C++ Programming

Index:

1.Basics of C++ Programming

1.1 Introduction-

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

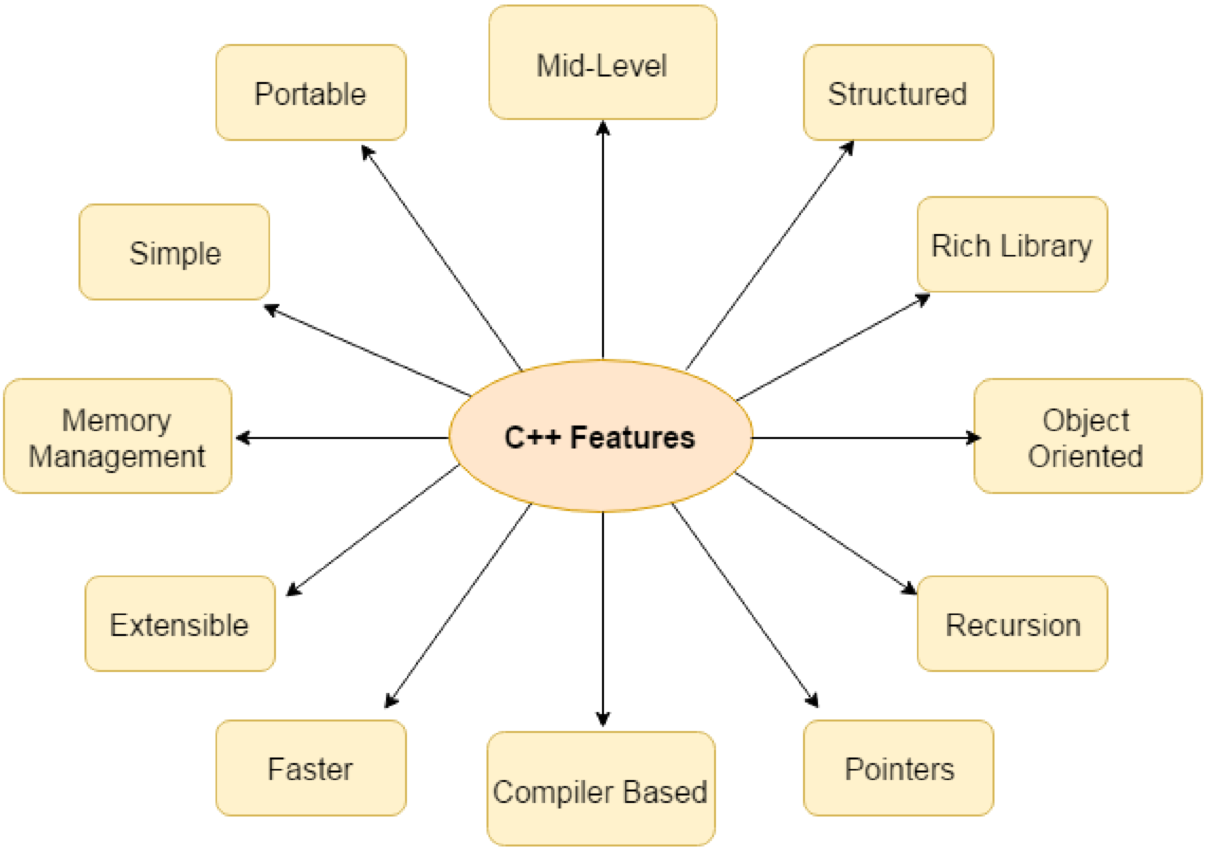
Why to Learn C++

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

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

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

1.2 Features:



1) Simple

C++ is a simple language in the sense that it provides structured approach (to break the problem into parts), rich set of library functions, data types etc.

2) Machine Independent or Portable

Unlike assembly language, c programs can be executed in many machines with little bit or no change. But it is not platform-independent.

3) Mid-level programming language

C++ is also used to do low level programming. It is used to develop system applications such as kernel, driver etc. It also supports the feature of high level language. That is why it is known as mid-level language.

4) Structured programming language

C++ is a structured programming language in the sense that we can break the program into parts using functions. So, it is easy to understand and modify.

5) Rich Library

C++ provides a lot of inbuilt functions that makes the development fast.

6) Memory Management

It supports the feature of dynamic memory allocation. In C++ language, we can free the allocated memory at any time by calling the free() function.

7) Speed

The compilation and execution time of C++ language is fast.

8) Pointer

C++ provides the feature of pointers. We can directly interact with the memory by using the pointers. We can use pointers for memory, structures, functions, array etc.

9) Recursion

In C++, we can call the function within the function. It provides code reusability for every function.

10) Extensible

C++ language is extensible because it can easily adopt new features.

11) Object Oriented

C++ is object oriented programming language. OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.

12) Compiler based

C++ is a compiler based programming language, it means without compilation no C++ program can be executed. First we need to compile our program using compiler and then we can execute our program.

1.3 Setup:

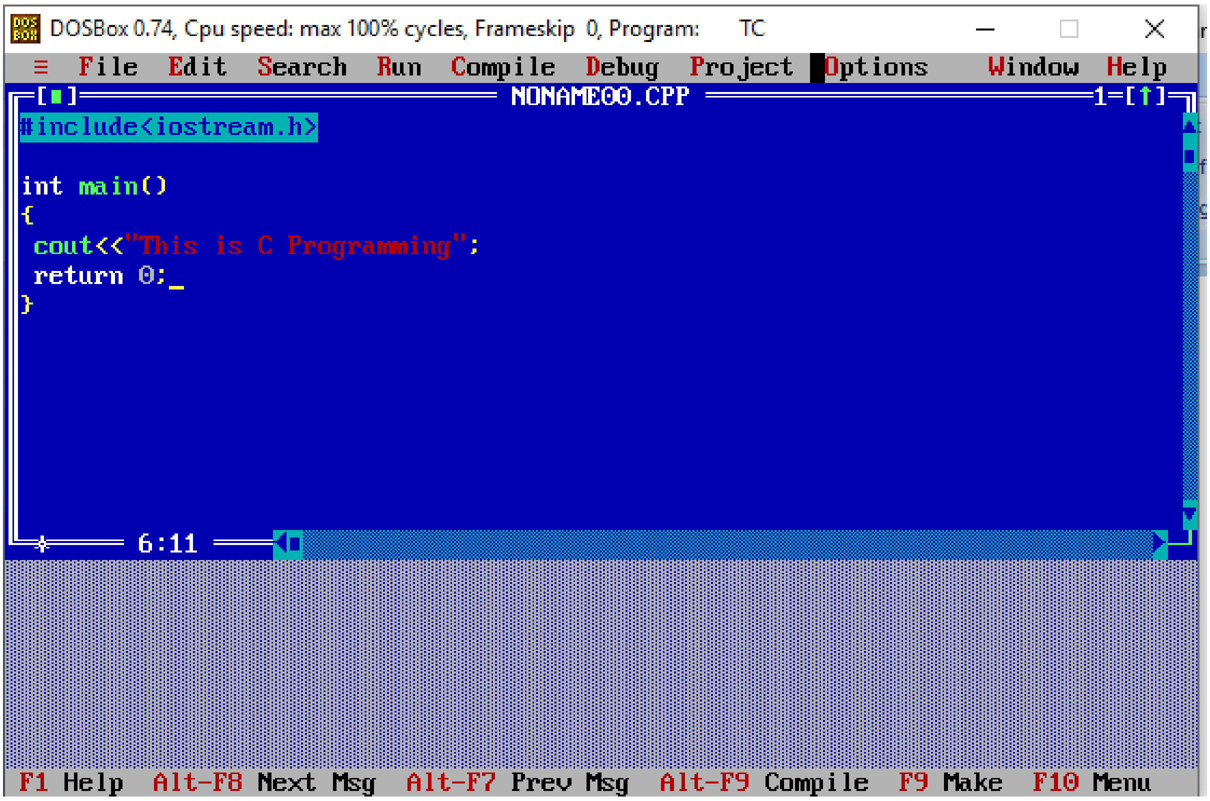
1) Download link of Turbo C- https://developerinsider.co/download-turbo-c-for-windows-7-8-8-1-and-windows-10-32-64-bit-full-screen/
2) Extract zip file
3) Run setup.exe file
4) Click next and accept terms and condition
5) Click Ok to open

Steps:-

1) File > New

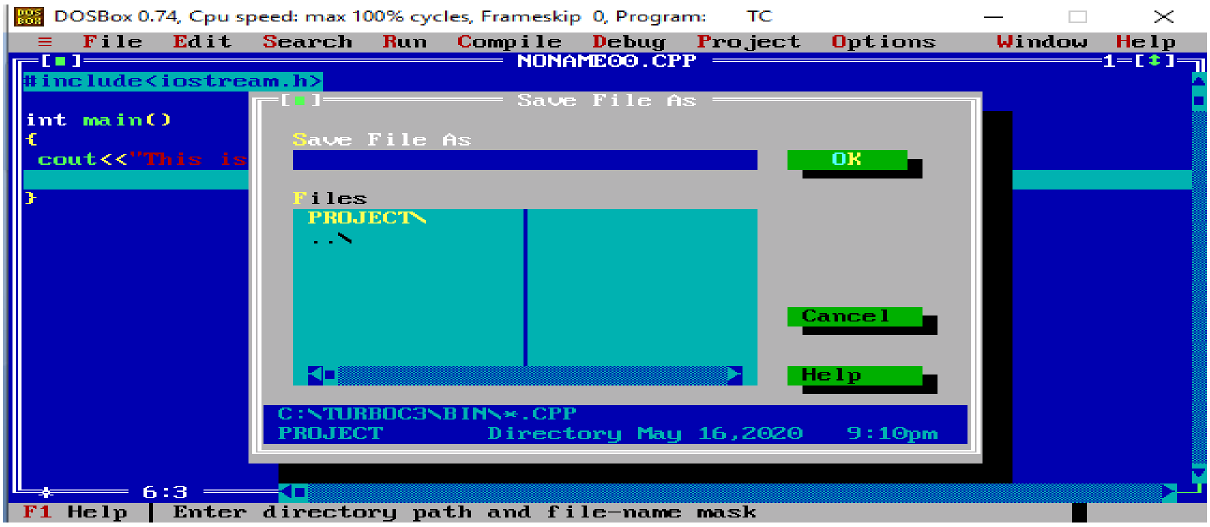
Write the following program –

```
#include <iostream>

using namespace std;

int main() {

   // prints the string enclosed in double quotes

   cout << "This is C++ Programming";

   return 0;

}
```


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC
≡ File Edit Search Run Compile Debug Project Options Window Help
[■] NONAME00.CPP 1=[↑]
6:11
F1 Help Alt-F8 Next Msg Alt-F7 Prev Msg Alt-F9 Compile F9 Make F10 Menu

```
#include<iostream.h>

int main()
{
 cout<<"This is C Programming";
 return 0;_
}
```

2) Save the program using F2 (OR file > Save), remember the extension should be ".c". In the below screenshot I have given the name as helloworld.cpp.

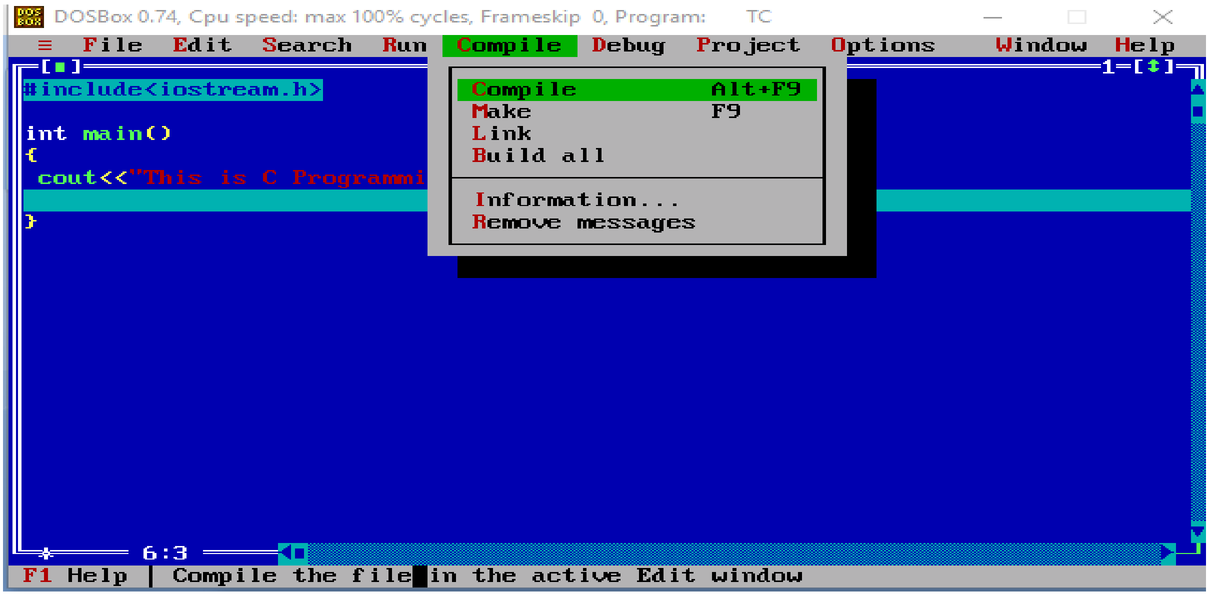

DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC
≡ File Edit Search Run Compile Debug Project Options Window Help
[■] NONAME00.CPP 1=[↕]
[■] Save File As
Save File As
OK
Files
PROJECT\
..\
Cancel
Help
6:3

```
#include<iostream.h>

int main()
{
 cout<<"This is
```

```
C:\TURBOC3\BIN\*.CPP
PROJECT          Directory May 16,2020    9:10pm
```

F1 Help | Enter directory path and file-name mask

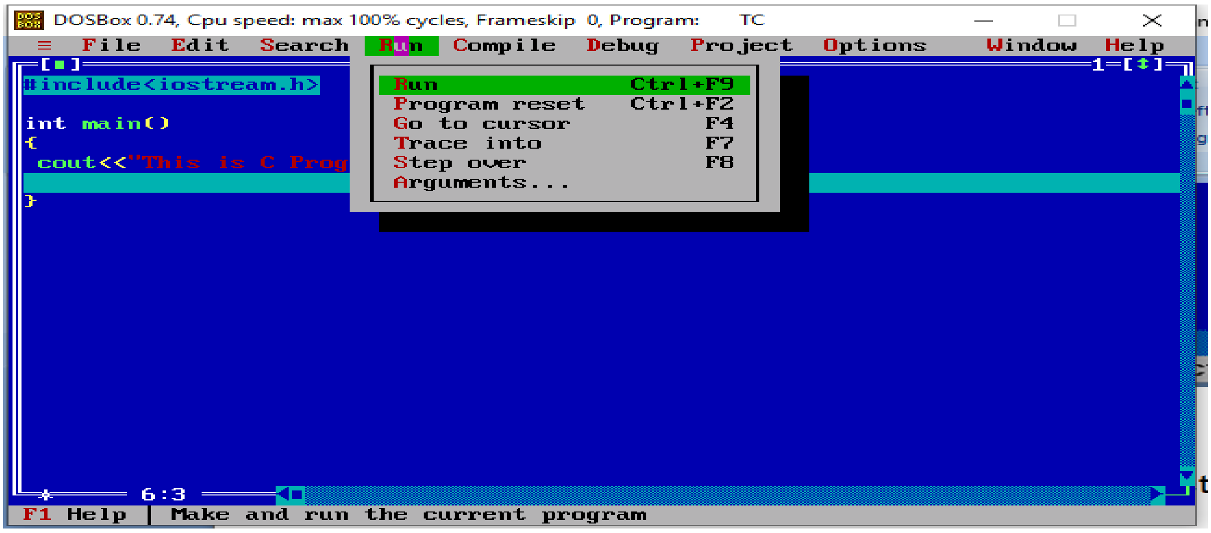
3) Compile the program using Alt + F9 OR Compile > Compile (as shown in the below screenshot).


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC
≡ File Edit Search Run Compile Debug Project Options Window Help
Compile Alt+F9
Make F9
Link
Build all
Information...
Remove messages
6:3

```
#include<iostream.h>

int main()
{
 cout<<"This is C Programmi
}
```

F1 Help | Compile the file in the active Edit window

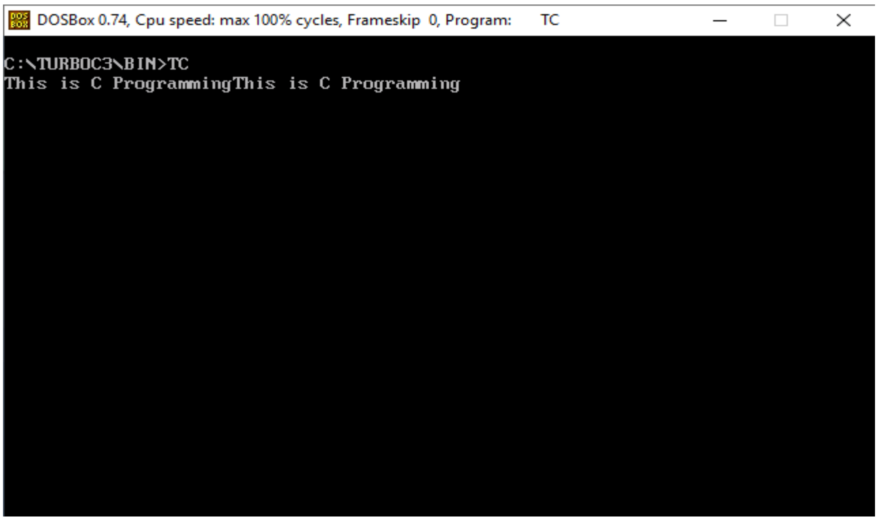
4) Press Ctrl + F9 to Run (or select Run > Run in menu bar) the C program.


DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC
≡ File Edit Search Run Compile Debug Project Options Window Help
Run Ctrl+F9
Program reset Ctrl+F2
Go to cursor F4
Trace into F7
Step over F8
Arguments...
6:3

```
#include<iostream.h>

int main()
{
 cout<<"This is C Prog
}
```

F1 Help | Make and run the current program

5) Alt+F5 to view the output of the program at the output screen.



DOSBox 0.74, Cpu speed: max 100% cycles, Frameskip 0, Program: TC

```
C:\TURBOC3\BIN>TC
This is C ProgrammingThis is C Programming
```

Explaination of Program:

1) We first include the iostream header file that allows us to display output.
2) The cout object is defined inside the std namespace. To use the std namespace, we used the using namespace std; statement.
3) Every C++ program starts with the main() function. The code execution begins from the start of the main() function.
4) cout is an object that prints the string inside quotation marks " ". It is followed by the << operator.
5) return 0; is the "exit status" of the main() function. The program ends with this statement, however, this statement is not mandatory.

1.4 C vs C++

No.	C	C++
1)	C follows the **procedural style programming.**	C++ is multi-paradigm. It supports both **procedural and object oriented.**
2)	Data is less secured in C.	In C++, you can use modifiers for class members to make it inaccessible for outside users.
3)	C follows the **top-down approach.**	C++ follows the **bottom-up approach.**
4)	C does not support function overloading.	C++ supports function overloading.
5)	In C, you can't use functions in structure.	In C++, you can use functions in structure.
6)	C does not support reference variables.	C++ supports reference variables.

7)	In C, **scanf() and printf()** are mainly used for input/output.	C++ mainly uses stream **cin and cout** to perform input and output operations.
8)	Operator overloading is not possible in C.	Operator overloading is possible in C++.
9)	C programs are divided into **procedures and modules**	C++ programs are divided into **functions and classes.**
10)	C does not provide the feature of namespace.	C++ supports the feature of namespace.
11)	Exception handling is not easy in C. It has to perform using other functions.	C++ provides exception handling using Try and Catch block.
12)	C does not support the inheritance.	C++ supports inheritance.

2.Data Types

2.1 C++ Variables

In programming, a variable is a container (storage area) to hold data.

To indicate the storage area, each variable should be given a unique name (identifier). For example,

```
int age = 14;
```

Here, age is a variable of the int data type, and we have assigned an integer value 14 to it.

Note: The int data type suggests that the variable can only hold integers. Similarly, we can use the double data type if we have to store decimals and exponentials.

The value of a variable can be changed, hence the name variable.

```
int age = 14;   // age is 14

age = 17;       // age is 17
```

Rules for naming a variable

1) A variable name can only have alphabets, numbers and the underscore _.
2) A variable name cannot begin with a number.
3) Variable names cannot begin with an uppercase character.
4) A variable name cannot be a keyword. For example, int is a keyword that is used to denote integers.
5) A variable name can start with an underscore. However, it's not considered a good practice.

Note: We should try to give meaningful names to variables. For example, first_name is a better variable name than fn.

2.2 C++ Literals

Literals are data used for representing fixed values. They can be used directly in the code. For example: 1, 2.5, 'c' etc.

Here, 1, 2.5 and 'c' are literals. Why? You cannot assign different values to these terms.

Here's a list of different literals in C++ programming.

1. Integers

An integer is a numeric literal(associated with numbers) without any fractional or exponential part. There are three types of integer literals in C programming:

decimal (base 10)

octal (base 8)

hexadecimal (base 16)

For example:

Decimal: 0, -9, 22 etc

Octal: 021, 077, 033 etc

Hexadecimal: 0x7f, 0x2a, 0x521 etc

In C++ programming, octal starts with a 0, and hexadecimal starts with a 0x.

2. Floating-point Literals

A floating-point literal is a numeric literal that has either a fractional form or an exponent form. For example:

-2.0

0.0000234

-0.22E-5

Note: E-5 = 10-5

3. Characters

A character literal is created by enclosing a single character inside single quotation marks. For example: 'a', 'm', 'F', '2', '}' etc.

4. Escape Sequences

Sometimes, it is necessary to use characters that cannot be typed or has special meaning in C++ programming. For example, newline (enter), tab, question mark, etc.

In order to use these characters, escape sequences are used.

Escape Sequences	Characters
\b	Backspace
\f	Form feed
\n	Newline
\r	Return
\t	Horizontal tab
\v	Vertical tab
\\	Backslash
\'	Single quotation mark
\"	Double quotation mark
\?	Question mark
\0	Null Character

5. String Literals

A string literal is a sequence of characters enclosed in double-quote marks. For example:

"good"	string constant
""	null string constant
" "	string constant of six white space
"x"	string constant having a single character
"Earth is round\n"	prints string with a newline

We will learn about strings in detail in the C++ string tutorial.

2.3 C++ Constants

In C++, we can create variables whose value cannot be changed. For that, we use the const keyword. Here's an example:

```
const int LIGHT_SPEED = 299792458;

LIGHT_SPEED = 2500 // Error! LIGHT_SPEED is a constant.
```

Here, we have used the keyword const to declare a constant named LIGHT_SPEED. If we try to change the value of LIGHT_SPEED, we will get an error.

Data types

The table below shows the fundamental data types, their meaning, and their sizes (in bytes):

Data Type	Meaning	Size (in Bytes)
int	Integer	2 or 4
float	Floating-point	4
double	Double Floating-point	8
char	Character	1
wchar_t	Wide Character	2
bool	Boolean	1
void	Empty	0

1).C++ int

The int keyword is used to indicate integers.

Its size is usually 4 bytes. Meaning, it can store values from -2147483648 to 214748647.

For example,

```
int salary = 85000;
```

2. C++ float and double

float and double are used to store floating-point numbers (decimals and exponentials).

The size of float is 4 bytes and the size of double is 8 bytes. Hence, double has two times the precision of float. To learn more, visit C++ float and double.

For example,

```
float area = 64.74;

double volume = 134.64534;
```

As mentioned above, these two data types are also used for exponentials. For example,

```
double distance = 45E12    // 45E12 is equal to 45*10^12
```

3. C++ char

Keyword char is used for characters.

Its size is 1 byte.

Characters in C++ are enclosed inside single quotes ' '.

For example,

char test = 'h';

Note: In C++, an integer value is stored in a char variable rather than the character itself. To learn more, visit C++ characters.

4. C++ bool

The bool data type has one of two possible values: true or false.

Booleans are used in conditional statements and loops (which we will learn in later chapters).

For example,

bool cond = false;

5. C++ void

The void keyword indicates an absence of data. It means "nothing" or "no value".

We will use void when we learn about functions and pointers.

Note: We cannot declare variables of the void type.

C++ Type Modifiers

We can further modify some of the fundamental data types by using type modifiers. There are 4 type modifiers in C++. They are:

signed

unsigned

short

long

We can modify the following data types with the above modifiers:

int

double

char

Data Type	Size (in Bytes)	Meaning
signed int	4	used for integers (equivalent to int)
unsigned int	4	can only store positive integers
short	2	used for small integers (range **-32768 to 32767**)
long	at least 4	used for large integers (equivalent to long int)
unsigned long	4	used for large positive integers or 0 (equivalent to unsigned long int)
long long	8	used for very large integers (equivalent to long long int).
unsigned long long	8	used for very large positive integers or 0 (equivalent to unsigned long long int)
long double	8	used for large floating-point numbers
signed char	1	used for characters (guaranteed range **-127 to 127**)
unsigned char	1	used for characters (range **0 to 255**)

Example-

```
long b = 4523232;

long int c = 2345342;

long double d = 233434.56343;

short d = 3434233; // Error! out of range

unsigned int a = -5;    // Error! can only store positive numbers or 0
```

3.Input and Output in C

3.1 cout

In C++, cout sends formatted output to standard output devices, such as the screen. We use the cout object along with the << operator for displaying output.

Example 1: String Output

```
#include <iostream>
using namespace std;
int main() {
    // prints the string enclosed in double quotes
    cout << "This is C++ Programming";
    return 0;
}
```

Output

```
This is C++ Programming
```

Note: If we don't include the using namespace std; statement, we need to use std::cout instead of cout.

```
#include <iostream>
int main() {
    // prints the string enclosed in double quotes
    std::cout << "This is C++ Programming";
    return 0;
}
```

To print the numbers and character variables, we use the same cout object but without using quotation marks.

```
#include <iostream>
using namespace std;
int main() {
    int num1 = 70;
    double num2 = 256.783;
    char ch = 'A';
    cout << num1 << endl;    // print integer
    cout << num2 << endl;    // print double
    cout << "character: " << ch << endl;    // print char
    return 0;
}
```

Output

```
70
256.783
character: A
```

Notes:

The endl manipulator is used to insert a new line. That's why each output is displayed in a new line.

The << operator can be used more than once if we want to print different variables, strings and so on in a single statement. For example:

```
cout << "character: " << ch << endl;
```

3.2 cin

In C++, cin takes formatted input from standard input devices such as the keyboard. We use the cin object along with the >> operator for taking input.

Example 3: Integer Input/Output

```
#include <iostream>
using namespace std;
int main() {
    int num;
    cout << "Enter an integer: ";
    cin >> num;   // Taking input
    cout << "The number is: " << num;
    return 0;
}
```

Output

```
Enter an integer: 70

The number is: 70
```

In the program, we used

```
cin >> num;
```

to take input from the user. The input is stored in the variable num. We use the >> operator with cin to take input.

Note: If we don't include the using namespace std; statement, we need to use std::cin instead of cin.

C++ Taking Multiple Inputs

```
#include <iostream>

using namespace std;

int main() {

   char a;

   int num;

   cout << "Enter a character and an integer: ";

   cin >> a >> num;

   cout << "Character: " << a << endl;

   cout << "Number: " << num;

   return 0;

}
```

Output

```
Enter a character and an integer: F

23

Character: F

Number: 23
```

3.3 Type Conversion

C++ allows us to convert data of one type to that of another. This is known as type conversion.

There are two types of type conversion in C++.

1) Implicit Conversion
2) Explicit Conversion (also known as Type Casting)

1) Implicit Type Conversion

The type conversion that is done automatically done by the compiler is known as implicit type conversion. This type of conversion is also known as automatic conversion.

Let us look at two examples of implicit type conversion.

Example 1: Conversion From int to double

```
// Working of implicit type-conversion


#include <iostream>

using namespace std;

int main() {

  // assigning an int value to num_int

  int num_int = 9;

  // declaring a double type variable

  double num_double;

  // implicit conversion

  // assigning int value to a double variable

  num_double = num_int;

  cout << "num_int = " << num_int << endl;

  cout << "num_double = " << num_double << endl;

  return 0;

}
```

Output

```
num_int = 9

num_double = 9
```

Example 2: Automatic Conversion from double to int

```
//Working of Implicit type-conversion

#include <iostream>

using namespace std;

int main() {

   int num_int;

   double num_double = 9.99;

   // implicit conversion

   // assigning a double value to an int variable

   num_int = num_double;

   cout << "num_int = " << num_int << endl;

   cout << "num_double = " << num_double << endl;

   return 0;

}
```

Output

```
num_int = 9

num_double = 9.99
```

2) C++ Explicit Conversion

When the user manually changes data from one type to another, this is known as explicit conversion. This type of conversion is also known as type casting.

The syntax for this style is:

```
(data_type)expression;
```

For example,

```
// initializing int variable

int num_int = 26;

// declaring double variable

double num_double;

// converting from int to double

num_double = (double)num_int;
```

Example:

```
#include <iostream>

using namespace std;

int main() {

    // initializing a double variable

    double num_double = 3.56;

    cout << "num_double = " << num_double << endl;

    // C-style conversion from double to int

    int num_int1 = (int)num_double;

    cout << "num_int1   = " << num_int1 << endl;


    // function-style conversion from double to int

    int num_int2 = int(num_double);

    cout << "num_int2   = " << num_int2 << endl;

    return 0;

}
```

Output

```
num_double = 3.56

num_int1   = 3

num_int2   = 3
```

4.Operators:

Operators are symbols that perform operations on variables and values. For example, + is an operator used for addition, while - is an operator used for subtraction.

Operators in C++ can be classified into 6 types:

1) Arithmetic Operators
2) Assignment Operators
3) Relational Operators
4) Logical Operators
5) Bitwise Operators
6) Other Operators

1. C++ Arithmetic Operators

Arithmetic operators are used to perform arithmetic operations on variables and data. For example,

a + b;

Here, the + operator is used to add two variables a and b. Similarly there are various other arithmetic operators in C++.

Operator	Operation
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo Operation (Remainder after division)

Example 1: Arithmetic Operators

```
#include <iostream>

using namespace std;

int main() {

   int a, b;

   a = 7;

   b = 2;
```

```
    // printing the sum of a and b

    cout << "a + b = " << (a + b) << endl;

    // printing the difference of a and b

    cout << "a - b = " << (a - b) << endl;

    // printing the product of a and b

    cout << "a * b = " << (a * b) << endl;

    // printing the division of a by b

    cout << "a / b = " << (a / b) << endl;

    // printing the modulo of a by b

    cout << "a % b = " << (a % b) << endl;

    return 0;

}
```

Output

```
a + b = 9

a - b = 5

a * b = 14

a / b = 3

a % b = 1
```

Here, the operators +, - and * compute addition, subtraction, and multiplication respectively as we might have expected.

Note: The % operator can only be used with integers.

2)Increment and Decrement Operators

C++ also provides increment and decrement operators: ++ and -- respectively. ++ increases the value of the operand by 1, while -- decreases it by 1.

For example,

```
int num = 5;

// increasing num by 1

++num;
```

Here, the value of num gets increased to 6 from its initial value of 5.

Example 2: Increment and Decrement Operators

```
// Working of increment and decrement operators

#include <iostream>

using namespace std;

int main() {

    int a = 10, b = 100, result_a, result_b;

    // incrementing a by 1 and storing the result in result_a

    result_a = ++a;

    cout << "result_a = " << result_a << endl;

    // decrementing b by 1 and storing the result in result_b

    result_b = --b;

    cout << "result_b = " << result_b << endl;

    return 0;

}
```

Output

```
result_a = 11

result_b = 99
```

In the above program, we used ++ and -- operator as prefixes. We can also use these operators as postfix.

There is a slight difference when these operators are used as a prefix versus when they are used as a postfix.

3. C++ Assignment Operators

In C++, assignment operators are used to assign values to variables. For example,

```
// assign 5 to a

a = 5;
```

Here, we have assigned a value of 5 to the variable a.

