Flow control statements

- The order in which statements are executed is called flow control (or control flow).
- Flow control is an important consideration because it determines what is executed during a run and what is not, therefore affecting the overall outcome of the program.

Loops

- Loops cause a section of your program to be repeated a certain number of times. The repetition continues while a condition is true. When the condition becomes false, the loop ends and control passes to the statements following the loop.
- There are three kinds of loops in C:
  - The for loop,
  - The while loop
  - The do loop.

The For loop

- The for loop executes a section of code a fixed number of times. It’s usually (although not always) used when you know, before entering the loop, how many times you want to execute the code.
- Syntax:

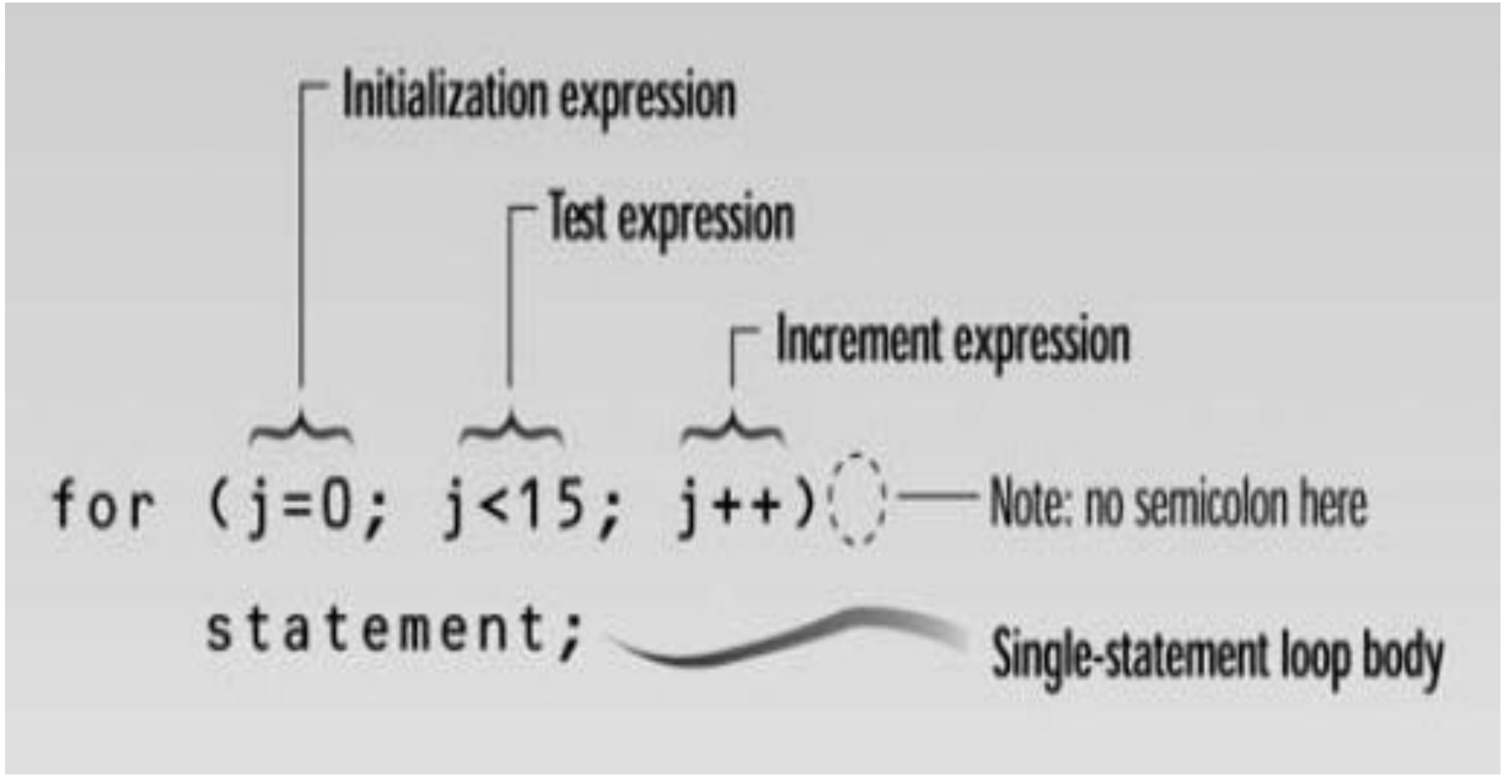
```
for(int x=c;x<d;x++)
```

- Example:

```
for(int j=0;j<15;j++)
```

- Write a program to calculate the squares of the numbers 0 - 15

The for loop..

