INTRODUCTION

A computer is a useful tool for solving a great variety of problems. To make a computer do anything (i.e. solve a problem), you have to write a computer program. In a computer program you tell a computer, step by step, exactly what you want it to do. The computer then executes the program, following each step mechanically, to accomplish the end goal.

Algorithm and flowchart are the powerful tools for learning programming. An algorithm is a step-by-step analysis of the process, while a flowchart explains the steps of a program in a graphical way. Algorithm and flowcharts helps to clarify all the steps for solving the problem.

ALGORITHMS

The word “algorithm” relates to the name of the mathematician Al-khowarizmi, which means a procedure or a technique. Software Engineer commonly uses an algorithm for planning and solving the problems. An algorithm is a sequence of steps to solve a particular problem or algorithm is an ordered set of unambiguous steps that produces a result and terminates in a finite time.

Algorithm has the following characteristics

- **Input**: An algorithm may or may not require input.
- **Output**: Each algorithm is expected to produce at least one result.
- **Definiteness**: Each instruction must be clear and unambiguous.
- **Finiteness**: If the instructions of an algorithm are executed, the algorithm should terminate after finite number of steps.

The algorithm and flowchart include following three types of control structures.

1. **Sequence**: In the sequence structure, statements are placed one after the other and the execution takes place starting from up to down.
2. **Branching** (Selection): In branch control, there is a condition and according to a condition, a decision of either TRUE or FALSE is achieved. Generally, the ‘IF-THEN’ is used to represent branch control.
3. **Loop** (Repetition): The Loop or Repetition allows a statement(s) to be executed repeatedly based on certain loop condition e.g. WHILE, FOR loops.

HOW TO WRITE ALGORITHMS

Step 1: Define your algorithms input: Many algorithms take in data to be processed, e.g. to calculate the area of rectangle input may be the rectangle height and rectangle width.

Step 2: Define the variables: Algorithm's variables allow you to use it for more than one place. We can define two variables for rectangle height and rectangle width as HEIGHT and WIDTH (or H & W). We should use meaningful variable name.

Step 3: **Outline the algorithm's operations**: Use input variable for computation purpose, e.g. to find area of rectangle multiply the HEIGHT and WIDTH variable and store the value in new variable (say) AREA. An algorithm's operations can take the form of multiple steps and even branch, depending on the value of the input variables.

Step 4: Output the results of your algorithm's operations: In case of area of rectangle output will be the value stored in variable AREA. if the input variables described a rectangle with a HEIGHT of 2 and a WIDTH of 3, the algorithm would output the value of 6.

FLOWCHART

The first design of flowchart goes back to 1945 which was designed by John Von Neumann. Unlike an algorithm, Flowchart uses different symbols to design a solution to a problem. It is another commonly used programming tool. By looking at a Flowchart can understand the operations and sequence of operations performed in a system. Flowchart is often considered as a blueprint of a design used for solving a specific problem.

Flowchart is diagrammatic /Graphical representation of sequence of steps to solve a problem. To draw a flowchart following standard symbols are use.

Symbol Name	Symbol	function
Oval		Used to represent start and end of flowchart
Parallelogram		Used for input and output operation
Rectangle		Processing: Used for arithmetic operations and data-manipulations
Diamond		Decision making. Used to represent the operation in which there are two/three alternatives, true and false etc
Arrows		Flow line Used to indicate the flow of logic by connecting symbols
Circle		Page Connector
		Off Page Connector
		Predefined Process /Function Used to represent a group of statements performing one processing task.
		Preprocessor
	|---------- -------- | |----------	Comments

The language used to write algorithm is simple and similar to day-to-day life language. The variable names are used to store the values. The value store in variable can change in the solution steps. In addition some special symbols are used as below.

Assignment Symbol (← or =) is used to assign value to the variable.

e.g. to assign value 5 to the variable HEIGHT, statement is

HEIGHT ← 5 or HEIGHT = 5

The symbol '=' is used in most of the programming language as an assignment symbol, the same has been used in all the algorithms and flowcharts in the manual.

The statement C = A + B means that add the value stored in variable A and variable B then assign/store the value in variable C.

The statement R = R + 1 means that add I to the value stored in variable R and then assign/store the new value in variable R, in other words increase the value of variable R by 1

Mathematical Operators

Operator	Meaning	Example
+	Addition	A + B
-	Subtraction	A – B
*	Multiplication	A * B
/	Division	A / B
^	Power	A^3 for A^3
%	Reminder	A % B

Relational Operators

Operator	Meaning	Example
<	Less than	A < B
<=	Less than or equal to	A <= B
= or ==	Equal to	A = B
# or !=	Not equal to	A # B or A !=B
>	Greater than	A > B
>=	Greater tha or equal to	A >= B

Logical Operators

Operator	Example	Meaning
AND	A < B AND B < C	Result is True if both A<B and B<C are true else false
OR	A< B OR B < C	Result is True if either A<B or B<C are true else false
NOT	NOT (A >B)	Result is True if A>B is false else true

Selection control Statements

Selection Control	Example	Meaning

IF (Condition) Then
…
ENDIF
	IF (X > 10) THEN
	Y=Y+5
	ENDIF
		If condition X>10 is True execute the statement between THEN and ENDIF

IF (Condition) Then
…
ELSE
…..
ENDIF
	IF (X > 10) THEN
	Y=Y+5
	ELSE
	Y=Y+8
	Z=Z+3
	ENDIF
		If condition X>10 is True execute the statement between THEN and ELSE otherwise execute the statements between ELSE and ENDIF

Loop control Statements

Selection Control	Example	Meaning

WHILE (Condition)
DO
..
..
ENDDO
	WHILE (X < 10)
	DO
	print x
	x=x+1
	ENDDO
		Execute the loop as long as the condition is TRUE

DO …. … UNTILL (Condition)	DO print x x=x+1 UNTILL (X >10)	Execute the loop as long as the condition is false

GO TO statement also called unconditional transfer of control statement is used to transfer control of execution to another step/statement. . e.g. the statement GOTO n will transfer control to step/statement n.

