

# Introduction to C++ Programming

By

Dr. Jyoti Ranjan Nayak

Asst. Prof.

Institute of Technical Education and Research

(Siksha 'O' Anusandhan University)

# Tokens in C++

- A token in C++ can be defined as the smallest individual element of the C++ programming language that is meaningful to the compiler.
- Therefore, we can say that tokens in C++ is the building block or the basic component for creating a program in C++ language.
- The tokens of C++ language can be classified into six types
  1. Keywords
  2. Identifiers
  3. Constants
  4. Strings
  5. Special Symbols
  6. Operators

# 1. Keywords

- The keywords are pre-defined or reserved words in a programming language.
- Each keyword is meant to perform a specific function in a program.
- Since keywords are the pre-defined words used by the compiler, so they cannot be used as the variable names.
- As C++ is a case sensitive language, all keywords must be written in lowercase.
- If the keywords are used as the variable names, then error messages will come

C++ language supports **36** keywords which are given below:

asm	double	new	switch
auto	else	operator	template
break	enum	private	this
case	extern	protected	throw
catch	float	public	try
char	for	register	typedef
class	friend	return	union
const	goto	short	unsigned
continue	if	signed	virtual
default	inline	sizeof	void
delete	int	static	volatile
do	long	struct	while

## 2. Identifiers

- Identifiers in C++ are used for naming variables, functions, arrays, structures, etc. Identifiers in C++ are the user-defined words.
- Rules for constructing identifiers in C++ are given below:
  - i. The first character of an identifier should be either an alphabet or an underscore or \$, and then it can be followed by any of the character, digit, or underscore.
  - ii. It should not begin with any numerical digit.
  - iii. In identifiers, both uppercase and lowercase letters are distinct. Therefore, we can say that identifiers are case sensitive.
  - iv. Commas or blank spaces cannot be specified within an identifier.
  - v. Keywords cannot be represented as an identifier.
  - vi. The length of the identifiers should not be more than 2048 characters.
  - vii. Identifiers should be written in such a way that it is meaningful, short, and easy to read.

### 3. Constants

- A constant is a value assigned to the variable which will remain the same throughout the program, i.e., the constant value cannot be changed.
- There are two ways of declaring constant:
- Syntax to Define Constant

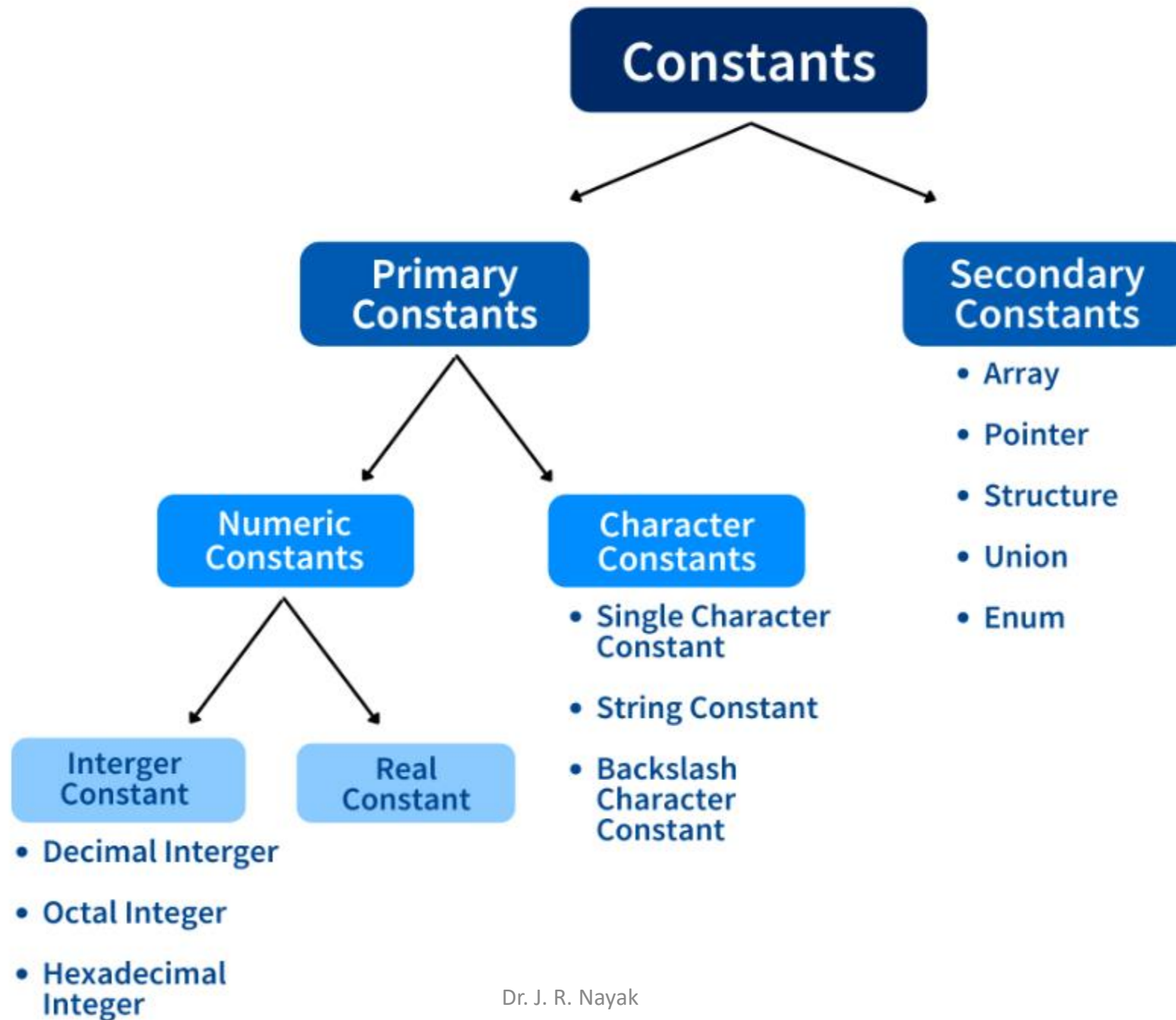
*const data\_type var\_name = value;*

```
#include<iostream>
using namespace std;
int main()
{
    const int b=25;
    cout<<"The value of b is "<<b<<endl;
    return(0);
}
```

**Output:** The value of b is 25

```
#include<iostream>
using namespace std;
int main()
{
    const int b=25;
    cout<<"The value of b is "<<b<<endl;
    b=15;
    cout<<"The value of b is "<<b;
    return 0;
}
```

**Output:** error: assignment of read-only variable 'b'



## Integer Constants

There are three types of integers:

- i. Decimal integer
- ii. Octal integer
- iii. Hexadecimal integer

### **i. Decimal integer:**

Decimal integer should not start with 0, embedded spaces, commas and non-digit characters are not permitted.



## ii. Octal integer:

Octal integer should start with 0.

*cout*<< *oct* will print octal integer.

Ex:-

```
#include<iostream>
using namespace std;
int main()
{
    int b=025;
    cout<<"The value of b in decimal is "<<b<<endl;
    cout<<"The value of b in octal is "<<oct<<b<<endl;
    return(0);
}
```

### **Output:**

The value of b in decimal is 21  
The value of b in octal is 25

### iii. Hexadecimal integer:

Hexadecimal integer should start with 0x or 0X.

*cout<< hex* will print hexadecimal integer.

Ex:

```
#include<iostream>
using namespace std;
int main()
{
    int b=0x2B5;
    cout<<"The value of b in decimal is "<<b<<endl;
    cout<<"The value of b in octal is "<<oct<<b<<endl;
    cout<<"The value of b in hexadecimal is "<<hex<<b<<endl;
    return(0);
}
```

#### **Output:**

The value of b in decimal is 693

The value of b in octal is 1265

The value of b in hexadecimal is 2b5

#### iv. Binary integer:

Binary integer should start with 0b or 0B.

```
#include <iostream>
using namespace std;
int main()
{

    int a = 0b00001111;
    cout << "The value of a in decimal is: "<<dec<<a<<endl;
    cout << "The value of a in octal is: "<<oct<<a<<endl;
    cout << "The value of a in hexadecimal is: "<<hex<<a<<endl;
    return 0;
}
```

#### **Output:**

*The value of a in decimal is: 15*

*The value of a in octal is: 17*

*The value of a in hexadecimal is: f*

## Real Constants

**Real Constants** also known as **Floating point Constants** are numbers that have a whole number followed by a decimal point followed by the fractional number. The real constants can be expressed in two forms:

- Fractional Form
- Exponential Form

Have a look at the rules that guide the construction of Real Constants in Fractional Form:

- It should have a floating point.
- It should have at least have one digit.
- It can have a positive or negative sign. In case it doesn't have any sign it is taken as positive.
- No blank spaces or commas are allowed.

Eg: 546.236, +453.89, -22.564

## Syntax:

*float variable\_name=real\_constant;*

*// Define a real constant*

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    float a = 11.25;
```

```
    cout << "The value of a is: "<<a<<endl;
```

```
    return 0;
```

```
}
```

***Output:*** *The value of a is: 11.25*

- In case the value we are dealing with is too small like **0.00003** or too big like **300000**, we can use the *exponential form* to express these constants as **3.0e-5** and **3.0e5**.
- The part before the ‘e’ is known as *mantissa* and the part after the ‘e’ is known as *exponent*.

Rules for construction of Real Constants in Exponential Form:

- The mantissa and the exponential part should have a letter ‘e’ between them.
- The exponent should have at least one digit.
- It can have a positive or negative sign. In case it doesn’t have any sign it is taken as positive.

**Eg:** 5.6e9, +2.0e-7, -5.6e+8, -7.8e-5.

```
#include <iostream>
using namespace std;
int main()
{

    float a = 11.25e-5;
    float b = 11.25e+3;
    cout << "The value of a is: "<<a<<endl;
    cout << "The value of b is: "<<b<<endl;
    return 0;
}
```

**Output:**

The value of a is: 0.0001125

The value of b is: 11250

## Character Constants

A **Character Constant** is a single alphabet, digit or special character enclosed within '*single quotes*'.

Rules for construction of Character Constants :

- Should be enclosed within single quotes.
- It can have only one character, digit or special character.

**Eg:** 'A', 'f', '4'

**Syntax:**

*char variable\_name = 'character';*

*// Define a character constant*



```
#include <iostream>
using namespace std;
int main()
{
    char a = 'I'; char b = 'T'; char c = 'E'; char d = 'R';
    cout << a<<b<<c<<d<<endl;
    cout<<"ASCII value of I is:"<<int(a);
    return 0;
}
```

**Output:**

ITER

ASCII value of I is:73

- **ASCII** stands for American Standard **Code** for Information Interchange.
- This ASCII value represents the character variable in numbers, and each character variable is assigned with some number range from 0 to 127.

## String constant

A **string constant** is a combination of alphabets, digits and special characters enclosed within “*double quotes*”. Have a look at the rules that guide the construction of String Constants:

- Should be enclosed within double quotes.
- It can be of any length.
- It ends with a null character assigned to it by the compiler.

**Eg:** “Welcome to ITER”, “Good to see you”, “8 apples”, “learning is fun”.

## Syntax:

*char variable\_name[] = "string";*

*// Define a string constant*

