
  printf ( “%d\n", i ) ;
  i=i+1;
 }
}
```

- loop counter, can even be decremented.

```
main( )
{ int i = 5 ;
 while ( i>5)
 {
  printf ( “%d\n", i ) ;
  i=i-1;
 }
}
```

- loop counter, can be float also.

Looping: A Real Example

- **Let's say that you would like to create a program that prints a Fahrenheit-to-Celsius conversion table. This is easily accomplished with a for loop or a while loop:**

```
main()
{
    int a;
    a = 0;
    while (a <= 100)
    {
        printf(“%d degrees F = %d degrees C\n", a, (a - 32)
          * 5 / 9);
        a = a + 10;
    }
}
```

Example:

```
int i = 0;
while(i < 5){
    printf("%d ",i);
    i++;
}
```

Output:

```
0 1 2 3 4
```

```
main( )
{
int i = 1,sum=0;
while(i <=100 ){
    sum=sum+i;
printf("Sum=%d \t &
  i=%d\n",sum,i++);
}}
```

```
main( )
{
int i = 100,sum=0;
while(i >=1 ){
    sum=sum+i;
printf("Sum=%d \t &
  i=%d\n",sum,i--);
}}
```

```
main( )
{
int i = 1,sum=0;
while(i <=100 ){
    sum=sum+i;
printf("Sum=%d \t &
  i=%d\n",sum,i=i+5
  );
}}
```

```
main( )
{
int i = 100,sum=0;
while(i >=1 ){
    sum=sum+i;
printf("Sum=%d \t &
  i=%d\n",sum,i=i-
  5);
}}
```

```
main( )
{
int i = 1,sum=0;
while(i <=100 ){
if(i%2==0)
{
sum=sum+i;
}
printf("Sum=%d \t &
   i=%d\n",sum,i++);
}
}
```

```
main( )
{
int i = 1,sum=0;
while(i <=100 ){
if(i%2==1)
{
sum=sum+i;
}
printf("Sum=%d \t &
   i=%d\n",sum,i++);
}
}
```

Solve same problem using decrement(- -)operator

```
main( )
{
int i = 1,sum=0;
while(i <=100 ){
if(i%2==0 &&
   i%5==0)
{
sum=sum+i;
}
printf("Sum=%d \t &
   i=%d\n",sum,i++);
}
}
```

```
main( )
{
int i = 100,sum=0;
while(i >=1){
   if(i%2==0 && i%5==0 &&
   i%10==0)
   {
   sum=sum+i;
   printf("Sum=%d \t & i=%d\n",sum,i);
}//bracket for if
   i=i-1;
}//bracket for while
}//bracket for main
```

Example:

Calculating a factorial 5!. The factorial n! is defined as n(n-1)!*

```
main() {
    /* declaration */
       int i, f, n;
    /* initialization */
    i = 1;
    f = 1;
/* processing */
    printf("Please input a number\n");
    scanf("%d", &n);
    while (i <= n) {
     f *= i;
     i++;
    }.
    /* termination */
    printf("factorial %d! = %d\n", n, f);
}
```

```
5
factorial 5! = 120
```

do-while Loop

- The loops that we have used so far executed the statements within them a finite number of times.
- However, in real life programming one comes across a situation when it is not known beforehand how many times the statements in the loop are to be executed.
- A do while loop is another type of repetition statement.
- It is exactly the same as the while loop except it initially performs operations and then evaluates the logical expression of the loop to determine what should happen next.
- This type of loop is referred to as an exit condition loop sequence. Normally with a while loop, some part of the logical expression in the loop must be initialized before execution of the loop.
- Most programmers prefer using a do while loop when validating user input.

do-while Loop

- The syntax of a do-while statement is as follows:

```
do
  statement
while(expression);
```

- The evaluation of the controlling expression takes place after each execution of the loop body.
- The loop body is executed repeatedly until the return value of the controlling expression is equal to 0.

The do-while Statement

- Structure

  do

  {

  statement;

  } while(expression);

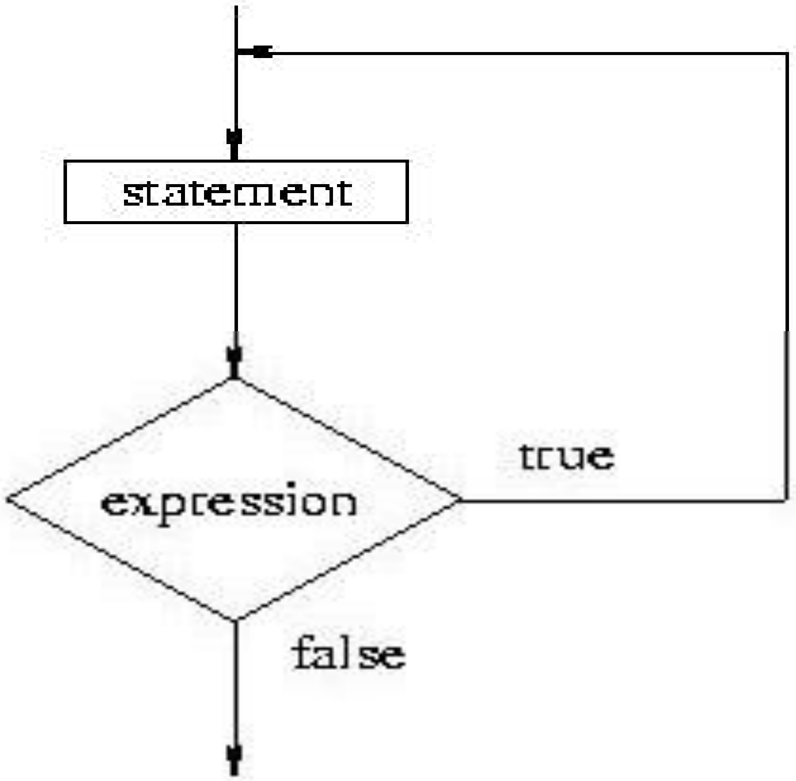
```
do {
 printf("%d\n", a);
a = a + 1;
} while (a < b);
```

- Operation: Similar to the *while* control except that *statement* is executed before the *expression* is evaluated. This guarantees that *statement* is always executed at least one time even if *expression* is *FALSE*. - ODD LOOP

Flowchart of a do-while loop

- The syntax of a do-while statement is as follows:

```
do
   statement
while(expression);
```



```
/* Execution of a loop an unknown number of times */

main( ) {
char another ; int num ;
do
  {
  printf ( "Enter a number " ) ;
  scanf ( "%d", &num ) ;
  printf ( "square of %d is %d", num, num * num ) ;
  printf ( "\nWant to enter another number y/n " ) ;
  scanf ( " %c", &another ) ;
  } while ( another == 'y' ) ;
}
```

And here is the sample output...

```
Enter a number 5
square of 5 is 25
Want to enter another number y/n y
Enter a number 7
square of 7 is 49
Want to enter another number y/n n
```

In this program the **do-while** loop would keep getting executed till the user continues to answer y.

The moment he answers n, the loop terminates, since the condition (**another == 'y'**) fails. Note that this loop ensures that statements within it are executed at least once even if **n** is supplied first time itself.

Example:

```
int i = 0;
do {
    printf(“%d ”, i);
    i++;
} while(i < 5);
```

Output:

```
0 1 2 3 4
```

– **What is the output of the following example?**

Example:

```
int i = 10;
do {
    printf(“%d ”, i);
    i++;
} while(i < 5);
```

do-while Loop

```
/* odd loop using a while loop */
main( ) { char another = 'y' ; int num ;
   while ( another == 'y' )
   {
   printf ( "Enter a number " ) ;
   scanf ( "%d", &num ) ;
   printf ( "square of %d is %d", num, num * num ) ;
   printf ( "\nWant to enter another number y/n " ) ;
   scanf ( " %c", &another ) ;
   }
}
```

The for Loop

- A for loop is another way to execute instructions that need to be repeated in C++ (repetition structure).
- Most programmers like to use for loops because the code can be written in a more compact manner compared to the same loop written with a while or do while loop. A for loop is generally used when you know exactly how many times a section of code needs to be repeated.
- The **for** allows us to specify three things about a loop in a single line:
  1. Setting a loop counter to an initial value.
  2. Testing the loop counter to determine whether its value has reached the number of repetitions desired.
  3. Increasing the value of loop counter each time the program segment within the loop has been executed.

Ex:

```
    for ( expression1; expression2; expression3 )
        stmt(s);
```

where stmt(s) is a single or compound statement. expression1 is used to initialize the loop; expression2 is used to determine whether or not the loop will continue; expression3 is evaluated at the end of each iteration or cycle.

The for Statement

- **Structure:** ***for(expr1; expr2; expr3)***

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

- Operation: The for loop in C is simply a shorthand way of expressing a while statement:

```
 expr1;
while(expr2)
{
            statement;
             expr3
}
```

- The comma operator lets you separate several different statements in the initialization and increment sections of the for loop (but not in the test section).

C Errors to Avoid

- **Putting = when you mean == in an if or while statement**
- **Forgetting to increment the counter inside the while loop - If you forget to increment the counter, you get an infinite loop (the loop never ends).**
- **Accidentally putting a ; at the end of a for loop or if statement so that the statement has no effect - For example:**

```
for (x=1; x<10; x++);
printf("%d\n",x);
```

only prints out one value because the semicolon after the for statement acts as the one line the for loop executes.

- **For Loop**
  - The syntax of a `for` statement is as follows:

```
for (initialise counter ; test counter ; increment counter)
   {
      statement
    }
```

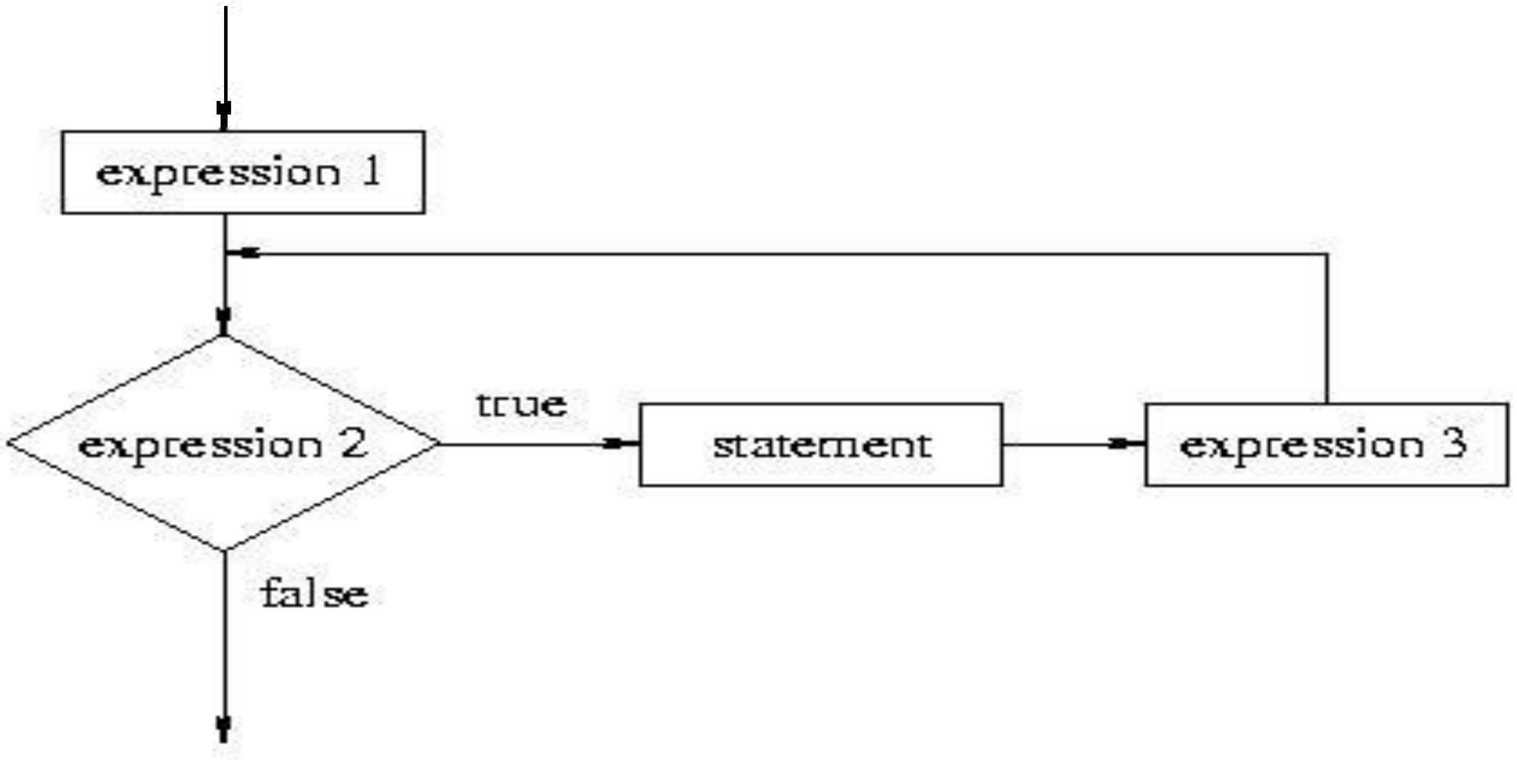
<u>Example: factorial of n numbers</u>