Note: We can use keyword INPUT or READ or GET to accept input(s) /value(s) and keywords PRINT or WRITE or DISPLAY to output the result(s).

Algorithm & Flowchart to find the sum of two numbers

Algorithm

Step-1 Start

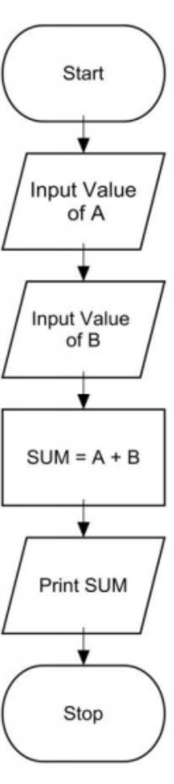
Step-2 Input first numbers say A

Step-3 Input second number say B

Step-4 SUM = A + B

Step-5 Display SUM

Step-6 Stop



OR

Algorithm

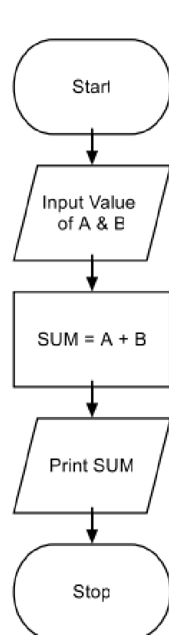
Step-1 Start

Step-2 Input two numbers say A & B

Step-3 SUM = A + B

Step-4 Display SUM

Step-5 Stop



Algorithm & Flowchart to convert temperature from Celsius to Fahrenheit

C : temperature in Celsius
F : temperature Fahrenheit

Algorithm

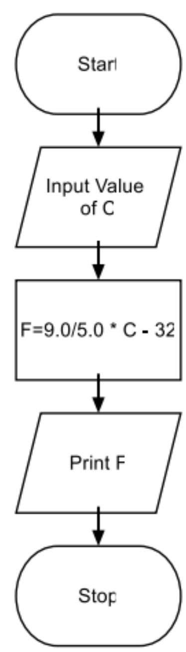
Step-1 Start

Step-2 Input temperature in Celsius say C

Step-3 F = (9.0/5.0 x C) + 32

Step-4 Display Temperature in Fahrenheit F

Step-5 Stop



Algorithm & Flowchart to find Area and Perimeter of Square

L : Side Length of Square
AREA : Area of Square
PERIMETER : Perimeter of Square

Algorithm

Step-1 Start

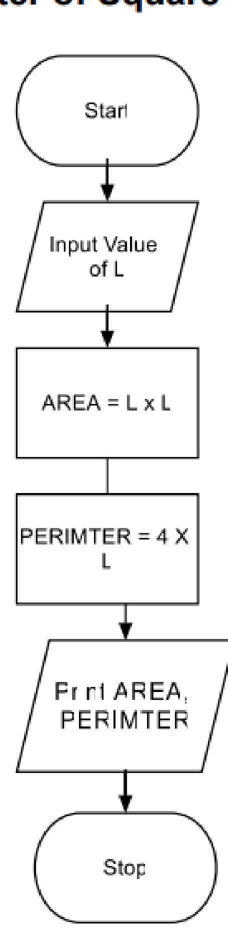
Step-2 Input Side Length of Square say L