```
#include <iostream>
using namespace std;
int main()
{
    char a []= "Welcome to ITER";
    cout <<a<<endl;
    cout <<a[1]<<endl;
    return 0;
}
```

## Output:

Welcome to ITER  
e

# Backslash Character Constants / Escape Sequences

- There are some characters which are impossible to enter into a string from keyboard like backspace, vertical tab etc.
- Because of this reason, C++ includes a set of backslash character which have special meaning in C++ language.
- The backslash character ('\') changes the interpretation of the characters by compiler.

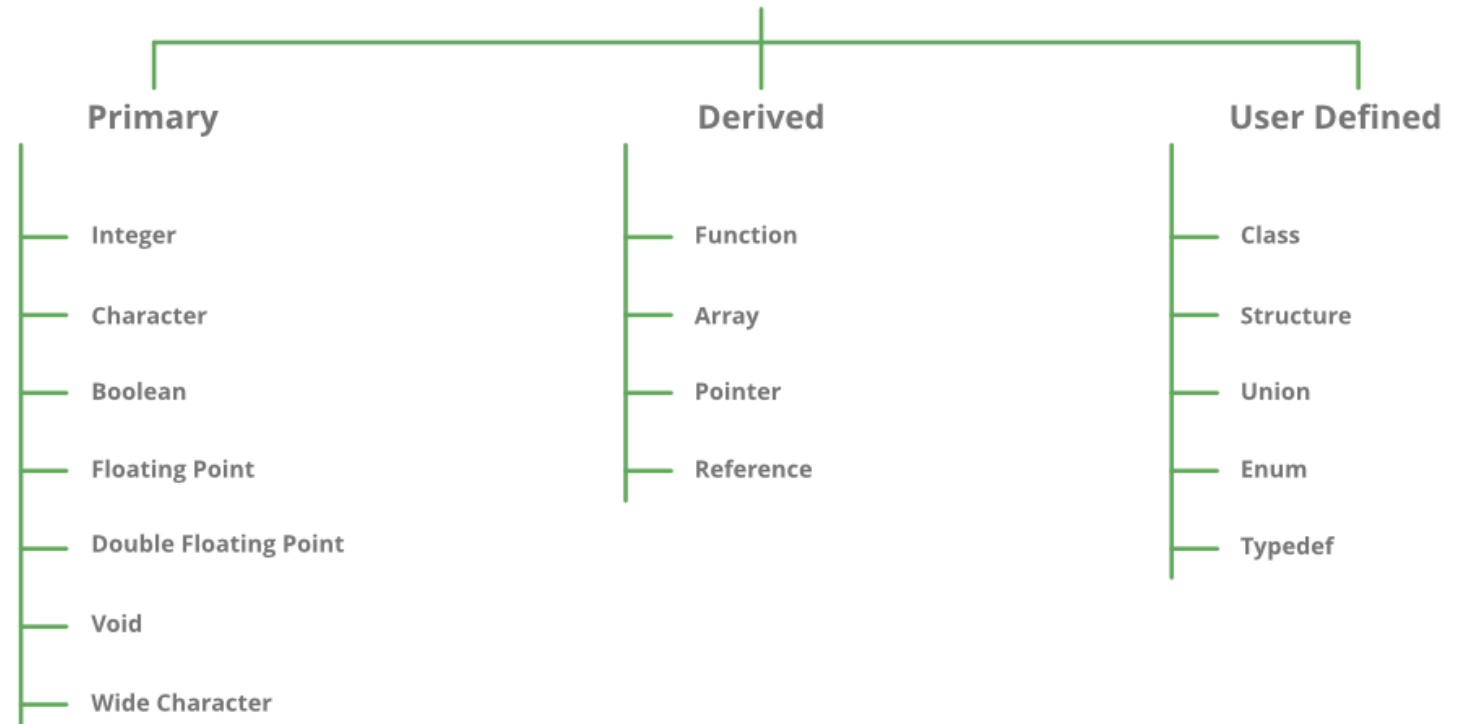
Escape Sequence	Name	Description
<code>\\</code>	Backslash	Inserts a backslash character.
<code>\'</code>	Single quote	Displays a single quotation mark.
<code>\''</code>	Double quote	Displays double quotation marks.
<code>\?</code>	Question mark	Displays a question mark.
<code>\a</code>	Alert (bell)	Generates a bell sound in the C++ program.
<code>\b</code>	Backspace	Moves the cursor one place backward.
<code>\f</code>	Form feed	Moves the cursor to the start of the next logical page.
<code>\n</code>	New line	Moves the cursor to the start of the next line.
<code>\r</code>	Carriage return	Moves the cursor to the start of the current line.
<code>\t</code>	Horizontal tab	Inserts some whitespace to the left of the cursor and moves the cursor accordingly.
<code>\v</code>	Vertical tab	Inserts vertical space.
<code>\0</code>	Null character	Represents the NULL character.
<code>\ooo</code>	Octal number	Represents an octal number.
<code>\xhh</code>	Hexadecimal number	Represents a hexadecimal number.

# Data Types

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

- Primary Data Types
- User Defined Data Types
- Derived data Types

## DataTypes in C / C++



## Primary Data Types:

Primitive data types are the most basic data types that are used for representing simple values such as integers, float, characters, etc.

Data Type	Size	Description
int	2 or 4 bytes	Stores whole numbers, without decimals
float	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits
double	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits
char	1 byte	Stores a single character/letter/number, or ASCII values

**sizeof() operator:** sizeof() operator is used to find the number of bytes occupied by a variable/data type in computer memory.

```
#include<iostream>
using namespace std;
int main()
{
    int b=25;
    cout<<sizeof(b)<<endl;
    cout<<sizeof(int)<<endl;
    return 0;
}
```

**Output:**  
4  
4

## Size and Range of the data types

- A signed data type can store both positive and negative values. Range of signed data type

$$-2^{\text{bits}/2} \text{ to } 2^{\text{bits}/2} - 1$$

- An unsigned data type can store positive values. Range of unsigned data type

$$0 \text{ to } 2^{\text{bits}} - 1$$

- int occupies four bytes of memory while short int occupies only two bytes of memory.
- float stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits.
- Double stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits.
- Long stores fractional numbers, containing one or more decimals. Sufficient for storing 19 decimal digits.
- A dot (.) followed by a number that specifies how many digits that should be shown after the decimal point

**Note:-** **limits.h** header file is defined to find the range of fundamental data-types.



Data Types	Memory Size	Format Specifier	Literal	Range
<b>char</b>	1 byte			−128 to 127
signed char	1 byte			−128 to 127
unsigned char	1 byte			0 to 255
<b>Short int</b>	1 byte or 2 byte			−128 to 127 or −32,768 to 32,767
signed short int	1 byte or 2 byte			−128 to 127 or −32,768 to 32,767
unsigned short int	1 byte or 2 byte			0 to 255 or 0 to 65,535
<b>int</b>	2 byte or 4 byte		-124	−32,768 to 32,767 or 2,147,483,648 to 2,147,483,647
signed int	2 byte or 4 byte		-124	−32,768 to 32,767 or 2,147,483,648 to 2,147,483,647
unsigned int	2 byte or 4 byte		45678U	0 to 65,535 or 0 to 4,294,967,295
<b>long int</b>	4 byte or 8 byte		-545445L	-2,147,483,648 to 2,147,483,647
signed long int	4 byte or 8 byte		-545445L	-2,147,483,648 to 2,147,483,647
unsigned long int	4 byte or 8 byte		987654UL	0 to 4,294,967,295
<b>float</b>	4 byte		-1.2f	1.2E-38 to 3.4E+38
<b>double</b>	8 byte		1.234	2.3E-308 to 1.7E+308
<b>long double</b>	10 byte		1.23456786L	3.4E-4932 to 1.1E+4932

## Program Explanation

The header file “**limits.h**” is used to find minimum and maximum constants of integer and character data type.

Name	Expresses
CHAR_MIN	The minimum value for an object of type char
CHAR_MAX	Maximum value for an object of type char
SCHAR_MIN	The minimum value for an object of type Signed char
SCHAR_MAX	Maximum value for an object of type Signed char
UCHAR_MAX	Maximum value for an object of type Unsigned char
CHAR_BIT	Number of bits in a char object
MB_LEN_MAX	Maximum number of bytes in a multi-byte character
SHRT_MIN	The minimum value for an object of type short int
SHRT_MAX	Maximum value for an object of type short int
USHRT_MAX	Maximum value for an object of type Unsigned short int
INT_MIN	The minimum value for an object of type int
INT_MAX	Maximum value for an object of type int

UINT_MAX	Maximum value for an object of type Unsigned int
LONG_MIN	The minimum value for an object of type long int
LONG_MAX	Maximum value for an object of type long int
ULONG_MAX	Maximum value for an object of type Unsigned long int
LLONG_MIN	The minimum value for an object of type long long int
LLONG_MAX	Maximum value for an object of type long long int
ULLONG_MAX	Maximum value for an object of type Unsigned long long int

**typeid** is an operator that is used where the dynamic type of an object needs to be known.

# Variables

- A variable in C++ language is the name associated with some memory location to store data of different types.

## *Syntax:*

*data\_type variable\_name = value; // defining single variable*

*or*

*data\_type variable\_name1, variable\_name2; // defining multiple variable*

## *Rules for naming a variable:*

- A variable name can only have letters (both uppercase and lowercase letters), digits and underscore.
- The first letter of a variable should be either a letter or an underscore or \$.
- There is no rule on how long a variable name (identifier) can be.

## Derived Data Types

➤ Derived data types are primary data types that are grouped together. You can group many elements of similar data types. These data types are defined by the user. The following are the derived data types in C:

- Array
- Pointers
- Structure
- Union

## User-defined data types

The data types that are defined by the user depending upon the use case of the programmer are termed user-defined data types.

There are 3 aspects of defining a variable:

1. Variable Declaration
2. Variable Definition
3. Variable Initialization

## **1. Variable Declaration**

Variable declaration in C++ tells the compiler about the existence of the variable with the given name and data type. When the variable is declared compiler automatically allocates the memory for it.

## **2. Variable Definition**

In the definition of a C++ variable, the compiler allocates some memory and some value to it. A defined variable will contain some random garbage value till it is not initialized.

## **3. Variable Initialization**

Initialization of a variable is the process where the user assigns some meaningful value to the variable.

`int var; // variable definition`

`var = 10; // initialization`

*or*

`int var = 10; // variable declaration and definition`

## Variable Types

1. Local Variables
2. Global Variables
3. Static Variables
4. Automatic Variables
5. Extern Variables
6. Register Variables

# 1. Local Variables

A local variable in C++ is a variable that is declared inside a function or a block of code. Its scope is limited to the block or function in which it is declared.

```
#include<iostream>
using namespace std;
void func()
{
    int a=18;
}
int main()
{
    cout<<"The value of a is: "<<a;
    return 0;
}
```

**Error:-**'a' was not declared in this scope

```
#include<iostream>
using namespace std;
int main()
{
    int a=18;
    cout<<"The value of a is: "<<a;
    return 0;
}
```

**Output:-** The value of a is: 18



## 2. Global Variables

A Global variable in C++ is a variable that is declared outside the function or a block of code. Its scope is the whole program i.e. we can access the global variable anywhere in the C++ program after it is declared.

```
#include<iostream>  
using namespace std;  
int a=18;  
int main()  
{  
    cout<<"The value of a is: "<<a;  
    return 0;  
}
```

**Output:-** *The value of a is: 18*

### 3. Static Variables

- Static variables have the property of preserving their value even after they are out of their scope!
- Hence, a static variable preserves its previous value in its previous scope and is not initialized again in the new scope.

**Syntax:** *static data\_type var\_name = var\_value;*

```
#include<iostream>
using namespace std;
int fun()
{
    int a=1;
    a=a+1;
    return a;
}
int main()
{
    cout<<"The value of a is: "<<fun()<<endl; //2
    cout<<"The value of a is: "<<fun()<<endl; //2
    cout<<"The value of a is: "<<fun()<<endl; //2
    return 0;
}
```

```
#include<iostream>
using namespace std;
int fun()
{
    static int a=1;
    a=a+1;
    return a;
}
int main()
{
    cout<<"The value of a is: "<<fun()<<endl; //2
    cout<<"The value of a is: "<<fun()<<endl; //3
    cout<<"The value of a is: "<<fun()<<endl; //4
    return 0;
}
```

## 4. Automatic Variables

Auto storage class is the default storage class for all the local variables. It is created when function is called.

When the execution of function is completed, variables are destroyed automatically.

**Syntax:** *auto var\_name = var\_value;*

```
#include<iostream>
using namespace std;
int main()
{
    auto a=1;
    if (1==1)
    {
        auto b=5;
        cout<<"The value of a is: "<<a<<endl;
        cout<<"The value of b is: "<<b<<endl;
    }
    cout<<"The value of a is: "<<a<<endl;
    cout<<"The value of b is: "<<b<<endl;
    return 0;
} error: 'b' was not declared in this scope
```

```
#include<iostream>
using namespace std;
int main()
{
    auto a=1;
    if (1==1)
    {
        auto b=5;
        cout<<"The value of a is: "<<a<<endl;
        cout<<"The value of b is: "<<b<<endl;
    }
    cout<<"The value of a is: "<<a<<endl;
    return 0;
}
```

**Output:**

The value of a is: 1  
The value of b is: 5  
The value of a is: 1

## 5. External Variables

- The external variable is used when a particular files need to access a variable from another file.

### Syntax:

*extern data\_type variable\_name;*

### Properties of extern Variable

- When we declare extern variable; no memory is allocated. Only the property of the variable is announced.
- Multiple declarations of extern variable is allowed within the file. This is not the case with automatic variables.
- The extern variable says to the compiler “Go outside my scope and you will find the definition of the variable that I declared.”
- The compiler believes that whatever that extern variable said is true and produces no error. Linker throws an error when it finds no such variable exists.
- When an extern variable is initialized, then memory for this is allocated and it will be considered defined.

- Keep all the files in same folder.

#### secondary.cpp

```
#include "main.cpp"
#include<iostream>
using namespace std;
extern int a;
extern int b;
int main()
{
    cout<<"Value of a is: "<<a<<endl;
    cout<<"Value of b is: "<<b<<endl;
}
```

#### **Output:**

Value of a is: 11

Value of b is: 15

#### main.cpp

```
#include<iostream>
using namespace std;
int a=11;
int b=15;
```

#### third.cpp

