Table 1.14 Assignment operators

Operator	Example

/=	float a=9.0;
	float b=3.0;
	a /= b;

\=	int a= 9;
	int b = 3;
	a \= b;

*=	int a= 9;
	int b = 3;
	a *= b;

+=	int a= 9;
	int b = 3;
	a += b;

-=	int a= 9;
	int b = 3;
	a -= b;

&=	int a = 10;
	int b = 20;
	a &= b;

^=	int a = 10;
	int b = 20;
	a ^= b;

<<=	int a= 9;
	int b = 3;
	a <<= b;

>>=	int a= 9;
	int b = 3;
	a >>= b;

```
a = b = c = 10;
```

is evaluated as

```
(a = (b = (c = 10)));
```

First 10 is assigned to `c`, then the value of `c` is assigned to `b`. Finally, the value of `b` is assigned to `a`. Table 1.14 contains a list of other assignment operators that are supported by C.

Comma Operator

The comma operator, which is also called the sequential-evaluation operator, takes two operands. It works by evaluating the first expression and discarding its value, and then evaluates the second expression and returns the value as the result of the expression. Comma-separated expressions when chained together are evaluated in left-to-right sequence with the right-most value yielding the result of the expression. Among all the operators, the comma operator has the lowest precedence.

Therefore, when a comma operator is used, the entire expression evaluates to the value of the right expression. For example, the following statement first increments `a`, then increments `b`, and then assigns the value of `b` to `x`.

```
int a=2, b=3, x=0;
x = (++a, b+=a);
```

Now, the value of `x = 6`.

`sizeof` Operator

`sizeof` is a unary operator used to calculate the size of data types. This operator can be applied to all data types. When using this operator, the keyword `sizeof` is followed by a type name, variable, or expression. The operator returns the size of the data type, variable, or expression in bytes. That is, the `sizeof` operator is used to determine the amount of memory space that the data type/variable/expression will take.

When a type name is used, it is enclosed in parentheses, but in case of variable names and expressions, they can be specified with or without parentheses. A `sizeof` expression returns an unsigned value that specifies the size of the space in bytes required by the data type, variable, or expression. For example, `sizeof(char)` returns 1, that is the size of a character data type. If we have,

```
int a = 10;
unsigned int result;
result = sizeof(a);
```

then `result = 2`, that is, space required to store the variable *a* in memory. Since `a` is an integer, it requires 2 bytes of storage space.

Operator Precedence Chart

Table 1.15 lists the operators that C language supports in the order of their *precedence* (highest to lowest). The *associativity* indicates the order in which the operators of equal precedence in an expression are evaluated.

Table 1.15 Operators precedence chart

Operator	Associativity
() [] . ->	left-to-right
++(postfix) --(postfix)	right-to-left
++(prefix) --(prefix) -(unary) - (unary) ! ~ (type) *(indirection) &(address) sizeof	right-to-left
* / %	left-to-right
+ -	left-to-right
<< >>	left-to-right
< <= > >=	left-to-right
== !=	left-to-right
&	left-to-right
^	left-to-right
|	left-to-right
&&	left-to-right
||	left-to-right
?:	right-to-left
= += -= *= /= %= &= ^= |= <<= >>=	right-to-left
,(comma)	left-to-right

Examples of Expressions Using the Precedence Chart

If we have the following variable declarations:

```
int a = 0, b = 1, c = -1;
float x = 2.5, y = 0.0;
```

then,

```
(a)  a && b = 0
(b)  a < b && c < b = 1
(c)  b + c || ! a
       = ( b + c) || (!a)
       = 0 ||1
       = 1
(d)  x * 5 && 5 || ( b / c)
       = ((x * 5) && 5) || (b / c)
       = (12.5 && 5) || (1/-1)
       = 1
(e)  a <= 10 && x >= 1 && b
       = ((a <= 10) && (x >= 1)) && b
       = (1 && 1) && 1
       = 1
(f)  !x || !c || b + c
       = ((!x) || (!c)) || (b + c)
       = (0 || 0) || 0
       = 0
(g)  x * y < a + b || c
       = ((x * y) < (a + b)) || c
       = (0 <1) || -1
       = 1
(h)  (x > y) + !a || c++
       = ((x > y) + (!a)) || (c++)
       = (1 + 1) || 0
       = 1
```