Step-3 Area = L x L

Step-4 PERIMETER = 4 x L

Step-5 Display AREA, PERIMETER

Step-6 Stop



Algorithm & Flowchart to Swap Two Numbers using Temporary Variable

Algorithm

Step-1 Start

Step-2 Input Two Numbers Say NUM1,NUM2

Step-3 Display Before Swap Values NUM1, NUM2

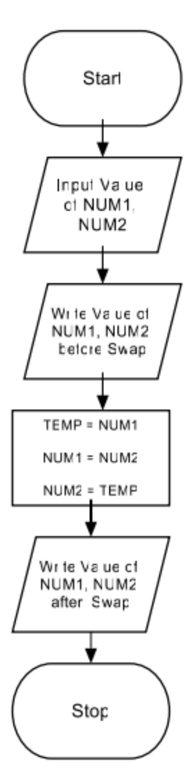
Step-4 TEMP = NUM1

Step-5 NUM1 = NUM2

Step-6 NUM2 = NUM1

Step-7 Display After Swap Values NUM1,NUM

Step-8 Stop



Algorithm & Flowchart to find Even number between 1 to 50

Algorithm

Step-1 Start

Step-2 I = 1

Step-3 IF (I >50) THEN

GO TO Step-7

ENDIF

Step-4 IF ((I % 2) =0) THEN

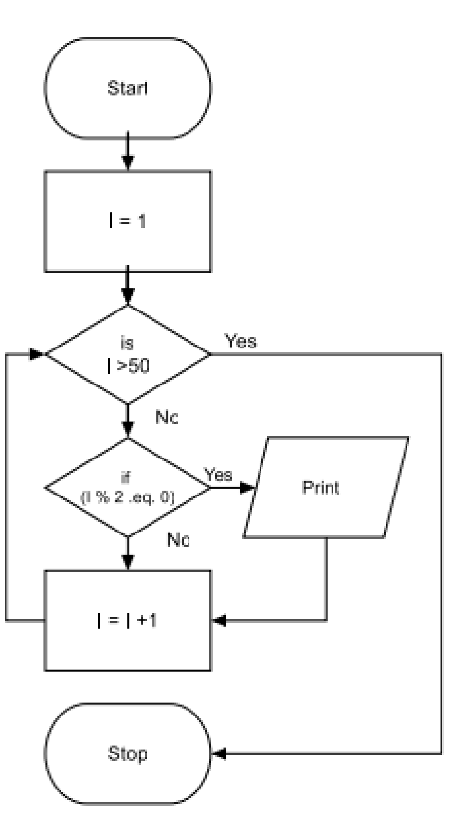
Display I

ENDIF

Step-5 I = I + 1

Step-6 GO TO Step--3

Step-7 Stop



C++ Introduction

C++ is a middle-level programming language developed by Bjarne Stroustrup starting in 1979 at Bell Labs. **C++** runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. This **C++** tutorial adopts a simple and practical approach to describe the concepts of **C++** for beginners to advanced software engineers.

Why to Learn C++

C++ is necessary for students and working professionals to become a great Software Engineer. I will list down some of the key advantages of learning C++:

- C++ is very close to hardware, so you get a chance to work at a low level which gives you lot of control in terms of memory management, better performance and finally a robust software development.
- **C++ programming** gives you a clear understanding about Object Oriented Programming. You will understand low-level implementation of polymorphism when you will implement virtual tables and virtual table pointers, or dynamic type identification.
- C++ is one of the every green programming languages and loved by millions of software developers. If you are a great C++ programmer, then you will never sit without work and more importantly, you will get highly paid for your work.
- C++ is the most widely used programming languages in application and system programming. So you can choose your area of interest of software development.
- C++ really teaches you the difference between compiler, linker and loader, different data types, storage classes, variable types their scopes etc.

There are 1000s of good reasons to learn C++ Programming. But one thing for sure, to learn any programming language, not only C++, you just need to code, and code and finally code until you become expert.

Applications of C++ Programming

As mentioned before, C++ is one of the most widely used programming languages. It has it's presence in almost every area of software development. I'm going to list few of them here:

- **Application Software Development** - C++ programming has been used in developing almost all the major Operating Systems like Windows, Mac OSX and Linux. Apart from the operating systems, the core part of many browsers like Mozilla Firefox and Chrome have been written using C++. C++ also has been used in developing the most popular database system called MySQL.
- **Programming Languages Development** - C++ has been used extensively in developing new programming languages like C#, Java, JavaScript, Perl, UNIX's C Shell, PHP and Python, and Verilog etc.
- **Computation Programming** - C++ is the best friends of scientists because of fast speed and computational efficiencies.
- **Games Development** - C++ is extremely fast which allows programmers to do procedural programming for CPU intensive functions and provides greater control over hardware, because of which it has been widely used in development of gaming engines.
- **Embedded System** - C++ is being heavily used in developing Medical and Engineering Applications like software's for MRI machines, high-end CAD/CAM systems etc.

This list goes on, there are various areas where software developers are happily using C++ to provide great software's. I highly recommend you to learn C++ and contribute great software's to the community.

Hello World – First C++ Program

Just to give you a little excitement about **C++ programming**, we going to give you a small conventional C++ Hello World program.

C++ is a super set of C programming with additional implementation of object-oriented concepts.

C++ Program Structure

In this guide, we will write and understand the **first program in C++** programming. We are writing a simple C++ program that prints “Hello World!” message. Let’s see the program first and then we will discuss each and every part of it in detail.

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

// main() is where program execution begins.
int main() {
   cout << "Hello World"; // prints Hello World
   return 0;
}
```

Let us look at the various parts of the above program −

- The C++ language defines several headers, which contain information that is either necessary or useful to your program. For this program, the header **<iostream>** is needed.
- The line **using namespace std;** tells the compiler to use the std namespace. Namespaces are a relatively recent addition to C++.
- The next line '**// main() is where program execution begins.**' is a single-line comment available in C++. Single-line comments begin with // and stop at the end of the line.
- The line **int main()** is the main function where program execution begins.
- The next line **cout << "Hello World";** causes the message "Hello World" to be displayed on the screen.
- The next line **return 0;** terminates main( )function and causes it to return the value 0 to the calling process.

Semicolons and Blocks in C++

In C++, the semicolon is a statement terminator. That is, each individual statement must be ended with a semicolon. It indicates the end of one logical entity.

For example, following are three different statements −

```
x = y;
y = y + 1;
add(x, y);
```

A block is a set of logically connected statements that are surrounded by opening and closing braces. For example −

