CHAPTER-2

part-1

□**Review of fundamental constructs of C used in C++**: Character set, Keyword, Constant, Variable, Data types, operator & expression, control structure (branching & looping),array & strings

part-2

□**C++ Programming basics**: Streams based I/O (Input with cin, Output using cout), Type bool, The setw manipulator, Type conversions, strict type checking, name space, scope resolution operator (::),typecasting.

part-3

□**Variables:** Scope & lifetime of variables, variable declaration at the point of use, Ordinary Variable Vs. Pointer Variable Vs. Reference Variable (variable aliases)

Function: Parameter passing by value Vs. by address Vs. by reference, inline function, function overloading, default arguments.

chapter-2 part-I

Character set in C++

Character set is a set of valid characters that a language can recognise. A character represents any letter, digits, or any other sign.

C++ has the following character set :

Letters : A-Z, a-z

Digits : 0-9

Special Symbols : Space + - ∗ ⁄ ^ \ ( ) [ ] { } = != < > . ′ ″ $, ; : % ! & _ # <= >= @

White Spaces : Blank space, Horizontal tab (→), Carriage return (↵), Newline, Form feed

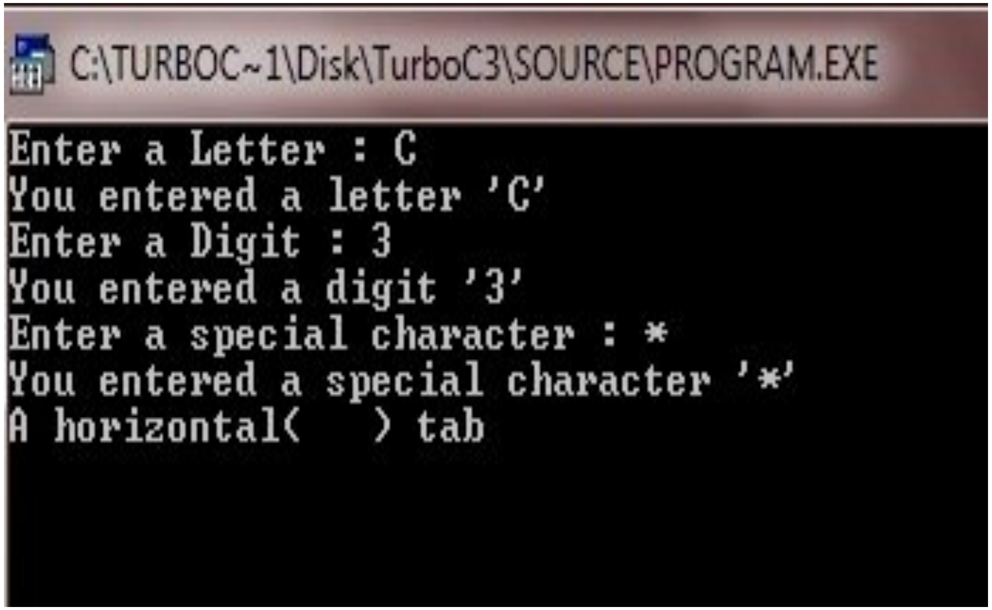
Other Characters : C++ can process any of the 256 ASCII characters as data or as literals.

C++ Character Set Example

```
/* C++ Character Set Example */

#include<iostream.h>

void main()
{
	clrscr();
	char letter, digit, special, white;

	cout<<"Enter a Letter : ";
	cin>>letter;
	cout<<"You entered a letter
'"<<letter<<"'"<<"\n";
```

```
cout<<"Enter a Digit : ";
	cin>>digit;
	cout<<"You entered a digit
'"<<digit<<"'"<<"\n";

	cout<<"Enter a special character
: ";
	cin>>special;
	cout<<"You entered a special
character '"<<special<<"'"<<"\n";

	cout<<"A horizontal(\t) tab";
}
```

output:



```
Enter a Letter : C
You entered a letter 'C'
Enter a Digit : 3
You entered a digit '3'
Enter a special character : *
You entered a special character '*'
A horizontal(    ) tab
```

Constants in c++

A constant, like a variable, is a memory location where a value can be stored. Unlike variables, constants never change in value. You must initialize a constant when it is created. C++ has two types of constants: literal and symbolic.

A literal constant is a value typed directly into your program wherever it is needed. For example, consider the following statement:

```
long width = 5;
```

This statement assigns the integer variable width the value 5. The 5 in the statement is a literal constant. You can't assign a value to 5, and its value can't be changed.

The values true and false, which are stored in bool variables, also are literal constants.

A symbolic constant is a constant represented by a name, just like a variable. The const keyword precedes the type, name, and initialization. Here's a statement that sets the point reward for killing a zombie:

```
const int KILL_BONUS = 5000;
```

C++ Keywords

A keyword is a reserved word. You cannot use it as a variable name, constant name etc. A list of 32 Keywords in C++ Language which are also available in C language are given below.

auto	break	case	char	const	continue	default	do
double	else	enum	extern	float	for	goto	if
int	long	register	return	short	signed	sizeof	static
struct	switch	typedef	union	unsigned	void	volatile	while

A list of 30 Keywords in C++ Language which are not available in C language are given below.

asm	dynamic_cast	namespace	reinterpret_cast	bool
explicit	new	static_cast	false	catch
operator	template	friend	private	class
this	inline	public	throw	const_cast
delete	mutable	protected	true	try
typeid	typename	using	virtual	wchar_t

C++ Variable

A variable is a name of memory location. It is used to store data. Its value can be changed and it can be reused many times.

It is a way to represent memory location through symbol so that it can be easily identified.

Let's see the syntax to declare a variable:

```
type variable_list;
```

The example of declaring variable is given below:

```
int x;
float y;
char z;
```

Here, x, y, z are variables and int, float, char are data types.

We can also provide values while declaring the variables as given below:

```
int x=5,b=10;  //declaring 2 variable of integer type
float f=30.8;
char c='A';
```

Rules for defining variables

✓A variable can have alphabets, digits and underscore.

✓A variable name can start with alphabet and underscore only. It can't start with digit.

✓No white space is allowed within variable name.

✓A variable name must not be any reserved word or keyword e.g. char, float etc.

Valid variable names:

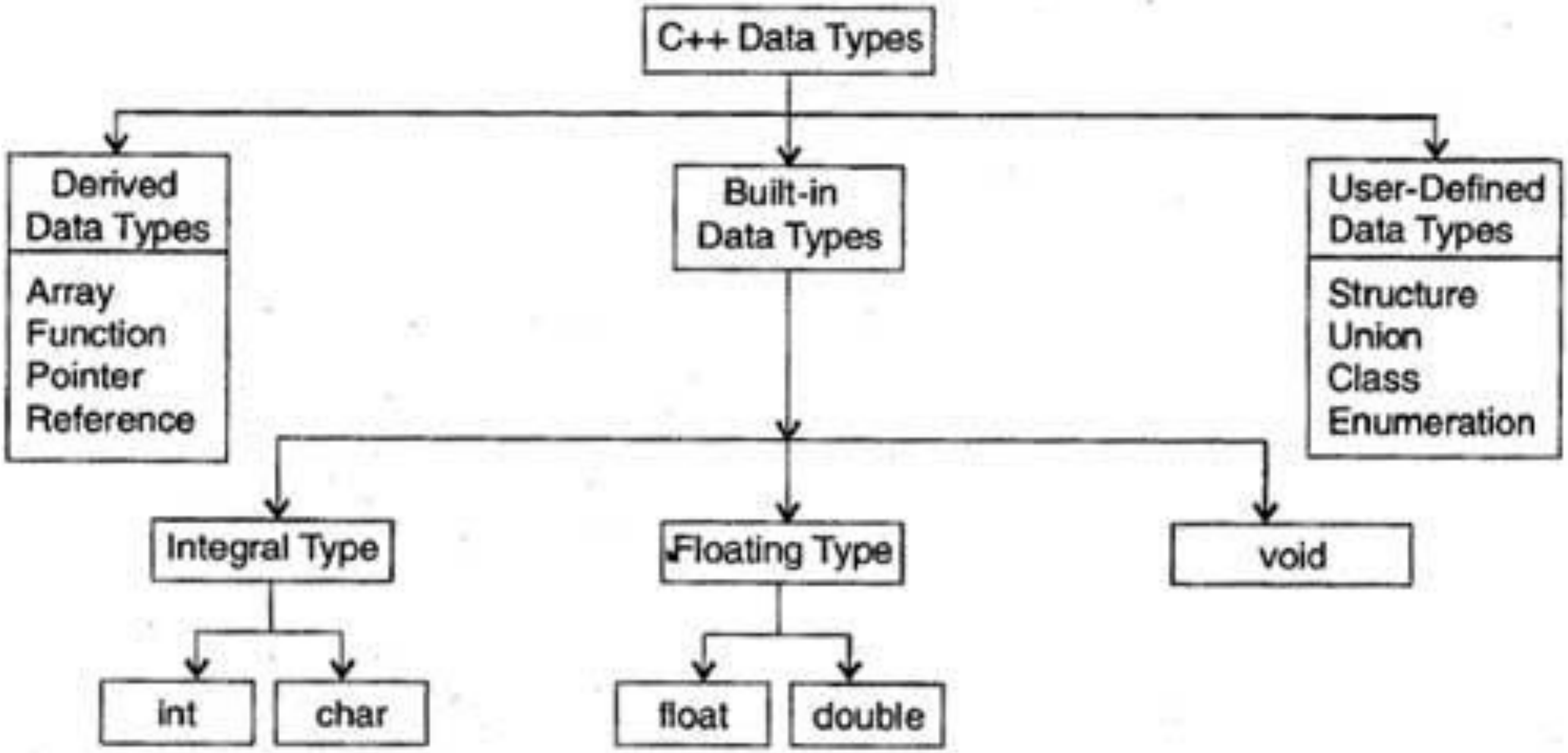
```
int a;
int _ab;
int a30;
```

Invalid variable names:

```
int 4;
int x y;
int double;
```

C++ Data Types

A data type specifies the type of data that a variable can store such as integer, floating, character etc.

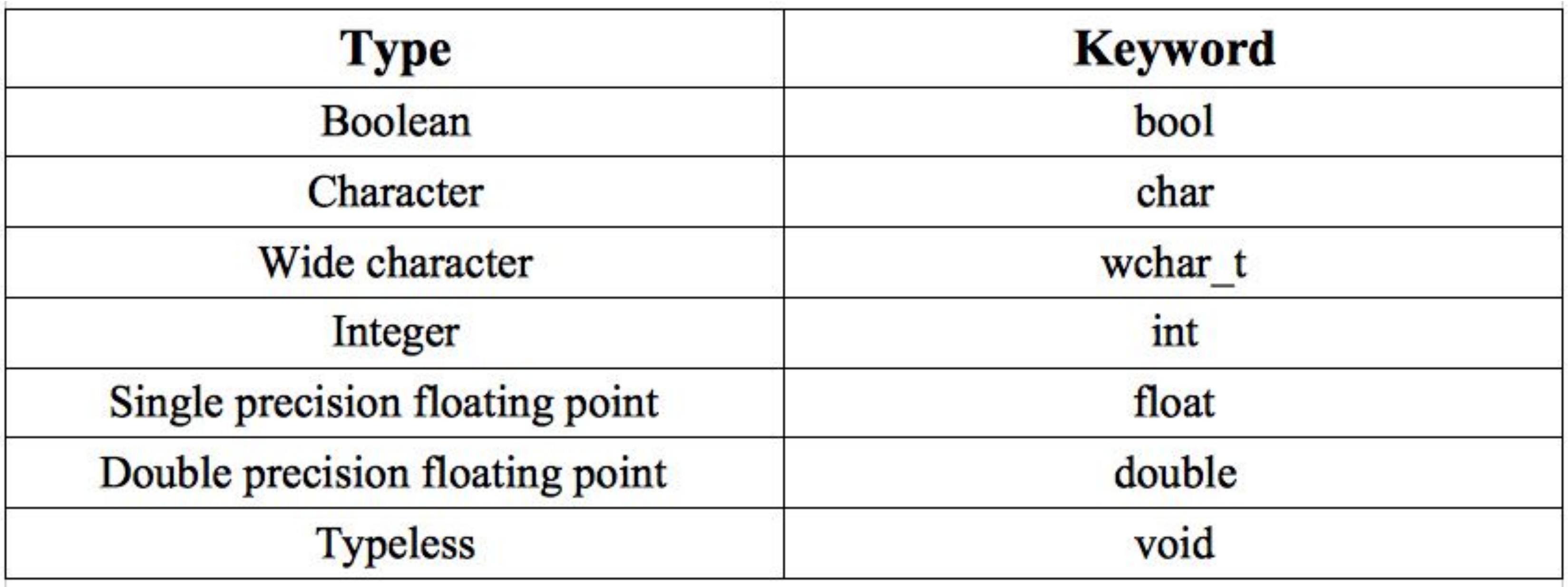


Various Data Types in C++

Basic Data Types

The basic data types are integer-based and floating-point based. C++ language supports both signed and unsigned literals.

The memory size of basic data types may change according to 32 or 64 bit operating system.



Type	Keyword
Boolean	bool
Character	char
Wide character	wchar_t
Integer	int
Single precision floating point	float
Double precision floating point	double
Typeless	void

Category	Type	Contents
Integral		
	char	Type char is an integral type that usually contains members of the basic execution character set — By default, this is ASCII in Microsoft C++. The C++ compiler treats variables of type char, signed char, and unsigned char as having different types. Variables of type char are promoted to int as if they are type signed char by default, unless the /J compilation option is used. In this case they are treated as type unsigned char and are promoted to int without sign extension.
	bool	Type bool is an integral type that can have one of the two values true or false. Its size is unspecified.
	short	Type short int (or simply short) is an integral type that is larger than or equal to the size of type char, and shorter than or equal to the size of type int. Objects of type short can be declared as signed short or unsigned short. Signed short is a synonym for short.
	int	Type int is an integral type that is larger than or equal to the size of type short int, and shorter than or equal to the size of type long. Objects of type int can be declared as signed int or unsigned int. Signed int is a synonym for int.

long Type long (or long int) is an integral type that is larger than or equal to the size of type int.

Objects of type long can be declared as signed long or unsigned long. Signed long is a synonym for long.

long long Larger than an unsigned long.

Objects of type long long can be declared as signed long long or unsigned long long. signed long long is a synonym for long long.

Floating point

float Type float is the smallest floating point type.

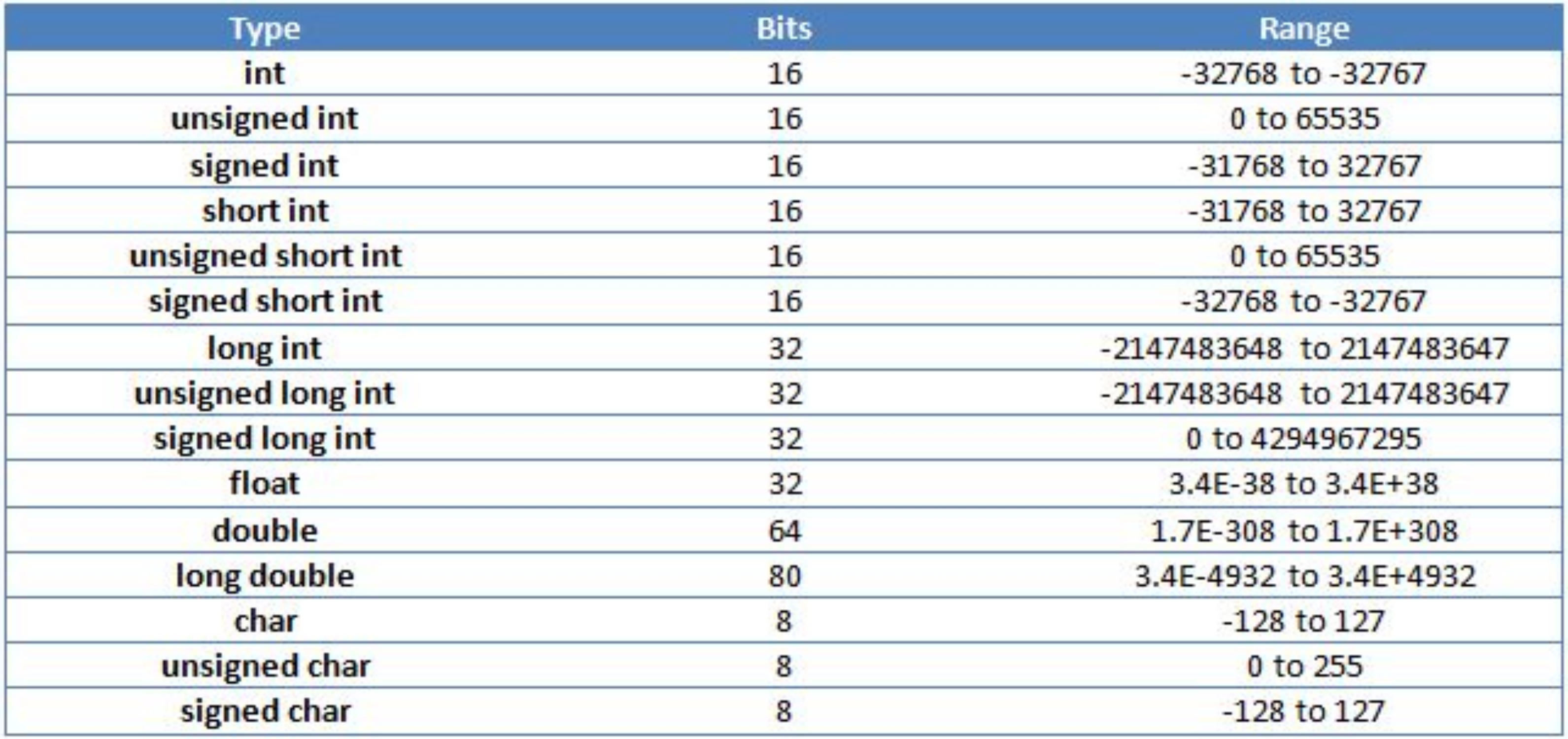
double Type double is a floating point type that is larger than or equal to type float, but shorter than or equal to the size of type long double.