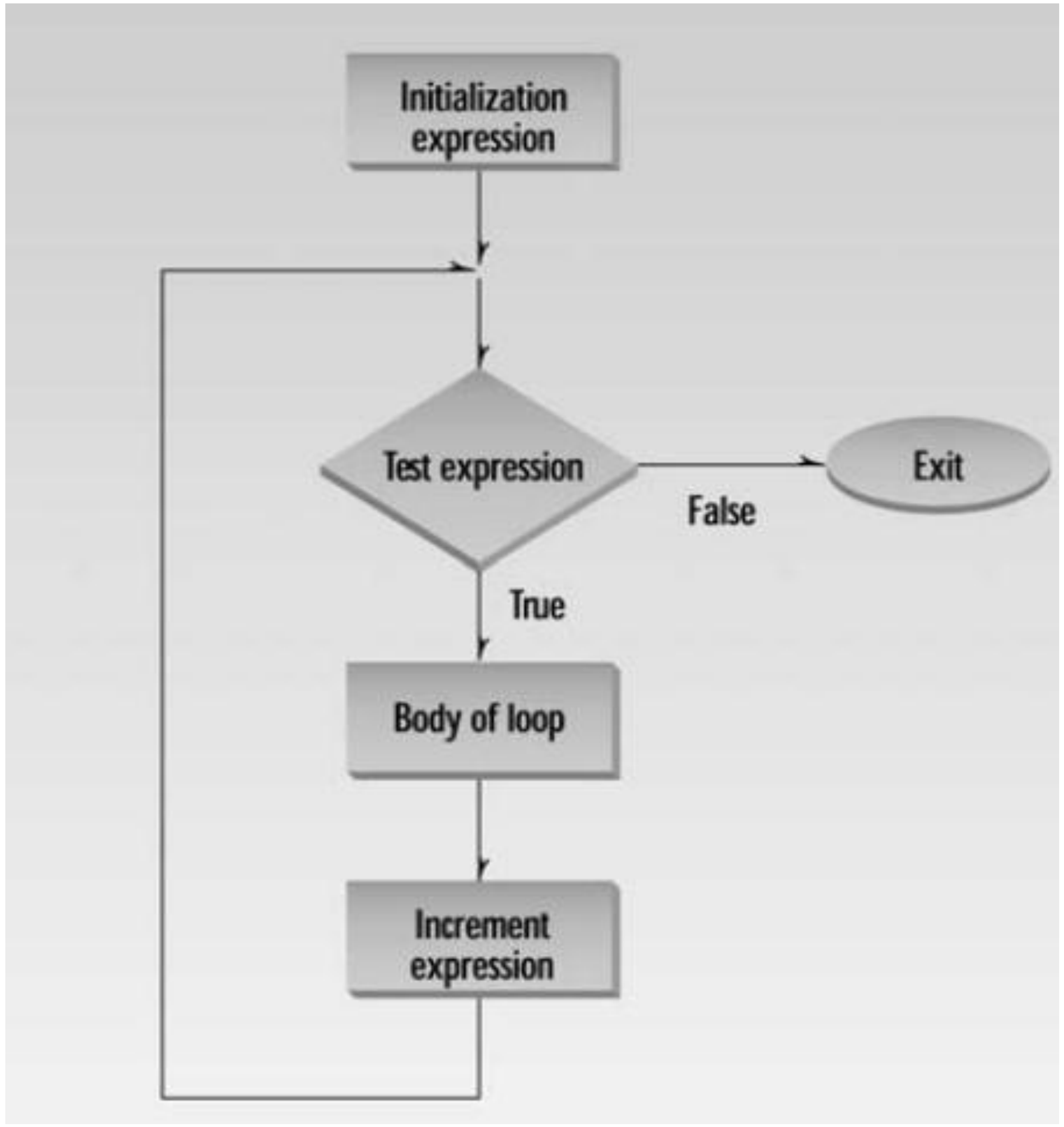

Initialization expression
Test expression
Increment expression
for (j=0; j<15; j++)
Note: no semicolon here
statement;
Single-statement loop body

The for loop..