Operator	Example	Equivalent to
=	a = b;	a = b;
+=	a += b;	a = a + b;
-=	a -= b;	a = a - b;
*=	a *= b;	a = a * b;
/=	a /= b;	a = a / b;
%=	a %= b;	a = a % b;

Example 2: Assignment Operators

```
#include <iostream>
using namespace std;
int main() {
    int a, b, temp;
    // 2 is assigned to a
    a = 2;
    // 7 is assigned to b
    b = 7;
   // value of a is assigned to temp
   temp = a;    // temp will be 2
   cout << "temp = " << temp << endl;
    // assigning the sum of a and b to a
    a += b;       // a = a +b
    cout << "a = " << a << endl;
    return 0;
}
```

Output

```
temp = 2
a = 9
```

4. C++ Relational Operators

A relational operator is used to check the relationship between two operands. For example,

```
// checks if a is greater than b

a > b;
```

Here, > is a relational operator. It checks if a is greater than b or not.

If the relation is true, it returns 1 whereas if the relation is false, it returns 0.

Operator	Meaning	Example
==	Is Equal To	3 == 5 gives us false
!=	Not Equal To	3 != 5 gives us true
>	Greater Than	3 > 5 gives us false
<	Less Than	3 < 5 gives us true
>=	Greater Than or Equal To	3 >= 5 give us false
<=	Less Than or Equal To	3 <= 5 gives us true

Example 4: Relational Operators

```
#include <iostream>

using namespace std;

int main() {

    int a, b;

    a = 3;

    b = 5;

    bool result;

    result = (a == b);   // false

    cout << "3 == 5 is " << result << endl;

    result = (a != b);  // true

    cout << "3 != 5 is " << result << endl;
    result = a > b;   // false
    cout << "3 > 5 is " << result << endl;
    result = a < b;   // true
    cout << "3 < 5 is " << result << endl;
    result = a >= b;  // false
    cout << "3 >= 5 is " << result << endl;
    result = a <= b;  // true
    cout << "3 <= 5 is " << result << endl;
    return 0;
}
```

Output

```
3 == 5 is 0
3 != 5 is 1
3 > 5 is 0
3 < 5 is 1
3 >= 5 is 0
3 <= 5 is 1
```

Note: Relational operators are used in decision making and loops.

5. C++ Logical Operators

Logical operators are used to check whether an expression is true or false. If the expression is true, it returns 1 whereas if the expression is false, it returns 0.

Operator Example

&& expression1 && expression 2 Logical AND.

True only if all the operands are true.

|| expression1 || expression 2 Logical OR.

True if at least one of the operands is true.

! !expression Logical NOT.

True only if the operand is false.

In C++, logical operators are commonly used in decision making. To further understand the logical operators, let's see the following examples,

Suppose,

a = 5

b = 8

Then,

(a > 3) && (b > 5) evaluates to true

(a > 3) && (b < 5) evaluates to false

(a > 3) || (b > 5) evaluates to true

(a > 3) || (b < 5) evaluates to true

(a < 3) || (b < 5) evaluates to false

!(a == 3) evaluates to true

!(a > 3) evaluates to false

Example 5: Logical Operators

```
#include <iostream>

using namespace std;

int main() {

   bool result;

   result = (3 != 5) && (3 < 5);     // true

   cout << "(3 != 5) && (3 < 5) is " << result << endl;

   result = (3 == 5) && (3 < 5);    // false
```

```
    cout << "(3 == 5) && (3 < 5) is " << result << endl;
    result = (3 == 5) && (3 > 5);    // false
    cout << "(3 == 5) && (3 > 5) is " << result << endl;
    result = (3 != 5) || (3 < 5);    // true
    cout << "(3 != 5) || (3 < 5) is " << result << endl;

    result = (3 != 5) || (3 > 5);    // true
    cout << "(3 != 5) || (3 > 5) is " << result << endl;
    result = (3 == 5) || (3 > 5);    // false
    cout << "(3 == 5) || (3 > 5) is " << result << endl;
    result = !(5 == 2);    // true
    cout << "!(5 == 2) is " << result << endl;
    result = !(5 == 5);    // false
    cout << "!(5 == 5) is " << result << endl;
    return 0;
}
```

Run Code

Output

```
(3 != 5) && (3 < 5) is 1
(3 == 5) && (3 < 5) is 0
(3 == 5) && (3 > 5) is 0
(3 != 5) || (3 < 5) is 1
(3 != 5) || (3 > 5) is 1
(3 == 5) || (3 < 5) is 0
!(5 == 2) is 1
!(5 == 5) is 0
```

Explanation of logical operator program

(3 != 5) && (3 < 5) evaluates to 1 because both operands (3 != 5) and (3 < 5) are 1 (true).

(3 == 5) && (3 < 5) evaluates to 0 because the operand (3 == 5) is 0 (false).

(3 == 5) && (3 > 5) evaluates to 0 because both operands (3 == 5) and (3 > 5) are 0 (false).

(3 != 5) || (3 < 5) evaluates to 1 because both operands (3 != 5) and (3 < 5) are 1 (true).

(3 != 5) || (3 > 5) evaluates to 1 because the operand (3 != 5) is 1 (true).

(3 == 5) || (3 > 5) evaluates to 0 because both operands (3 == 5) and (3 > 5) are 0 (false).

!(5 == 2) evaluates to 1 because the operand (5 == 2) is 0 (false).

!(5 == 5) evaluates to 0 because the operand (5 == 5) is 1 (true).

6. C++ Bitwise Operators

In C++, bitwise operators are used to perform operations on individual bits. They can only be used alongside char and int data types.

Operator	Description
&	Binary AND
|	Binary OR
^	Binary XOR
~	Binary One's Complement
<<	Binary Shift Left
>>	Binary Shift Right

Shift Operators in C programming

There are two shift operators in C programming:

1) Right shift operator
2) Left shift operator.

Right Shift Operator

Right shift operator shifts all bits towards right by certain number of specified bits. It is denoted by >>.

212 = 11010100 (In binary)

212>>2 = 00110101 (In binary) [Right shift by two bits]

212>>7 = 00000001 (In binary)

212>>8 = 00000000

212>>0 = 11010100 (No Shift)

Left Shift Operator

Left shift operator shifts all bits towards left by certain number of specified bits. It is denoted by <<.

212 = 11010100 (In binary)

212<<1 = 110101000 (In binary) [Left shift by one bit]

212<<0 =11010100 (Shift by 0)

212<<4 = 110101000000 (In binary) =3392(In decimal)

7) Ternary operator-

Programmers use the ternary operator for decision making in place of longer if and else conditional statements.

The ternary operator take three arguments:

1)The first is a comparison argument

2) The second is the result upon a true comparison

3) The third is the result upon a false comparison

Syntax

condition ? value_if_true : value_if_false

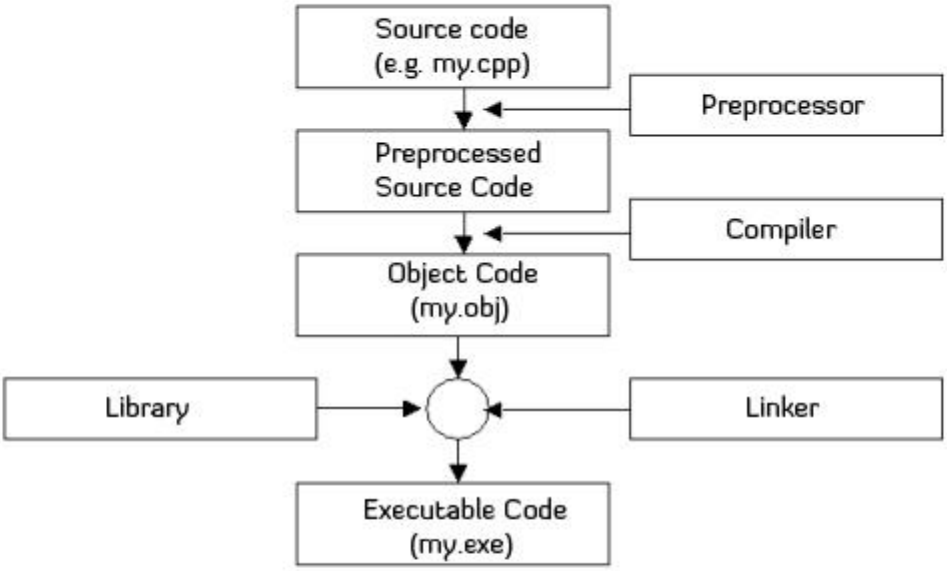
The statement evaluates to value_if_true if condition is met, and value_if_false otherwise.

int a = 10, b = 20, c;

c = (a < b) ? a : b;

count<<c;
is set equal to a, because the condition a < b was true.

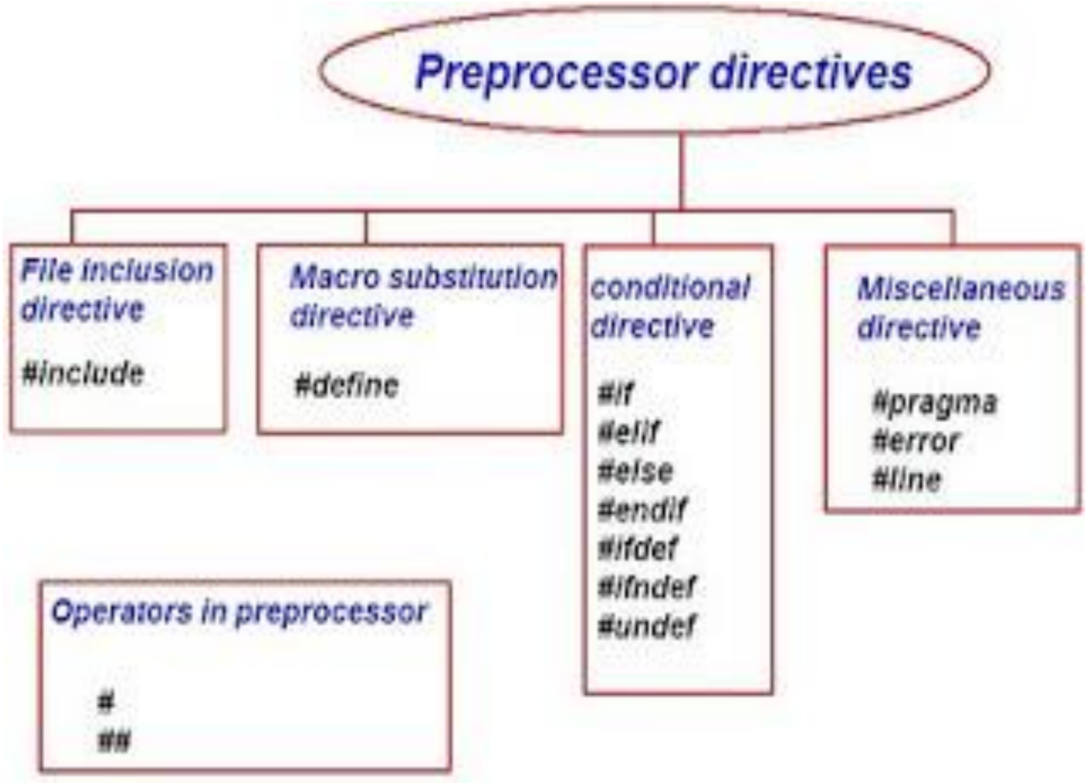
5.C Preprocessor



The C preprocessor is a macro preprocessor (allows you to define macros) that transforms your program before it is compiled. These transformations can be the inclusion of header file, macro expansions etc.

All preprocessing directives begin with a # symbol. For example,

#define PI 3.14



1) **#define:**

The #define preprocessor directive is used to define constant or micro substitution. It can use any basic data type.

```
#define token value
```

Example-

```
#include <iostream>
#define PI 3.14
main() {
  std::cout<<PI;;
}
```

2) **#include-**

The #include preprocessor directive is used to paste code of given file into current file. It is used include system-defined and user-defined header files. If included file is not found, compiler renders error.

By the use of #include directive, we provide information to the preprocessor where to look for the header files. There are two variants to use #include directive.

```
#include <filename>
#include "filename"
```

The #include <filename> tells the compiler to look for the directory where system header files are held. In UNIX, it is \usr\include directory.

Example-

```
#include <iostream>
using namespace std;
 int main(){
   cout<<"Hello C";
   return 0;
 }
```

3) **#undef**

The #undef preprocessor directive is used to undefine the constant or macro defined by #define.

Syntax:
#undef token

Example-

```
#include <iostream>
using namespace std;
#define PI 3.14
#undef PI
main() {
   cout<<PI;
}
```

Output:

```
Compile Time Error: 'PI' undeclared
```

4) **#ifdef**

The #ifdef preprocessor directive checks if macro is defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

```
#ifdef MACRO
//code
#endif
```

Example-

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

```
int a=0;
#ifdef NOINPUT
a=2;
#else
cout<<"Enter a:”;
cin>>&a;
#endif
cout<< a;

   return 0;
}
Output-Value of a: 2
```

5) **#ifndef**

The #ifndef preprocessor directive checks if macro is not defined by #define. If yes, it executes the code otherwise #else code is executed, if present.

Syntax:

```
#ifndef MACRO
//code
#endif
```

```
#include <iostream>
using namespace std;
#define INPUT
int main() {
int a=0;
#ifndef INPUT
a=2;
#else
cout<<"Enter a:";
cin>> a;
#endif
cout<<a;
}
Output:
Enter a:5
Value of a: 5
```

But, if you don't define INPUT, it will execute the code of #ifndef.

6) **#if**

The #if preprocessor directive evaluates the expression or condition. If condition is true, it executes the code otherwise #elseif or #else or #endif code is executed.

Syntax:

```
#if expression
//code
#endif
#include <iostream>
using namespace std;
#define NUMBER 0
int main() {
#if (NUMBER==0)
cout<<NUMBER;
#endif

}
```

Output:

```
Value of Number is: 0
```

7) #else

The #else preprocessor directive evaluates the expression or condition if condition of #if is false. It can be used with #if, #elif, #ifdef and #ifndef directives.

Syntax:

```
#if expression

//if code
#else
//else code
#endif
```

```
#include <iostream>
using namespace std;
#define NUMBER 1
int main() {
#if NUMBER==0
cout<<NUMBER;
#else
cout<<"Value of Number is non-zero";
#endif

}
```

Output:

```
Value of Number is non-zero
```

8) #error

The #error preprocessor directive indicates error. The compiler gives fatal error if #error directive is found and skips further compilation process.

```
#include <iostream>
using namespace std;
#ifndef __MATH_H
#error First include then compile
#else
void main(){
    float a;
    a=sqrt(7);
    cout<<a;
}
#endif
```

Output:Compile Time Error: First include then compile

6 Header files

Let’s have a look at these Header files in C and C++:.

1. #include<stdio.h> (Standard input-output header)

Used to perform input and output operations in C like scanf() and printf().

2. #include<string.h> (String header)

Perform string manipulation operations like strlen and strcpy.

3. #include<conio.h> (Console input-output header)

Perform console input and console output operations like clrscr() to clear the screen and getch() to get the character from the keyboard.

4. #include<stdlib.h> (Standard library header)

Perform standard utility functions like dynamic memory allocation, using functions such as malloc() and calloc().

5. #include<math.h> (Math header)

Perform mathematical operations like sqrt() and pow(). To obtain the square root and the power of a number respectively.

6. #include<ctype.h>(Character type header)

Perform character type functions like isaplha() and isdigit(). To find whether the given character is an alphabet or a digit respectively.

7. #include<time.h>(Time header)

Perform functions related to date and time like setdate() and getdate(). To modify the system date and get the CPU time respectively.

8. #include<assert.h> (Assertion header)

It is used in program assertion functions like assert(). To get an integer data type in C/C++ as a parameter which prints stderr only if the parameter passed is 0.

9. #include<locale.h> (Localization header)

Perform localization functions like setlocale() and localeconv(). To set locale and get locale conventions respectively.

10. #include<signal.h> (Signal header)

Perform signal handling functions like signal() and raise(). To install signal handler and to raise the signal in the program respectively

11. #include<setjmp.h> (Jump header)

Perform jump functions.

12. #include<stdarg.h> (Standard argument header)

Perform standard argument functions like va_start and va_arg(). To indicate start of the variable-length argument list and to fetch the arguments from the variable-length argument list in the program respectively.

13. #include<errno.h> (Error handling header)

Used to perform error handling operations like errno(). To indicate errors in the program by initially assigning the value of this function to 0 and then later changing it to indicate errors.

Following are some C++ header files which are not supported in C-

#inlcude<iostream> (Input Output Stream) – Used as a stream of Input and Output.

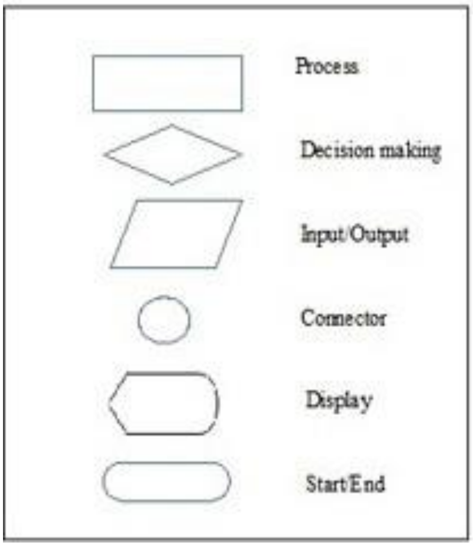
#include<iomanip.h> (Input-Output Manipulation) – Used to access set() and setprecision().

#include<fstream.h> (File stream) – Used to control the data to read from a file as an input and data to write into the file as an output.

7.Conditional Statement:

Conditional statements help you to make a decision based on certain conditions. These conditions are specified by a set of conditional statements having boolean expressions which are evaluated to a boolean value true or false

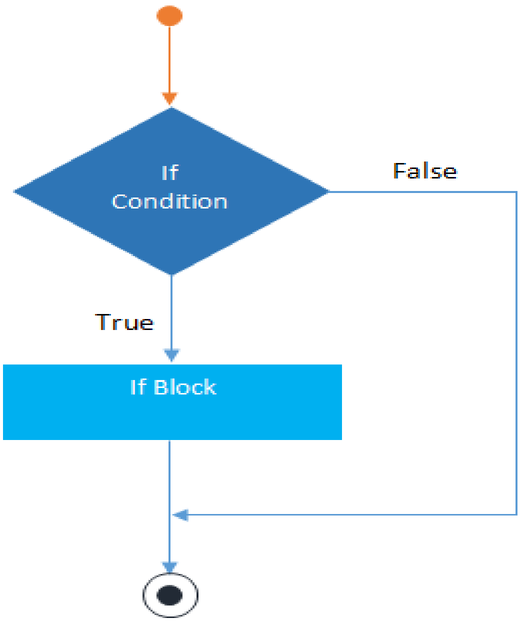
Flow chart shapes:



1) **If Statement**
   The single if statement in C language is used to execute the code if a condition is true. It is also called one-way selection statement.

   Syntax

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



If statement

```
// Program to print positive number entered by the user
// If user enters negative number, it is skipped
#include <iostream>
using namespace std;
int main()
{
   int number;
   cout << "Enter an integer: ";
   cin >> number;
   // checks if the number is positive
   if ( number > 0)
   {
      cout << "You entered a positive integer: " << number << endl;
   }

   cout << "This statement is always executed.";
```

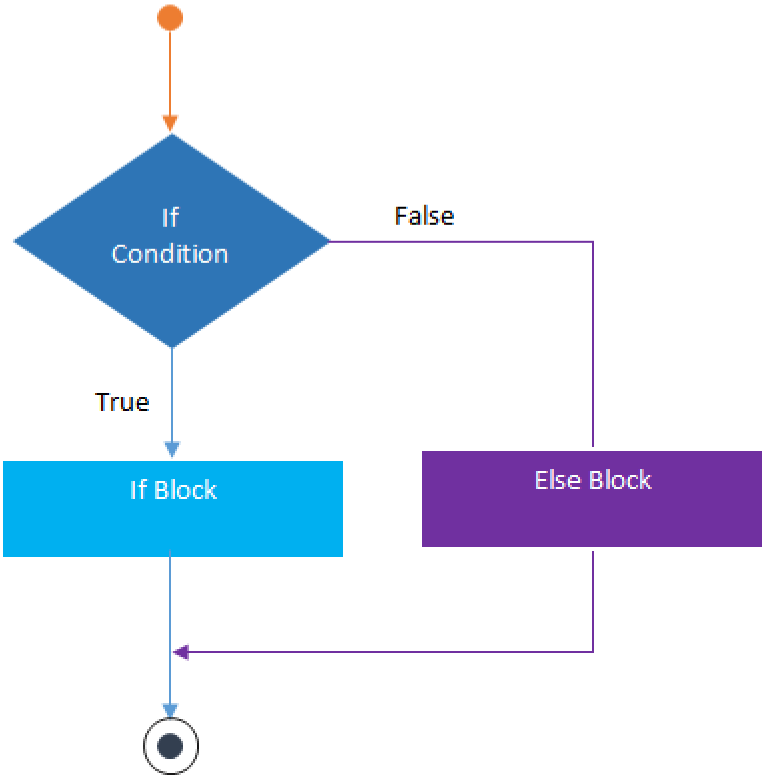
```
    return 0;

}
```

1) If else statement

The if-else statement in C language is used to execute the code if condition is true or false. It is also called two-way selection statement.

Syntax

```
if(expression)
{
 //Statements
}
else
{
 //Statements
}
```



If-Else statement

```
// Program to check whether an integer is positive or negative

// This program considers 0 as positive number
```

```
#include <iostream>

using namespace std;

int main()

{

   int number;

   cout << "Enter an integer: ";

   cin >> number;

   if ( number >= 0)

   {

      cout << "You entered a positive integer: " << number << endl;

   }

   else

   {

      cout << "You entered a negative integer: " << number << endl;

   }

   cout << "This line is always printed.";

   return 0;

}
```

1) **Nested if else**

   The nested if...else statement is used when a program requires more than one test expression. It is also called a multi-way selection statement. When a series of the decision are involved in a statement, we use if else statement in nested form.

   Syntax

```
if( expression )
{
 if( expression1 )
 {
 statement-block1;
 }
 else
 {
 statement-block 2;
 }
```

```
            }
            else
            {
             statement-block 3;
            }
```

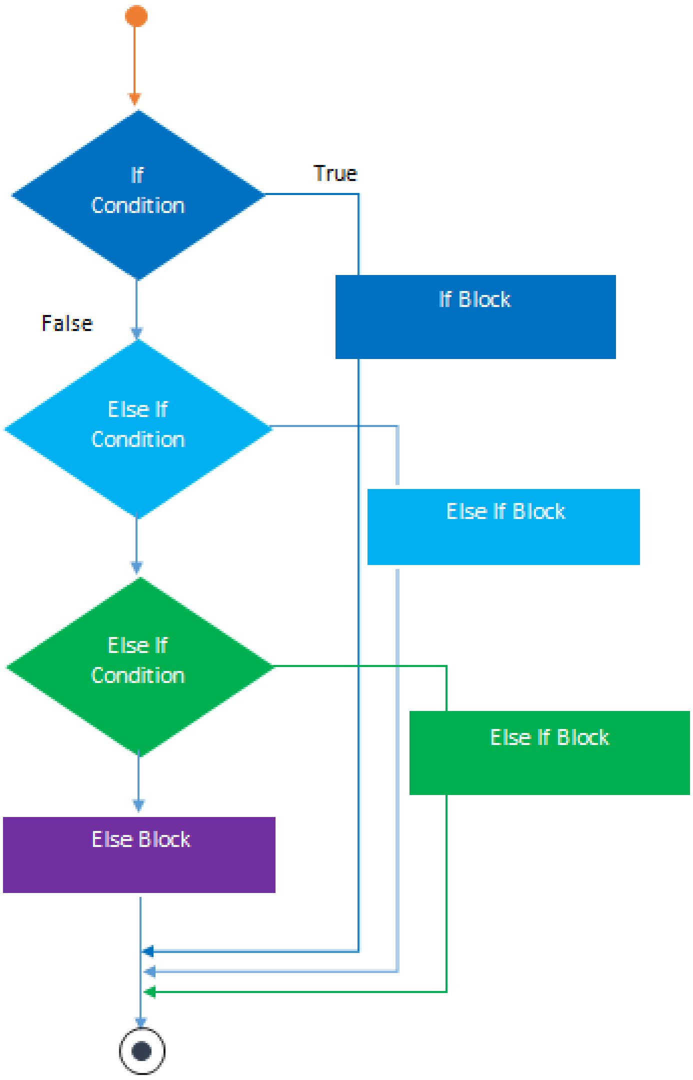
```
#include <iostream>
using namespace std;
int main()
{
    int number;
    cout << "Enter an integer: ";
    cin >> number;
    if ( number > 0)
    {
        cout << "You entered a positive integer: " << number << endl;
    }
    else if (number < 0)
    {
        cout<<"You entered a negative integer: " << number << endl;
    }
    else
    {
        cout << "You entered 0." << endl;
    }

    cout << "This line is always printed.";
    return 0;
}
```

1) If else ladder-

The if-else-if statement is used to execute one code from multiple conditions. It is also called multipath decision statement. It is a chain of if..else statements in which each if statement is associated with else if statement and last would be an else statement.

Syntax

```
if(condition1)
{
 //statements
}
else if(condition2)
{
 //statements
}
else if(condition3)
{
 //statements
}
else
{
 //statements
}
```



If-Else-If Ladder

```
#include <iostream>
using namespace std;
int main( )
{
 int a;
cout<<"enter a number";
   cin >>a;
 if( a%5==0 && a%8==0)
 {
```

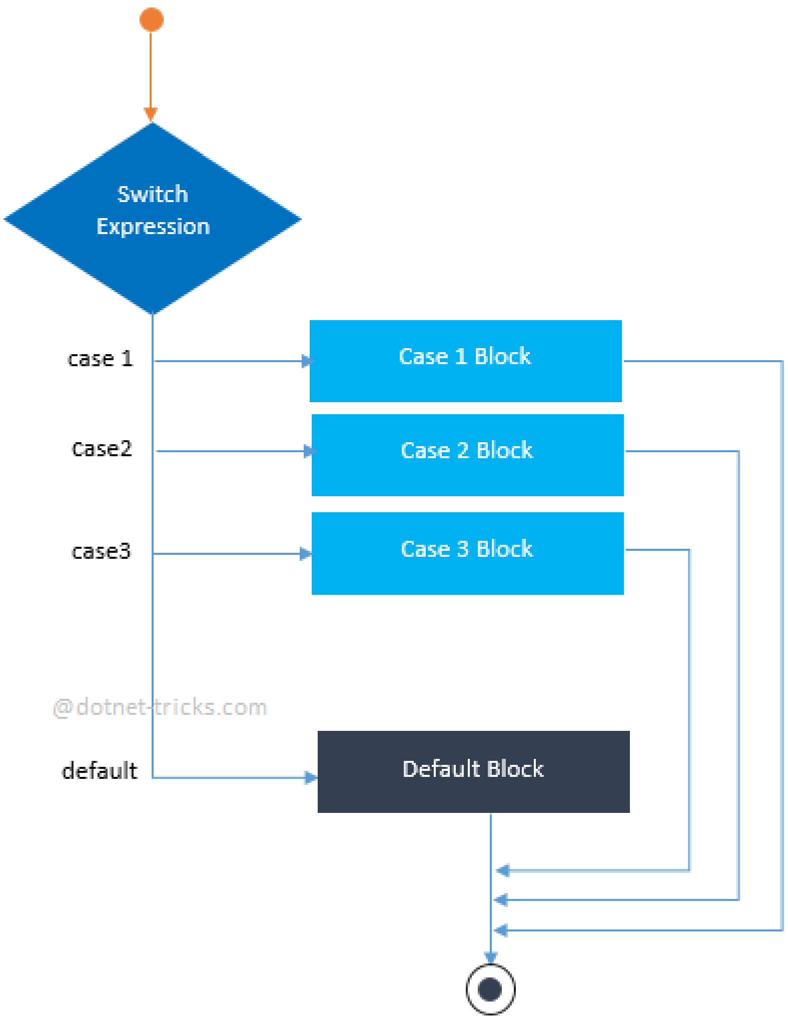
```
 cout<<"divisible by both 5 and 8";

 }

 else if( a%8==0 )

 {

 cout<<"divisible by 8";

 }

 else if(a%5==0)

 {

cout<<"divisible by 5";

 }

 else

 {

 cout<<"divisible by none";

 }

 return 0;

}
```

5)Switch statement-

switch statement acts as a substitute for a long if-else-if ladder that is used to test a list of cases. A switch statement contains one or more case labels which are tested against the switch expression. When the expression match to a case then the associated statements with that case would be executed.

Syntax

```
Switch (expression)
{
 case value1:
 //Statements
 break;
 case value 2:
 //Statements
 break;
 case value 3:
 //Statements
 case value n:
 //Statements
 break;
 Default:
 //Statements
}
```



Switch Statement

```
#include <iostream>

using namespace std;

int main() {

  int day = 4;

  switch (day) {

  case 1:

    cout << "Monday";
    break;
  case 2:
    cout << "Tuesday";
    break;
  case 3:
    cout << "Wednesday";
    break;
  case 4:
    cout << "Thursday";
    break;
  case 5:
    cout << "Friday";
    break;
  case 6:
    cout << "Saturday";
   break;
  case 7:
    cout << "Sunday";
    break;
  }
  return 0;
}
```

8.Loops:

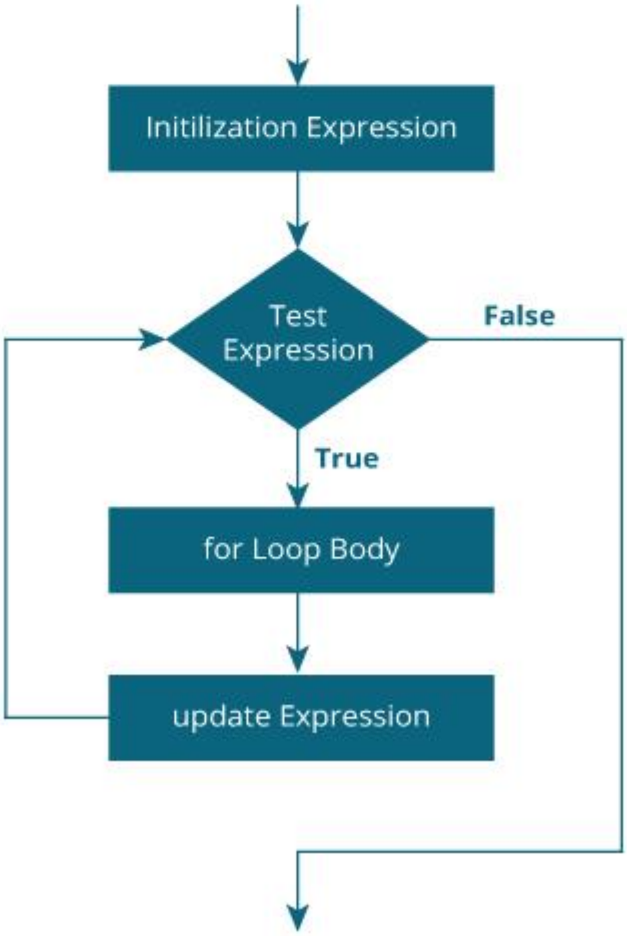
1) For loop-

The syntax of the for loop is:

```
for (initializationStatement; testExpression; updateStatement)
{
   // statements inside the body of loop
}
```

How for loop works?

1) The initialization statement is executed only once.
2) Then, the test expression is evaluated. If the test expression is evaluated to false, the for loop is terminated.
3) However, if the test expression is evaluated to true, statements inside the body of for loop are executed, and the update expression is updated.
4) Again the test expression is evaluated.
5) This process goes on until the test expression is false. When the test expression is false, the loop terminates.



```
#include <iostream>

using namespace std;


int main()

{

    int i, n, factorial = 1;

    cout << "Enter a positive integer: ";

    cin >> n;

    for (i = 1; i <= n; ++i) {

        factorial *= i;   // factorial = factorial * i;

    }

    cout<< "Factorial of "<<n<<" = "<<factorial;

    return 0;

}
```

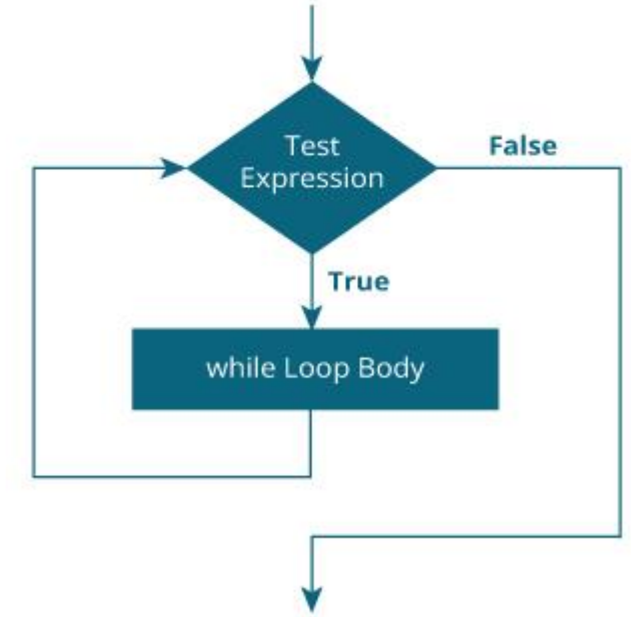
2) While loop

The syntax of the while loop is:

```
while (testExpression)

{

    // statements inside the body of the loop

}
```

How while loop works?