long double Type long double is a floating point type that is larger than or equal to type double.

Sizes of Fundamental Types

The memory size of basic data types may change according to 32 or 64 bit operating system.

Type	Size
•bool, char, unsigned char, signed char,	1 byte
•short, unsigned short,int,unsigned int	2 bytes
•float, long int, unsigned long int	4 bytes
•double, long double, long long	8 bytes



Type	Bits	Range
int	16	-32768 to -32767
unsigned int	16	0 to 65535
signed int	16	-31768 to 32767
short int	16	-31768 to 32767
unsigned short int	16	0 to 65535
signed short int	16	-32768 to -32767
long int	32	-2147483648 to 2147483647
unsigned long int	32	-2147483648 to 2147483647
signed long int	32	0 to 4294967295
float	32	3.4E-38 to 3.4E+38
double	64	1.7E-308 to 1.7E+308
long double	80	3.4E-4932 to 3.4E+4932
char	8	-128 to 127
unsigned char	8	0 to 255
signed char	8	-128 to 127

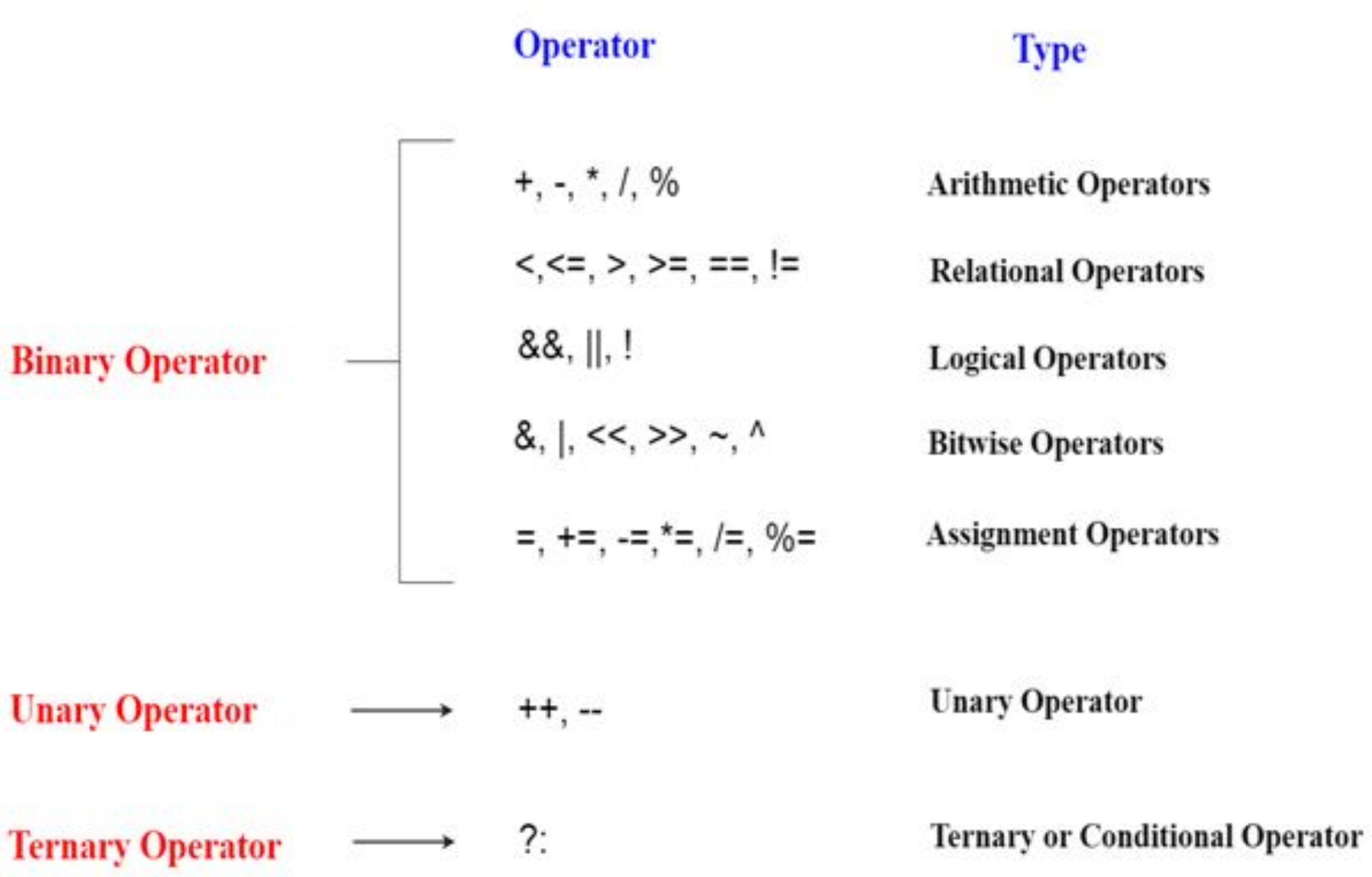
C++ Operators

An operator is simply a symbol that is used to perform operations. There can be many types of operations like arithmetic, logical, bitwise etc.

There are following types of operators to perform different types of operations in C++ language.

- Arithmetic Operators
- Relational Operators
- Logical Operators
- Bitwise Operators
- Assignment Operator
- Unary operator
- Ternary or Conditional Operator
- Misc Operator

operators



<table>
<tr><th></th><th>Operator</th><th>Type</th></tr>
<tr><td>Binary Operator</td><td>+, -, *, /, %</td><td>Arithmetic Operators</td></tr>
<tr><td></td><td><,<=, >, >=, ==, !=</td><td>Relational Operators</td></tr>
<tr><td></td><td>&&, ||, !</td><td>Logical Operators</td></tr>
<tr><td></td><td>&, |, <<, >>, ~, ^</td><td>Bitwise Operators</td></tr>
<tr><td></td><td>=, +=, -=,*=, /=, %=</td><td>Assignment Operators</td></tr>
<tr><td>Unary Operator</td><td>++, --</td><td>Unary Operator</td></tr>
<tr><td>Ternary Operator</td><td>?:</td><td>Ternary or Conditional Operator</td></tr>
</table>

Precedence of Operators in C++

The precedence of operator species that which operator will be evaluated first and next. The associativity specifies the operators direction to be evaluated, it may be left to right or right to left.

Let's understand the precedence by the example given below:

```
    int data=5+10*10;
```

The "data" variable will contain 105 because * (multiplicative operator) is evaluated before + (additive operator).

