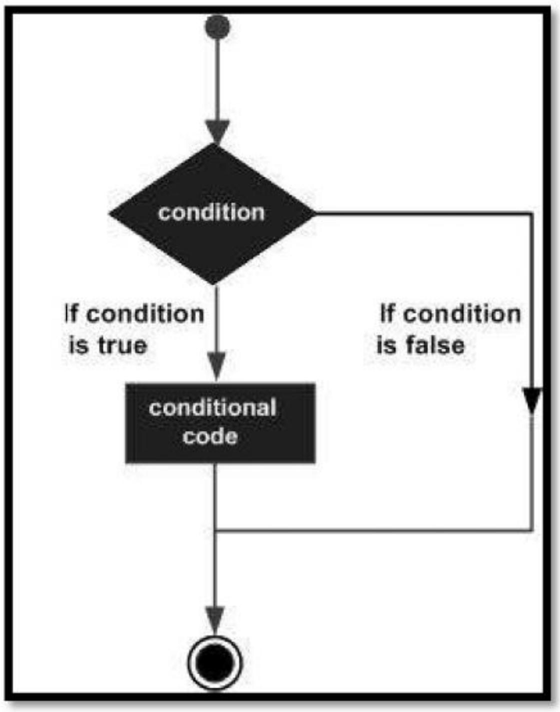
7. Control Structures

7.1 Decision making statements: if, nested if, if - else. Else if ladder

Decision making structures require that the programmer specify one or more conditions to be evaluated or tested by the program, along with a statement or statements to be executed if the condition is determined to be true, and optionally, other statements to be executed if the condition is determined to be false.

Following is the general form of a typical decision making structure found in most of the programming languages:



C++ programming language provides following types of decision making statements. Click the following links to check their detail.

Statement	Description
if statement	An if statement consists of a boolean expression followed by one or more statements.
if...else statement	An if statement can be followed by an optional else statement, which executes when the boolean expression is false.
switch statement	A switch statement allows a variable to be tested for equality against a list of values.
nested if statements	You can use one if or else if statement inside another if or else if statement(s).
nested switch statements	You can use one swicth statement inside another switch statement(s).

If Statement

An **if** statement consists of a boolean expression followed by one or more statements.

Syntax:

```
if(boolean_expression)
{
  // statement(s) will execute if the boolean expression is true
}
```

If the boolean expression evaluates to **true**, then the block of code inside the if statement will be executed. If boolean expression evaluates to **false**, then the first set of code after the end of the if statement (after the closing curly brace) will be executed.

Example:

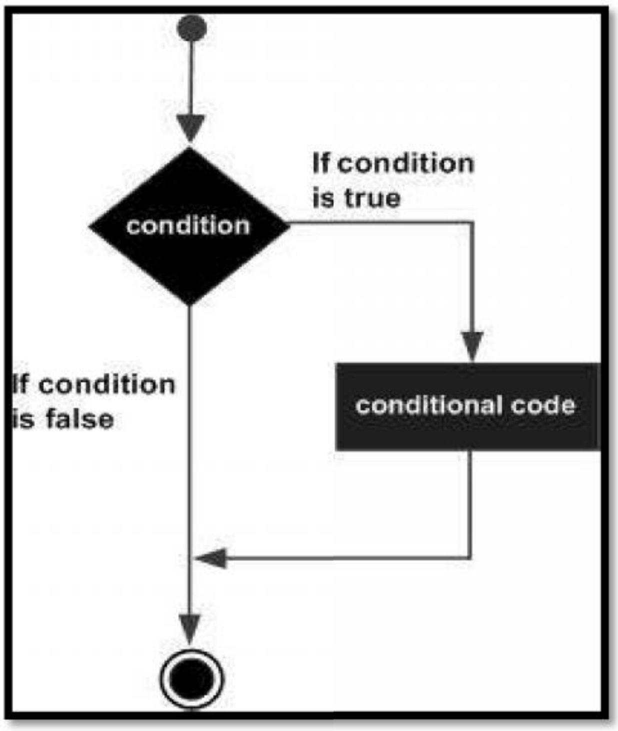
```
#include <iostream.h>

void main ()
{
  int a = 10;

  if( a < 20 )
  {
        cout << "a is less than 20;" << endl;
  }
  cout << "value of a is : " << a << endl;
  getch();
}
```

When the above code is compiled and executed, it produces the following result:

```
a is less than 20;
value of a is : 10
```



If else Statement

An **if** statement can be followed by an optional **else** statement, which executes when the boolean expression is false.