```
#include "main.cpp"
#include<iostream>
using namespace std;
extern int a;
extern int b;
int main()
{
    int c=a+b;
    cout<<"Value of c is: "<<c<<endl;
}
```

#### **Output:**

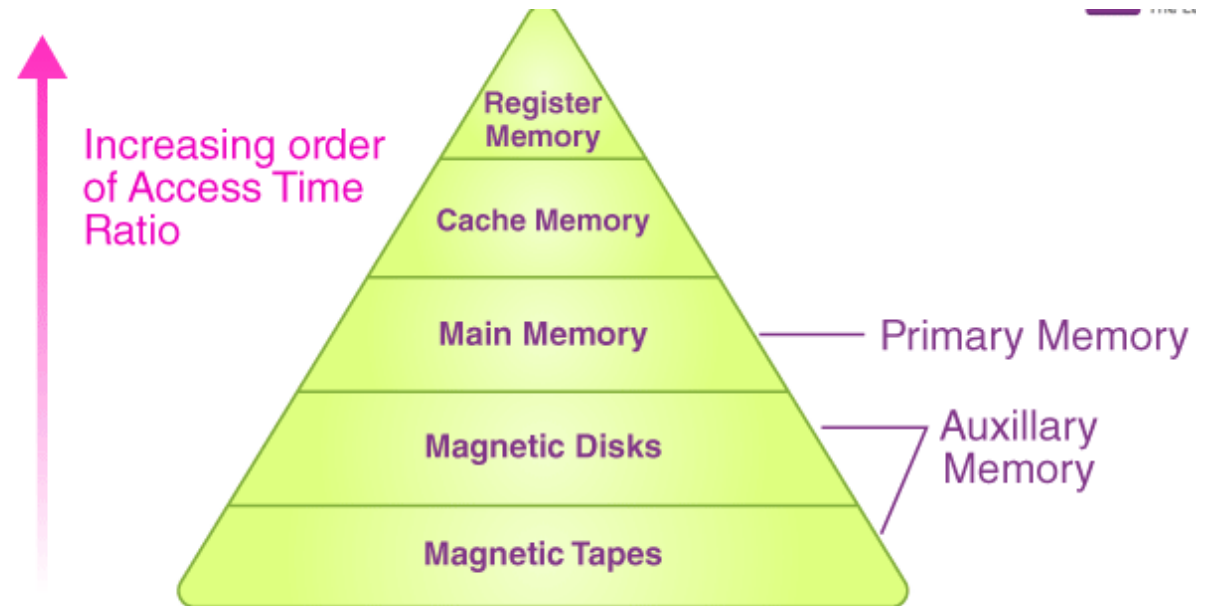
Value of c is: 26

## 6. Register Variables

- Register variables tell the compiler to store the variable in CPU register instead of memory. Frequently used variables are kept in registers and they have faster accessibility.
- They are local to the function.
- Till the end of the execution of the block in which it is defined.

### Syntax:

*register data\_type variable\_name = value;*



```

#include <bits/stdc++.h>
using namespace std;
int fun()
{
    int a=5;
    for (int i=0; i<10000000000; i++)
    {
        a=a++;
    }
    return a;
}
int main()
{
    clock_t start, end;
    start = clock();
    fun();
    end = clock();
    long double time_taken = double(end - start) /
double(CLOCKS_PER_SEC);
    cout << "Time taken by program is : " << fixed
        << time_taken << setprecision(10);
    cout << " sec " << endl;
    return 0;
} Output: Time taken by program is : 1.973000 sec

```

```

#include <bits/stdc++.h>
using namespace std;
int fun()
{
    register int a=5;
    for (int i=0; i<10000000000; i++)
    {
        a=a++;
    }
    return a;
}
int main()
{
    clock_t start, end;
    start = clock();
    fun();
    end = clock();
    long double time_taken = double(end - start) /
double(CLOCKS_PER_SEC);
    cout << "Time taken by program is : " << fixed
        << time_taken << setprecision(10);
    cout << " sec " << endl;
    return 0;
} Output: Time taken by program is : 1.596000 sec

```

# Constants

- The constants in C are the read-only variables whose values cannot be modified once they are declared in the C program.

## Syntax:

*const data\_type var\_name = value;*

```
#include <iostream>
using namespace std;
int main()
{
    const int a=11;
    cout<<"The value of a is: "<<a;
    return 0;
}
```

*Output:*

*The value of a is: 11*



## Alternate Syntax:

```
#define const_name value;
```

```
#include <iostream>
using namespace std;
#define a 5
int main()
{
    cout <<"Value of a is: "<<a;
    return 0;
}
```

### **Output:**

*Value of a is: 5*

## Alternate Syntax:

*constexpr data\_type const\_name = value;*

```
#include <iostream>
using namespace std;
int main()
{
    constexpr int a = 4;
    cout <<"Value of a is:"<<a;
    return 0;
}
```

### **Output:**

*Value of a is: 4*

# Input/output Statements

- Input and Output statement are used to read and write the data in C++ programming. These are embedded in `iostream.h` (standard Input/Output header file).
- Input means to provide the program with some data to be used in the program and Output means to display data on screen or write the data to a printer or a file.
- There are mainly two of Input/Output functions are used for this purpose.

Formatted I/O functions

Unformatted I/O functions

**Formatted I/O functions** which refers to an Input or Output data that has been arranged in a particular format.

There are mainly 6 formatted I/O functions discussed as follows:

`cout<<`

`cin>>` `endl`

`cerr<<`

`clog<<`

## Unformatted I/O functions

There are mainly six unformatted I/O functions discussed as follows:

`getchar()`

`putchar()`

`getch()`

`getche()`

`gets()`

`puts()`

## getchar()

This function is an Input function. It is used for reading a single character from the keyboard. It is a buffered function.

### Syntax:

```
char n;  
n = getchar();
```

```
#include <iostream>  
using namespace std;  
int main()  
{  
    char a;  
    a = getchar();  
    cout << "The entered character is: " << a;  
    return 0;  
}
```

**Input:** B

**Output:** The entered character is: B

```
#include <iostream>  
using namespace std;  
int main()  
{  
    int a;  
    a = getchar();  
    cout << "The entered character is: " << a;  
    return 0;  
}
```

**Input:** B

**Output:** The entered character is: 66

## **putchar()**

This function is an output function. It is used to display a single character on the screen.

This function is an Output function.

### **Syntax:**

```
char n;
```

```
putchar(n);
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    char a;
```

```
    a = getchar();
```

```
    cout << "The entered character is: ";
```

```
    putchar(a);
```

```
    return 0;
```

```
}
```

*Input: B*

*Output: The entered character is: B*

## getch()

- This is also an input function. This is used to read a single character from the keyboard like getchar() function.
- But getchar() function is a buffered is function, getch() function is a non-buffered function.
- The character data read by this function is directly assigned to a variable rather it goes to the memory buffer, the character data is directly assigned to a variable without the need to press the Enter key.
- Another use of this function is to maintain the output on the screen till you have not press the Enter Key.

### Syntax:

*v = getch();*

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
    char a,b;
    a = getchar();
    cout << "The entered character is: ";
    putchar(a);
    b = getch();
    cout << "\nThe entered character is: ";
    putchar(b);
    return 0;
}
```

F

The entered character is: F

The entered character is: j

## **getche()**

getche() function reads a single character from the keyboard and displays immediately on the output screen without waiting for enter key.

### **Syntax:**

*v = getche();*

```
#include <iostream>
#include<conio.h>
using namespace std;
int main()
{
    char a;
    a = getche();
    return 0;
}
```

**Output: j**



## gets()

- This function is an input function. It is used to read a string from the keyboard.
- It will read a string when you type the string from the keyboard and press the Enter key from the keyboard.
- It is defined in <stdio> header file.

### Syntax:

```
char n[20];
```

```
gets(n);
```

```
#include <iostream>
#include <stdio>
using namespace std;
int main()
{
    char a[100];
    cout << "Enter string: ";
    gets(a);
    cout << "You entered: "<< a;
    return 0;
}
```

**Input:** Enter string: Training class

**Output:** You entered: Training class

## puts()

- This is an output function. It is used to display a string inputted by gets() function.
- This function appends a newline (“\n”) character to the output.

### Syntax:

*puts(v);*

```
#include <iostream>
#include <cstdio>
using namespace std;
int main()
{
    char a[100];
    cout << "Enter string: ";
    gets(a);
    cout << "You entered: ";
    puts(a);
    return 0;
}
```

**Input:** Enter string: Class

**Output:** You entered: Class

# Operators in C++

- An operator is a symbol that tells the compiler to perform specific mathematical or logical functions.
- C++ has many built-in operators and can be classified into 8 types.
  - i. Arithmetic Operators
  - ii. Relational Operators
  - iii. Logical Operators
  - iv. Assignment Operators
  - v. Bitwise Operators
  - vi. Increment and Decremental Operators
  - vii. Conditional Operators
  - viii. Special Operators

## i. Arithmetic Operators

These operators are used to perform arithmetic/mathematical operations on operands. Examples: (+, -, \*, /, %).

Operator	Name	Description	Example
+	Addition	Adds together two values	$x + y$
-	Subtraction	Subtracts one value from another	$x - y$
*	Multiplication	Multiplies two values	$x * y$
/	Division	Divides one value by another	$x / y$
%	Modulus	Returns the division remainder	$x \% y$

- These operators are called as binary operators. Operators that operate or work with two operands are binary operators.

```
#include <iostream>

using namespace std;

int main()
{
    int a,b,c,d,e,f,g;

    a=4; b=45;

    cout << "Addition: "<<a+b<<endl;
    cout << "Subtraction: "<<a-b<<endl;
    cout << "Multiplication: "<<a*b<<endl;
    cout << "Divison: "<<b/a<<endl;
    cout << "Modulus: "<<b%a<<endl;

    return 0;
}
```

### **Output:**

Addition: 49

Subtraction: -41

Multiplication: 180

Divison: 11

Modulus: 1

## ii. Relational Operators

- A relational operator checks the relationship between two operands.
- *If the relation is true, it returns 1; if the relation is false, it returns value 0.*
- Relational operators are used in decision making and loops.

Operator	Meaning of Operator	Example
==	Equal to	5 == 3 is evaluated to 0 and 5 == 5 is evaluated to 1 .
>	Greater than	5 > 3 is evaluated to 1 and 3 > 3 is evaluated to 0.
<	Less than	5 < 3 is evaluated to 0 and 3 < 3 is evaluated to 0.
!=	Not equal to	5 != 3 is evaluated to 1 and 3 != 3 is evaluated to 0.
>=	Greater than or equal to	5 >= 3 is evaluated to 1 and 3 >= 3 is evaluated to 1
<=	Less than or equal to	5 <= 3 is evaluated to 0 and 3 <= 3 is evaluated to 1

```

#include <iostream>

using namespace std;

int main()
{
    int a,b,c;

    a=5; b=3; c=3;

    cout << (a==b) <<'\t'<<(b==c)<<endl;

    cout << (a>b)<<'\t'<<(b>c)<<'\t'<<(b<a)<<endl;

    cout << (a!=b)<<'\t'<<(b!=c)<<endl;

    cout << (a>=b)<<'\t'<<(b>=c)<<'\t'<<(b<=a)<<endl;

    return 0;
}

```

**Output:**

```

0    1

1    0    1

1    0

1    1    1

```

### iii. Logical Operators

- An expression containing logical operator returns either 0 or 1 depending upon whether expression results true or false respectively.

Operator	Meaning
&& or and	Logical AND. True only if all operands are true
or or	Logical OR. True only if either one operand is true
!	Logical NOT. True only if the operand is 0
xor	Logical XOR. True only if one operand is true and other is false



```

#include <iostream>

using namespace std;

int main()
{
    int a,b,c;
    a=5; b=3; c=3;
    cout << (a==b && b==c)<<endl;
    cout << (a==b and a==c)<<endl;
    cout << (a==b || b==c)<<endl;
    cout << (a==b or a==c)<<endl;
    cout << !(a==b)<<'\\t'<<not(b==c)<<endl;
    cout << (a!=b ^ a!=b)<<'\\t'<<(b==c xor c==a)<<endl;
    return 0;
}

```

**Output:**

0

0

1

0

1    0

0    1

## iv. Assignment Operators

The assignment operator is used to assign the value, variable and function to another variable.