The precedence and associativity of C++ operators is given below:

Category	Operator	Associativity
Postfix	() [] -> . ++ - -	Left to right
Unary	+ - ! ~ ++ - - (type)* & sizeof	Right to left
Multiplicative	* / %	Left to right
Additive	+ -	Right to left
Shift	<< >>	Left to right
Relational	< <= > >=	Left to right
Equality	== !=/td>	Right to left
Bitwise AND	&	Left to right
Bitwise XOR	^	Left to right

The precedence and associativity of C++ operators :

Bitwise OR	**|**	**Right to left**
Logical AND	&&	Left to right
Logical OR	||	Left to right
Conditional	?:	Right to left
Assignment	= += -= *= /= %=>>= <<= &= ^= |=	Right to left
Comma	,	Left to right

control structure (branching & looping)

C++ if-else

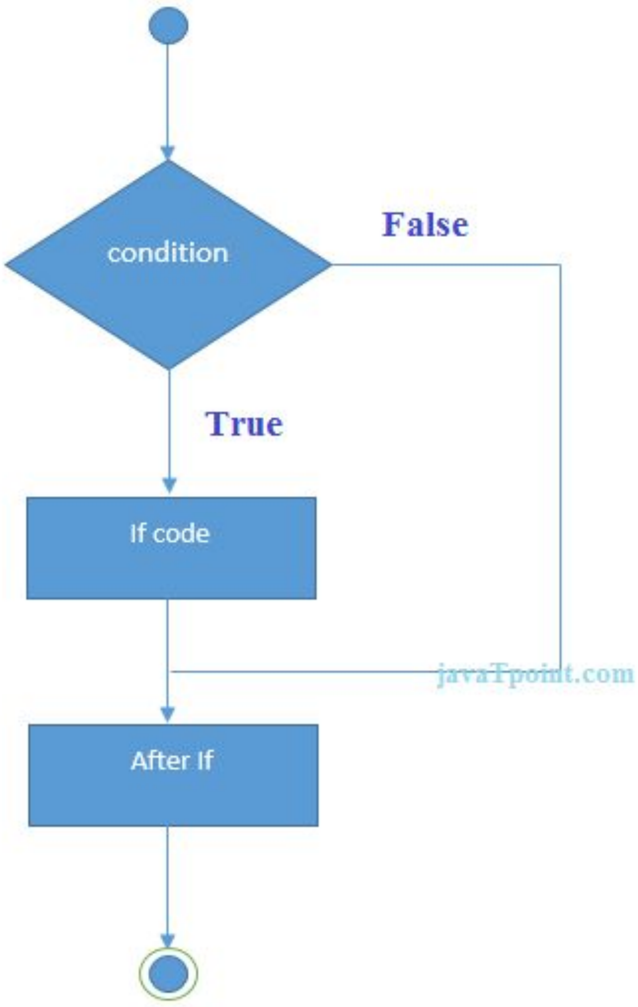
In C++ programming, if statement is used to test the condition. There are various types of if statements in C++.

- if statement
- if-else statement
- nested if statement
- if-else-if ladder

C++ IF Statement

The C++ if statement tests the condition. It is executed if condition is true.

```
if(condition){
//code to be executed
}
```



C++ If Example

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

int main () {
  int num = 10;
        if (num % 2 == 0)
        {
           cout<<"It is even number";
        }
  return 0;
}
```

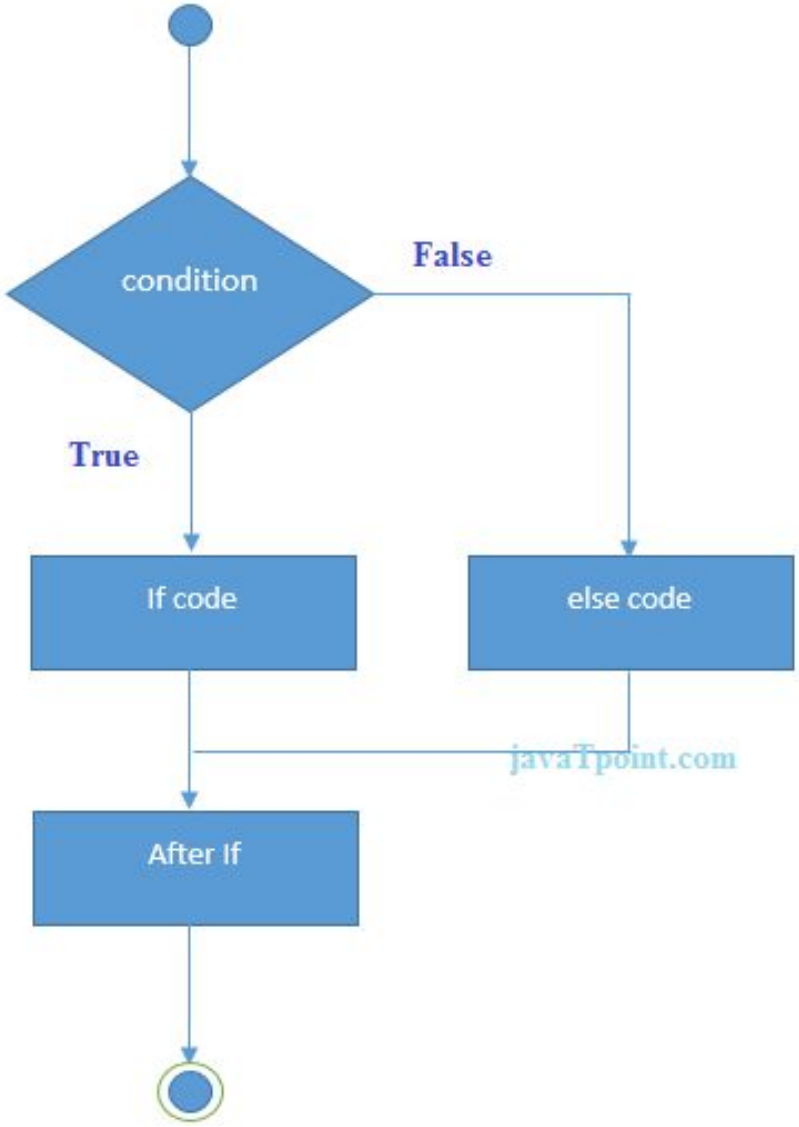
Output:/p>

```
It is even number
```

C++ IF-else Statement

The C++ if-else statement also tests the condition. It executes if block if condition is true otherwise else block is executed.

```
if(condition){
//code if condition is true
}else{
//code if condition is false
}
```



C++ If-else Example

```
#include <iostream>
using namespace std;
int main () {
   int num = 11;
            if (num % 2 == 0)
            {
                cout<<"It is even number";
            }
            else
            {
                cout<<"It is odd number";
            }
   return 0;
}
```

Output:

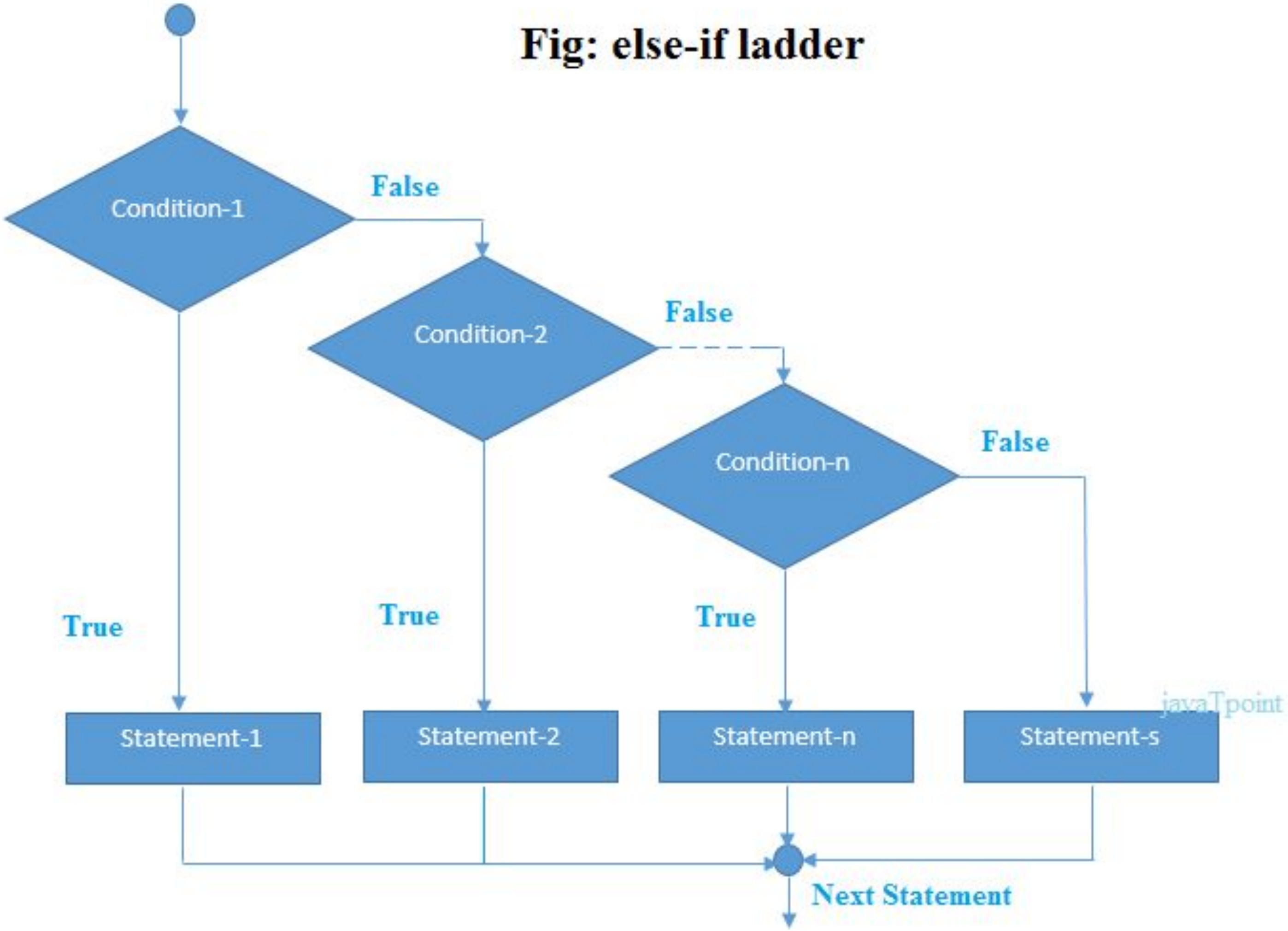
```
It is odd number
```

C++ IF-else-if ladder Statement

The C++ if-else-if ladder statement executes one condition from multiple statements.

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
...
else{
//code to be executed if all the conditions are false
}
```