```
{
   cout << "Hello World"; // prints Hello World
   return 0;
}
```

C++ does not recognize the end of the line as a terminator. For this reason, it does not matter where you put a statement in a line. For example −

```
x = y;
y = y + 1;
add(x, y);
```

is the same as

```
x = y; y = y + 1; add(x, y);
```

C++ Identifiers

A C++ identifier is a name used to identify a variable, function, class, module, or any other user-defined item. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores, and digits (0 to 9).

C++ does not allow punctuation characters such as @, $, and % within identifiers. C++ is a case-sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in C++.

Here are some examples of acceptable identifiers −

```
mohd        zara    abc    move_name   a_123
myname50    _temp   j      a23b9       retVal
```

C++ Keywords

The following list shows the reserved words in C++. These reserved words may not be used as constant or variable or any other identifier names.

asm	else	new	this
auto	enum	operator	throw
bool	explicit	private	true
break	export	protected	try
case	extern	public	typedef
catch	false	register	typeid
char	float	reinterpret_cast	typename
class	for	return	union
const	friend	short	unsigned
const_cast	goto	signed	using
continue	if	sizeof	virtual
default	inline	static	void
delete	int	static_cast	volatile
do	long	struct	wchar_t
double	mutable	switch	while
dynamic_cast	namespace	template	

Whitespace in C++

A line containing only whitespace, possibly with a comment, is known as a blank line, and C++ compiler totally ignores it.

Whitespace is the term used in C++ to describe blanks, tabs, newline characters and comments. Whitespace separates one part of a statement from another and enables the compiler to identify where one element in a statement, such as int, ends and the next element begins.

Statement 1

```
int age;
```

In the above statement there must be at least one whitespace character (usually a space) between int and age for the compiler to be able to distinguish them.

Statement 2

```
fruit = apples + oranges;   // Get the total fruit
```

In the above statement 2, no whitespace characters are necessary between fruit and =, or between = and apples, although you are free to include some if you wish for readability purpose.

Comments in C++

Program comments are explanatory statements that you can include in the C++ code. These comments help anyone reading the source code. All programming languages allow for some form of comments.

C++ supports single-line and multi-line comments. All characters available inside any comment are ignored by C++ compiler.

C++ comments start with /* and end with */. For example −

```
/* This is a comment */

/* C++ comments can also
   * span multiple lines
*/
```

A comment can also start with //, extending to the end of the line. For example −

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

main() {
   cout << "Hello World"; // prints Hello World

   return 0;
}
```

When the above code is compiled, it will ignore **// prints Hello World** and final executable will produce the following result −

```
Hello World
```

Within a /* and */ comment, // characters have no special meaning. Within a // comment, /* and */ have no special meaning. Thus, you can "nest" one kind of comment within the other kind. For example −

```
/* Comment out printing of Hello World:

cout << "Hello World"; // prints Hello World

*/
```

Variables in C++

A variable is a name which is associated with a value that can be changed. For example when I write int num=20; here variable name is num which is associated with value 20, int is a data type that represents that this variable can hold integer values. We will cover the data types in the next tutorial. In this tutorial, we will discuss about variables.

Syntax of declaring a variable in C++

```
data_type variable1_name = value1, variable2_name = value2;
```

For example:

```
int num1=20, num2=100;
```

We can also write it like this:

```
int num1,num2;
num1=20;
num2=100;
```

Types of variables

Variables can be categorised based on their data type. For example, in the above example we have seen integer types variables. Following are the types of variables available in C++.

int: These type of of variables holds integer value.

char: holds character value like ‘c’, ‘F’, ‘B’, ‘p’, ‘q’ etc.

bool: holds boolean value true or false.

double: double-precision floating point value.

float: Single-precision floating point value.

Types of variables based on their scope

Before going further lets discuss what is scope first. When we discussed the Hello World Program, we have seen the curly braces in the program like this:

```
int main {

//Some code

}
```

Any variable declared inside these curly braces have scope limited within these curly braces, if you declare a variable in main() function and try to use that variable outside main() function then you will get compilation error.

Now that we have understood what is scope. Lets move on to the types of variables based on the scope.

1. Global variable
2. Local variable

Global Variable

A variable declared outside of any function (including main as well) is called global variable. Global variables have their scope throughout the program, they can be accessed anywhere in the program, in the main, in the user defined function, anywhere.

Lets take an example to understand it:

Global variable example

Here we have a global variable `myVar`, that is declared outside of main. We have accessed the variable twice in the main() function without any issues.

```
#include <iostream>
using namespace std;
// This is a global variable
char myVar = 'A';
int main()
```

```
{
   cout <<"Value of myVar: "<< myVar<<endl;
   myVar='Z';
   cout <<"Value of myVar: "<< myVar;
   return 0;
}
```

Output:

```
Value of myVar: A
Value of myVar: Z
```

Local variable

Local variables are declared inside the braces of any user defined function, main function, loops or any control statements(if, if-else etc) and have their scope limited inside those braces.