```
main() {
int i, f, n;
i = 1;
f = 1;
/* processing */
   printf("Please input a number\n");
   scanf("%d", &n);
   for (i = 1; i <= n; i++)
   {
     f *= i;
   }
 /* termination
 */printf("factorial %d! = %d\n", n, f);}
```

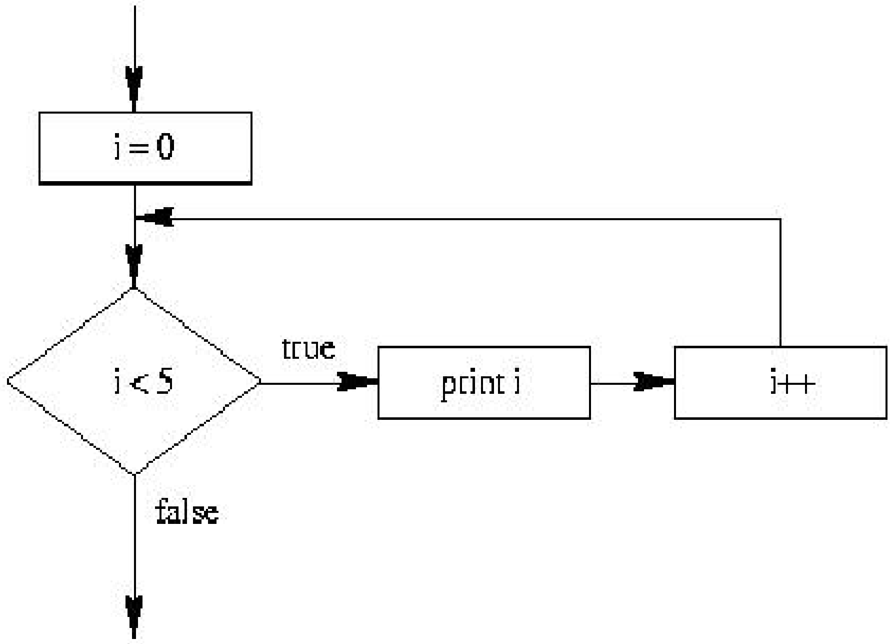
Flowchart of a for Loop

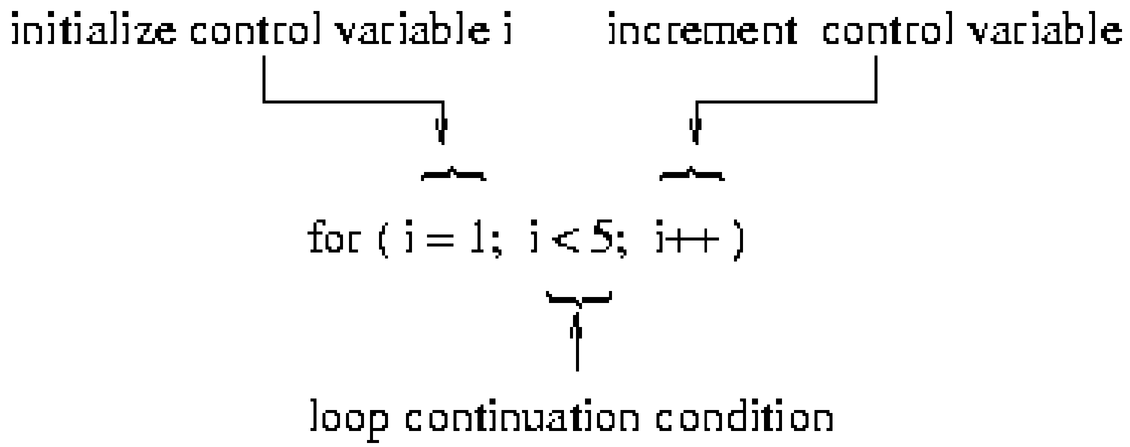
The syntax of a for loop is as follows:

```
for(expression1; expression2; expression3)
  statement
```



```
int i;
for(i = 0; i < 5; i++)
 {
     printf("%d ", i);
}
```




initialize control variable i
increment control variable
for (i = 1; i < 5; i++)
loop continuation condition

Output:

```
0 1 2 3 4
```

Valid for loops

- It is important to note that the initialization, testing and incrementation part of a **for** loop can be replaced by any valid expression.
- Thus the following **for** loops are perfectly ok.

