

C++ Programming Language

C++ is a general-purpose programming language and is widely used nowadays for competitive programming. It has imperative, object-oriented and generic programming features. C++ runs on lots of platforms like Windows, Linux, Unix, Mac etc.

1. Introduction to C++ Programming Language

C++ is a general-purpose programming language that was developed as an enhancement of the C language to include object-oriented paradigm. It is an imperative and a compiled language.



C++ is a middle-level language rendering it the advantage of programming low-level (drivers, kernels) and even higher-level applications (games, GUI, desktop apps etc.). The basic syntax and code structure of both C and C++ are the same.

1.1. Some of the features & key-points to note about the programming language are as follows:

- Simple: It is a simple language in the sense that programs can be broken down into logical units and parts, has a rich library support and a variety of data-types.
- Machine Independent but Platform Dependent: A C++ executable is not platform-independent (compiled programs on Linux won't run on Windows), however they are machine independent.
- Mid-level language: It is a mid-level language as we can do both systems-programming (drivers, kernels, networking etc.) and build large-scale user applications (Media Players, Photoshop, Game Engines etc.)
- Rich library support: Has a rich library support (Both standard ~ built-in data structures, algorithms etc.) as well 3rd party libraries (e.g. Boost libraries) for fast and rapid development.
- Speed of execution: C++ programs excel in execution speed. Since, it is a compiled language, and also hugely procedural. Newer languages have extra in-built default features such as

garbage-collection, dynamic typing etc. which slow the execution of the program overall. Since there is no additional processing overhead like this in C++, it is blazing fast.

- Pointer and direct Memory-Access: C++ provides pointer support which aids users to directly manipulate storage address. This helps in doing low-level programming (where one might need to have explicit control on the storage of variables).
- Object-Oriented: One of the strongest points of the language which sets it apart from C. Object-Oriented support helps C++ to make maintainable and extensible programs. i.e. Large-scale applications can be built. Procedural code becomes difficult to maintain as code-size grows.
- Compiled Language: C++ is a compiled language, contributing to its speed.

1.2.Applications of C++:

C++ finds varied usage in applications such as:

- Operating Systems & Systems Programming. e.g. Linux-based OS (Ubuntu etc.)
- Browsers (Chrome & Firefox)
- Graphics & Game engines (Photoshop, Blender, Unreal-Engine)
- Database Engines (MySQL, MongoDB, Redis etc.)
- Cloud/Distributed Systems

1.3Some interesting facts about C++:

Here are some awesome facts about C++ that may interest you:

The name of C++ signifies the evolutionary nature of the changes from C. “++” is the C increment operator.

- C++ is one of the predominant languages for the development of all kind of technical and commercial software.
- C++ introduces Object-Oriented Programming, not present in C. Like other things, C++ supports the four primary features of OOP: encapsulation, polymorphism, abstraction, and inheritance.
- C++ got the OOP features from Simula67 Programming language.
- A function is a minimum requirement for a C++ program to run.(at least main() function)

2.C++ Programming Basics

C++ is a general-purpose programming language and widely used nowadays for competitive programming. It has imperative, object-oriented and generic programming features. C++ runs on lots of platform like Windows, Linux, Unix, Mac, etc.

Therefore, below are the basics of C++ in the format in which it will help you the most to get the headstart:

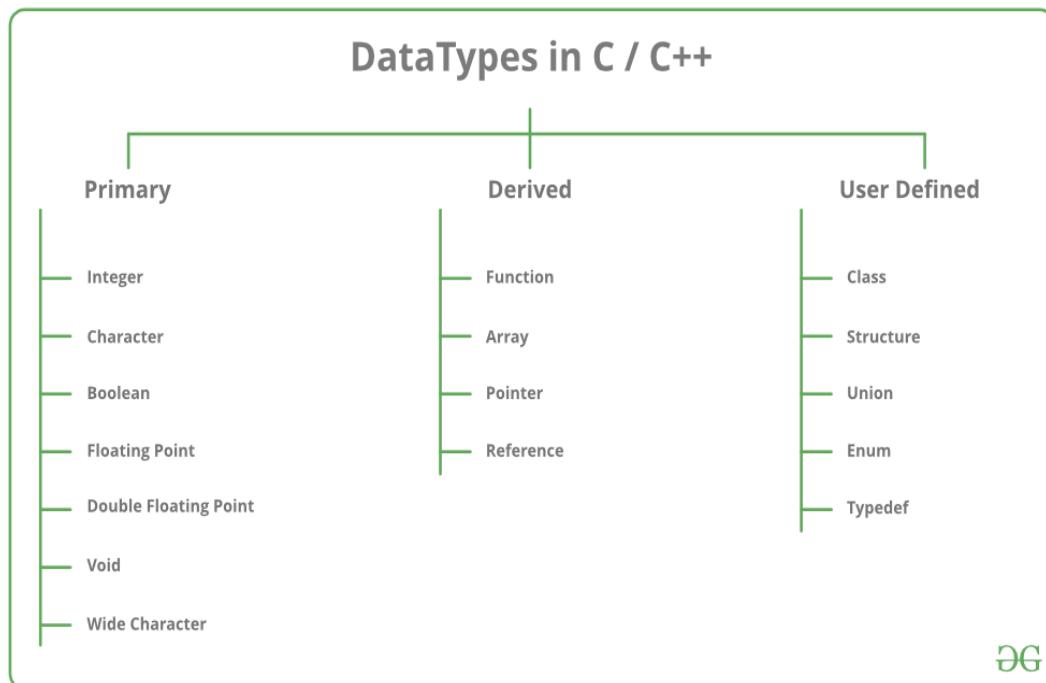
- **Basic Syntax and First Program in C++:** Learning C++ programming can be simplified into writing your program in a text editor and saving it with correct extension(.CPP, .C, .CP) and compiling your program using a compiler or online IDE. The “Hello World” program is the first step towards learning any programming language and also one of the simplest programs you will learn.
- **Basic I/O in C++:** C++ comes with libraries which provides us with many ways for performing input and output. In C++ input and output is performed in the form of a sequence of bytes or more commonly known as streams. The two keywords cin and cout are used very often for taking inputs and printing outputs respectively. These two are the most basic methods of taking input and output in C++.
- **Comments in C++:** A well-documented program is a good practice as a programmer. It makes a program more readable and error finding become easier. One important part of good documentation is Comments. In computer programming, a comment is a programmer-readable explanation or annotation in the source code of a computer program. These are statements that are not executed by the compiler and interpreter.
- **Data Types and Modifiers in C++:** All variables use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data it can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data-type with which it is declared. Every data type requires a different amount of memory.
- **Uninitialized variable in C++:** “One of the things that has kept C++ viable is the zero-overhead rule: What you don’t use, you don’t pay for.” -Stroustrup. The overhead of initializing a stack variable is costly as it hampers the speed of execution, therefore these variables can contain indeterminate values. It is considered a best practice to initialize a primitive data type variable before using it in code.
- **Undefined Behaviour in C++:** If a user starts learning in C/C++ environment and is unclear with the concept of undefined behaviour then that can bring plenty of problems in the future like while debugging someone else code might be actually difficult in tracing the root to the undefined error.
- **Variables and Types in C++:** A variable is a name given to a memory location. It is the basic unit of storage in a program. The value stored in a variable can be changed during program execution. A variable is only a name given to a memory location, all the operations done on the variable effects that memory location. In C++, all the variables must be declared before use.
- **Variable Scope in C++:** In general, scope is defined as the extent up to which something can be worked with. In programming also the scope of a variable is defined as the extent

of the program code within which the variable can be accessed or declared or worked with. There are mainly two types of variable scopes, Local and Global Variables.

- Constants and Literals in C++: As the name suggests the name constants is given to such variables or values in C++ programming language which cannot be modified once they are defined. They are fixed values in a program. There can be any types of constants like integer, float, octal, hexadecimal, character constants, etc. Every constant has some range. The integers that are too big to fit into an int will be taken as long. Now there are various ranges that differ from unsigned to signed bits. Under the signed bit, the range of an int varies from -128 to +127 and under the unsigned bit, int varies from 0 to 255. Literals are kind of constants and both the terms are used interchangeably in C++.
- Types of Literals in C++: In this article we will analyse the various kind of literals that C++ provides. The values assigned to each constant variables are referred to as the literals. Generally, both terms, constants and literals are used interchangeably. For eg, “const int = 5;”, is a constant expression and the value 5 is referred to as constant integer literal.
- Access Modifiers in C++: Access modifiers are used to implement an important feature of Object-Oriented Programming known as Data Hiding. Access modifiers or Access Specifiers in a class are used to set the accessibility of the class members. That is, it sets some restrictions on the class members not to get directly accessed by the outside functions.
- Storage Classes in C++: Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility, and life-time which help us to trace the existence of a particular variable during the runtime of a program.
- Operators in C++: Operators are the foundation of any programming language. Thus the functionality of C/C++ programming language is incomplete without the use of operators. We can define operators as symbols that help us to perform specific mathematical and logical computations on operands. In other words, we can say that an operator operates the operands.
- Loops in C++: Loops in programming comes into use when we need to repeatedly execute a block of statements. For example: Suppose we want to print “Hello World” 10 times. This can be done in two ways, Iterative method and by using Loops.
- Decision Making in C++: There comes situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code. Decision-making statements in programming languages decide the direction of flow of program execution.
- Forward declarations in C++: It refers to the beforehand declaration of the syntax or signature of an identifier, variable, function, class, etc. prior to its usage (done later in the program). In C++, Forward declarations are usually used for Classes. In this, the class is pre-defined before its use so that it can be called and used by other classes that are defined before this.
- Errors in C++: Error is an illegal operation performed by the user which results in abnormal working of the program. Programming errors often remain undetected until the program is compiled or executed. Some of the errors inhibit the program from getting compiled or executed. Thus errors should be removed before compiling and executing.

3.C++ Data Types

All variables use data-type during declaration to restrict the type of data to be stored. Therefore, we can say that data types are used to tell the variables the type of data it can store. Whenever a variable is defined in C++, the compiler allocates some memory for that variable based on the data type with which it is declared. Every data type requires a different amount of memory



Data types in C++ are mainly divided into three types:

3.1. Primitive Data Types: These data types are built-in or predefined data types and can be used directly by the user to declare variables. example: int, char, float, bool, etc.

Primitive data types available in C++ are:

- Integer
- Character
- Boolean
- Floating Point
- Double Floating Point
- Valueless or Void
- Wide Character

3.2. Derived Data Types: The data types that are derived from the primitive or built-in datatypes are referred to as Derived Data Types.

These can be of four types namely:

- Function
- Array
- Pointer
- Reference

3.3. Abstract or User-Defined Data Types: These data types are defined by the user itself. Like, as defining a class in C++ or a structure.

C++ provides the following user-defined datatypes:

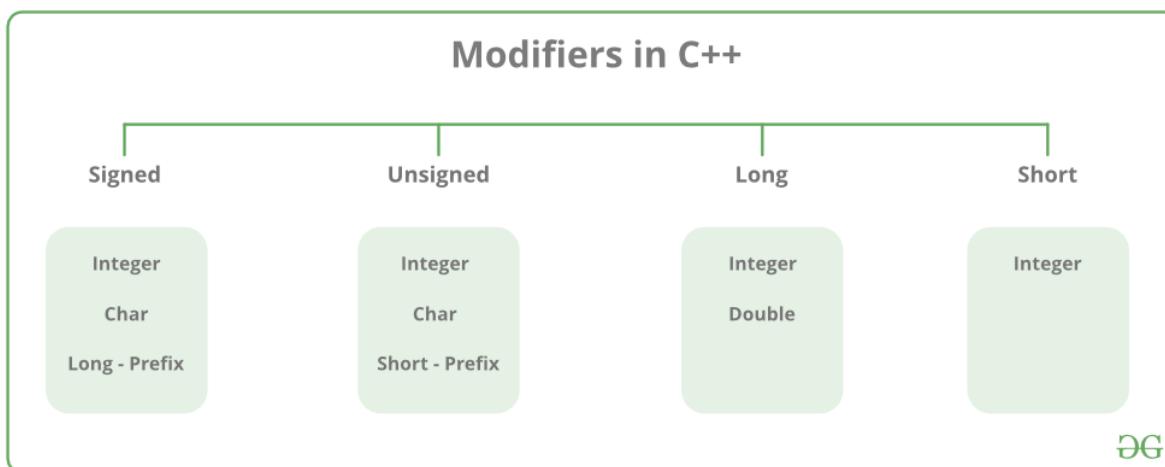
- Class
- Structure
- Union
- Enumeration
- Typedef defined DataType

This article discusses primitive data types available in C++.

- Integer: The keyword used for integer data types is int. Integers typically require 4 bytes of memory space and range from -2147483648 to 2147483647.
- Character: Character data type is used for storing characters. The keyword used for the character data type is char. Characters typically require 1 byte of memory space and range from -128 to 127 or 0 to 255.
- Boolean: Boolean data type is used for storing boolean or logical values. A boolean variable can store either true or false. The keyword used for the boolean data type is bool.
- Floating Point: Floating Point data type is used for storing single-precision floating-point values or decimal values. The keyword used for the floating-point data type is float. Float variables typically require 4 bytes of memory space.
- Double Floating Point: Double Floating Point data type is used for storing double-precision floating-point values or decimal values. The keyword used for the double floating-point data type is double. Double variables typically require 8 bytes of memory space.
- void: Void means without any value. void data type represents a valueless entity. A void data type is used for those function which does not return a value.
- Wide Character: Wide character data type is also a character data type but this data type has a size greater than the normal 8-bit datatype. Represented by wchar_t. It is generally 2 or 4 bytes long.

4.1 Datatype Modifiers

As the name implies, datatype modifiers are used with the built-in data types to modify the length of data that a particular data type can hold.



Data type modifiers available in C++ are:

- Signed
- Unsigned
- Short
- Long

The below table summarizes the modified size and range of built-in datatypes when combined with the type modifiers:

Data Type	Size (in bytes)	Range
short int	2	-32,768 to 32,767
unsigned short int	2	0 to 65,535
unsigned int	4	0 to 4,294,967,295
int	4	-2,147,483,648 to 2,147,483,647

Data Type	Size (in bytes)	Range
long int	4	-2,147,483,648 to 2,147,483,647
unsigned long int	4	0 to 4,294,967,295
long long int	8	-(2^63) to (2^63)-1
unsigned long long int	8	0 to 18,446,744,073,709,551,615
signed char	1	-128 to 127
unsigned char	1	0 to 255
float	4	
double	8	
long double	12	
wchar_t	2 or 4	1 wide character

Note: Above values may vary from compiler to compiler. In the above example, we have considered GCC 32 bit.

We can display the size of all the data types by using the sizeof() operator and passing the keyword of the datatype as an argument to this function as shown below:

Now to get the range of data types refer to the following chart

Note: syntax<limits.h> header file is defined to find the range of fundamental data-types. Unsigned modifiers have minimum value is zero. So, no macro constants are defined for the unsigned minimum value.

Macro Constants	
Name	
CHAR_MIN	Minimum value for an

object of type char	
CHAR_MAX	Maximum value for an object of type char
object of type char	
SCHAR_MIN	Minimum value for an object of type Signed char
object of type Signed char	
SCHAR_MAX	Maximum value for an object of type Signed char
object of type Unsigned char	
UCHAR_MAX	Maximum value for an object of type Unsigned char
CHAR_BIT	Number of bits in a char object
MB_LEN_MAX	Maximum number of bytes in a multi-byte character
SHRT_MIN	Minimum value for an object of type short int
SHRT_MAX	Maximum value for an object of type short int
USHRT_MAX	Maximum value for an object of type Unsigned short int
INT_MIN	Minimum value for an object of type int
INT_MAX	Maximum value for an object of type int
UINT_MAX	Maximum value for an object of type Unsigned int
LONG_MIN	Minimum value for an object of type long int
LONG_MAX	Maximum value for an object of type long int
ULONG_MAX	Maximum value for an object of type Unsigned long int
LLONG_MIN	Minimum value for an object of type long long int
LLONG_MAX	Maximum value for an object of type long long int
ULLONG_MAX	Maximum value for an object of type Unsigned long long int

The actual value depends on the particular system and library implementation but shall reflect the limits of these types in the target platform. LLONG_MIN, LLONG_MAX, and ULLONG_MAX are defined for libraries complying with the C standard of 1999 or later (which only includes the C++ standard since 2011: C++11).

C++ Program to Find the Range of Data Types using Macro Constants

```

// C++ program to sizes of data types
#include <iostream>
#include <limits.h>
using namespace std;

int main()
{
    cout << "Size of char : " << sizeof(char) << " byte"
        << endl;
    cout << "char minimum value: " << CHAR_MIN << endl;

    cout << "char maximum value: " << CHAR_MAX << endl;

    cout << "Size of int : " << sizeof(int) << " bytes"
        << endl;
    cout << "Size of short int : " << sizeof(short int)
        << " bytes" << endl;
    cout << "Size of long int : " << sizeof(long int)
        << " bytes" << endl;
    cout << "Size of signed long int : "
        << sizeof(signed long int) << " bytes" << endl;
    cout << "Size of unsigned long int : "
        << sizeof(unsigned long int) << " bytes" << endl;
    cout << "Size of float : " << sizeof(float) << " bytes"
        << endl;
    cout << "Size of double : " << sizeof(double)
        << " bytes" << endl;
    cout << "Size of wchar_t : " << sizeof(wchar_t)
        << " bytes" << endl;

    return 0;
}

```

Output:

```

Size of char : 1 byte
Size of int : 4 bytes
Size of short int : 2 bytes
Size of long int : 8 bytes
Size of signed long int : 8 bytes
Size of unsigned long int : 8 bytes
Size of float : 4 bytes
Size of double : 8 bytes
Size of wchar_t : 4 bytes

```

Interesting facts about data-types and modifiers in C/C++

Here are some logical and interesting facts about data-types and the modifiers associated with data-types:-

- 1. If no data type is given to a variable, the compiler automatically converts it to int data type.**

- C++
- C

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

int main()
{
    signed a;
    signed b;

    // size of a and b is equal to the size of int
    cout << "The size of a is " << sizeof(a) << endl;
    cout << "The size of b is " << sizeof(b);
    return (0);
}

// This code is contributed by shubhamsingh10
```

Output:

- 2. Signed is the default modifier for char and int data types.**

- C++
- C

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

int main()
{
    int x;
    char y;
    x = -1;
    y = -2;
    cout << "x is "<< x << " and y is " << y << endl;
```

```
}
```

// This code is contributed by shubhamsingh10

Output:

3. We can't use any modifiers in float data type. If programmer tries to use it ,the compiler automatically gives compile time error.

- C++
- C

```
#include <iostream>
using namespace std;
int main()
{
    signed float a;
    short float b;
    return (0);
}
//This article is contributed by shivanisinghss2110
```

Output:

4. Only long modifier is allowed in double data types. We cant use any other specifier with double data type. If we try any other specifier, compiler will give compile time error.

- C++
- C

```
#include <iostream>
using namespace std;
int main()
{
    long double a;
    return (0);
}

// This code is contributed by shubhamsingh10
```

- C++

- C

```
#include <iostream>
using namespace std;
int main()
{
    short double a;
    signed double b;
    return (0);
}

// This code is contributed by shubhamsingh10
```

Output:

4.2.Uninitialized primitive data types in C/C++

What do you think happens when you use an uninitialized primitive data type?

Well you may assume that the compiler should assign your primitive type variable with meaningful values like 0 for int, 0.0 for float. What about char data type?

Let's find the answer to that by running the code in the IDE.

```
#include <iostream>

using namespace std;

int main()
{
    // The following primitive data type variables will not
    // be initialized with any default values
    char ch;
    float f;
    int i;
    double d;
    long l;

    cout << ch << endl;
    cout << f << endl;
    cout << i << endl;
    cout << d << endl;
    cout << l << endl;
```

```
    return 0;  
}
```

Output in GFGs IDE:

Output in Codechef IDE:

Output on my machine:

Why C/C++ compiler does not initialize variables with default values?

“One of the things that has kept C++ viable is the zero-overhead rule: What you don’t use, you don’t pay for.” -Stroustrup.