Operator	Meaning Of Operator	Example	Same as
=	Simple assignment operator	x=y	x=y
+=	Add left operand to right operand then assign result to left operand	x+=y	x=x+y
-=	subtract right operand from left operand then assign result to left operand	x-=y	x=x-y
*=	multiply left operand with right operand then assign result to left operand	x*=y	x=x*y
/=	divide left operand with right operand then assign result to left operand	x/=y	x=x/y
%=	take modulus left operand with right operand then assigned result in left operand	x%=y	x=x%y

```
#include <iostream>
using namespace std;
int main()
{
    int a,b,c;
    a=5; b=3;
    cout <<(c=a)<<endl;
    cout << (a+=b)<<endl;
    cout << (a-=b)<<endl;
    cout << (a*=b)<<endl;
    cout << (a/=b)<<endl;
    cout << (a%=b)<<endl;
    return 0;
}
```

**Output:**

5

8

5

15

5

2

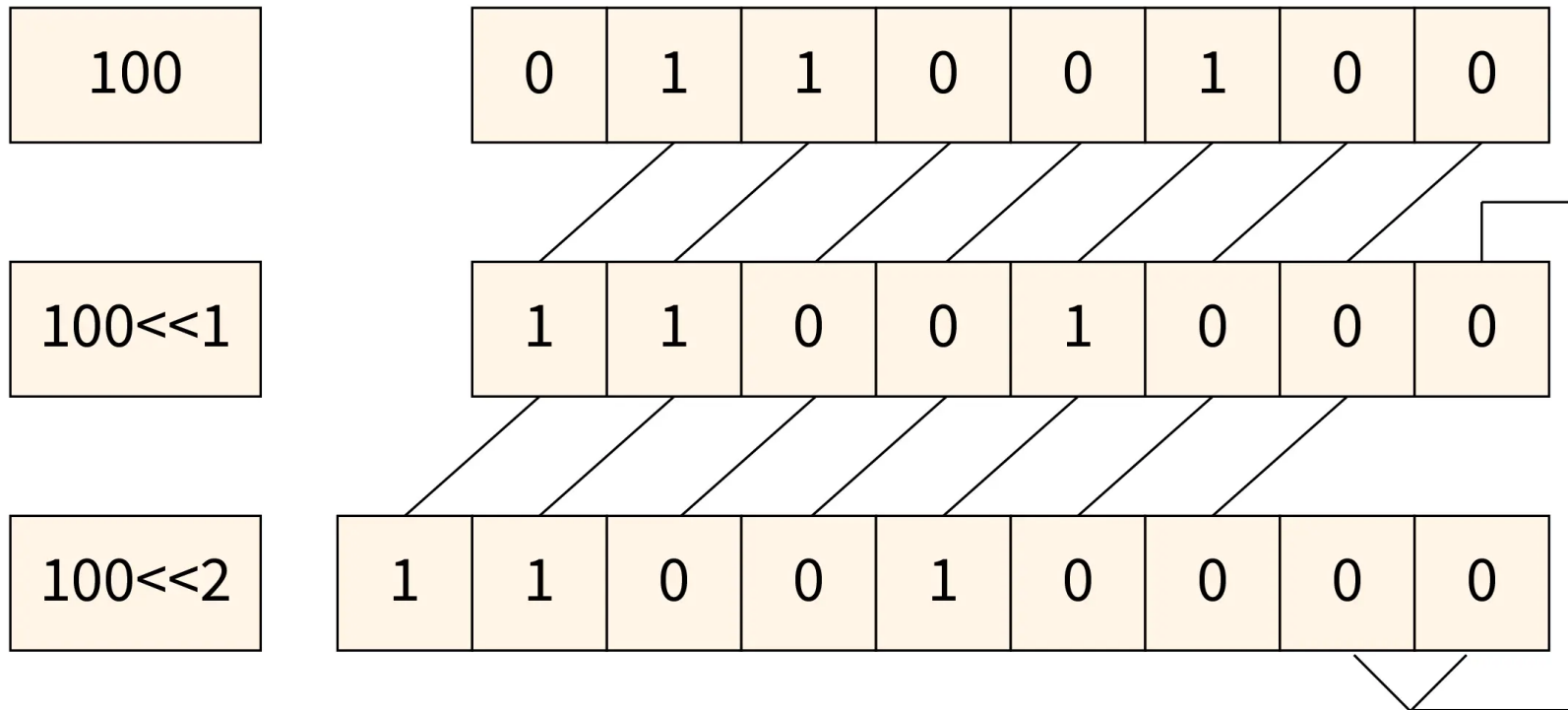
## v. Bitwise Operators

The bitwise operators are the operators used to perform the operations on the data at the bit-level.

Operator	Meaning Of Operator	Example	Same as
<<=	Left Shift Assignment Operator means the left operand is left shifted by right operand value and assigned value to left operand	x<<=y	x=x<<y
>>=	Right shift Assignment Operator means the left operand is right shifted by right operand value and assigned value to left operand	x>>=y	x=x>>y
&=	Bitwise AND Assignment Operator means does AND on every bit of left operand and right operand and assigned value to left operand	x&=y	x=x&y
=	Bitwise inclusive OR Assignment Operator means does OR on every bit of left operand and right operand and assigned value to left operand	x =y	x=x y
^=	Bitwise exclusive OR Assignment Operator means does XOR on every bit of left operand and right operand and assigned value to left operand	x^=y	x=x^y

## Left Shift(<<)

It is a binary operator that takes two numbers, left shifts the bits of the first operand, and the second operand decides the number of places to shift.



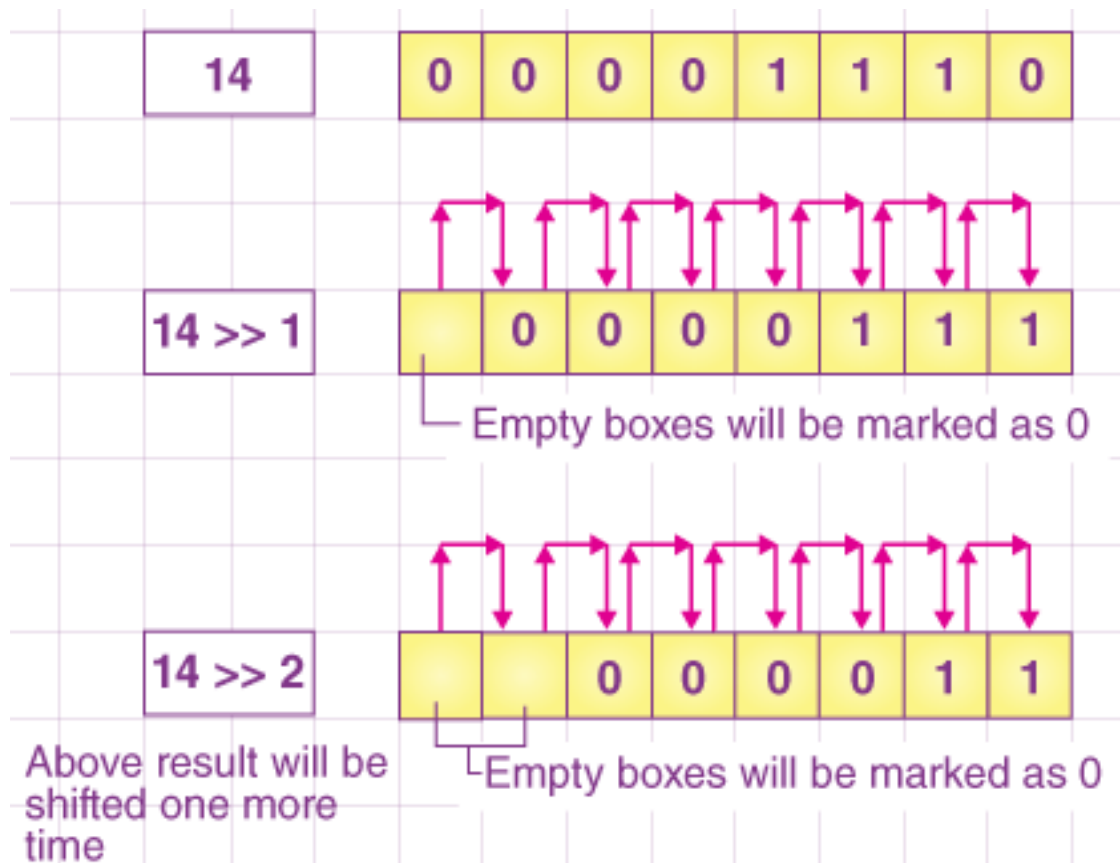
The value of  $a \ll n$  can be found

by using this formula

$$a * 2^n$$

## Right Shift(>>)

It is a binary operator that takes two numbers, right shifts the bits of the first operand, and the second operand decides the number of places to shift.



The value of  $a \gg n$  can be found by using

this formula

$$a/2^n$$

Truth table of the bitwise operators.

X	Y	X&Y	X Y	X^Y
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

### Bitwise AND (&=)

x =6; y=4;

The binary representation of the above two variables are  
given below:

x =      0110

y =      0100

x&y =    0100

## Bitwise OR (|=)

x =6; y=4;

The binary representation of the above two variables are given below:

x =        0110

y =        0100

x|y =      0110

## Bitwise exclusive OR (^=)

x =6; y=4;

The binary representation of the above two variables are given below:

x =        0110

y =        0100

x^y =      0010



```

#include <iostream>
using namespace std;
int main()
{
    int a,b;
    a=100; b=2;
    cout <<(a<<=b)<<endl;
    a=14;
    cout << (a>>=b)<<endl;
    a=6; b=4;
    cout << (a&=b)<<endl;
    a=6; b=4;
    cout << (a|=b)<<endl;
    a=6; b=4;
    cout << (a^=b)<<endl;
    return 0;
}

```

**Output:**

400

3

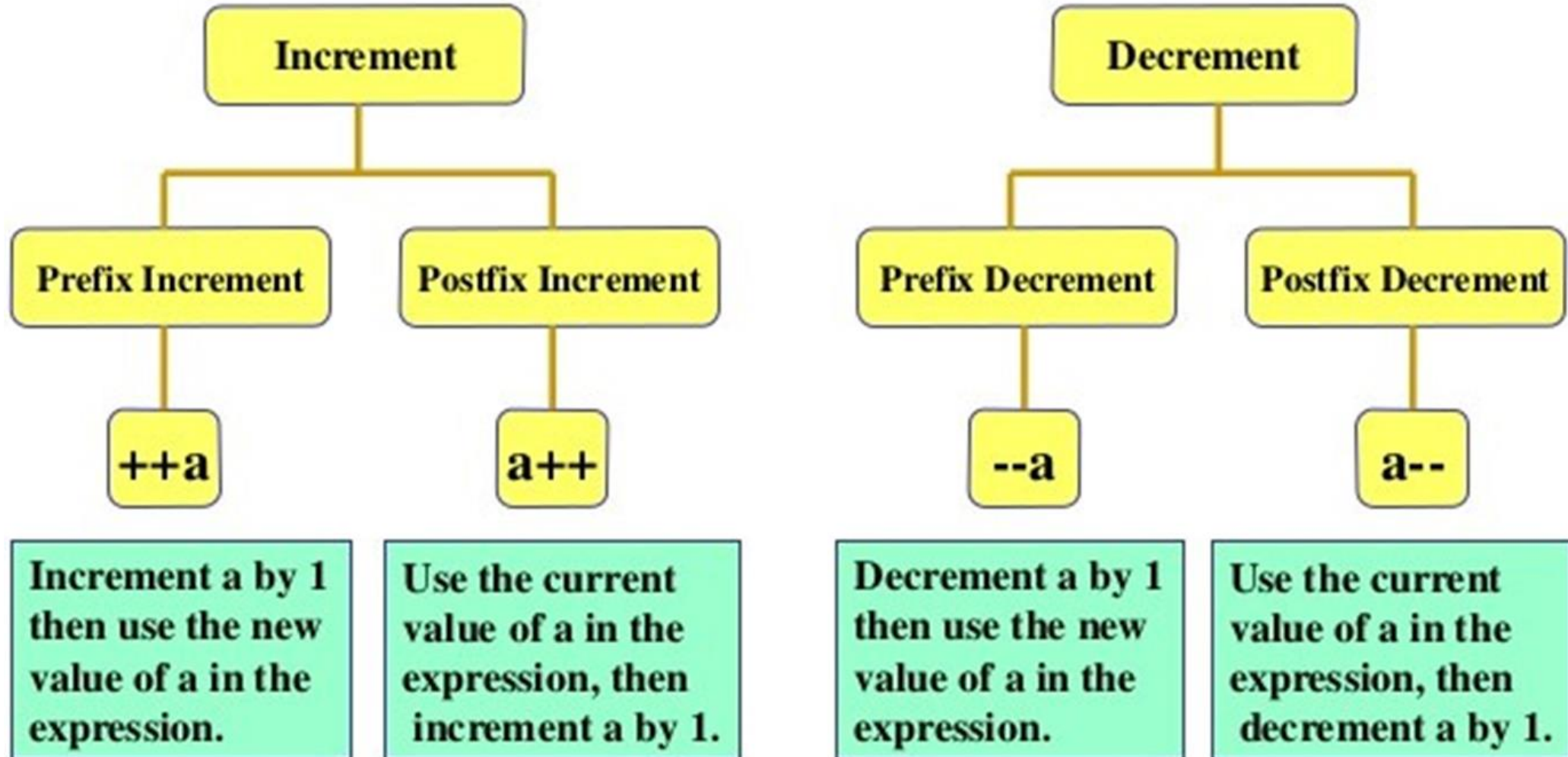
4

6

2

## vi. Increment and Decremental Operators

- The increment ( ++ ) and decrement ( -- ) operators in C++ are unary operators for incrementing and decrementing the numeric values by 1 respectively.



```
#include <iostream>
using namespace std;
int main()
{
    int a=10;
    cout << (++a) << endl;
    cout << (a++) << endl;
    cout << (a) << endl;
    cout << (--a) << endl;
    cout << (a--) << endl;
    cout << a << endl;
    return 0;
}
```

**Output:**

11

11

12

11

11

10

## vii. Conditional Operators

- The conditional operator in C++ is kind of similar to the if-else statement as it follows the same algorithm as of if-else statement but the conditional operator takes less space and helps to write the if-else statements in the shortest way possible.
- It is also known as the ternary operator in C++ as it operates on three operands.

