9. MODIFIER TYPES

C++ allows the char, int, and double data types to have modifiers preceding them. A modifier is used to alter the meaning of the base type so that it more precisely fits the needs of various situations.
The data type modifiers are listed here:

- ☐ signed
- ☐ unsigned
- ☐ long
- ☐ short

The modifiers signed, unsigned, long, and short can be applied to integer base types. In addition, signed and unsigned can be applied to char, and long can be applied to double.
The modifiers signed and unsigned can also be used as prefix to long or short modifiers. For example, unsigned long int.
C++ allows a shorthand notation for declaring unsigned, short, or long integers. You can simply use the word unsigned, short, or long, without int. It automatically implies int. For example, the following two statements both declare unsigned integer variables.

```
unsigned x;
unsigned int y;
```

To understand the difference between the way signed and unsigned integer modifiers are interpreted by C++, you should run the following short program:

```
#include <iostream>
using namespace std;

/* This program shows the difference between
* signed and unsigned integers.
*/
int main()
{
short int i; // a signed short integer
short unsigned int j; // an unsigned short integer
j = 50000;
i = j;
cout << i << " " << j;
return 0;
}
```

When this program is run, following is the output:

```
-15536 50000
```

The above result is because the bit pattern that represents 50,000 as a short unsigned integer is interpreted as -15,536 by a short.

Type Qualifiers in C++

The type qualifiers provide additional information about the variables they precede.

Qualifier	Meaning
const	Objects of type const cannot be changed by your program during execution
volatile	The modifier volatile tells the compiler that a variable's value may be changed in ways not explicitly specified by the program.
restrict	A pointer qualified by restrict is initially the only means by which the object it points to can be accessed. Only C99 adds a new type qualifier called restrict.

10. STORAGE CLASSES

A storage class defines the scope (visibility) and life-time of variables and/or functions within a C++ Program. These specifiers precede the type that they modify. There are following storage classes, which can be used in a C++ Program

☐ auto

☐ register

☐ static

☐ extern

☐ mutable

The auto Storage Class

The `auto` storage class is the default storage class for all local variables.

```
{
int mount;
auto int month;
}
```

The example above defines two variables with the same storage class, auto can only be used within functions, i.e., local variables.

The register Storage Class

The `register` storage class is used to define local variables that should be stored in a register instead of RAM. This means that the variable has a maximum size equal to the register size (usually one word) and can't have the unary '&' operator applied to it (as it does not have a memory location).

```
{
register int miles;
}
```

The register should only be used for variables that require quick access such as counters. It should also be noted that defining 'register' does not mean that the variable will be stored in a register. It means that it MIGHT be stored in a register depending on hardware and implementation restrictions.

10. STORAGE CLASSES C++

The static Storage Class

The `static` storage class instructs the compiler to keep a local variable in existence during the life-time of the program instead of creating and destroying it each time it comes into and goes out of scope. Therefore, making local variables static allows them to maintain their values between function calls.
The static modifier may also be applied to global variables. When this is done, it causes that variable's scope to be restricted to the file in which it is declared.
In C++, when static is used on a class data member, it causes only one copy of that member to be shared by all objects of its class.

```
#include <iostream>
// Function declaration
void func(void);
static int count = 10; /* Global variable */
main()
{
while(count--)
{
func();
}
return 0;
}
// Function definition
void func( void )
{
static int i = 5; // local static variable
i++;
std::cout << "i is " << i ;
std::cout << " and count is " << count << std::endl;
}
```

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

```
i is 6 and count is 9 C++
i is 7 and count is 8
i is 8 and count is 7
i is 9 and count is 6
i is 10 and count is 5
i is 11 and count is 4
i is 12 and count is 3
i is 13 and count is 2
i is 14 and count is 1

i is 15 and count is 0
```

The extern Storage Class

The `extern` storage class is used to give a reference of a global variable that is visible to ALL the program files. When you use 'extern' the variable cannot be initialized as all it does is point the variable name at a storage location that has been previously defined.

When you have multiple files and you define a global variable or function, which will be used in other files also, then *extern* will be used in another file to give reference of defined variable or function. Just for understanding *extern* is used to declare a global variable or function in another file.

The extern modifier is most commonly used when there are two or more files sharing the same global variables or functions as explained below.