PROGRAMMING EXAMPLE

1. Write a program to calculate the area of a circle.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        float radius;
        double area;
        clrscr();
        printf("\n Enter the radius of the circle : ");
        scanf("%f", &radius);
        area = 3.14 * radius * radius;
        printf(" \n Area = %.2lf", area);
        return 0;
}
```

Output

```
Enter the radius of the circle : 7
Area = 153.86
```

1.8 TYPE CONVERSION AND TYPECASTING

Type conversion or typecasting of variables refers to changing a variable of one data type into another. While type conversion is done implicitly, casting has to be done explicitly by the programmer. We will discuss both these concepts here.

Type Conversion

Type conversion is done when the expression has variables of different data types. So to evaluate the expression, the data type is promoted from lower to higher level where the hierarchy of data types can be given as: `double`, `float`, `long`, `int`, `short`, and `char`. For example, type conversion is automatically done when we assign an integer value to a floating point variable. Consider the following code:

```
float x;
int y = 3;
x = y;
```

Now, `x = 3.0`, as integer value is automatically converted into its equivalent floating point representation.

Typecasting

Typecasting is also known as *forced conversion*. It is done when the value of one data type has to be converted into the value of another data type. The code to perform typecasting can be given as:

```
float salary = 10000.00;
int sal;
sal = (int) salary;
```

When floating point numbers are converted to integers, the digits after the decimal are truncated. Therefore, data is lost when floating point representations are converted to integral representations.

As we can see in the code, typecasting can be done by placing the destination data type in parentheses followed by the variable name that has to be converted. Hence, we conclude that typecasting is done to make a variable of one data type to act like a variable of another type.

PROGRAMMING EXAMPLE

2. Write a program to convert an integer into the corresponding floating point number.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        float f_num;
        int i_num;
        clrscr();
        printf("\n Enter any integer: ");
        scanf("%d", &i_num);
        f_num = (float)i_num;
        printf("\n The floating point variant of %d is = %f", i_num, f_num);
        return 0;
}
```

Output

```
Enter any integer: 56
The floating point variant of 56 is = 56.000000
```

1.9 CONTROL STATEMENTS

Till now we know that the code in the C program is executed sequentially from the first line of the program to its last line. That is, the second statement is executed after the first, the third statement is executed after the second, so on and so forth. Although this is true, in some cases we want only selected statements to be executed. Control flow statements enable programmers to conditionally execute a particular block of code. There are three types of control statements: decision control (branching), iterative (looping), and jump statements. While branching means deciding what actions have to be taken, looping, on the other hand, decides how many times the action has to be taken. Jump statements transfer control from one point to another point.

1.9.1 Decision Control Statements

C supports decision control statements that can alter the flow of a sequence of instructions. These statements help to jump from one part of the program to another depending on whether a particular condition is satisfied or not. These decision control statements include:

(a) `if` statement,
(b) `if-else` statement,
(c) `if-else-if` statement, and
(d) `switch-case` statement.

`if` *Statement*

`if` statement is the simplest decision control statement that is frequently used in decision making. The general form of a simple `if` statement is shown in Fig. 1.2.