```
for ( i = 10 ; i ; i -- )
printf ( "%d", i ) ;

for ( i < 4 ; j = 5 ; j = 0 )
printf ( "%d", i ) ;

for ( i = 1; i <=10 ; printf ( "%d",i++ ) ;

for ( scanf ( "%d", &i ) ; i <= 10 ; i++ )
printf ( "%d", i ) ;
```

Valid for loops

Example shows some programs to print numbers from 1 to 10 in different ways.

```
(a)  main( )
   {
       int i ;
           for ( i = 1 ; i <= 10 ; i = i + 1 )
            printf ( "%d\n", i ) ;
 }
```

- Note that the initialisation, testing and incrementation of loop counter is done in the **for** statement itself. Instead of **i = i + 1**, the statements **i++** or **i += 1** can also be used.

- Since there is only one statement in the body of the **for** loop, the pair of braces have been dropped. As with the **while**, the default scope of **for** is the immediately next statement after **for**.

Valid for loops

Example shows some programs to print numbers from 1 to 10 in different ways.

```
(b) main( )
 {
          int i ;
          for ( i = 1 ; i <= 10 ; )
          {
            printf ( "%d\n", i ) ;
            i = i + 1 ;
          }
 }
```

Here, the incrementation is done within the body of the **for** loop and not in the **for** statement. Note that inspite of this the semicolon after the condition is necessary.

Valid for loops

Example shows some programs to print numbers from 1 to 10 in different ways.

```
(c) main( )
{
        int i = 1 ;
        for ( ; i <= 10 ; i = i + 1 )
                printf ( "%d\n", i ) ;
}
```

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

```
(d) main( )
{
        int i = 1 ;
        for ( ; i <= 10 ; )
        {
         printf ( "%d\n", i ) ;
         i = i + 1 ;
        }
}
```

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

Valid for loops

```
(e) main( ) {
          int i ;
          for ( i = 0 ; i++ < 10 ; )
           printf ( "%d\n", i ) ;
 }
```

Here, the comparison as well as the incrementation is done through the same statement, **i++ < 10**. Since the **++** operator comes after **i** firstly comparison is done, followed by incrementation. Note that it is necessary to initialize **i** to 0.

```
(f) main( )
  {
          int i ;
          for ( i = 0 ; ++i <= 10 ; )
            printf ( "%d\n", i ) ;
 }
```

Here, both, the comparison and the incrementation is done through the same statement, **++i <= 10**. Since **++** precedes **i** firstly incrementation is done, followed by comparison. Note that it is necessary to initialize **i** to 0.

Nesting of Loops

The way **if** statements can be nested, similarly **while**s and **for**s can also be nested. To understand how nested loops work, look at the program given below:

```
/* Demonstration of nested loops */
main( )
{
  int r, c, sum ;
  for ( r = 1 ; r <= 3 ; r++ ) /* outer loop */
    {
        for ( c = 1 ; c <= 2 ; c++ ) /* inner loop */
        {
          sum = r + c ;
          printf ( "r = %d c = %d sum = %d\n", r, c, sum ) ;
        }
    }
}
```

As you can see, the body of the outer **for** loop is indented, and the body of the inner **for** loop is further indented. These multiple indentations make the program easier to understand.

Nesting of Loops

```
main( )
{
  int r, c, sum ;
  for ( r = 1 ; r <= 3 ; r++ ) /* outer loop */
    {
        for ( c = 1 ; c <= 2 ; c++ ) /* inner loop */
        {
          sum = r + c ;
          printf ( "r = %d c = %d sum = %d\n", r, c, sum ) ;
        }
    }
}
```

When you run this program you will get the following output:

```
r = 1 c = 1 sum = 2
r = 1 c = 2 sum = 3
r = 2 c = 1 sum = 3
r = 2 c = 2 sum = 4
r = 3 c = 1 sum = 4
r = 3 c = 2 sum = 5
```

Here, for each value of **r** the inner loop is cycled through twice, with the variable **c** taking values from 1 to 2. The inner loop terminates when the value of **c** exceeds 2, and the outer loop terminates when the value of **r** exceeds 3.

Nesting of Loops

The way **for** loops have been nested here, similarly, two **while** loops can also be nested. Not only this, a **for** loop can occur within a **while** loop, or a **while** within a **for**.

Nested Loop

Nested loop = loop inside loop

<u>Program flow</u>

The inner loops must be
 finished before the outer loop resumes iteration.

Write a program to print a multiplication table.