IF-else-if ladder Statement



C++ If else-if Example

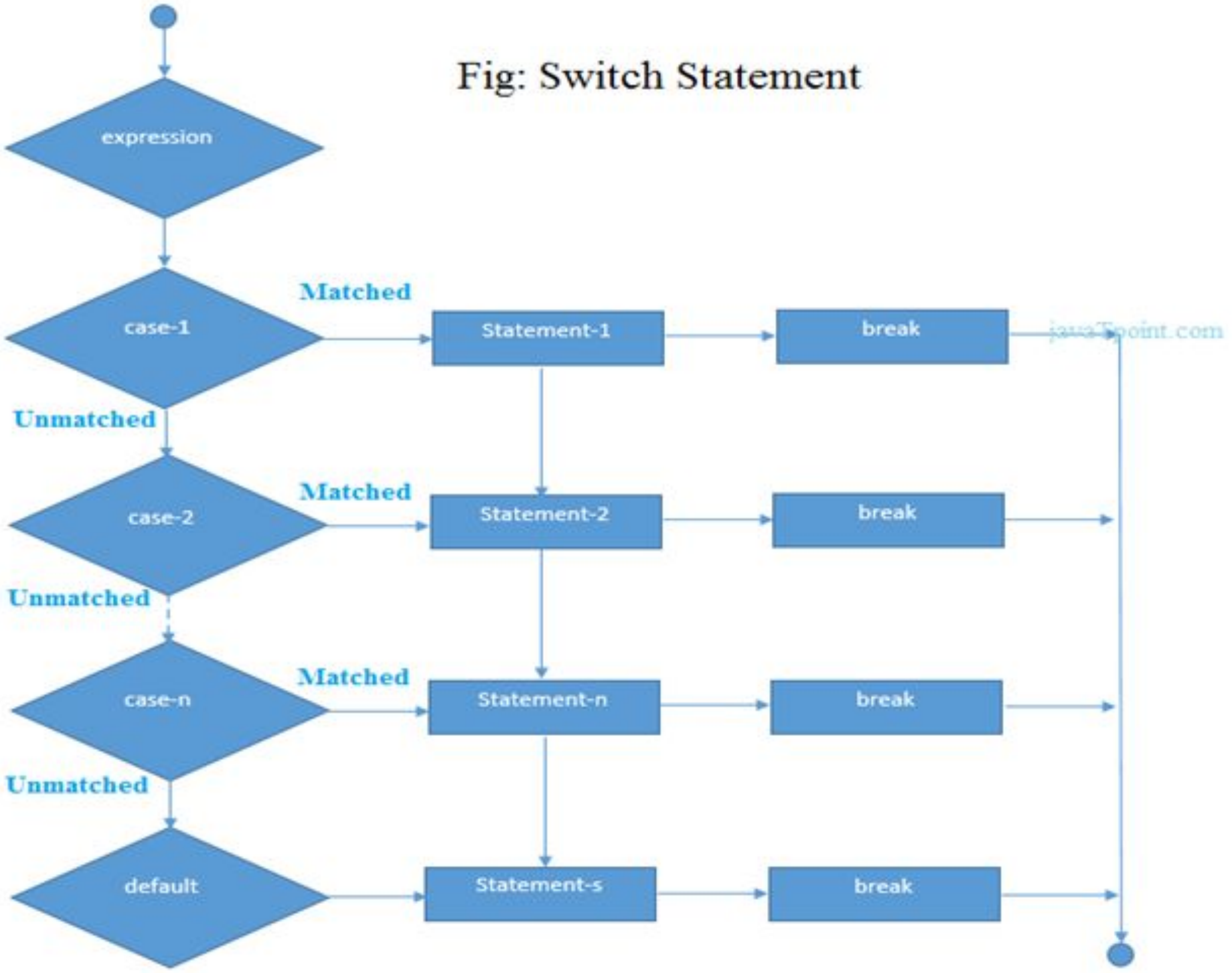
```
#include <iostream>
using namespace std;
int main () {
    int num;
    cout<<"Enter a number to check grade:";
    cin>>num;
       if (num <0 || num >100)
       {
         cout<<"wrong number";
       }
       else if(num >= 0 && num < 50){
         cout<<"Fail";
       }
       else if (num >= 50 && num < 60)
       {
         cout<<"D Grade";
       }
   else if (num >= 60 && num < 70)
       {
         cout<<"C Grade";
       }
       else if (num >= 70 && num < 80)
       {
         cout<<"B Grade";
       }
       else if (num >= 80 && num < 90)
       {
         cout<<"A Grade";
       }
       else if (num >= 90 && num <= 100)
       {
         cout<<"A+ Grade";
       }   Output:
}
Enter a number to check grade:66
C Grade
Output:Enter a number to check grade:-2
wrong number
```

C++ switch

The C++ switch statement executes one statement from multiple conditions. It is like if-else-if ladder statement in C++.

```
switch(expression){
case value1:
 //code to be executed;
 break;
case value2:
 //code to be executed;
 break;
......

default:
 //code to be executed if all cases are not matched;
 break;
}
```

switch statement flow chart



C++ Switch Example

```
#include <iostream>
using namespace std;
int main () {
    int num;
    cout<<"Enter a number to check grade:";
    cin>>num;
      switch (num)
     {
        case 10: cout<<"It is 10"; break;
        case 20: cout<<"It is 20"; break;
        case 30: cout<<"It is 30"; break;
        default: cout<<"Not 10, 20 or 30"; break;
     }
  }
```