Syntax:

```
if(boolean_expression)
{
  // statement(s) will execute if the boolean expression is true
}
else
{
 // statement(s) will execute if the boolean expression is false
}
```

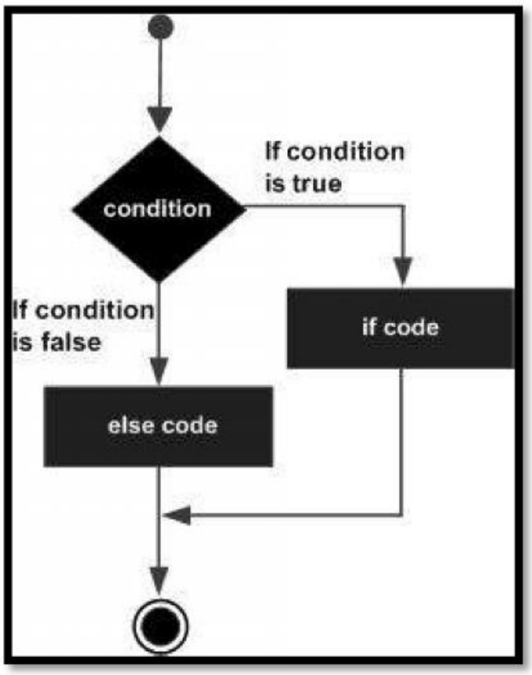
If the boolean expression evaluates to **true**, then the **if block** of code will be executed, otherwise **else block** of code will be executed.

Example:

```
#include <iostream.h>

void main ()
{
  int a = 100;

  if( a < 20 )
  {
     cout << "a is less than 20;" << endl;
  }
  else
  {
     cout << "a is not less than 20;" << endl;
  }
  cout << "value of a is : " << a << endl;
  getch();
}
```



When the above code is compiled and executed, it produces the following result:

```
a is not less than 20;
value of a is : 100
```

The if...else if...else Statement:

An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.

When using if , else if , else statements there are few points to keep in mind.

- An if can have zero or one else's and it must come after any else if's.
- An if can have zero to many else if's and they must come before the else.
- Once an else if succeeds, none of he remaining else if's or else's will be tested.

Syntax:

```
if(boolean_expression 1)
{
  // Executes when the boolean expression 1 is true
}
else if( boolean_expression 2)
{
  // Executes when the boolean expression 2 is true
}
else if( boolean_expression 3)
{
  // Executes when the boolean expression 3 is true
}
else
```

```
{
  // executes when the none of the above condition is true.
}
```

Example:

```
#include <iostream.h>

void main ()
{
  int a = 100;

  if( a == 10 )
  {
     cout << "Value of a is 10" << endl;
  }
  else if( a == 20 )
  {
     cout << "Value of a is 20" << endl;
  }
  else if( a == 30 )
  {
     cout << "Value of a is 30" << endl;
  }
  else
  {
     cout << "Value of a is not matching" << endl;
  }
  cout << "Exact value of a is : " << a << endl;
  getch();
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of a is not matching
Exact value of a is : 100
```

C++ Nested Loops

It is always legal to **nest** if-else statements, which means you can use one if or else if statement inside another if or else if statement(s).

Syntax:

```
if( boolean_expression 1)
{
  // Executes when the boolean expression 1 is true
  if(boolean_expression 2)
  {
    // Executes when the boolean expression 2 is true
  }
}
```

You can nest **else if...else** in the similar way as you have nested *if* statement.

Example:

```
#include <iostream.h>

void main ()
{
  int a = 100;
  int b = 200;

  if( a == 100 )
  {
     if( b == 200 )
     {
       cout << "Value of a is 100 and b is 200" << endl;
     }
  }

  cout << "Exact value of a is : " << a << endl;
  cout << "Exact value of b is : " << b << endl;
  getch();
}
```

When the above code is compiled and executed, it produces the following result:

```
Value of a is 100 and b is 200
Exact value of a is : 100
Exact value of b is : 200
```

7.2 Switch

A **switch** statement allows a variable to be tested for equality against a list of values. Each value is called a case, and the variable being switched on is checked for each case.