- **The initialization expression** is executed only once, when the loop first starts. It gives the loop variable an initial value.
- **The test expression** usually involves a relational operator. It is evaluated each time through the loop, just before the body of the loop is executed. It determines whether the loop will be executed again. If the test expression is true, the loop is executed one more time. If it’s false, the loop ends, and control passes to the statements following the loop.

The for loop..

- **The increment expression** changes the value of the loop variable, often by incrementing it. It is always executed at the end of the loop, after the loop body has been executed. Here the increment operator ++ adds 1 to j each time through the loop.
- The program in the example loops exactly 15 times, I.e from j=0 to j = 14. when j=15 the loop exits



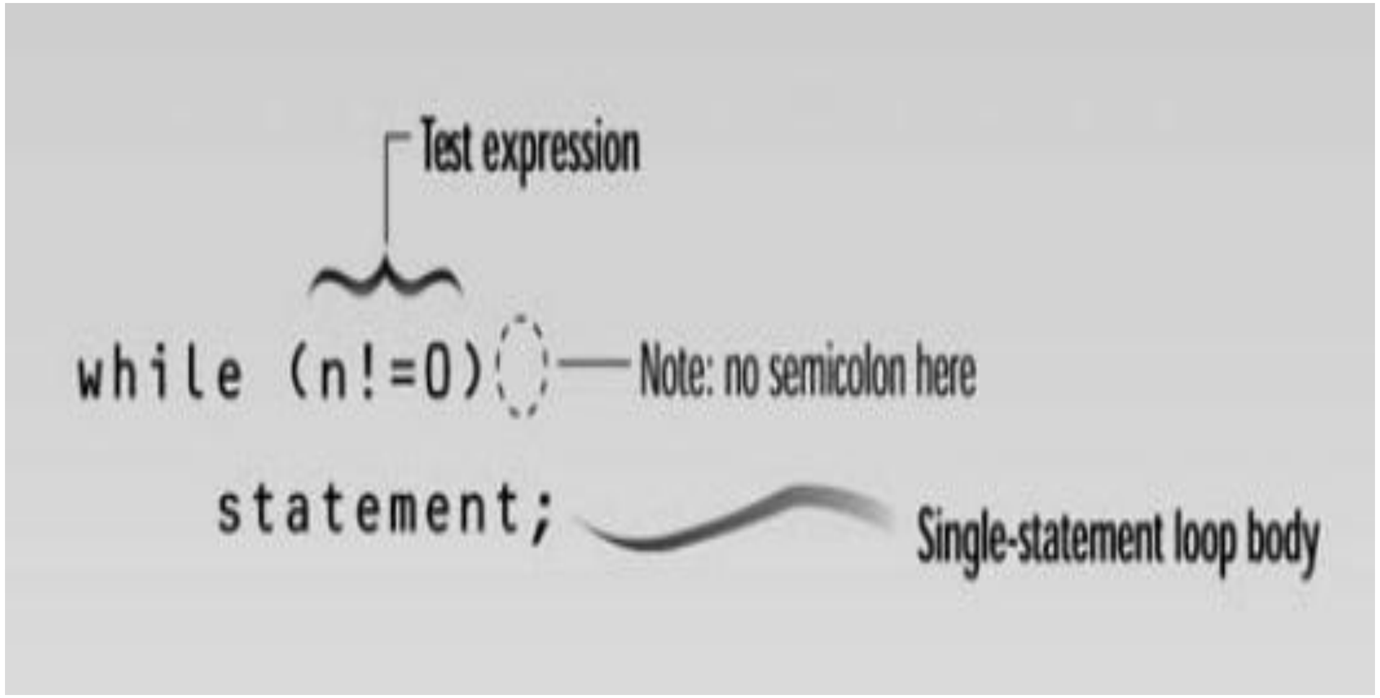
Operation of the `for` *loop.*

The while loop

- The for loop does something a fixed number of times. What happens if you don’t know how many times you want to do something before you start the loop? In this case a different kind of loop may be used: the while loop
- Syntax:

```
while(condition)
{
     Statement
}
```