First File: main.cpp

```
#include <iostream>
int count ;
extern void write_extern();
main()
{
count = 5;
write_extern();
}
```

Second File: support.cpp

```
#include <iostream>
```

```
extern int count;
void write_extern(void)
{
std::cout << "Count is " << count << std::endl;
}
```

Here, *extern* keyword is being used to declare count in another file. Now compile these two files as follows:

```
$g++ main.cpp support.cpp -o write
```

This will produce `write` executable program, try to execute `write` and check the result as follows:

```
$./write
5
```

The mutable Storage Class

The `mutable` specifier applies only to class objects, which are discussed later in this tutorial. It allows a member of an object to override const member function. That is, a mutable member can be modified by a const member function. C++

11. OPERATORS

An operator is a symbol that tells the compiler to perform specific mathematical or logical manipulations. C++ is rich in built-in operators and provide the following types of operators:

☐ Arithmetic Operators

☐ Relational Operators

☐ Logical Operators

☐ Bitwise Operators

☐ Assignment Operators

☐ Misc Operators

This chapter will examine the arithmetic, relational, logical, bitwise, assignment and other operators one by one.

Arithmetic Operators

There are following arithmetic operators supported by C++ language:

Assume variable A holds 10 and variable B holds 20, then: Operator	Description	Example
+	Adds two operands	A + B will give 30
-	Subtracts second operand from the first	A - B will give -10
*	Multiplies both operands	A * B will give 200
/	Divides numerator by de-numerator	B / A will give 2
%	Modulus Operator and remainder of after an integer division	B % A will give 0
++	Increment operator, increases integer value by one	A++ will give 11
--	Decrement operator, decreases integer value by one	A-- will give 9

Try the following example to understand all the arithmetic operators available in C++.
Copy and paste the following C++ program in test.cpp file and compile and run this program.

```
#include <iostream>
using namespace std;
main()
{
int a = 21;
int b = 10;
int c ;
c = a + b;
cout << "Line 1 - Value of c is :" << c << endl ;
c = a - b;
cout << "Line 2 - Value of c is :" << c << endl ;
c = a * b;
cout << "Line 3 - Value of c is :" << c << endl ;
c = a / b;
cout << "Line 4 - Value of c is :" << c << endl ;
c = a % b;
cout << "Line 5 - Value of c is :" << c << endl ;
c = a++;
cout << "Line 6 - Value of c is :" << c << endl ;
c = a--;
cout << "Line 7 - Value of c is :" << c << endl ;
return 0; C++
```

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

```
Line 1 - Value of c is :31
Line 2 - Value of c is :11
Line 3 - Value of c is :210
Line 4 - Value of c is :2
Line 5 - Value of c is :1
Line 6 - Value of c is :21
Line 7 - Value of c is :22
```

Relational Operators

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then: Operator	Description	Example
==	Checks if the values of two operands are equal or not, if yes then condition becomes true.	(A == B) is not true.
!=	Checks if the values of two operands are equal or not, if values are not equal then condition becomes true.	(A != B) is true.
>	Checks if the value of left operand is greater than the value of right operand, if yes then condition becomes true.	(A > B) is not true.
<	Checks if the value of left operand is less than the value of right operand, if yes then condition becomes true.	(A < B) is true.
>=	Checks if the value of left operand is greater than or equal to the value of right operand, if yes then condition becomes true.	(A >= B) is not true.
<=	Checks if the value of left operand is less than or equal to the value of right operand, if yes then	(A <= B) is true.

	condition becomes true.	

Try the following example to understand all the relational operators available in C++.
Copy and paste the following C++ program in test.cpp file and compile and run this program.

```
#include <iostream>
using namespace std;
main()
{
int a = 21;
int b = 10;
int c ;
if( a == b )
{
cout << "Line 1 - a is equal to b" << endl ;
}
else
{
cout << "Line 1 - a is not equal to b" << endl ;
}
if ( a < b )
{
```

```
cout << "Line 2 - a is less than b" << endl ;
}
else
{
cout << "Line 2 - a is not less than b" << endl ;
}
if ( a > b )
{
cout << "Line 3 - a is greater than b" << endl ;
}
else
{
cout << "Line 3 - a is not greater than b" << endl ;
}
/* Let's change the values of a and b */
a = 5;
b = 20;
if ( a <= b )
{
cout << "Line 4 - a is either less than \
or equal to b" << endl ;
}
if ( b >= a )
{
cout << "Line 5 - b is either greater than \
or equal to b" << endl ;
}
return 0;
}
```

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

```
Line 1 - a is not equal to b
Line 2 - a is not less than b
Line 3 - a is greater than b C++
Line 4 - a is either less than or equal to b
Line 5 - b is either greater than or equal to b
```

Logical Operators

There are following logical operators supported by C++ language.