Syntax:

```
switch(expression){
   case constant-expression  :
     statement(s);
     break; //optional
   case constant-expression  :
     statement(s);
     break; //optional

   default : //Optional
     statement(s);
}
```

The following rules apply to a switch statement:

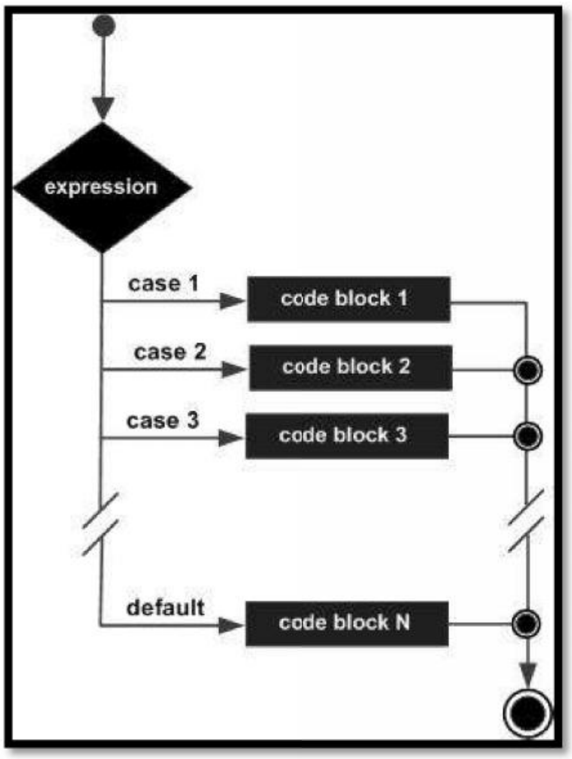
- The **expression** used in a **switch** statement must have an integral or enumerated type, or be of a class type in which the class has a single conversion function to an integral or enumerated type.
- You can have any number of case statements within a switch. Each case is followed by the value to be compared to and a colon.
- The **constant-expression** for a case must be the same data type as the variable in the switch, and it must be a constant or a literal.
- When the variable being switched on is equal to a case, the statements following that case will execute until a **break** statement is reached.
- When a break statement is reached, the switch terminates, and the flow of control jumps to the next line following the switch statement.
- Not every case needs to contain a break. If no break appears, the flow of control will *fall through* to subsequent cases until a break is reached.
- A **switch** statement can have an optional **default** case, which must appear at the end of the switch. The default case can be used for performing a task when none of the cases is true. No break is needed in the default case.

Example:

```
#include <iostream.h>

void main ()
{
  char grade = 'D';

  switch(grade)
  {
  case 'A' :
    cout << "Excellent!" << endl;
    break;
  case 'B' :
  case 'C' :
    cout << "Well done" << endl;
    break;
  case 'D' :
    cout << "You passed" << endl;
    break;
  case 'F' :
    cout << "Better try again" << endl;
    break;
  default :
    cout << "Invalid grade" << endl;
  }
  cout << "Your grade is " << grade << endl;
  getch();
}
```



This would produce the following result:

```
You passed
Your grade is D
```

Nested Switch

It is possible to have a switch as part of the statement sequence of an outer switch. Even if the case constants of the inner and outer switch contain common values, no conflicts will arise.

C++ specifies that at least 256 levels of nesting be allowed for switch statements.

Syntax:

```
switch(ch1) {
  case 'A':
    cout << "This A is part of outer switch";
    switch(ch2) {
      case 'A':
        cout << "This A is part of inner switch";
        break;
      case 'B': // ...
    }
    break;
  case 'B': // ...
}
```

Example:

```
#include <iostream.h>
void main ()
{
  int a = 100, b = 200;

  switch(a) {
    case 100:
      cout << "This is part of outer switch" << endl;
      switch(b) {
        case 200:
          cout << "This is part of inner switch" << endl;
      }
  }
  cout << "Exact value of a is : " << a << endl;
  cout << "Exact value of b is : " << b << endl;
  getch();
}
```

This would produce the following result:

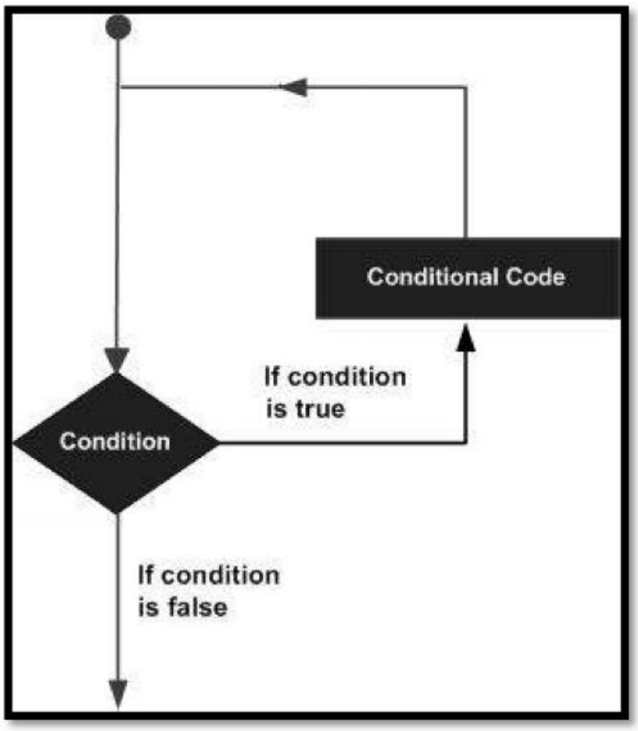
```
This is part of outer switch
This is part of inner switch
Exact value of a is : 100
Exact value of b is : 200
```

7.3 Loops and iteration: while loop, for loop, do - while loop, nesting of loops, Break statement, Continue statement, Goto statement, Use of control structures through illustrative programming examples.

There may be a situation, when you need to execute a block of code several number of times. In general statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times and following is the general from of a loop statement in most of the programming languages:



C++ programming language provides the following types of loop to handle looping requirements. Click the following links to check their detail.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is true. It tests the condition before executing the loop body.
for loop	Execute a sequence of statements multiple times and abbreviates the code that manages the loop variable.
do...while loop	Like a while statement, except that it tests the condition at the end of the loop body
nested loops	You can use one or more loop inside any another while, for or do..while loop.

While Loop

A **while** loop statement repeatedly executes a target statement as long as a given condition is true.

Syntax:

The syntax of a while loop in C++ is:

```
while(condition)
{
  statement(s);
}
```

Here, **statement(s)** may be a single statement or a block of statements.

The**condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

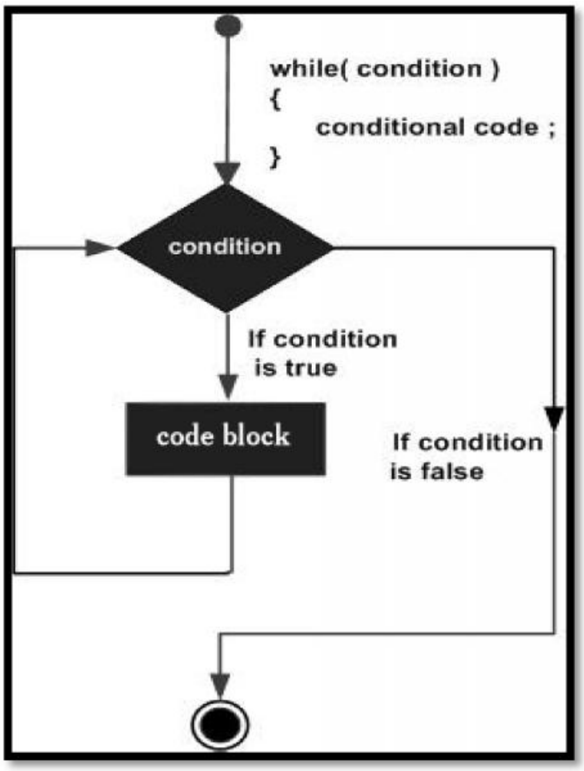
Flow Diagram:

Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

Example:

```
#include <iostream.h>
void main ()
{
  int a = 10;

  while( a < 20 )
  {
     cout << "value of a: " << a << endl;
     a++;
  }
  getch();
}
```



When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

For Loop

A **for** loop is a repetition control structure that allows you to efficiently write a loop that needs to execute a specific number of times.