The while loop

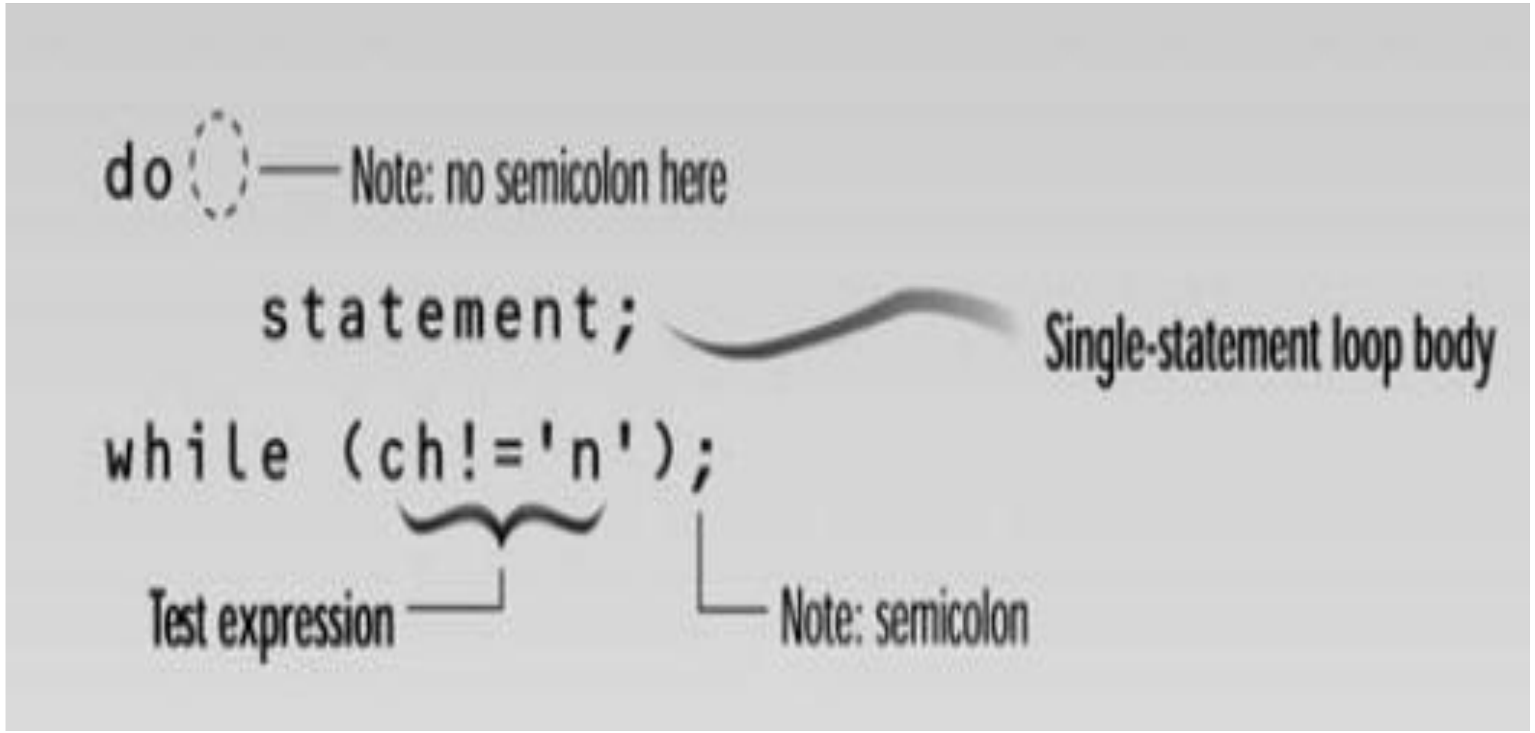


```
while (n!=0)
    statement;
```

The do while loop

- In a while loop, the test expression is evaluated at the beginning of the loop. If the test expression is false when the loop is entered, the loop body won’t be executed at all. In some situations this is what you want. But sometimes you want to guarantee that the loop body is executed at least once, no matter what the initial state of the test expression. When this is the case you should use the do loop, which places the test expression at the end of the loop.