Assume variable A holds 1 and variable B holds 0, then: Operator	Description	Example
&&	Called Logical AND operator. If both the operands are non-zero, then condition becomes true.	(A && B) is false.
||	Called Logical OR Operator. If any of the two operands is non-zero, then condition becomes true.	(A || B) is true.
!	Called Logical NOT Operator. Use to reverses the logical state of its operand. If a condition is true, then Logical NOT operator will make false.	!(A && B) is true.

C++.
Copy and paste the following C++ program in test.cpp file and compile and run this program.

```
#include <iostream>
using namespace std;
main()
{
int a = 5;
int b = 20; C++
int c ;
if ( a && b )
{
cout << "Line 1 - Condition is true"<< endl ;
}
if ( a || b )
{
cout << "Line 2 - Condition is true"<< endl ;
}
/* Let's change the values of a and b */
a = 0;
b = 10;
if ( a && b )
{
cout << "Line 3 - Condition is true"<< endl ;
}
else
{
cout << "Line 4 - Condition is not true"<< endl ;
}
if ( !(a && b) )
{
cout << "Line 5 - Condition is true"<< endl ;
}
return 0;
}
```

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

```
Line 1 - Condition is true
Line 2 - Condition is true
Line 4 - Condition is not true
Line 5 - Condition is true
```

Bitwise Operators

Bitwise operator works on bits and perform bit-by-bit operation. The truth tables for &, |, and ^ are as follows:

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

Assume if A = 60; and B = 13; now in binary format they will be as follows:
A = 0011 1100
B = 0000 1101

A&B = 0000 1100
A|B = 0011 1101
A^B = 0011 0001
~A = 1100 0011

The Bitwise operators supported by C++ language are listed in the following table. Assume variable A holds 60 and variable B holds 13, then:

Operator	Description	Example
&	Binary AND Operator copies a bit to the result if it exists in both operands.	(A & B) will give 12 which is 0000 1100
|	Binary OR Operator copies a bit if it exists in either operand.	(A | B) will give 61 which is 0011 1101
^	Binary XOR Operator copies the bit if it is set in one operand but not both.	(A ^ B) will give 49 which is 0011 0001
~	Binary Ones Complement Operator is unary and has the effect of 'flipping' bits.	(~A) will give -61 which is 1100 0011 in 2's complement form due to a signed binary number.
<<	Binary Left Shift Operator. The left operands value is moved left by the number of bits specified by the right operand.	A << 2 will give 240 which is 1111 0000
>>	Binary Right Shift Operator. The left operands value is moved right by the number of bits specified by the right operand.	A >> 2 will give 15 which is 0000 1111

Try the following example to understand all the bitwise operators available in C++.
Copy and paste the following C++ program in test.cpp file and compile and run this program.

```
#include <iostream>
using namespace std;
main()
{
unsigned int a = 60; // 60 = 0011 1100
unsigned int b = 13; // 13 = 0000 1101
int c = 0;
c = a & b; // 12 = 0000 1100
cout << "Line 1 - Value of c is : " << c << endl ;
c = a | b; // 61 = 0011 1101
cout << "Line 2 - Value of c is: " << c << endl ;

c = a ^ b; // 49 = 0011 0001
cout << "Line 3 - Value of c is: " << c << endl ;
c = ~a; // -61 = 1100 0011
cout << "Line 4 - Value of c is: " << c << endl ;
c = a << 2; // 240 = 1111 0000
cout << "Line 5 - Value of c is: " << c << endl ;
c = a >> 2; // 15 = 0000 1111
cout << "Line 6 - Value of c is: " << c << endl ;
return 0;
}
```

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

```
Line 1 - Value of c is : 12
Line 2 - Value of c is: 61
Line 3 - Value of c is: 49
Line 4 - Value of c is: -61
Line 5 - Value of c is: 240
Line 6 - Value of c is: 15
```

Assignment Operators

There are following assignment operators supported by C++ language:

Operator	Description	Example
=	Simple assignment operator, Assigns values from right side operands to left side operand.	C = A + B will assign value of A + B into C
+=	Add AND assignment operator, It adds right operand to the left operand and assign the result to left operand.	C += A is equivalent to C = C +
-=	Subtract AND assignment operator, It subtracts right operand from the left operand and assign the result to left operand.	C -= A is equivalent to C = C - A
*=	Multiply AND assignment operator, It multiplies right operand with the left operand and assign the result to left operand.	C *= A is equivalent to C = C * A
/=	Divide AND assignment operator, It divides left operand with the right operand and assign the result to left operand.	C /= A is equivalent to C = C / A
%=	Modulus AND assignment operator, It takes modulus using two operands and	C %= A is equivalent to C = C % A

	assign the result to left operand.	
<<=	Left shift AND assignment operator.	C <<= 2 is same as C = C << 2
>>=	Right shift AND assignment operator.	C >>= 2 is same as C = C >> 2
&=	Bitwise AND assignment operator.	C &= 2 is same as C = C & 2
^=	Bitwise exclusive OR and assignment operator.	C ^= 2 is same as C = C ^ 2
|=	Bitwise inclusive OR and assignment operator.	C |= 2 is same as C = C | 2

in C++.
Copy and paste the following C++ program in test.cpp file and compile and run this program.

```
#include <iostream>
using namespace std;
main()
{
int a = 21;
int c ;
c = a;
cout << "Line 1 - = Operator, Value of c = : " <<c<< endl ;
c += a;
cout << "Line 2 - += Operator, Value of c = : " <<c<< endl ;
c -= a;
cout << "Line 3 - -= Operator, Value of c = : " <<c<< endl ;
c *= a;
cout << "Line 4 - *= Operator, Value of c = : " <<c<< endl ;
c /= a;
cout << "Line 5 - /= Operator, Value of c = : " <<c<< endl ;
c = 200;
c %= a;
cout << "Line 6 - %= Operator, Value of c = : " <<c<< endl ;
c <<= 2;
cout << "Line 7 - <<= Operator, Value of c = : " <<c<< endl ; C++
c >>= 2;
cout << "Line 8 - >>= Operator, Value of c = : " <<c<< endl ;
c &= 2;
cout << "Line 9 - &= Operator, Value of c = : " <<c<< endl ;
c ^= 2;
cout << "Line 10 - ^= Operator, Value of c = : " <<c<< endl ;
c |= 2;
cout << "Line 11 - |= Operator, Value of c = : " <<c<< endl ;
return 0;
}
```

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

```
Line 1 - = Operator, Value of c = : 21
Line 2 - += Operator, Value of c = : 42
Line 3 - -= Operator, Value of c = : 21
Line 4 - *= Operator, Value of c = : 441
Line 5 - /= Operator, Value of c = : 21
Line 6 - %= Operator, Value of c = : 11
Line 7 - <<= Operator, Value of c = : 44
Line 8 - >>= Operator, Value of c = : 11
Line 9 - &= Operator, Value of c = : 2
Line 10 - ^= Operator, Value of c = : 0
Line 11 - |= Operator, Value of c = : 2
```

Misc Operators

The following table lists some other operators that C++ supports.

Operator	Description
sizeof	sizeof operator returns the size of a variable. For example, sizeof(a), where ‘a’ is integer, and will return 4.
Condition ? X : Y	Conditional operator (?). If Condition is true then it returns value of X otherwise returns value of Y.
,	Comma operator causes a sequence of operations to be performed. The value of the entire comma expression is the value of the last expression of the comma-separated list.
. (dot) and -> (arrow)	Member operators are used to reference individual members of classes, structures, and unions.
Cast	Casting operators convert one data type to another. For example, int(2.2000) would return 2.
&	Pointer operator ‘&’ returns the address of a variable. For example &a; will give actual

	address of the variable.
*	Pointer operator * is pointer to a variable. For example *var; will pointer to a variable var.

Operators Precedence in C++

Operator precedence determines the grouping of terms in an expression. This affects how an expression is evaluated. Certain operators have higher precedence than others; for example, the multiplication operator has higher precedence than the addition operator:
For example x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first gets multiplied with 3*2 and then adds into 7. Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom. Within an expression, higher precedence operators will be evaluated first.