### Syntax:

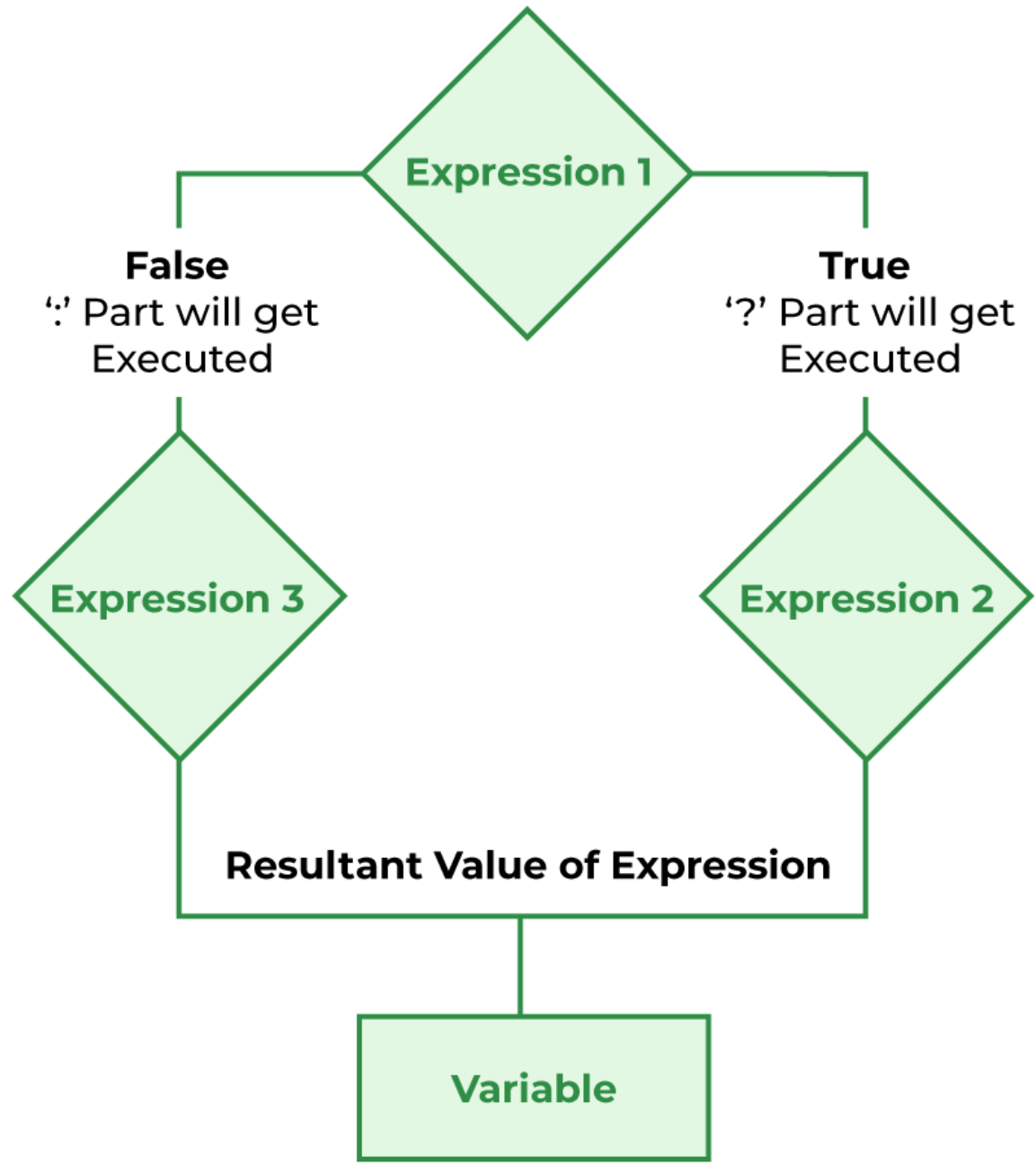
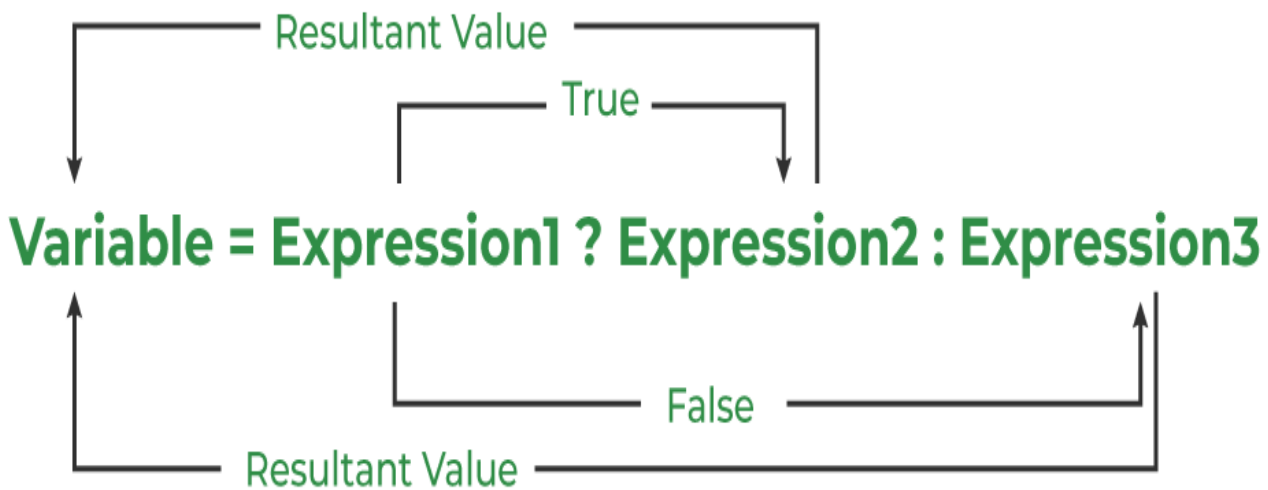
*variable = Expression1 ? Expression2 : Expression3;*

*Or*

*variable = (condition) ? Expression2 : Expression3;*

*Or*

*(condition) ? (variable = Expression2) : (variable = Expression3);*



## Examples

```
//WAP to find the greater number
#include <iostream>
using namespace std;
int main()
{
    int a=1; int b=5;
    (a>b) ? cout<<"a is greater than b" : cout<<"b is greater than a";
    return 0;
}
```

**Output:**

b is greater than a

```
3  int main()
4  {
5      int yr = 1900; |
6      (yr%4==0) ? (yr%100!=0? printf("The year %d is a leap year",yr)
7          : (yr%400==0 ? printf("The year %d is a leap year",yr)
8              : printf("The year %d is not a leap year",yr)))
9          : printf("The year %d is not a leap year",yr);
10     return 0;
11 }
```

The year 1900 is not a leap year

## viii. Special Operators

- Apart from the above operators, there are some other operators available in C used to perform some specific tasks.
  - a. Comma Operator
  - b. Type Cast Operator
  - c. Reference Operator
  - d. Dereference Operator
  - e. Double Pointer
  - f. sizeof

## a. Comma Operator

- The comma operator is type of special operators in C++ which evaluates first operand and then discards the result of the same, then the second operand is evaluated and result of same is returned.
- The comma operator has the lowest precedence of any C++ operator.

### **For Example:**

```
int val= (10, 30);
```

In this example 10 is discarded and 30 is assigned to val variable.

- Comma acts as both operator and separator. In C++, comma ( , ) can be used in three contexts:
  - Comma as an operator
  - Comma as a separator
  - Comma operator in place of a semicolon



```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
int a,b,c; //as separator
```

```
a=(5,15,76);//as operator
```

```
b=45;c=85;
```

```
cout <<a<<endl,cout <<b<<endl, cout <<c<<endl;//as semicolon
```

```
return 0;
```

```
}
```

**Output:**

76

45

85

## **b. Type Cast Operator**

- Typecasting in C++ is the process of converting one data type to another data type by the programmer using the casting operator during program design.
- In C++ there are two major types to perform type casting.
  - ✓ Implicit type casting
  - ✓ Explicit type casting

## ✓ **Implicit type casting**

- When the type conversion is performed automatically by the compiler without programmers intervention, such type of conversion is known as implicit type conversion or type promotion.
- Implicit type conversion happens automatically when a value is copied to its compatible data type.
- If the operands are of two different data types, then an operand having lower data type is automatically converted into a higher datatype. It is automatically done by the compiler by converting smaller data type into a larger data type.
- bool -> char -> short int -> int -> unsigned int -> long -> unsigned -> long long -> float -> double -> long double

```
#include <iostream>

using namespace std;

int main()
{
    int a=10; float b=21.92;

    int x=a+b;

    float y=a+b;

    cout <<x<<endl;

    cout <<y<<endl;

    return 0;

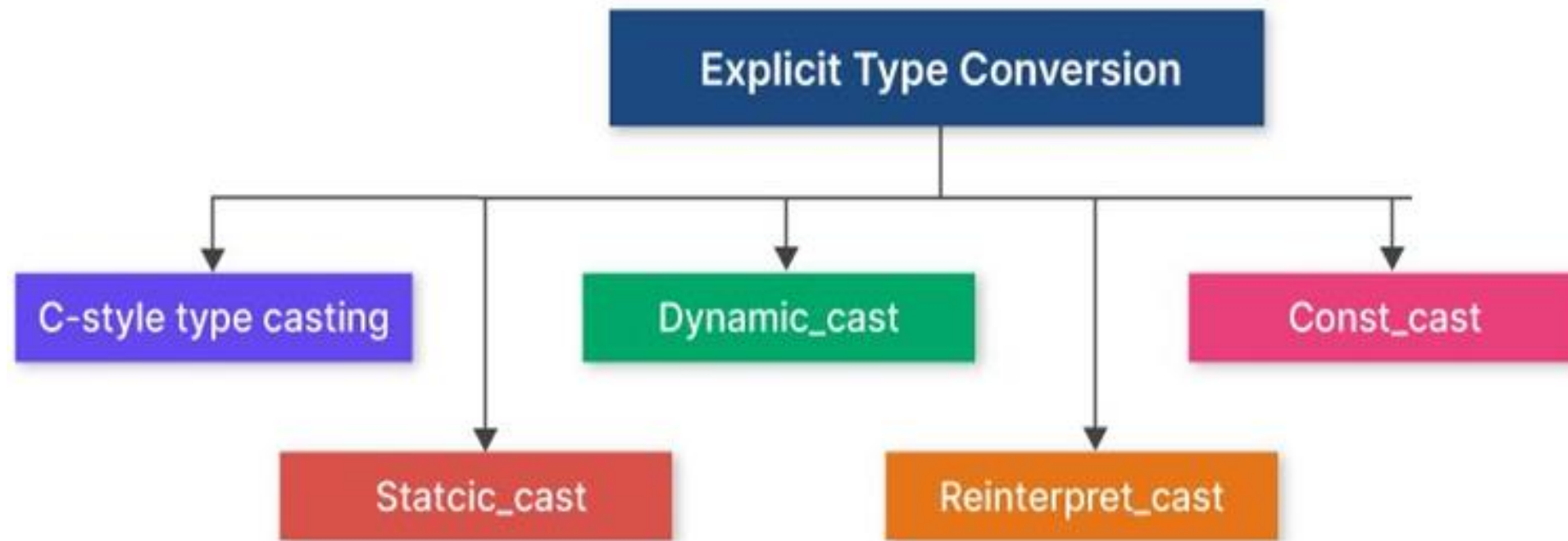
}
```

**Output:**

31

31.92

- ✓ **Explicit type casting**
- In this conversion, the user can define the type to which the expression is to be converted. It can be a larger or smaller data type.
- Explicit type conversion can be achieved by using the cast operator in C++.