Local variable example

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

char myFuncn() {
// This is a local variable
char myVar = 'A';
}
int main()
{
   cout <<"Value of myVar: "<< myVar<<endl;
   myVar='Z';
   cout <<"Value of myVar: "<< myVar;
   return 0;
}
```

Output:
Compile time error, because we are trying to access the variable `myVar` outside of its scope. The scope of `myVar` is limited to the body of function `myFuncn()`, inside those braces.

Can global and local variable have same name in C++?

Lets see an example having same name global and local variable.

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

```
// This is a global variable
char myVar = 'A';
char myFuncn() {
   // This is a local variable
   char myVar = 'B';
   return myVar;
}
int main()
{
   cout <<"Funcn call: "<< myFuncn()<<endl;
   cout <<"Value of myVar: "<< myVar<<endl;
   myVar='Z';
   cout <<"Funcn call: "<< myFuncn()<<endl;
   cout <<"Value of myVar: "<< myVar<<endl;
   return 0;
}
```

Output:

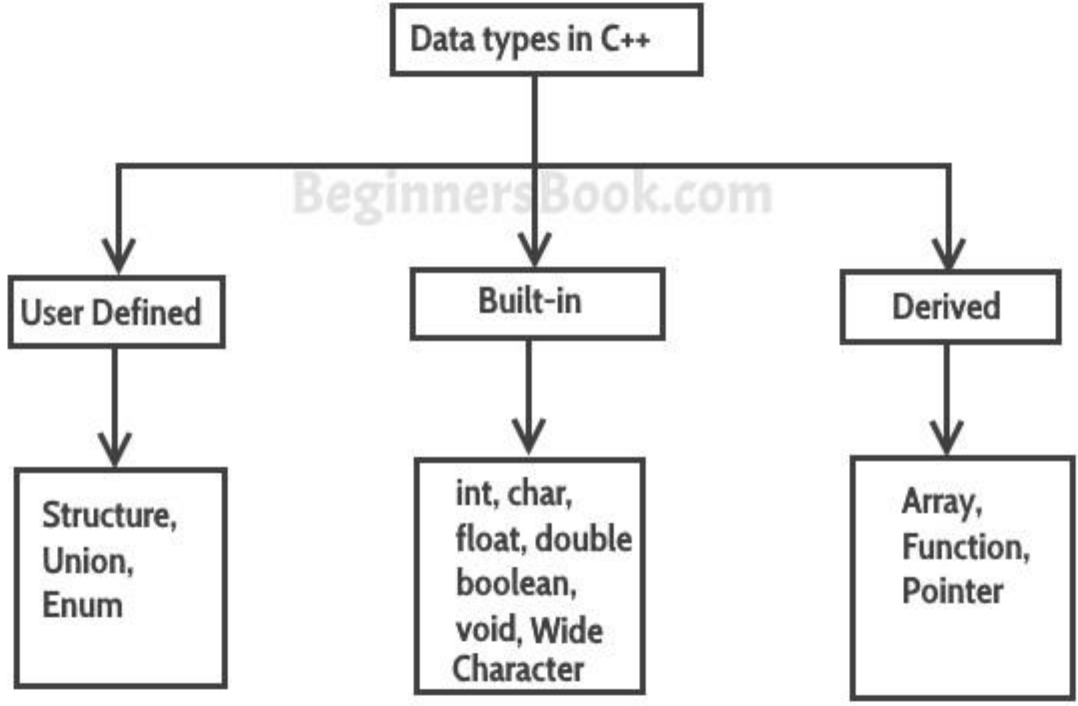
```
Funcn call: B
Value of myVar: A
Funcn call: B
Value of myVar: Z
```

As you can see that when I changed the value of `myVar` in the main function, it only changed the value of global variable `myVar` because local variable `myVar` scope is limited to the function `myFuncn()`.

C++ Data Types

Data types define the type of data a variable can hold, for example an integer variable can hold integer data, a character type variable can hold character data etc.

Data types in C++ are categorised in three groups: **Built-in**, **user-defined** and **Derived**.



Built in data types

char: For characters. Size 1 byte.

```
char ch = 'A';
```

int: For integers. Size 2 bytes.

```
int num = 100;
```

float: For single precision floating point. Size 4 bytes.

```
float num = 123.78987;
```

double: For double precision floating point. Size 8 bytes.

```
double num = 10098.98899;
```

bool: For booleans, true or false.

```
bool b = true;
```

The following table shows the variable type, how much memory it takes to store the value in memory, and what is maximum and minimum value which can be stored in such type of variables.

Type	Typical Bit Width	Typical Range
char	1byte	-127 to 127 or 0 to 255
unsigned char	1byte	0 to 255
signed char	1byte	-127 to 127
int	4bytes	-2147483648 to 2147483647
unsigned int	4bytes	0 to 4294967295
signed int	4bytes	-2147483648 to 2147483647
short int	2bytes	-32768 to 32767
unsigned short int	2bytes	0 to 65,535
signed short int	2bytes	-32768 to 32767
long int	8bytes	-2,147,483,648 to 2,147,483,647
signed long int	8bytes	same as long int
unsigned long int	8bytes	0 to 4,294,967,295
long long int	8bytes	-(2^63) to (2^63)-1
unsigned long long int	8bytes	0 to 18,446,744,073,709,551,615
float	4bytes	
double	8bytes	
long double	12bytes	
wchar_t	2 or 4 bytes	1 wide character

The size of variables might be different from those shown in the above table, depending on the compiler and the computer you are using.

User-defined data types

We have three types of user-defined data types in C++
1. struct
2. union
3. enum

I have covered them in detail in separate tutorials. For now just remember that these comes under user-defined data types.

Derived data types in C++

We have three types of derived-defined data types in C++
1. Array
2. Function
3. Pointer

They are wide topics of C++ and I have covered them in separate tutorials. Just follow the tutorials in given sequence and you would be fine.

Variable Scope in C++

A scope is a region of the program and broadly speaking there are three places, where variables can be declared −

- Inside a function or a block which is called local variables,
- In the definition of function parameters which is called formal parameters.
- Outside of all functions which is called global variables.

We will learn what is a function and it's parameter in subsequent chapters. Here let us explain what are local and global variables.

Local Variables

Variables that are declared inside a function or block are local variables. They can be used only by statements that are inside that function or block of code. Local variables are not known to functions outside their own. Following is the example using local variables −

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

int main () {
   // Local variable declaration:
   int a, b;
   int c;

   // actual initialization
   a = 10;
   b = 20;
   c = a + b;

   cout << c;

   return 0;
}
```