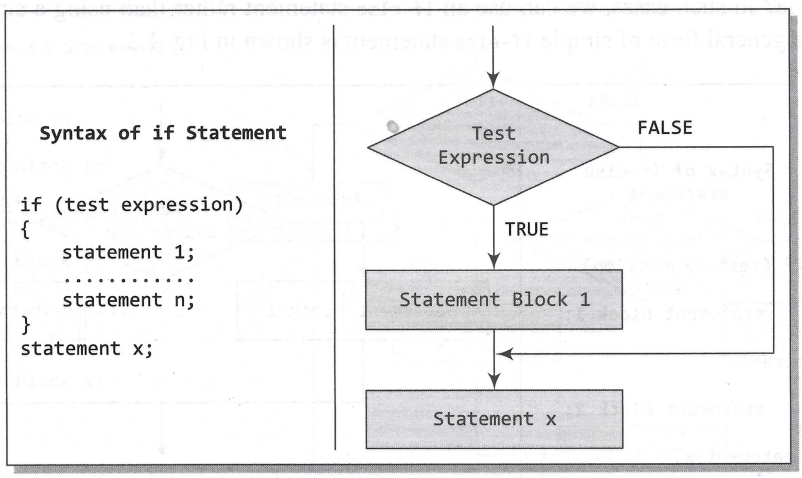

Figure 1.2 `if` statement construct

The `if` block may include 1 statement or *n* statements enclosed within curly brackets. First the test expression is evaluated. If the test expression is true, the statements of the `if` block are executed, otherwise these statements will be skipped and the execution will jump to statement `x`.

The statement in an `if` block is any valid C language statement, and the test expression is any valid C language expression that evaluates to either true or false. In addition to simple relational expressions, we can also use compound expressions formed using logical operators. Note that there is no semi-colon after the test expression. This is because the condition and statement should be put together as a single statement.

```
#include <stdio.h>
int main()
{
        int x=10;
        if (x>0) x++;
        printf("\n x = %d", x);
        return 0;
}
```

In the above code, we take a variable x and initialize it to 10. In the test expression, we check if the value of x is greater than 0. As 10 > 0, the test expression evaluates to true, and the value of x is incremented. After that, the value of x is printed on the screen. The output of this program is

```
x = 11
```

Observe that the `printf` statement will be executed even if the test expression is false.

Note In case the statement block contains only one statement, putting curly brackets becomes optional. If there are more than one statement in the statement block, putting curly brackets becomes mandatory.

`if-else` *Statement*

We have studied that using `if` statement plays a vital role in conditional branching. Its usage is very simple. The test expression is evaluated, if the result is true, the statement(s) followed by the expression is executed, else if the expression is false, the statement is skipped by the compiler.

What if you want a separate set of statements to be executed if the expression returns a false value? In such cases, we can use an `if-else` statement rather than using a simple `if` statement. The general form of simple `if-else` statement is shown in Fig. 1.3.

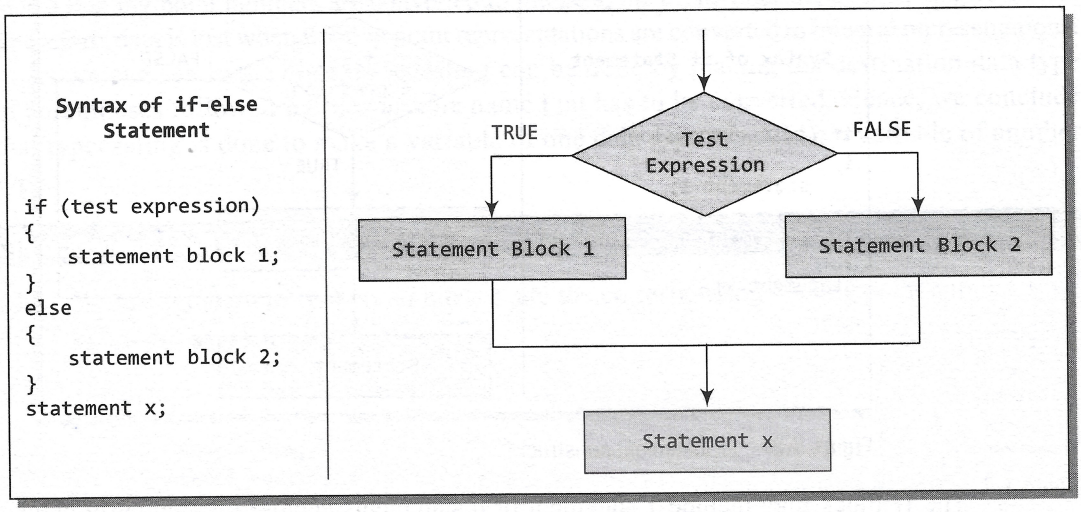

Figure 1.3 `if-else` statement construct

In the `if-else` construct, first the test expression is evaluated. If the expression is true, statement block 1 is executed and statement block 2 is skipped. Otherwise, if the expression is false, statement block 2 is executed and statement block 1 is ignored. In any case after the statement block 1 or 2 gets executed, the control will pass to statement x. Therefore, statement x is executed in every case.