Output:

```
Enter a number:
10
It is 10
```

Output:

```
Enter a number:
55
Not 10, 20 or 30
```

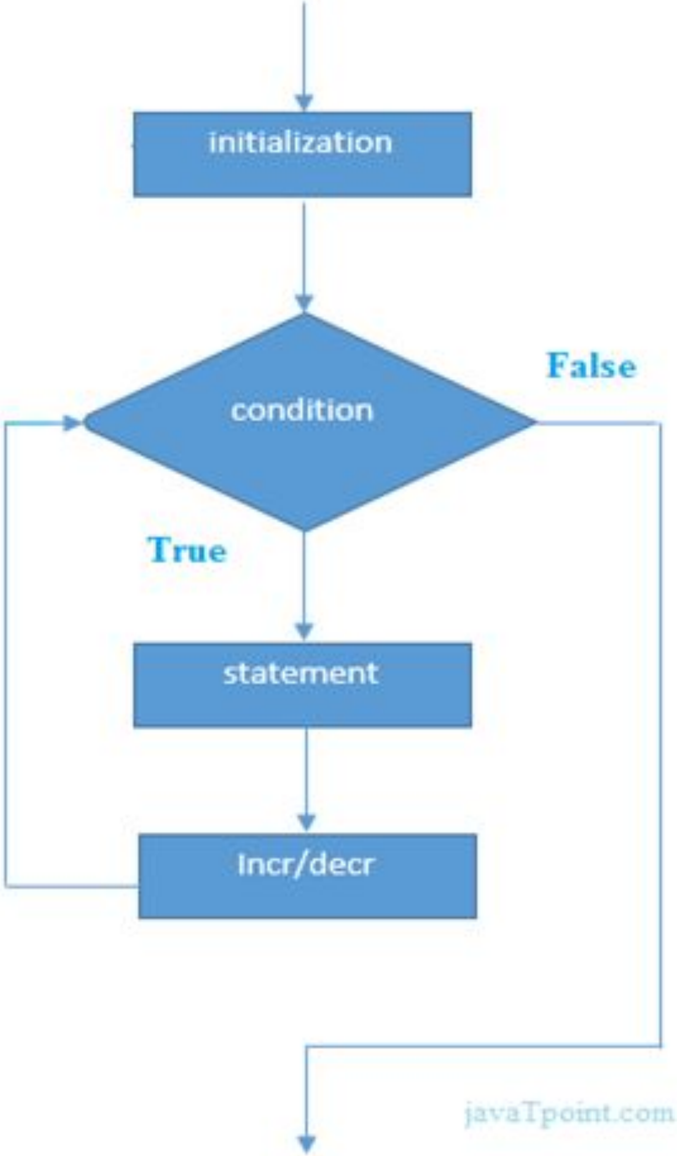
C++ For Loop

The C++ for loop is used to iterate a part of the program several times. If the number of iteration is fixed, it is recommended to use for loop than while or do-while loops.

The C++ for loop is same as C/C#. We can initialize variable, check condition and increment/decrement value.

```
for(initialization; condition; incr/decr){
//code to be executed
}
```

For Loop flowchart



C++ For Loop Example

```
#include <iostream>
using namespace std;
int main() {
         for(int i=1;i<=10;i++){
            cout<<i <<"\n";
          }
    }
```

Output:

```
1
2
3
4
5
6
7
8
9
10
```

C++ Nested For Loop

In C++, we can use for loop inside another for loop, it is known as nested for loop. The inner loop is executed fully when outer loop is executed one time. So if outer loop and inner loop are executed 4 times, inner loop will be executed 4 times for each outer loop i.e. total 16 times.

C++ Nested For Loop Example

Let's see a simple example of nested for loop in C++.

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

int main () {
      for(int i=1;i<=3;i++){
          for(int j=1;j<=3;j++){
         cout<<i<<" "<<j<<"\n";
        }
      }
   }
```

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

C++ Infinite For Loop

If we use double semicolon in for loop, it will be executed infinite times. Let's see a simple example of infinite for loop in C++.

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

int main () {
        for (; ;)
          {
                  cout<<"Infinitive For Loop";
          }
    }
```

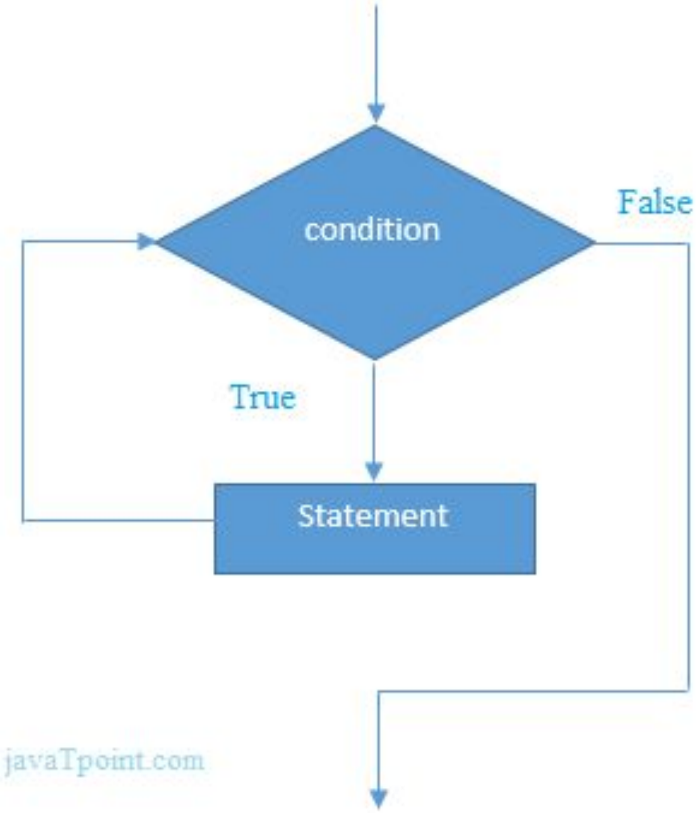
Output:

```
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
Infinitive For Loop
ctrl+c
```

C++ While loop

In C++, while loop is used to iterate a part of the program several times. If the number of iteration is not fixed, it is recommended to use while loop than for loop.

```
while(condition){
//code to be executed
}
```



C++ While Loop Example

Let's see a simple example of while loop to print table of 1.

```
#include <iostream>
using namespace std;
int main() {
 int i=1;
        while(i<=10)
       {
           cout<<i <<"\n";
           i++;
         }
    }
```

Output:

```
1
2
3
4
.
.
10
```

C++ Nested While Loop Example

In C++, we can use while loop inside another while loop, it is known as nested while loop. The nested while loop is executed fully when outer loop is executed once.

Let's see a simple example of nested while loop in C++ programming language.

```
#include <iostream>
using namespace std;
int main () {
        int i=1;
          while(i<=3)
          {
              int j = 1;
              while (j <= 3)
{
            cout<<i<<" "<<j<<"\n";
            j++;
          }
           i++;
        }      }
```

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

C++ Infinitive While Loop Example:

We can also create infinite while loop by passing true as the test condition.

```
#include <iostream>
using namespace std;
int main () {
        while(true)
          {
                    cout<<"Infinitive While Loop";
          }
    }
```

Output:

```
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
Infinitive While Loop
ctrl+c
```

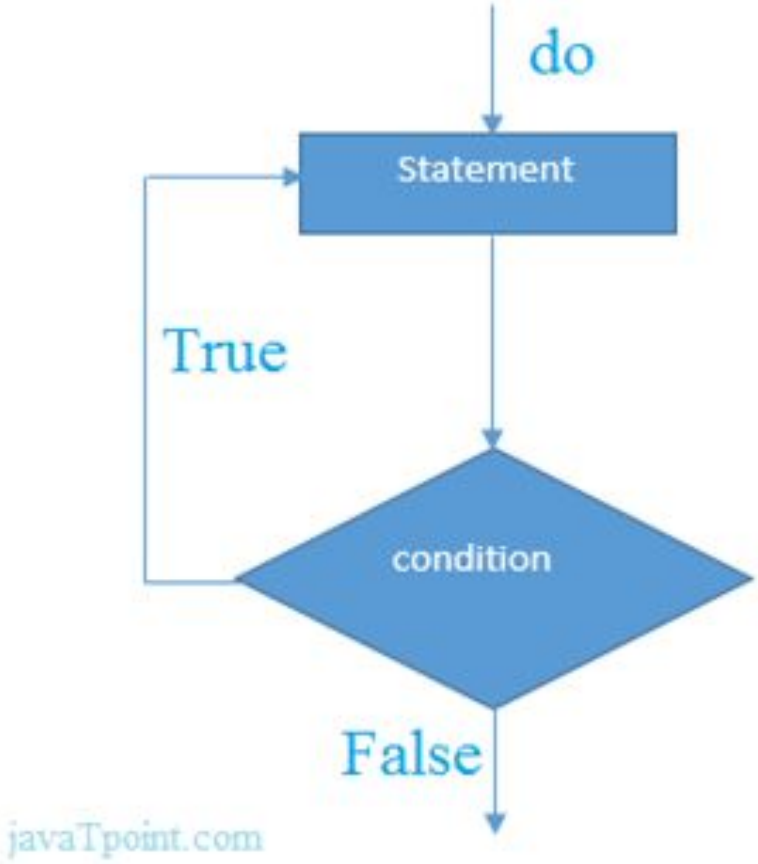
C++ Do-While Loop

The C++ do-while loop is used to iterate a part of the program several times. If the number of iteration is not fixed and you must have to execute the loop at least once, it is recommended to use do-while loop.