1) The while loop evaluates the test expression inside the parenthesis ().
2) If the test expression is true, statements inside the body of while loop are executed. Then, the test expression is evaluated again.
3) The process goes on until the test expression is evaluated to false.
4) If the test expression is false, the loop terminates (ends).



```
#include <iostream>

using namespace std;


int main()

{

   int number, i = 1, factorial = 1;

   cout << "Enter a positive integer: ";

   cin >> number;


   while ( i <= number) {

      factorial *= i;      //factorial = factorial * i;

      ++i;

   }

   cout<<"Factorial of "<< number <<" = "<< factorial;

   return 0;

}
```

3) Do While loop

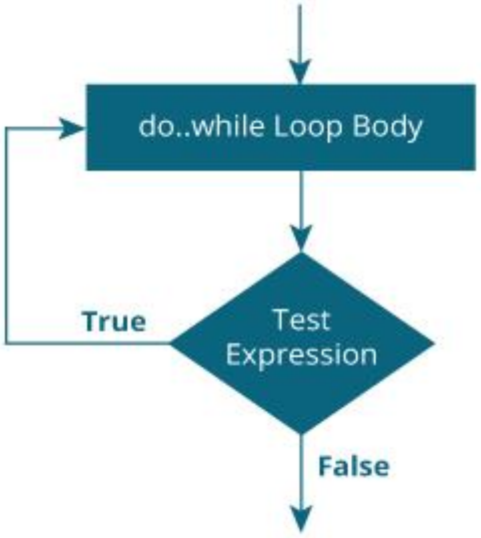
The do..while loop is similar to the while loop with one important difference. The body of do...while loop is executed at least once. Only then, the test expression is evaluated.

The syntax of the do...while loop is:

```
do

{

  // statements inside the body of the loop

}

while (testExpression);
```

How do...while loop works?

1) The body of do...while loop is executed once. Only then, the test expression is evaluated.
2) If the test expression is true, the body of the loop is executed again and the test expression is evaluated.
3) This process goes on until the test expression becomes false.
4) If the test expression is false, the loop ends.

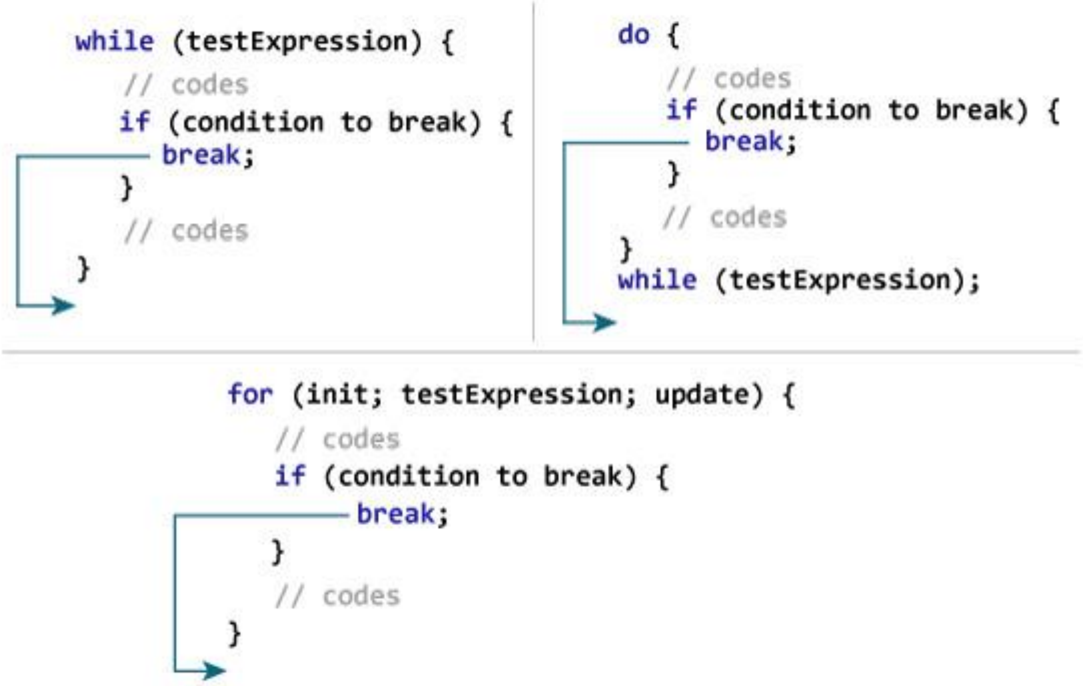


```
#include <iostream>

using namespace std;

int main()

{

   float number, sum = 0.0;
```

```
    do {
        cout<<"Enter a number: ";
        cin>>number;
        sum += number;
    }
    while(number != 0.0);
    cout<<"Total sum = "<<sum;
    return 0;
}
```

1) **Break statement**
   The break statement ends the loop immediately when it is encountered. Its syntax is:
   break;

   The break statement is almost always used with if...else statement inside the loop.



```
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
}
```

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

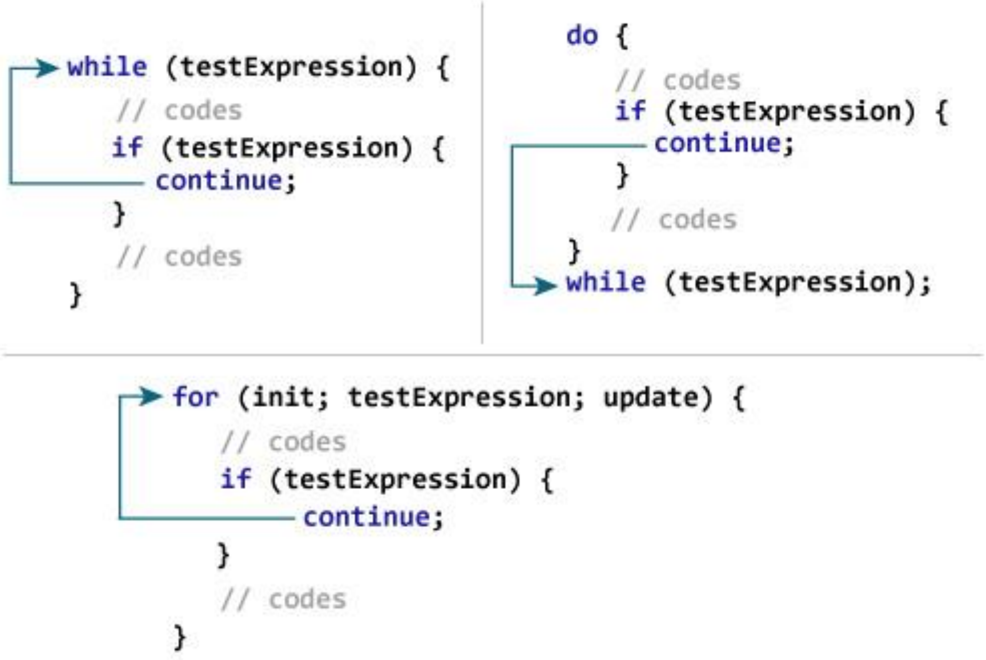
```
int main() {
    float number, sum = 0.0;
    // test expression is always true
    while (true)
    {
        cout << "Enter a number: ";
        cin >> number;
        if (number != 0.0)
        {
            sum += number;
        }
        else
        {
            // terminates the loop if number equals 0.0
            break;
        }
    }
    cout << "Sum = " << sum;

    return 0;
}
```

1) Continue statement

The continue statement skips the current iteration of the loop and continues with the next iteration. Its syntax is:

continue;

The continue statement is almost always used with the if...else statement.



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

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

```
for (init; testExpression; update) {
    // codes
    if (testExpression) {
        continue;
    }
    // codes
}
```

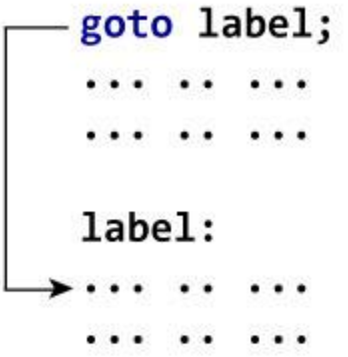
```
#include <iostream>

using namespace std;

int main()

{

    for (int i = 1; i <= 10; ++i)

    {

        if ( i == 6 || i == 9)

        {

            continue;

        }

        cout << i << "\t";

    }

    return 0;

}
```

1) Goto statement

The goto statement allows us to transfer control of the program to the specified label.

Syntax of goto Statement

```
goto label;

... .. ...

... .. ...

label:

statement;
```



```
goto label;
... .. ...
... .. ...

label:
... .. ...
... .. ...
```

```
# include <iostream>

using namespace std;


int main()

{

    float num, average, sum = 0.0;

    int i, n;


    cout << "Maximum number of inputs: ";

    cin >> n;


    for(i = 1; i <= n; ++i)

    {

        cout << "Enter n" << i << ": ";
        cin >> num;

        if(num < 0.0)
        {
           // Control of the program move to jump:
            goto jump;
        }
        sum += num;
    }

jump:
    average = sum / (i - 1);
    cout << "\nAverage = " << average;
    return 0;
}
```

Reasons to avoid goto

The use of goto statement may lead to code that is buggy and hard to follow. For example,

```
one:
for (i = 0; i < number; ++i)
{
    test += i;
    goto two;
}
two:
if (test > 5) {
  goto three;
}
```

... .

Also, the goto statement allows you to do bad stuff such as jump out of the scope.

That being said, goto can be useful sometimes. For example: to break from nested loops

4) Difference

1) Break and continue

Break	**continue**
A break can appear in both switch and loop (for, while, do) statements.	A continue can appear only in loop (for, while, do) statements.
A break causes the switch or loop statements to terminate the moment it is executed. Loop or switch ends abruptly when break is encountered.	A continue doesn't terminate the loop, it causes the loop to go to the next iteration. All iterations of the loop are executed even if continue is encountered. The continue statement is used to skip statements in the loop that appear after the continue.
The break statement can be used in both switch and loop statements.	The continue statement can appear only in loops. You will get an error if this appears in switch statement.
When a break statement is encountered, it terminates the block and gets the control out of the switch or loop.	When a continue statement is encountered, it gets the control to the next iteration of the loop.
A break causes the innermost enclosing loop or switch to be exited immediately.	A continue inside a loop nested within a switch causes the next loop iteration.

2) While and do while

WHILE	DO-WHILE
Condition is checked first then statement(s) is executed.	Statement(s) is executed atleast once, thereafter condition is checked.
It might occur statement(s) is executed zero times, If condition is false.	At least once the statement(s) is executed.
No semicolon at the end of while. while(condition)	Semicolon at the end of while. while(condition);
If there is a single statement, brackets are not required.	Brackets are always required.
Variable in condition is initialized before the execution of loop.	variable may be initialized before or within the loop.
while loop is entry controlled loop.	do-while loop is exit controlled loop.

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

3) For loop,while loop.do while loop

Sr. No	**For loop**	**While loop**	**Do while loop**
1.	Syntax: For(initialization; condition;updating), { . Statements; }	Syntax: While(condition), { . Statements; . }	Syntax: Do { . Statements; } While(condition);
2.	It is known as entry controlled loop	It is known as entry controlled loop.	It is known as exit controlled loop.
3.	If the condition is not true first time than control will never enter in a loop	If the condition is not true first time than control will never enter in a loop.	Even if the condition is not true for the first time the control will enter in a loop.
4.	There is no semicolon; after the condition in the syntax of the for loop.	There is no semicolon; after the condition in the syntax of the while loop.	There is semicolon; after the condition in the syntax of the do while loop.
5.	Initialization and updating is the part of the syntax.	Initialization and updating is not the part of the syntax.	Initialization and updating is not the part of the syntax

4) If else and switch

BASIS FOR COMPARISON	IF-ELSE	SWITCH
Basic	Which statement will be executed depend upon the output of the expression inside if statement.	Which statement will be executed is decided by user.
Expression	if-else statement uses multiple statement for multiple choices.	switch statement uses single expression for multiple choices.
Testing	if-else statement test for equality as well as for logical expression.	switch statement test only for equality.
Evaluation	if statement evaluates integer, character, pointer or floating-point type or boolean type.	switch statement evaluates only character or integer value.
Sequence of	Either if statement will be executed or else statement	switch statement execute one case after another till a break

BASIS FOR COMPARISON	IF-ELSE	SWITCH
Execution	is executed.	statement is appeared or the end of switch statement is reached.
Default Execution	If the condition inside if statements is false, then by default the else statement is executed if created.	If the condition inside switch statements does not match with any of cases, for that instance the default statements is executed if created.
Editing	It is difficult to edit the if-else statement, if the nested if-else statement is used.	It is easy to edit switch cases as, they are recognized easily.

9.Functions

In programming, function refers to a segment that groups code to perform a specific task.

Depending on whether a function is predefined or created by programmer; there are two types of function:

1) Library Function

2) User-defined Function

1) Library Function

Library functions are the built-in function in C++ programming.

Programmer can use library function by invoking function directly; they don't need to write it themselves.

Example 1: Library Function

```
#include <iostream>

#include <cmath>


using namespace std;

int main()

{

    double number, squareRoot;

    cout << "Enter a number: ";

    cin >> number;

    // sqrt() is a library function to calculate square root

    squareRoot = sqrt(number);

    cout << "Square root of " << number << " = " << squareRoot;

    return 0;

}
```

1) **User defined functions:**

You can also create functions as per your need. Such functions created by the user are known as user-defined functions.

How user-defined function works?

```
#include <stdio.h>
void functionName()
{
    ... .. ...
    ... .. ...
}
```

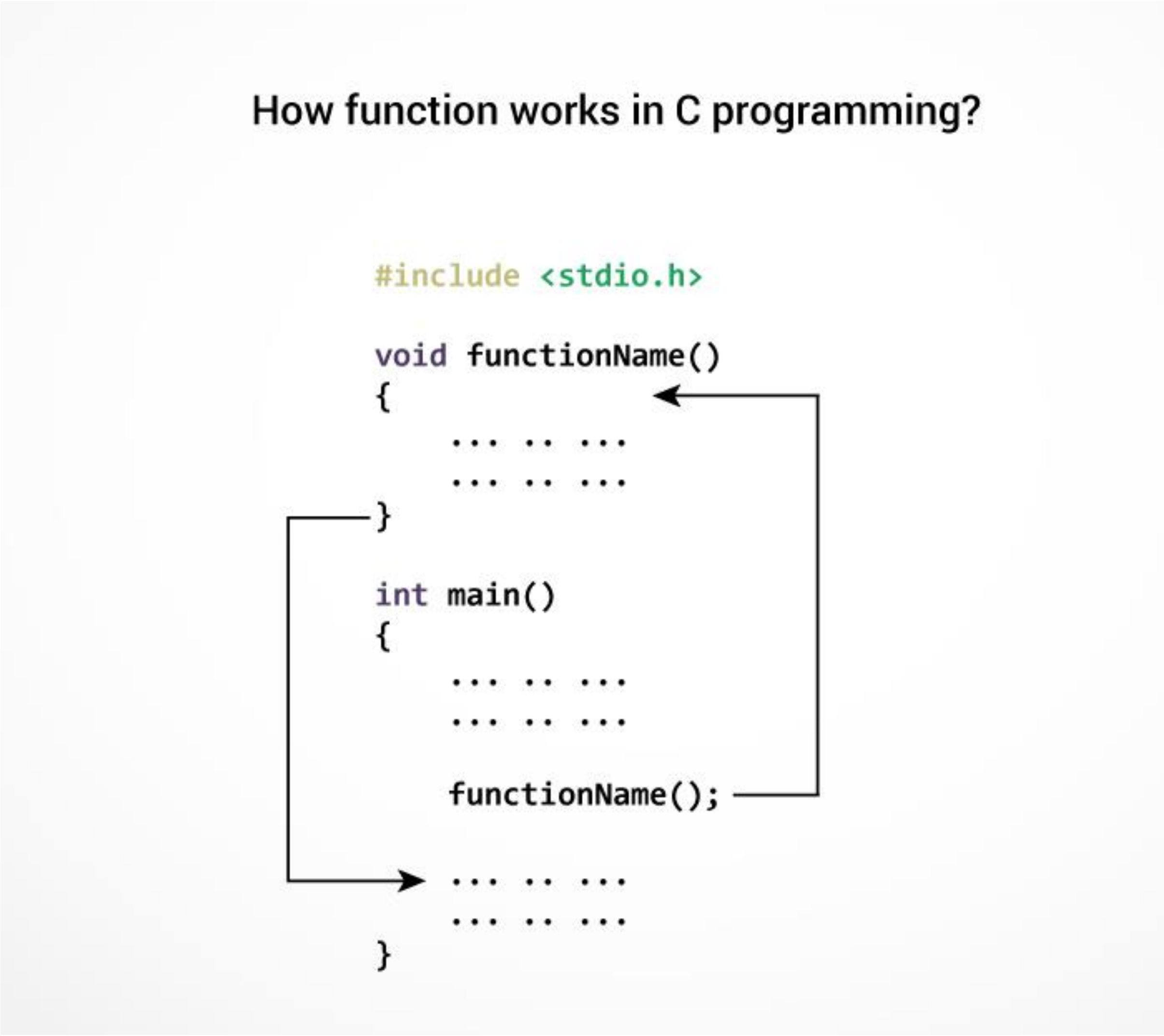
```
int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

The execution of a C program begins from the main() function.

When the compiler encounters functionName();, control of the program jumps to

void functionName()

And, the compiler starts executing the codes inside functionName().
The control of the program jumps back to the main() function once code inside the function definition is executed.



How function works in C programming?

```
#include <stdio.h>

void functionName()
{
    ... .. ...
    ... .. ...
}

int main()
{
    ... .. ...
    ... .. ...

    functionName();

    ... .. ...
    ... .. ...
}
```

```
#include <iostream>

using namespace std;

// Function prototype (declaration)

int add(int, int);

int main()

{

    int num1, num2, sum;

    cout<<"Enters two numbers to add: ";

    cin >> num1 >> num2;


    // Function call

    sum = add(num1, num2);

    cout << "Sum = " << sum;

    return 0;

}

// Function definition

int add(int a, int b)

{

    int add;

    add = a + b;


    // Return statement

    return add;

}
```

Function prototype

A function prototype is simply the declaration of a function that specifies function's name, parameters and return type. It doesn't contain function body.

A function prototype gives information to the compiler that the function may later be used in the program.

Syntax of function prototype

```
returnType functionName(type1 argument1, type2 argument2, ...);
```

In the above example, int addNumbers(int a, int b); is the function prototype which provides the following information to the compiler:

1) name of the function is addNumbers()
2) return type of the function is int
3) two arguments of type int are passed to the function

The function prototype is not needed if the user-defined function is defined before the main() function.

Calling a function

Control of the program is transferred to the user-defined function by calling it.

Syntax of function call

```
functionName(argument1, argument2, ...);
```

In the above example, the function call is made using addNumbers(n1, n2); statement inside the main() function.

Function definition

Function definition contains the block of code to perform a specific task. In our example, adding two numbers and returning it.

Syntax of function definition

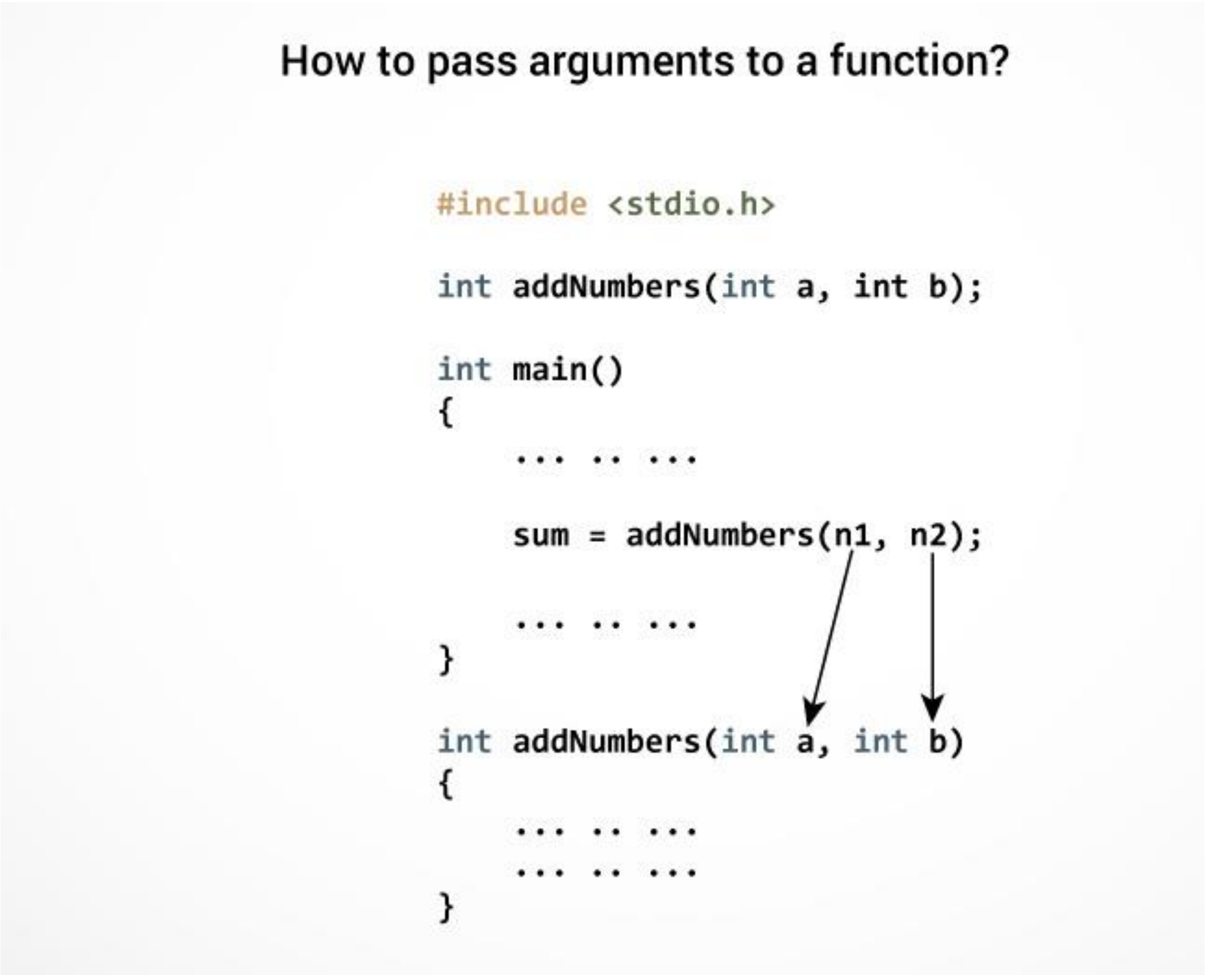
```
returnType functionName(type1 argument1, type2 argument2, ...)

{

   //body of the function

}
```

When a function is called, the control of the program is transferred to the function definition. And, the compiler starts executing the codes inside the body of a function.

Passing arguments to a function

In programming, argument refers to the variable passed to the function. In the above example, two variables n1 and n2 are passed during the function call.

The parameters a and b accepts the passed arguments in the function definition. These arguments are called formal parameters of the function.



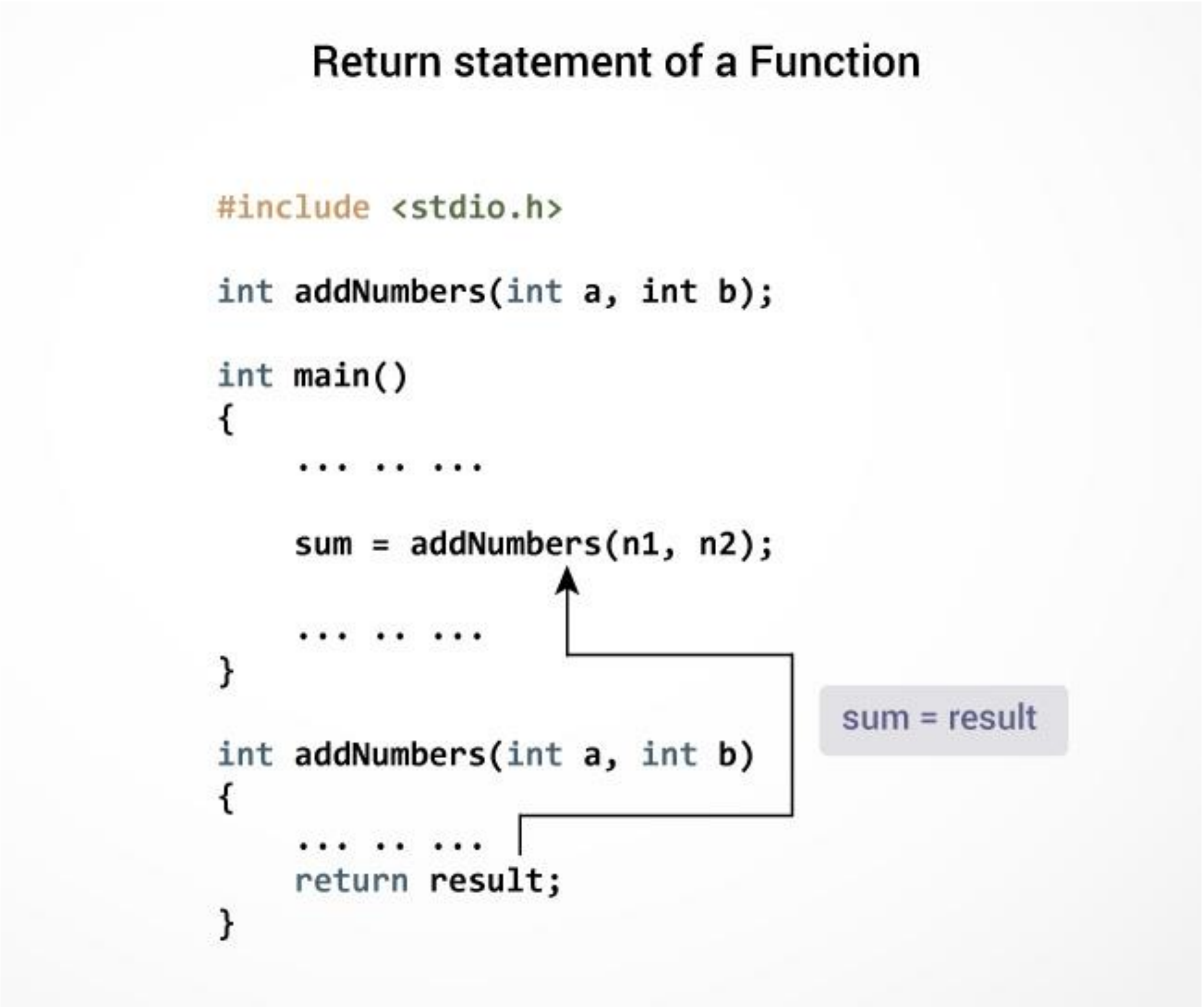
How to pass arguments to a function?

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    ... .. ...
}
```

Return Statement

The return statement terminates the execution of a function and returns a value to the calling function. The program control is transferred to the calling function after the return statement.

In the above example, the value of the result variable is returned to the main function. The sum variable in the main() function is assigned this value.



Return statement of a Function

```
#include <stdio.h>

int addNumbers(int a, int b);

int main()
{
    ... .. ...

    sum = addNumbers(n1, n2);

    ... .. ...
}

int addNumbers(int a, int b)
{
    ... .. ...
    return result;
}
```

Syntax of return statement

return (expression);

For example,

return a;

return (a+b);

Example:

```
#include <iostream>

using namespace std;

int prime(int n);


int main()

{
    int num, flag = 0;
    cout << "Enter positive integer to check: ";
    cin >> num;

    // Argument num is passed to check() function
    flag = prime(num);

    if(flag == 1)
        cout << num << " is not a prime number.";
    else
        cout<< num << " is a prime number.";
    return 0;
}

/* This function returns integer value.  */
int prime(int n)
{
    int i;
    for(i = 2; i <= n/2; ++i)
    {
        if(n % i == 0)
            return 1;
    }
    return 0;
}
```

Function Overloading

Two or more functions having same name but different argument(s) are known as overloaded functions. In this article, you will learn about function overloading with examples.

Function refers to a segment that groups code to perform a specific task.

In C++ programming, two functions can have same name if number and/or type of arguments passed are different.

These functions having different number or type (or both) of parameters are known as overloaded functions. For example:

```
int test() { }

int test(int a) { }

float test(double a) { }

int test(int a, double b) { }
```

Here, all 4 functions are overloaded functions because argument(s) passed to these functions are different.

Notice that, the return type of all these 4 functions are not same. Overloaded functions may or may not have different return type but it should have different argument(s).

```
// Error code

int test(int a) { }

double test(int b){ }
```

The number and type of arguments passed to these two functions are same even though the return type is different. Hence, the compiler will throw error.

```
#include <iostream>

using namespace std;

void display(int);

void display(float);

void display(int, float);

int main() {

    int a = 5;

    float b = 5.5;


    display(a);

    display(b);

    display(a, b);

    return 0;
}

void display(int var) {
    cout << "Integer number: " << var << endl;
}

void display(float var) {
    cout << "Float number: " << var << endl;
}

void display(int var1, float var2) {
    cout << "Integer number: " << var1;
    cout << " and float number:" << var2;
}
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and float number: 5.5
```

Example 2

```
#include <iostream>
using namespace std;
int absolute(int);
float absolute(float);
int main() {
    int a = -5;
    float b = 5.5;
    cout << "Absolute value of " << a << " = " << absolute(a) << endl;
    cout << "Absolute value of " << b << " = " << absolute(b);

    return 0;

}

int absolute(int var) {

     if (var < 0)

         var = -var;

    return var;

}

float absolute(float var){

    if (var < 0.0)

        var = -var;

    return var;

}
```

Output

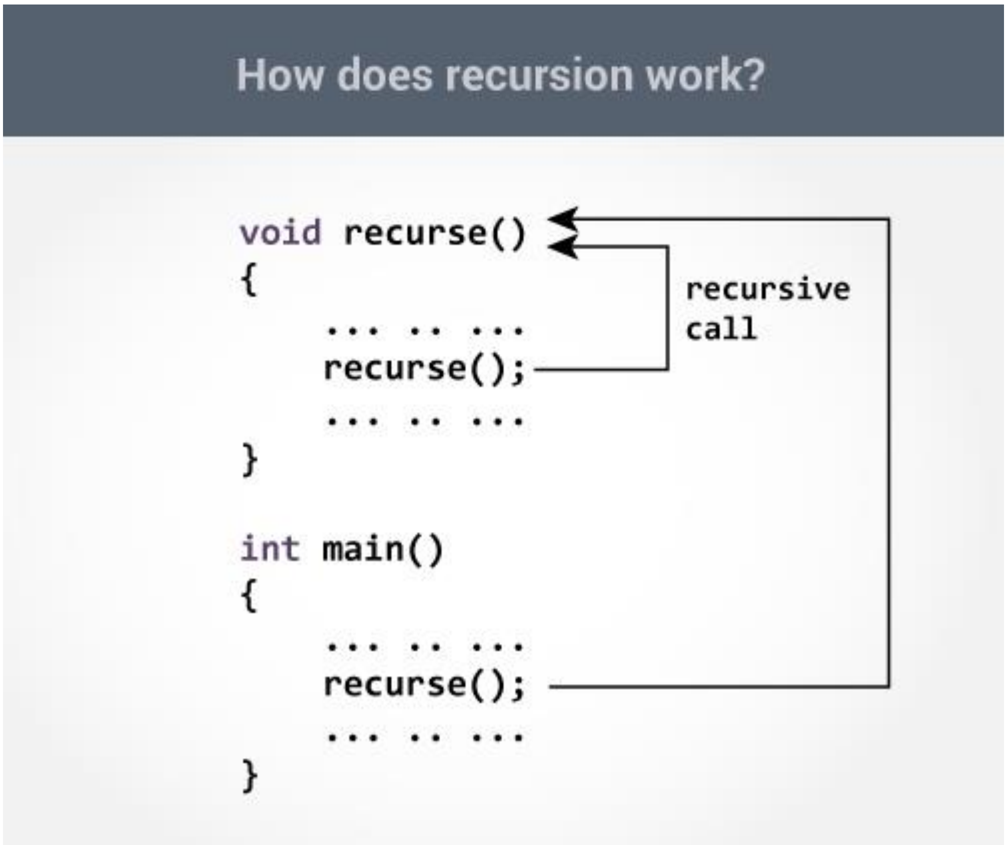
```
Absolute value of -5 = 5

Absolute value of 5.5 = 5.5
```

10.Recursion

A function that calls itself is known as a recursive function. And, this technique is known as recursion.