The overhead of initializing a stack variable is costly as it hampers the speed of execution, therefore these variables can contain indeterminate values or garbage values as memory space is provided when we define a data type. It is considered a best practice to initialize a primitive data type variable before using it in code.

What Happen When We Exceed Valid Range of Built-in Data Types in C++?

In this article, we will look at what happened when we exceed the valid range of built-in data types in C++ with some examples.

Example 1: Program to show what happens when we cross the range of ‘char’.

Here, a is declared as char. Here the loop is working from 0 to 225. So, it should print from 0 to 225, then stop. But it will generate an infinite loop. The reason for this is the valid range of character data is -128 to 127. When ‘a’ becomes 128 through a++, the range is exceeded and as a result, the first number from the negative side of the range (i.e. -128) gets assigned to a. As a result of this ‘a’ will never reach point 225. so it will print the infinite series of characters.

- CPP

```
// C++ program to demonstrate
// the problem with 'char'
#include <iostream>

using namespace std;

int main()
{
    for (char a = 0; a <= 225; a++)
        cout << a;
    return 0;
}
```

This code will print ‘1’ infinite time because here ‘a’ is declared as ‘bool’ and its valid range is 0 to 1. And for a Boolean variable, anything else than 0 is 1 (or true). When ‘a’ tries to become 2 (through a++), 1 gets assigned to ‘a’. The condition a<=5 is satisfied and the control remains within the loop.

Example 2: Program to show what happens when we cross the range of ‘bool’

- CPP

```
// C++ program to demonstrate
// the problem with 'bool'
#include <iostream>

using namespace std;

int main()
{
    // declaring Boolean
    // variable with true value
    bool a = true;
```

```

for (a = 1; a <= 5; a++)
    cout << a;

return 0;
}

```

Will this code print ‘a’ till it becomes 32770? Well, the answer is an indefinite loop because here ‘a’ is declared as a short and its valid range is -32768 to +32767. When ‘a’ tries to become 32768 through a++, the range is exceeded and as a result, the first number from the negative side of the range(i.e. -32768) gets assigned to a. Hence the condition “a < 32770” is satisfied and control remains within the loop.

Example 3: Program to show what happens when we cross the range of ‘short’

Note that short is short for short int. They are synonymous. short, short int, signed short, and signed short int are all the same data type.

- CPP

```

// C++ program to demonstrate
// the problem with 'short'
#include <iostream>

using namespace std;

int main()
{
    // declaring short variable
    short a;

    for (a = 32767; a < 32770; a++)
        cout << a << "\n";

    return 0;
}

```

Example 4: Program to show what happens when we cross the range of ‘unsigned short’

- CPP

```

// C++ program to demonstrate
// the problem with 'unsigned short'
#include <iostream>

using namespace std;

```

```

int main()
{
    unsigned short a;

    for (a = 65532; a < 65536; a++)
        cout << a << "\n";

    return 0;
}

```

Will this code print ‘a’ till it becomes 65536? Well, the answer is an indefinite loop, because here ‘a’ is declared as a short and its valid range is 0 to +65535. When ‘a’ tries to become 65536 through a++, the range is exceeded and as a result, the first number from the range(i.e. 0) gets assigned to a. Hence the condition “a < 65536” is satisfied and control remains within the loop.

Explanation: We know that the computer uses 2’s complement to represent data. For example, if we have 1 byte (We can use char and use %d as a format specifier to view it as a decimal), we can represent -128 to 127. If we add 1 to 127 we will get -128. That’s because 127 is 01111111 in binary. And if we add 1 into 01111111 we will get 10000000. 10000000 is -128 in 2’s complement form. The same will happen if we use unsigned integers. 255 is 11111111 when we add 1 to 11111111 we will get 100000000. But we are using only the first 8 bits, so that’s 0. Hence we get 0 after adding 1 in 255.

Data types that supports std::numeric_limits() in C++

https://www.geeksforgeeks.org/data-types-that-supports-stdnumeric_limits-in-cpp/?ref=rp

5. Variables in C++

A variable is a name given to a memory location. It is the basic unit of storage in a program.

The value stored in a variable can be changed during program execution.

A variable is only a name given to a memory location, all the operations done on the variable effects that memory location.

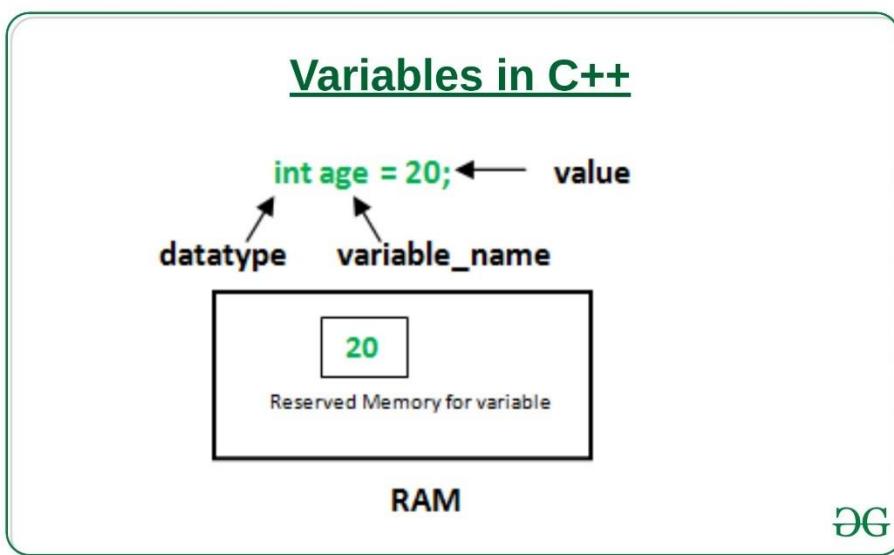
In C++, all the variables must be declared before use.

How to declare variables?

A typical variable declaration is of the form:

```
// Declaring a single variable  
type variable_name;  
  
// Declaring multiple variables:  
type variable1_name, variable2_name, variable3_name;
```

A variable name can consist of alphabets (both upper and lower case), numbers and the underscore '_' character. However, the name must not start with a number.



Examples:

```
// Declaring float variable  
float simpleInterest;
```

```
// Declaring integer variable  
int time, speed;
```

```
// Declaring character variable  
char var;
```

5.1.Difference between variable declaration and definition

The variable declaration refers to the part where a variable is first declared or introduced before its first use. A variable definition is a part where the variable is assigned a memory location and a value. Most of the time, variable declaration and definition are done together.

See the following C++ program for better clarification:

```
// C++ program to show difference b/w definition and declaration of a  
variable  
  
#include <iostream>  
using namespace std;  
  
int main()  
{  
    // this is declaration of variable a  
    int a;  
    // this is initialisation of a  
    a = 10;  
    // this is definition = declaration + initialisation  
    int b = 20;  
  
    // declaration and definition  
    // of variable 'a123'  
    char a123 = 'a';  
  
    // This is also both declaration and definition  
    // as 'b' is allocated memory and
```

```

// assigned some garbage value.
float c;

// multiple declarations and definitions
int _c, _d45, e;

// Let us print a variable
cout << a123 << endl;

return 0;
}

```

Output:

a

5.2.Types of variables

There are three types of variables based on the scope of variables in C++:

Local Variables

Instance Variables

Static Variables

Types of variables in C++

```

class GFG {

public:
    static int a; — Static Variable
    int b; — Instance Variable

public:
    func()
    {
        int c; — Local Variable
    };
}

```

DG

Let us now learn about each one of these variables in detail.

5.2.1.Local Variables: A variable defined within a block or method or constructor is called a local variable.

- These variables are created when the block is entered or the function is called and destroyed after exiting from the block or when the call returns from the function.
- The scope of these variables exists only within the block in which the variable is declared. i.e. we can access this variable only within that block.
- Initialization of Local Variable is Mandatory.

5.2.2.Instance Variables: Instance variables are non-static variables and are declared in a class outside any method, constructor or block.

- As instance variables are declared in a class, these variables are created when an object of the class is created and destroyed when the object is destroyed.
- Unlike local variables, we may use access specifiers for instance variables. If we do not specify any access specifier then the default access specifier will be used.
- Initialization of Instance Variable is not Mandatory.
- Instance Variable can be accessed only by creating objects.

5.2.3.Static Variables: Static variables are also known as Class variables.

- These variables are declared similarly as instance variables, the difference is that static variables are declared using the static keyword within a class outside any method constructor or block.
- Unlike instance variables, we can only have one copy of a static variable per class irrespective of how many objects we create.
- Static variables are created at the start of program execution and destroyed automatically when execution ends.
- Initialization of Static Variable is not Mandatory. Its default value is 0
- If we access the static variable like the Instance variable (through an object), the compiler will show the warning message and it won't halt the program. The compiler will replace the object name with the class name automatically.
- If we access the static variable without the class name, the Compiler will automatically append the class name.

5.3.Instance variable Vs Static variable

- Each object will have its own copy of the instance variable whereas We can only have one copy of a static variable per class irrespective of how many objects we create.
- Changes made in an instance variable using one object will not be reflected in other objects as each object has its own copy of the instance variable. In the case of static, changes will be reflected in other objects as static variables are common to all object of a class.
- We can access instance variables through object references and Static Variables can be accessed directly using the class name.

Syntax for static and instance variables:

class Example

```
{  
    static int a; // static variable  
    int b;      // instance variable  
}
```

6.C++ Keywords

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

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

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

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

7.C++ Identifiers

C++ identifiers in a program are used to refer to the name of the variables, functions, arrays, or other user-defined data types created by the programmer. They are the basic requirement of any language. Every language has its own rules for naming the identifiers.

In short, we can say that the C++ identifiers represent the essential elements in a program which are given below:

- **Constants**
- **Variables**
- **Functions**
- **Labels**
- **Defined data types**

Some naming rules are common in both C and C++ . They are as follows:

- Only alphabetic characters, digits, and underscores are allowed.
- The identifier name cannot start with a digit, i.e., the first letter should be alphabetical. After the first letter, we can use letters, digits, or underscores.
- In C++, uppercase and lowercase letters are distinct. Therefore, we can say that C++ identifiers are case-sensitive.
- A declared keyword cannot be used as a variable name.

8.C++ Expression

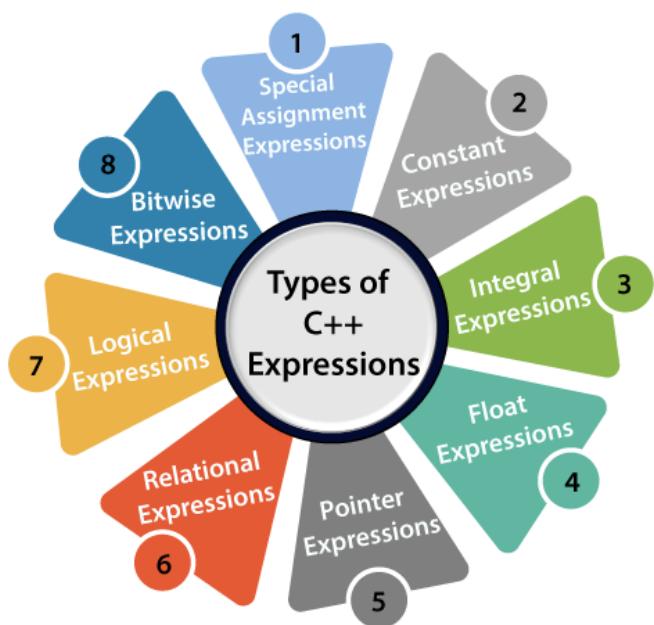
C++ expression consists of operators, constants, and variables which are arranged according to the rules of the language. It can also contain function calls which return values. An expression can consist of one or more operands, zero or more operators to compute a value. Every expression produces some value which is assigned to the variable with the help of an assignment operator.

Examples of C++ expression:

1. $(a+b) - c$
2. $(x/y) - z$
3. $4a^2 - 5b + c$
4. $(a+b) * (x+y)$

An expression can be of following types:

- Constant expressions
- Integral expressions
- Float expressions
- Pointer expressions
- Relational expressions
- Logical expressions
- Bitwise expressions
- Special assignment expressions



If the expression is a combination of the above expressions, such expressions are known as compound expressions.

Constant expressions

A constant expression is an expression that consists of only constant values. It is an expression whose value is determined at the compile-time but evaluated at the run-time. It can be composed of integer, character, floating-point, and enumeration constants.

Constants are used in the following situations:

- It is used in the subscript declarator to describe the array bound.
- It is used after the case keyword in the switch statement.
- It is used as a numeric value in an **enum**
- It specifies a bit-field width.
- It is used in the pre-processor **#if**

In the above scenarios, the constant expression can have integer, character, and enumeration constants. We can use the static and extern keyword with the constants to define the function-scope.

The following table shows the expression containing constant value:

Expression containing constant	Constant value
<code>x = (2/3) * 4</code>	<code>(2/3) * 4</code>
<code>extern int y = 67</code>	67
<code>int z = 43</code>	43
<code>static int a = 56</code>	56

Let's see a simple program containing constant expression:

1. `#include <iostream>`
2. `using namespace std;`
3. `int main()`

```

4. {
5.     int x;      // variable declaration.
6.     x=(3/2) + 2; // constant expression
7.     cout<<"Value of x is :"<<x; // displaying the value of x.
8.     return 0;
9. }
```

In the above code, we have first declared the 'x' variable of integer type. After declaration, we assign the simple constant expression to the 'x' variable.

Output

```
Value of x is : 3
```

Integral Expressions

An integer expression is an expression that produces the integer value as output after performing all the explicit and implicit conversions.

Following are the examples of integral expression:

1. $(x * y) -5$
2. $x + \text{int}(9.0)$
3. where x and y are the integers.

Let's see a simple example of integral expression:

```

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int x; // variable declaration.
6.     int y; // variable declaration
7.     int z; // variable declaration
8.     cout<<"Enter the values of x and y";
9.     cin>>x>>y;
10.    z=x+y;
11.    cout<<"\n"<<"Value of z is :"<<z; // displaying the value of z.
12.    return 0;
13. }
```

In the above code, we have declared three variables, i.e., x, y, and z. After declaration, we take the user input for the values of 'x' and 'y'. Then, we add the values of 'x' and 'y' and stores their result in 'z' variable.

Output

```
Enter the values of x and y
8
9
Value of z is :17
```

Let's see another example of integral expression.

1. `#include <iostream>`
2. `using namespace std;`
3. `int main()`
4. `{`
5.
6. `int x; // variable declaration`
7. `int y=9; // variable initialization`
8. `x=y+int(10.0); // integral expression`
9. `cout<<"Value of x :"<<x; // displaying the value of x.`
10. `return 0;`
11. `}`

In the above code, we declare two variables, i.e., x and y. We store the value of expression ($y+int(10.0)$) in a 'x' variable.

Output

```
Value of x : 19
```

Float Expressions

A float expression is an expression that produces floating-point value as output after performing all the explicit and implicit conversions.

The following are the examples of float expressions:

1. `x+y`
2. `(x/10) + y`
3. `34.5`
4. `x+float(10)`

Let's understand through an example.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.
6.     float x=8.9;    // variable initialization
7.     float y=5.6;    // variable initialization
8.     float z;        // variable declaration
9.     z=x+y;
10.    std::cout <<"value of z is :" << z<<std::endl; // displaying the value of z.
11.
12.
13.    return 0;
14. }
```

Output

```
value of z is :14.5
```

Let's see another example of float expression.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     float x=6.7;    // variable initialization
6.     float y;        // variable declaration
7.     y=x+float(10); // float expression
8.     std::cout <<"value of y is :" << y<<std::endl; // displaying the value of y
9.     return 0;
10. }
```

In the above code, we have declared two variables, i.e., x and y. After declaration, we store the value of expression ($x+float(10)$) in variable 'y'.

Output

```
value of y is :16.7
```

Pointer Expressions

A pointer expression is an expression that produces address value as an output.

The following are the examples of pointer expression:

1. &x
2. ptr
3. ptr++
4. ptr-

Let's understand through an example.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.
6.     int a[]={1,2,3,4,5}; // array initialization
7.     int *ptr; // pointer declaration
8.     ptr=a; // assigning base address of array to the pointer ptr
9.     ptr=ptr+1; // incrementing the value of pointer
10.    std::cout << "value of second element of an array : " << *ptr<<std::endl;
11.    return 0;
12. }
```

In the above code, we declare the array and a pointer ptr. We assign the base address to the variable 'ptr'. After assigning the address, we increment the value of pointer 'ptr'. When pointer is incremented then 'ptr' will be pointing to the second element of the array.

Output

```
value of second element of an array : 2
```

Relational Expressions

A relational expression is an expression that produces a value of type bool, which can be either true or false. It is also known as a boolean expression. When arithmetic expressions are used on both sides of the relational operator, arithmetic expressions are evaluated first, and then their results are compared.

The following are the examples of the relational expression:

1. $a > b$
2. $a - b \geq x - y$
3. $a + b > 80$

Let's understand through an example

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int a=45; // variable declaration
6.     int b=78; // variable declaration
7.     bool y= a>b; // relational expression
8.     cout<<"Value of y is :"<<y; // displaying the value of y.
9.     return 0;
10. }
```

In the above code, we have declared two variables, i.e., 'a' and 'b'. After declaration, we have applied the relational operator between the variables to check whether 'a' is greater than 'b' or not.

Output

```
Value of y is :0
```

Let's see another example.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int a=4; // variable declaration
6.     int b=5; // variable declaration
7.     int x=3; // variable declaration
8.     int y=6; // variable declaration
9.     cout<<((a+b)>=(x+y)); // relational expression
10.    return 0;
11. }
```

In the above code, we have declared four variables, i.e., 'a', 'b', 'x' and 'y'. Then, we apply the relational operator (\geq) between these variables.

Output

1

Logical Expressions

A logical expression is an expression that combines two or more relational expressions and produces a bool type value. The logical operators are '`&&`' and '`||`' that combines two or more relational expressions.

The following are some examples of logical expressions:

1. `a>b && x>y`
2. `a>10 || b==5`

Let's see a simple example of logical expression.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int a=2;
6.     int b=7;
7.     int c=4;
8.     cout<<((a>b)|| (a>c));
9.     return 0;
10. }
```

Output

0

Bitwise Expressions

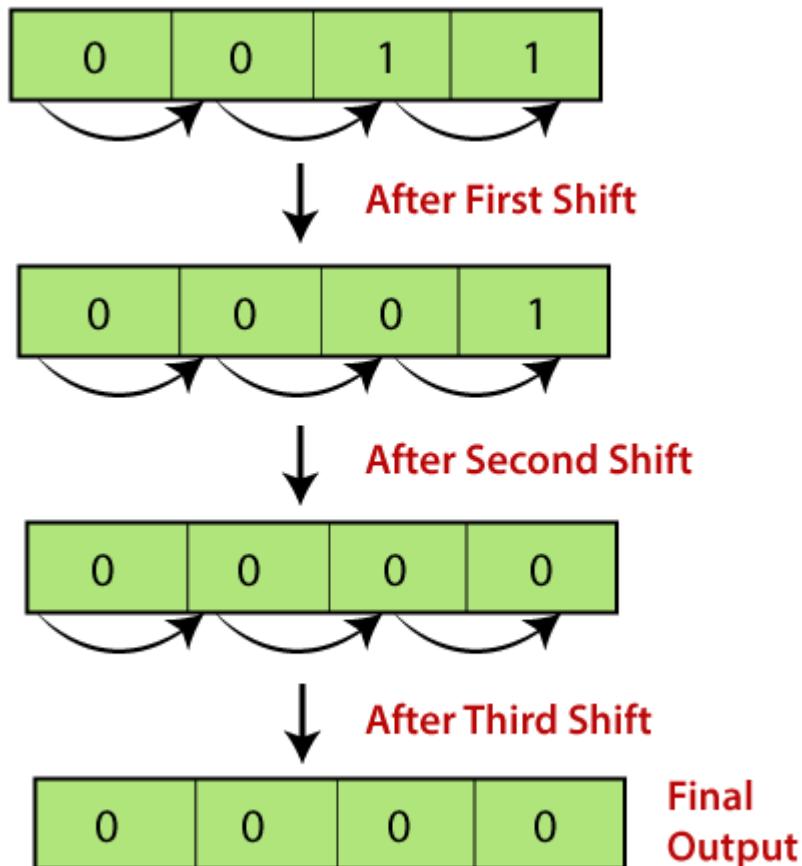
A bitwise expression is an expression which is used to manipulate the data at a bit level. They are basically used to shift the bits.