Syntax:

```
for ( init; condition; increment )
{
  statement(s);
}
```

Here is the flow of control in a for loop:

- The **init** step is executed first, and only once. This step allows you to declare and initialize any loop control variables. You are not required to put a statement here, as long as a semicolon appears.
- Next, the **condition** is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.
- After the body of the for loop executes, the flow of control jumps back up to the **increment** statement. This statement allows you to update any loop control variables. This statement can be left blank, as long as a semicolon appears after the condition.
- The condition is now evaluated again. If it is true, the loop executes and the process repeats itself (body of loop, then increment step, and then again condition). After the condition becomes false, the for loop terminates.

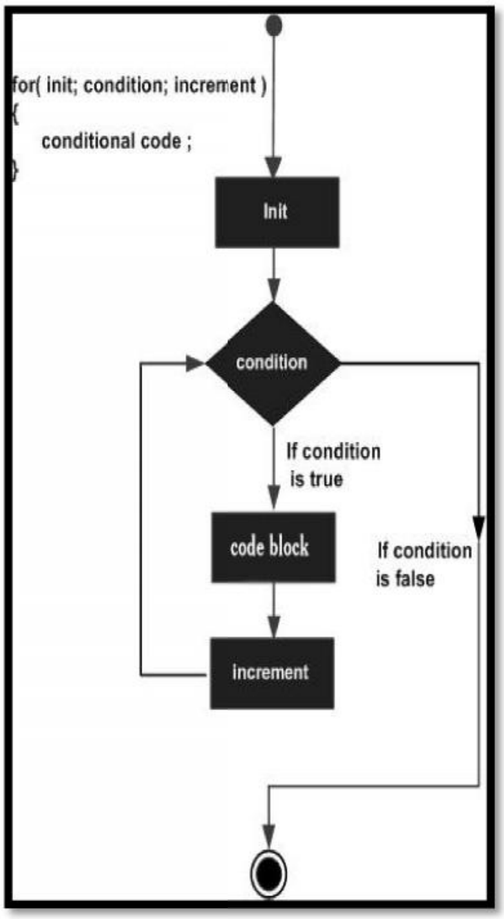
Example:

```
#include <iostream.h>

void main ()
{
   for( int a = 10; a < 20; a = a + 1 )
   {
       cout << "value of a: " << a << endl;
   }
   getch();
}
```

When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```



Do-While Loop

Unlike **for** and **while** loops, which test the loop condition at the top of the loop, the **do...while** loop checks its condition at the bottom of the loop.

A **do...while** loop is similar to a while loop, except that a do...while loop is guaranteed to execute at least one time.

Syntax:

```
do
{
  statement(s);
}while( condition );
```

Notice that the conditional expression appears at the end of the loop, so the statement(s) in the loop execute once before the condition is tested.

If the condition is true, the flow of control jumps back up to do, and the statement(s) in the loop execute again. This process repeats until the given condition becomes false.

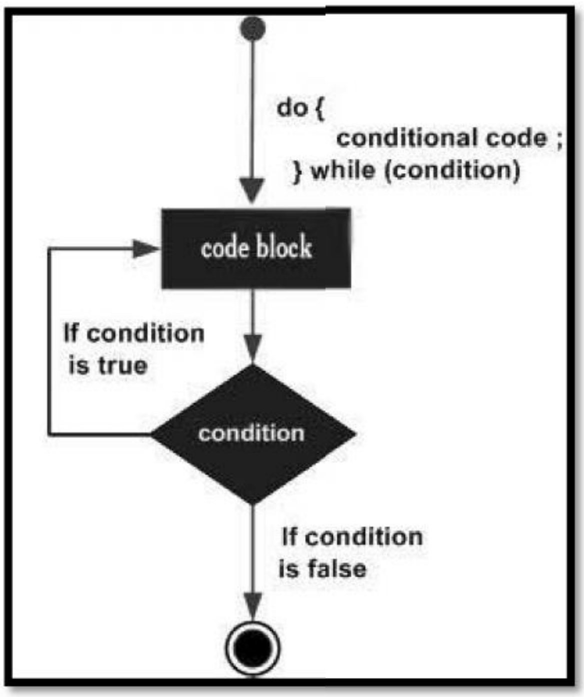
Example:

```
#include <iostream.h>

void main ()
{
  int a = 10;

  do
  {
     cout << "value of a: " << a << endl;
     a = a + 1;
  }while( a < 20 );

  getch();
}
```



When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Nested Loop

A loop can be nested inside of another loop. C++ allows at least 256 levels of nesting.