How recursion works?

```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}
int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

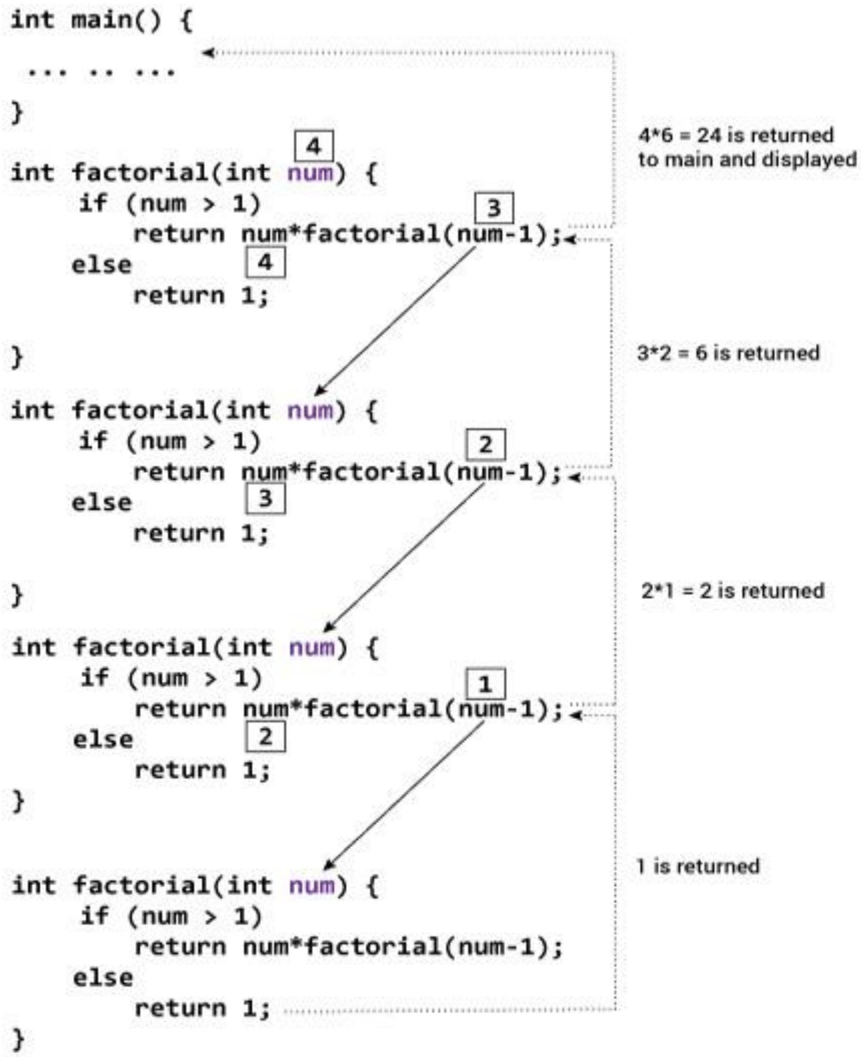


```
void recurse()
{
    ... .. ...
    recurse();
    ... .. ...
}

int main()
{
    ... .. ...
    recurse();
    ... .. ...
}
```

```
#include <iostream>

using namespace std;

int factorial(int);

int main()

{

    int n;

    cout<<"Enter a number to find factorial: ";

    cin >> n;

    cout << "Factorial of " << n <<" = " << factorial(n);

    return 0;

}


int factorial(int n)

{
    if (n > 1)
    {
        return n*factorial(n-1);
    }
    else
    {
        return 1;
    }
}
```

Output

```
Enter a number to find factorial: 4
Factorial of 4 = 24
```



```
int main() {
 ... .. ...
}

int factorial(int num) {
    if (num > 1)
        return num*factorial(num-1);
    else
        return 1;

}

int factorial(int num) {
    if (num > 1)
        return num*factorial(num-1);
    else
        return 1;

}

int factorial(int num) {
    if (num > 1)
        return num*factorial(num-1);
    else
        return 1;
}

int factorial(int num) {
    if (num > 1)
        return num*factorial(num-1);
    else
        return 1;
}
```

Advantages

1) Reduce unnecessary calling of function.
2) Through Recursion one can Solve problems in easy way while its iterative solution is very big and complex.

Disdvantages

1) Recursive solution is always logical and it is very difficult to trace.(debug and understand).
2) In recursive we must have an if statement somewhere to force the function to return without the recursive call being executed, otherwise the function will never return.
3) Recursion takes a lot of stack space, usually not considerable when the program is small and running on a PC.
4) Recursion uses more processor time.

11.Storage classes in C

A storage class represents the visibility and a location of a variable. It tells from what part of code we can access a variable. A storage class is used to describe the following things:

1) The variable scope.
2) The location where the variable will be stored.
3) The initialized value of a variable.
4) A lifetime of a variable.
5) Who can access a variable?

Storage classes

Auto-It is a default storage class.

Extern-It is a global variable.

Static-It is a local variable which is capable of returning a value even when control is transferred to the function call.

Register-It is a variable which is stored inside a Register.

Auto storage class

The variables defined using auto storage class are called as local variables. Auto stands for automatic storage class. A variable is in auto storage class by default if it is not explicitly specified.

The scope of an auto variable is limited with the particular block only. Once the control goes out of the block, the access is destroyed. This means only the block in which the auto variable is declared can access it.

A keyword auto is used to define an auto storage class. By default, an auto variable contains a garbage value.

Example, auto int age;

The program below defines a function with has two local variables

```
int add(void) {

  int a=13;

  auto int b=48;

return a+b;}
```

We take another program which shows the scope level "visibility level" for auto variables in each block code which are independently to each other:

```
#include <iostream>
using namespace std;
int main( )
{
   auto  j = 1;
  {
   auto   j= 2;
    {
      auto  j = 3;
cout<<j;
    }
    cout<<j;
  }
cout<<j;
}OUTPUT:
```

```
 3 2 1
```

Extern storage class

Extern stands for external storage class. Extern storage class is used when we have global functions or variables which are shared between two or more files.

Keyword extern is used to declaring a global variable or function in another file to provide the reference of variable or function which have been already defined in the original file.

The variables defined using an extern keyword are called as global variables. These variables are accessible throughout the program. Notice that the extern variable cannot be initialized it has already been defined in the original file

Example, extern void display();

First File: main.cpp

```
#include <iostram>
extern i;
main() {
```

```
cout<<i;
   return 0;}
```

Second File: original.cpp

```
#include <iostream>
i=48;
```

Result:

```
 value of the external integer is = 48
```

Example 2

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

Second File: support.cpp

```
#include <iostream>
extern int cnt;

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

Static storage class

The static variables are used within function/ file as local static variables. They can also be used as a global variable

Static local variable is a local variable that retains and stores its value between function calls or block and remains visible only to the function or block in which it is defined.

Static global variables are global variables visible only to the file in which it is declared.

Example: static int count = 10;

Keep in mind that static variable has a default initial value zero and is initialized only once in its lifetime.

Example

```
#include <iostream>

void function(void);

static int c = 5; // Global static variable

main() {

   while(c--) {

      function();

   }

   return 0;

}

void function( void ) {

   static int cnt = 2;

   cnt++;

   std::cout << "cnt is " << cnt ;

   std::cout << " and c is " << c << std::endl;

}
```

Register storage class

You can use the register storage class when you want to store local variables within functions or blocks in CPU registers instead of RAM to have quick access to these variables. For example, "counters" are a good candidate to be stored in the register.

Example: register int age;

The keyword register is used to declare a register storage class. The variables declared using register storage class has lifespan throughout the program.

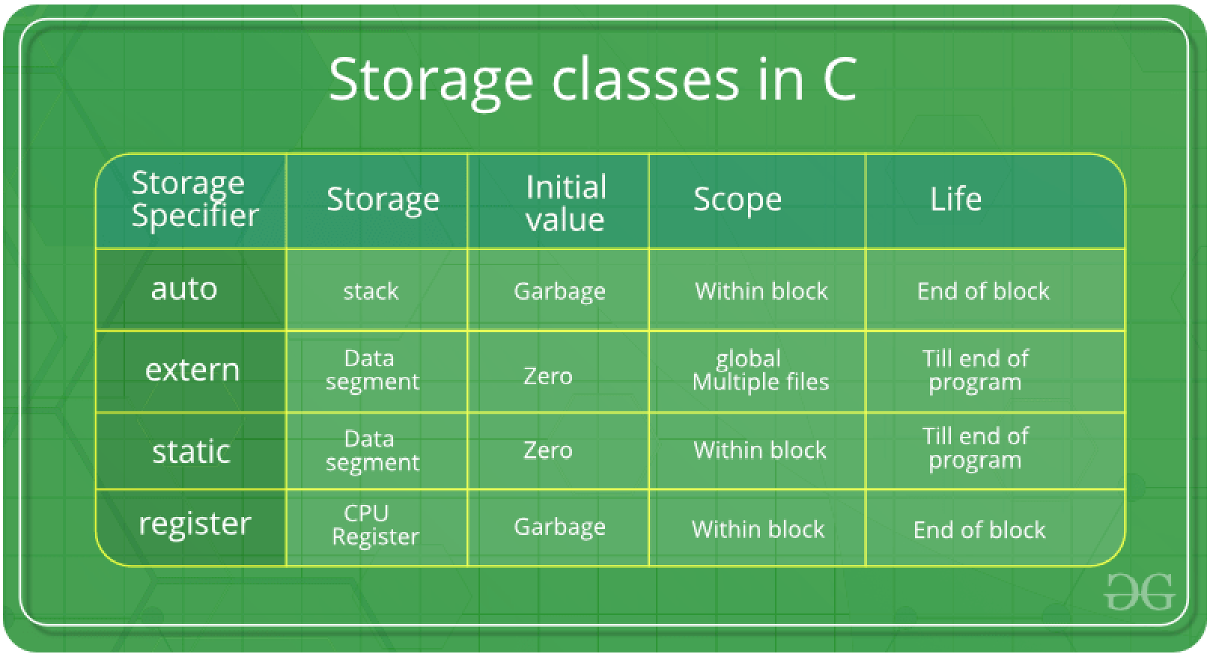
It is similar to the auto storage class. The variable is limited to the particular block. The only difference is that the variables declared using register storage class are stored inside CPU registers instead of a memory. Register has faster access than that of the main memory.

The variables declared using register storage class has no default value. These variables are often declared at the beginning of a program.

```
#include <iostream> /* function declaration */

main() {

{

register int  weight;

int *ptr=&weight ;/*it produces an error when the compilation occurs ,we cannot get a memory location when dealing with CPU register*/}

}
```

OUTPUT:

```
error: address of register variable 'weight' requested
```



Storage classes in C

Storage Specifier	Storage	Initial value	Scope	Life
auto	stack	Garbage	Within block	End of block
extern	Data segment	Zero	global Multiple files	Till end of program
static	Data segment	Zero	Within block	Till end of program
register	CPU Register	Garbage	Within block	End of block

12 Arrays

An array is a variable that can store multiple values of same data type. For example, if you want to store 100 integers, you can create an array for it.

int data[100];

How to declare an array?

dataType arrayName[arraySize];

For example,

float mark[5];

Here, we declared an array, mark, of floating-point type. And its size is 5. Meaning, it can hold 5 floating-point values.

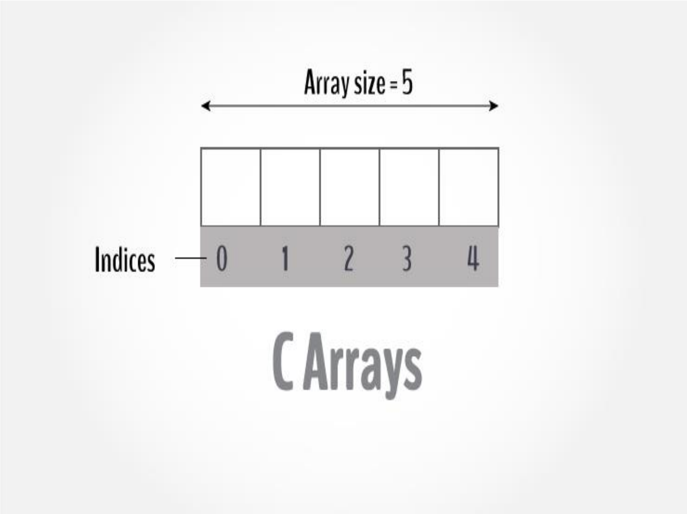
It's important to note that the size and type of an array cannot be changed once it is declared.

Types of array:-

1)Single dimensional array

2)Multidimentional array

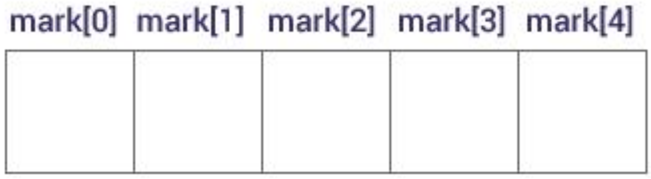
1)Single dimension array:-



Access Array Elements

You can access elements of an array by indices.

Suppose you declared an array mark as above. The first element is mark[0], the second element is mark[1] and so on.



mark[0]	mark[1]	mark[2]	mark[3]	mark[4]

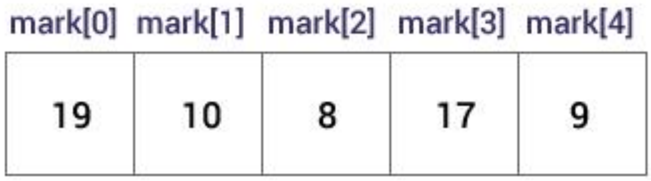
How to initialize an array?

It is possible to initialize an array during declaration. For example,

int mark[5] = {19, 10, 8, 17, 9};

You can also initialize an array like this.

int mark[] = {19, 10, 8, 17, 9};



mark[0]	mark[1]	mark[2]	mark[3]	mark[4]
19	10	8	17	9

How to insert and print array elements?

int mark[5] = {19, 10, 8, 17, 9}

// change 4th element to 9

```
mark[3] = 9;

// take input from the user and insert in third element

cin >> mark[2];

// take input from the user and insert in (i+1)th element

cin >> mark[i];

// print first element of the array

cout << mark[0];

// print ith element of the array

cout >> mark[i-1];
```

Example:

```
#include <iostream>

using namespace std;

int main()

{

    int numbers[5], sum = 0;

    cout << "Enter 5 numbers: ";

    //  Storing 5 number entered by user in an array

    //  Finding the sum of numbers entered

    for (int i = 0; i < 5; ++i)

    {

        cin >> numbers[i];

        sum += numbers[i];

    }

    cout << "Sum = " << sum << endl;

    return 0;

}
```

Output

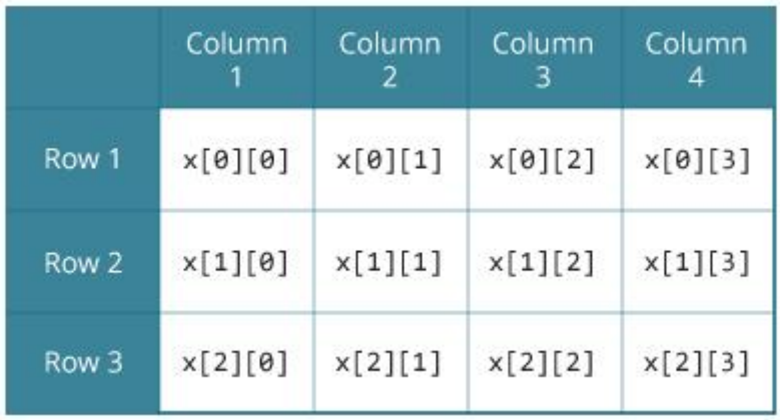
```
Enter 5 numbers: 3

4

5

4

2

Sum = 18
```

2)Multidimension array(2D,3D....)

In C programming, you can create an array of arrays. These arrays are known as multidimensional arrays. For example,

```
float x[3][4];
```

Here, x is a two-dimensional (2d) array. The array can hold 12 elements. You can think the array as a table with 3 rows and each row has 4 columns.



	Column 1	Column 2	Column 3	Column 4
Row 1	x[0][0]	x[0][1]	x[0][2]	x[0][3]
Row 2	x[1][0]	x[1][1]	x[1][2]	x[1][3]
Row 3	x[2][0]	x[2][1]	x[2][2]	x[2][3]

Initialization of a 2d array

```
// Different ways to initialize two-dimensional array

int c[2][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[][3] = {{1, 3, 0}, {-1, 5, 9}};

int c[2][3] = {1, 3, 0, -1, 5, 9};
```

Initialization of a 3d array

You can initialize a three-dimensional array in a similar way like a two-dimensional array. Here's an example,

```
int test[2][3][4] = {
    {{3, 4, 2, 3}, {0, -3, 9, 11}, {23, 12, 23, 2}},

    {{13, 4, 56, 3}, {5, 9, 3, 5}, {3, 1, 4, 9}}};
```

C++ Program to display all elements of an initialised two dimensional array.

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

int main()
{
    int test[3][2] ={
        {2, -5},
        {4, 0},
        {9, 1}
    };
    // Accessing two dimensional array using
    // nested for loops
    for(int i = 0; i < 3; ++i)
    {
        for(int j = 0; j < 2; ++j)
        {
            cout<< "test[" << i << "][" << j << "] = " << test[i][j] << endl;
        }
    }
    return 0;
}
```

Example 2D array

C++ Program to store temperature of two different cities for a week and display it.

```
#include <iostream>
using namespace std;
const int CITY = 2;
const int WEEK = 7;
int main()
{
    int temperature[CITY][WEEK];

    cout << "Enter all temperature for a week of first city and then second city. \n";

    // Inserting the values into the temperature array
    for (int i = 0; i < CITY; ++i)
    {
        for(int j = 0; j < WEEK; ++j)
        {
            cout << "City " << i + 1 << ", Day " << j + 1 << " : ";
            cin >> temperature[i][j];
        }
    }
    cout << "\n\nDisplaying Values:\n";
    // Accessing the values from the temperature array
    for (int i = 0; i < CITY; ++i)
    {
        for(int j = 0; j < WEEK; ++j)
        {
            cout << "City " << i + 1 << ", Day " << j + 1 << " = " << temperature[i][j] << endl;
        }
    }

    return 0;
}
```

Example 3D

C++ Program to Store value entered by user in three dimensional array and display it.

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

int main()
{
    // This array can store upto 12 elements (2x3x2)
    int test[2][3][2];

    cout << "Enter 12 values: \n";

    // Inserting the values into the test array
    // using 3 nested for loops.
    for(int i = 0; i < 2; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            for(int k = 0; k < 2; ++k )
            {
                cin >> test[i][j][k];
            }
        }
    }

    cout<<"\nDisplaying Value stored:"<<endl;

    // Displaying the values with proper index.
    for(int i = 0; i < 2; ++i)
    {
        for (int j = 0; j < 3; ++j)
        {
            for(int k = 0; k < 2; ++k)
            {
                cout << "test[" << i << "][" << j << "][" << k << "] = " << test[i][j][k] << endl;
            }
        }
    }

    return 0;
}
```

Passing array to function

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

void display(int marks[5]);
int main()
{
    int marks[5] = {88, 76, 90, 61, 69};
    display(marks);
    return 0;
}
void display(int m[5])
{
    cout << "Displaying marks: "<< endl;
    for (int i = 0; i < 5; ++i)
    {
        cout << "Student "<< i + 1 <<": "<< m[i] << endl;
    }
}
```

Passing two dimension array to function

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

void display(int n[3][2]);

int main()
{
    int num[3][2] = {
        {3, 4},
        {9, 5},
        {7, 1}

    };
    display(num);

    return 0;
}
```

```
void display(int n[3][2])
{

    cout << "Displaying Values: " << endl;
    for(int i = 0;  i < 3; ++i)
    {
        for(int j = 0; j < 2; ++j)
        {
            cout << n[i][j] << " ";
        }
    }
}
```

13 Pointers

Pointers are powerful features of C++ that differentiates it from other programming languages like Java and Python.

Pointers are used in C++ program to access the memory and manipulate the address.

Address in C++

To understand pointers, you should first know how data is stored on the computer.

Each variable you create in your program is assigned a location in the computer's memory. The value the variable stores is actually stored in the location assigned.

To know where the data is stored, C++ has an & operator. The & (reference) operator gives you the address occupied by a variable.

If var is a variable then, &var gives the address of that variable.

Example 1: Address in C++

```
#include <iostream>

using namespace std;

int main()

{

    int var1 = 3;

    int var2 = 24;

    int var3 = 17;

    cout << &var1 << endl;

    cout << &var2 << endl;

    cout << &var3 << endl;

}
```

Output

```
0x7fff5fbff8ac

0x7fff5fbff8a8

0x7fff5fbff8a4
```

Pointers Variables

C++ gives you the power to manipulate the data in the computer's memory directly. You can assign and de-assign any space in the memory as you wish. This is done using Pointer variables.

Pointers variables are variables that points to a specific address in the memory pointed by another variable.

How to declare a pointer?

```
int *p;

    OR,

int* p;
```

The statement above defines a pointer variable p. It holds the memory address

The asterisk is a dereference operator which means pointer to.

Here, pointer p is a pointer to int, i.e., it is pointing to an integer value in the memory address.

Reference operator (&) and Deference operator (*)

Reference operator (&) as discussed above gives the address of a variable.

To get the value stored in the memory address, we use the dereference operator (*).

For example: If a number variable is stored in the memory address 0x123, and it contains a value 5.

The reference (&) operator gives the value 0x123, while the dereference (*) operator gives the value 5.

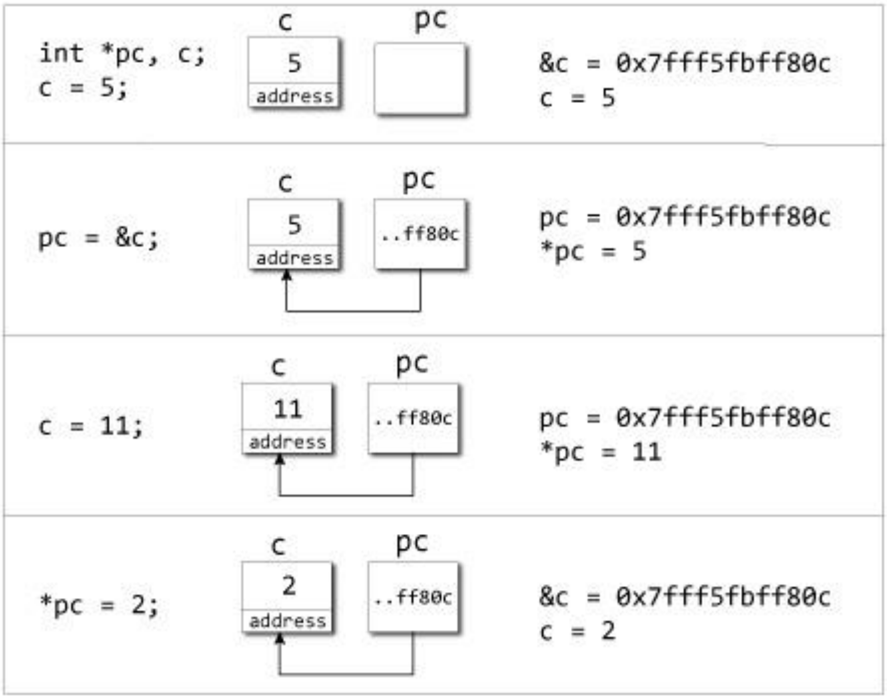
Note: The (*) sign used in the declaration of C++ pointer is not the dereference pointer. It is just a similar notation that creates a pointer.

Example 2: C++ Pointers

C++ Program to demonstrate the working of pointer.

```
#include <iostream>

using namespace std;

int main() {

   int *pc, c;



   c = 5;
```

```
    cout << "Address of c (&c): " << &c << endl;
    cout << "Value of c (c): " << c << endl << endl;

    pc = &c;    // Pointer pc holds the memory address of variable c
    cout << "Address that pointer pc holds (pc): "<< pc << endl;
    cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;

    c = 11;    // The content inside memory address &c is changed from 5 to 11.
    cout << "Address pointer pc holds (pc): " << pc << endl;
    cout << "Content of the address pointer pc holds (*pc): " << *pc << endl << endl;

    *pc = 2;
    cout << "Address of c (&c): " << &c << endl;
    cout << "Value of c (c): " << c << endl << endl;

    return 0;
}
```



```
int *pc, c;
c = 5;
```

```
&c = 0x7fff5fbff80c
c = 5
```

```
pc = &c;
```

```
pc = 0x7fff5fbff80c
*pc = 5
```

```
c = 11;
```

```
pc = 0x7fff5fbff80c
*pc = 11
```

```
*pc = 2;
```

```
&c = 0x7fff5fbff80c
c = 2
```

Explanation of program

When c = 5; the value 5 is stored in the address of variable c - 0x7fff5fbff8c.

When pc = &c; the pointer pc holds the address of c - 0x7fff5fbff8c, and the expression (dereference operator) *pc outputs the value stored in that address, 5.

When c = 11; since the address pointer pc holds is the same as c - 0x7fff5fbff8c, change in the value of c is also reflected when the expression *pc is executed, which now outputs 11.

When *pc = 2; it changes the content of the address stored by pc - 0x7fff5fbff8c. This is changed from 11 to 2. So, when we print the value of c, the value is 2 as well.

Pointers are the variables that hold address. Not only can pointers store address of a single variable, it can also store address of cells of an array.

Consider this example:

int* ptr;

int a[5];

ptr = &a[2]; // &a[2] is the address of third element of a[5].

12.2 Pointer and Array

Pointers are the variables that hold address. Not only can pointers store address of a single variable, it can also store address of cells of an array.

Consider this example:

int* ptr;

int a[5];

ptr = &a[2]; // &a[2] is the address of third element of a[5].

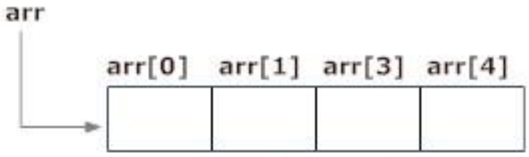


Figure: Array as Pointer

C++ Program to display address of elements of an array using both array and pointers

```
#include <iostream>
using namespace std;
int main()
{
    float arr[5];
    float *ptr;
    cout << "Displaying address using arrays: " << endl;
    for (int i = 0; i < 5; ++i)
    {
        cout << "&arr[" << i << "] = " << &arr[i] << endl;
    }
    // ptr = &arr[0]
    ptr = arr;
    cout<<"\nDisplaying address using pointers: "<< endl;
    for (int i = 0; i < 5; ++i)
    {
        cout << "ptr + " << i << " = "<< ptr + i << endl;
    }
    return 0;
}
```

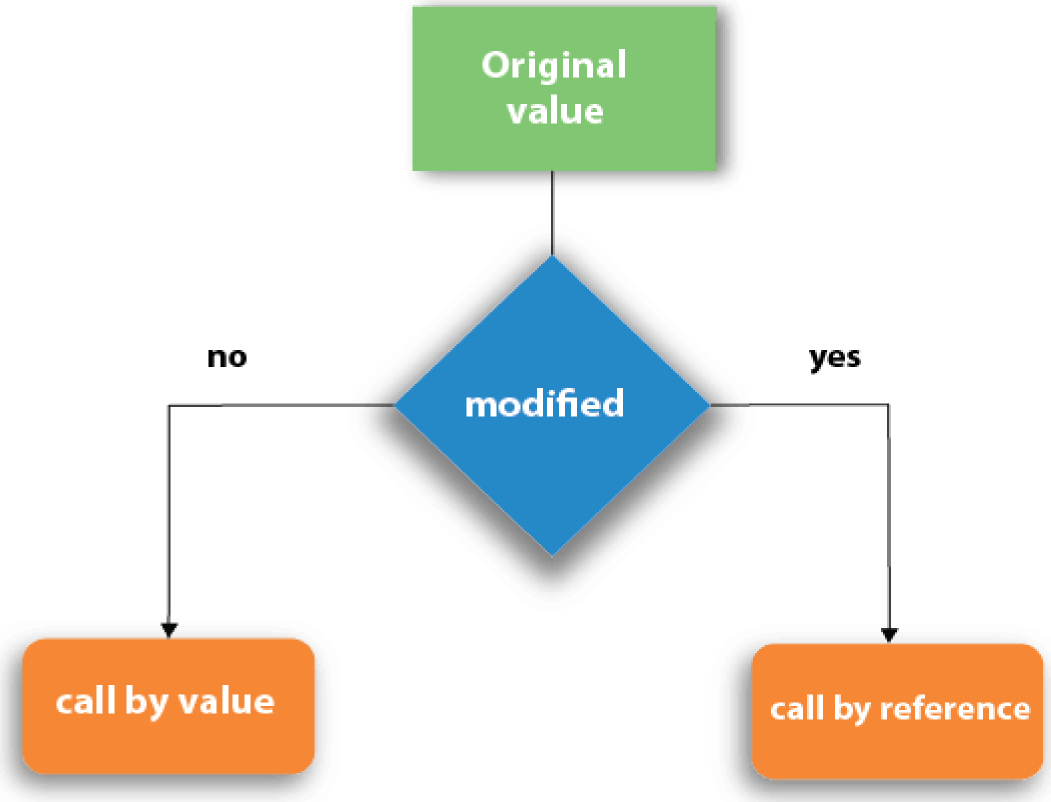
C++ Program to display address of array elements using pointer notation.

```
#include <iostream>
using namespace std;
int main() {
    float arr[5];
    cout<<"Displaying address using pointers notation: "<< endl;
    for (int i = 0; i < 5; ++i) {
        cout << arr + i <<endl;
    }
    return 0;
}
```

C++ Program to insert and display data entered by using pointer notation.

```
#include <iostream>
using namespace std;
int main() {
    float arr[5];
   // Inserting data using pointer notation
    cout << "Enter 5 numbers: ";
    for (int i = 0; i < 5; ++i) {
        cin >> *(arr + i) ;
    }
    // Displaying data using pointer notation
    cout << "Displaying data: " << endl;
    for (int i = 0; i < 5; ++i) {
        cout << *(arr + i) << endl ;
    }
    return 0;
}
```

12.3 Pointer and Functions



1) Call by value

   1) In call by value method, the value of the actual parameters is copied into the formal parameters. In other words, we can say that the value of the variable is used in the function call in the call by value method.
   2) In call by value method, we can not modify the value of the actual parameter by the formal parameter.
   3) In call by value, different memory is allocated for actual and formal parameters since the value of the actual parameter is copied into the formal parameter.
   4) The actual parameter is the argument which is used in the function call whereas formal parameter is the argument which is used in the function definition.

Call by Value Example: Swapping the values of the two variables

```
#include <iostream>
using namespace std;
// Function prototype
void swap(int, int);
int main()
{
    int a = 1, b = 2;
    cout << "Before swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    swap(a, b);
    cout << "\nAfter swapping" << endl;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
    return 0;
}

void swap(int n1, int n2) {
    int temp;
    temp = n1;
    n1 = n2;
    n2 = temp;
}
```

Output

```
Before swapping
a = 1
b = 2
```

```
After swapping

a = 2

b = 1
```

2) Call by reference
    1) In call by reference, the address of the variable is passed into the function call as the actual parameter.
    2) The value of the actual parameters can be modified by changing the formal parameters since the address of the actual parameters is passed.
    3) In call by reference, the memory allocation is similar for both formal parameters and actual parameters. All the operations in the function are performed on the value stored at the address of the actual parameters, and the modified value gets stored at the same address.