For example:

`x=3`

`x>>3` // This statement means that we are shifting the three-bit position to the right.

In the above example, the value of 'x' is 3 and its binary value is 0011. We are shifting the value of 'x' by three-bit position to the right. Let's understand through the diagrammatic representation.



Let's see a simple example.

```

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int x=5; // variable declaration
6.     std::cout << (x>>1) << std::endl;
7.     return 0;
8. }
```

In the above code, we have declared a variable 'x'. After declaration, we applied the bitwise operator, i.e., right shift operator to shift one-bit position to right.

Output

2

Let's look at another example.

```
1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int x=7; // variable declaration
6.     std::cout << (x<<3) << std::endl;
7.     return 0;
8. }
```

In the above code, we have declared a variable 'x'. After declaration, we applied the left shift operator to variable 'x' to shift the three-bit position to the left.

Output

56

Special Assignment Expressions

Special assignment expressions are the expressions which can be further classified depending upon the value assigned to the variable.

- **Chained Assignment**

Chained assignment expression is an expression in which the same value is assigned to more than one variable by using single statement.

For example:

1. a=b=20
2. or
3. (a=b) = 20

Let's understand through an example.

```
1. #include <iostream>
2. using namespace std;
3. int main()
```

```

4.
5. int a; // variable declaration
6. int b; // variable declaration
7. a=b=80; // chained assignment
8. std::cout <<"Values of 'a' and 'b' are :" <<a<<","<<b<< std::endl;
9. return 0;
10. }

```

In the above code, we have declared two variables, i.e., 'a' and 'b'. Then, we have assigned the same value to both the variables using chained assignment expression.

Output

```
Values of 'a' and 'b' are : 80,80
```

Note: Using chained assignment expression, the value cannot be assigned to the variable at the time of declaration. For example, `int a=b=c=90` is an invalid statement.

- **Embedded Assignment Expression**

An embedded assignment expression is an assignment expression in which assignment expression is enclosed within another assignment expression.

Let's understand through an example.

```

1. #include <iostream>
2. using namespace std;
3. int main()
4. {
5.     int a; // variable declaration
6.     int b; // variable declaration
7.     a=10+(b=90); // embedded assignment expression
8.     std::cout <<"Values of 'a' is " <<a<< std::endl;
9.     return 0;
10. }

```

In the above code, we have declared two variables, i.e., 'a' and 'b'. Then, we applied embedded assignment expression ($a=10+(b=90)$).

Output

```
Values of 'a' is 100
```

- o **Compound Assignment**

A compound assignment expression is an expression which is a combination of an assignment operator and binary operator.

For example,

1. `a+=10;`

In the above statement, 'a' is a variable and '+=' is a compound statement.

Let's understand through an example.

1. `#include <iostream>`
2. `using namespace std;`
3. `int main()`
4. `{`
5. `int a=10; // variable declaration`
6. `a+=10; // compound assignment`
7. `std::cout << "Value of a is :" <<a<< std::endl; // displaying the value of a.`
8. `return 0;`
9. `}`

In the above code, we have declared a variable 'a' and assigns 10 value to this variable. Then, we applied compound assignment operator (+=) to 'a' variable, i.e., `a+=10` which is equal to $(a=a+10)$. This statement increments the value of 'a' by 10.

Output

```
Value of a is :20
```

9. Loops in C and C++

In programming, sometimes there is a need to perform some operation more than once or (say) n number of times. Loops come into use when we need to repeatedly execute a block of statements.

For example: Suppose we want to print “Hello World” 10 times. This can be done in two ways as shown below:

Manual(general) Method (Iterative Method)

Manually we have to write the print() for C and cout for the C++ statement 10 times. Let's say you have to write it 20 times (it would surely take more time to write 20 statements) now imagine you have to write it 100 times, it would be really hectic to re-write the same statement again and again. So, here loops have their role.

```
// C++ program to illustrate need of loops
#include <iostream>
using namespace std;

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

Output:

```
Hello World
Hello World
Hello World
Hello World
Hello World
Hello World
```

```
Hello World  
Hello World  
Hello World  
Hello World
```

Using Loops

In Loop, the statement needs to be written only once and the loop will be executed 10 times as shown below. In computer programming, a loop is a sequence of instructions that is repeated until a certain condition is reached.

An operation is done, such as getting an item of data and changing it, and then some condition is checked such as whether a counter has reached a prescribed number.

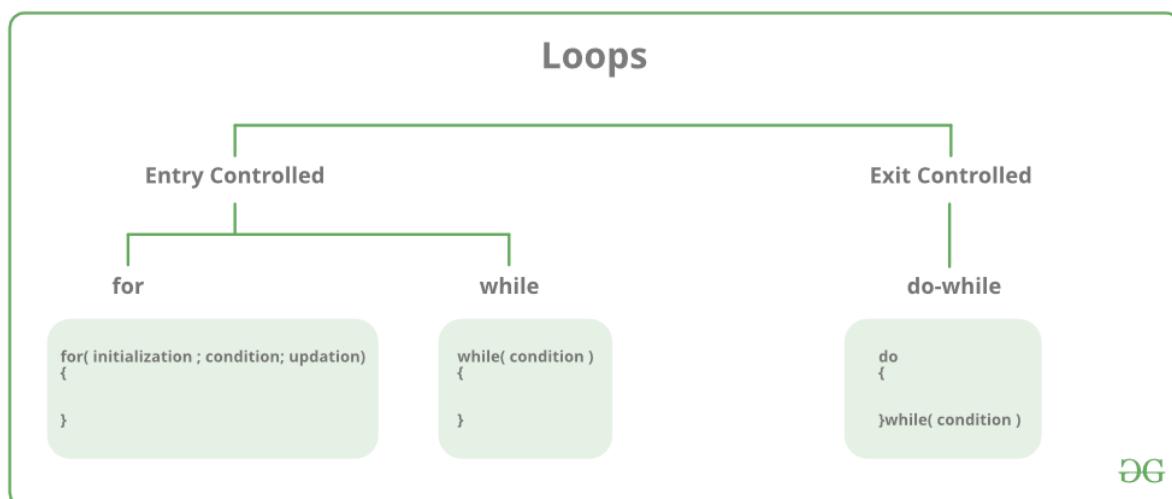
Counter not Reached: If the counter has not reached the desired number, the next instruction in the sequence returns to the first instruction in the sequence and repeats it.

Counter reached: If the condition has been reached, the next instruction “falls through” to the next sequential instruction or branches outside the loop.

There are mainly two types of loops:

Entry Controlled loops: In this type of loop, the test condition is tested before entering the loop body. For Loop and While Loop is entry-controlled loops.

Exit Controlled Loops: In this type of loop the test condition is tested or evaluated at the end of the loop body. Therefore, the loop body will execute at least once, irrespective of whether the test condition is true or false. the do-while loop is exit controlled loop.



S.No. Loop Type and Description

1. **while loop** – First checks the condition, then executes the body.
2. **for loop** – firstly initializes, then, condition check, execute body, update.
3. **do-while** – firstly, execute the body then condition check

9.1. for Loop

A for loop is a repetition control structure that allows us to write a loop that is executed a specific number of times. The loop enables us to perform n number of steps together in one line.

Syntax:

```
for (initialization expr; test expr; update expr)
{
    // body of the loop
    // statements we want to execute
}
```

Example:

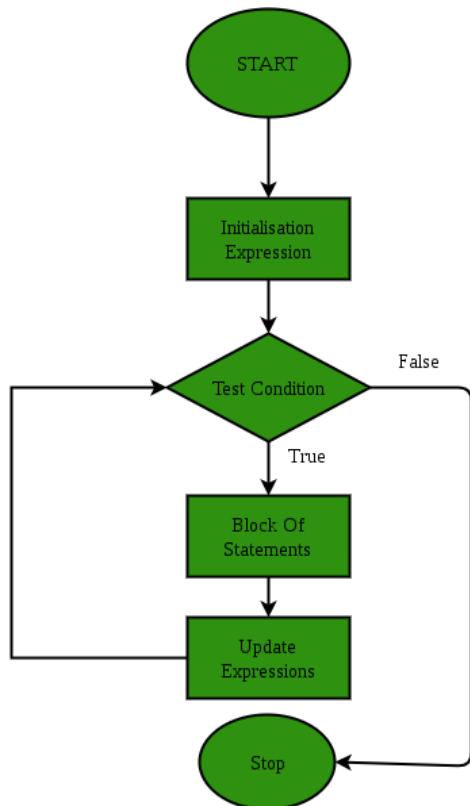
```
for(int i = 0; i < n; i++){
}
```

In for loop, a loop variable is used to control the loop. First, initialize this loop variable to some value, then check whether this variable is less than or greater than the counter value. If the statement is true, then the loop body is executed and the loop variable gets updated. Steps are repeated till the exit condition comes.

- Initialization Expression: In this expression, we have to initialize the loop counter to some value. For example: int i=1;
- Test Expression: In this expression, we have to test the condition. If the condition evaluates to true then we will execute the body of the loop and go to update expression otherwise we will exit from the for a loop. For example: i <= 10;

- Update Expression: After executing the loop body this expression increments/decrements the loop variable by some value. for example: `i++;`

Equivalent Flow Diagram for loop:



Example:

```

// C program to illustrate for loop
#include <stdio.h>

int main()
{
    int i=0;

    for (i = 1; i <= 10; i++)
    {
        printf( "Hello World\n");
    }

    return 0;
}

```

Output:

```

Hello World
Hello World

```

```
Hello World  
Hello World
```

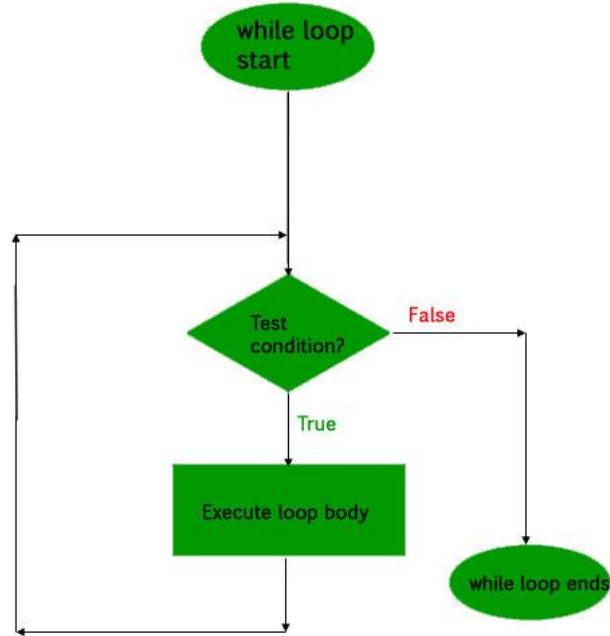
9.2. While Loop

While studying for loop we have seen that the number of iterations is known beforehand, i.e. the number of times the loop body is needed to be executed is known to us. while loops are used in situations where we do not know the exact number of iterations of the loop beforehand. The loop execution is terminated on the basis of the test conditions.

Syntax: We have already stated that a loop mainly consists of three statements – initialization expression, test expression, and update expression. The syntax of the three loops – For, while, and do while mainly differs in the placement of these three statements.

```
initialization expression;  
while (test_expression)  
{  
    // statements  
    update_expression;  
}
```

Flow Diagram:



Example:

```
// C program to illustrate while loop
#include <stdio.h>

int main()
{
    // initialization expression
    int i = 1;

    // test expression
    while (i < 6)
    {
        printf( "Hello World\n");

        // update expression
        i++;
    }

    return 0;
}
```

Output:

```
Hello World
Hello World
Hello World
```

```
Hello World
```

```
Hello World
```

9.3.do-while loop

In do-while loops also the loop execution is terminated on the basis of test conditions. The main difference between a do-while loop and the while loop is in the do-while loop the condition is tested at the end of the loop body, i.e do-while loop is exit controlled whereas the other two loops are entry controlled loops.

Note: In a do-while loop, the loop body will execute at least once irrespective of the test condition.

Syntax:

```
initialization expression;
```

```
do
```

```
{
```

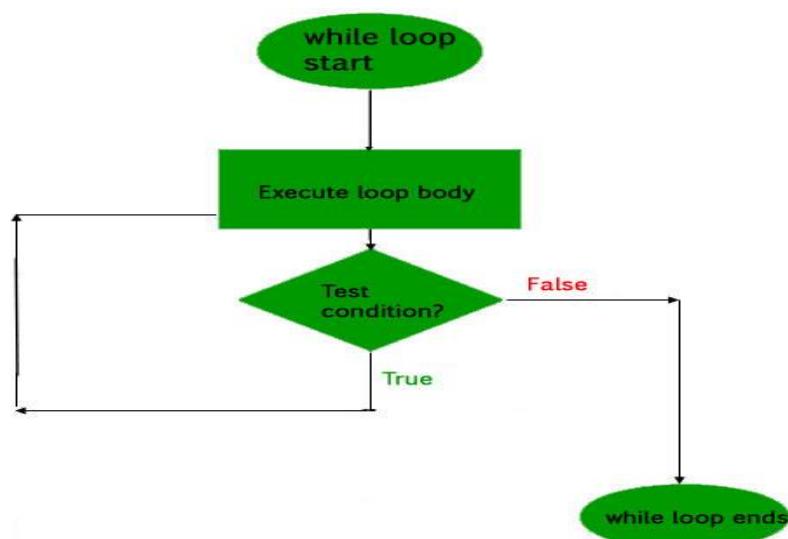
```
// statements
```

```
update_expression;
```

```
} while (test_expression);
```

Note: Notice the semi – colon(“;”) in the end of loop.

Flow Diagram:



Example:

- C
- C++

```
// C++ program to illustrate do-while loop
#include <iostream>
using namespace std;

int main()
{
    int i = 2; // Initialization expression

    do
    {
        // loop body
        cout << "Hello World\n";

        // update expression
        i++;

    } while (i < 1); // test expression

    return 0;
}
```

Output:

```
Hello World
```

In the above program, the test condition ($i < 1$) evaluates to false. But still, as the loop is an exit – controlled the loop body will execute once.

9.4.What about an Infinite Loop?

An infinite loop (sometimes called an endless loop) is a piece of coding that lacks a functional exit so that it repeats indefinitely. An infinite loop occurs when a condition is always evaluated to be true. Usually, this is an error.

```
// C++ program to demonstrate infinite loops
// using for and while
// Uncomment the sections to see the output

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

    // This is an infinite for loop as the condition
    // expression is blank
    for ( ; ; )
    {
        cout << "This loop will run forever.\n";
    }

    // This is an infinite for loop as the condition
    // given in while loop will keep repeating infinitely
    /*
    while (i != 0)
    {
        i-- ;
        cout << "This loop will run forever.\n";
    }
    */

    // This is an infinite for loop as the condition
    // given in while loop is "true"
    /*
    while (true)
    {
        cout << "This loop will run forever.\n";
    }
    */
}
```

Output: This loop will run forever.
This loop will run forever.

Important Points:

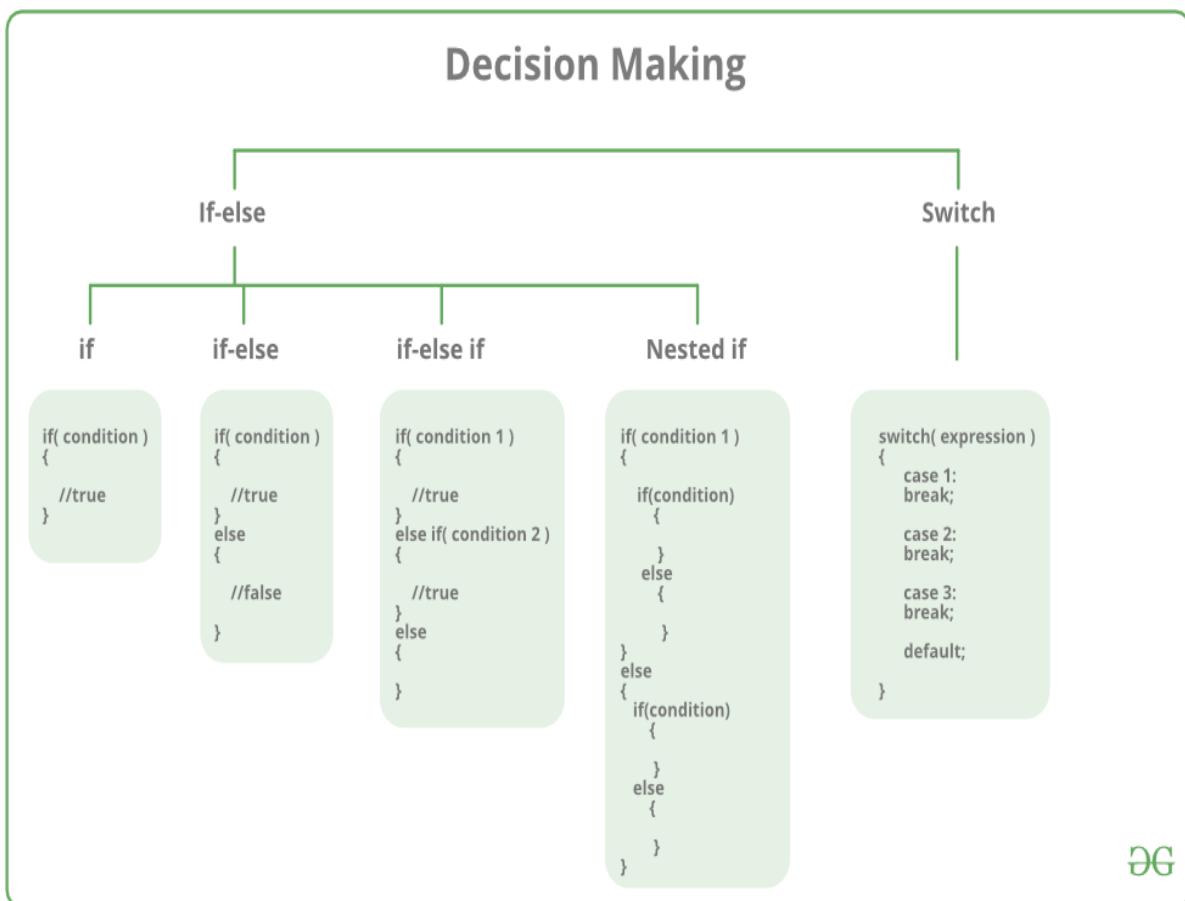
- Use for loop when a number of iterations are known beforehand, i.e. the number of times the loop body is needed to be executed is known.
- Use while loops, where an exact number of iterations is not known but the loop termination condition, is known.
- Use do while loop if the code needs to be executed at least once like in Menu-driven programs

[Range-based for loop in C++](#)

[for each loop in C++](#)

10. Decision Making in C / C++ (if , if..else, Nested if, if-else-if)

There come situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code. For example, in C if x occurs then execute y else execute z. There can also be multiple conditions like in C if x occurs then execute p, else if condition y occurs execute q, else execute r. This condition of C else-if is one of the many ways of importing multiple conditions.



Decision-making statements in programming languages decide the direction of the flow of program execution. Decision-making statements available in C or C++ are:

1. [if statement](#)
2. [if..else statements](#)
3. [nested if statements](#)
4. [if-else-if ladder](#)
5. [switch statements](#)

6. Jump Statements:

0. [break](#)
1. [continue](#)
2. [goto](#)
3. [return](#)

10.1. if statement in C/C++

if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

Syntax:

```
if(condition)
{
    // Statements to execute if
    // condition is true
}
```

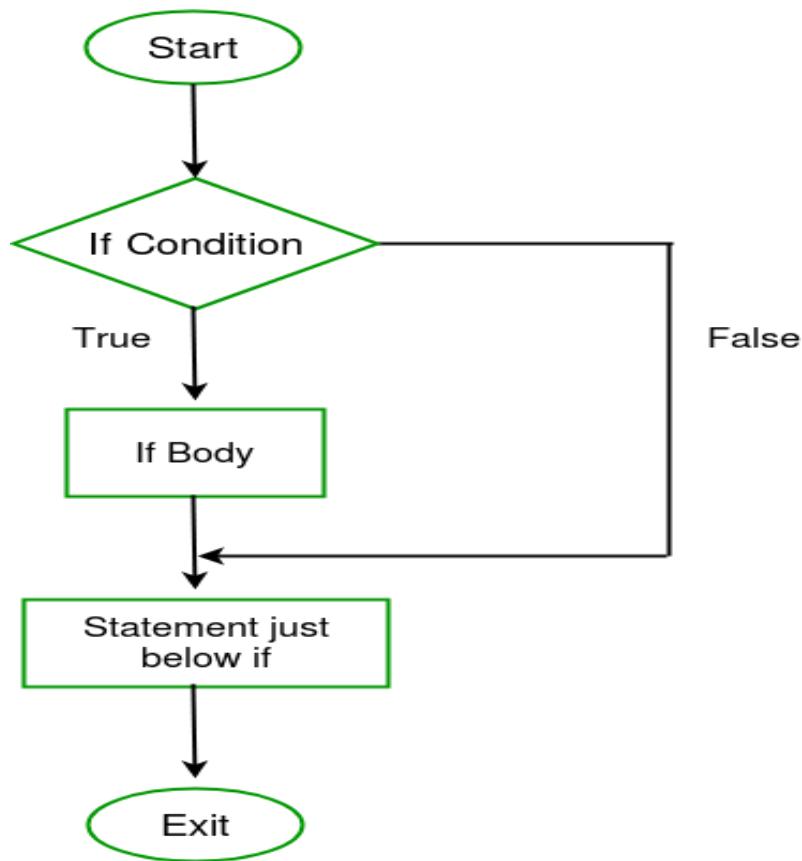
Here, the **condition** after evaluation will be either true or false. C if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not. If we do not provide the curly braces '{' and '}' after if(condition) then by default if statement will consider the first immediately below statement to be inside its block.

Example:

```
if(condition)
    statement1;
    statement2;

// Here if the condition is true, if block
// will consider only statement1 to be inside
// its block.
```

Flowchart



- C
- C++

```
// C++ program to illustrate If statement
#include<iostream>
using namespace std;

int main()
{
    int i = 10;
    if (i > 15)
    {
        cout<<"10 is less than 15";
    }
    cout<<"I am Not in if"; }
```

Output:

```
I am Not in if
```

As the condition present in the if statement is false. So, the block below the if statement is not executed.

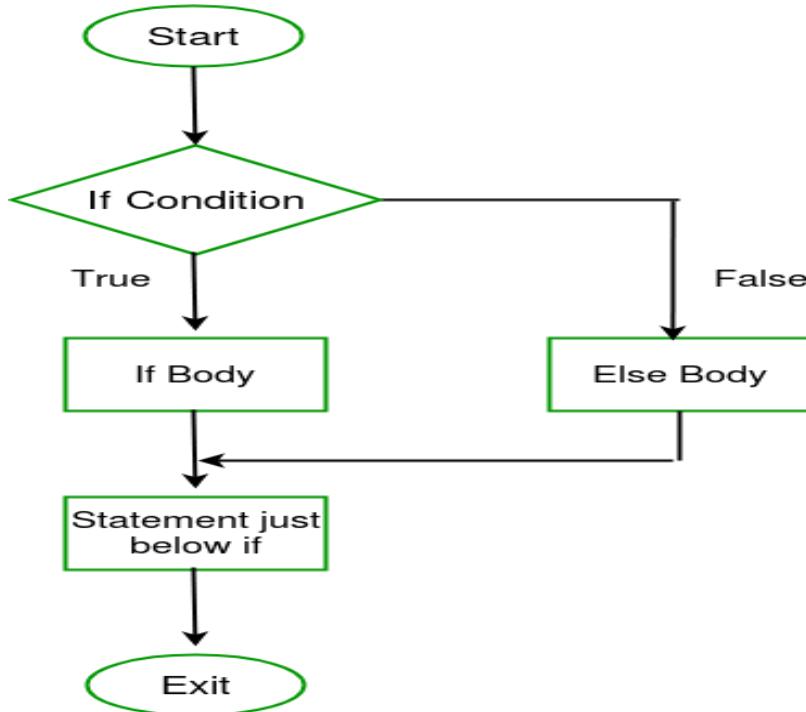
10.2.if-else in C/C++

The *if* statement alone tells us that if a condition is true it will execute a block of statements and if the condition is false it won't. But what if we want to do something else if the condition is false. Here comes the C *else* statement. We can use the *else* statement with *if* statement to execute a block of code when the condition is false.

Syntax:

```
if (condition)
{
    // Executes this block if
    // condition is true
}
else
{
    // Executes this block if
    // condition is false
}
```

Flowchart:



Example:

- C
- C++

```
// C++ program to illustrate if-else statement
#include<iostream>
using namespace std;

int main()
{
    int i = 20;
    if (i < 15)
        cout<<"i is smaller than 15";
    else
        cout<<"i is greater than 15";
    return 0;
}
```

Output:

i is greater than 15

The block of code following the `else` statement is executed as the condition present in the `if` statement is false.

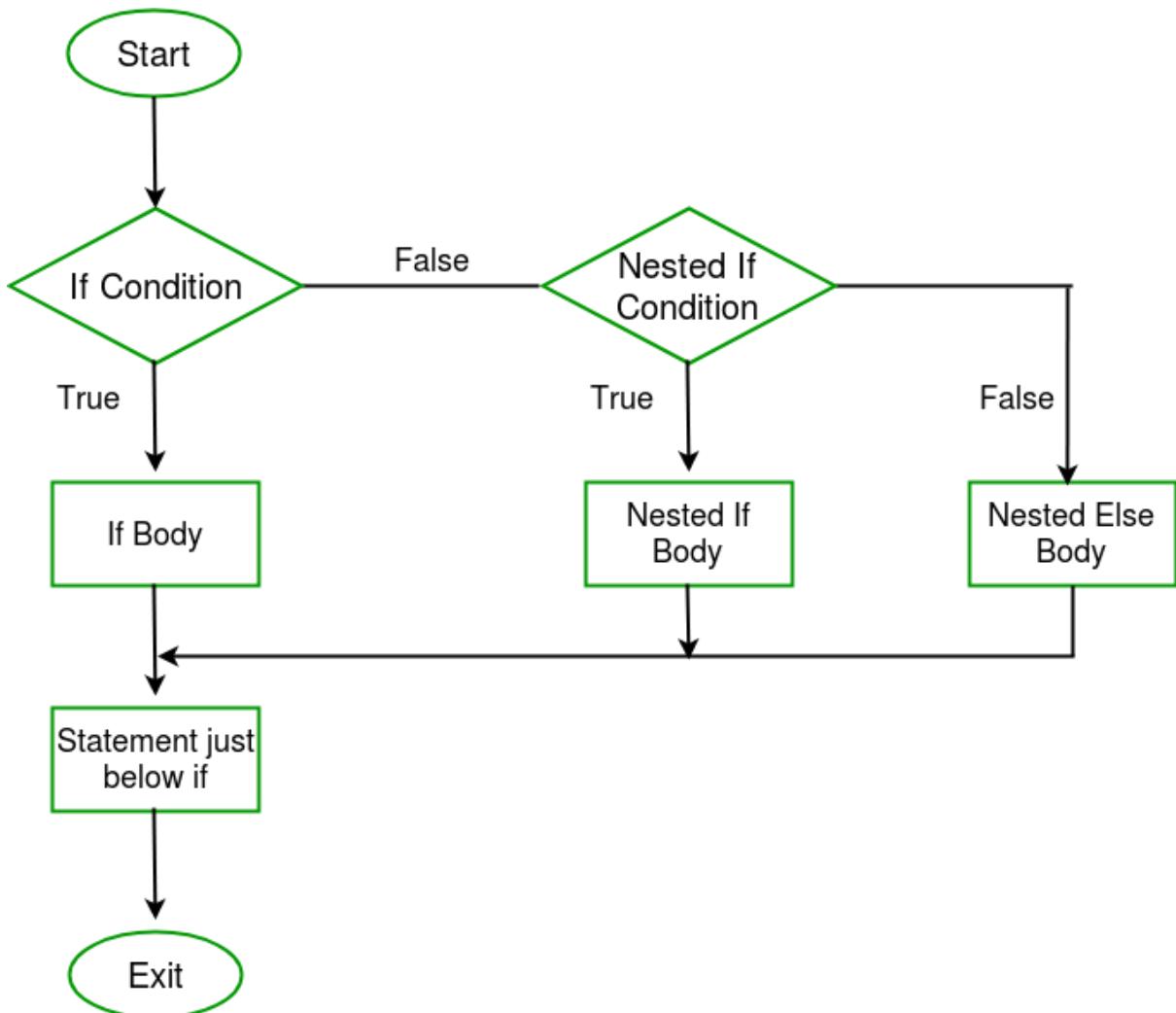
10.3.nested-if in C/C++

A nested if in C is an if statement that is the target of another if statement. Nested if statements mean an if statement inside another if statement. Yes, both C and C++ allow us to nested if statements within if statements, i.e, we can place an if statement inside another if statement.

Syntax:

```
if (condition1)
{
    // Executes when condition1 is true
    if (condition2)
    {
        // Executes when condition2 is true
    }
}
```

Flowchart



Example:

- C
- C++

```

// C++ program to illustrate nested-if statement

#include <iostream>
using namespace std;

int main()
{
    int i = 10;

    if (i == 10)

```

```
{  
    // First if statement  
    if (i < 15)  
        cout<<"i is smaller than 15\n";  
  
    // Nested - if statement  
    // Will only be executed if statement above  
    // is true  
    if (i < 12)  
        cout<<"i is smaller than 12 too\n";  
    else  
        cout<<"i is greater than 15";  
}  
  
return 0;  
}
```

Output:

```
i is smaller than 15  
i is smaller than 12 too
```

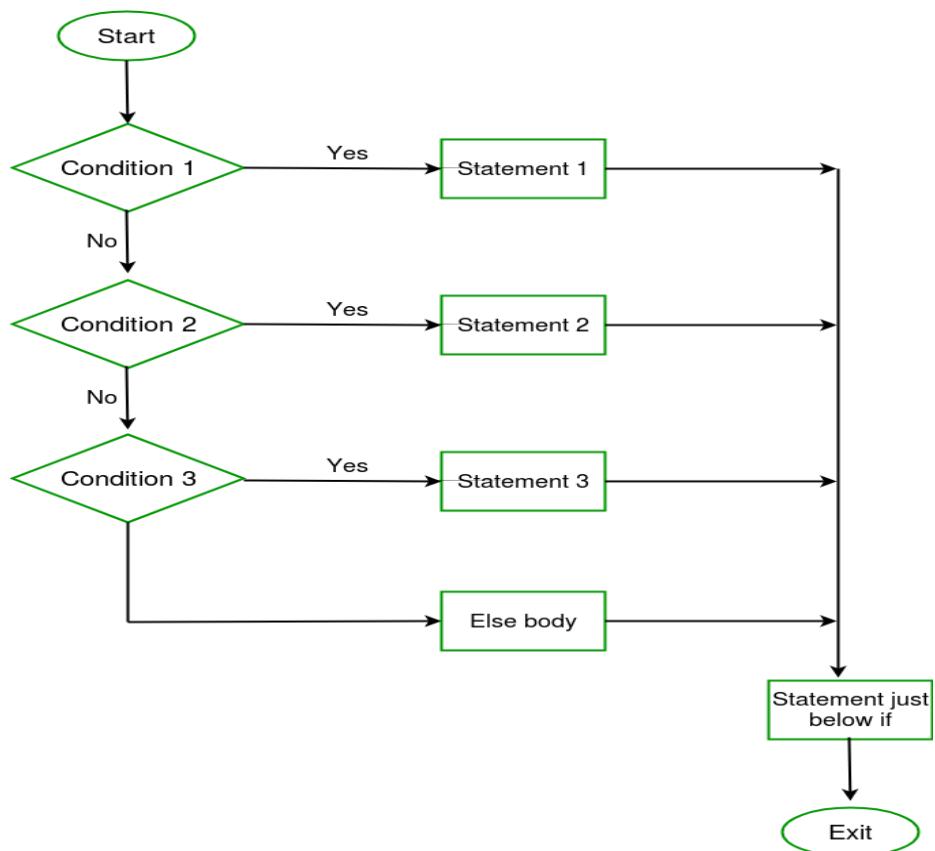
10.4.if-else-if ladder in C/C++

Here, a user can decide among multiple options. The C if statements are executed from the top down. As soon as one of the conditions controlling the if is true, the statement associated with that if is executed, and the rest of the C else-if ladder is bypassed. If none of the conditions are true, then the final else statement will be executed.

Syntax:

```
if (condition)
    statement;
else if (condition)
    statement;
.
.
else
    statement;
```

Flowchart:



Example:

- C

- C++

```
// C++ program to illustrate if-else-if ladder

#include<iostream>

using namespace std;

int main()
{
    int i = 20;

    if (i == 10)
        cout<<"i is 10";
    else if (i == 15)
        cout<<"i is 15";
    else if (i == 20)
        cout<<"i is 20";
    else
        cout<<"i is not present";
}
```

Output:

i is 20

10.5.Jump Statements in C/C++

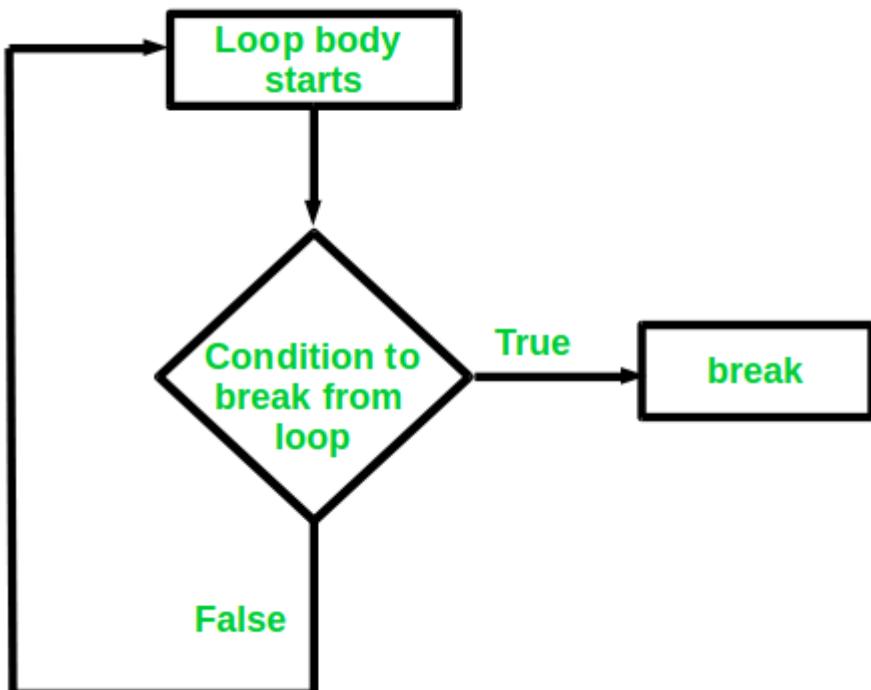
These statements are used in C or C++ for the unconditional flow of control throughout the functions in a program. They support four types of jump statements:

1. **C break:** This loop control statement is used to terminate the loop. As soon as the break statement is encountered from within a loop, the loop iterations stop there, and control returns from the loop immediately to the first statement after the loop.

Syntax:

```
break;
```

1. Basically, break statements are used in situations when we are not sure about the actual number of iterations for the loop or we want to terminate the loop based on some condition.



1. Example:

- C
- C++

```
// CPP program to illustrate  
// Linear Search  
#include <iostream>  
  
using namespace std;  
  
  
void findElement(int arr[], int size, int key)  
{  
    // loop to traverse array and search for key  
    for (int i = 0; i < size; i++) {
```

```

    if (arr[i] == key) {
        cout << "Element found at position: " << (i + 1);
        break;
    }
}

// Driver program to test above function
int main()
{
    int arr[] = { 1, 2, 3, 4, 5, 6 };
    int n = 6; // no of elements
    int key = 3; // key to be searched

    // Calling function to find the key
    findElement(arr, n, key);

    return 0;
}

```

1.
Output:

Element found at position: 3

1.

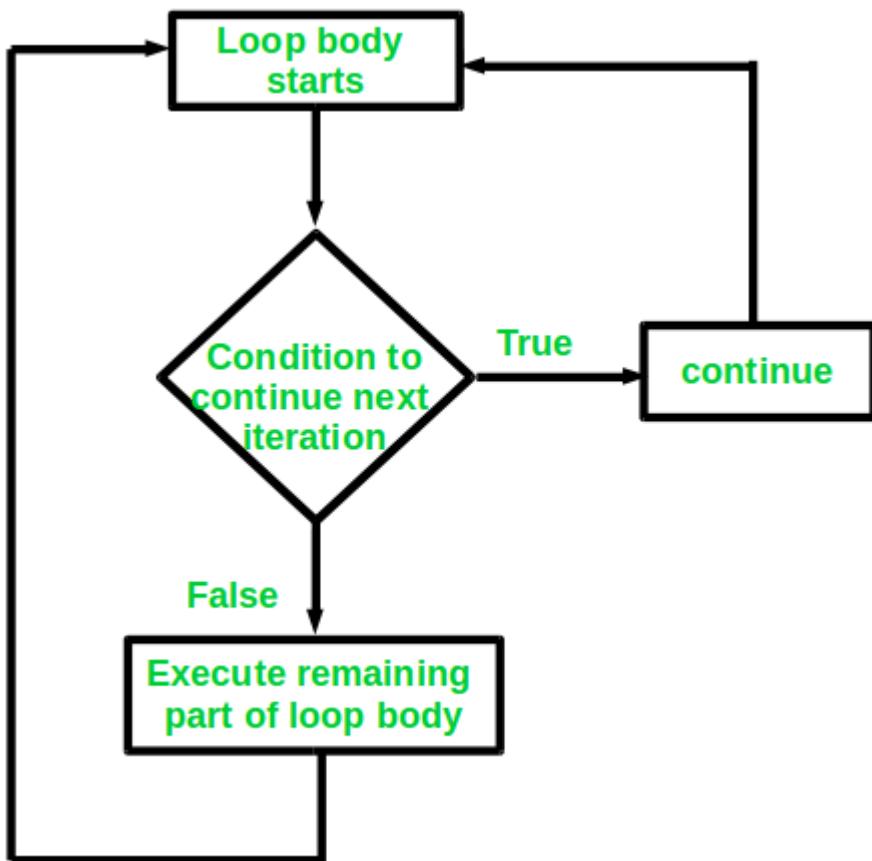
2. **C continues:** This loop control statement is just like the [break statement](#). The *continue* statement is opposite to that of the *break statement*, instead of terminating the loop, it forces to execute the next iteration of the loop.

As the name suggests the *continue* statement forces the loop to continue or execute the next iteration. When the *continue* statement is executed in the loop, the code inside the loop following the *continue* statement will be skipped and the next iteration of the loop will begin.

Syntax:

```
continue;
```

1.



1. Example:

- C
- C++

```
// C++ program to explain the use  
// of continue statement
```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    // loop from 1 to 10
```

```

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

    // If i is equals to 6,
    // continue to next iteration
    // without printing

    if (i == 6)

        continue;

    else

        // otherwise print the value of i
        cout << i << " ";

}

return 0;
}

```

1.
Output:

1 2 3 4 5 7 8 9 10

1.
If you create a variable in if-else in C/C++, it will be local to that if/else block only. You can use global variables inside the if/else block. If the name of the variable you created in if/else is as same as any global variable then priority will be given to `local variable`.