```
      1  2   3   4   5   6   7   8   9  10
 ------------------------------------------
 1|   1
 2|   2   4
 3|   3   6   9
 4|   4   8  12  16
 5|   5  10  15  20  25
 6|   6  12  18  24  30  36
 7|   7  14  21  28  35  42  49
 8|   8  16  24  32  40  48  56  64
 9|   9  18  27  36  45  54  63  72  81
10|  10  20  30  40  50  60  70  80  90 100
 ------------------------------------------
```

Example: A program to print a multiplication table.

```
main()
{
int i, j;
printf("1  2  3  4  5  6  7  8  9 10\n");
printf("-------------------------------------\n");
for(i=1; i<= 10; i++)
       { /* outer loop */
            printf("%d  ", i);
                  for(j=1; j<=i; j++)
                  { /* inner loop */
                        printf("%d  ", i*j);
                  }
            printf("\n");
    }
printf("------------------------------------\n");
}
```

Jump Statements

- **Break Statements**
  - The break statement provides an early exit from the for, while, do-while, and for each loops as well as switch statement.
  - A break causes the innermost enclosing loop or switch to be exited immediately. The keyword **break**, breaks the control only from the **block** in which it is placed.
  - When **break** is encountered inside any loop, control automatically passes to the first statement after the loop.

Example:

```
int i;
   for(i=0; i<5; i++)
  {
     if(i == 3)
       {
       break;
       }
       printf("%d", i);
}
```

```
Output :0 1 2
```

Continue Statements

- The continue statement causes the next iteration of the enclosing for, while and do-while loop to begin.
- When **continue** is encountered inside any loop, control automatically passes to the beginning of the loop.
- A continue statement should only appear in a loop body.

```
 int i;
 for(i=0; i<5; i++)
{
   if(i == 3)
   {
     continue;
   }
   printf("%d", i);
 }
```

```
Output : 0 1 2 4
```

```
main( )
{
int i, j ;
    for ( i = 1 ; i <= 2 ; i++ )
    {
           for ( j = 1 ; j <= 2 ; j++ )
            {
             if ( i == j )
                continue ;
                printf ( "\n i = %d, j=%d\n", i, j ) ;
             }
      }
}
```

The output of the above program would be...

```
i = 1, j= 2
i=2, j=1
```

when the value of **i** equals that of **j**, the **continue** statement takes the control to the **for** loop (inner) bypassing rest of the statements pending execution in the **for** loop (inner).

Assignments Solve all problem using while,do – while and for

- WAP to find sum of ten numbers.
- WAP to display all numbers between 1 to 1000 that are perfectly divisible by 10.
- WAP to display all numbers between 2000 to 100 that are perfectly divisible by 13 and 15 and 17 and 19.
- WAP to find sum of all numbers between 500 to 1500 that are perfectly divisible by 3 and 5 and 15 and 45.
- WAP to find sum of all numbers between 10000 to 1000 that are perfectly divisible by 3 and 7 and 9 and 42.
- WAP to find factorial of given number
  - N! = N * (N-1)!
- WAP to check whether a given number is prime or not.
  - A number is prime if it is divisible by 1 and the number itself only
- WAP to generate all prime numbers between 100 to 1.

- WAP to generate multiplication table of any given number
- WAP to generate multiplication table of number 1 to 5.
- WAP to check whether a given program in Armstrong or not.
  - 153 is a Armstrong number because $1^3+5^3+3^3=153$
- WAP to check whether a number is strong or not.
  - 145 is a strong number because 1! + 4! +5 ! = 145
- WAP to check whether a number is perfect or not.
  - 28 is a perfect number because 1+2+4+7+14 = 28
- WAP to find GCD (HCF) of given two numbers.
- WAP to find LCM of given numbers.