PROGRAMMING EXAMPLE

3. Write a program to find whether a number is even or odd.

```
#include <stdio.h>
int main()
{
        int a;
        printf("\n Enter the value of a : ");
        scanf("%d", &a);
        if(a%2==0)
                printf("\n %d is even", a);
        else
                printf("\n %d is odd", a);
        return 0;
}
```

Output

```
Enter the value of a : 6
6 is even
```

if-else-if *Statement*

C language supports `if-else-if` statements to test additional conditions apart from the initial test expression. The `if-else-if` construct works in the same way as a normal `if` statement. Its construct is given in Fig. 1.4.

Syntax of if-else-if Statement

```
if (test expression 1)
{
      statement block 1;
}
else if (test expression 2)
{
      statement block 2;
}
.....
else
{
      statement block x;
}

statement y;
```

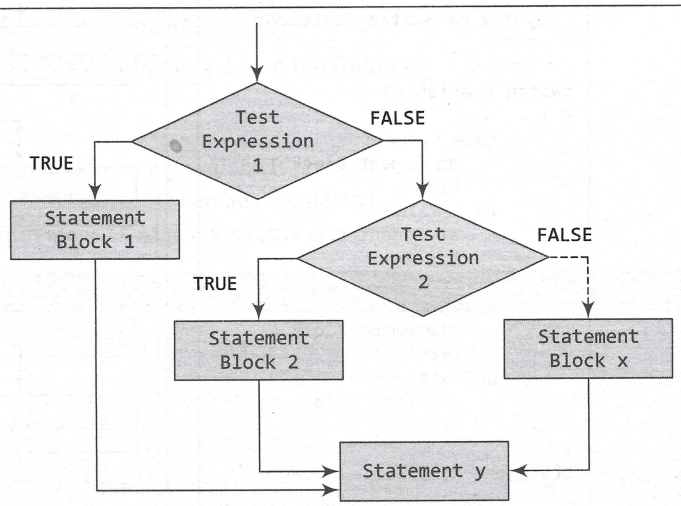

Figure 1.4 `if-else-if` statement construct

Note that it is not necessary that every `if` statement should have an `else` block as C supports simple `if` statements. After the first test expression or the first `if` branch, the programmer can have as many `else-if` branches as he wants depending on the expressions that have to be tested. For example, the following code tests whether a number entered by the user is negative, positive, or equal to zero.

```
#include <stdio.h>
int main()
{
```

```
        int num;
        printf("\n Enter any number : ");
        scanf("%d", &num);
        if(num==0)
                printf("\n The value is equal to zero");
        else if(num>0)
                printf("\n The number is positive");
        else
                printf("\n The number is negative");
        return 0;
}
```

Note that if the first test expression evaluates to a true value, i.e., `num=0`, then the rest of the statements in the code will be ignored and after executing the `printf` statement that displays 'The value is equal to zero', the control will jump to `return 0` statement.

`switch-case` *Statement*

A `switch-case` statement is a multi-way decision statement that is a simplified version of an `if-else-if` block. The general form of a `switch` statement is shown in Fig. 1.5.