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Left to right
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=	Left to right
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right
Bitwise OR	|	Left to right
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

in C++. Copy and paste the following C++ program in test.cpp file and compile and run this program.
Check the simple difference with and without parenthesis. This will produce different results because (), /, * and + have different precedence. Higher precedence operators will be evaluated first:

```
#include <iostream>                                                  C++
using namespace std;
main()
{
int a = 20;
int b = 10;
int c = 15;
int d = 5;
int e;
e = (a + b) * c / d; // ( 30 * 15 ) / 5
cout << "Value of (a + b) * c / d is :" << e << endl ;
e = ((a + b) * c) / d; // (30 * 15 ) / 5
cout << "Value of ((a + b) * c) / d is :" << e << endl ;
e = (a + b) * (c / d); // (30) * (15/5)
cout << "Value of (a + b) * (c / d) is :" << e << endl ;
e = a + (b * c) / d; // 20 + (150/5)
cout << "Value of a + (b * c) / d is :" << e << endl ;
return 0;
}
```

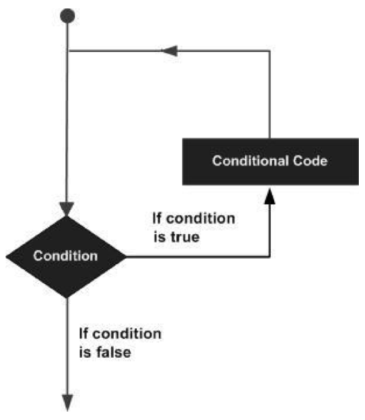
When the above code is compiled and executed, it produces the following result:

```
Value of (a + b) * c / d is :90
Value of ((a + b) * c) / d is :90
Value of (a + b) * (c / d) is :90
Value of a + (b * c) / d is :50
```

12. LOOP TYPES

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 type of loops to handle looping requirements.

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

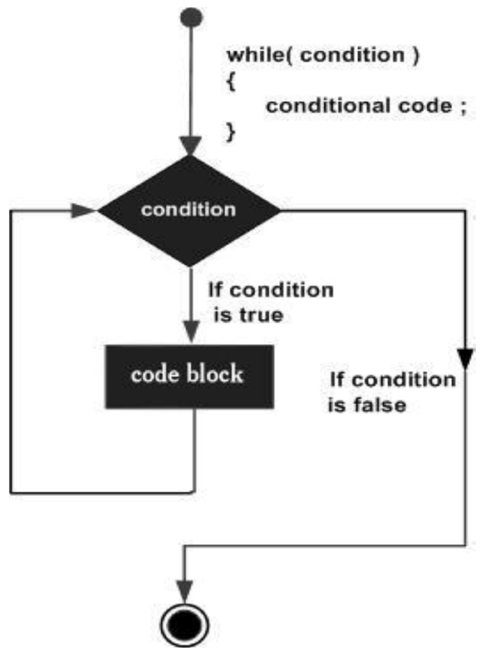
The syntax of a while loop in C++ is:

```
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>
using namespace std;
int main ()
{
// Local variable declaration:
int a = 10;
// while loop execution
while( a < 20 )
{
cout << "value of a: " << a << endl;
a++;
} return 0;
}
```

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

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

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

Here is the flow of control in a for loop:

1. 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.