- C++
- C

```

#include<iostream>

using namespace std;

int main(){

    int gfg=0; // local variable for main

```

```

cout<<"Before if-else block "<<gfg<<endl;
if(1){
    int gfg = 100; // new local variable of if block
    cout<<"if block "<<gfg<<endl;
}
cout<<"After if block "<<gfg<<endl;
return 0;
}
/*
Before if-else block 0
if block 100
After if block 0
*/

```

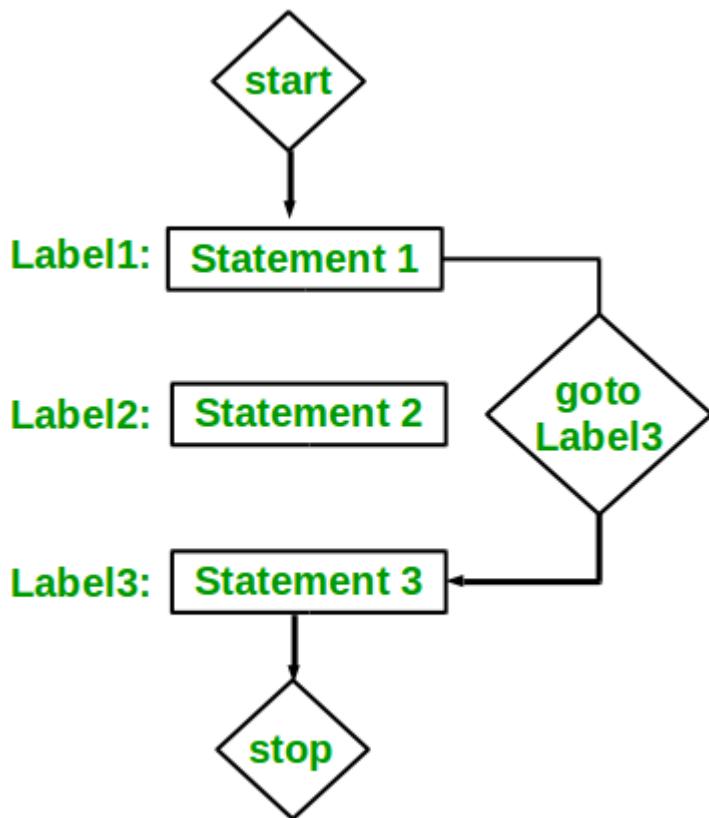
1. **C goto:** The goto statement in C/C++ also referred to as unconditional jump statement can be used to jump from one point to another within a function.

Syntax:

Syntax1		Syntax2

goto label;		label:
.		.
.		.
.		.
label:		goto label;

1. In the above syntax, the first line tells the compiler to go to or jump to the statement marked as a label. Here label is a user-defined identifier that indicates the target statement. The statement immediately followed after 'label:' is the destination statement. The 'label:' can also appear before the 'goto label;' statement in the above syntax.



1. Below are some examples of how to use goto statement:
Examples:

- C
- C++

```

// C++ program to print numbers
// from 1 to 10 using goto statement
#include <iostream>

using namespace std;

// function to print numbers from 1 to 10
void printNumbers()
{
    int n = 1;
label:
    cout << n << " ";
}

```

```

n++;

if (n <= 10)

    goto label;

}

// Driver program to test above function

int main()

{

    printNumbers();

    return 0;

}

```

1.
Output:

1 2 3 4 5 6 7 8 9 10

- 1.
2. **C return:** The return in C or C++ returns the flow of the execution to the function from where it is called. This statement does not mandatorily need any conditional statements. As soon as the statement is executed, the flow of the program stops immediately and return the control from where it was called. The return statement may or may not return anything for a void function, but for a non-void function, a return value is must be returned.

Syntax:

return[expression];

1. **Example:**

- C
- C++

```

// C++ code to illustrate return

// statement

#include <iostream>

```

```

using namespace std;

// non-void return type
// function to calculate sum

int SUM(int a, int b)
{
    int s1 = a + b;
    return s1;
}

// returns void
// function to print

void Print(int s2)
{
    cout << "The sum is " << s2;
    return;
}

int main()
{
    int num1 = 10;
    int num2 = 10;
    int sum_of = SUM(num1, num2);
    Print(sum_of);
    return 0;
}

```

1.
Output:

The sum is 20

11.Basic Input/Output

11.1.I/O Redirection in C++

- Difficulty Level : [Medium](#)
- Last Updated : 17 Mar, 2021

In C, we could use the function [freopen\(\)](#) to redirect an existing FILE pointer to another stream. The prototype for freopen() is given as

```
FILE * freopen ( const char * filename, const char * mode, FILE * stream );
```

For Example, to redirect the stdout to say a textfile, we could write

```
freopen ("text_file.txt", "w", stdout);
```

While this method is still supported in C++, this article discusses another way to redirect I/O streams.

C++ being an object-oriented programming language gives us the ability to not only define our own streams but also redirect standard streams. Thus, in C++, a stream is an object whose behavior is defined by a class. Thus, anything that behaves like a stream is also a stream.

Streams Objects in C++ are mainly of three types :

- **istream** : Stream object of this type can only perform input operations from the stream
- **ostream** : These objects can only be used for output operations.
- **iostream** : Can be used for both input and output operations

All these classes, as well as file stream classes, derived from the classes: ios and streambuf. Thus, ifstream and IO stream objects behave similarly.

All stream objects also have an associated data member of class streambuf.

Simply put streambuf object is the buffer for the stream. When we read data from a stream, we don't read it directly from the source, but instead, we read it from the buffer which is linked to the source. Similarly, output operations are first performed on the buffer, and then the buffer is flushed (written to the physical device) when needed.

C++ allows us to set the stream buffer for any stream. So the task of redirecting the stream simply reduces to changing the stream buffer associated with the stream. Thus, to redirect a Stream A to Stream B we need to do:-

1. Get the stream buffer of A and store it somewhere
2. Set the stream buffer of A to the stream buffer of B
3. If needed to reset the stream buffer of A to its previous stream buffer

We can use the function [ios::rdbuf\(\)](#) to perform two operations.

1) `stream_object.rdbuf()`: Returns pointer to the stream buffer of `stream_object`

2) `stream_object.rdbuf(streambuf * p)`: Sets the stream buffer to the object pointed by p

Here is an example program below to show the steps

- CPP

```
// Cpp program to redirect cout to a file

#include <fstream>
#include <iostream>
#include <string>

using namespace std;

int main()
{
    fstream file;
    file.open("cout.txt", ios::out);
    string line;

    // Backup streambuffers of cout
    streambuf* stream_buffer_cout = cout.rdbuf();
    streambuf* stream_buffer_cin = cin.rdbuf();

    // Get the streambuffer of the file
    streambuf* stream_buffer_file = file.rdbuf();

    // Redirect cout to file
    cout.rdbuf(stream_buffer_file);

    cout << "This line written to file" << endl;

    // Redirect cout back to screen
```

```
cout.rdbuf(stream_buffer_cout);

cout << "This line is written to screen" << endl;

file.close();

return 0;

}
```

Output:

This line is written to screen

Contents of file cout.txt:

This line written to file

Note:

The above steps can be condensed into a single step

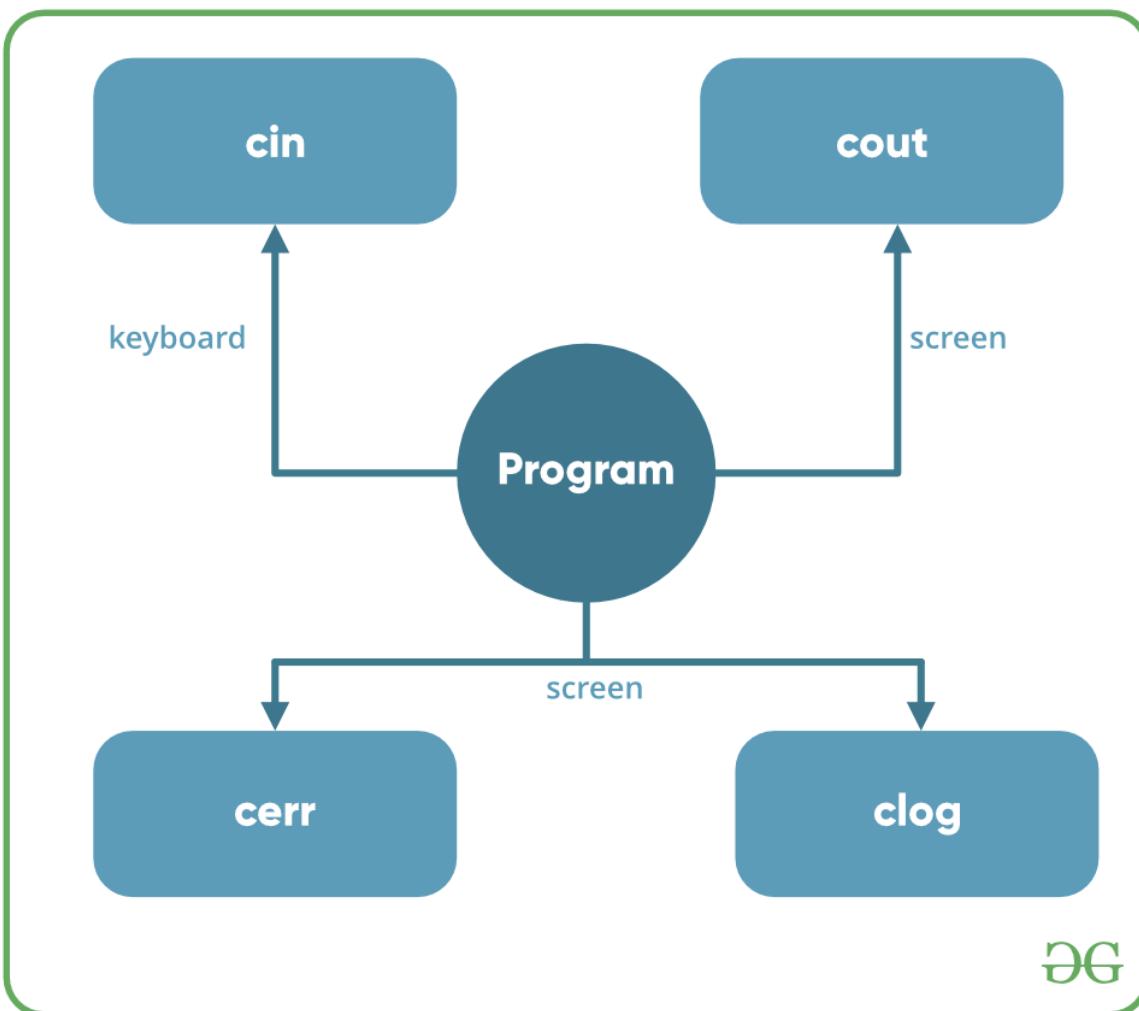
```
auto cout_buf = cout.rdbuf(file.rdbuf())

// sets couts streambuffer and returns the old
streambuffer back to cout_buf
```

11.2.Basic Input / Output in C++

C++ comes with libraries that provide us with many ways for performing input and output. In C++ input and output are performed in the form of a sequence of bytes or more commonly known as **streams**.

- **Input Stream:** If the direction of flow of bytes is from the device(for example, Keyboard) to the main memory then this process is called input.
- **Output Stream:** If the direction of flow of bytes is opposite, i.e. from main memory to device(display screen) then this process is called output.



Header files available in C++ for Input/Output operations are:

1. **iostream:** iostream stands for standard input-output stream. This header file contains definitions of objects like cin, cout, cerr, etc.
2. **iomanip:** iomanip stands for input-output manipulators. The methods declared in these files are used for manipulating streams. This file contains definitions of setw, setprecision, etc.

3. **fstream**: This header file mainly describes the file stream. This header file is used to handle the data being read from a file as input or data being written into the file as output.

The two instances **cout in C++** and **cin in C++** of iostream class are used very often for printing outputs and taking inputs respectively. These two are the most basic methods of taking input and printing output in C++. To use cin and cout in C++ one must include the header file *iostream* in the program.

This article mainly discusses the objects defined in the header file *iostream* like the cin and cout.

- **Standard output stream (cout)**: Usually the standard output device is the display screen. The C++ **cout** statement is the instance of the **ostream** class. It is used to produce output on the standard output device which is usually the display screen. The data needed to be displayed on the screen is inserted in the standard output stream (cout) using the insertion operator(**<<**).

- C++

```
#include <iostream>

using namespace std;

int main()
{
    char sample[] = "GeeksforGeeks";

    cout << sample << " - A computer science portal for geeks";

    return 0;
}
```

Output:

GeeksforGeeks - A computer science portal for geeks

In the above program, the insertion operator(**<<**) inserts the value of the string variable **sample** followed by the string “A computer science portal for geeks” in the standard output stream **cout** which is then displayed on the screen.

- **standard input stream (cin)**: Usually the input device in a computer is the keyboard. C++ **cin** statement is the instance of the class **istream** and is used to read input from the standard input device which is usually a keyboard.
The extraction operator(**>>**) is used along with the object **cin** for reading inputs. The extraction operator extracts the data from the object **cin** which is entered using the keyboard.

- C++

```

#include <iostream>
using namespace std;

int main()
{
    int age;

    cout << "Enter your age:" ;
    cin >> age;
    cout << "\nYour age is: " << age;

    return 0;
}

```

Input :

18

Output:

Enter your age:

Your age is: 18

The above program asks the user to input the age. The object `cin` is connected to the input device. The age entered by the user is extracted from `cin` using the extraction operator(`>>`) and the extracted data is then stored in the variable `age` present on the right side of the extraction operator.

- **Un-buffered standard error stream (cerr):** The C++ `cerr` is the standard error stream that is used to output the errors. This is also an instance of the `iostream` class. As `cerr` in C++ is un-buffered so it is used when one needs to display the error message immediately. It does not have any buffer to store the error message and display it later.
- The main difference between `cerr` and `cout` comes when you would like to redirect output using “`cout`” that gets redirected to file if you use “`cerr`” the error doesn’t get stored in file.(This is what un-buffered means ..It cant store the message)

- C++

```

#include <iostream>

using namespace std;

int main()
{
    cerr << "An error occurred";
    return 0;
}

```

Output:

An error occurred

- **buffered standard error stream (clog):** This is also an instance of ostream class and used to display errors but unlike cerr the error is first inserted into a buffer and is stored in the buffer until it is not fully filled. or the buffer is not explicitly flushed (using flush()). The error message will be displayed on the screen too.

- C++

```
#include <iostream>

using namespace std;

int main()
{
    clog << "An error occurred";

    return 0;
}
```

Output:

An error occurred

11.3.Clearing The Input Buffer In C/C++

What is a buffer?

A temporary storage area is called a buffer. All standard input and output devices contain an input and output buffer. In standard C/C++, streams are buffered, for example in the case of standard input, when we press the key on the keyboard, it isn't sent to your program, rather it is buffered by the operating system till the time is allotted to that program.

How does it affect Programming?

On various occasions, you may need to clear the unwanted buffer so as to get the next input in the desired container and not in the buffer of the previous variable. For example, in the case of C after encountering “scanf()”, if we need to input a character array or character, and in the case of C++, after encountering the “cin” statement, we require to input a character array or a string, we require to clear the input buffer or else the desired input is occupied by a buffer of the previous variable, not by the desired container. On pressing “Enter” (carriage return) on the output screen after the first input, as the buffer of the previous variable was the space for a new container(as we didn't clear it), the program skips the following input of the container.

In the case of [C++](#)

- C++

```
// C++ Code to explain why
// not clearing the input
// buffer causes undesired
// outputs
#include<iostream>
#include<vector>
using namespace std;

int main()
{
    int a;
    char ch[80];

    // Enter input from user
    // - 4 for example
    cin >> a;

    // Get input from user -
    // "GeeksforGeeks" for example
    cin.getline(ch,80);
```

```

// Prints 4
cout << a << endl;

// Printing string : This does
// not print string
cout << ch << endl;

return 0;
}

```

Input:

4

GeeksforGeeks

Output:

4

In both the above codes, the output is not printed as desired. The reason for this is an occupied Buffer. The “\n” character goes remains there in buffer and is read as the next input.

How can it be resolved?

In the case of C++ :

1. Using “ cin.ignore(numeric_limits::max(),'\\n'); ” :- Typing “cin.ignore(numeric_limits::max(),'\\n');” after the “cin” statement discards everything in the input stream including the newline.

- C++

```

// C++ Code to explain how
// "cin.ignore(numeric_limits
// <streamsize>::max(),'\\n');"
// discards the input buffer
#include<iostream>

// for <streamsize>
#include<iostream>

// for numeric_limits
#include<limits>
using namespace std;

int main()
{
    int a;
    char str[80];

```

```

// Enter input from user
// - 4 for example
cin >> a;

// discards the input buffer
cin.ignore(numeric_limits<streamsize>::max(), '\n');

// Get input from user -
// GeeksforGeeks for example
cin.getline(str, 80);

// Prints 4
cout << a << endl;

// Printing string : This
// will print string now
cout << str << endl;

return 0;
}

```

Input:

4

GeeksforGeeks

Output:

4

GeeksforGeeks

2. Using “`cin.sync()`”: Typing “`cin.sync()`” after the “`cin`” statement discards all that is left in the buffer. Though “`cin.sync()`” **does not work** in all implementations (According to C++11 and above standards).

- C++

```

// C++ Code to explain how " cin.sync();"
// discards the input buffer
#include<iostream>
#include<ios>
#include<limits>
using namespace std;

int main()
{

```

```

int a;
char str[80];

// Enter input from user
// - 4 for example
cin >> a;

// Discards the input buffer
cin.sync();

// Get input from user -
// GeeksforGeeks for example
cin.getline(str, 80);

// Prints 4
cout << a << endl;

// Printing string - this
// will print string now
cout << str << endl;

return 0;
}

```

Input:

4

GeeksforGeeks

Output:

4

3. Using “ cin >> ws ”: Typing “ cin>>ws ” after “ cin ” statement tells the compiler to ignore buffer and also to discard all the whitespaces before the actual content of string or character array.

- C++

```

// C++ Code to explain how "cin >> ws"
// discards the input buffer along with
// initial white spaces of string

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

int main()

```

```
{  
    int a;  
    string s;  
  
    // Enter input from user -  
    // 4 for example  
    cin >> a;  
  
    // Discards the input buffer and  
    // initial white spaces of string  
    cin >> ws;  
  
    // Get input from user -  
    // GeeksforGeeks for example  
    getline(cin, s);  
  
    // Prints 4 and GeeksforGeeks :  
    // will execute print a and s  
    cout << a << endl;  
    cout << s << endl;  
  
    return 0;  
}
```

Input:

```
4  
GeeksforGeeks
```

Output:

```
4  
GeeksforGeeks
```

12. Operators in C / C++

ds, we can say that an operator operates the operands. For example, ‘+’ is an operator used for addition, as shown below:

```
c = a + b;
```

Here, ‘+’ is the operator known as the addition operator and ‘a’ and ‘b’ are operands. The addition operator tells the compiler to add both of the operands ‘a’ and ‘b’.

The functionality of the C/C++ programming language is incomplete without the use of operators.

C/C++ has many built-in operators and can be classified into 6 types:

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

Operators in C	
Unary operator	Operator Type
	+ +, - - Unary operator
	+, -, *, /, % Arithmetic operator
Binary operator	<, <=, >, >=, ==, != Relational operator
	&&, , ! Logical operator
	&, , <<, >>, ~, ^ Bitwise operator
Ternary operator	=, +=, -=, *=, /=, %= Assignment operator
	?:
	Ternary or conditional operator

The above operators have been discussed in detail:

1. Arithmetic Operators:

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

a) Unary Operators: Operators that operate or work with a single operand are unary operators. For example: Increment(++) and Decrement(–) Operators

```
int val = 5;  
++val; // 6
```

b) Binary Operators: Operators that operate or work with two operands are binary operators. For example: Addition(+), Subtraction(-), multiplication(*), Division(/) operators

```
int a = 7;  
int b = 2;  
cout<<a+b; // 9
```

2. Relational Operators:

These are used for the comparison of the values of two operands. For example, checking if one operand is equal to the other operand or not, an operand is greater than the other operand or not, etc. Some of the relational operators are (==, >= , <=)(See [this](#) article for more reference).

```
int a = 3;  
int b = 5;  
a < b;  
// operator to check if a is smaller than b
```

3. Logical Operators:

Logical Operators are used to combining two or more conditions/constraints or to complement the evaluation of the original condition in consideration.