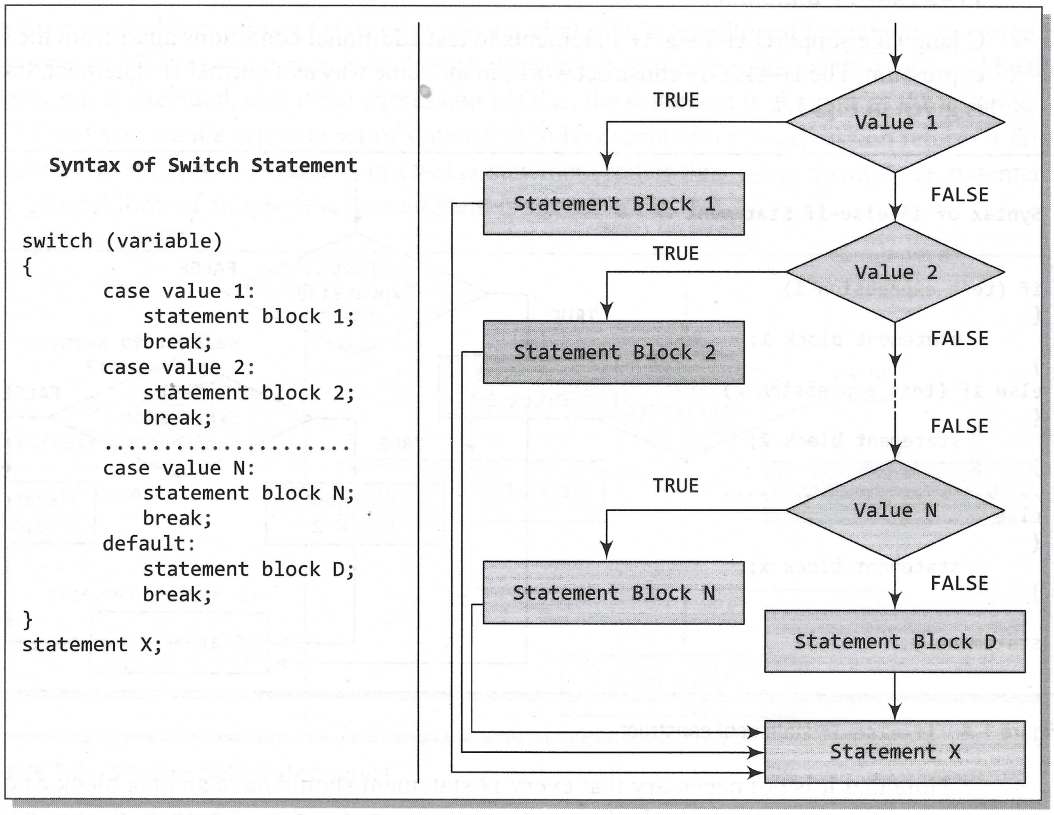

Figure 1.5 `switch-case` statement construct

The power of nested `if-else-if` statements lies in the fact that it can evaluate more than one expression in a single logical structure. `switch` statements are mostly used in two situations:

- When there is only one variable to evaluate in the expression
- When many conditions are being tested for

When there are many conditions to test, using the `if` and `else-if` constructs becomes complicated and confusing. Therefore, `switch case` statements are often used as an alternative to long `if` statements that compare a variable to several 'integral' values (integral values are those values that can be expressed as an integer, such as the value of a `char`). `Switch` statements are also used to handle the input given by the user.

We have already seen the syntax of the `switch` statement. The `switch case` statement compares the value of the variable given in the `switch` statement with the value of each case statement that follows. When the value of the `switch` and the `case` statement matches, the statement block of that particular case is executed.

Did you notice the keyword `default` in the syntax of the `switch case` statement? Default is the case that is executed when the value of the variable does not match with any of the values of the `case` statements. That is, default case is executed when no match is found between the values of `switch` and `case` statements and thus there are no statements to be executed. Although the default case is optional, it is always recommended to include it as it handles any unexpected case.

In the syntax of the `switch-case` statement, we have used another keyword `break`. The `break` statement must be used at the end of each case because if it is not used, then the case that matched and all the following cases will be executed. For example, if the value of `switch` statement matched with that of case 2, then all the statements in case 2 as well as the rest of the cases including default will be executed. The `break` statement tells the compiler to jump out of the switch case statement and execute the statement following the `switch-case` construct. Thus, the keyword `break` is used to break out of the case statements.