The C++ do-while loop is executed at least once because condition is checked after loop body.

```
do{
//code to be executed
}while(condition);
```

Do-While Loop flow chart



C++ do-while Loop Example

Let's see a simple example of C++ do-while loop to print the table of 1.

```
#include <iostream>
using namespace std;
int main() {
     int i = 1;
          do{
              cout<<i<<"\n";
              i++;
          } while (i <= 10) ;
}
```

Output:

```
1
2
3
4
.....
10
```

C++ Nested do-while Loop

In C++, if you use do-while loop inside another do-while loop, it is known as nested do-while loop. The nested do-while loop is executed fully for each outer do-while loop.

Let's see a simple example of nested do-while loop in C++.

```
#include <iostream>
using namespace std;
int main() {
    int i = 1;
       do{
           int j = 1;
           do{
             cout<<i<<"\n";
               j++;
           } while (j <= 3) ;
           i++;
        } while (i <= 3) ;
}
```

Output:

```
1 1
1 2
1 3
2 1
2 2
2 3
3 1
3 2
3 3
```

C++ Infinitive do-while Loop

In C++, if you pass true in the do-while loop, it will be infinitive do-while loop.

```
do{
//code to be executed
}while(true);
```

C++ Infinitive do-while Loop Example

```
#include <iostream>
using namespace std;
int main() {
      do{
              cout<<"Infinitive do-while Loop";
          } while(true);
}
```

Output:

```
Infinitive do-while Loop
Infinitive do-while Loop
Infinitive do-while Loop
Infinitive do-while Loop
Infinitive do-while Loop
ctrl+c
```

C++ Break Statement

The C++ break is used to break loop or switch statement. It breaks the current flow of the program at the given condition. In case of inner loop, it breaks only inner loop.

jump-statement;

break;

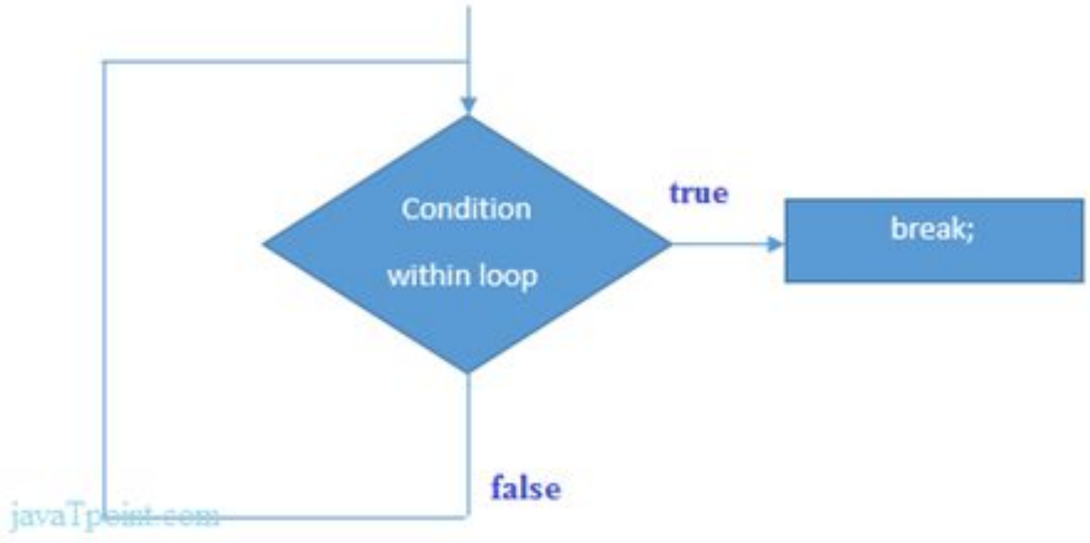


Figure: Flowchart of break statement

C++ Break Statement Example

Let's see a simple example of C++ break statement which is used inside the loop.

```
#include <iostream>
using namespace std;
int main() {
      for (int i = 1; i <= 10; i++)
          {
              if (i == 5)
              {
                  break;
              }
        cout<<i<<"\n";
          }
}
```

Output:

```
1
2
3
4
```

C++ Break Statement with Inner Loop

The C++ break statement breaks inner loop only if you use break statement inside the inner loop.

Let's see the example code:

```
#include <iostream>
using namespace std;
int main()
{
    for(int i=1;i<=3;i++){
            for(int j=1;j<=3;j++){
                if(i==2&&j==2){
                    break;
                        }
                    cout<<i<<" "<<j<<"\n";
                    }
          }
}
```

Output:

```
1 1
1 2
1 3
2 1
3 1
3 2
3 3
```

C++ Continue Statement

The C++ continue statement is used to continue loop. It continues the current flow of the program and skips the remaining code at specified condition. In case of inner loop, it continues only inner loop.

```
jump-statement;
continue;
```

C++ Continue Statement Example

```
#include <iostream>
using namespace std;
int main()
{
    for(int i=1;i<=10;i++){
        if(i==5){
          continue;
        }
        cout<<i<<"\n";
      }
}
```

Output:

```
1
2
3
4
6
7
8
9
10
```

C++ Continue Statement with Inner Loop

C++ Continue Statement continues inner loop only if you use continue statement inside the inner loop.

```
#include <iostream>
using namespace std;
int main()
{
 for(int i=1;i<=3;i++){
            for(int j=1;j<=3;j++){
             if(i==2&&j==2){
                continue;
                        }
                cout<<i<<" "<<j<<"\n";
                    }
            }
}
```

Output:

```
1 1
1 2
1 3
2 1
2 3
3 1
3 2
3 3
```

C++ Goto Statement

The C++ goto statement is also known as jump statement. It is used to transfer control to the other part of the program. It unconditionally jumps to the specified label.

It can be used to transfer control from deeply nested loop or switch case label.

Goto Statement Example

Let's see the simple example of goto statement in C++.

```
#include <iostream>
using namespace std;
int main()
{
ineligible:
      cout<<"You are not eligible to vote!\n";
    cout<<"Enter your age:\n";
    int age;
    cin>>age;
    if (age < 18){
        goto ineligible;
    }
    else
    {
        cout<<"You are eligible to vote!";
    }
}
```

Output:

```
You are not eligible to vote!
Enter your age:
16
You are not eligible to vote!
Enter your age:
7
You are not eligible to vote!
Enter your age:
22
You are eligible to vote!
```

C++ Comments

The C++ comments are statements that are not executed by the compiler. The comments in C++ programming can be used to provide explanation of the code, variable, method or class. By the help of comments, you can hide the program code also.

There are two types of comments in C++.

Single Line comment

Multi Line comment

C++ Single Line Comment

The single line comment starts with // (double slash). Let's see an example of single line comment in C++.

```
#include <iostream>
using namespace std;
int main()
{
 int x = 11; // x is a variable
 cout<<x<<"\n";
}
```

Output:

```
11
```

C++ Multi Line Comment

The C++ multi line comment is used to comment multiple lines of code. It is surrounded by slash and asterisk (/∗ ..... ∗/). Let's see an example of multi line comment in C++.

```
#include <ostream>
using namespace std;
int main()
{
/* declare and
print variable in C++. */
 int x = 35;
 cout<<x<<"\n";
}
```

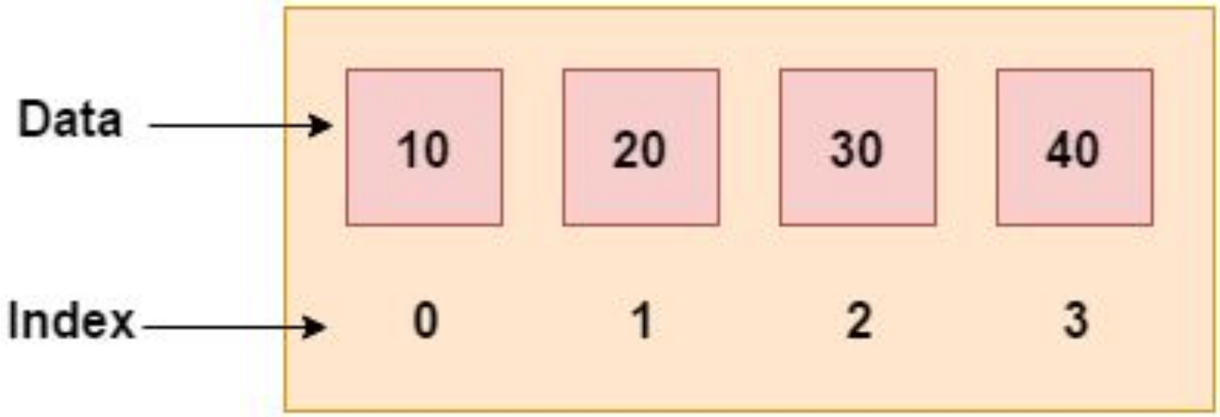
Output:

```
35
```

C++ Arrays

Like other programming languages, array in C++ is a group of similar types of elements that have contiguous memory location.