The result of the operation of a logical operator is a Boolean value either **true** or **false**.

For example, the **logical AND** represented as '**&&**' operator in C or C++ returns true when both the conditions under consideration are satisfied. Otherwise, it returns false. Therefore, a && b returns true when both a and b are true (i.e. non-zero)(See [this](#) article for more reference).

```
(4 != 5) && (4 < 5); // true
```

4. Bitwise Operators:

The Bitwise operators are used to perform bit-level operations on the operands. The operators are first converted to bit-level and then the calculation is performed on the operands. The mathematical operations such as addition, subtraction, multiplication, etc. can be performed at bit-level for faster processing. For example, the **bitwise AND** represented as **&** operator in C or C++ takes two numbers as operands and does AND on every bit of two numbers. The result of AND is 1 only if both bits are 1. (See [this](#) article for more reference).

```
int a = 5, b = 9;    // a = 5(00000101), b = 9(00001001)
cout << (a ^ b);    // 00001100
cout << (~a);      // 11111010
```

5. Assignment Operators:

Assignment operators are used to assigning value to a variable. The left side operand of the assignment operator is a variable and the right side operand of the assignment operator is a value. The value on the right side must be of the same data type as the variable on the left side otherwise the compiler will raise an error.

Different types of assignment operators are shown below:

a. “=”: This is the simplest assignment operator. This operator is used to assign the value on the right to the variable on the left.

For example:

```
a = 10;
b = 20;
ch = 'y';
```

b. “+=”: This operator is combination of ‘+’ and ‘=’ operators. This operator first adds the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.

For example:

(a += b) can be written as (a = a + b)

If initially value stored in a is 5. Then (a += 6) = 11.

c. “-=”: This operator is a combination of ‘-’ and ‘=’ operators. This operator first subtracts the value on the right from the current value of the variable on left and then assigns the result to the variable on the left.

For example:

(a -= b) can be written as (a = a - b)

If initially value stored in a is 8. Then (a -= 6) = 2.

d. “*=”: This operator is a combination of ‘*’ and ‘=’ operators. This operator first multiplies the current value of the variable on left to the value on the right and then assigns the result to the variable on the left.

For example:

(a *= b) can be written as (a = a * b)

If initially, the value stored in a is 5. Then (a *= 6) = 30.

e. “/=”: This operator is a combination of ‘/’ and ‘=’ operators. This operator first divides the current value of the variable on left by the value on the right and then assigns the result to the variable on the left.

For example:

(a /= b) can be written as (a = a / b)

If initially, the value stored in a is 6. Then (a /= 2) = 3.

6. Other Operators:

Apart from the above operators, there are some other operators available in C or C++ used to perform some specific tasks. Some of them are discussed here:

a. sizeof operator:

- sizeof is much used in the C/C++ programming language.
- It is a compile-time unary operator which can be used to compute the size of its operand.
- The result of sizeof is of the unsigned integral type which is usually denoted by size_t.
- Basically, the sizeof operator is used to compute the size of the variable.(See [this](#) article for reference)

b. Comma Operator:

- The comma operator (represented by the token) is a binary operator that evaluates its first operand and discards the result, it then evaluates the second operand and returns this value (and type).
- The comma operator has the lowest precedence of any C operator.
- Comma acts as both operator and separator. (See [this](#) article for reference)

c. Conditional Operator:

- The conditional operator is of the form *Expression1? Expression2: Expression3*.
- Here, Expression1 is the condition to be evaluated. If the condition(Expression1) is *True* then we will execute and return the result of Expression2 otherwise if the condition(Expression1) is *false* then we will execute and return the result of Expression3.
- We may replace the use of if..else statements with conditional operators. (See [this](#) article for reference)

d. . (dot) and -> (arrow)

- Member operators are used to reference individual members of classes, structures, and unions.
- The dot operator is applied to the actual object. The arrow operator is used with a pointer to an object.

e. Cast

- Casting operators convert one data type to another. For example, `int(2.2000)` would return 2.
- A cast is a special operator that forces one data type to be converted into another.
- The most general cast supported by most of the C++ compilers is as follows – **[(type) expression]**

f. &, *

- Pointer operator & returns the address of a variable. For example `&a;` will give actual address of the variable.

- Pointer operator * is pointer to a variable. For example *var; will point to a variable var.

Operator Precedence Chart

The below table describes the precedence order and associativity of operators in C / C++. The precedence of the operator decreases from top to bottom.

Precedence	Operator	Description	Associativity
	()	Parentheses (function call)	left-to-right
	[]	Brackets (array subscript)	
	.	Member selection via object name	
	->	Member selection via a pointer	
1	++/-	Postfix increment/decrement	
	++/-	Prefix increment/decrement	right-to-left
	+/-	Unary plus/minus	
	!~	Logical negation/bitwise complement	
	(type)	Cast (convert value to temporary value of type)	
2	*	Dereference	

Precedence	Operator	Description	Associativity
	&	Address (of operand)	
	sizeof	Determine size in bytes on this implementation	
3	* , / , %	Multiplication/division/modulus	left-to-right
4	+/-	Addition/subtraction	left-to-right
5	<< , >>	Bitwise shift left, Bitwise shift right	left-to-right
	< , <=	Relational less than/less than or equal to	left-to-right
6	> , >=	Relational greater than/greater than or equal to	left-to-right
7	== , !=	Relational is equal to/is not equal to	left-to-right
8	&	Bitwise AND	left-to-right
9	^	Bitwise exclusive OR	left-to-right
10		Bitwise inclusive OR	left-to-right
11	&&	Logical AND	left-to-right
12		Logical OR	left-to-right

Precedence	Operator	Description	Associativity
13	?:	Ternary conditional	right-to-left
	=	Assignment	right-to-left
	+= , -=	Addition/subtraction assignment	
	*= , /=	Multiplication/division assignment	
	%= , &=	Modulus/bitwise AND assignment	
	^= , =	Bitwise exclusive/inclusive OR assignment	
14	<>=	Bitwise shift left/right assignment	
15	,	expression separator	left-to-right

12.1 Unary operators in C/C++

Unary operator: are operators that act upon a single operand to produce a new value.

Types of unary operators:

1. unary minus(-)
2. increment(++)
3. decrement(--)
4. NOT(!)
5. Addressof operator(&)
6. sizeof()

1. unary minus

The minus operator changes the sign of its argument. A positive number becomes negative, and a negative number becomes positive.

2. int a = 10;
3. int b = -a; // b = -10

unary minus is different from subtraction operator, as subtraction requires two operands.

4. increment

It is used to increment the value of the variable by 1. The increment can be done in two ways:

0. prefix increment

In this method, the operator precedes the operand (e.g., `++a`). The value of operand will be altered *before* it is used.

1. int a = 1;
2. int b = ++a; // b = 2

3. postfix increment

In this method, the operator follows the operand (e.g., `a++`). The value operand will be altered *after* it is used.

4. int a = 1;
5. int b = a++; // b = 1
6. int c = a; // c = 2

5. decrement

It is used to decrement the value of the variable by 1. The decrement can be done in two ways:

0. prefix decrement

In this method, the operator precedes the operand (e.g., `-a`). The value of operand will be altered *before* it is used.

1. int a = 1;

2. int b = --a; // b = 0

3. postfix decrement

In this method, the operator follows the operand (e.g., a--). The value of operand will be altered *after* it is used.

4. int a = 1;

5. int b = a--; // b = 1

6. int c = a; // c = 0

C++ program for combination of prefix and postfix operations:

```
// C++ program to demonstrate working of unary increment
// and decrement operators
#include <iostream>
using namespace std;

int main()
{
    // Post increment
    int a = 1;
    cout << "a value: " << a << endl;
    int b = a++;
    cout << "b value after a++ : " << b << endl;
    cout << "a value after a++ : " << a << endl;

    // Pre increment
    a = 1;
    cout << "a value:" << a << endl;
    b = ++a;
    cout << "b value after ++a : " << b << endl;
    cout << "a value after ++a : " << a << endl;

    // Post decrement
    a = 5;
    cout << "a value before decrement: " << a << endl;
    b = a--;
    cout << "b value after a-- : " << b << endl;
    cout << "a value after a-- : " << a << endl;

    // Pre decrement
    a = 5;
    cout << "a value: " << a << endl;
    b = --a;
    cout << "b value after --a : " << b << endl;
    cout << "a value after --a : " << a << endl;

    return 0;
}
```

```
}
```

Output:

```
a value: 1  
b value after a++ : 1  
a value after a++ : 2  
a value:1  
b value after ++a : 2  
a value after ++a : 2  
a value before decrement: 5  
b value after a-- : 5  
a value after a-- : 4  
a value: 5  
b value after --a : 4  
a value after --a : 4
```

The above program shows how the postfix and prefix works.

6. **NOT(!):** It is used to reverse the logical state of its operand. If a condition is true, then Logical NOT operator will make it false.
7. If x is true, then !x is false
8. If x is false, then !x is true
9. **Addressof operator(&):** It gives an address of a variable. It is used to return the memory address of a variable. These addresses returned by the address-of operator are known as pointers because they “point” to the variable in memory.
10. & gives an address on variable n
11. int a;
12. int *ptr;
13. ptr = &a; // address of a is copied to the location ptr.
14. **sizeof():** This operator returns the size of its operand, in bytes. The sizeof operator always precedes its operand. The operand is an expression, or it may be a cast.

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

```
{  
    float n = 0;  
    cout << "size of n: " << sizeof(n);  
    return 1;  
}
```

15. **Output:**

16. size of n: 4

12.2.Pre-increment (or pre-decrement) With Reference to L-value in C++

Prerequisite: [Pre-increment and post-increment in C/C++](#)

In C++, pre-increment (or pre-decrement) can be used as [l-value](#), but post-increment (or post-decrement) can not be used as l-value.

For example, following program prints $a = 20$ ($++a$ is used as l-value)

- CPP

```
// CPP program to illustrate  
// Pre-increment (or pre-decrement)  
  
#include <cstdio>  
  
  
int main()  
{  
    int a = 10;  
  
    ++a = 20; // works  
    printf("a = %d", a);  
    printf("\n");  
    --a = 10;  
    printf("a = %d", a);  
    return 0;  
}
```

Output:

a = 20

a = 10

The above program works whereas the following program fails in compilation with error “*non-lvalue in assignment*” ($a++$ is used as l-value)

- CPP

```
// CPP program to illustrate
```

```

// Post-increment (or post-decrement)

#include <cstdio>

int main()
{
    int a = 10;
    a++ = 20; // error
    printf("a = %d", a);
    return 0;
}

```

Error:

```

prog.cpp: In function 'int main()':
prog.cpp:6:5: error: lvalue required as left operand of assignment
    a++ = 20; // error
    ^

```

How `++a` is Different From `a++` as lvalue?

It is because `++a` returns an *lvalue*, which is basically a reference to the variable to which we can further assign — just like an ordinary variable. It could also be assigned to a reference as follows:

```

int &ref = ++a; // valid
int &ref = a++; // invalid

```

Whereas if you recall how `a++` works, it doesn't immediately increment the value it holds. For clarity, you can think of it as getting incremented in the next statement. So what basically happens is that `a++` returns an *rvalue*, which is basically just a value like the value of an expression that is not stored. You can think of `a++ = 20;` as follows after being processed:

```
int a = 10;
```

```

// On compilation, a++ is replaced by the value of a which is an
rvalue:

```

```

10 = 20; // Invalid
// Value of a is incremented
a = a + 1;

```

That should help to understand why `a++ = 20;` won't work.

12.3.new and delete Operators in C++ For Dynamic Memory

Dynamic memory allocation in C/C++ refers to performing memory allocation manually by a programmer. Dynamically allocated memory is allocated on **Heap** and non-static and local variables get memory allocated on **Stack** (Refer to [Memory Layout C Programs](#) for details).

What are applications?

- One use of dynamically allocated memory is to allocate memory of variable size which is not possible with compiler allocated memory except for [variable-length arrays](#).
- The most important use is the flexibility provided to programmers. We are free to allocate and deallocate memory whenever we need it and whenever we don't need it anymore. There are many cases where this flexibility helps. Examples of such cases are [Linked List](#), [Tree](#), etc.

How is it different from memory allocated to normal variables?

For normal variables like “int a”, “char str[10]”, etc, memory is automatically allocated and deallocated. For dynamically allocated memory like “int *p = new int[10]”, it is the programmer's responsibility to deallocate memory when no longer needed. If the programmer doesn't deallocate memory, it causes a [memory leak](#) (memory is not deallocated until the program terminates).

How is memory allocated/deallocated in C++?

C uses the [malloc\(\)](#) and [calloc\(\)](#) function to allocate memory dynamically at run time and uses a [free\(\)](#) function to free dynamically allocated memory.

C++ supports these functions and also has two operators **new** and **delete** that perform the task of allocating and freeing the memory in a better and easier way.

new operator

The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.

Syntax to use new operator

```
pointer-variable = new data-type;
```

Here, pointer-variable is the pointer of type data-type. Data-type could be any built-in data type including array or any user-defined data type including structure and class.

Example:

```
// Pointer initialized with NULL
```

```
// Then request memory for the variable  
int *p = NULL;  
p = new int;
```

OR

```
// Combine declaration of pointer  
// and their assignment  
int *p = new int;
```

Initialize memory: We can also initialize the memory for built-in data types using a new operator. For custom data types a constructor is required (with the data type as input) for initializing the value. Here's an example of the initialization of both data types :

```
pointer-variable = new data-type(value);
```

Example:

```
int *p = new int(25);  
float *q = new float(75.25);  
  
// Custom data type  
struct cust  
{  
    int p;  
    cust(int q) : p(q) {}  
};  
  
// Works fine, doesn't require constructor  
cust* var1 = new cust();
```

OR

```
// Works fine, doesn't require constructor  
cust* var1 = new cust();  
  
// Notice error if you comment this line  
// cust* var = new cust(25)
```

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

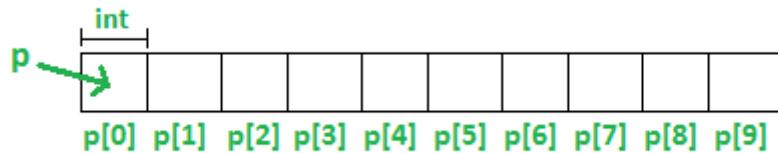
```
pointer-variable = new data-type[size];  
where size(a variable) specifies the number of elements in an array.
```

Example:

```
int *p = new int[10]
```

Dynamically allocates memory for 10 integers continuously of type int and returns a pointer to the first element of the sequence, which is assigned to(p(a

pointer). $p[0]$ refers to the first element, $p[1]$ refers to the second element, and so on.



Normal Array Declaration vs Using new

There is a difference between declaring a normal array and allocating a block of memory using new. The most important difference is, that normal arrays are deallocated by the compiler (If the array is local, then deallocated when the function returns or completes). However, dynamically allocated arrays always remain there until either they are deallocated by the programmer or the program terminates.

What if enough memory is not available during runtime?

If enough memory is not available in the heap to allocate, the new request indicates failure by throwing an exception of type `std::bad_alloc`, unless “`nothrow`” is used with the new operator, in which case it returns a NULL pointer (scroll to section “Exception handling of new operator” in [this article](#)). Therefore, it may be a good idea to check for the pointer variable produced by the new before using its program.

```
int *p = new(nothrow) int;  
if (!p)  
{  
    cout << "Memory allocation failed\n";  
}
```

delete operator

Since it is the programmer’s responsibility to deallocate dynamically allocated memory, programmers are provided delete operator in C++ language.

Syntax:

```
// Release memory pointed by pointer-variable
```

```
delete pointer-variable;
```

Here, pointer-variable is the pointer that points to the data object created by `new`.

Examples:

```
delete p;
```

```
delete q;
```

To free the dynamically allocated array pointed by pointer-variable, use the following form of *delete*:

```
// Release block of memory  
// pointed by pointer-variable  
delete[] pointer-variable;
```

Example:

```
// It will free the entire array  
// pointed by p.  
delete[] p;  
  
• CPP  
  
// C++ program to illustrate dynamic allocation  
// and deallocation of memory using new and delete  
  
#include <iostream>  
  
using namespace std;  
  
  
int main ()  
{  
    // Pointer initialization to null  
    int* p = NULL;  
  
    // Request memory for the variable  
    // using new operator  
    p = new(nothrow) int;  
    if (!p)  
        cout << "allocation of memory failed\n";  
    else  
    {  
        // Store value at allocated address  
        *p = 29;  
        cout << "Value of p: " << *p << endl;
```

```

}

// Request block of memory
// using new operator
float *r = new float(75.25);

cout << "Value of r: " << *r << endl;

// Request block of memory of size n
int n = 5;
int *q = new(nothrow) int[n];

if (!q)
    cout << "allocation of memory failed\n";
else
{
    for (int i = 0; i < n; i++)
        q[i] = i+1;

    cout << "Value store in block of memory: ";
    for (int i = 0; i < n; i++)
        cout << q[i] << " ";
}

// freed the allocated memory
delete p;
delete r;

// freed the block of allocated memory
delete[] q;

```

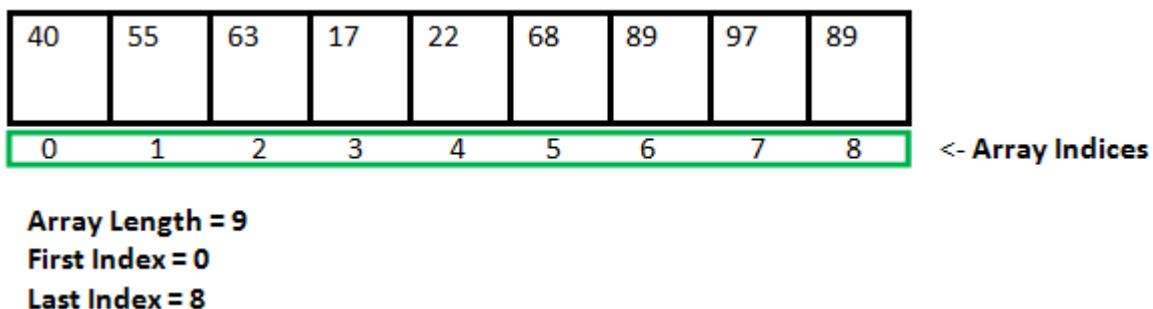
```
    return 0;  
}
```

Output:

```
Value of p: 29  
Value of r: 75.25  
Value store in block of memory: 1 2 3 4 5
```

13.Arrays in C/C++

An array in C/C++ or be it in any programming language is a collection of similar data items stored at contiguous memory locations and elements can be accessed randomly using indices of an array. They can be used to store collection of primitive data types such as int, float, double, char, etc of any particular type. To add to it, an array in C/C++ can store derived data types such as the structures, pointers etc. Given below is the picture representation of an array.



Why do we need arrays?

We can use normal variables (v_1, v_2, v_3, \dots) when we have a small number of objects, but if we want to store a large number of instances, it becomes difficult to manage them with normal variables. The idea of an array is to represent many instances in one variable.

Array declaration in C/C++:

The diagram is titled "Array Declaration in C". It shows five different ways to declare an integer array of size 3, each with its memory layout shown as a 3x3 grid of numbers. Yellow arrows point from the declarations to their respective memory representations.

- 1. `int a[3];` (Memory: 2192 | 451 | 13918)
- 2. `int a[3]={1, 2, 3};` (Memory: 1 | 2 | 3)
- 3. `int a[3]={ };` (Memory: 0 | 0 | 0)
- 4. `int a[3]={ [0...1]=3 };` (Memory: 3 | 3 | 0)
- 5. `int *a;` (Memory: 3 | 3)
- 6. `int* a;` (Memory: 3 | 3)
- 7. `int *a;` (Memory: 3 | 3)
- 8. `int *a;` (Memory: 3 | 3)

Note: In above image int a[3]={[0...1]=3}; this kind of declaration has been obsolete since GCC 2.5

There are various ways in which we can declare an array. It can be done by specifying its type and size, by initializing it or both.