Advantages of Using a `switch-case` *Statement*

`Switch-case` statement is preferred by programmers due to the following reasons:

- Easy to debug
- Easy to read and understand
- Ease of maintenance as compared to its equivalent `if-else` statements
- Like `if-else` statements, `switch` statements can also be nested
- Executes faster than its equivalent `if-else` construct

PROGRAMMING EXAMPLE

4. Write a program to determine whether the entered character is a vowel or not.

```
#include <stdio.h>
int main()
{
        char ch;
        printf("\n Enter any character : ");
        scanf("%c", &ch);
        switch(ch)
        {
                case 'A':
                case 'a':
                        printf("\n %c is VOWEL", ch);
                        break;
                case 'E':
                case 'e':
                        printf("\n %c is VOWEL", ch);
                        break;
                case 'I':
                case 'i':
```

```
                printf("\n %c is VOWEL", ch);
                break;
        case 'O':
        case 'o':
                printf("\n %c is VOWEL", ch);
                break;
        case 'U':
        case 'u':
                printf("\n %c is VOWEL", ch);
                break;
        default: printf("\n %c is not a vowel", ch);
    }
    return 0;
}
```

Output

```
Enter any character : j
j is not a vowel
```

Note that there is no `break` statement after case `A`, so if the character `A` is entered then control will execute the statements given in case `a`.

1.9.2 Iterative Statements

Iterative statements are used to repeat the execution of a sequence of statements until the specified expression becomes false. C supports three types of iterative statements also known as looping statements. They are

- `while` loop
- `do-while` loop
- `for` loop

In this section, we will discuss all these statements.

`while` *Loop*

The `while` loop provides a mechanism to repeat one or more statements while a particular condition is true. Figure 1.6 shows the syntax and general form of a `while` loop.

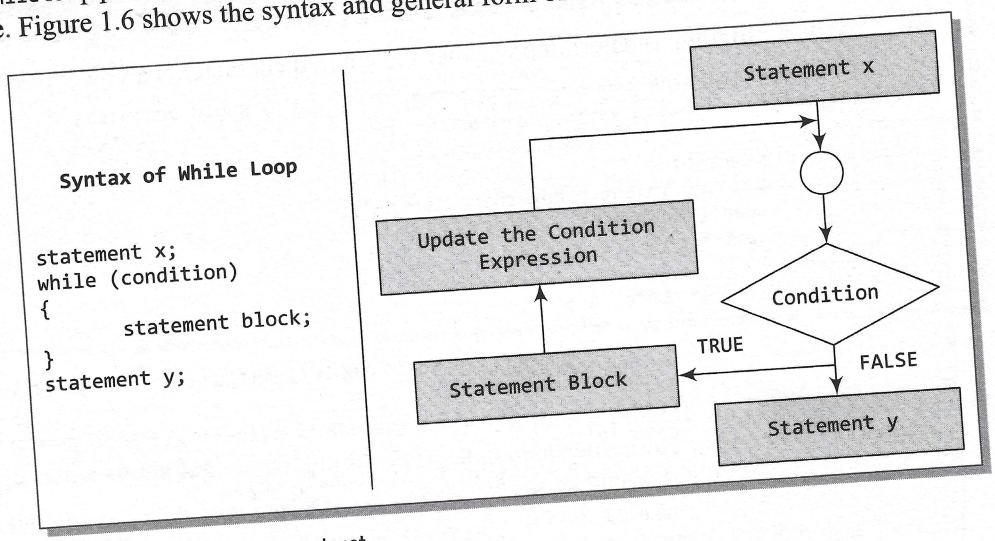

Figure 1.6 `While` loop construct

Note that in the `while` loop, the condition is tested before any of the statements in the statement block is executed. If the condition is true, only then the statements will be executed, otherwise if the condition is false, the control will jump to statement `y`, that is the immediate statement outside the `while` loop block.