Syntax:

```
for ( init; condition; increment )
{
  for ( init; condition; increment )
  {
    statement(s);
  }
  statement(s);
}
```

The syntax for a **nested while loop** statement in C++ is as follows:

```
while(condition)
{
  while(condition)
  {
    statement(s);
  }
  statement(s);
}
```

The syntax for a **nested do...while loop** statement in C++ is as follows:

```
do
{
  statement(s);
  do
  {
    statement(s);
  }while( condition );

}while( condition );
```

Example:

The following program uses a nested for loop to find the prime numbers from 2 to 100:

```
#include <iostream.h>

void main ()
{
  int i, j;

  for(i=2; i<100; i++) {
    for(j=2; j <= (i/j); j++)
      if(!(i%j)) break;
      if(j > (i/j)) cout << i << " is prime\n";
  }
  getch();
}
```

This would produce the following result:

```
2 is prime
3 is prime
5 is prime
7 is prime
11 is prime
13 is prime
17 is prime
19 is prime
23 is prime
29 is prime
31 is prime
37 is prime
41 is prime
43 is prime
47 is prime
53 is prime
59 is prime
61 is prime
67 is prime
71 is prime
73 is prime
79 is prime
83 is prime
89 is prime
97 is prime
```

Loop Control Statements:

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

C++ supports the following control statements. Click the following links to check their detail.

Control Statement	Description
break statement	Terminates the **loop** or **switch** statement and transfers execution to the statement immediately following the loop or switch.
continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
goto statement	Transfers control to the labeled statement. Though it is not advised to use goto statement in your program.

The Infinite Loop:

A loop becomes infinite loop if a condition never becomes false. The **for** loop is traditionally used for this purpose. Since none of the three expressions that form the for loop are required, you can make an endless loop by leaving the conditional expression empty.

```
#include <iostream.h>

void main ()
{
  for( ; ; )
  {
    printf("This loop will run forever.\n");
  }
  getch();
}
```

When the conditional expression is absent, it is assumed to be true. You may have an initialization and increment expression, but C++ programmers more commonly use the for(;;) construct to signify an infinite loop.

NOTE: You can terminate an infinite loop by pressing Ctrl + C keys.

Break Statement

The **break** statement has the following two usages in C++:

- When the **break** statement is encountered inside a loop, the loop is immediately terminated and program control resumes at the next statement following the loop.
- It can be used to terminate a case in the **switch** statement

If you are using nested loops (i.e., one loop inside another loop), the break statement will stop the execution of the innermost loop and start executing the next line of code after the block.

The syntax of a break statement in C++ is:

```
break;
```

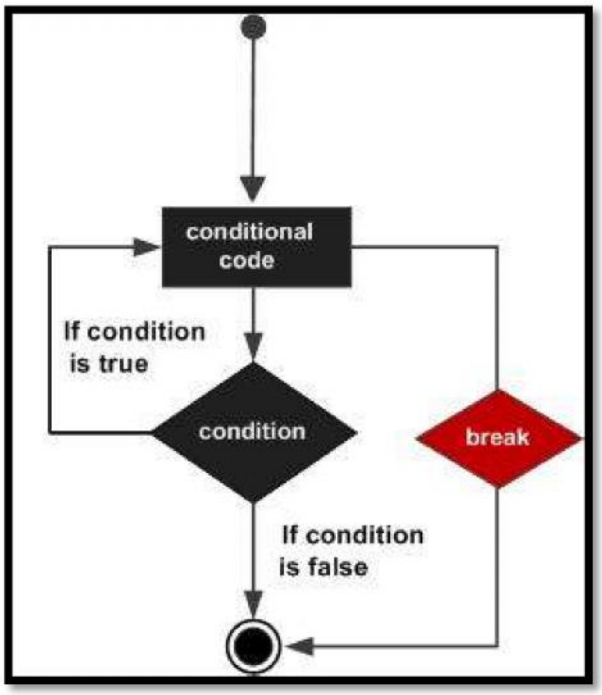
Example:

```
#include <iostream.h>

void main ()
{
  int a = 10;

  do
  {
    cout << "value of a: " << a << endl;
    a = a + 1;
    if( a > 15)
    {
      break;
    }
  }while( a < 20 );

  getch();
}
```



When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

Continue Statement

The **continue** statement works somewhat like the break statement. Instead of forcing termination, however, continue forces the next iteration of the loop to take place, skipping any code in between.