Array declaration by specifying size

- C

```
// Array declaration by specifying size  
int arr1[10];  
  
// With recent C/C++ versions, we can also  
// declare an array of user specified size  
int n = 10;  
int arr2[n];
```

Array declaration by initializing elements

- C

```
// Array declaration by initializing elements  
int arr[] = { 10, 20, 30, 40 }  
  
// Compiler creates an array of size 4.  
// above is same as "int arr[4] = {10, 20, 30, 40}"
```

Array declaration by specifying size and initializing elements

- C

```
// Array declaration by specifying size and initializing  
// elements  
int arr[6] = { 10, 20, 30, 40 }  
  
// Compiler creates an array of size 6, initializes first  
// 4 elements as specified by user and rest two elements as  
// 0. above is same as "int arr[] = {10, 20, 30, 40, 0, 0}"
```

Advantages of an Array in C/C++:

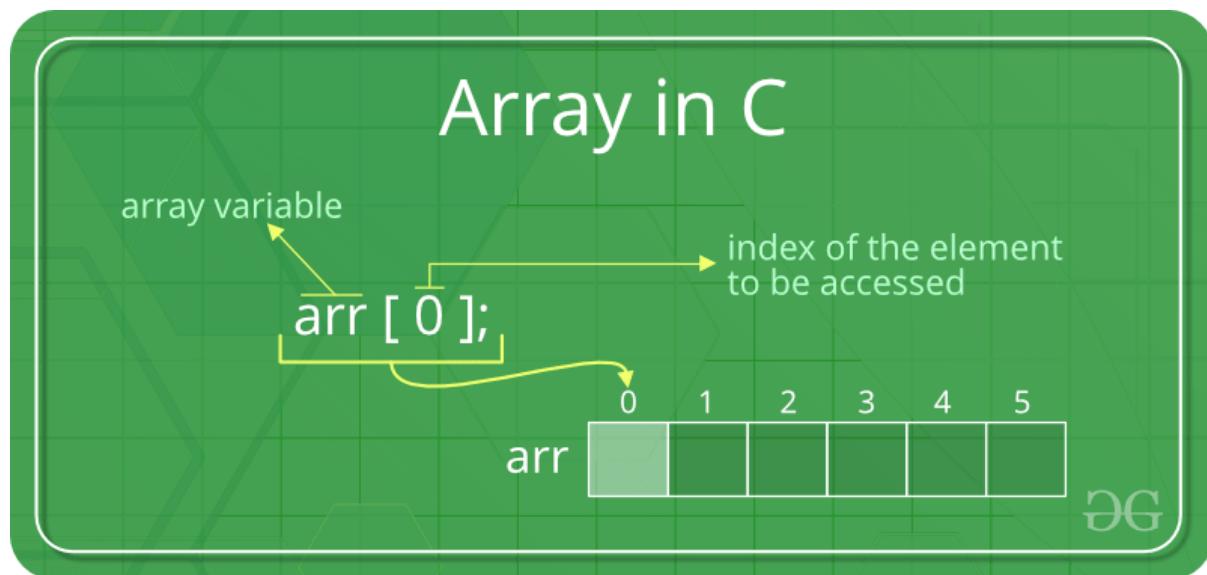
1. Random access of elements using array index.
2. Use of fewer line of code as it creates a single array of multiple elements.
3. Easy access to all the elements.
4. Traversal through the array becomes easy using a single loop.
5. Sorting becomes easy as it can be accomplished by writing fewer line of code.

Disadvantages of an Array in C/C++:

1. Allows a fixed number of elements to be entered which is decided at the time of declaration. Unlike a linked list, an array in C is not dynamic.
2. Insertion and deletion of elements can be costly since the elements are needed to be managed in accordance with the new memory allocation.

Facts about Array in C/C++:

- **Accessing Array Elements:**
Array elements are accessed by using an integer index. Array index starts with 0 and goes till size of array minus 1.
- Name of the array is also a pointer to the first element of array.



Example:

- C
- C++

```
#include <iostream>

using namespace std;

int main()
```

```

{
    int arr[5];
    arr[0] = 5;
    arr[2] = -10;

    // this is same as arr[1] = 2
    arr[3 / 2] = 2;
    arr[3] = arr[0];

    cout << arr[0] << " " << arr[1] << " " << arr[2] << " "
        << arr[3];

    return 0;
}

```

Output

5 2 -10 5

No Index Out of bound Checking:

There is no index out of bounds checking in C/C++, for example, the following program compiles fine but may produce unexpected output when run.

- C
- C++

```

// This C++ program compiles fine
// as index out of bound
// is not checked in C.

```

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```

int arr[2];

cout << arr[3] << " ";
cout << arr[-2] << " ";

return 0;
}

```

Output

-449684907 4195777

In C, it is not a compiler error to initialize an array with more elements than the specified size. For example, the below program compiles fine and shows just Warning.

- C

```

#include <stdio.h>

int main()
{

    // Array declaration by initializing it
    // with more elements than specified size.

    int arr[2] = { 10, 20, 30, 40, 50 };



    return 0;
}

```

Warnings:

```

prog.c: In function 'main':
prog.c:7:25: warning: excess elements in array initializer
    int arr[2] = { 10, 20, 30, 40, 50 };
                           ^
prog.c:7:25: note: (near initialization for 'arr')
prog.c:7:29: warning: excess elements in array initializer

```

```

int arr[2] = { 10, 20, 30, 40, 50 };
^
prog.c:7:29: note: (near initialization for 'arr')
prog.c:7:33: warning: excess elements in array initializer
int arr[2] = { 10, 20, 30, 40, 50 };
^
prog.c:7:33: note: (near initialization for 'arr')

```

- **Note:** The program won't compile in C++. If we save the above program as a .cpp, the program generates compiler error “error: too many initializers for ‘int [2]’”.

The elements are stored at contiguous memory locations

Example:

- C
- C++

```
// C++ program to demonstrate that array elements
// are stored contiguous locations
```

```
#include <iostream>

using namespace std;

int main()
{
    // an array of 10 integers.
    // If arr[0] is stored at
    // address x, then arr[1] is
    // stored at x + sizeof(int)
    // arr[2] is stored at x +
    // sizeof(int) + sizeof(int)
    // and so on.

    int arr[5], i;
```

```

cout << "Size of integer in this compiler is "
    << sizeof(int) << "\n";

for (i = 0; i < 5; i++)
    // The use of '&' before a variable name, yields
    // address of variable.
    cout << "Address arr[" << i << "] is " << &arr[i]
        << "\n";

return 0;
}

```

Output

Size of integer in this compiler is 4
 Address arr[0] is 0x7ffe75c32210
 Address arr[1] is 0x7ffe75c32214
 Address arr[2] is 0x7ffe75c32218
 Address arr[3] is 0x7ffe75c3221c
 Address arr[4] is 0x7ffe75c32220

Another way to traverse the array

- C++

```

#include<bits/stdc++.h>

using namespace std;

int main()
{
    int arr[6]={11,12,13,14,15,16};
    // Way 1
    for(int i=0;i<6;i++)
        cout<<arr[i]<<" ";
}

```

```

cout<<endl;

// Way 2

cout<<"By Other Method:"<<endl;

for(int i=0;i<6;i++)
    cout<<i[arr]<<" ";

cout<<endl;

return 0;
}

// Contributed by Akshay Pawar ( Username - akshaypawar4)

```

Output

11 12 13 14 15 16

By Other Method:

11 12 13 14 15 16

13.1.Array vs Pointers

Arrays and pointers are two different things (we can check by applying `sizeof`). The confusion happens because array name indicates the address of first element and arrays are always passed as pointers (even if we use square bracket). Please see [Difference between pointer and array in C?](#) for more details.

13.2.What is vector in C++?

A vector in C++ is a class in STL that represents an array. The advantages of vector over normal arrays are,

- We do not need pass size as an extra parameter when we declare a vector i.e, Vectors support dynamic sizes (we do not have to initially specify size of a vector). We can also resize a vector.
- Vectors have many in-built functions like, removing an element, etc.

13.3. Multidimensional Arrays in C / C++

Prerequisite: [Arrays in C/C++](#)

A multi-dimensional array can be termed as an array of arrays that stores homogeneous data in tabular form. Data in multidimensional arrays are stored in row-major order.

The **general form of declaring N-dimensional arrays** is:

```
data_type array_name[size1][size2]....[sizeN];
```

- **data_type**: Type of data to be stored in the array.
- **array_name**: Name of the array
- **size1, size2,... ,sizeN**: Sizes of the dimension

Examples:

```
Two dimensional array: int two_d[10][20];
```

```
Three dimensional array: int three_d[10][20][30];
```

Size of Multidimensional Arrays:

The total number of elements that can be stored in a multidimensional array can be calculated by multiplying the size of all the dimensions.

For example:

- The array **int x[10][20]** can store total $(10 \times 20) = 200$ elements.
- Similarly array **int x[5][10][20]** can store total $(5 \times 10 \times 20) = 1000$ elements.

Two-Dimensional Array

Two – dimensional array is the simplest form of a multidimensional array. We can see a two – dimensional array as an array of one-dimensional array for easier understanding.

The basic form of declaring a two-dimensional array of size x, y:

Syntax:

```
data_type array_name[x][y];
```

Here, **data_type** is the type of data to be stored.

We can declare a two-dimensional integer array say ‘x’ of size 10,20 as:

```
int x[10][20];
```

Elements in two-dimensional arrays are commonly referred to by $x[i][j]$ where i is the row number and ‘j’ is the column number.

A two – dimensional array can be seen as a table with ‘x’ rows and ‘y’ columns where the row number ranges from 0 to $(x-1)$ and the column number ranges from 0 to $(y-1)$. A two – dimensional array ‘x’ with 3 rows and 3 columns is shown below:

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

Initializing Two – Dimensional Arrays: There are various ways in which a Two-Dimensional array can be initialized.

First Method:

```
int x[3][4] = {0, 1 ,2 ,3 ,4 , 5 , 6 , 7 , 8 , 9 , 10 , 11}
```

The above array has 3 rows and 4 columns. The elements in the braces from left to right are stored in the table also from left to right. The elements will be filled in the array in order, the first 4 elements from the left in the first row, the next 4 elements in the second row, and so on.

Second Method:

```
int x[3][4] = {{0,1,2,3}, {4,5,6,7}, {8,9,10,11}};
```

Third Method:

```
int x[3][4];
for(int i = 0; i < 3; i++){
    for(int j = 0; j < 4; j++){
        cin >> x[i][j];
    }
}
```

Fourth Method(Dynamic Allocation):

```
int** x = new int*[3];
for(int i = 0; i < 3; i++){
    x[i] = new int[4];
    for(int j = 0; j < 4; j++){
        cin >> x[i][j];
    }
}
```

This type of initialization makes use of nested braces. Each set of inner braces represents one row. In the above example, there is a total of three rows so there are three sets of inner braces.

Accessing Elements of Two-Dimensional Arrays: Elements in Two-Dimensional arrays are accessed using the row indexes and column indexes.

Example:

```
int x[2][1];
```

The above example represents the element present in the third row and second column.

Note: In arrays, if the size of an array is N. Its index will be from 0 to N-1. Therefore, for row index 2 row number is $2+1 = 3$. To output all the elements of a Two-Dimensional array we can use nested for loops. We will require two 'for' loops. One to traverse the rows and another to traverse columns.

Example:

- CPP

```
// C++ Program to print the elements of a
// Two-Dimensional array

#include<iostream>

using namespace std;

int main()
{
    // an array with 3 rows and 2 columns.

    int x[3][2] = {{0,1}, {2,3}, {4,5}};

    // output each array element's value
    for (int i = 0; i < 3; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            cout << "Element at x[" << i
                << "][" << j << "]": ";
```

```

        cout << x[i][j]<<endl;
    }

}

return 0;
}

```

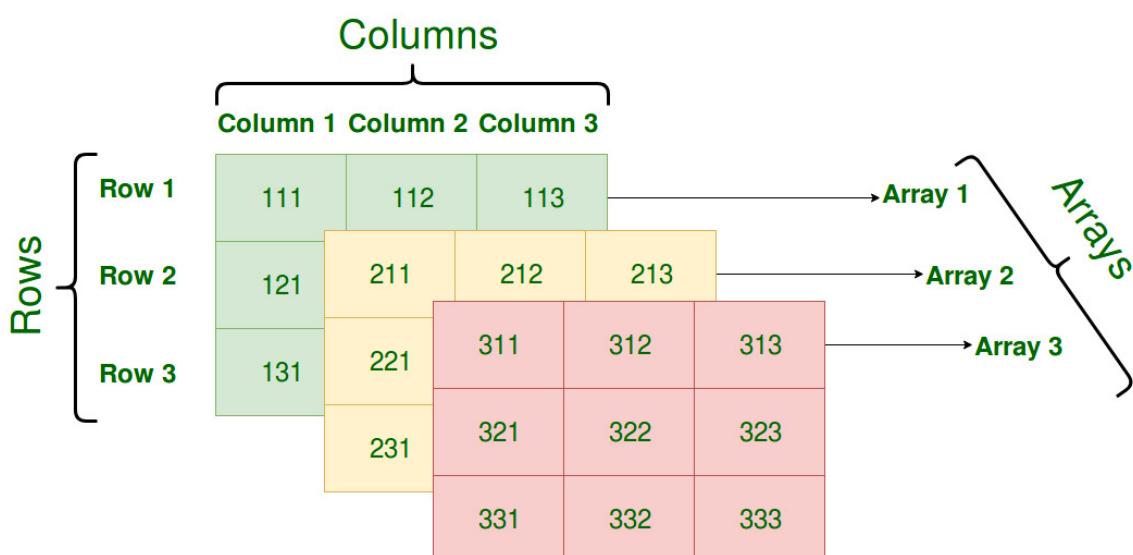
Output:

```

Element at x[0][0]: 0
Element at x[0][1]: 1
Element at x[1][0]: 2
Element at x[1][1]: 3
Element at x[2][0]: 4
Element at x[2][1]: 5

```

Three-Dimensional Array



Initializing Three-Dimensional Array: Initialization in a Three-Dimensional array is the same as that of Two-dimensional arrays. The difference is as the number of dimensions increases so the number of nested braces will also increase.

Method 1:

```

int x[2][3][4] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
                  11, 12, 13, 14, 15, 16, 17, 18, 19,

```

```
20, 21, 22, 23};
```

Method 2(Better):

```
int x[2][3][4] =  
{  
    { {0,1,2,3}, {4,5,6,7}, {8,9,10,11} },  
    { {12,13,14,15}, {16,17,18,19}, {20,21,22,23} }  
};
```

Accessing elements in Three-Dimensional Arrays: Accessing elements in Three-Dimensional Arrays is also similar to that of Two-Dimensional Arrays. The difference is we have to use three loops instead of two loops for one additional dimension in Three-dimensional Arrays.

- CPP

```
// C++ program to print elements of Three-Dimensional  
// Array  
  
#include <iostream>  
  
using namespace std;  
  
  
int main()  
{  
    // initializing the 3-dimensional array  
    int x[2][3][2] = { { { 0, 1 }, { 2, 3 }, { 4, 5 } },  
                      { { 6, 7 }, { 8, 9 }, { 10, 11 } } };  
  
    // output each element's value  
    for (int i = 0; i < 2; ++i) {  
        for (int j = 0; j < 3; ++j) {  
            for (int k = 0; k < 2; ++k) {  
                cout << "Element at x[" << i << "][" << j  
                     << "][" << k << "] = " << x[i][j][k]  
                     << endl;  
            }  
        }  
    }
```

```
    }  
    return 0;  
}
```

Output:

```
Element at x[0][0][0] = 0  
Element at x[0][0][1] = 1  
Element at x[0][1][0] = 2  
Element at x[0][1][1] = 3  
Element at x[0][2][0] = 4  
Element at x[0][2][1] = 5  
Element at x[1][0][0] = 6  
Element at x[1][0][1] = 7  
Element at x[1][1][0] = 8  
Element at x[1][1][1] = 9  
Element at x[1][2][0] = 10  
Element at x[1][2][1] = 11
```

In similar ways, we can create arrays with any number of dimensions. However, the complexity also increases as the number of dimensions increases. The most used multidimensional array is the Two-Dimensional Array.

13.4.How to print size of array parameter in C++?

How to compute the size of an array parameter in a function?
Consider below C++ program:

- CPP

```
// A C++ program to show that it is wrong to
// compute size of an array parameter in a function

#include <iostream>

using namespace std;

void findSize(int arr[])
{
    cout << sizeof(arr) << endl;
}

int main()
{
    int a[10];
    cout << sizeof(a) << " ";
    findSize(a);
    return 0;
}
```

Output:

40 8

The above output is for a machine where the size of an integer is 4 bytes and the size of a pointer is 8 bytes.

The **cout** statement inside main prints 40, and **cout** in **findSize** prints 8. The reason is, arrays are always passed pointers in functions, i.e., **findSize(int arr[])** and **findSize(int *arr)** mean exactly same thing. Therefore the **cout** statement inside **findSize()** prints the size of a pointer. See [this](#) and [this](#) for

details.

How to find the size of an array in function?

We can pass a 'reference to the array'.

- CPP

```
// A C++ program to show that we can use reference to
// find size of array

#include <iostream>

using namespace std;

void findSize(int (&arr)[10])
{
    cout << sizeof(arr) << endl;
}

int main()
{
    int a[10];
    cout << sizeof(a) << " ";
    findSize(a);
    return 0;
}
```

Output:

40 40

The above program doesn't look good as we have a hardcoded size of the array parameter. We can do it better using [templates in C++](#).

- CPP

```
// A C++ program to show that we use template and
// reference to find size of integer array parameter

#include <iostream>
```

```

using namespace std;

template <size_t n>
void findSize(int (&arr)[n])
{
    cout << sizeof(int) * n << endl;
}

int main()
{
    int a[10];
    cout << sizeof(a) << " ";
    findSize(a);
    return 0;
}

```

Output:

40 40

We can make a generic function as well:

- CPP

```

// A C++ program to show that we use template and
// reference to find size of any type array parameter
#include <iostream>

using namespace std;

template <typename T, size_t n>
void findSize(T (&arr)[n])
{
    cout << sizeof(T) * n << endl;
}

```

```

}

int main()
{
    int a[10];
    cout << sizeof(a) << " ";
    findSize(a);

    float f[20];
    cout << sizeof(f) << " ";
    findSize(f);

    return 0;
}

```

Output:

```

40 40
80 80

```

Now the next step is to print the size of a dynamically allocated array. It's your task man! I'm giving you a hint.

- CPP

```

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

int main()
{
    int *arr = (int*)malloc(sizeof(int) * 20);
    return 0;
}

```

14.Strings

14.1.std::string class in C++

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

String vs Character Array

String

A string is a **class that defines objects** that be represented as a stream of characters.

In the case of strings, memory is **allocated dynamically**. More memory can be allocated at run time on demand. As no memory is preallocated, **no memory is wasted**.

As strings are represented as objects, **no array decay** occurs.

Strings are slower when compared to implementation than character array.

String class defines **a number of functionalities** that allow manifold operations on strings.

Char Array

A character array is simply an **array of characters** that can be terminated by a null character.

The size of the character array has to be **allocated statically**, more memory cannot be allocated at run time if required. Unused allocated **memory is also wasted**

There is a **threat of array decay** in the case of the character array.

Implementation of **character array is faster** than std:: string.

Character arrays **do not offer many inbuilt functions** to manipulate strings.

Operations on Strings

1) Input Functions

Function Definition

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

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

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

Example:

- CPP

```
// C++ Program to demonstrate the working of  
// getline(), push_back() and pop_back()  
  
#include <iostream>  
  
#include <string> // for string class  
  
using namespace std;  
  
  
// Driver Code  
  
int main()  
{  
    // Declaring string  
    string str;  
  