In the flow diagram of Fig. 1.6, it is clear that we need to constantly update the condition of the `while` loop. It is this condition which determines when the loop will end. The `while` loop will execute as long as the condition is true. Note that if the condition is never updated and the condition never becomes false, then the computer will run into an infinite loop which is never desirable. For example, the following code prints the first 10 numbers using a `while` loop.

```
#include <stdio.h>
int main()
{
      int i = 1;
      while(i<=10)
      {
            printf("\n %d", i);
            i = i + 1;      // condition updated
      }
      return 0;
}
```

Note that initially `i = 1` and is less than 10, i.e., the condition is true, so in the `while` loop the value of `i` is printed and its value is incremented by 1. When `i=11`, the condition becomes false and the loop ends.

PROGRAMMING EXAMPLE

5. Write a program to calculate the sum of numbers from `m` to `n`.

```
#include <stdio.h>
int main()
{
        int n, m, i, sum =0;
        printf("\n Enter the value of m : ");
        scanf("%d", &m);
        i=m;
        printf("\n Enter the value of n : ");
        scanf("%d", &n);
        while(i<=n)
        {
                sum = sum + i;
                i = i + 1;
        }
        printf("\n The sum of numbers from %d to %d = %d", m, n, sum);
        return 0;
}
```

Output

```
Enter the value of m : 2
Enter the value of n : 10
The sum of numbers from 2 to 10  = 54
```

do-while Loop

The `do-while` loop is similar to the `while` loop. The only difference is that in a `do-while` loop, the test condition is tested at the end of the loop. As the test condition is evaluated at the end, this

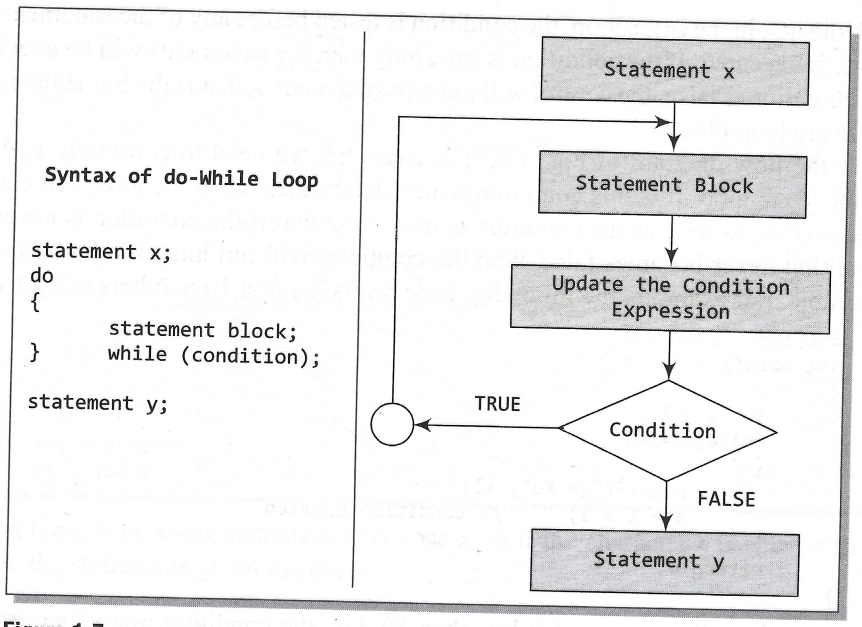

Figure 1.7 Do-while construct

Note that the test condition is enclosed in parentheses and followed by a semi-colon. The statements in the statement block are enclosed within curly brackets. The curly brackets are optional if there is only one statement in the body of the do-while loop.

The do-while loop continues to execute while the condition is true and when the condition becomes false, the control jumps to the statement following the do-while loop.

The major disadvantage of using a do-while loop is that it always executes at least once, so even if the user enters some invalid data, the loop will execute. However, do-while loops are widely used to print a list of options for menu-driven programs. For example, consider the following code.

```
#include <stdio.h>
int main()
{
      int i = 1;
      do
      {
            printf("\n %d", i);
            i = i + 1;
      } while(i<=10);
      return 0;
}
```

What do you think will be the output? Yes, the code will print numbers from 1 to 10.