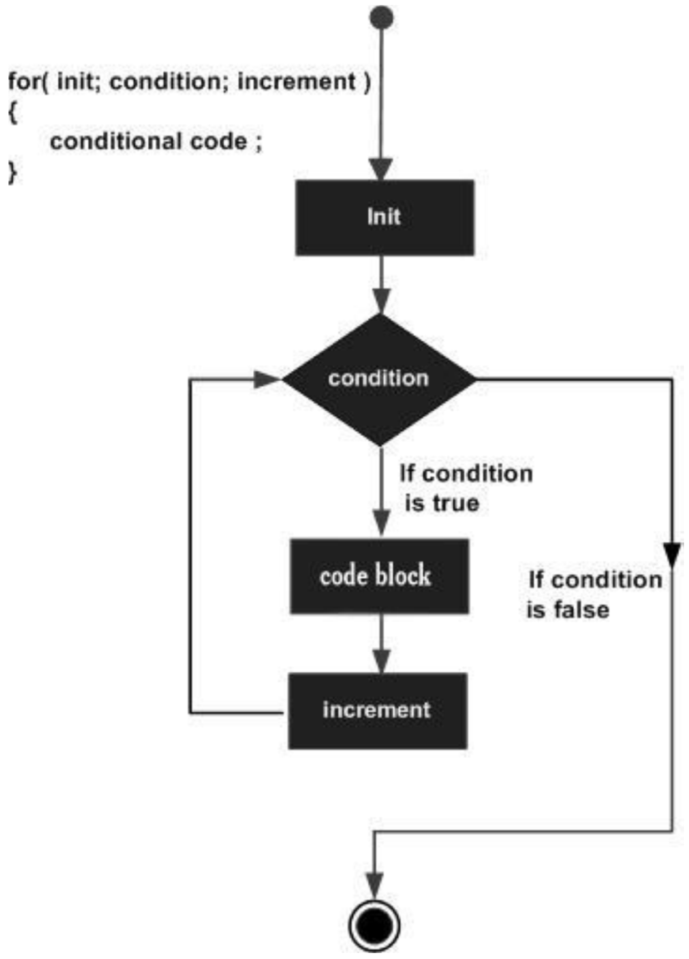
2. Next, the `condition` is evaluated. If it is true, the body of the loop is executed. If it is false, the body of the loop does not execute and flow of control jumps to the next statement just after the for loop.

3. 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.

4. 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.

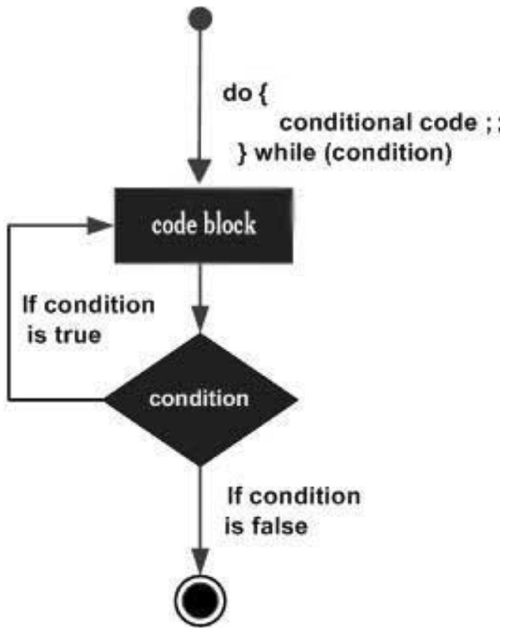
Flow Diagram



Example

```
#include <iostream>
using namespace std;
int main ()
{
// for loop execution C++
for( int a = 10; a < 20; a = a + 1 )
{
cout << "value of a: " << a << endl;
}
return 0;
}
```
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
The syntax of a do...while loop in C++ is:
```
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.



Flow Diagram

Example

```
#include <iostream>
using namespace std;
int main ()
{
// Local variable declaration:
int a = 10;
// do loop execution
do
{
cout << "value of a: " << a << endl;
a = a + 1;
}while( a < 20 ); C++
62
return 0;
}
```

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 Loops

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

Syntax

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

```
for ( init; condition; increment )
{
for ( init; condition; increment )
{
statement(s);
}
statement(s); // you can put more statements.
}
```

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

```
while(condition)
{
while(condition)
{
statement(s);
}
statement(s); // you can put more statements.
}
```

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

```
do
{
statement(s); // you can put more statements.
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>
using namespace std;
int main ()
{
int i, j;
for(i=2; i<100; i++) {
for(j=2; j <= (i/j); j++)
if(!(i%j)) break; // if factor found, not prime
if(j > (i/j)) cout << i << " is prime\n";
}
return 0;
}
```

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.

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.

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.

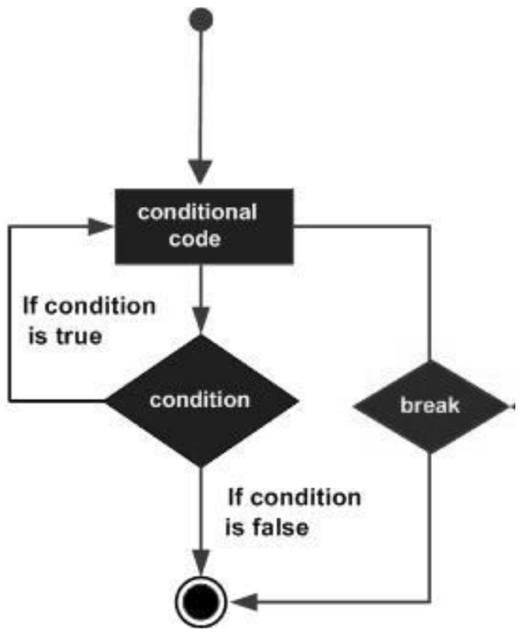
☐ It can be used to terminate a case in the `switch` statement (covered in the next chapter).

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.

Syntax

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