Call by reference Example: Swapping the values of the two variables

```
#include <iostream>

using namespace std;

// Function prototype

void swap(int*, int*);

int main()

{

    int a = 1, b = 2;

    cout << "Before swapping" << endl;

    cout << "a = " << a << endl;

    cout << "b = " << b << endl;

    swap(&a, &b);

    cout << "\nAfter swapping" << endl;

    cout << "a = " << a << endl;

    cout << "b = " << b << endl;

    return 0;

}

void swap(int* n1, int* n2) {

    int temp;

    temp = *n1;
    *n1 = *n2;

    *n2 = temp;

}
```

Output

```
Before swapping the values in main a = 10, b = 20

After swapping values in function a = 20, b = 10

After swapping values in main a = 20, b = 10
```

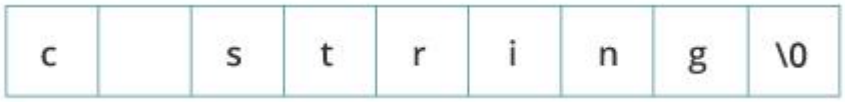
No.	Call by value	Call by reference
1	A copy of the value is passed into the function	An address of value is passed into the function
2	Changes made inside the function is limited to the function only. The values of the actual parameters do not change by changing the formal parameters.	Changes made inside the function validate outside of the function also. The values of the actual parameters do change by changing the formal parameters.
3	Actual and formal arguments are created at the different memory location	Actual and formal arguments are created at the same memory location

14 Strings in C++

In C programming, a string is a sequence of characters terminated with a null character \0. For example:

```
char c[] = "c string";
```

When the compiler encounters a sequence of characters enclosed in the double quotation marks, it appends a null character \0 at the end by default.

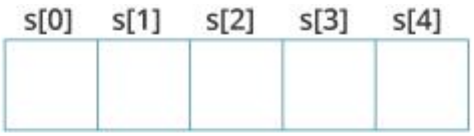


c		s	t	r	i	n	g	\0

How to declare a string?

Here's how you can declare strings:

```
char s[5];
```

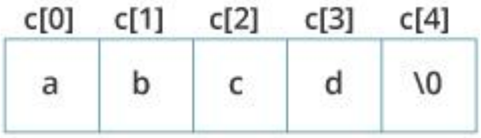


s[0]	s[1]	s[2]	s[3]	s[4]

How to initialize strings?

You can initialize strings in a number of ways.

```
char c[] = "abcd";

char c[50] = "abcd";

char c[] = {'a', 'b', 'c', 'd', '\0'};

char c[5] = {'a', 'b', 'c', 'd', '\0'};
```



c[0]	c[1]	c[2]	c[3]	c[4]
a	b	c	d	\0

Assigning Values to Strings

Arrays and strings are second-class citizens in C; they do not support the assignment operator once it is declared. For example,

```
char c[100];

c = "C programming";  // Error! array type is not assignable.
```

Example 1: C++ String to read a word

C++ program to display a string entered by user.

```
#include <iostream>

using namespace std;

int main()

{

    char str[100];

    cout << "Enter a string: ";

    cin >> str;

    cout << "You entered: " << str << endl;

    cout << "\nEnter another string: ";

    cin >> str;

    cout << "You entered: "<<str<<endl;

    return 0;

}
```

Output

```
Enter a string: C++

You entered: C++


Enter another string: Programming is fun.

You entered: Programming
```

Example 2: C++ String to read a line of text

C++ program to read and display an entire line entered by user.

```
#include <iostream>
using namespace std;
int main()
{
    char str[100];
    cout << "Enter a string: ";
    cin.get(str, 100);
    cout << "You entered: " << str << endl;
    return 0;
}
```

Output

```
Enter a string: Programming is fun.
You entered: Programming is fun.
```

Example 3: C++ string using string data type

```
#include <iostream>
using namespace std;
int main()
{
    // Declaring a string object
    string str;
                cout << "Enter a string: ";
    getline(cin, str);
    cout << "You entered: " << str << endl;
    return 0;
}
```

Output

```
Enter a string: Programming is fun.

You entered: Programming is fun.
```

Passing String to a Function

Strings are passed to a function in a similar way arrays are passed to a function.

```
#include <iostream>

using namespace std;

void display(char *);

void display(string);

int main()

{

    string str1;

    char str[100];

    cout << "Enter a string: ";

    getline(cin, str1);

    cout << "Enter another string: ";

    cin.get(str, 100, '\n');

    display(str1);

    display(str);

    return 0;

}

void display(char s[])

{

    cout << "Entered char array is: " << s << endl;

}

void display(string s)

{
    cout << "Entered string is: " << s << endl;

}
```

Output

```
Enter a string:  Programming is fun.

Enter another string:  Really?

Entered string is: Programming is fun.

Entered char array is: Really?
```

String Library functions-

Commonly Used String Functions

1)strlen() - calculates the length of a string

2)strcpy() - copies a string to another

3)strcmp() - compares two strings

4)strcat() - concatenates two strings

Strings handling functions are defined under "string.h" header file.

1) **Strlen-**
   The strlen() function takes a string as an argument and returns its length. The returned value is of type long int.

   It is defined in the <string.h> header file.

```
// c program to demonstrate

// example of strlen() function.

#include <stdio.h>

#include <string.h>

int main()

{

        char ch[] = { 'V', 'I', 'N',

                                        'I', 'T', '\0' };
```

```
    printf("Length of string is: %lu",
        strlen(ch));

    return 0;
}
```

2) **strcpy():**-

The function prototype of strcpy() is:

char* strcpy(char* destination, const char* source);

The strcpy() function copies the string pointed by source (including the null character) to the destination.

The strcpy() function also returns the copied string.

The strcpy() function is defined in the string.h header file.

Example: C strcpy()

```
// C program to illustrate
// strcpy() function ic C/C++
#include <stdio.h>
#include <string.h>
int main()
{
    char str1[] = "Hello Vineet!";
    char str2[] = "VineetforKapoor";
    char str3[40];
    char str4[40];
    char str5[] = "VKS";

    strcpy(str2, str1);
    strcpy(str3, "Copy successful");
```

```
        strcpy(str4, str5);

        printf("str1: %s\nstr2: %s\nstr3: %s\nstr4: %s\n",

                str1, str2, str3, str4);

        return 0;

}
```

3)strcat:-

The function definition of strcat() is:

```
char *strcat(char *destination, const char *source)
```

It is defined in the string.h header file.

strcat() arguments

As you can see, the strcat() function takes two arguments:

destination - destination string

source - source string

The strcat() function concatenates the destination string and the source string, and the result is stored in the destination string.

Example: C strcat() function

```
// CPP program to demonstrate

// strcat

        #include <cstring>

        #include <iostream>

        using namespace std;

        int main()

        {

                char dest[50] = "This is an";

                char src[50] = " example";

                strcat(dest, src);

                cout << dest;

                return 0;

        }
```

4)strcmp()

The strcmp() function compares two strings and returns 0 if both strings are identical.

C strcmp() Prototype

int strcmp (const char* str1, const char* str2);

The strcmp() function takes two strings and returns an integer.

The strcmp() compares two strings character by character.

If the first character of two strings is equal, the next character of two strings are compared. This continues until the corresponding characters of two strings are different or a null character '\0' is reached.

It is defined in the string.h header file.

Return Value from strcmp()

Return Value	Remarks
0	if both strings are identical (equal)
negative	if the ASCII value of the first unmatched character is less than second.
positive integer	if the ASCII value of the first unmatched character is greater than second.

Example: C strcmp() function

```
// C program to illustrate

// strcmp() function

#include <stdio.h>

#include <string.h>

int main()

{


        char leftStr[] = "g f g";

        char rightStr[] = "g f g";

        // Using strcmp()

        int res = strcmp(leftStr, rightStr);
        if (res == 0)
                printf("Strings are equal");
        else
                printf("Strings are unequal");

        printf("\nValue returned by strcmp() is: %d", res);
        return 0;
}
```

Task-Creating own functions for String manipulations.

15.Structure and Union

Structure is a collection of variables of different data types under a single name. It is similar to a class in that, both holds a collecion of data of different data types.

For example: You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables name, citNo, salary to store these information separately.

However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person: name1, citNo1, salary1, name2, citNo2, salary2

You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.

A better approach will be to have a collection of all related information under a single name Person, and use it for every person. Now, the code looks much cleaner, readable and efficient as well.

This collection of all related information under a single name Person is a structure.

Declaration of Structure

```
struct Person

{

   char name[50];

   int age;

   float salary;

};
```

Here a structure person is defined which has three members: name, age and salary

Once you declare a structure person as above. You can define a structure variable as:

```
Person bill;
```

The members of structure variable is accessed using a dot (.) operator.

Suppose, you want to access age of structure variable bill and assign it 50 to it. You can perform this task by using following code below:

```
bill.age = 50;
```

Example: C++ Structure

C++ Program to assign data to members of a structure variable and display it.

```
#include <iostream>

using namespace std;

struct Person

{

    char name[50];

    int age;

    float salary;

};


int main()

{

    Person p1;


    cout << "Enter Full name: ";

    cin.get(p1.name, 50);

    cout << "Enter age: ";

    cin >> p1.age;

    cout << "Enter salary: ";

    cin >> p1.salary;


    cout << "\nDisplaying Information." << endl;

    cout << "Name: " << p1.name << endl;

    cout <<"Age: " << p1.age << endl;

    cout << "Salary: " << p1.salary;


    return 0;
}
```

Passing structure to function-

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

struct Person
{
    char name[50];
    int age;
    float salary;
};

void displayData(Person);   // Function declaration

int main()
{
    Person p;

    cout << "Enter Full name: ";
    cin.get(p.name, 50);
    cout << "Enter age: ";
    cin >> p.age;
    cout << "Enter salary: ";
    cin >> p.salary;

    // Function call with structure variable as an argument
    displayData(p);


    return 0;
}


void displayData(Person p)
{
    cout << "\nDisplaying Information." << endl;
    cout << "Name: " << p.name << endl;
    cout <<"Age: " << p.age << endl;
    cout << "Salary: " << p.salary;
}
```

Returning structure to functions

```
include <iostream>
using namespace std;


struct Person {
    char name[50];
    int age;
    float salary;
};


Person getData(Person);
void displayData(Person);
int main()
{
    Person p;
    p = getData(p);
    displayData(p);

    return 0;
}

Person getData(Person p) {

    cout << "Enter Full name: ";
    cin.get(p.name, 50);

    cout << "Enter age: ";
    cin >> p.age;

    cout << "Enter salary: ";
    cin >> p.salary;
    return p;
}
void displayData(Person p)
{
    cout << "\nDisplaying Information." << endl;
    cout << "Name: " << p.name << endl;
    cout <<"Age: " << p.age << endl;
    cout << "Salary: " << p.salary;
}
```

Unions:

A union is a user-defined type similar to structs in C programming.

How to define a union?

We use the union keyword to define unions. Here's an example:

```
union car

{

  char name[50];

  int price;

};
```

The above code defines a derived type union car.

Create union variables

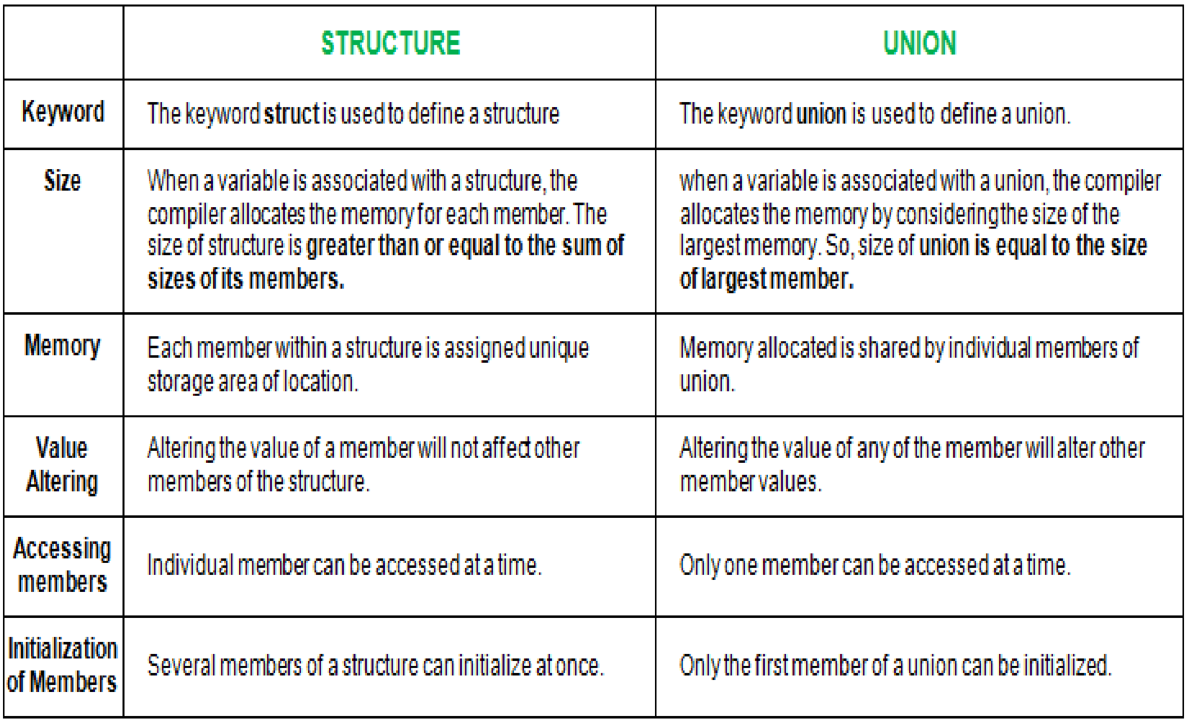
When a union is defined, it creates a user-defined type. However, no memory is allocated. To allocate memory for a given union type and work with it, we need to create variables.

```
#include<iostream>

#include<stdio.h>

using namespace std;


    union Employee

    {

        int Id;

        char Name[25];

        int Age;

        long Salary;

    };

    int main()

    {

        Employee E;

            cout << "\nEnter Employee Id : ";

            cin >> E.Id;

            cout << "Employee Id : " << E.Id;

            cout << "\n\nEnter Employee Name : ";

            cin >> E.Name;

            cout << "Employee Name : " << E.Name;
```

```
                    cout << "\n\nEnter Employee Age : ";
                    cin >> E.Age;
                    cout << "Employee Age : " << E.Age;
                    cout << "\n\nEnter Employee Salary : ";
                    cin >> E.Salary;
                    cout << "Employee Salary : " << E.Salary;
return 0;
   }
```

Comparing the size of structure and union-

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

        struct Employee1
        {
                char name[32];
  float salary;
  int workerNo;

        };
        union Employee2
        {
                        char name[32];
  float salary;
  int workerNo;

        };
         int main()
         {
                Employee1 e1;

                Employee2 e2;

                        cout << "\nSize of Employee1 is : " << sizeof(e1);

                        cout << "\nSize of Employee2 is : " << sizeof(e2);

return 0;

      }       Output :


                        Size of Employee1 is : 40

                        Size of Employee2 is : 32
```



	STRUCTURE	UNION
Keyword	The keyword **struct** is used to define a structure	The keyword **union** is used to define a union.
Size	When a variable is associated with a structure, the compiler allocates the memory for each member. The size of structure is **greater than or equal to the sum of sizes of its members.**	when a variable is associated with a union, the compiler allocates the memory by consideringthe size of the largest memory. So, size of **union is equal to the size of largest member.**
Memory	Each member within a structure is assigned unique storage area of location.	Memory allocated is shared by individual members of union.
Value Altering	Altering the value of a member will not affect other members of the structure.	Altering the value of any of the member will alter other member values.
Accessing members	Individual member can be accessed at a time.	Only one member can be accessed at a time.
Initialization of Members	Several members of a structure can initialize at once.	Only the first member of a union can be initialized.

16.File Handling

Why files are needed?

When a program is terminated, the entire data is lost. Storing in a file will preserve your data even if the program terminates.

If you have to enter a large number of data, it will take a lot of time to enter them all.

However, if you have a file containing all the data, you can easily access the contents of the file using a few commands in C.

You can easily move your data from one computer to another without any changes.

Types of Files

When dealing with files, there are two types of files you should know about:

1) Text files

2) Binary files

1. Text files

Text files are the normal .txt files. You can easily create text files using any simple text editors such as Notepad.

When you open those files, you'll see all the contents within the file as plain text. You can easily edit or delete the contents.

They take minimum effort to maintain, are easily readable, and provide the least security and takes bigger storage space.

2. Binary files

Binary files are mostly the .bin files in your computer.

Instead of storing data in plain text, they store it in the binary form (0's and 1's).

They can hold a higher amount of data, are not readable easily, and provides better security than text files.

File Operations

In C, you can perform four major operations on files, either text or binary:

1)Creating a new file

2)Opening an existing file

3)Closing a file

4) Reading from and writing information to a file

1)Opening a File

Generally, the first operation performed on an object of one of these classes is to associate it to a real file. This procedure is known to open a file.

We can open a file using any one of the following methods:

1. First is bypassing the file name in constructor at the time of object creation.

2. Second is using the open() function.

To open a file use

open() function

Syntax

void open(const char* file_name,ios::openmode mode);

Here, the first argument of the open function defines the name and format of the file with the address of the file.

The second argument represents the mode in which the file has to be opened. The following modes are used as per the requirements.

Modes

1) In-Opens the file to read(default for ifstream)
2) Out-Opens the file to write(default for ofstream)
3) Binary-Opens the file in binary mode
4) App-Opens the file and appends all the outputs at the end
5) Ate-Opens the file and moves the control to the end of the file
6) Trunk-Removes the data in the existing file
7) Nocreate-Opens the file only if it already exists
8) Noreplace-Opens the file only if it does not already exist

Example

fstream new_file;

new_file.open(“newfile.txt”, ios::out);

In the above example, new_file is an object of type fstream, as we know fstream is a class so we need to create an object of this class to use its member functions. So we create new_file object and call open() function. Here we use out mode that allows us to open the file to write in it.

Default Open Modes :

ifstream ios::in

ofstream ios::out

fstream ios::in | ios::out

We can combine the different modes using or symbol | .

Example

```
ofstream new_file;

new_file.open(“new_file.txt”, ios::out | ios::app );
```

Here, input mode and append mode are combined which represents the file is opened for writing and appending the outputs at the end.

As soon as the program terminates, the memory is erased and frees up the memory allocated and closes the files which are opened.

But it is better to use the close() function to close the opened files after the use of the file.

Using a stream insertion operator << we can write information to a file and using stream extraction operator >> we can easily read information from a file.

Example of opening/creating a file using the open() function

```
#include<iostream>

#include <fstream>

using namespace std;

int main()

{

fstream new_file;

new_file.open("new_file",ios::out);

if(!new_file)

{

cout<<"File creation failed";

}

else

{

cout<<"New file created";
new_file.close(); // Step 4: Closing file
}
return 0;
}
```

Explanation

In the above example we first create an object to class fstream and name it ‘new_file’. Then we apply the open() function on our ‘new_file’ object. We give the name ‘new_file’ to the new file we wish to create and we set the mode to ‘out’ which allows us to write in our file. We use a ‘if’ statement to find if the file already exists or not if it does exist then it will going to print “File creation failed” or it will gonna create a new file and print “New file created”.

Writing to a File

Example:

```
#include <iostream>
#include <fstream>
using namespace std;
int main()
{
fstream new_file;
new_file.open("new_file_write.txt",ios::out);
if(!new_file)
{
cout<<"File creation failed";
}
else
{
cout<<"New file created";
new_file<<"Learning File handling";    //Writing to file
new_file.close();
}
return 0;
}
```

Explanation

Here we first create a new file “new_file_write” using open() function since we wanted to send output to the file so, we use ios::out. As given in the program, information typed inside the quotes after Insertion Pointer “<<” got passed to the output file.

Reading from a File

Example

```
#include <iostream>

#include <fstream>

using namespace std;

int main()

{

fstream new_file;

new_file.open("filename.txt",ios::in);

if(!new_file)

cout<<"No such file";

else

{

    char ch;

    while (!new_file.eof())

    {

        new_file >>ch;

        cout << ch;

    }

}

new_file.close();

return 0;

}
```

Explanation

In this example, we read the file that generated id previous example i.e. new_file_write.

To read a file we need to use ‘in’ mode with syntax ios::in. In the above example, we print the content of the file using extraction operator >>. The output prints without any space because we use only one character at a time, we need to use getline() with a character array to print the whole line as it is.

Close a File

It is simply done with the help of close() function.

Syntax: File Pointer.close()

Example

```
#include <iostream>

#include <fstream>

using namespace std;

int main()

{

fstream new_file;

new_file.open("new_file.txt",ios::out);

new_file.close();

return 0;

}
```

Output:

The file gets closed.

17 Command line Argument-

Command line argument is a parameter supplied to the program when it is invoked. Command line argument is an important concept in C programming. It is mostly used when you need to control your program from outside. Command line arguments are passed to the main() method.

Syntax:

int main(int argc, char *argv[])

Here argc counts the number of arguments on the command line and argv[ ] is a pointer array which holds pointers of type char which points to the arguments passed to the program.

Example for Command Line Argument

```
#include <stdio.h>

int main( int argc, char *argv[] ) {

  if( argc == 2 ) {

    cout<<argv[1]

  }

  else if( argc > 2 ) {
        cout<<"Too many arguments supplied.\n";
    }
    else {
        cout<<"One argument expected.\n";
    }
}
```

How to Run-

1. Open DOS Shell

2. First use cd.. for coming back to Turbo C++ main directory

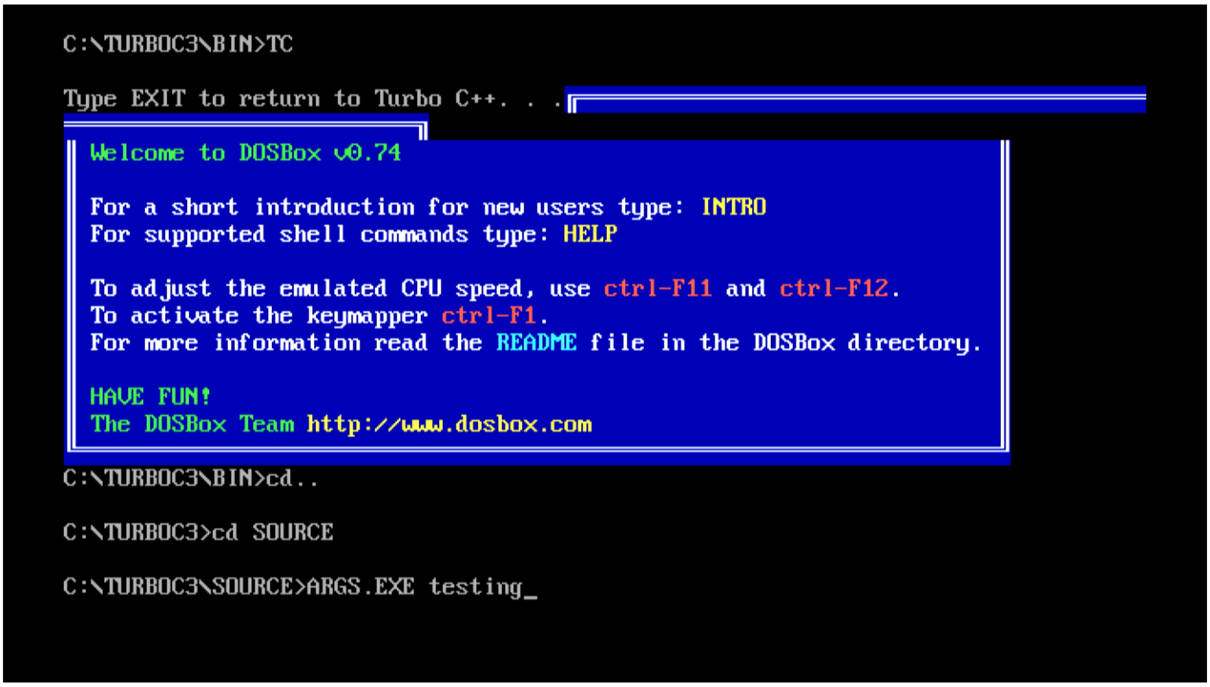
```
cd..
```

3. Now use cd SOURCE to access the SOURCE directory

```
cd SOURCE
```

4.Execute Program with Command Line Arguments

ARGS.EXE testing



```
C:\TURBOC3\BIN>TC

Type EXIT to return to Turbo C++. . .
  Welcome to DOSBox v0.74

  For a short introduction for new users type: INTRO
  For supported shell commands type: HELP

  To adjust the emulated CPU speed, use ctrl-F11 and ctrl-F12.
  To activate the keymapper ctrl-F1.
  For more information read the README file in the DOSBox directory.

  HAVE FUN!
  The DOSBox Team http://www.dosbox.com
C:\TURBOC3\BIN>cd..

C:\TURBOC3>cd SOURCE

C:\TURBOC3\SOURCE>ARGS.EXE testing_
```

18 Exception Handing

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: try, catch, and throw.

throw − A program throws an exception when a problem shows up. This is done using a throw keyword.

catch − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

try − A try block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows −

```
try {

   // protected code
} catch( ExceptionName e1 ) {
   // catch block
} catch( ExceptionName e2 ) {
   // catch block
} catch( ExceptionName eN ) {
   // catch block
}
```

You can list down multiple catch statements to catch different type of exceptions in case your try block raises more than one exception in different situations.

Throwing Exceptions

Exceptions can be thrown anywhere within a code block using throw statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs −

```
double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}
```

Catching Exceptions

The catch block following the try block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {
   // protected code
} catch( ExceptionName e ) {
  // code to handle ExceptionName exception
}
```

Above code will catch an exception of ExceptionName type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows −

```
try {
   // protected code
} catch(...) {
  // code to handle any exception
}
```

The following is an example, which throws a division by zero exception and we catch it in catch block.

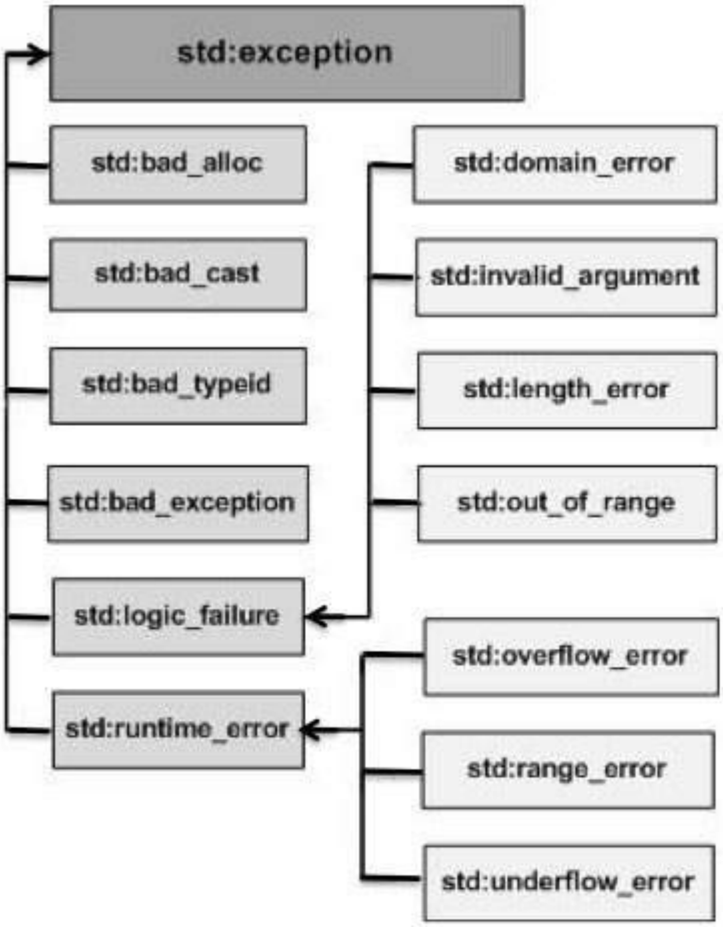
```
#include <iostream>
using namespace std;
double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}
int main () {
   int x = 50;
   int y = 0;
   double z = 0;
   try {
      z = division(x, y);
      cout << z << endl;
   } catch (const char* msg) {
     cerr << msg << endl;
   }
   return 0;
}
```

Because we are raising an exception of type const char*, so while catching this exception, we have to use const char* in catch block. If we compile and run above code, this would produce the following result −

```
Division by zero condition!
```

C++ Standard Exceptions

C++ provides a list of standard exceptions defined in <exception> which we can use in our programs. These are arranged in a parent-child class hierarchy shown below –



C++ Exceptions Hierarchy

Here is the small description of each exception mentioned in the above hierarchy −

Sr.No Exception & Description

1std::exception An exception and parent class of all the standard C++ exceptions.

2 std::bad_alloc-This can be thrown by new.

3 std::bad_cast-This can be thrown by dynamic_cast.

4 std::bad_exception-This is useful device to handle unexpected exceptions in a C++ program.

5 std::bad_typeid-This can be thrown by typeid.

6 std::logic_error-An exception that theoretically can be detected by reading the code.

7 std::domain_error-This is an exception thrown when a mathematically invalid domain is used.

8-std::invalid_argument-This is thrown due to invalid arguments.

9 std::length_error-This is thrown when a too big std::string is created.

10 std::out_of_range-This can be thrown by the 'at' method, for example a std::vector and std::bitset<>::operator[]().

11 std::runtime_error-An exception that theoretically cannot be detected by reading the code.

12 std::overflow_error-This is thrown if a mathematical overflow occurs.

13 std::range_error-This is occurred when you try to store a value which is out of range.

14 std::underflow_error-This is thrown if a mathematical underflow occurs.

Define New Exceptions

You can define your own exceptions by inheriting and overriding exception class functionality. Following is the example, which shows how you can use std::exception class to implement your own exception in standard way −

```
#include <iostream>

#include <exception>

using namespace std;


struct MyException : public exception {

   const char * what () const throw () {
      return "C++ Exception";
   }
};

int main() {
   try {
      throw MyException();
   } catch(MyException& e) {
      std::cout << "MyException caught" << std::endl;
      std::cout << e.what() << std::endl;
   } catch(std::exception& e) {
      //Other errors
   }
}
```

This would produce the following result −

```
MyException caught
C++ Exception
```

19 Memory management in C++

Dynamic Memory Allocation In C++ is a very important feature that lets you consider your requirements to deal with the need for real time resources.

Need for Dynamic memory allocation?

Let say, we want to input a sentence as an array of characters but we are not sure about the exact number of characters required in the array.

Now, while declaring the character array, if we specify its size smaller than the size of the desired string, then we will get an error because the space in the memory allocated to the array is lesser compared to the size of the input string. If we specify its size larger than the size of the input string, then the array will be allocated a space in the memory which is much larger than the size of the desired string, thus unnecessarily consuming more memory even when it is not required.

In the above case, we don’t have the idea about the exact size of the array until the compile-time (when computer compiles the code and the string is input by the user). In such cases, we use the new operator.

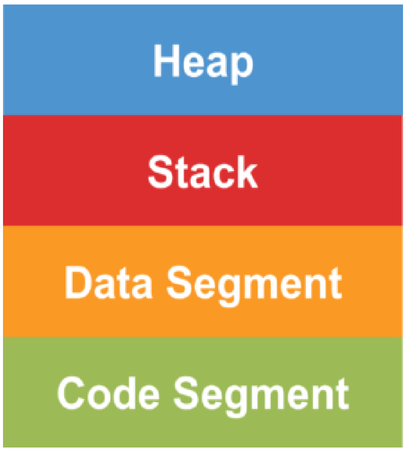
C++ defines two unary operators new and delete that perform the task of allocating and deallocating memory during runtime. Since these operators (new and delete) operate upon free store memory (Heap memory), they are also called free store operator. Pointers provide the necessary support for dynamic memory allocation system in C++.

With the help of Dynamic Allocation, a program can obtain memory during runtime.

The global and local variables are allocated to memory during compile-time. However, we cannot add any global or local variables during runtime. If the program needs to use a variable amount of memory we would need to allocate memory during runtime, as and when needed. And of course, here the dynamic allocation routines can serve the purpose.

Differences between Static memory allocation and Dynamic memory allocation:

This is a basic memory architecture used for any C++ program:



The stack is used for static memory allocation and Heap for dynamic memory allocation, both are stored in the computer's RAM.

Variables that get allocated on the stack while static memory allocation is stored directly to the memory and access to this memory is very fast, also its allocation is dealt with when the program is compiled. When a function or a method calls another function which might in turn calls another function and so on, the execution of all these functions remains suspended until the very last function returns its value. The stack is always stored in a LIFO(last in first out) order, the most recently reserved block is always the next block to be freed. This helps to keep track of the stack, freeing a block from the stack is nothing more than adjusting one pointer.

Variables allocated on the heap have their memory allocated at run time while dynamic memory allocation. Accessing this memory is a bit slower as compared to stack, but the size of the heap is only limited by the size of virtual memory. The element of the heap has no dependencies with each other and can always be accessed randomly at any moment of time. We can allocate a block at any time and free it at any time. This makes it difficult to keep track of which parts of the heap are allocated or deallocated at any given time.

Allocation of Memory using new Keyword

In C++ the new operator is used to allocate memory at runtime and the memory is allocated in bytes. The new operator denotes a request for dynamic memory allocation on the Heap. If sufficient memory is available then the new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax:

```
datatype *pointer_name = new datatype
```

Example:

```
int *ptr = new int;

//We can declare a variable while dynamical allocation in the following two ways.

int *ptr = new int (10);
int *ptr = new int {15};

// new operator is also used to allocate a block(an array) of memory of type data-type.

int *ptr = new int[20];

// The above statement dynamically allocates memory for 20 integers continuously of type int and returns a pointer to the

first element of the sequence to ‘ptr’ pointer.
```

Deallocation of memory using delete Keyword:

Once heap memory is allocated to a variable or class object using the new keyword, we can deallocate that memory space using the delete keyword.

Syntax:

```
delete pointer_variable;

// Here, pointer_variable is the pointer that points to the data object created by new.

delete[] pointer_variable;

//To free the dynamically allocated array memory pointed by pointer-variable we use the following form of delete:
```

Example:

```
delete ptr;

delete[] ptr;
```

Dynamically Allocating Arrays

The major use of the concept of dynamic memory allocation is for allocating memory to an array when we have to declare it by specifying its size but are not sure about it.

Let’s see, an example to understand its usage.