Programming Example

6. Write a program to calculate the average of first *n* numbers.

```
#include <stdio.h>
int main()
{
      int n, i = 0, sum =0;
      float avg = 0.0;
```

```
        printf("\n Enter the value of n : ");
        scanf("%d", &n);
        do
        {
                sum = sum + i;
                i = i + 1;
        } while(i<=n);
        avg = (float)sum/n;
        printf("\n The sum of first %d numbers = %d",n, sum);
        printf("\n The average of first %d numbers = %.2f", n, avg);
        return 0;
}
```

Output

```
Enter the value of n : 20
The sum of first 20 numbers = 210
The average of first 20 numbers = 10.05
```

for *Loop*

Like the `while` and `do-while` loops, the `for` loop provides a mechanism to repeat a task till a particular condition is true. The synax and general form of a `for` loop is given in Fig. 1.8.

Syntax of for Loop

```
for (initialization; condition;
increment/decrement/update)
{
     statement block;
}
statement y;
```

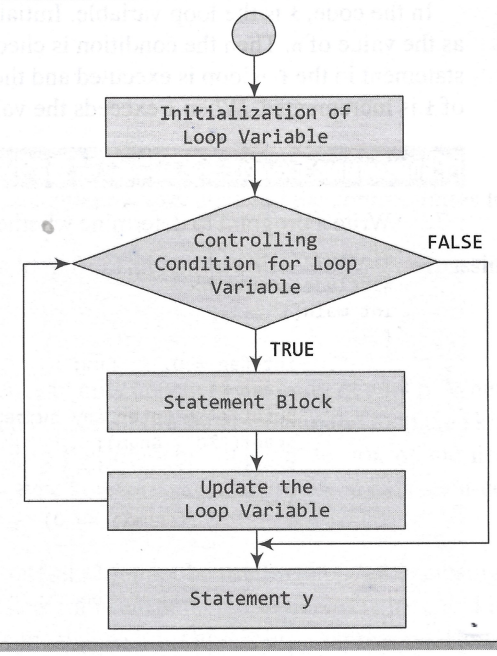

Figure 1.8 `for` loop construct

When a `for` loop is used, the loop variable is initialized only once. With every iteration, the value of the loop variable is updated and the condition is checked. If the condition is true, the statement block of the loop is executed, else the statements comprising the statement block of the `for` loop are skipped and the control jumps to the statement following the `for` loop body.

In the syntax of the `for` loop, initialization of the loop variable allows the programmer to give it a value. Second, the condition specifies that while the conditional expression is true, the loop

should continue to repeat itself. Every iteration of the loop must make the condition to exit the loop approachable. So, with every iteration, the loop variable must be updated. Updating the loop variable may include incrementing the loop variable, decrementing the loop variable or setting it to some other value like, `i +=2`, where `i` is the loop variable.

Note that every section of the `for` loop is separated from the other with a semi-colon. It is possible that one of the sections may be empty, though the semi-colons still have to be there. However, if the condition is empty, it is evaluated as true and the loop will repeat until something else stops it.

The `for` loop is widely used to execute a single or a group of statements for a limited number of times. The following code shows how to print the first *n* numbers using a `for` loop.

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

In the code, `i` is the loop variable. Initially, it is initialized with 1. Suppose the user enters 10 as the value of `n`. Then the condition is checked, since the condition is true as `i` is less than `n`, the statement in the `for` loop is executed and the value of `i` is printed. After every iteration, the value of `i` is incremented. When `i` exceeds the value of `n`, the control jumps to the `return 0` statement.

PROGRAMMING EXAMPLE

7. Write a program to determine whether a given number is a prime or a composite number.

```
#include <stdio.h>
#include <conio.h>
int main()
{
        int flag = 0, i, num;
        clrscr();
        printf("\n Enter any number : ");
        scanf("%d", &num);
        for(i=2; i<num/2;i++)
        {
                if(num%i == 0)
                {
                        flag =1;
                        break;
                }
        }
if(flag == 1)
        printf("\n %d is a composite number", num);
else
        printf("\n %d is a prime number", num);
return 0;
}
```

Output

```
Enter any number : 37
37 is a prime number
```