Global Variables

Global variables are defined outside of all the functions, usually on top of the program. The global variables will hold their value throughout the life-time of your program.

A global variable can be accessed by any function. That is, a global variable is available for use throughout your entire program after its declaration. Following is the example using global and local variables −

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

// Global variable declaration:
int g;

int main () {
   // Local variable declaration:
   int a, b;

   // actual initialization
   a = 10;
   b = 20;
   g = a + b;

   cout << g;

   return 0;
}
```

A program can have same name for local and global variables but value of local variable inside a function will take preference. For example −

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

// Global variable declaration:
int g = 20;

int main () {
   // Local variable declaration:
   int g = 10;

   cout << g;

   return 0;
}
```

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

```
10
```

Initializing Local and Global Variables

When a local variable is defined, it is not initialized by the system, you must initialize it yourself. Global variables are initialized automatically by the system when you define them as follows −

Data Type	Initializer
int	0
char	'\0'
float	0
double	0
pointer	NULL

It is a good programming practice to initialize variables properly, otherwise sometimes program would produce unexpected result.

C++ Constants/Literals

Constants refer to fixed values that the program may not alter and they are called **literals**.

Constants can be of any of the basic data types and can be divided into Integer Numerals, Floating-Point Numerals, Characters, Strings and Boolean Values.

Again, constants are treated just like regular variables except that their values cannot be modified after their definition.

Integer Literals

An integer literal can be a decimal, octal, or hexadecimal constant. A prefix specifies the base or radix: 0x or 0X for hexadecimal, 0 for octal, and nothing for decimal.

An integer literal can also have a suffix that is a combination of U and L, for unsigned and long, respectively. The suffix can be uppercase or lowercase and can be in any order.

Here are some examples of integer literals −

```
212         // Legal
215u        // Legal
0xFeeL      // Legal
078         // Illegal: 8 is not an octal digit
032UU       // Illegal: cannot repeat a suffix
```

Following are other examples of various types of Integer literals −

```
85         // decimal
0213       // octal
0x4b       // hexadecimal
30         // int
30u        // unsigned int
30l        // long
30ul       // unsigned long
```

Floating-point Literals

A floating-point literal has an integer part, a decimal point, a fractional part, and an exponent part. You can represent floating point literals either in decimal form or exponential form.

While representing using decimal form, you must include the decimal point, the exponent, or both and while representing using exponential form, you

must include the integer part, the fractional part, or both. The signed exponent is introduced by e or E.

Here are some examples of floating-point literals −

```
3.14159       // Legal
314159E-5L    // Legal
510E          // Illegal: incomplete exponent
210f          // Illegal: no decimal or exponent
.e55          // Illegal: missing integer or fraction
```

Boolean Literals

There are two Boolean literals and they are part of standard C++ keywords −

- A value of **true** representing true.
- A value of **false** representing false.

You should not consider the value of true equal to 1 and value of false equal to 0.

Character Literals

Character literals are enclosed in single quotes. If the literal begins with L (uppercase only), it is a wide character literal (e.g., L'x') and should be stored in **wchar_t** type of variable . Otherwise, it is a narrow character literal (e.g., 'x') and can be stored in a simple variable of **char** type.

A character literal can be a plain character (e.g., 'x'), an escape sequence (e.g., '\t'), or a universal character (e.g., '\u02C0').

There are certain characters in C++ when they are preceded by a backslash they will have special meaning and they are used to represent like newline (\n) or tab (\t). Here, you have a list of some of such escape sequence codes

Escape sequence	Meaning
\\	\ character
\'	' character
\"	" character

\?	? character
\a	Alert or bell
\b	Backspace
\f	Form feed
\n	Newline
\r	Carriage return
\t	Horizontal tab
\v	Vertical tab
\ooo	Octal number of one to three digits
\xhh . . .	Hexadecimal number of one or more digits

Following is the example to show a few escape sequence characters −

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

int main() {
   cout << "Hello\tWorld\n\n";
   return 0;
}
```

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

```
Hello   World
```

String Literals

String literals are enclosed in double quotes. A string contains characters that are similar to character literals: plain characters, escape sequences, and universal characters.

You can break a long line into multiple lines using string literals and separate them using whitespaces.

Here are some examples of string literals. All the three forms are identical strings.