The do while loop..



```
do
    statement;
while (ch!='n');
```

Nested loops

Refers to a loop inside a loop.

Example: Output even numbers within a range I.e. y to x

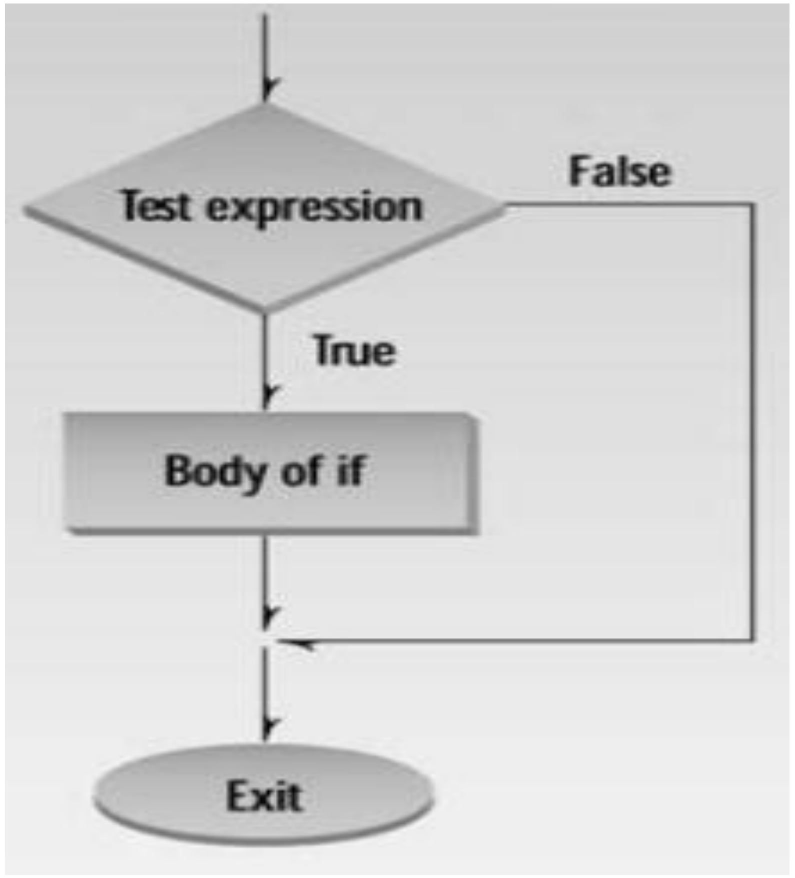
When to use which loop

- The for loop is appropriate when you know in advance how many times the loop will be executed.
- The while and do loops are used when you don’t know in advance when the loop will terminate (the while loop when you may not want to execute the loop body even once, and the do loop when you’re sure you want to execute the loop body at least once).

Decisions

- The decisions in a loop always relate to the same question: Should we do this (the loop body) again?
- In a program a decision causes a one time jump to a different part of the program, depending on the value of an expression.

The if statement



The library function exit()

The exit function causes the program to terminate, no matter where it is in execution.

Syntax: exit(0)