```
#include <iostream>

using namespace std;

int main()

{

int len, sum = 0;

cout << "Enter the no. of students in the class" << endl; cin >> len;

int *marks = new int[len];  //Dynamic memory allocation

cout << "Enter the marks of each student" << endl;
for( int i = 0; i < len; i++ ) { cin >> *(marks+i);

}

for( int i = 0; i < len; i++ )

{

sum += *(marks+i);

}

cout << "sum is " << sum << endl;

return 0;

}
```

20 OOPS Concept-

20.1 Basic Concepts

The major purpose of C++ programming is to introduce the concept of object orientation to the C programming language.

Object Oriented Programming is a paradigm that provides many concepts such as inheritance, data binding, polymorphism etc.

OOPs (Object Oriented Programming System)

Object means a real word entity such as pen, chair, table etc. Object-Oriented Programming is a methodology or paradigm to design a program using classes and objects. It simplifies the software development and maintenance by providing some concepts:

1) Object
2) Class
3) Inheritance
4) Polymorphism
5) Abstraction
6) Encapsulation

Object

Any entity that has state and behavior is known as an object. For example: chair, pen, table, keyboard, bike etc. It can be physical and logical.

Class

Collection of objects is called class. It is a logical entity.

Inheritance

When one object acquires all the properties and behaviours of parent object i.e. known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

Polymorphism

When one task is performed by different ways i.e. known as polymorphism. For example: to convince the customer differently, to draw something e.g. shape or rectangle etc.

In C++, we use Function overloading and Function overriding to achieve polymorphism.

Abstraction

Hiding internal details and showing functionality is known as abstraction. For example: phone call, we don't know the internal processing.

In C++, we use abstract class and interface to achieve abstraction.

Encapsulation

Binding (or wrapping) code and data together into a single unit is known as encapsulation. For example: capsule, it is wrapped with different medicines.

Advantage of OOPs over Procedure-oriented programming language

1) OOPs makes development and maintenance easier where as in Procedure-oriented programming language it is not easy to manage if code grows as project size grows.
2) OOPs provide data hiding whereas in Procedure-oriented programming language a global data can be accessed from anywhere.
3) OOPs provide ability to simulate real-world event much more effectively. We can provide the solution of real word problem if we are using the Object-Oriented Programming language.

20.2 Class:-

A class is a blueprint for the object.
We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.
As, many houses can be made from the same description, we can create many objects from a class.

How to define a class in C++?
A class is defined in C++ using keyword class followed by the name of class.

The body of class is defined inside the curly brackets and terminated by a semicolon at the end.

```
class className
  {
  // some data
  // some functions
  };
```

Let's see an example of C++ class that has three fields only.

```
class Student
 {
    public:
    int id;  //field or data member
    float salary; //field or data member
    String name;//field or data member
}
```

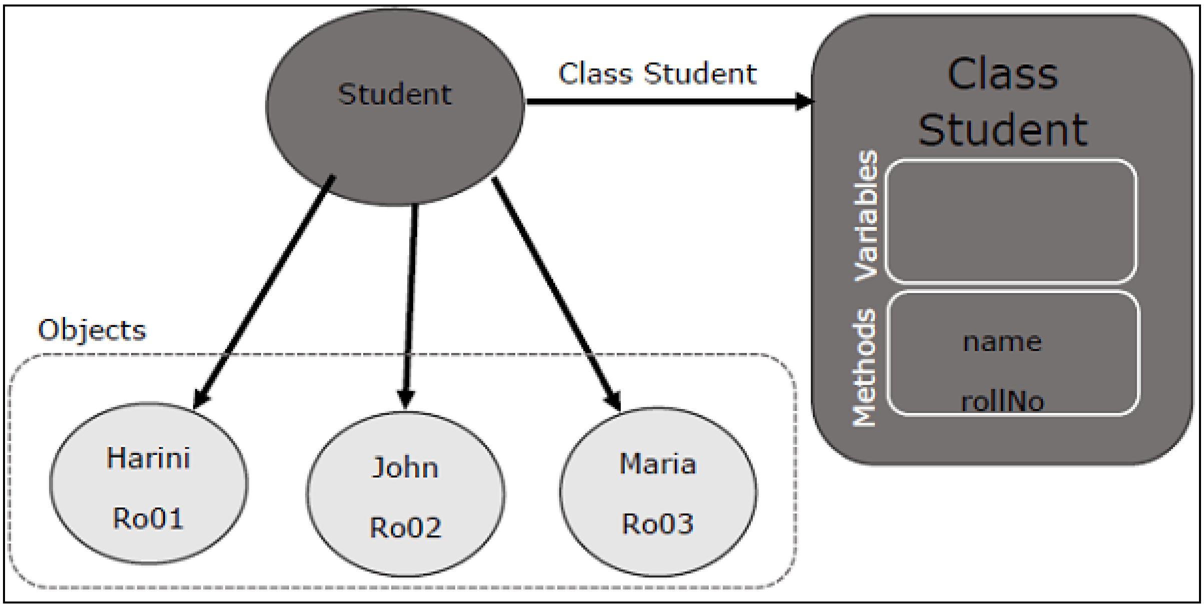
20.3 Object

In C++, Object is a real world entity, for example, chair, car, pen, mobile, laptop etc.

In other words, object is an entity that has state and behavior. Here, state means data and behavior means functionality.

Object is a runtime entity, it is created at runtime.

Object is an instance of a class. All the members of the class can be accessed through object.



Let's see an example to create object of student class using s1 as the reference variable.

```
Student s1;  //creating an object of Student
```

Example

```
#include <iostream>

using namespace std;

class Student {

  public:

     int id;//data member (also instance variable)

     string name;//data member(also instance variable)

};
```

```
int main() {
    Student s1; //creating an object of Student
    s1.id = 201;
    s1.name = "Vineet Kapoor";
    cout<<s1.id<<endl;
    cout<<s1.name<<endl;
    return 0;
}
```

Example 2: Let's see another example of C++ class where we are storing and displaying employee information using method.

```
#include <iostream>
using namespace std;
class Employee {
   public:
       int id;//data member (also instance variable)
       string name;//data member(also instance variable)
       float salary;
       void insert(int i, string n, float s)
        {
            id = i;
            name = n;
            salary = s;
        }
       void display()
        {
            cout<<id<<"  "<<name<<"  "<<salary<<endl;
        }
};
```

```
int main(void) {

    Employee e1; //creating an object of Employee

    Employee e2; //creating an object of Employee

    e1.insert(201, "Sonoo",990000);

    e2.insert(202, "Nakul", 29000);

    e1.display();

    e2.display();

    return 0;

}
```

20.4 Constructors:

In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

There can be two types of constructors in C++.

1) Default constructor
2) Parameterized constructor

1) C++ Default Constructor

A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

```
#include <iostream>

using namespace std;

class Employee

 {

   public:

        Employee()

        {

            cout<<"Default Constructor Invoked"<<endl;

        }

};
```

```
int main(void)
{
   Employee e1; //creating an object of Employee
   Employee e2;
   return 0;
}
```

Output:

```
Default Constructor Invoked
Default Constructor Invoked
```

2) C++ Parameterized Constructor

A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.

Let's see the simple example of C++ Parameterized Constructor.

```
#include <iostream>
using namespace std;
class Employee {
  public:
      int id;//data member (also instance variable)
      string name;//data member(also instance variable)
      float salary;
      Employee(int i, string n, float s)
       {
          id = i;
          name = n;
          salary = s;
       }
```

```
        void display()
         {
             cout<<id<<"  "<<name<<"  "<<salary<<endl;
         }
};
int main(void) {
    Employee e1 =Employee(101, "Vineet", 890000); //creating an object of Employee
    Employee e2=Employee(102, "Kapoor", 59000);
    e1.display();
    e2.display();
    return 0;
}
```

Output:

```
101  Vineet  890000
102  Kapoor  59000
```

20.5 Destructor:

A destructor works opposite to constructor; it destructs the objects of classes. It can be defined only once in a class. Like constructors, it is invoked automatically.

A destructor is defined like constructor. It must have same name as class. But it is prefixed with a tilde sign (~).

Note: C++ destructor cannot have parameters. Moreover, modifiers can't be applied on destructors.

Let's see an example of constructor and destructor in C++ which is called automatically.

```
#include <iostream>
using namespace std;
class Employee
 {
   public:
    Employee()
    {
      cout<<"Constructor Invoked"<<endl;
    }
    ~Employee()
    {
      cout<<"Destructor Invoked"<<endl;
    }
};
int main(void)
{
  Employee e1; //creating an object of Employee
  Employee e2; //creating an object of Employee
  return 0;
}
```

Output:

```
Constructor Invoked
Constructor Invoked
Destructor Invoked
Destructor Invoked
```

20.6 This Keyword

In C++ programming, this is a keyword that refers to the current instance of the class. There can be 3 main usage of this keyword in C++.

It can be used to pass current object as a parameter to another method.

It can be used to refer current class instance variable.

It can be used to declare indexers.

C++ this Pointer Example

Let's see the example of this keyword in C++ that refers to the fields of current class.

```
#include <iostream>
```

```
using namespace std;
class Employee {
  public:
     int id; //data member (also instance variable)
     string name; //data member(also instance variable)
     float salary;
     Employee(int id, string name, float salary)
      {
           this->id = id;
         this->name = name;
         this->salary = salary;
      }
     void display()
      {
         cout<<id<<"  "<<name<<"  "<<salary<<endl;
      }
};
int main(void) {
   Employee e1 =Employee(101, "Vineet", 890000); //creating an object of Employee
   Employee e2=Employee(102, "Kapoor", 59000); //creating an object of Employee
   e1.display();
   e2.display();
   return 0;
}
```

Output:

```
101  Vineet  890000
102  Kapoor  59000
```

20.7 Static keyword

In C++, static is a keyword or modifier that belongs to the type not instance. So instance is not required to access the static members. In C++, static can be field, method, constructor, class, properties, operator and event.

Advantage of C++ static keyword

Memory efficient: Now we don't need to create instance for accessing the static members, so it saves memory. Moreover, it belongs to the type, so it will not get memory each time when instance is created.

C++ Static Field

A field which is declared as static is called static field. Unlike instance field which gets memory each time whenever you create object, there is only one copy of static field created in the memory. It is shared to all the objects.

It is used to refer the common property of all objects such as rateOfInterest in case of Account, companyName in case of Employee etc.

C++ static field example

Let's see the simple example of static field in C++.

```
#include <iostream>

using namespace std;

class Account {

  public:

      int accno; //data member (also instance variable)

      string name; //data member(also instance variable)

      static float rateOfInterest;

      Account(int accno, string name)

       {

           this->accno = accno;

          this->name = name;

       }
```

```
        void display()
         {
             cout<<accno<< "<<name<< " "<<rateOfInterest<<endl;
         }
};
float Account::rateOfInterest=6.5;
int main(void) {
    Account a1 =Account(201, "Vineet"); //creating an object of Employee
    Account a2=Account(202, "Kapoor"); //creating an object of Employee
    a1.display();
    a2.display();
    return 0;
}
```

Output:

```
201 Vineet 6.5
202 Kapoor 6.5
```

20.8 Difference between structs and class:

Structs:-

In C++, classes and structs are blueprints that are used to create the instance of a class. Structs are used for lightweight objects such as Rectangle, color, Point, etc.

Unlike class, structs in C++ are value type than reference type. It is useful if you have data that is not intended to be modified after creation of struct.

C++ Structure is a collection of different data types. It is similar to the class that holds different types of data.

The Syntax Of Structure

```
struct structure_name
{
     // member declarations.
}
```

In the above declaration, a structure is declared by preceding the struct keyword followed by the identifier(structure name). Inside the curly braces, we can declare the member variables of different types. Consider the following situation:

```
struct Student
{
    char name[20];
     int id;
     int age;
}
```

In the above case, Student is a structure contains three variables name, id, and age. When the structure is declared, no memory is allocated. When the variable of a structure is created, then the memory is allocated. Let's understand this scenario.

How to create the instance of Structure?

Structure variable can be defined as:

```
Student s;
```

Here, s is a structure variable of type Student. When the structure variable is created, the memory will be allocated. Student structure contains one char variable and two integer variable. Therefore, the memory for one char variable is 1 byte and two ints will be 2*4 = 8. The total memory occupied by the s variable is 9 byte.

How to access the variable of Structure:

The variable of the structure can be accessed by simply using the instance of the structure followed by the dot (.) operator and then the field of the structure.

For example:

```
s.id = 4;
```

In the above statement, we are accessing the id field of the structure Student by using the dot(.) operator and assigns the value 4 to the id field.

C++ Struct Example

Let's see a simple example of struct Rectangle which has two data members width and height.

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

```
 struct Rectangle
{
   int width, height;

 };
int main(void) {
    struct Rectangle rec;
    rec.width=8;
    rec.height=5;
   cout<<"Area of Rectangle is: "<<(rec.width * rec.height)<<endl;
 return 0;
}
```

Output:

```
Area of Rectangle is: 40
```

C++ Struct Example: Using Constructor and Method

Let's see another example of struct where we are using the constructor to initialize data and method to calculate the area of rectangle.

```
#include <iostream>
using namespace std;
 struct Rectangle    {
   int width, height;
  Rectangle(int w, int h)
    {
        width = w;
        height = h;
    }
  void areaOfRectangle() {
    cout<<"Area of Rectangle is: "<<(width*height); }
 };
```

```
int main(void) {

    struct Rectangle rec=Rectangle(4,6);

    rec.areaOfRectangle();

   return 0;

}
```

Output:

```
Area of Rectangle is: 24
```

Structure v/s Class

Structure	Class
If access specifier is not declared explicitly, then by default access specifier will be public.	If access specifier is not declared explicitly, then by default access specifier will be private.

Syntax of Structure:
struct structure_name
{
// body of the structure.
}
	Syntax of Class:
	class class_name
	{
	// body of the class.
	}

The instance of the structure is known as "Structure variable".	The instance of the class is known as "Object of the class".

20.9 Inheritance

In C++, inheritance is a process in which one object acquires all the properties and behaviors of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviors which are defined in other class.

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

Types Of Inheritance

C++ supports five types of inheritance:

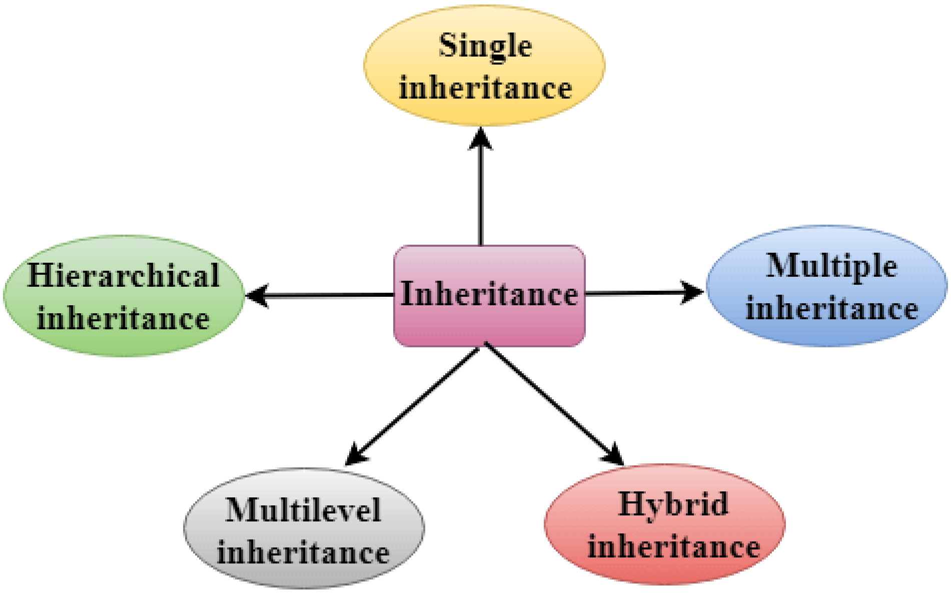
1) Single inheritance

2) Multiple inheritance

3) Hierarchical inheritance

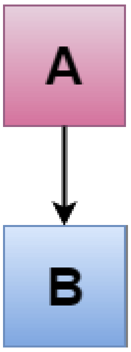
4) Multilevel inheritance

5) Hybrid inheritance



1)Single Inheritance:

Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



Where 'A' is the base class, and 'B' is the derived class.

Example:

```
#include <iostream>
using namespace std;
 class Account {
   public:
   float salary = 60000;
 };
   class Programmer: public Account {
   public:
   float bonus = 5000;
   };
int main(void) {
     Programmer p1;
     cout<<"Salary: "<<p1.salary<<endl;
     cout<<"Bonus: "<<p1.bonus<<endl;
    return 0;
}
```

Output:

```
Salary: 60000
Bonus: 5000
```

Example 2-C++ Single Level Inheritance Example: Inheriting Methods

```
#include <iostream>
using namespace std;
 class Animal {
   public:
```

```
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
       public:
     void bark(){
    cout<<"Barking...";
     }
   };
int main(void) {
    Dog d1;
    d1.eat();
    d1.bark();
    return 0;
}
```

How to make a Private Member Inheritable

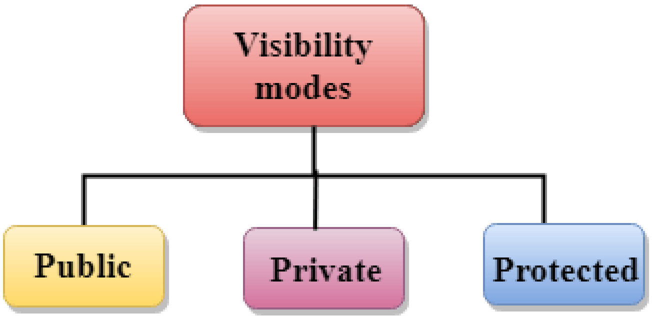
The private member is not inheritable. If we modify the visibility mode by making it public, but this takes away the advantage of data hiding.

C++ introduces a third visibility modifier, i.e., protected. The member which is declared as protected will be accessible to all the member functions within the class as well as the class immediately derived from it.

Visibility modes can be classified into three categories:

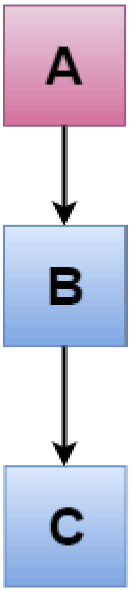
1) **Public**: When the member is declared as public, it is accessible to all the functions of the program.
2) **Private**: When the member is declared as private, it is accessible within the class only.
3) **Protected**: When the member is declared as protected, it is accessible within its own class as well as the class immediately derived from it.

Access	public	protected	private
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no



2) C++ Multilevel Inheritance

Multilevel inheritance is a process of deriving a class from another derived class.



Example:

#include <iostream>

using namespace std;

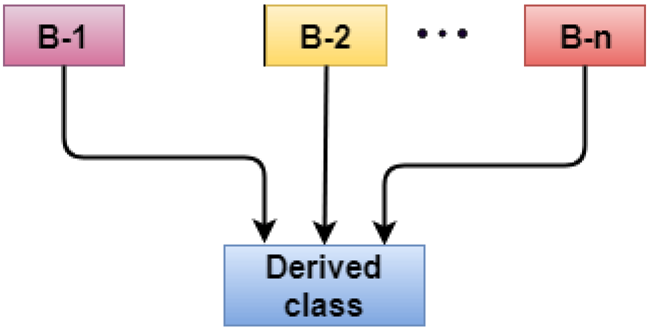
```
 class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
       public:
     void bark(){
    cout<<"Barking..."<<endl;
     }
   };
   class BabyDog: public Dog
   {
       public:
     void weep() {
    cout<<"Weeping...";
     }
   };
int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
     d1.weep();
     return 0;
}
```

Output:

```
Eating...

Barking...

Weeping...
```

3) C++ Multiple Inheritance

Multiple inheritance is the process of deriving a new class that inherits the attributes from two or more classes.



Example:

```
#include <iostream>

using namespace std;

class A

{

    protected:

     int a;

    public:

    void get_a(int n)

    {

        a = n;

    }

};


class B

{

    protected:
```

```
    int b;
    public:
    void get_b(int n)
    {
        b = n;
    }
};
class C : public A,public B
{
   public:
    void display()
    {
        std::cout << "The value of a is : " <<a<< std::endl;
        std::cout << "The value of b is : " <<b<< std::endl;
        cout<<"Addition of a and b is : "<<a+b;
    }
};
int main()
{
   C c;
   c.get_a(10);
   c.get_b(20);
   c.display();


    return 0;
}
```

Output:

```
The value of a is : 10
The value of b is : 20

Addition of a and b is : 30
```

Ambiquity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

Let's understand this through an example:

```
#include <iostream>

using namespace std;

class A

{

   public:

   void display()

   {

      std::cout << "Class A" << std::endl;

   }

};

class B

{

   public:

   void display()

   {

      std::cout << "Class B" << std::endl;

   }

};

class C : public A, public B

{

   void view()
```

```
    {
        display();
    }
};
int main()
{
    C c;
    c.display();
    return 0;
}
```

Output:

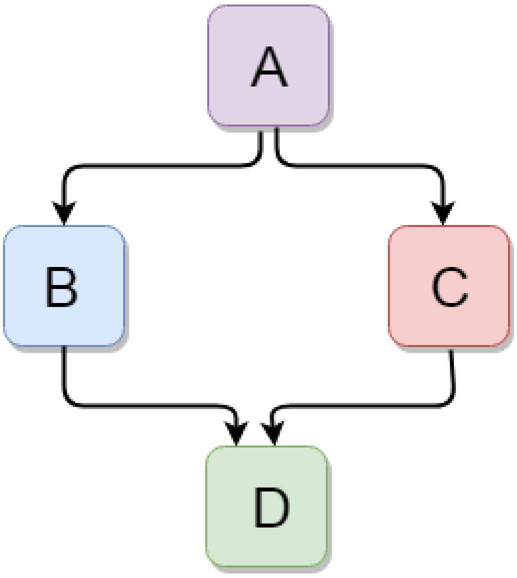
```
error: reference to 'display' is ambiguous
        display();
```

The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
class C : public A, public B
{
    void view()
    {
        A :: display();         // Calling the display() function of class A.
        B :: display();         // Calling the display() function of class B.

    }
};
```

4) C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance.



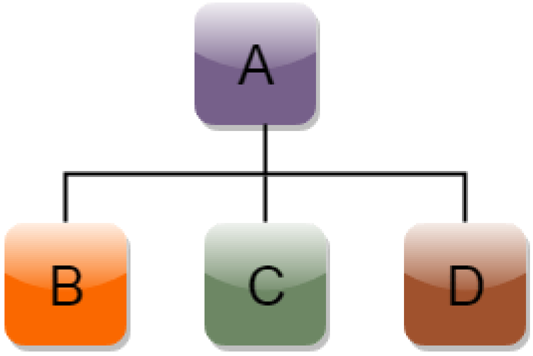
```
#include <iostream>

using namespace std;

class A

{

   protected:

   int a;

   public:

   void get_a()

   {

      std::cout << "Enter the value of 'a' : " << std::endl;

      cin>>a;

   }

};


class B : public A

{

   protected:

   int b;

   public:
   void get_b()
   {
       std::cout << "Enter the value of 'b' : " << std::endl;
      cin>>b;
   }
};
class C
{
   protected:
   int c;
   public:
   void get_c()
   {
       std::cout << "Enter the value of c is : " << std::endl;
       cin>>c;
   }
};

class D : public B, public C
{
   protected:
   int d;
   public:
   void mul()
   {
        get_a();
        get_b();
        get_c();
        std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
    }
};
int main()
{
    D d;
    d.mul();
    return 0;
}
```

Output:

```
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

5) C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



Syntax of Hierarchical inheritance:

```
class A
{
    // body of the class A.
}
class B : public A
{
    // body of class B.
}
class C : public A
{
    // body of class C.
}
class D : public A
{
    // body of class D.
}
```

Let's see a simple example:

```
#include <iostream>
using namespace std;
class Shape                  // Declaration of base class.
{
    public:
    int a;
    int b;
    void get_data(int n,int m)
    {
        a= n;
        b = m;
    }
};
class Rectangle : public Shape  // inheriting Shape class
{
    public:
    int rect_area()
    {
        int result = a*b;
        return result;
    }
};
class Triangle : public Shape    // inheriting Shape class
{
    public:
    int triangle_area()
    {
        float result = 0.5*a*b;
        return result;
    }
};
int main()
{
    Rectangle r;
    Triangle t;
    int length,breadth,base,height;
    std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
    cin>>length>>breadth;
    r.get_data(length,breadth);

    int m = r.rect_area();

    std::cout << "Area of the rectangle is : " <<m<< std::endl;

    std::cout << "Enter the base and height of the triangle: " << std::endl;

    cin>>base>>height;

    t.get_data(base,height);

    float n = t.triangle_area();

    std::cout <<"Area of the triangle is : "  << n<<std::endl;

    return 0;

}
```

Output:

```
Enter the length and breadth of a rectangle:

23

20

Area of the rectangle is : 460

Enter the base and height of the triangle:

2

5

Area of the triangle is : 5
```

20.10 Aggregation:-

In C++, aggregation is a process in which one class defines another class as any entity reference. It is another way to reuse the class. It is a form of association that represents HAS-A relationship.

C++ Aggregation Example

Let's see an example of aggregation where Employee class has the reference of Address class as data member. In such way, it can reuse the members of Address class.

```
#include <iostream>

using namespace std;

class Address {
   public:
  string addressLine, city, state;
    Address(string addressLine, string city, string state)
   {
       this->addressLine = addressLine;
       this->city = city;
       this->state = state;
   }
};
class Employee
   {
       private:
       Address* address;  //Employee HAS-A Address
       public:
       int id;
       string name;
       Employee(int id, string name, Address* address)
      {
          this->id = id;
          this->name = name;
          this->address = address;
      }
    void display()
      {
          cout<<id <<" "<<name<< " "<<
            address->addressLine<< " "<< address->city<< " "<<address->state<<endl;
      }
  };
```

```
int main(void) {
    Address a1= Address("C-146, Sec-15","Noida","UP");
    Employee e1 = Employee(101,"Nakul",&a1);
            e1.display();
   return 0;
}
```

Output:

```
101 Nakul C-146, Sec-15 Noida UP
```

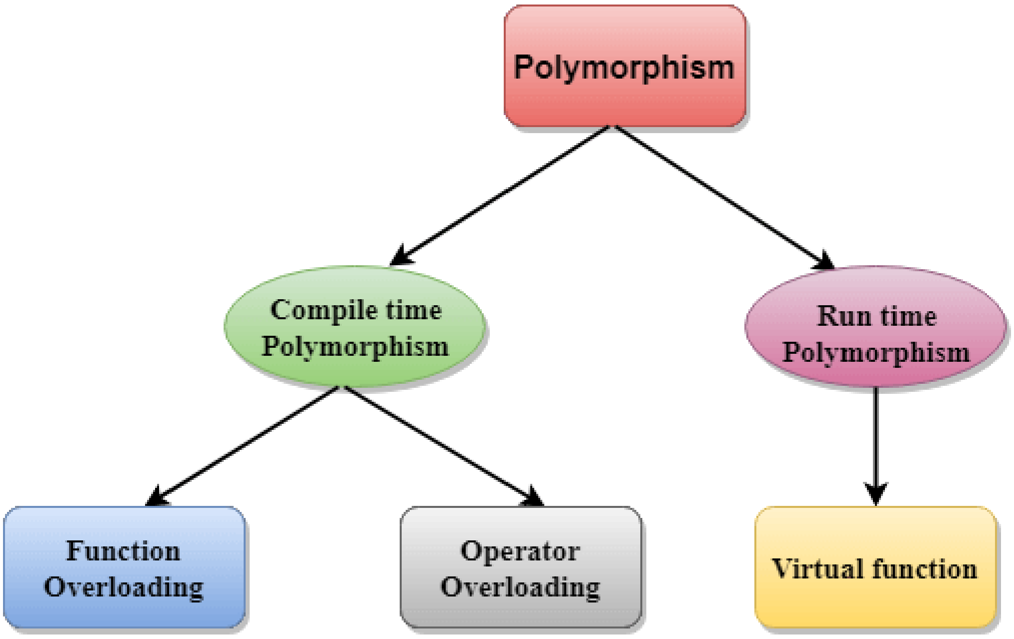
20.11Polymorphism:

The term "Polymorphism" is the combination of "poly" + "morphs" which means many forms. It is a greek word. In object-oriented programming, we use 3 main concepts: inheritance, encapsulation, and polymorphism.

Real Life Example Of Polymorphism

Let's consider a real-life example of polymorphism. A lady behaves like a teacher in a classroom, mother or daughter in a home and customer in a market. Here, a single person is behaving differently according to the situations.

There are two types of polymorphism in C++:



Compile time polymorphism: The overloaded functions are invoked by matching the type and number of arguments. This information is available at the compile time and, therefore, compiler selects the appropriate function at the compile time. It is achieved by function overloading and operator overloading which is also known as static binding or early binding. Now, let's consider the case where function name and prototype is same.

```
   class A                                   //  base class declaration.
  {
       int a;
       public:
       void display()
       {
             cout<< "Class A ";
        }
  };
class B : public A                           //  derived class declaration.
{
    int b;
    public:
   void display()
  {
        cout<<"Class B";
  }
};
```

In the above case, the prototype of display() function is the same in both the base and derived class. Therefore, the static binding cannot be applied. It would be great if the appropriate function is selected at the run time. This is known as run time polymorphism.

Run time polymorphism: Run time polymorphism is achieved when the object's method is invoked at the run time instead of compile time. It is achieved by method overriding which is also known as dynamic binding or late binding.

Compile time polymorphism	Run time polymorphism
The function to be invoked is known at the compile time.	The function to be invoked is known at the run time.
It is also known as overloading, early binding and static binding.	It is also known as overriding, Dynamic binding and late binding.
Overloading is a compile time polymorphism where more than one method is having the same name but with the different number of parameters or the type of the parameters.	Overriding is a run time polymorphism where more than one method is having the same name, number of parameters and the type of the parameters.
It is achieved by function overloading and operator overloading.	It is achieved by virtual functions and pointers.
It provides fast execution as it is known at the compile time.	It provides slow execution as it is known at the run time.
It is less flexible as mainly all the things execute at the compile time.	It is more flexible as all the things execute at the run time.

Example:Run Time Polymorphism

```
#include <iostream>
using namespace std;
class Animal {
    public:
void eat(){
cout<<"Eating...";
    }
};
class Dog: public Animal
{
 public:
 void eat()
    {           cout<<"Eating bread...";
    }
};
int main(void) {
   Dog d = Dog();
   d.eat();
   return 0;
}
```

Output:

```
Eating bread...
```

C++ Run time Polymorphism Example: By using two derived class

Let's see another example of run time polymorphism in C++ where we are having two derived classes.