In C++ std::array is a container that encapsulates fixed size arrays. In C++, array index starts from 0. We can store only fixed set of elements in C++ array.



Advantages of C++ Array

Code Optimization (less code)

Random Access

Easy to traverse data

Easy to manipulate data

Easy to sort data etc.

Disadvantages of C++ Array

Fixed size

C++ Array Types

There are 2 types of arrays in C++ programming:

Single Dimensional Array

Multidimensional Array

C++ Single Dimensional Array

Let's see a simple example of C++ array, where we are going to create, initialize and traverse array.

```
#include <iostream>
using namespace std;
int main()
{
 int arr[5]={10, 0, 20, 0, 30};  //creating and initializing array
        //traversing array
        for (int i = 0; i < 5; i++)
        {
            cout<<arr[i]<<"\n";
        }
}
```

```
Output:/p>10
0
20
0
30
```

C++ Array Example: **Traversal using foreach loop**

We can also traverse the array elements using foreach loop. It returns array element one by one.

```
#include <iostream>
using namespace std;
int main()
{
 int arr[5]={10, 0, 20, 0, 30}; //creating and initializing array
        //traversing array
       for (int i: arr)
        {
            cout<<i<<"\n";
        }
}
```

```
Output:10
0
20
0
30
```

C++ Passing Array to Function

In C++, to reuse the array logic, we can create function. To pass array to function in C++, we need to provide only array name.

```
functionname(arrayname); //passing array to function
```

C++ Passing Array to Function Example: print array elements

Let's see an example of C++ function which prints the array elements.

```
#include <iostream>
using namespace std;
void printArray(int arr[5]);
int main()
{
    int arr1[5] = { 10, 20, 30, 40, 50 };
    int arr2[5] = { 5, 15, 25, 35, 45 };
    printArray(arr1); //passing array to function
    printArray(arr2);
}
void printArray(int arr[5])
{
  cout << "Printing array elements:"<< endl;
  for (int i = 0; i < 5; i++)
  {
          cout<<arr[i]<<"\n";
  }
}
```

Output:

```
Printing array elements:
10
20
30
40
50
Printing array elements:
5
15
25
35
45
```

C++ Passing Array to Function

Example: **Print minimum number**

Let's see an example of C++ array which prints minimum number in an array using function.

```
#include <iostream>
using namespace std;
void  printMin(int arr[5]);
int main()
{
   int arr1[5] = { 30, 10, 20, 40, 50 };
        int arr2[5] = { 5, 15, 25, 35, 45 };
        printMin(arr1);//passing array to function
         printMin(arr2);
}
```

```
void  printMin(int arr[5])
{
    int min = arr[0];
        for (int i = 0; i > 5; i++)
        {
            if (min > arr[i])
            {
                min = arr[i];
            }
        }
        cout<< "Minimum element is: "<< min <<"\n";
}
```

Output:

```
Minimum element is: 10
Minimum element is: 5
```

C++ Passing Array to Function Example: **Print maximum number**

Let's see an example of C++ array which prints maximum number in an array using function.

```
#include <iostream>
using namespace std;
void  printMax(int arr[5]);
int main()
{
        int arr1[5] = { 25, 10, 54, 15, 40 };
        int arr2[5] = { 12, 23, 44, 67, 54 };
        printMax(arr1); //Passing array to function
         printMax(arr2);
}
```

```
void  printMax(int arr[5])
{
    int max = arr[0];
        for (int i = 0; i < 5; i++)
        {
            if (max < arr[i])
            {
                max = arr[i];
            }
        }
        cout<< "Maximum element is: "<< max <<"\n";
}
```

Output:

```
Maximum element is: 54
Maximum element is: 67
```

C++ Multidimensional Arrays

The multidimensional array is also known as rectangular arrays in C++. It can be two dimensional or three dimensional. The data is stored in tabular form (row ∗ column) which is also known as matrix.

C++ Multidimensional Array Example

Let's see a simple example of multidimensional array in C++ which declares, initializes and traverse two dimensional arrays.

```
#include <iostream>
using namespace std;
int main()
{
  int test[3][3];  //declaration of 2D array
    test[0][0]=5;  //initialization
    test[0][1]=10;
    test[1][1]=15;
    test[1][2]=20;
    test[2][0]=30;
    test[2][2]=10;
    //traversal
```

```
 for(int i = 0; i < 3; ++i)
    {
        for(int j = 0; j < 3; ++j)
        {
            cout<< test[i][j]<<" ";
        }
        cout<<"\n"; //new line at each row
    }
    return 0;
}
```

Output:

```
5 10 0
0 15 20
30 0 10
```

C++ Multidimensional Array Example: Declaration and initialization at same time

Let's see a simple example of multidimensional array which initializes array at the time of declaration.

```
#include <iostream>
using namespace std;
int main()
{
  int test[3][3] =
    {
        {2, 5, 5},
        {4, 0, 3},
        {9, 1, 8}  };  //declaration and initialization
  //traversal
    for(int i = 0; i < 3; ++i)
    {
        for(int j = 0; j < 3; ++j)
        {
            cout<< test[i][j]<<" ";
        }
        cout<<"\n"; //new line at each row
    }
    return 0;
}
```

Output:"

```
2 5 5
4 0 3
9 1 8
```

std::string class in C++

C++ has in its definition a way to represent sequence of characters as an object of class. This class is called std:: string. String class stores the characters as a sequence of bytes with a functionality of allowing access to single byte character.

std:: string vs Character Array

A character array is simply an array of characters can terminated by a null character. A string is a class which defines objects that be represented as stream of characters.

Size of the character array has to allocated statically, more memory cannot be allocated at run time if required. Unused allocated memory is wasted in case of character array. In case of strings, memory is allocated dynamically. More memory can be allocated at run time on demand. As no memory is preallocated, no memory is wasted.

Implementation of character array is faster than std:: string. Strings are slower when compared to implementation than character array.

Character array do not offer much inbuilt functions to manipulate strings. String class defines a number of functionalities which allow manifold operations on strings.

Operations on strings

Some Functions in String Class

1. getline() :- This function is used to store a stream of characters as entered by the user in the object memory.

2. push_back() :- This function is used to input a character at the end of the string.

3. pop_back() :- Introduced from C++11(for strings), this function is used to delete the last character from the string.

4. capacity() :- This function returns the capacity allocated to the string, which can be equal to or more than the size of the string. Additional space is allocated so that when the new characters are added to the string, the operations can be done efficiently.

5. resize() :- This function changes the size of string, the size can be increased or decreased.

6.shrink_to_fit() :- This function decreases the capacity of the string and makes it equal to its size. This operation is useful to save additional memory if we are sure that no further addition of characters have to be made.

Manipulating Functions

7. copy("char array", len, pos) :- This function copies the substring in target character array mentioned in its arguments. It takes 3 arguments, target char array, length to be copied and starting position in string to start copying.

8. swap() :- This function swaps one string with other.

```
#include<iostream>
#include<string> // for string class
using namespace std;
int main()
{
    // Declaring string
    string str;

    // Taking string input using getline()
    // "geeksforgeek" in given output
    getline(cin,str);

    // Displaying string
    cout << "The initial string is : ";
    cout << str << endl;

    // Using push_back() to insert a character
    // at end
    // pushes 's' in this case
    str.push_back('s');
// Displaying string
    cout << "The string after push_back operation is : ";
    cout << str << endl;

    // Using pop_back() to delete a character
    // from end
    // pops 's' in this case
    str.pop_back();

    // Displaying string
    cout << "The string after pop_back operation is : ";
    cout << str << endl;

    return 0;

}
```

Input:

```
geeksforgeek
```

Output:

```
The initial string is : geeksforgeek
The string after push_back operation is : geeksforgeeks
The string after pop_back operation is : geeksforgeek
```