The if-else

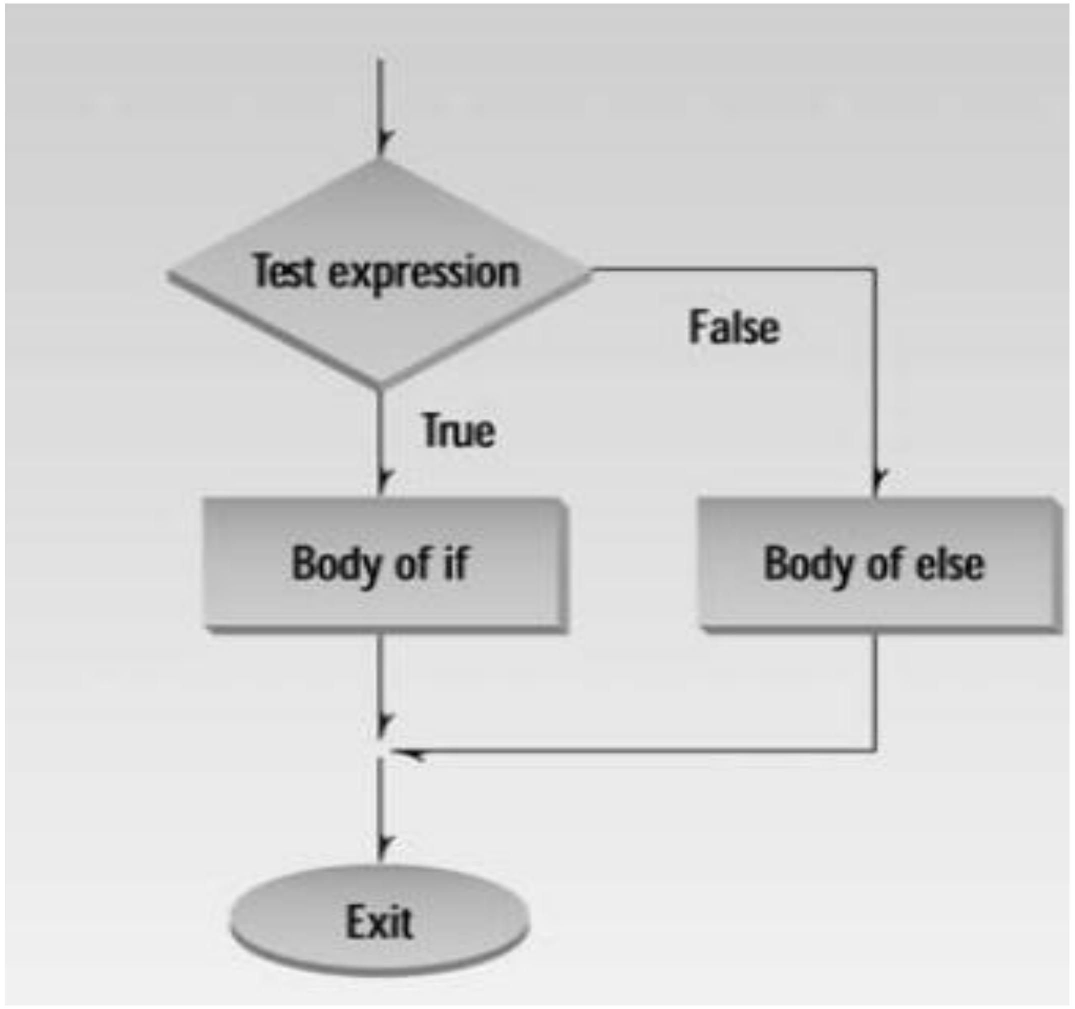
The if statement lets you do something if a condition is true. If it isn’t true, nothing happens. But suppose we want to do one thing if a condition is true, and do something else if it’s false.

That’s where the if...else statement comes in.

Example:

```
If ((a%2)==0)
Printf(”even number”);
else
Printf(”Odd number”);
```

The if else…



If elseif else statement

This statement is useful when we have many possible scenarios, like assigning grades:

```
If (mark >= 70)
    Grade = ‘A’
Elseif (mark >= 60)
    Grade = ‘B’
Elseif (mark >= 50)
    Grade = ‘C’
Elseif (mark >= 40)
    Grade = ‘D’
Else
    Grade = ‘F’
```

The switch statement

The switch statement provides a way of choosing between a set of alternatives, based on the value of an expression. The general form of the switch statement is:

```
switch (expression) {
case constant1 :
statements;
...
case constantn :
statements;
default:
statements;
}
```

The switch statement…

For example, suppose we have parsed a binary arithmetic operation into its three components and stored these in variables operator, operand1, and operand2. The following switch statement performs the operation and stores the result in result.

```
switch (operator) {
    case '+':   result = operand1 + operand2;
                break;
    case '-':   result = operand1 - operand2;
                break;
    case '*':   result = operand1 * operand2;
                break;
    case '/':   result = operand1 / operand2;
                break;
    default:    cout << "unknown operator: " << ch << '\n';
                break;
}
```

The break statement

The break statement causes an exit from a loop, just as it does from a switch statement. The next statement after the break is executed is the statement following(after) the loop.

Write a program that asks the user to type 10 integers and writes the sum of these integers.