```
#include <iostream>

using namespace std;

class Shape {                                    //  base class

    public:

virtual void draw(){                            // virtual function

cout<<"drawing..."<<endl;

    }

};

class Rectangle: public Shape                  //  inheriting Shape class.

{

 public:

 void draw()

   {

       cout<<"drawing rectangle..."<<endl;

    }

};

class Circle: public Shape                       //  inheriting Shape class.


{

 public:

 void draw()

   {

      cout<<"drawing circle..."<<endl;

   }
};

int main(void) {

    Shape *s;                         //  base class pointer.

    Shape sh;                         // base class object.

       Rectangle rec;

        Circle cir;

      s=&sh;

     s->draw();

        s=&rec;

     s->draw();

    s=?

    s->draw();

}
```

Output:

```
drawing...

drawing rectangle...

drawing circle...
```

Runtime Polymorphism with Data Members

Runtime Polymorphism can be achieved by data members in C++. Let's see an example where we are accessing the field by reference variable which refers to the instance of derived class.

```
#include <iostream>

using namespace std;

class Animal {                                //  base class declaration.

    public:

    string color = "Black";

};

class Dog: public Animal                      // inheriting Animal class.
```

```
{
 public:
    string color = "Grey";
};
int main(void) {
     Animal d= Dog();
    cout<<d.color;
}
```

Output:

```
Black
```

20.12 Overloading:

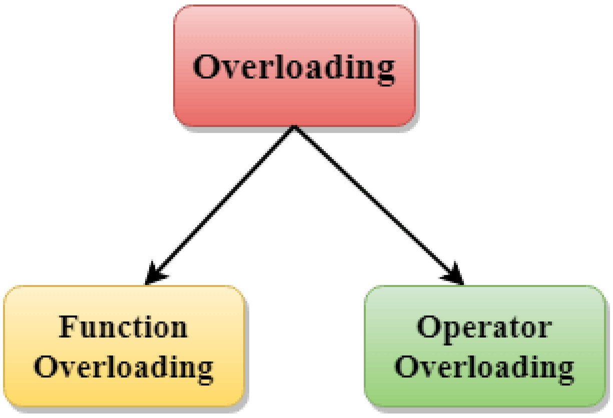
If we create two or more members having the same name but different in number or type of parameter, it is known as C++ overloading. In C++, we can overload:

1) methods,
2) constructors, and
3) indexed properties

It is because these members have parameters only.

Types of overloading in C++ are:

1) Function overloading
2) Operator overloading



C++ Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The advantage of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

C++ Function Overloading Example

Let's see the simple example of function overloading where we are changing number of arguments of add() method.

```
// program of function overloading when number of arguments vary.

#include <iostream>

using namespace std;

class Cal {

    public:

static int add(int a,int b){

        return a + b;

    }

static int add(int a, int b, int c)

    {

        return a + b + c;

    }

};

int main(void) {

    Cal C;                                                   //     class object declaration.

    cout<<C.add(10, 20)<<endl;

    cout<<C.add(12, 20, 23);

   return 0;

}
```

Output:

```
30
```

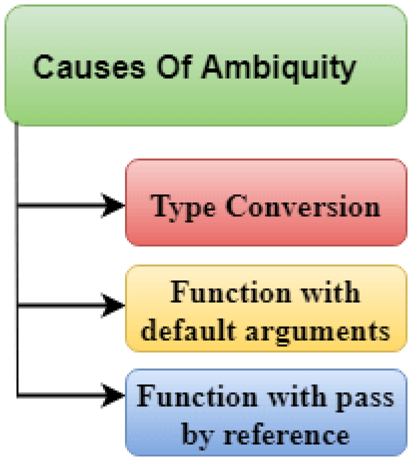
Let's see the simple example when the type of the arguments vary.

```
// Program of function overloading with different types of arguments.
#include<iostream>
using namespace std;
int mul(int,int);
float mul(float,int);
int mul(int a,int b)
{
   return a*b;
}
float mul(double x, int y)
{
   return x*y;
}
int main()
{
   int r1 = mul(6,7);
   float r2 = mul(0.2,3);
   std::cout << "r1 is : " <<r1<< std::endl;
   std::cout <<"r2 is : "  <<r2<< std::endl;
   return 0;
}
```

Output:

```
r1 is : 42
r2 is : 0.6
```

Ambuiguity: When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as Ambuigity.


Causes Of Ambiquity
Type Conversion
Function with default arguments
Function with pass by reference

Example:

```
#include<iostream>

using namespace std;

void fun(int);

void fun(float);

void fun(int i)

{

    std::cout << "Value of i is : " <<i<< std::endl;

}

void fun(float j)

{

    std::cout << "Value of j is : " <<j<< std::endl;

}

int main()

{

    fun(12);

    fun(1.2);

    return 0;

}
```

The above example shows an error "call of overloaded 'fun(double)' is ambiguous". The fun(10) will call the first function. The fun(1.2) calls the second function according to our prediction. But, this does not refer to any function as in C++, all the floating point constants are treated as double not as a float. If we replace float to double, the program works. Therefore, this is a type conversion from float to double.

Function with Default Arguments

```
#include<iostream>
using namespace std;
void fun(int);
void fun(int,int);
void fun(int i)
{
   std::cout << "Value of i is : " <<i<< std::endl;
}
void fun(int a,int b=9)
{
   std::cout << "Value of a is : " <<a<< std::endl;
   std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
   fun(12);

   return 0;
}
```

The above example shows an error "call of overloaded 'fun(int)' is ambiguous". The fun(int a, int b=9) can be called in two ways: first is by calling the function with one argument, i.e., fun(12) and another way is calling the function with two arguments, i.e., fun(4,5). The fun(int i) function is invoked with one argument. Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

Function with pass by reference

Let's see a simple example.

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int &);
int main()
{
int a=10;
fun(a); // error, which f()?
return 0;
}
void fun(int x)
{
std::cout << "Value of x is : " <<x<< std::endl;
}
void fun(int &b)
{
std::cout << "Value of b is : " <<b<< std::endl;
}
```

The above example shows an error "call of overloaded 'fun(int&)' is ambiguous". The first function takes one integer argument and the second function takes a reference parameter as an argument. In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

C++ Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

The advantage of Operators overloading is to perform different operations on the same operand.

Operator that cannot be overloaded are as follows:

Scope operator (::)

Sizeof

member selector(.)

member pointer selector(*)

ternary operator(?:)

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)

{

    // body of the function.

}
```

Where the return type is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

Existing operators can only be overloaded, but the new operators cannot be overloaded.

The overloaded operator contains atleast one operand of the user-defined data type.

We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.

When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.

When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

C++ Operators Overloading Example

Let's see the simple example of operator overloading in C++. In this example, void operator ++ () operator function is defined (inside Test class).

```
// program to overload the unary operator ++.
```

```
#include <iostream>
using namespace std;
class Test
{
  private:
    int num;
  public:
     Test(): num(8){}
     void operator ++()      {
       num = num+2;
     }
     void Print() {
        cout<<"The Count is: "<<num;
     }
};
int main()
{
   Test tt;
   ++tt;  // calling of a function "void operator ++()"
   tt.Print();
   return 0;
}
```

Output:

```
The Count is: 10
```

20.13 Function Overriding

If derived class defines same function as defined in its base class, it is known as function overriding in C++. It is used to achieve runtime polymorphism. It enables you to provide specific implementation of the function which is already provided by its base class.

C++ Function Overriding Example

Let's see a simple example of Function overriding in C++. In this example, we are overriding the eat() function.

```
#include <iostream>

using namespace std;

class Animal {

   public:

void eat(){

cout<<"Eating...";

   }

};

class Dog: public Animal

{

 public:

 void eat()

   {

      cout<<"Eating bread...";

   }

};

int main(void) {

  Dog d = Dog();

  d.eat();

  return 0;

}
```

Output:

```
Eating bread...
```

20.14 Abstraction:

Data abstraction refers to providing only essential information to the outside world and hiding their background details, i.e., to represent the needed information in program without presenting the details.

Data abstraction is a programming (and design) technique that relies on the separation of interface and implementation.

Let's take one real life example of a TV, which you can turn on and off, change the channel, adjust the volume, and add external components such as speakers, VCRs, and DVD players, BUT you do not know its internal details, that is, you do not know how it receives signals over the air or through a cable, how it translates them, and finally displays them on the screen.

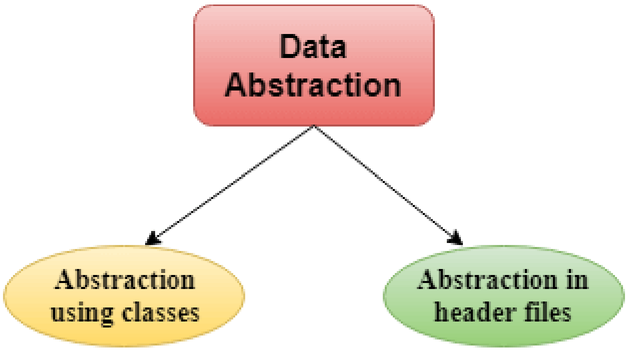
Thus, we can say a television clearly separates its internal implementation from its external interface and you can play with its interfaces like the power button, channel changer, and volume control without having any knowledge of its internals.

In C++, classes provides great level of **data abstraction**. They provide sufficient public methods to the outside world to play with the functionality of the object and to manipulate object data, i.e., state without actually knowing how class has been implemented internally.

For example, your program can make a call to the **sort()** function without knowing what algorithm the function actually uses to sort the given values. In fact, the underlying implementation of the sorting functionality could change between releases of the library, and as long as the interface stays the same, your function call will still work.

Data Abstraction can be achieved in two ways:

- Abstraction using classes
- Abstraction in header files.



Abstraction using classes: An abstraction can be achieved using classes. A class is used to group all the data members and member functions into a single unit by using the access specifiers. A class has the responsibility to determine which data member is to be visible outside and which is not.

Abstraction in header files: An another type of abstraction is header file. For example, pow() function available is used to calculate the power of a number without actually knowing which algorithm function uses to calculate the power. Thus, we can say that header files hides all the implementation details from the user.

Access Specifiers Implement Abstraction:

Public specifier: When the members are declared as public, members can be accessed anywhere from the program.

Private specifier: When the members are declared as private, members can only be accessed only by the member functions of the class.

Let's see a simple example of abstraction in header files.

```
// program to calculate the power of a number.

#include <iostream>

#include<math.h>

using namespace std;

int main()

{

 int n = 4;

   int power = 3;

   int result = pow(n,power);         // pow(n,power) is the  power function

   std::cout << "Cube of n is : " <<result<< std::endl;

   return 0;

}
```

Output:

```
Cube of n is : 64
```

In the above example, pow() function is used to calculate 4 raised to the power 3. The pow() function is present in the math.h header file in which all the implementation details of the pow() function is hidden.

Let's see a simple example of data abstraction using classes.

```
#include <iostream>

using namespace std;

 class Sum

{

private: int x, y, z; // private variables

public:

void add()

{

cout<<"Enter two numbers: ";

cin>>x>>y;

z= x+y;

cout<<"Sum of two number is: "<<z<<endl;

}

};

int main()

{

Sum sm;

sm.add();

return 0;

}
```

Output:

```
Enter two numbers:
```

```
3

6

Sum of two number is: 9
```

In the above example, abstraction is achieved using classes. A class 'Sum' contains the private members x, y and z are only accessible by the member functions of the class.

Advantages Of Abstraction:

Implementation details of the class are protected from the inadvertent user level errors.

A programmer does not need to write the low level code.

Data Abstraction avoids the code duplication, i.e., programmer does not have to undergo the same tasks every time to perform the similar operation.

The main aim of the data abstraction is to reuse the code and the proper partitioning of the code across the classes.

Internal implementation can be changed without affecting the user level code.

20.15 Encapsulation

Data encapsulation is a mechanism of bundling the data, and the functions that use them and data abstraction is a mechanism of exposing only the interfaces and hiding the implementation details from the user.

C++ supports the properties of encapsulation and data hiding through the creation of user-defined types, called classes. We already have studied that a class can contain private, protected and public members. By default, all items defined in a class are private. For example −

```
class Box {
   public:
      double getVolume(void) {
         return length * breadth * height;
      }

   private:
      double length;      // Length of a box
```

```
    double breadth;     // Breadth of a box
    double height;      // Height of a box
};
```

The variables length, breadth, and height are private. This means that they can be accessed only by other members of the Box class, and not by any other part of your program. This is one way encapsulation is achieved.

To make parts of a class public (i.e., accessible to other parts of your program), you must declare them after the public keyword. All variables or functions defined after the public specifier are accessible by all other functions in your program.

Making one class a friend of another exposes the implementation details and reduces encapsulation. The ideal is to keep as many of the details of each class hidden from all other classes as possible.

Data Encapsulation Example

Any C++ program where you implement a class with public and private members is an example of data encapsulation and data abstraction. Consider the following example −

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

class Adder {
  public:
    // constructor
    Adder(int i = 0) {
      total = i;
    }

    // interface to outside world
    void addNum(int number) {
      total += number;
    }
```

```
      // interface to outside world
      int getTotal() {
         return total;
      };

   private:
      // hidden data from outside world
      int total;
};

int main() {
   Adder a;
   a.addNum(10);
   a.addNum(20);
   a.addNum(30);

   cout << "Total " << a.getTotal() <<endl;
   return 0;
}
```

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

```
Total 60
```

Above class adds numbers together, and returns the sum. The public members addNum and getTotal are the interfaces to the outside world and a user needs to know them to use the class. The private member total is something that is hidden from the outside world, but is needed for the class to operate properly.

21 Namespace:

Namespaces in C++ are used to organize too many classes so that it can be easy to handle the application.

For accessing the class of a namespace, we need to use namespacename::classname. We can use using keyword so that we don't have to use complete name all the time.

In C++, global namespace is the root namespace. The global::std will always refer to the namespace "std" of C++ Framework.

C++ namespace Example

Let's see the simple example of namespace which include variable and functions.

```
#include <iostream>

using namespace std;

namespace First {

    void sayHello() {

        cout<<"Hello First Namespace"<<endl;

    }

}

namespace Second  {

       void sayHello() {

           cout<<"Hello Second Namespace"<<endl;

       }

}

int main()

{

 First::sayHello();

 Second::sayHello();

return 0;
}
```

Output:

```
Hello First Namespace

Hello Second Namespace
```

C++ namespace example: by using keyword

Let's see another example of namespace where we are using "using" keyword so that we don't have to use complete name for accessing a namespace program.

```
#include <iostream>

using namespace std;

namespace First{

  void sayHello(){

    cout << "Hello First Namespace" << endl;

  }

}

namespace Second{

  void sayHello(){

    cout << "Hello Second Namespace" << endl;

  }

}

using namespace First;

int main () {

  sayHello();

  return 0;

}
```

Output:

```
Hello First Namespace
```

22.STL(Standard Template Library)

22.1 Introduction to STL

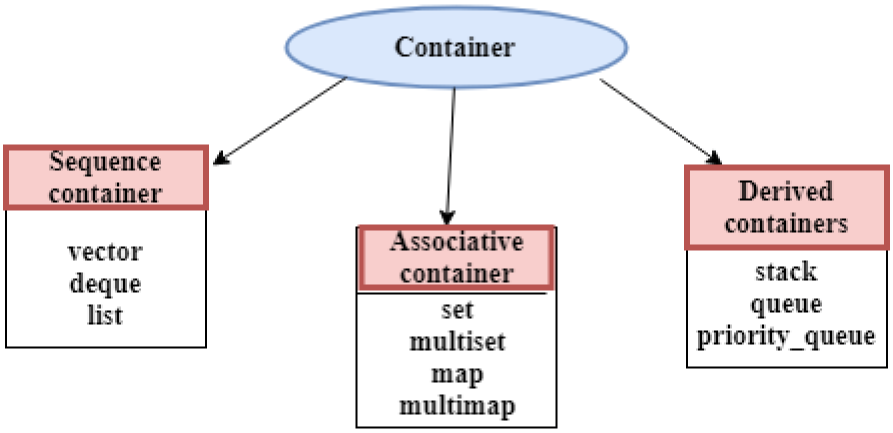
The C++ STL (Standard Template Library) is a powerful set of C++ template classes to provide general-purpose classes and functions with templates that implement many popular and commonly used algorithms and data structures like vectors, lists, queues, and stacks.

CONTAINERS

Containers can be described as the objects that hold the data of the same type. Containers are used to implement different data structures for example arrays, list, trees, etc.

Following are the containers that give the details of all the containers as well as the header file and the type of iterator associated with them :

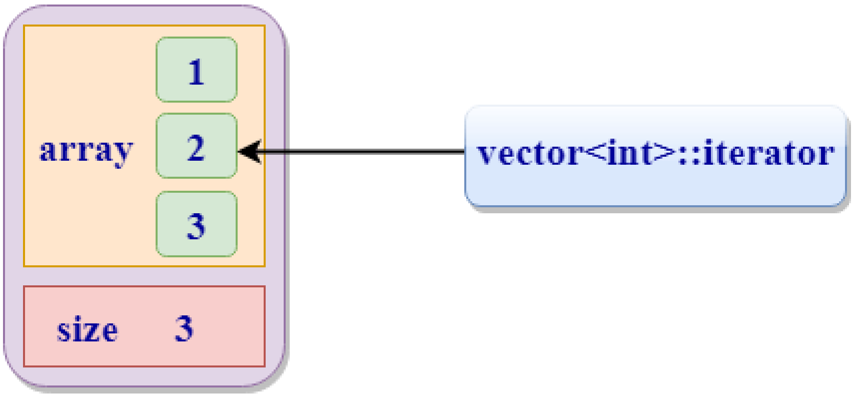
Container	Description	Header file	iterator
vector	vector is a class that creates a dynamic array allowing insertions and deletions at the back.	<vector>	Random access
list	list is the sequence containers that allow the insertions and deletions from anywhere.	<list>	Bidirectional
deque	deque is the double ended queue that allows the insertion and deletion from both the ends.	<deque>	Random access
set	set is an associate container for storing unique sets.	<set>	Bidirectional
multiset	Multiset is an associate container for storing non- unique sets.	<set>	Bidirectional
map	Map is an associate container for storing unique key-value pairs, i.e. each key is associated with only one value(one to one mapping).	<map>	Bidirectional
multimap	multimap is an associate container for storing key- value pair, and each key can be associated with more than one value.	<map>	Bidirectional
stack	It follows last in first out(LIFO).	<stack>	No iterator
queue	It follows first in first out(FIFO).	<queue>	No iterator
Priority-queue	First element out is always the highest priority element.	<queue>	No iterator



ITERATOR

Iterators are pointer-like entities used to access the individual elements in a container.

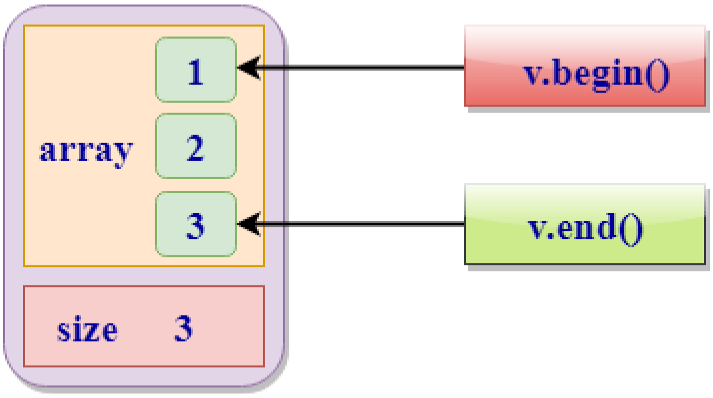
Iterators are moved sequentially from one element to another element. This process is known as iterating through a container.

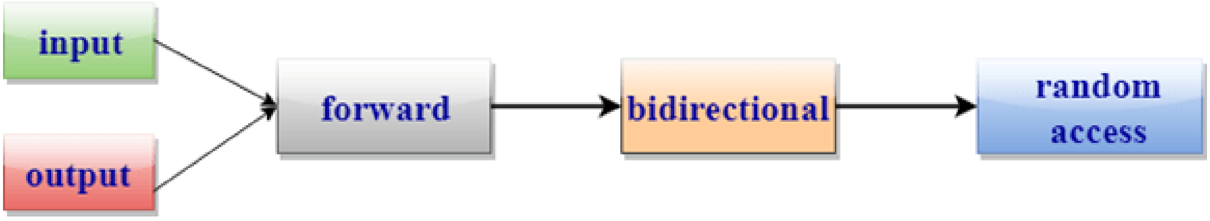


Iterator contains mainly two functions:

begin(): The member function begin() returns an iterator to the first element of the vector.

end(): The member function end() returns an iterator to the past-the-last element of a container.





1) Input iterator:

An Input iterator is an iterator that allows the program to read the values from the container.

Dereferencing the input iterator allows us to read a value from the container, but it does not alter the value.

An Input iterator is a one way iterator.

An Input iterator can be incremented, but it cannot be decremented.

2) Output iterator:

An output iterator is similar to the input iterator, except that it allows the program to modify a value of the container, but it does not allow to read it.

It is a one-way iterator.

It is a write only iterator.

3) Forward iterator:

Forward iterator uses the ++ operator to navigate through the container.

Forward iterator goes through each element of a container and one element at a time.

4) Bidirectional iterator:

A Bidirectional iterator is similar to the forward iterator, except that it also moves in the backward direction.

It is a two way iterator.

It can be incremented as well as decremented.

5) Random Access Iterator:

Random access iterator can be used to access the random element of a container.

Random access iterator has all the features of a bidirectional iterator, and it also has one more additional feature, i.e., pointer addition. By using the pointer addition operation, we can access the random element of a container.

22.2 C++ Vector

A vector is a sequence container class that implements dynamic array, means size automatically changes when appending elements. A vector stores the elements in contiguous memory locations and allocates the memory as needed at run time.

Difference between vector and array

An array follows static approach, means its size cannot be changed during run time while vector implements dynamic array means it automatically resizes itself when appending elements.

Syntax

Consider a vector 'v1'. Syntax would be:

```
vector<object_type> v1;
```

Example

Let's see a simple example.

```
#include<iostream>

#include<vector>

using namespace std;

int main()

{

vector<string> v1;

v1.push_back("javaTpoint ");

v1.push_back("tutorial");

for(vector<string>::iterator itr=v1.begin();itr!=v1.end();++itr)

cout<<*itr;

return 0;

}
```

Output:

```
javaTpoint tutorial
```

C++ Vector Functions

Function	Description
at()	It provides a reference to an element.
back()	It gives a reference to the last element.
front()	It gives a reference to the first element.
swap()	It exchanges the elements between two vectors.
push_back()	It adds a new element at the end.
pop_back()	It removes a last element from the vector.
empty()	It determines whether the vector is empty or not.
insert()	It inserts new element at the specified position.
erase()	It deletes the specified element.
resize()	It modifies the size of the vector.
clear()	It removes all the elements from the vector.
size()	It determines a number of elements in the vector.
capacity()	It determines the current capacity of the vector.
assign()	It assigns new values to the vector.
operator=()	It assigns new values to the vector container.
operator[]()	It access a specified element.
end()	It refers to the past-lats-element in the vector.
emplace()	It inserts a new element just before the position pos.
emplace_back()	It inserts a new element at the end.
rend()	It points the element preceding the first element of the vector.
rbegin()	It points the last element of the vector.
begin()	It points the first element of the vector.
max_size()	It determines the maximum size that vector can hold.
cend()	It refers to the past-last-element in the vector.
cbegin()	It refers to the first element of the vector.
crbegin()	It refers to the last character of the vector.
crend()	It refers to the element preceding the first element of the vector.
data()	It writes the data of the vector into an array.
shrink_to_fit()	It reduces the capacity and makes it equal to the size of the vector.

22.3 C++ List

List is a contiguous container while vector is a non-contiguous container i.e list stores the elements on a contiguous memory and vector stores on a non-contiguous memory.

Insertion and deletion in the middle of the vector is very costly as it takes lot of time in shifting all the elements. Linklist overcome this problem and it is implemented using list container.

List supports a bidirectional and provides an efficient way for insertion and deletion operations.

Traversal is slow in list as list elements are accessed sequentially while vector supports a random access.

Template for list

```
#include<iostream>

#include<list>

using namespace std;

int main()

{

  list<int> l;

}
```

It creates an empty list of integer type values.

List can also be initalised with the parameters.

```
#include<iostream>

#include<list>

using namespace std;

int main()

{

  list<int> l{1,2,3,4};

}
```

List can be initialised in two ways.

list<int> new_list{1,2,3,4};

or

list<int> new_list = {1,2,3,4};

Following are the member functions of the list:

Method	Description
insert()	It inserts the new element before the position pointed by the iterator.
push_back()	It adds a new element at the end of the vector.
push_front()	It adds a new element to the front.
pop_back()	It deletes the last element.
pop_front()	It deletes the first element.
empty()	It checks whether the list is empty or not.
size()	It finds the number of elements present in the list.
max_size()	It finds the maximum size of the list.
front()	It returns the first element of the list.
back()	It returns the last element of the list.
swap()	It swaps two list when the type of both the list are same.
reverse()	It reverses the elements of the list.
sort()	It sorts the elements of the list in an increasing order.
merge()	It merges the two sorted list.
splice()	It inserts a new list into the invoking list.
unique()	It removes all the duplicate elements from the list.
resize()	It changes the size of the list container.
assign()	It assigns a new element to the list container.
emplace()	It inserts a new element at a specified position.

emplace_back()	It inserts a new element at the end of the vector.
emplace_front()	It inserts a new element at the beginning of the list.

22.4 C++ stack

In computer science we go for working on a large variety of programs. Each of them has their own domain and utility. Based on the purpose and environment of the program creation, we have a large number of data structures available to choose from. One of them is 'stack'. Before discussing about this data type let us take a look at its syntax.

Syntax

template<class T, class Container = deque<T> > class stack;

This data structure works on the LIFO technique, where LIFO stands for Last In First Out. The element which was first inserted will be extracted at the end and so on. There is an element called as 'top' which is the element at the upper most position. All the insertion and deletion operations are made at the top element itself in the stack.

Stacks in the application areas are implied as the container adaptors.

The containers should have a support for the following list of operations:

1) empty
2) size
3) back
4) push_back
5) pop_back

```
#include <iostream>

#include <stack>

using namespace std;

void newstack(stack <int> ss)

{

   stack <int> sg = ss;

   while (!sg.empty())

   {

      cout << '\t' << sg.top();

      sg.pop();

   }
```

```
    cout << '\n';
}
int main ()
{
    stack <int> newst;
    newst.push(55);
    newst.push(44);
    newst.push(33);
    newst.push(22);
    newst.push(11);
    cout << "The stack newst is : ";
    newstack(newst);
    cout << "\n newst.size() : " << newst.size();
    cout << "\n newst.top() : " << newst.top();
    cout << "\n newst.pop() : ";
    newst.pop();
    newstack(newst);
    return 0;
}
```

22.5 C++ queue

In computer science we go for working on a large variety of programs. Each of them has their own domain and utility. Based on the purpose and environment of the program creation, we have a large number of data structures available to choose from. One of them is 'queues. Before discussing about this data type let us take a look at its syntax.

Syntax

```
template<class T, class Container = deque<T> > class queue;
```

This data structure works on the FIFO technique, where FIFO stands for First In First Out. The element which was first inserted will be extracted at the first and so on. There is an element called as 'front' which is the element at the front most position or say the first position, also there is an element called as 'rear' which is the element at the last position. In normal queues insertion of elements take at the rear end and the deletion is done from the front.

Queues in the application areas are implied as the container adaptors.

The containers should have a support for the following list of operations:

1) empty
2) size
3) push_back
4) pop_front
5) front
6) back

Example

```
#include <iostream>
#include <queue>
using namespace std;
void showsg(queue <int> sg)
{
    queue <int> ss = sg;
    while (!ss.empty())
    {
        cout << '\t' << ss.front();
        ss.pop();
    }
    cout << '\n';
}
int main()
{
    queue <int> fquiz;
    fquiz.push(10);
    fquiz.push(20);
    fquiz.push(30);
    cout << "The queue fquiz is : ";
    showsg(fquiz);
    cout << "\nfquiz.size() : " << fquiz.size();
    cout << "\nfquiz.front() : " << fquiz.front();
    cout << "\nfquiz.back() : " << fquiz.back();
    cout << "\nfquiz.pop() : ";
    fquiz.pop();
    showsg(fquiz);
    return 0;
}
```

Function	Description
(constructor)	The function is used for the construction of a queue container.
empty	The function is used to test for the emptiness of a queue. If the queue is empty the function returns true else false.
size	The function returns the size of the queue container, which is a measure of the number of elements stored in the queue.
front	The function is used to access the front element of the queue. The element plays a very important role as all the deletion operations a re performed at the front element.
back	The function is used to access the rear element of the queue. The element plays a very important role as all the insertion operations are performed at the rear element.
push	The function is used for the insertion of a new element at the rear end of the queue.
pop	The function is used for the deletion of element; the element in the queue is deleted from the front end.
emplace	The function is used for insertion of new elements in the queue above the current rear element.
swap	The function is used for interchanging the contents of two containers in reference.

relational operators	The non member function specifies the relational operators that are needed for the queues.
uses allocator<queue>	As the name suggests the non member function uses the allocator for the queues.

22.6 Priority Queue in C++

The priority queue in C++ is a derived container in STL that considers only the highest priority element. The queue follows the FIFO policy while priority queue pops the elements based on the priority, i.e., the highest priority element is popped first.

It is similar to the ordinary queue in certain aspects but differs in the following ways:

In a priority queue, every element in the queue is associated with some priority, but priority does not exist in a queue data structure.

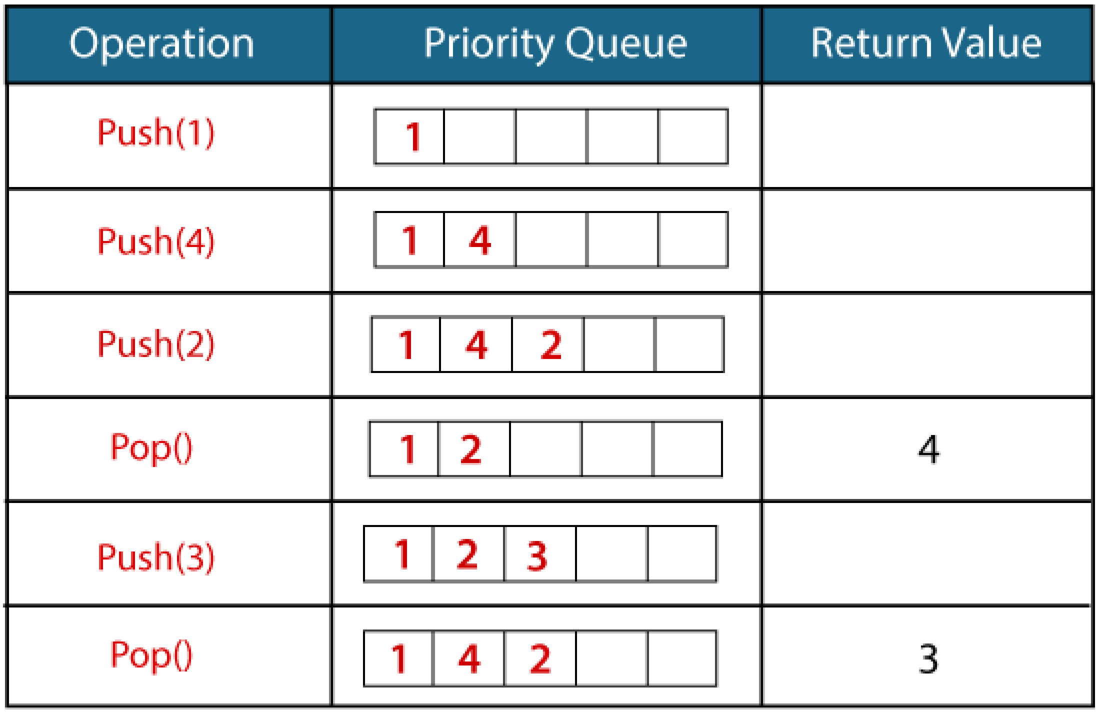
The element with the highest priority in a priority queue will be removed first while queue follows the FIFO(First-In-First-Out) policy means the element which is inserted first will be deleted first.