```
break;
```



Flow Diagram

Example

```
#include <iostream>
using namespace std;
int main ()
{
// Local variable declaration:
int a = 10;
// do loop execution
do
{
cout << "value of a: " << a << endl;
a = a + 1;
if( a > 15)
{
// terminate the loop
break;
}
                             }while( a < 20 )

return 0;
}
```

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

```
value of a: prettyprint notranslate10
value of a: 11
value of a: 12
value of a: 13
value of a: 14
value of a: 15
```

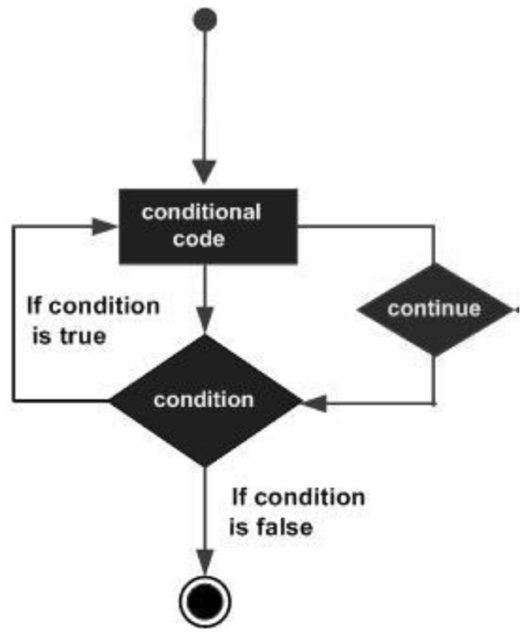
continue Statement

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

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

```
continue;
```



Flow Diagram

Example

```
#include <iostream>
using namespace std;
int main ()
{
// Local variable declaration:
int a = 10;
// do loop execution
do
{
if( a == 15)
{
// skip the iteration.
a = a + 1;
continue;
}
```

```
cout << "value of a: " << a << endl;
a = a + 1;
```

```
}while( a < 20 );
return 0;
}
```

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.

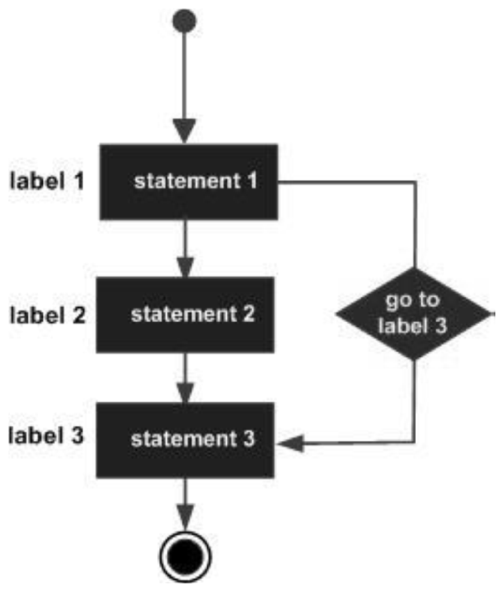
Syntax

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

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

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

Flow Diagram



Example

```
#include <iostream>
```

```
using namespace std;
int main ()
{
// Local variable declaration:
int a = 10;
// do loop execution
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 ); return 0;
}
```

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
```

One good use of goto is to exit from a deeply nested routine. For example, consider the following code fragment:

```
for(...) {
for(...) {
while(...) {
if(...) goto stop;
.
.
.
}
}
}
stop:
cout << "Error in program.\n";
```

Eliminating the `goto` would force a number of additional tests to be performed. A simple`break` statement would not work here, because it would only cause the program to exit from the innermost loop.

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>
```

```
using namespace std;
int main ()
{
for( ; ; )
{
printf("This loop will run forever.\n");
}
return 0;
}
```

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.