## c style casing:

### Syntax:

*(data\_type) Expression;*

```
#include <iostream>
using namespace std;
int main()
{
    int a=10; int b=21;
    float y=b/a;
    float z=float(b)/a;
    cout <<y<<endl;
    cout <<z<<endl;
    return 0;
}
```

### **Output:**

2

2.1

### c. Reference Operator & Dereference Operator

- **Referencing** means taking the address of an existing variable (using &) to set a pointer variable. In order to be valid, a pointer has to be set to the address of a variable of the same type as the pointer.
- For e.g., if we write “&x”, it will return the address of the variable ‘x’.
- **Dereferencing** a pointer means using the \* operator (asterisk character) to retrieve the value from the memory address that is pointed by the pointer.
- For e.g., if we write “\*p”, it will return the value of the variable pointed by the pointer “p”.

```
#include<iostream>

using namespace std;

int main ()
{
    int x;

    int * ptr;

    ptr = &x;

    *ptr = 5;

    cout << x<<endl;

    cout << ptr<<endl;

    cout << *ptr<<endl;

    return 0;

}
```

**Output:**

```
5
0x61ff08
5
```

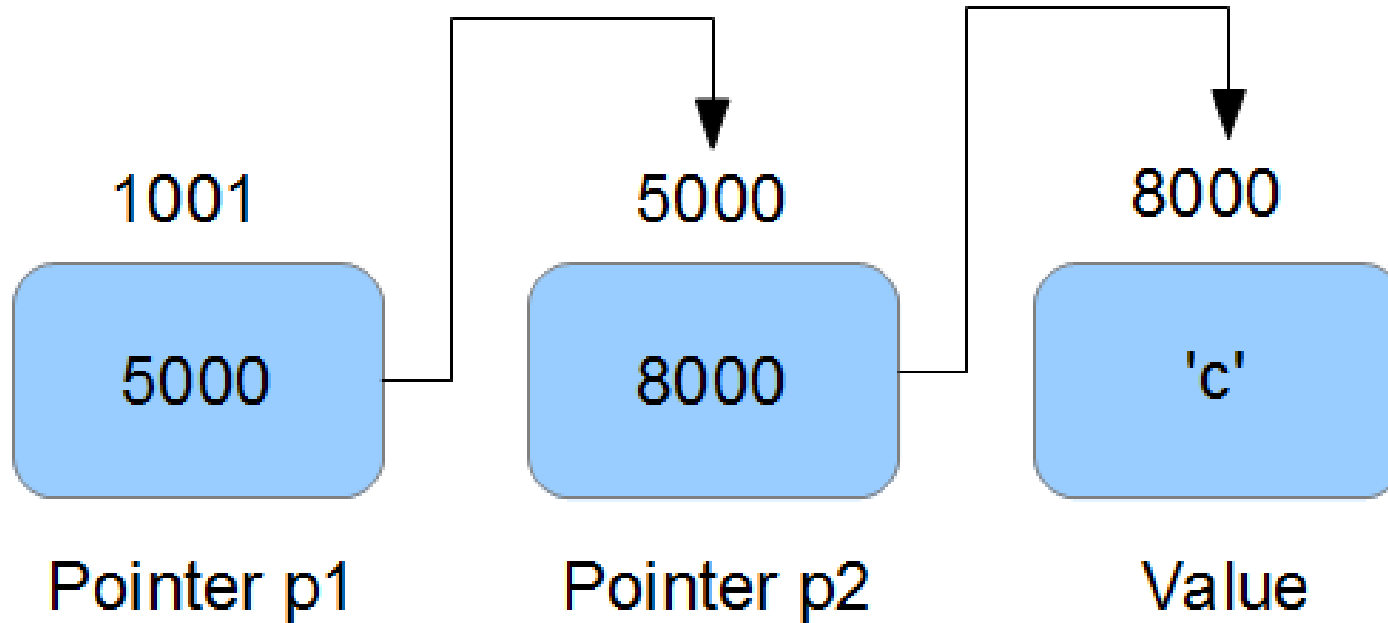


## e. Double Pointer (Pointer to Pointer)

- The pointer to a pointer in C++ is used when we want to store the address of another pointer.
- The first pointer is used to store the address of the variable.
- And the second pointer is used to store the address of the first pointer.

### Syntax:

*data\_type\_of\_pointer \*\*name\_of\_variable = & normal\_pointer\_variable;*



```
#include<iostream>
using namespace std;
int main ()
{
    int x;
    int * ptr1;
    ptr1 = &x;
    int **ptr2=&ptr1;
    *ptr1 = 5;
    cout << x<<endl;
    cout << ptr1<<endl;
    cout << ptr2<<endl;
    cout << *ptr1<<endl;
    cout << *ptr2<<endl;
    cout << **ptr2<<endl;
    return 0;
}
```

### Output:

5

0x61ff08

0x61ff04

5

0x61ff08

5

## f. sizeof

- It is a compile-time unary operator and used to compute the size of its operand. It returns the size of a variable.
- When sizeof() is used with the data types, it simply returns the amount of memory allocated to that data type.

```
#include<iostream>
```

```
using namespace std;
```

```
int main ()
```

```
{
```

```
    int x=5;
```

```
    double y=2.3582145;
```

```
    cout << sizeof(x)<<endl;
```

```
    cout << sizeof(y)<<endl;
```

```
    return 0;
```

```
}
```

**Output:**

4

8

## Flow Control Statements:

- By default, statements in a C++ program are executed in a sequential order.
- The order in which the program statements are executed is known as '**flow of program control**' or just '**flow of control**'. By default, the program control flows sequentially from top to bottom.
- All the programs that we have developed till now have default flow of control. Many practical situations like decision making, repetitive execution of a certain task, etc. require deviation or alteration from the default flow of program control.
- The default flow of control can be altered by using **flow control statements**. Flow control statements are of two types:
  1. Branching statements
    - i. Selection statements
    - ii. Jump statements
  2. Iteration statements

# 1. Branching Statements

➤ Branching statements are used to transfer the program control from one point to another.

They are categorized as:

**a. *Conditional branching:*** In conditional branching, also known as selection, program control is transferred from one point to another based upon the outcome of a certain condition. The selection statements are:

if statement

if-else statement

switch statement

**b. *Unconditional branching:*** In unconditional branching, also known as jumping, program control is transferred from one point to another without checking any condition. The jump statements are:

goto statement

break statement

continue statement

return statement

## i. Selection Statements

Based upon the outcome of a particular condition, **selection statements** transfer control from one point to another. Selection statements select a statement to be executed among a set of various statements. The selection statements available in C++ are as follows:

1. if statement
2. if-else statement
3. switch statement

### 1. if statement

- Use the if statement to specify a block of code to be executed if a condition is true.

#### **Syntax**

```
if (condition)
{
    // block of code to be executed if the condition is true;
}
```

- If the *if* controlling expression evaluates to true, the statement constituting if body is executed.
- If the *if* controlling expression evaluates to false, if body is skipped and the execution continues from the statement following the if statement.

```
#include<iostream>
using namespace std;
int main()
{
    int mark=75;
    if (mark<80)
    {
        cout<<"Average student with "<<mark<<"% mark";
    }
    if (mark<60)
    {
        cout<<"Poor student with "<<mark<<"% mark";
    }
    return (0);
}
```

**Output:** Average student with 75% mark

## 2. if-else statement

- Most of the problems require one set of actions to be performed if a particular condition is true, and another set of actions to be performed if the condition is false. To implement such a decision, C language provides an if-else statement.

### Syntax

```
if (condition)
{
    // block of code to be executed if the condition is true;
}
else
{
    // block of code to be executed if the condition is false;
}
```



- If the if-else controlling expression evaluates to true, the statement constituting the if body is executed and the else body is skipped.
- If the if-else controlling expression evaluates to false, the if body is skipped and the else body is executed.
- After the execution of the if body or the else body, the execution continues from the statement following the if-else statement.

```
#include<iostream>
using namespace std;
int main()
{
    int age;
    cout<<"Enter the age:-";
    cin>>age;
    if (age>=18)
    {
        cout<<"Eligible for voting";
    }
    else
    {
        cout<<"Not eligible for voting";
    }
    return (0);
} Dr. J. R. Nayak
```

**Input:** Enter the age:-22

**Output:** Eligible for voting

### 3. Nested if statement

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

#### Syntax

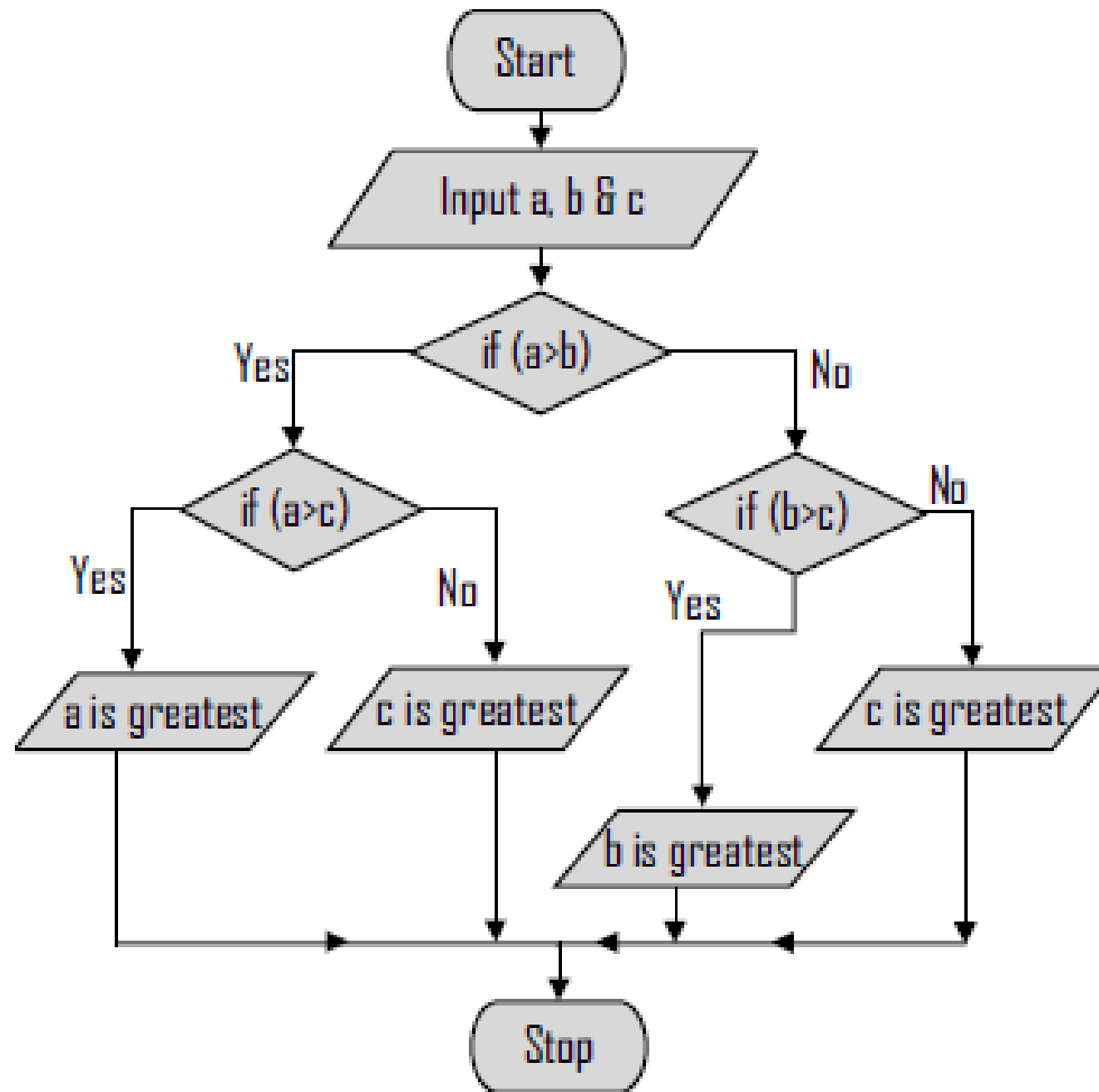
```
if (condition)
{
    if (condition)
    {
        // block of code to be executed if the condition is true;
    }
}
```

### 3. Nested if-else statement

If the body of the if statement is another if statement or contains another if statement (as shown below), then we say that if's are nested and the statement is known as a nested if statement.

#### Syntax:

```
if (condition1) {  
    /* code to be executed if condition1 is true */  
    if (condition2) {  
        /* code to be executed if condition2 is true */  
    } else {  
        /* code to be executed if condition2 is false */  
    }  
} else {  
    /* code to be executed if condition1 is false */  
}
```



*//Program to find greatest number among  
3 numbers using nested if*

```
#include<iostream>
using namespace std;
int main()
{
    int a, b, c;
    cout<<"Enter three integer numbers:";
    cin>>a>>b>>c;
    if (a>b)
    {
        if (a>c)
        {
            cout<<a<<" is greater number";
        }
        if (a<c)
        {
            cout<<c<<" is greater number";
        }
    }
}
```

```
if (a<b)
{
    if (b>c)
    {
        cout<<b<<" is greater number";
    }
    if (b<c)
    {
        cout<<c<<" is greater number";
    }
}
return (0);
}
```

***Input:*** Enter three integer numbers:5 8 2

***Output:*** 8 is greater number

*//Program to find greatest number among  
3 numbers using nested if else*

```
#include<iostream>
using namespace std;
int main()
{
    int a, b, c;
    cout<<"Enter three integer numbers:";
    cin>>a>>b>>c;
    if (a>b)
    {
        if (a>c)
        {
            cout<<a<<" is greater number";
        }
        else
        {
            cout<<c<<" is greater number";
        }
    }
}
```

```
else
{
    if (b>c)
    {
        cout<<b<<" is greater number";
    }
    if (b<c)
    {
        cout<<c<<" is greater number";
    }
}
return (0);
}
```

***Input:*** Enter three integer numbers:5 8 2

***Output:*** 8 is greater number

## Questions of if-else statement.

1. Write a C++ program that takes two integers as input and uses an if-else statement to determine which one is larger.
2. How can you use if-else statements in C++ to check if a number is even or odd? Provide an example.
3. How would you create a C++ program to check if a year is a leap year or not, using if-else conditions?
4. Create a C++ program that determines whether a character entered by the user is a vowel or a consonant using if-else statements.
5. How can you use if-else logic in C++ to check if a number is positive, negative, or zero? Provide a code example.
6. Explain how you can use if-else statements in CPP to implement a simple calculator that performs addition, subtraction, multiplication, and division based on user input.
7. Write a CPP program that checks if a user-provided number is prime or not using if-else statements.