For the **for** loop, continue causes the conditional test and increment portions of the loop to execute. For the **while** and **do...while** loops, program control passes to the conditional tests.

Syntax:

```
continue;
```

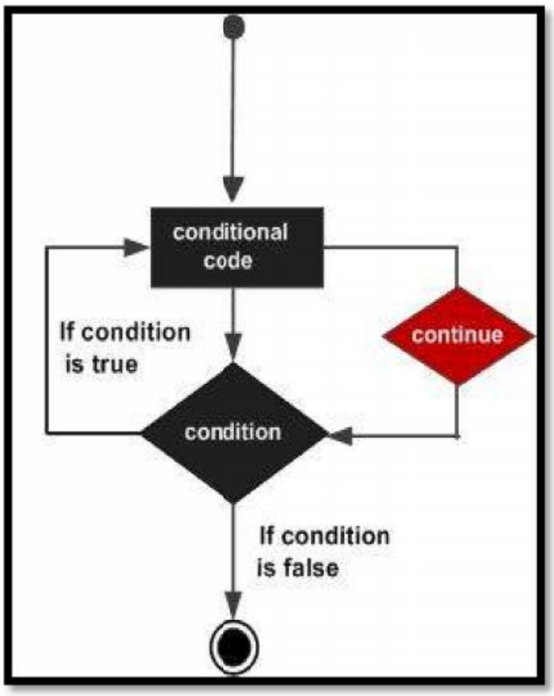
Example:

```
#include <iostream.h>

void main ()
{
   int a = 10;

   do
   {
       if( a == 15)
       {
          a = a + 1;
          continue;
       }
       cout << "value of a: " << a << endl;
       a = a + 1;
   }while( a < 20 );

   getch();
}
```



When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```

Goto Statement

A **goto** statement provides an unconditional jump from the goto to a labeled statement in the same function.

NOTE: Use of **goto** statement is highly discouraged because it makes difficult to trace the control flow of a program, making the program hard to understand and hard to modify. Any program that uses a goto can be rewritten so that it doesn't need the goto.

Syntax:

```
goto label;
..
.
label: statement;
```

Where **label** is an identifier that identifies a labeled statement. A labeled statement is any statement that is preceded by an identifier followed by a colon (:).

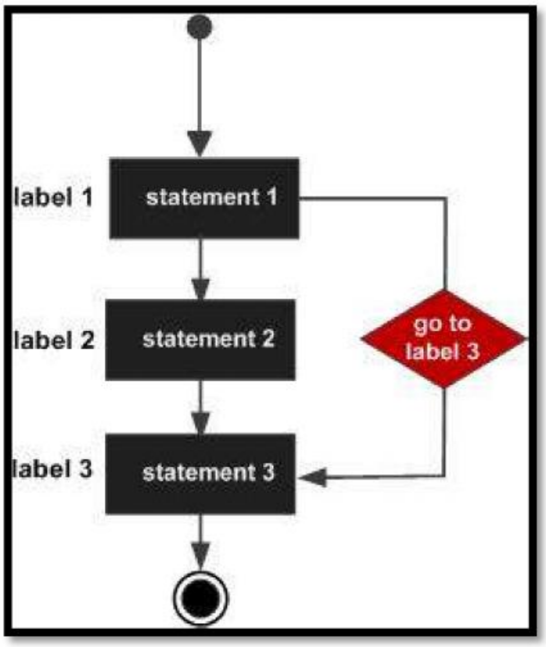
Example:

```
#include <iostream.h>

void main ()
{
  int a = 10;

  LOOP:do
  {
     if( a == 15)
     {
        // skip the iteration.
        a = a + 1;
        goto LOOP;
     }
     cout << "value of a: " << a << endl;
     a = a + 1;
  }while( a < 20 );

  getch();
}
```



When the above code is compiled and executed, it produces the following result:

```
value of a: 10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 16
value of a: 17
value of a: 18
value of a: 19
```