```
"hello, dear"

"hello, \

dear"

"hello, " "d" "ear"
```

Defining Constants

There are two simple ways in C++ to define constants −

- Using **#define** preprocessor.
- Using **const** keyword.

The #define Preprocessor

Following is the form to use #define preprocessor to define a constant −

```
#define identifier value
```

Following example explains it in detail −

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

#define LENGTH 10
#define WIDTH  5
#define NEWLINE '\n'

int main() {
   int area;

   area = LENGTH * WIDTH;
   cout << area;
   cout << NEWLINE;
   return 0;
}
```

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

```
50
```

The const Keyword

You can use **const** prefix to declare constants with a specific type as follows

```
const type variable = value;
```

Following example explains it in detail −

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

int main() {
   const int  LENGTH = 10;
   const int  WIDTH  = 5;
   const char NEWLINE = '\n';
   int area;

   area = LENGTH * WIDTH;
   cout << area;
   cout << NEWLINE;
   return 0;
}
```

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

```
50
```

Note that it is a good programming practice to define constants in CAPITALS.

Operators in C++

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 −

Show Examples

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

Relational Operators

There are following relational operators supported by C++ language

Assume variable A holds 10 and variable B holds 20, then −

Show Examples

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 condition becomes true.	(A <= B) is true.

Logical Operators

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

Assume variable A holds 1 and variable B holds 0, then −

Show Examples

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.

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 −

Show Examples

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

Assignment Operators

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

Show Examples

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 + A
-=	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 assign the result to left operand.	C %= A is equivalent to C = C % A
<<=	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

Misc Operators

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

Sr.No	Operator & Description
1	**sizeof** sizeof operator returns the size of a variable. For example, sizeof(a), where 'a' is integer, and will return 4.
2	**Condition ? X : Y** Conditional operator (?). If Condition is true then it returns value of X otherwise returns value of Y.
3	**,** 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.
4	**. (dot) and -> (arrow)** Member operators are used to reference individual members of classes, structures, and unions.
5	**Cast** Casting operators convert one data type to another. For example, int(2.2000) would return 2.
6	**&** Pointer operator & returns the address of a variable. For example &a; will give actual address of the variable.
7	***** 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.

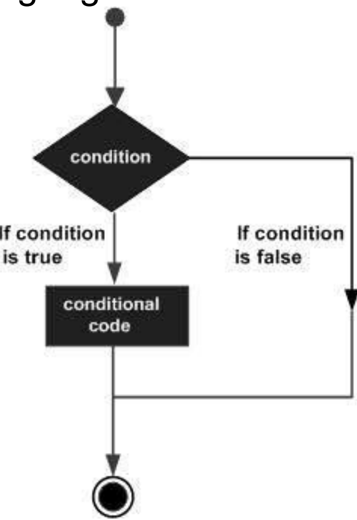
Show Examples

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

C++ decision making statements

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.

Sr.No	Statement & Description
1	if statement An 'if' statement consists of a boolean expression followed by one or more statements.
2	if...else statement An 'if' statement can be followed by an optional 'else' statement, which executes when the boolean expression is false.
3	switch statement A 'switch' statement allows a variable to be tested for equality against a list of values.
4	nested if statements You can use one 'if' or 'else if' statement inside another 'if' or 'else if' statement(s).
5	nested switch statements You can use one 'switch' statement inside another 'switch' statement(s).

C++ if statement

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

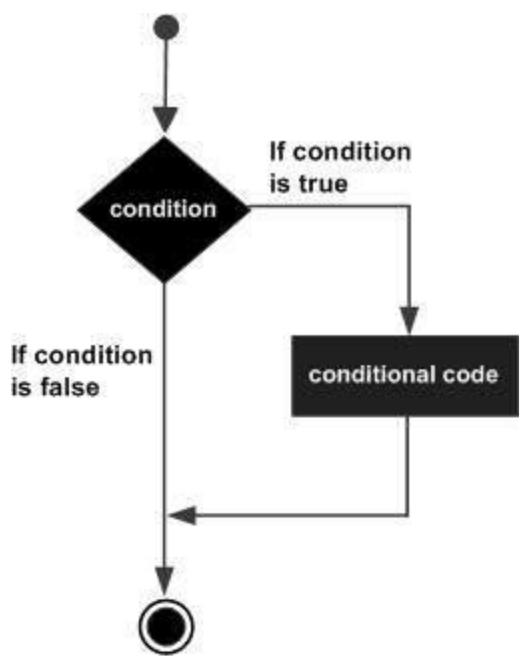
Syntax

The syntax of an if statement in C++ is −

```
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.

Flow Diagram



Example

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

int main () {
   // local variable declaration:
   int a = 10;

   // check the boolean condition
```

```
   if( a < 20 ) {
      // if condition is true then print the following
      cout << "a is less than 20;" << endl;
   }
   cout << "value of a is : " << a << endl;

   return 0;
}
```

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

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

C++ if...else statement

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

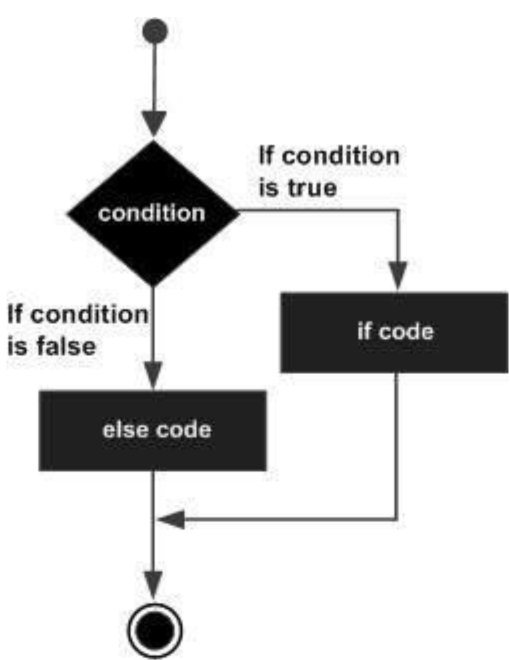
Syntax

The syntax of an if...else statement in C++ is −

```
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.