## 4. If...else if...else Statement

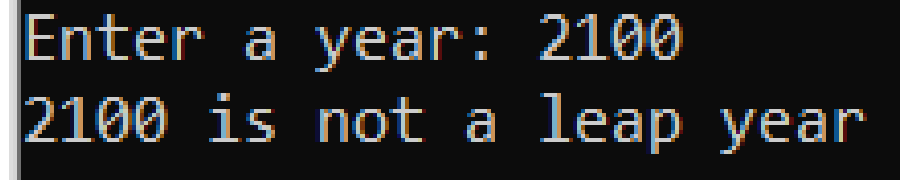
- An **if** statement can be followed by an optional **else if...else** statement, which is very useful to test various conditions using single if...else if statement.
- Once an else if succeeds, none of the remaining else if's or else's will be tested.

### Syntax:

```
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  
}
```



```
#include <iostream>
int main()
{
    int year;
    cout<<"Enter a year: ";
    cin>> year;
    if (year % 400 == 0)
    {
        cout<<year<< " is a leap year."<< endl;
    }
    else if (year % 100 == 0)
    {
        cout<<year<< " is not a leap year."<< endl;
    }
    else if (year % 4 == 0)
    {
        cout<<year<< " is a leap year."<< endl;
    }
    else
    {
        cout<<year<< " is not a leap year."<< endl;
    }
    return 0;
}
```



```
Enter a year: 2100
2100 is not a leap year
```

### 3. Switch-case statement:

- A **switch statement** is used to control complex branching operations.
- The switch statement allows us to execute one code block among many alternatives.
- You can do the same thing with the if...else..if ladder. However, the syntax of the switch statement is much easier to read and write.
- If there is a match, the corresponding statements after the matching label are executed. For example, if the value of the expression is equal to constant2, statements after case constant2: are executed until break is encountered.
- If there is no match, the default statements are executed.
- The default clause inside the switch statement is optional.

## Syntax:

```
switch(expression) {  
  
    case x:  
  
        // code block  
  
        break;  
  
    case y:  
  
        // code block  
  
        break;  
  
    default:  
  
        // code block  
  
}
```

## **Rules for switch statement:**

1. The switch expression must be of an integer or character type.
2. The case value must be an integer or character constant.
3. The case value can be used only inside the switch statement.
4. The break statement in switch case is not must. It is optional. If there is no break statement found in the case, all the cases will be executed present after the matched case. It is known as fall through the state of C++ switch statement.

```
#include<iostream>
using namespace std;
int main()
{
    char op; int a=15; int b=25;
    cout<<"Enter the operator";
    cin>>op;
    switch (op)
    {
        case '+':
            cout<<a+b;
            break;
        case '-':
            cout<<a-b;
            break;
```

```
        case '*':
            cout<<a*b;
            break;
        case '/':
            cout<<float(a)/b;
            break;
        default:
            cout<<"no ope";
    }
    return 0;
}
```

***Input:*** Enter the operator\*

***Output:*** 375

## ii. Jump Statements

- In C++, jump statements are used to jump from one part of the code to another altering the normal flow of the program. They are used to transfer the program control to somewhere else in the program.
- There are 5 types of jump statements in C++:
  - a. break
  - b. continue
  - c. goto
  - d. exit
  - e. return

We will discuss jump statements after iteration statements.

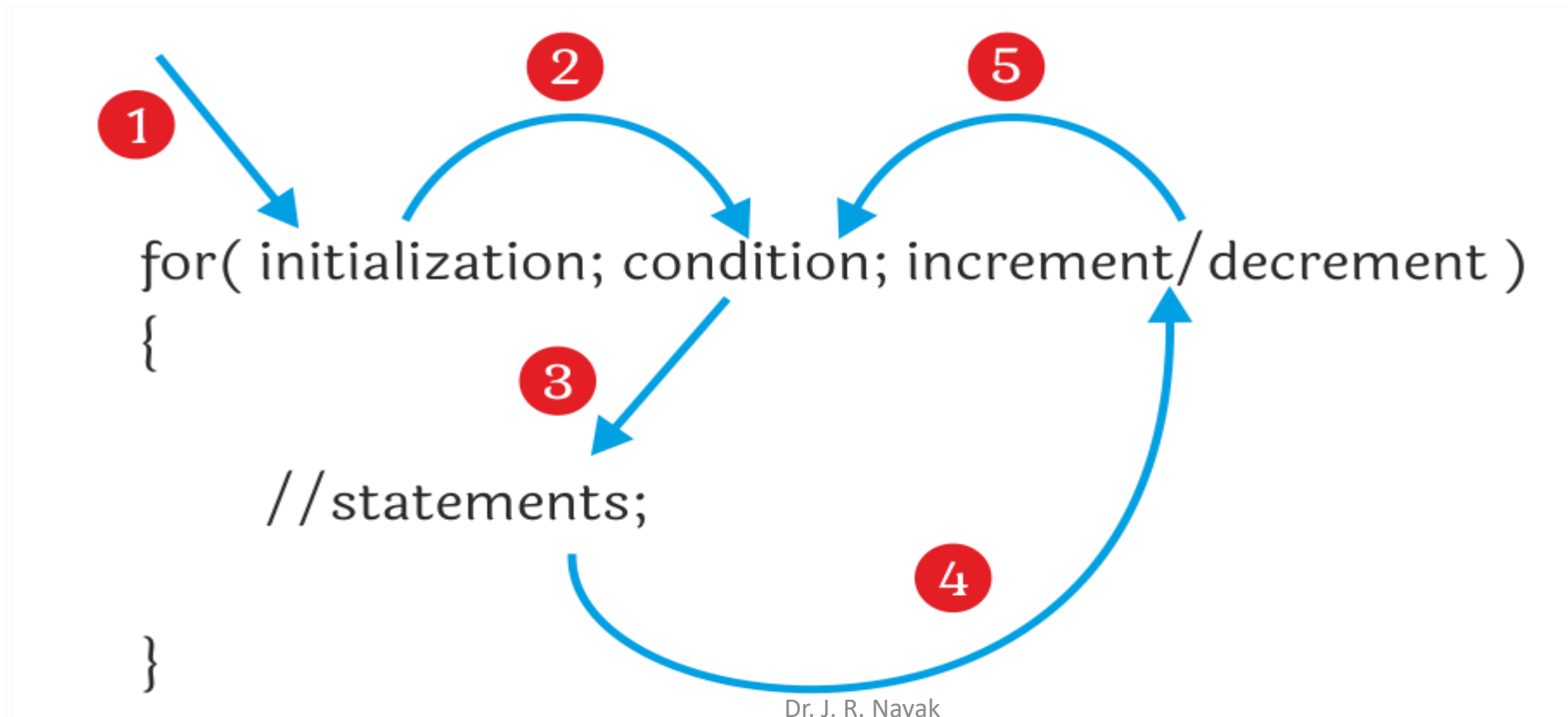
## 2. Iteration/Loop statements

- Iteration is a process of repeating the same set of statements again and again until the specified condition holds true.
- In C++, every loop has a controlling expression.
- Each time the loop body is executed (an iteration of the loop), the controlling expression is evaluated.
- If the expression is true (has a value that's not zero) the loop continues to execute.
- There are three types of iteration statements:
  - a. for
  - b. while
  - c. do-while

## a. for Loop

- The for loop in C++ language is used to iterate the statements or a part of the program several times.
- When you know exactly how many times you want to loop through a block of code, use for loop.
- It is an entry controlled loop.

### Syntax:





**Initialization** is executed (one time) before the execution of the code block. It is optional.

**Condition** defines the condition for executing the code block.

**Increment/decrement** is executed (every time) after the code block has been executed.

**Examples:**

```
int i;  
for (i = 0; i < 5; i++)  
{  
    cout<< i;  
}
```

```
int i = 0;  
for (i; i < 5; i++)  
{  
    cout<< i;  
}
```

Write a program in C++ to display the first 10 natural numbers and their summation.

```
#include<iostream>

using namespace std;

int main()
{
    int i; int sum=0;
    for(i=1;i<=10;i++)
    {
        cout<<"Number is "<<i;
        sum=sum+i;
        cout<<"Summation is "<<sum<<endl;
    }
    return (0);
}
```

**Output:**

Number is 1Summation is 1  
Number is 2Summation is 3  
Number is 3Summation is 6  
Number is 4Summation is 10  
Number is 5Summation is 15  
Number is 6Summation is 21  
Number is 7Summation is 28  
Number is 8Summation is 36  
Number is 9Summation is 45  
Number is 10Summation is 55

Write a program in C to display a pattern like a right angle triangle using an asterisk. The pattern like :

```
*
**
***
****

#include<iostream>
using namespace std;
int main()
{
    int i,j;
    for(i=1;i<=10;i++)
    {
        for(j=1;j<=i;j++)
        {
            cout<<'*';
        }
        cout<<endl;
    }
    return (0);
}
```

### Output:

```
*
**
***
****
*****
*****
*****
*****
*****
*****
*****
```

Write a C++ program to make such a pattern as a pyramid with asterisk:

```
  *
 * *
* * *
* * * *
```

```
#include<iostream>
using namespace std;
int main()
{
    int i,j,s,k,m;
    cout<<"Enter the size of pyramid: ";
    cin>>s;
    m=s+s-1;
    for(i=1;i<=s;i++)
    {
        for(j=m;j>=1;j--)
        {
            cout<<" ";
        }
        for(k=1;k<=i;k++)
        {
            cout<<"* ";
        }
        cout<<endl;
        m--;
    }
    return (0);
}
```

**Output:**

```
  *
 * *
* * *
* * * *
* * * * *
```

Write a C++ program to make such a pattern as a pyramid with asterisk:

```
#include<iostream>
using namespace std;
int main()
{
    int i,j,s,k,m;
    cout<<"Enter the size of pyramid: ";
    cin>>s;
    m=s+s-1;
    for(i=1;i<=s;i++)
    {
        for(k=i;k<=s-1;k++)
        {
            cout<<" ";
        }
        for(m=1;m<=i;m++)
        {
            cout<<"* ";
        }
        cout<<endl;
    }
    return (0);
}
```

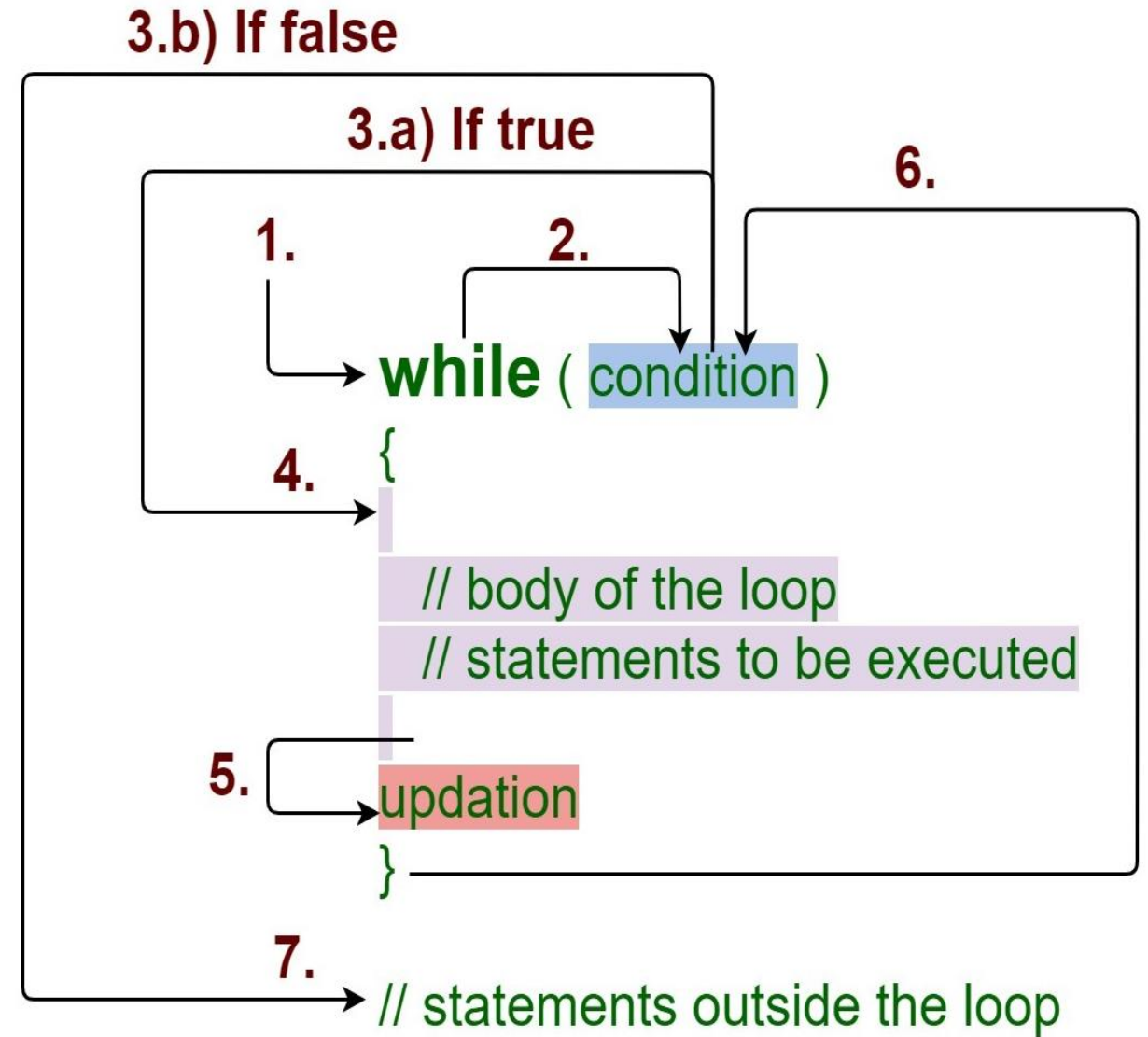
```
      *
     * *
    * * *
   * * * *
```

Enter the size of pyramid:5

```
      *
     * *
    * * *
   * * * *
  * * * * *
```

## b. while:

- In general, a while loop allows a part of the code to be executed multiple times depending upon a given condition.
- It can be viewed as a repeating if statement.
- The while loop is mostly used in the case where the number of iterations is not known in advance.
- Running a while loop without a body is possible.
- It is an entry controlled loop.