If more than one element exists with the same priority, then the order of the element in a queue will be taken into consideration.

Note: The priority queue is the extended version of a normal queue except that the element with the highest priority will be removed first from the priority queue.

Syntax of Priority Queue

priority_queue<int> variable_name;



Operation	Priority Queue	Return Value
Push(1)	1	
Push(4)	1 4	
Push(2)	1 4 2	
Pop()	1 2	4
Push(3)	1 2 3	
Pop()	1 4 2	3

Member Function of Priority Queue

Function	Description
push()	It inserts a new element in a priority queue.
pop()	It removes the top element from the queue, which has the highest priority.
top()	This function is used to address the topmost element of a priority queue.
size()	It determines the size of a priority queue.
empty()	It verifies whether the queue is empty or not. Based on the verification, it returns the status.
swap()	It swaps the elements of a priority queue with another queue having the same type and size.
emplace()	It inserts a new element at the top of the priority queue.

```
Example: #include <iostream>

#include<queue>

using namespace std;

int main()

{

 priority_queue<int> p;  // variable declaration.

 p.push(10); // inserting 10 in a queue, top=10

 p.push(30); // inserting 30 in a queue, top=30

 p.push(20); // inserting 20 in a queue, top=20

 cout<<"Number of elements available in 'p' :"<<p.size()<<endl;

 while(!p.empty())

 {

     std::cout << p.top() << std::endl;

     p.pop();

 }

 return 0;

}
```

22.7 C++ map function

Maps are part of the C++ STL (Standard Template Library). Maps are the associative containers that store sorted key-value pair, in which each key is unique and it can be inserted or deleted but cannot be altered. Values associated with keys can be changed.

For example: A map of Employees where employee ID is the key and name is the value can be represented as:

Keys	Values
101	Nikita
102	Robin
103	Deep
104	John

Example:

```
#include <string.h>
#include <iostream>
#include <map>
#include <utility>
using namespace std;
int main()
{
   map<int, string> Employees;
   // 1) Assignment using array index notation
   Employees[101] = "Nikita";
   Employees[105] = "John";
   Employees[103] = "Dolly";
   Employees[104] = "Deep";
   Employees[102] = "Aman";
   cout << "Employees[104]=" << Employees[104] << endl << endl;
   cout << "Map size: " << Employees.size() << endl;
   cout << endl << "Natural Order:" << endl;
   for( map<int,string>::iterator ii=Employees.begin(); ii!=Employees.end(); ++ii)
   {
       cout << (*ii).first << ": " << (*ii).second << endl;
   }
   cout << endl << "Reverse Order:" << endl;
   for( map<int,string>::reverse_iterator ii=Employees.rbegin(); ii!=Employees.rend(); ++ii)
   {
       cout << (*ii).first << ": " << (*ii).second << endl;
   }  }
```

23.Programs

C++ "Hello, World!" Program

C++ Program to Print Number Entered by User

C++ Program to Add Two Numbers

C++ Program to Find Quotient and Remainder

C++ Program to Find Size of int, float, double and char in Your System

C++ Program to Swap Two Numbers

C++ Program to Check Whether Number is Even or Odd

C++ Program to Check Whether a character is Vowel or Consonant.

C++ Program to Find Largest Number Among Three Numbers

C++ Program to Find All Roots of a Quadratic Equation

C++ Program to Calculate Sum of Natural Numbers

C++ Program to Check Leap Year

C++ Program to Find Factorial

C++ Program to Generate Multiplication Table

C++ Program to Display Fibonacci Series

C++ Program to Find GCD

C++ Program to Find LCM

C++ Program to Reverse a Number

C++ Program to Calculate Power of a Number

Increment ++ and Decrement -- Operator Overloading in C++ Programming

C++ Program to Subtract Complex Number Using Operator Overloading

C++ Program to Find ASCII Value of a Character

C++ Program to Multiply two Numbers

C++ Program to Check Whether a Number is Palindrome or Not

C++ Program to Check Whether a Number is Prime or Not

C++ Program to Display Prime Numbers Between Two Intervals

C++ Program to Check Armstrong Number

C++ Program to Display Armstrong Number Between Two Intervals

C++ Program to Display Factors of a Number

C++ Programs To Create Pyramid and Pattern

C++ Program to Make a Simple Calculator to Add, Subtract, Multiply or Divide Using switch...case

C++ Program to Display Prime Numbers Between Two Intervals Using Functions

C++ Program to Check Prime Number By Creating a Function

C++ Program to Check Whether a Number can be Express as Sum of Two Prime Numbers

C++ program to Find Sum of Natural Numbers using Recursion

C++ program to Calculate Factorial of a Number Using Recursion

C++ Program to Find G.C.D Using Recursion

C++ Program to Convert Binary Number to Decimal and vice-versa

C++ Program to Convert Octal Number to Decimal and vice-versa

C++ Program to Convert Binary Number to Octal and vice-versa

C++ program to Reverse a Sentence Using Recursion

C++ Program to Calculate Power Using Recursion

C++ Program to Calculate Average of Numbers Using Arrays

C++ Program to Find Largest Element of an Array

C++ Program to Calculate Standard Deviation

C++ Program to Add Two Matrix Using Multi-dimensional Arrays

C++ Program to Multiply Two Matrix Using Multi-dimensional Arrays

C++ Program to Find Transpose of a Matrix

C++ Program to Multiply two Matrices by Passing Matrix to Function

C++ Program to Access Elements of an Array Using Pointer

C++ Program to Swap Numbers in Cyclic Order Using Call by Reference

C++ Program to Find the Frequency of Characters in a String

C++ Program to Find the Number of Vowels, Consonants, Digits and White Spaces in a String

C++ Program to Remove all Characters in a String Except Alphabets.

C++ Program to Find the Length of a String

C++ Program to Concatenate Two Strings

C++ Program to Copy Strings

C++ Program to Sort Elements in Lexicographical Order (Dictionary Order)

C++ Program to Store Information of a Student in a Structure

C++ Program to Add Two Distances (in inch-feet) System Using Structures

C++ Program to Add Complex Numbers by Passing Structure to a Function

C++ Program to Calculate Difference Between Two Time Period

C++ Program to Store and Display Information Using Structure

C++ Program to Print Pyramids and Patterns

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

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5

A
B B
C C C
D D D D
E E E E E

* * * * *
* * * *
* * *
* *
*
1 2 3 4 5
1 2 3 4
1 2 3
1 2
1
```

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

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

24.Question and Answers:

A list of top frequently asked C++ interview questions and answers are given below.

1) What is C++?

C++ is an object-oriented programming language created by Bjarne Stroustrup. It was released in 1985.

C++ is a superset of C with the major addition of classes in C language.

Initially, Stroustrup called the new language "C with classes". However, after sometime the name was changed to C++. The idea of C++ comes from the C increment operator ++.

2) What are the advantages of C++?

C++ doesn't only maintains all aspects from C language, it also simplifies memory management and adds several features like:

- C++ is a highly portable language means that the software developed using C++ language can run on any platform.
- C++ is an object-oriented programming language which includes the concepts such as classes, objects, inheritance, polymorphism, abstraction.
- C++ has the concept of inheritance. Through inheritance, one can eliminate the redundant code and can reuse the existing classes.
- Data hiding helps the programmer to build secure programs so that the program cannot be attacked by the invaders.
- Message passing is a technique used for communication between the objects.
- C++ contains a rich function library.

3) What is the difference between C and C++?

Following are the differences between C and C++:

C	C++
C language was developed by Dennis Ritchie.	C++ language was developed by Bjarne Stroustrup.
C is a structured programming language.	C++ supports both structural and object-oriented programming language.
C is a subset of C++.	C++ is a superset of C.
In C language, data and functions are the free entities.	In the C++ language, both data and functions are encapsulated together in the form of a
C does not support the data hiding. Therefore, the data can be used by the outside world.	C++ supports data hiding. Therefore, the data cannot be accessed by the outside world.
C supports neither function nor operator overloading.	C++ supports both function and operator overloading.
In C, the function cannot be implemented inside the structures.	In the C++, the function can be implemented inside the structures.
Reference variables are not supported in C language.	C++ supports the reference variables.
C language does not support the virtual and friend functions.	C++ supports both virtual and friend functions.
In C, scanf() and printf() are mainly used for input/output.	C++ mainly uses stream cin and cout to perform input and output operations.

4) What is the difference between reference and pointer?

Following are the differences between reference and pointer:

Reference	Pointer
Reference behaves like an alias for an existing variable, i.e., it is a temporary variable.	The pointer is a variable which stor of a variable.
Reference variable does not require any indirection operator to access the value. A reference variable can be used directly to access the value.	Pointer variable requires an indirec access the value of a variable.
Once the reference variable is assigned, then it cannot be reassigned with different address values.	The pointer variable is an independ means that it can be reassigned to different objects.
A null value cannot be assigned to the reference variable.	A null value can be assigned to the variable.
It is necessary to initialize the variable at the time of declaration.	It is not necessary to initialize the va time of declaration.

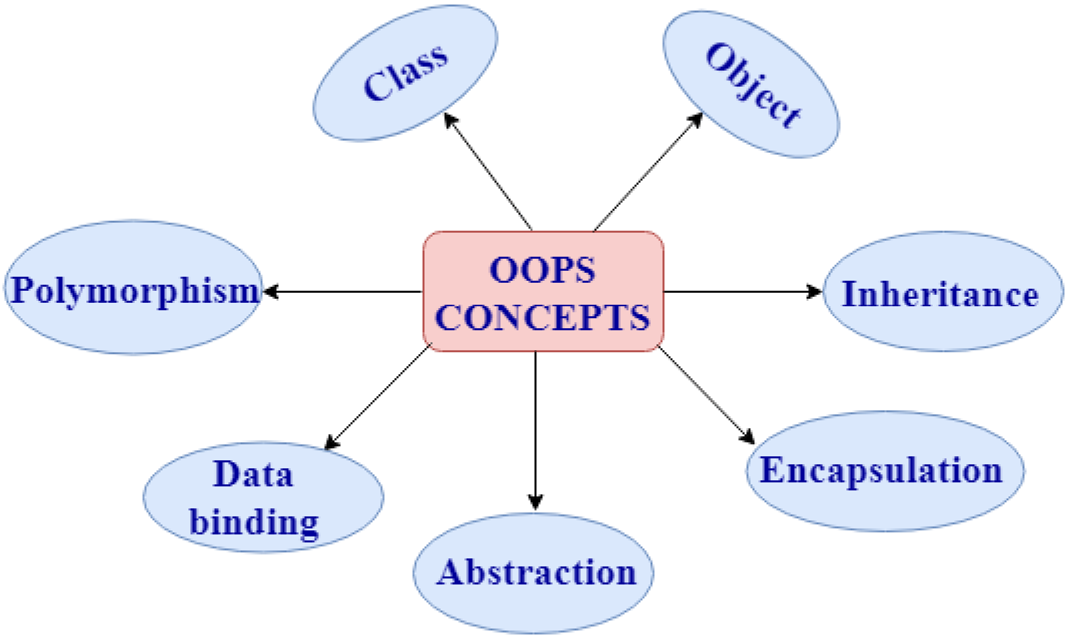
5) What is a class?

The class is a user-defined data type. The class is declared with the keyword class. The class contains the data members, and member functions whose access is defined by the three modifiers are private, public and protected. The class defines the type definition of the category of things. It defines a datatype, but it does not define the data it just specifies the structure of data.

You can create N number of objects from a class.

6) What are the various OOPs concepts in C++?

The various OOPS concepts in C++ are:



- Class:

The class is a user-defined data type which defines its properties and its functions. For example, Human being is a class. The body parts of a human being are its properties, and the actions performed by the body parts are known as functions. The class does not occupy any memory space. Therefore, we can say that the class is the only logical representation of the data.

The syntax of declaring the class:

```
    class student
{
//data members;
//Member functions
}
```

- Object:

An object is a run-time entity. An object is the instance of the class. An object can represent a person, place or any other item. An object can operate on both data members and member functions. The class does not occupy any memory space. When an object is created using a new keyword, then space is allocated for the variable in a heap, and the starting address is stored in the stack memory. When an object is created without a new keyword, then space is not allocated in the heap memory, and the object contains the null value in the stack.

```
class Student
{
//data members;
//Member functions
}
```

The syntax for declaring the object:

Student s = new Student();

- Inheritance:

Inheritance provides reusability. Reusability means that one can use the functionalities of the existing class. It eliminates the redundancy of code. Inheritance is a technique of deriving a new class from the old class. The old class is known as the base class, and the new class is known as derived class.

Syntax

class derived_class :: visibility-mode base_class;

- Encapsulation:

Encapsulation is a technique of wrapping the data members and member functions in a single unit. It binds the data within a class, and no outside method can access the data. If the data member is private, then the member function can only access the data.

- Abstraction:

Abstraction is a technique of showing only essential details without representing the implementation details. If the members are defined with a public keyword, then the members are accessible outside also. If the members are defined with a private keyword, then the members are not accessible by the outside methods.

- Data binding:

Data binding is a process of binding the application UI and business logic. Any change made in the business logic will reflect directly to the application UI.

- Polymorphism:

Polymorphism means multiple forms. Polymorphism means having more than one function with the same name but with different functionalities. Polymorphism is of two types:

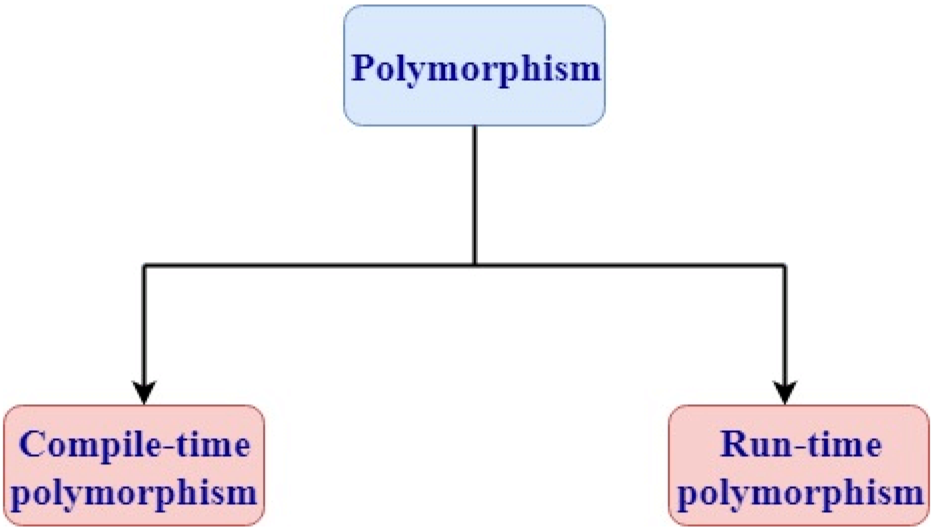
1. Static polymorphism is also known as early binding.
2. Dynamic polymorphism is also known as late binding.

---

7) What are the different types of polymorphism in C++?

Polymorphism: Polymorphism means multiple forms. It means having more than one function with the same function name but with different functionalities.

Polymorphism is of two types:



- Runtime polymorphism

Runtime polymorphism is also known as dynamic polymorphism. Function overriding is an example of runtime polymorphism. Function overriding means when the child class contains the method which is already present in the parent class. Hence, the child class overrides the method of the parent class. In case of function overriding, parent and child class both contains the same function with the different definition. The call to the function is determined at runtime is known as runtime polymorphism.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
class Base
{
   public:
   virtual void show()
   {
      cout<<"javaTpoint";
    }
     };
class Derived:public Base
{
   public:
   void show()
   {
      cout<<"javaTpoint tutorial";
   }
};

     int main()
{
   Base* b;
   Derived d;
```

```
    b=&d;
    b->show();
            return 0;
}
```

Output:

```
javaTpoint tutorial
```

- Compile time polymorphism

Compile-time polymorphism is also known as static polymorphism. The polymorphism which is implemented at the compile time is known as compile-time polymorphism. Method overloading is an example of compile-time polymorphism.

Method overloading: Method overloading is a technique which allows you to have more than one function with the same function name but with different functionality.

Method overloading can be possible on the following basis:

- The return type of the overloaded function.
- The type of the parameters passed to the function.
- The number of parameters passed to the function.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
class Multiply
{
   public:
   int mul(int a,int b)
   {
       return(a*b);
   }
   int mul(int a,int b,int c)
   {
       return(a*b*c);
  }
 };
int main()
{
    Multiply multi;
    int res1,res2;
    res1=multi.mul(2,3);
    res2=multi.mul(2,3,4);
    cout<<"\n";
    cout<<res1;
    cout<<"\n";
    cout<<res2;
    return 0;
}
```

Output:

```
6
24
```

- In the above example, mul() is an overloaded function with the different number of parameters.

---

8) Define namespace in C++.

- The namespace is a logical division of the code which is designed to stop the naming conflict.
- The namespace defines the scope where the identifiers such as variables, class, functions are declared.
- The main purpose of using namespace in C++ is to remove the ambiguity. Ambiquity occurs when the different task occurs with the same name.
- For example: if there are two functions exist with the same name such as add(). In order to prevent this ambiguity, the namespace is used. Functions are declared in different namespaces.
- C++ consists of a standard namespace, i.e., std which contains inbuilt classes and functions. So, by using the statement "using namespace std;" includes the namespace "std" in our program.
- Syntax of namespace:

```
namespace namespace_name
{
 //body of namespace;
}
```

Syntax of accessing the namespace variable:

```
namespace_name::member_name;
```

Let's understand this through an example:

```
#include <iostream>
using namespace std;
namespace addition
{
    int a=5;
    int b=5;
    int add()
    {
        return(a+b);
    }
      }
int main() {
int result;
result=addition::add();
cout<<result;
return 0;
}
```

Output:

```
10
```

---

9) Define token in C++.

A token in C++ can be a keyword, identifier, literal, constant and symbol.

---

10) Who was the creator of C++?

Bjarne Stroustrup.

---

11) Which operations are permitted on pointers?

Following are the operations that can be performed on pointers:

- Incrementing or decrementing a pointer: Incrementing a pointer means that we can increment the pointer by the size of a data type to which it points.

There are two types of increment pointers:

1. Pre-increment pointer: The pre-increment operator increments the operand by 1, and the value of the expression becomes the resulting value of the incremented. Suppose ptr is a pointer then pre-increment pointer is represented as ++ptr.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
int main()
{
int a[5]={1,2,3,4,5};
int *ptr;
ptr=&a[0];
cout<<"Value of *ptr is : "<<*ptr<<"\n";
cout<<"Value of *++ptr : "<<*++ptr;
return 0;
}
```

Output:

```
Value of *ptr is : 1
Value of *++ptr : 2
```

2. Post-increment pointer: The post-increment operator increments the operand by 1, but the value of the expression will be the value of the operand prior to the incremented value of the operand. Suppose ptr is a pointer then post-increment pointer is represented as ptr++.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
int main()
{
int a[5]={1,2,3,4,5};
int *ptr;
ptr=&a[0];
cout<<"Value of *ptr is : "<<*ptr<<"\n";
cout<<"Value of *ptr++ : "<<*ptr++;
return 0;
}
```

Output:

```
Value of *ptr is : 1
Value of *ptr++ : 1
```

- Subtracting a pointer from another pointer: When two pointers pointing to the members of an array are subtracted, then the number of elements present between the two members are returned.

---

12) Define 'std'.

Std is the default namespace standard used in C++.

---

13) Which programming language's unsatisfactory performance led to the discovery of C++?

C++was discovered in order to cope with the disadvantages of C.

---

14) How delete [] is different from delete?

Delete is used to release a unit of memory, delete[] is used to release an array.

---

15) What is the full form of STL in C++?

STL stands for Standard Template Library.

---

16) What is an object?

The Object is the instance of a class. A class provides a blueprint for objects. So you can create an object from a class. The objects of a class are declared with the same sort of declaration that we declare variables of basic types.

---

17) What are the C++ access specifiers?

The access specifiers are used to define how to functions and variables can be accessed outside the class.

There are three types of access specifiers:

- Private: Functions and variables declared as private can be accessed only within the same class, and they cannot be accessed outside the class they are declared.
- Public: Functions and variables declared under public can be accessed from anywhere.
- Protected: Functions and variables declared as protected cannot be accessed outside the class except a child class. This specifier is generally used in inheritance.

---

18) What is Object Oriented Programming (OOP)?

OOP is a methodology or paradigm that provides many concepts. The basic concepts of Object Oriented Programming are given below:

Classes and Objects: Classes are used to specify the structure of the data. They define the data type. You can create any number of objects from a class. Objects are the instances of classes.

Encapsulation: Encapsulation is a mechanism which binds the data and associated operations together and thus hides the data from the outside world. Encapsulation is also known as data hiding. In C++, It is achieved using the access specifiers, i.e., public, private and protected.

Abstraction: Abstraction is used to hide the internal implementations and show only the necessary details to the outer world. Data abstraction is implemented using interfaces and abstract classes in C++.

Some people confused about Encapsulation and abstraction, but they both are different.

Inheritance: Inheritance is used to inherit the property of one class into another class. It facilitates you to define one class in term of another class.

---

19) What is the difference between an array and a list?

- An Array is a collection of homogeneous elements while a list is a collection of heterogeneous elements.
- Array memory allocation is static and continuous while List memory allocation is dynamic and random.
- In Array, users don't need to keep in track of next memory allocation while In the list, the user has to keep in track of next location where memory is allocated.

---

20) What is the difference between new() and malloc()?

- new() is a preprocessor while malloc() is a function.
- There is no need to allocate the memory while using "new" but in malloc() you have to use sizeof().
- "new" initializes the new memory to 0 while malloc() gives random value in the newly allotted memory location.
- The new() operator allocates the memory and calls the constructor for the object initialization and malloc() function allocates the memory but does not call the constructor for the object initialization.
- The new() operator is faster than the malloc() function as operator is faster than the function.

---

21) What are the methods of exporting a function from a DLL?

There are two ways:

- By using the DLL's type library.
- Taking a reference to the function from the DLL instance.

---

22) Define friend function.

Friend function acts as a friend of the class. It can access the private and protected members of the class. The friend function is not a member of the class, but it must be listed in the class definition. The non-member function cannot access the private data of the class. Sometimes, it is necessary for the non-member function to access the data. The friend function is a non-member function and has the ability to access the private data of the class.

To make an outside function friendly to the class, we need to declare the function as a friend of the class as shown below:

```
class sample
{
   // data members;
 public:
friend void abc(void);
};
```

Following are the characteristics of a friend function:

- The friend function is not in the scope of the class in which it has been declared.
- Since it is not in the scope of the class, so it cannot be called by using the object of the class. Therefore, friend function can be invoked like a normal function.
- A friend function cannot access the private members directly, it has to use an object name and dot operator with each member name.
- Friend function uses objects as arguments.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
class Addition
{
 int a=5;
 int b=6;
 public:
 friend int add(Addition a1)
 {
    return(a1.a+a1.b);
 }
};
int main()
{
int result;
Addition a1;
 result=add(a1);
 cout<<result;
return 0;
}
```

Output:

23) What is a virtual function?

- A virtual function is used to replace the implementation provided by the base class. The replacement is always called whenever the object in question is actually of the derived class, even if the object is accessed by a base pointer rather than a derived pointer.
- A virtual function is a member function which is present in the base class and redefined by the derived class.
- When we use the same function name in both base and derived class, the function in base class is declared with a keyword virtual.
- When the function is made virtual, then C++ determines at run-time which function is to be called based on the type of the object pointed by the base class pointer. Thus, by making the base class pointer to point different objects, we can execute different versions of the virtual functions.

Rules of a virtual function:

- The virtual functions should be a member of some class.
- The virtual function cannot be a static member.
- Virtual functions are called by using the object pointer.
- It can be a friend of another class.
- C++ does not contain virtual constructors but can have a virtual destructor.

24) When should we use multiple inheritance?

You can answer this question in three manners:

1. Never
2. Rarely
3. If you find that the problem domain cannot be accurately modeled any other way.

25) What is a destructor?

A Destructor is used to delete any extra resources allocated by the object. A destructor function is called automatically once the object goes out of the scope.

Rules of destructor:

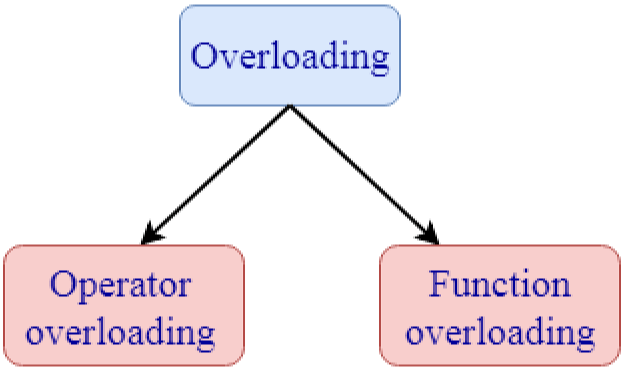
- Destructors have the same name as class name and it is preceded by tilde.
- It does not contain any argument and no return type.

26) What is an overflow error?

It is a type of arithmetical error. It happens when the result of an arithmetical operation been greater than the actual space provided by the system.

27) What is overloading?

- When a single object behaves in many ways is known as overloading. A single object has the same name, but it provides different versions of the same function.
- C++ facilitates you to specify more than one definition for a function name or an operator in the same scope. It is called function overloading and operator overloading respectively.
- Overloading is of two types:



1. Operator overloading: Operator overloading is a compile-time polymorphism in which a standard operator is overloaded to provide a user-defined definition to it. For example, '+' operator is overloaded to perform the addition operation on data types such as int, float, etc.

Operator overloading can be implemented in the following functions:

- Member function
- Non-member function
- Friend function

Syntax of Operator overloading:

```
    Return_type classname :: Operator Operator_symbol(argument_list)
{
    // body_statements;
}
```

2. Function overloading: Function overloading is also a type of compile-time polymorphism which can define a family of functions with the same name. The function would perform different operations based on the argument list in the function call. The function to be

invoked depends on the number of arguments and the type of the arguments in the argument list.

28) What is function overriding?

If you inherit a class into a derived class and provide a definition for one of the base class's function again inside the derived class, then this function is called overridden function, and this mechanism is known as function overriding.

29) What is virtual inheritance?

Virtual inheritance facilitates you to create only one copy of each object even if the object appears more than one in the hierarchy.

30) What is a constructor?

A Constructor is a special method that initializes an object. Its name must be same as class name.

31) What is the purpose of the "delete" operator?

The "delete" operator is used to release the dynamic memory created by "new" operator.

32) Explain this pointer?

This pointer holds the address of the current object.

33) What does Scope Resolution operator do?

A scope resolution operator(::) is used to define the member function outside the class.

34) What is the difference between delete and delete[]?

Delete [] is used to release the array of allocated memory which was allocated using new[] whereas delete is used to release one chunk of memory which was allocated using new.

35) What is a pure virtual function?

The pure virtual function is a virtual function which does not contain any definition. The normal function is preceded with a keyword virtual. The pure virtual function ends with 0.

Syntax of a pure virtual function:

```
virtual void abc()=0;   //pure virtual function.
```

Let's understand this through an example:

```
#include<iostream>
using namespace std;
class Base
{
    public:
    virtual void show()=0;
};

class Derived:public Base
{
    public:
    void show()
    {
        cout<<"javaTpoint";
    }
      };
      int main()
{
    Base* b;
    Derived d;
    b=&d;
    b->show();
    return 0;
}
```

Output:

```
javaTpoint
```

---

36) What is the difference between struct and class?

Structures	class
A structure is a user-defined data type which contains variables of dissimilar data types.	The class is a user-defined data type which contains member variables and member functions.
The variables of a structure are stored in the stack memory.	The variables of a class are stored in the heap memory.
We cannot initialize the variables directly.	We can initialize the member variables directly.
If access specifier is not specified, then by default the access specifier of the variable is "public".	If access specifier is not specified, then by default the access specifier of a variable is "private".
The instance of a structure is a "structure variable".	

Declaration of a structure:
struct structure_name
{
// body of structure;
} ;
	Declaration of class:
	class class_name
	{
	// body of class;
	}

A structure is declared by using a struct keyword.	The class is declared by using a class keyword.
The structure does not support the inheritance.	The class supports the concept of inheritance.
The type of a structure is a value type.	The type of a class is a reference type.

37) What is a class template?

A class template is used to create a family of classes and functions. For example, we can create a template of an array class which will enable us to create an array of various types such as int, float, char, etc. Similarly, we can create a template for a function, suppose we have a function add(), then we can create multiple versions of add().

The syntax of a class template:

```
template<class T>
class classname
{
 // body of class;
};
```

Syntax of a object of a template class:

```
classname<type> objectname(arglist);
```

38) What is the difference between function overloading and operator overloading?

Function overloading: Function overloading is defined as we can have more than one version of the same function. The versions of a function will have different signature means that they have a different set of parameters.

Operator overloading: Operator overloading is defined as the standard operator can be redefined so that it has a different meaning when applied to the instances of a class.

39) What is a virtual destructor?

A virtual destructor in C++ is used in the base class so that the derived class object can also be destroyed. A virtual destructor is declared by using the ~ tilde operator and then virtual keyword before the constructor.

Let's understand this through an example

- Example without using virtual destructor

```
#include <iostream>
using namespace std;
class Base
{
    public:
    Base()
    {
        cout<<"Base constructor is called"<<"\n";
    }
    ~Base()
    {
        cout<<"Base class object is destroyed"<<"\n";
    }
};
class Derived:public Base
{
    public:
    Derived()
    {
        cout<<"Derived class constructor is called"<<"\n";
    }
    ~Derived()
    {
        cout<<"Derived class object is destroyed"<<"\n";
    }
};
int main()
{
  Base* b= new Derived;
  delete b;
  return 0;

}
```

Output:

```
Base constructor is called
Derived class constructor is called
Base class object is destroyed
```

In the above example, delete b will only call the base class destructor due to which derived class destructor remains undestroyed. This leads to the memory leak.

- Example with a virtual destructor

```
#include <iostream>
using namespace std;
class Base
{
    public:
    Base()
    {
        cout<<"Base constructor is called"<<"\n";
    }
    virtual ~Base()
    {
        cout<<"Base class object is destroyed"<<"\n";
    }
};
```

```
class Derived:public Base
{
    public:
    Derived()
    {
        cout<<"Derived class constructor is called"<<"\n";
    }
    ~Derived()
    {
        cout<<"Derived class object is destroyed"<<"\n";
    }
};
int main()
{
  Base* b= new Derived;
  delete b;
  return 0;

}
```

Output:

```
Base constructor is called
Derived class constructor is called
Derived class object is destroyed
Base class object is destroyed
```

Write output of following program-

https://www.indiabix.com/cpp-programming/references/discussion-152

Tips and Tricks-

1) There is no better way to do well in Coding interviews than practicing as many coding problems as possible. This will not only train your mind to recognize algorithmic patterns in problems but also give you the much-needed confidence to solve the problem you have never seen before.
2) My second tips are to learn about as many data structure and algorithms as possible. This is an extension of the previous tip but it also involves reading and not just practicing. For example, If you know about the hash table you can also many array and counter-based problems easily. Same is true for tree and graph.
3) Choosing the right data structure is a very important part of software development and coding interview and unless and until you know them, you won't be able to choose.
4) Time yourself — candidates who solve interview problems within the time limit and quickly are more likely to do well in the interview so you should also time yourself.
5) Think of edge cases and run your code through them. Some good edge cases might be the empty input, some weird input or some really large input to test the boundary conditions and limits.
6) After solving the problem, try explaining it to a friend or colleagues how is also interested in coding problems. This will tell you whether you have really understood the problem or not. If you can explain easily means you understood. Also, the discussion makes your mind work and you could come up with an alternative solution and able to find some flaws in your existing algorithms.

Another useful tip to excel Coding interviews is to appear in the coding interview and lots of them. You will find yourself getting better after every interview and this also helps you to get multiple offers which further allows you to better negotiate and get those extra 30K to 50K which you generally leave on a table if you just have one offer in hand.