Flow Diagram



Example

Live Demo

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

int main () {
   // local variable declaration:
   int a = 100;
```

```
   // check the boolean condition
   if( a < 20 ) {
      // if condition is true then print the following
      cout << "a is less than 20;" << endl;
   } else {
      // if condition is false then print the following
      cout << "a is not less than 20;" << endl;
   }
   cout << "value of a is : " << a << endl;

   return 0;
}
```

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

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

if...else if...else Statement

An **if** statement can be followed by an optional **else if...else** statement, which is very usefull 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

The syntax of an if...else if...else statement in C++ is −

```
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>
using namespace std;

int main () {
   // local variable declaration:
   int a = 100;

   // check the boolean condition
   if( a == 10 ) {
      // if condition is true then print the following
      cout << "Value of a is 10" << endl;
   } else if( a == 20 ) {
      // if else if condition is true
      cout << "Value of a is 20" << endl;
   } else if( a == 30 ) {
      // if else if condition is true
      cout << "Value of a is 30" << endl;
   } else {
      // if none of the conditions is true
      cout << "Value of a is not matching" << endl;
   }
   cout << "Exact value of a is : " << a << endl;

   return 0;
}
```

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++ switch statement

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

The syntax for a **switch** statement in C++ is as follows −

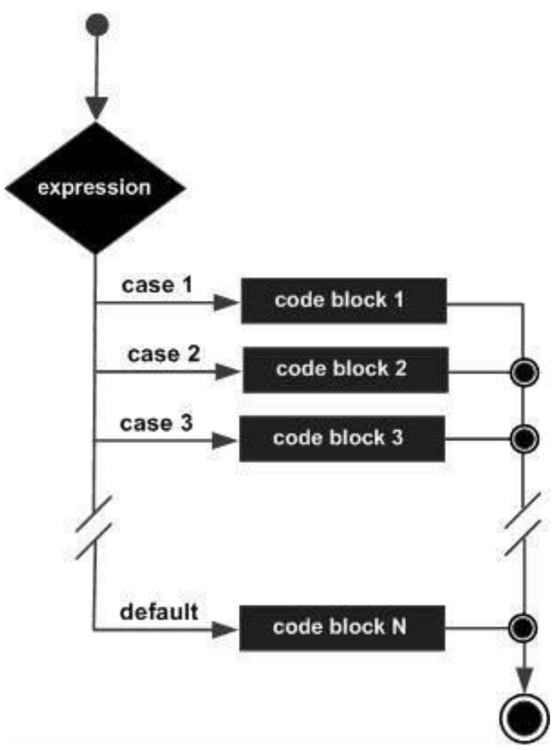
```
switch(expression) {
   case constant-expression  :
      statement(s);
      break; //optional
   case constant-expression  :
      statement(s);
      break; //optional

   // you can have any number of case statements.
   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.

Flow Diagram



Example

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

int main () {
   // local variable declaration:
   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;

   return 0;
}
```

This would produce the following result −

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

C++ nested if statements

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

The syntax for a **nested if** statement is as follows −

```
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>
using namespace std;

int main () {
   // local variable declaration:
   int a = 100;
   int b = 200;

   // check the boolean condition
   if( a == 100 ) {
      // if condition is true then check the following
      if( b == 200 ) {
         // if condition is true then print the following
         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;

   return 0;
}
```

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
```

C++ switch statement

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

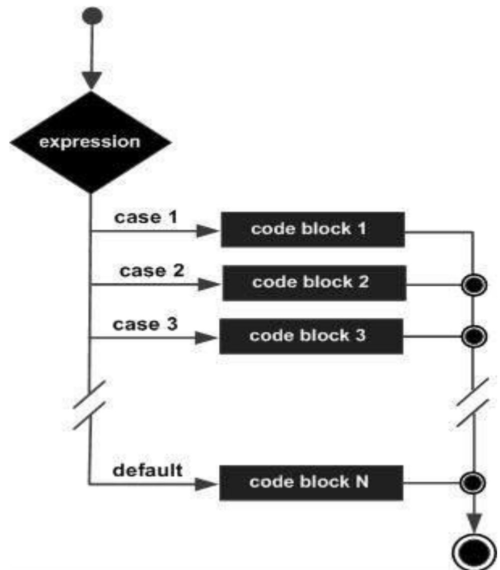
The syntax for a **switch** statement in C++ is as follows −

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

   // you can have any number of case statements.
   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.

Flow Diagram



Example

```
#include <iostream>
using namespace std;
int main () {
   // local variable declaration:
   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;
   return 0;
}
```

This would produce the following result −

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