## Use of while statement to find the factorial of a number

```
#include<iostream>
using namespace std;
int main()
{
    int i,j;
    int fact=1; i=1;
    cout<<"Enter the number: ";
    cin>>j;
    while (i<=j)
    {
        fact=fact*i;
        i++;
    }
    cout<<"Factorial of "<<j<<" is "<<fact;
    return (0);
}
```

***Input:*** Enter the number: 5

***Output:*** Factorial of 5 is 120

Program to print table for the given number using while loop in C++

```
#include<iostream>
using namespace std;
int main()
{
    int i,j;
    int fact=1; i=1;
    cout<<"Enter the number: ";
    cin>>j;
    while (i<=10)
    {
        cout<<j<<" * "<<i<<" = "<<j*i<<endl;
        i++;
    }
    return (0);
}
```

**Input:** Enter the number: 12

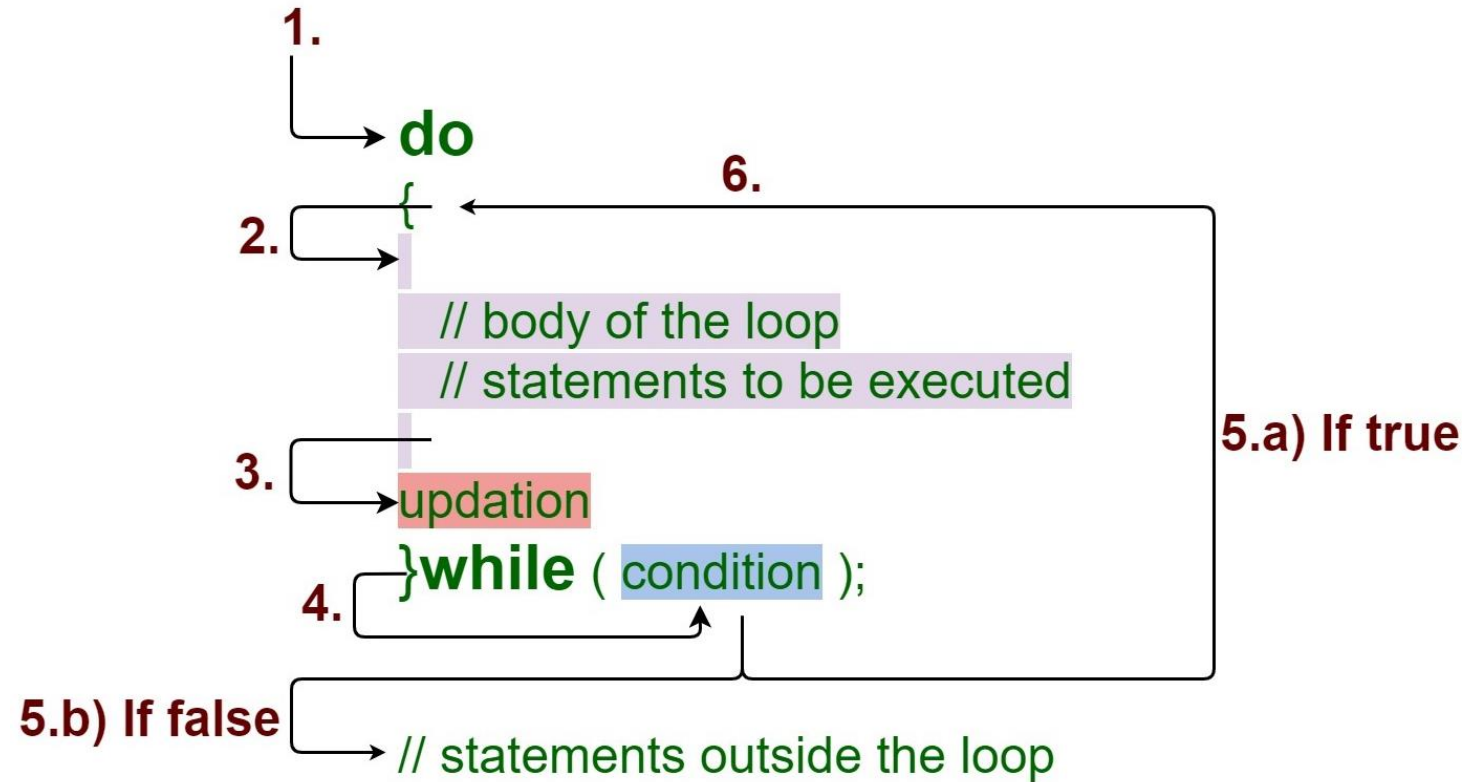
**Output:**

```
12 * 1 = 12
12 * 2 = 24
12 * 3 = 36
12 * 4 = 48
12 * 5 = 60
12 * 6 = 72
12 * 7 = 84
12 * 8 = 96
12 * 9 = 108
12 * 10 = 120
```



## b. do while:

- The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.
- The do-while loop is mainly used in the case where we need to execute the loop at least once.
- It is an exit controlled loop.



## Program to add numbers until the user enters zero

```
#include<iostream>
using namespace std;
int main()
{
    int j;
    int sum=0;
    do
    {
        cout<<"Enter the value:";
        cin>>j;
        sum=sum+j;
    }while (j!=0);
    cout<<"Summation is:"<<sum;
    return (0);
}
```

### **Input:**

*Enter the value:12*

*Enter the value:5*

*Enter the value:10*

*Enter the value:0*

### **Output:**

*Summation is:27*

Program to print table for the given number using do while loop in C++

```
#include<iostream>
using namespace std;
int main()
{
    int i,j;
    int fact=1; i=1;
    cout<<"Enter the number: ";
    cin>>j;
    do
    {
        cout<<j<<" * "<<i<<" = "<<j*i<<endl;
        i++;
    }while (i<=10);
    return (0);
}
```

**Input:** Enter the number: 12

**Output:**

```
12 * 1 = 12
12 * 2 = 24
12 * 3 = 36
12 * 4 = 48
12 * 5 = 60
12 * 6 = 72
12 * 7 = 84
12 * 8 = 96
12 * 9 = 108
12 * 10 = 120
```

1. Write a program in C++ to display the first 10 natural numbers.
2. Write a C++ program to compute the sum of the first 10 natural numbers.
3. Write a program in C++ to display n terms of natural numbers and their sum.
4. Write a program in C++ to display the multiplication table for a given integer.
5. Write a program in C++ to display a pattern like a right angle triangle using an asterisk. The pattern like :

```
*
**
***
****
```

6. Write a C++ program to display a pattern like a right angle triangle with a number. The pattern like :

```
1
12
123
1234
```

7. Write a program in C++ to make a pyramid pattern with numbers increased by 1.

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

8. Write a C++ program to make such a pattern as a pyramid with an asterisk.
9. Write a C++ program to check whether a given number is an Armstrong number or not
10. Write a C++ program to display Pascal's triangle.
11. Write a C++ program to check whether a number is a palindrome or not.
12. Write a C++ program to find the HCF (Highest Common Factor) of two numbers.

```

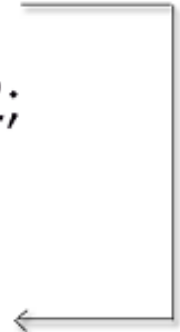
          *
        ***
       *****
      *********
     **********
    **********
   **********
  **********
 **********
*

```


## a. break

- The break statement is used to terminate the loop or switch statement it is contained within. It allows the program to exit the loop or switch and continue executing the next statement after the loop or switch.
- The break statement shall appear only in the “switch” or “loop”. The statements appearing after the break in the loop will be skipped.
- If we use the break statement inside of a nested loop, the break statement will first break the inner loop. So, it won't break all the loops.

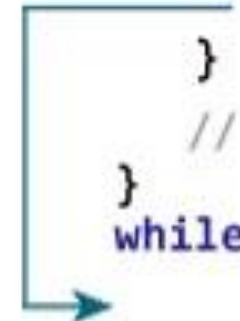
```
loop(condition)  
{  
    statement1;  
    if(condition)  
        break;  
    statement2;  
}  
statement3;
```



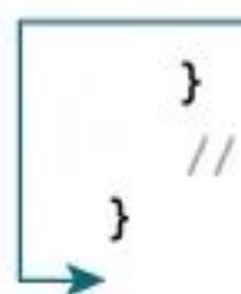
```
while (testExpression) {  
    // codes  
    if (condition to break) {  
        break;  
    }  
    // codes  
}
```



```
do {  
    // codes  
    if (condition to break) {  
        break;  
    }  
    // codes  
} while (testExpression);
```



```
for (init; testExpression; update) {  
    // codes  
    if (condition to break) {  
        break;  
    }  
    // codes  
}
```



## Example

```
#include<iostream>
using namespace std;
int main()
{
    int i=1;
    for(i;i<=10;i++)
    {
        if (i==5)
        {
            break;
        }
        cout<<i;
    }
    cout<<endl<<"Complete";
    return (0);
}
```

**Output:**

1234

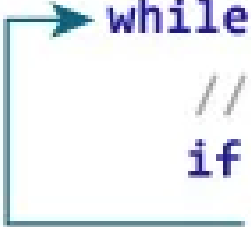
Complete

## **b. Continue**

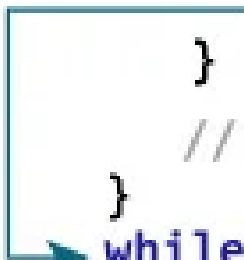
- The continue statement breaks one iteration (in the loop), if a specified condition occurs, and continues with the next iteration in the loop.
- We can use the continue statement in the while loop, for loop, or do..while loop to alter the normal flow of the program execution. Unlike break, it cannot be used with a C++ switch case.




```
→ while (testExpression) {  
    // codes  
    if (testExpression) {  
        continue;  
    }  
    // codes  
}
```



```
do {  
    // codes  
    if (testExpression) {  
        continue;  
    }  
    // codes  
} → while (testExpression);
```



```
for (initilization; condition; modification)  
{  
    ....  
    continue;  
    ....  
}
```



## Example

```
#include<iostream>
using namespace std;
int main()
{
    int i=1;
    for(i;i<=10;i++)
    {
        if (i==5)
        {
            continue;
        }
        cout<<i;
    }
    cout<<endl<<"Complete";
    return (0);
}
```

**Output:**  
1234678910  
Complete

### c. **goto**

- The goto statement is used to jump from one line to another line in the program. Using goto statement we can jump from top to bottom or bottom to top.
- To jump from one line to another line, the goto statement requires a label. Label is a name given to the instruction or line in the program.
- When we use a goto statement in the program, the execution control directly jumps to the line with the specified label.
- The goto statement can be used with any statement like if, switch, while, do-while, and for, etc.

### Forward jump

goto label;

.....

.....

label: ←

statement;

### Backward jump

label: ←

.....

.....

goto label;

statement;

## Example

```
#include<iostream>

using namespace std;

int main()
{
    int i=1;
    label:
    cout<<i<<endl;
    ++i;
    if(i<=10)
    {
        goto label;
    }
    return (0);
}
```

### **Output:**

1  
2  
3  
4  
5  
6  
7  
8  
9  
10

## d. exit

- In C++, exit() terminates the calling process without executing the rest code which is after the exit() function call. It is defined in the <stdlib.h> header file.

### Syntax:

```
void exit(int exit_code);
```

### Example-1

```
#include<iostream>
using namespace std;
int main()
{
    cout<<"First sentence";
    exit(0);
    cout<<"Second sentence";
    return (0);
}
```

**Output:** First sentence

## Example-2

```
#include<iostream>
using namespace std;
int main()
{
    int i=1;
    for(i;i<=10;i++)
    {
        if (i==5)
        {
            exit(0);
        }
        cout<<i;
    }
    cout<<endl<<"Complete";
    return (0);
}
```

**Output: 1234**

## e. return

- A return statement ends the execution of a function, and returns control to the calling function.
- We can only return a single value from a function using return statement. To return multiple values, we can use pointers, structures, classes, etc.

```
int main()
{
    int n = adder(25, 17);
    printf("adder's result is = %d", n);
}

int adder(int a, int b)
{
    int c = a + b;
    return c;
}
```



```
#include<iostream>

using namespace std;

int add(int a, int b)
{
    int c=a+b;
    return c;
}

int main()
{
    int d=add(21,14);
    cout<<"Sum is "<<d;
    return (0);
}
```

**Output:** Sum is 35

Thank You