    // Taking string input using getline()  
    getline(cin, str);
```

```

// Displaying string

cout << "The initial string is : ";

cout << str << endl;

// Inserting a character

str.push_back('s');

// Displaying string

cout << "The string after push_back operation is : ";

cout << str << endl;

// Deleting a character

str.pop_back();

// Displaying string

cout << "The string after pop_back operation is : ";

cout << str << endl;

return 0;
}

```

Output:

The initial string is : geeksforgeek

The string after push_back operation is : geeksforgeeks

The string after pop_back operation is : geeksforgeek

2) Capacity Functions

Function	Definition
----------	------------

capacity()	This function returns the capacity allocated to the string, which can be equal to or more than the size of the string. Additional space is
------------	--

Function	Definition
	allocated so that when the new characters are added to the string, the operations can be done efficiently.
<u>resize()</u>	This function changes the size of the string, the size can be increased or decreased.
length()	This function finds the length of the string.
shrink_to_fit()	This function decreases the capacity of the string and makes it equal to the minimum capacity of the string. This operation is useful to save additional memory if we are sure that no further addition of characters has to be made.

Example:

- CPP

```
// C++ Program to demonstrate the working of
// capacity(), resize() and shrink_to_fit()

#include <iostream>
#include <string> // for string class
using namespace std;

// Driver Code

int main()
{
    // Initializing string
    string str = "geeksforgeeks is for geeks";

    // Displaying string
    cout << "The initial string is : ";
}
```

```

cout << str << endl;

// Resizing string using resize()
str.resize(13);

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

// Displaying capacity of string
cout << "The capacity of string is : ";
cout << str.capacity() << endl;

// Displaying length of the string
cout << "The length of the string is :" << str.length()
    << endl;

// Decreasing the capacity of string
// using shrink_to_fit()
str.shrink_to_fit();

// Displaying string
cout << "The new capacity after shrinking is : ";
cout << str.capacity() << endl;

return 0;
}

```

Output

The initial string is : geeksforgeeks is for geeks
The string after resize operation is : geeksforgeeks

```
The capacity of string is : 26
The length of the string is :13
The new capacity after shrinking is : 15
```

3) Iterator Functions

Function Definition

begin() This function returns an iterator to the beginning of the string.

end() This function returns an iterator to the end of the string.

rbegin() This function returns a reverse iterator pointing at the end of the string.

rend() This function returns a reverse iterator pointing at beginning of the string.

Example:

- CPP

```
// C++ Program to demonstrate the working of
// begin(), end(), rbegin(), rend()

#include <iostream>
#include <string> // for string class
using namespace std;

// Driver Code
int main()
{
    // Initializing string
    string str = "geeksforgeeks";
```

```

// Declaring iterator

std::string::iterator it;

// Declaring reverse iterator

std::string::reverse_iterator it1;

// Displaying string

cout << "The string using forward iterators is : ";

for (it = str.begin(); it != str.end(); it++)

    cout << *it;

cout << endl;

// Displaying reverse string

cout << "The reverse string using reverse iterators is "

": ";

for (it1 = str.rbegin(); it1 != str.rend(); it1++)

    cout << *it1;

cout << endl;

return 0;

}

```

Output

The string using forward iterators is : geeksforgeeks

The reverse string using reverse iterators is : skeegrofskeeg

4) Manipulating Functions:

Function	Definition
copy("char array", len, pos)	This function copies the substring in the target character array mentioned in its arguments. It takes 3 arguments, target char

Function	Definition
	array, length to be copied, and starting position in the string to start copying.
swap()	This function swaps one string with other.

Example:

- CPP

```
// C++ Program to demonstrate the working of

// copy() and swap()

#include <iostream>

#include <string> // for string class

using namespace std;

// Driver Code

int main()
{
    // Initializing 1st string
    string str1 = "geeksforgeeks is for geeks";

    // Declaring 2nd string
    string str2 = "geeksforgeeks rocks";

    // Declaring character array
    char ch[80];

    // using copy() to copy elements into char array
    // copies "geeksforgeeks"
    str1.copy(ch, 13, 0);
```

```

// Displaying char array

cout << "The new copied character array is : ";

cout << ch << endl;

// Displaying strings before swapping

cout << "The 1st string before swapping is : ";

cout << str1 << endl;

cout << "The 2nd string before swapping is : ";

cout << str2 << endl;

// using swap() to swap string content

str1.swap(str2);

// Displaying strings after swapping

cout << "The 1st string after swapping is : ";

cout << str1 << endl;

cout << "The 2nd string after swapping is : ";

cout << str2 << endl;

return 0;
}

```

Output

The new copied character array is : geeksforgeeks
 The 1st string before swapping is : geeksforgeeks is for geeks
 The 2nd string before swapping is : geeksforgeeks rocks
 The 1st string after swapping is : geeksforgeeks rocks
 The 2nd string after swapping is : geeksforgeeks is for geeks

14.2.C++ string class and its applications

In C++ we can store string by one of the two ways –

1. [C style strings](#)
2. string class (discussed in this post)

In this post, the second method is discussed. string class is part of C++ library that supports a lot much functionality over C style strings.

C++ string class internally uses char array to store character but all memory management, allocation, and null termination is handled by string class itself that is why it is easy to use. The length of the C++ string can be changed at runtime because of dynamic allocation of memory similar to vectors. As string class is a container class, we can iterate over all its characters using an iterator similar to other containers like vector, set and maps, but generally, we use a simple for loop for iterating over the characters and index them using the [] operator.

C++ string class has a lot of functions to handle string easily. Most useful of them are demonstrated in below code.

```
// C++ program to demonstrate various function string class
#include <bits/stdc++.h>
using namespace std;

int main()
{
    // various constructor of string class

    // initialization by raw string
    string str1("first string");

    // initialization by another string
    string str2(str1);

    // initialization by character with number of occurrence
    string str3(5, '#');
```

```

// initialization by part of another string
string str4(str1, 6, 6); //      from 6th index (second parameter)
                           // 6 characters (third parameter)

// initialization by part of another string : iterator version
string str5(str2.begin(), str2.begin() + 5);

cout << str1 << endl;
cout << str2 << endl;
cout << str3 << endl;
cout << str4 << endl;
cout << str5 << endl;

// assignment operator
string str6 = str4;

// clear function deletes all character from string
str4.clear();

// both size() and length() return length of string and
// they work as synonyms
int len = str6.length(); // Same as "len = str6.size();"

cout << "Length of string is : " << len << endl;

// a particular character can be accessed using at /
// [] operator
char ch = str6.at(2); //  Same as "ch = str6[2];"

```

```

cout << "third character of string is : " << ch << endl;

// front return first character and back returns last character
// of string

char ch_f = str6.front(); // Same as "ch_f = str6[0];"
char ch_b = str6.back(); // Same as below
                        // "ch_b = str6[str6.length() - 1];"

cout << "First char is : " << ch_f << ", Last char is : "
    << ch_b << endl;

// c_str returns null terminated char array version of string
const char* charstr = str6.c_str();
printf("%s\n", charstr);

// append add the argument string at the end
str6.append(" extension");
// same as str6 += " extension"

// another version of append, which appends part of other
// string
str4.append(str6, 0, 6); // at 0th position 6 character

cout << str6 << endl;
cout << str4 << endl;

// find returns index where pattern is found.
// If pattern is not there it returns predefined

```

```

// constant npos whose value is -1

if (str6.find(str4) != string::npos)
    cout << "str4 found in str6 at " << str6.find(str4)
    << " pos" << endl;
else
    cout << "str4 not found in str6" << endl;

// substr(a, b) function returns a substring of b length
// starting from index a
cout << str6.substr(7, 3) << endl;

// if second argument is not passed, string till end is
// taken as substring
cout << str6.substr(7) << endl;

// erase(a, b) deletes b characters at index a
str6.erase(7, 4);
cout << str6 << endl;

// iterator version of erase
str6.erase(str6.begin() + 5, str6.end() - 3);
cout << str6 << endl;

str6 = "This is a examples";

// replace(a, b, str) replaces b characters from a index by str
str6.replace(2, 7, "ese are test");

cout << str6 << endl;

```

```
    return 0;  
}
```

Output :

```
first string  
first string  
#####  
string  
first  
Length of string is : 6  
third character of string is : r  
First char is : s, Last char is : g  
string  
string extension  
string  
str4 found in str6 at 0 pos  
ext  
extension  
string nsion  
strinon  
These are test examples
```

As seen in the above code, we can get the length of the string by size() as well as length() but length() is preferred for strings. We can concat a string to another string by += or by append(), but += is slightly slower than append() because each time + is called a new string (creation of new buffer) is made which is returned that is a bit overhead in case of many append operation.

Applications :

On basis of above string function some application are written below :

```
// C++ program to demonstrate uses of some string function  
#include <bits/stdc++.h>
```

```

using namespace std;

// this function returns floating point part of a number-string
string returnFloatingPart(string str)
{
    int pos = str.find(".");
    if (pos == string::npos)
        return "";
    else
        return str.substr(pos + 1);
}

// This function checks whether a string contains all digit or not
bool containsOnlyDigit(string str)
{
    int l = str.length();
    for (int i = 0; i < l; i++)
    {
        if (str.at(i) < '0' || str.at(i) > '9')
            return false;
    }
    // if we reach here all character are digits
    return true;
}

// this function replaces all single space by %20
// Used in URLs
string replaceBlankWith20(string str)
{
    string replaceby = "%20";

```

```

int n = 0;

// loop till all space are replaced
while ((n = str.find(" ", n)) != string::npos )
{
    str.replace(n, 1, replaceby);
    n += replaceby.length();
}

return str;
}

// driver function to check above methods
int main()
{
    string fnum = "23.342";
    cout << "Floating part is : " << returnFloatingPart(fnum)
        << endl;

    string num = "3452";
    if (containsOnlyDigit(num))
        cout << "string contains only digit" << endl;

    string urlex = "google com in";
    cout << replaceBlankWith20(urlex) << endl;

    return 0;
}

```

Output :

```

Floating part is : 342
string contains only digit

```

14.3.Raw String Literal in C++

A Literal is a constant variable whose value does not change during the lifetime of the program. Whereas, a raw string literal is a string in which the escape characters like '\n, \t, or \"' of C++ are not processed. Hence, a raw string literal that starts with R"(and ends in)".

The syntax for Raw string Literal:

```
R "delimiter( raw_characters )delimiter" // delimiter is the end of logical entity
```

Here, delimiter is optional and it can be a character except the backslash{ / }, whitespaces{ }, and parentheses { () }.

These raw string literals allow a series of characters by writing precisely its contents like raw character sequence.

Example:

Ordinary String Literal

```
"\\\"\\n"
```

Raw String Literal

```
\/- Delimiter
```

```
R"(\\\"\\n)"
```

```
/\-- Delimiter
```

Difference between an Ordinary String Literal and a Raw String Literal:

Ordinary String Literal

It does not need anything to be defined.

It does not allow/include nested characters.

It does not ignore any special meaning of character and implements their special characteristic.

Raw String Literal

It needs a defined line{ parentheses ()} to start with the prefix R.

It allows/includes nested character implementation.

It ignores all the special characters like \n and \t and treats them like normal text.

Example of Raw String Literal:

- CPP

```
// C++ program to demonstrate working of raw string literal
#include <iostream>
using namespace std;

// Driver Code
int main()
{
    // A Normal string
    string string1 = "Geeks.\nFor.\nGeeks.\n";

    // A Raw string
    string string2 = R"(Geeks.\nFor.\nGeeks.\n)";

    cout << string1 << endl;

    cout << string2 << endl;

    return 0;
}
```

Output

Geeks.
For.
Geeks.

Geeks.\nFor.\nGeeks.\n

14.4.Array of Strings in C++ – 5 Different Ways to Create

In C++, a string is usually just an array of (or a reference/points to) characters that ends with the NULL character '\0'. A string is a 1-dimensional array of characters and an array of strings is a 2-dimensional array of characters.

Below are the 5 different ways to create an Array of Strings in C++:

- Using Pointers
- Using 2-D Array
- Using the String Class
- Using the Vector Class
- Using the Array Class

1. Using Pointers

Pointers are the symbolic representation of an address. In simple words, a pointer is something that stores the address of a variable in it. In this method, an array of string literals is created by an array of pointers in which the character or string pointer points to the very first value of the created array and will always point to it until it is traversed.

Example:

- CPP

```
// C++ program to demonstrate array of strings using
// pointers character array

#include <iostream>
#include <stdio.h>

int main()
{
    // Initialize array of pointer
    const char* colour[4]
        = { "Blue", "Red", "Orange", "Yellow" };

    // Printing Strings stored in 2D array
    for (int i = 0; i < 4; i++)

```

```

        std::cout << colour[i] << "\n";

    return 0;
}

```

Output:

Blue

Red

Orange

Yellow

- The number of strings is fixed, but needn't be. The 4 may be omitted, and the compiler will compute the correct size.
- These strings are constants and their contents cannot be changed. Because string literals (literally, the quoted strings) exist in a read-only area of memory, we must specify "const" here to prevent unwanted accesses that may crash the program.

2. Using a 2D array

A 2-D array is the simplest form of a multidimensional array in which it stores the data in a tabular form. This method is useful when the length of all strings is known and a particular memory footprint is desired. Space for strings will be allocated in a single block

Example:

- CPP

```

// C++ program to demonstrate array of strings using
// 2D character array

#include <iostream>

int main()
{
    // Initialize 2D array
    char colour[4][10]
        = { "Blue", "Red", "Orange", "Yellow" };

```

```

// Printing Strings stored in 2D array

for (int i = 0; i < 4; i++)
    std::cout << colour[i] << "\n";

return 0;
}

```

Output:

Blue

Red

Orange

Yellow

- Both the number of strings and the size of strings are fixed. The 4, again, may be left out, and the appropriate size will be computed by the compiler. The second dimension, however, must be given (in this case, 10), so that the compiler can choose an appropriate memory layout.
- Each string can be modified but will take up the full space given by the second dimension. Each will be laid out next to each other in memory, and can't change size.
- Sometimes, control over the memory footprint is desirable, and this will allocate a region of memory with a fixed, regular layout.

3. Using the String class

The STL [string](#) or [string class](#) may be used to create an array of mutable strings. In this method, the size of the string is not fixed, and the strings can be changed which somehow makes it dynamic in nature nevertheless **std::string** can be used to create a string array using in-built functions.

Example:

- CPP

```

// C++ program to demonstrate array of strings using
// array of strings.

#include <iostream>
#include <string>

```

```

int main()
{
    // Initialize String Array
    std::string colour[4]
        = { "Blue", "Red", "Orange", "Yellow" };

    // Print Strings
    for (int i = 0; i < 4; i++)
        std::cout << colour[i] << "\n";
}

```

Output:

Blue
Red
Orange
Yellow

The array is of fixed size, but needn't be. Again, the 4 here may be omitted, and the compiler will determine the appropriate size of the array. The strings are also mutable, allowing them to be changed.

4. Using the vector class

A **vector** is a dynamic array that doubles its size whenever a new character is added that exceeds its limit. The STL container vector can be used to dynamically allocate an array that can vary in size.

This is only usable in C++, as C does not have classes. Note that the initializer-list syntax here requires a compiler that supports the 2011 C++ standard, and though it is quite likely your compiler does, it is something to be aware of.

- CPP

```

// C++ program to demonstrate array of strings using vector
#include <iostream>
#include <string>

```

```

#include <vector>

int main()
{
    // Declaring Vector of String type
    // Values can be added here using initializer-list
    // syntax

    std::vector<std::string> colour{ "Blue", "Red",
                                    "Orange" };

    // Strings can be added at any time with push_back
    colour.push_back("Yellow");

    // Print Strings stored in Vector
    for (int i = 0; i < colour.size(); i++)
        std::cout << colour[i] << "\n";
}

```

Output:

Blue
Red
Orange
Yellow

- Vectors are dynamic arrays, and allow you to add and remove items at any time.
- Any type or class may be used in vectors, but a given vector can only hold one type.

5. Using the Array Class

An array is a homogeneous mixture of data that is stored continuously in the memory space. The STL [container array](#) can be used to allocate a fixed-size array. It may be used very similarly to a vector, but the size is always fixed.

Example:

- C++

```

// C++ program to demonstrate array of string

// using STL array

#include <array>
#include <iostream>
#include <string>

int main()
{
    // Initialize array
    std::array<std::string, 4> colour{ "Blue", "Red",
                                      "Orange", "Yellow" };

    // Printing Strings stored in array
    for (int i = 0; i < 4; i++)
        std::cout << colour[i] << "\n";

    return 0;
}

```

Output

Blue

Red

Orange

Yellow

These are by no means the only ways to make a collection of strings. C++ offers several [container](#) classes, each of which has various tradeoffs and features, and all of them exist to fill requirements that you will have in your projects. Explore and have fun!

Conclusion: Out of all the methods, Vector seems to be the best way for creating an array of Strings in C++.

14.5.Tokenizing a string in C++

Tokenizing a string denotes splitting a string with respect to some delimiter(s). There are many ways to tokenize a string. In this article four of them are explained:

Using stringstream

A **stringstream** associates a string object with a stream allowing you to read from the string as if it were a stream.

Below is the C++ implementation :

- C++

```
// Tokenizing a string using stringstream

#include <bits/stdc++.h>

using namespace std;

int main()
{

    string line = "GeeksForGeeks is a must try";

    // Vector of string to save tokens
    vector <string> tokens;

    // stringstream class check1
    stringstream check1(line);

    string intermediate;

    // Tokenizing w.r.t. space ' '
    while(getline(check1, intermediate, ' '))
}
```

```

{
    tokens.push_back(intermediate);
}

// Printing the token vector

for(int i = 0; i < tokens.size(); i++)
    cout << tokens[i] << '\n';

}

```

Output

GeeksForGeeks

is

a

must

try

Using strtok()

// Splits str[] according to given delimiters.
 // and returns next token. It needs to be called
 // in a loop to get all tokens. It returns NULL
 // when there are no more tokens.

char * strtok(char str[], const char *delims);
 Below is the C++ implementation :

- C++

```

// C/C++ program for splitting a string

// using strtok()

#include <stdio.h>
#include <string.h>

int main()
{
    char str[] = "Geeks-for-Geeks";

```

```

// Returns first token

char *token = strtok(str, "-");

// Keep printing tokens while one of the
// delimiters present in str[].

while (token != NULL)

{
    printf("%s\n", token);

    token = strtok(NULL, "-");
}

return 0;
}

```

Output

Geeks

for

Geeks

Another Example of strtok() :

- C

```

// C code to demonstrate working of
// strtok

#include <string.h>
#include <stdio.h>

// Driver function

int main()
{

```

```

// Declaration of string

char gfg[100] = " Geeks - for - geeks - Contribute";

// Declaration of delimiter

const char s[4] = "-";

char* tok;

// Use of strtok

// get first token

tok = strtok(gfg, s);

// Checks for delimiter

while (tok != 0) {

    printf("%s\n", tok);

    // Use of strtok

    // go through other tokens

    tok = strtok(0, s);

}

return (0);
}

```

Output

Geeks
for
geeks
Contribute

Using strtok_r()

Just like strtok() function in C, **strtok_r()** does the same task of parsing a string into a sequence of tokens. strtok_r() is a reentrant version of strtok(). There are two ways we can call strtok_r()

```
// The third argument saveptr is a pointer to a char *
// variable that is used internally by strtok_r() in
// order to maintain context between successive calls
// that parse the same string.
char *strtok_r(char *str, const char *delim, char **saveptr);
```

Below is a simple C++ program to show the use of strtok_r() :

- C++

```
// C/C++ program to demonstrate working of strtok_r()
// by splitting string based on space character.

#include<stdio.h>
#include<string.h>

int main()
{
    char str[] = "Geeks for Geeks";
    char *token;
    char *rest = str;

    while ((token = strtok_r(rest, " ", &rest)))
        printf("%s\n", token);

    return(0);
}
```

Output

Geeks
for
Geeks

Using std::sregex_token_iterator

In this method the tokenization is done on the basis of regex matches. Better for use cases when multiple delimiters are needed.

Below is a simple C++ program to show the use of std::sregex_token_iterator:

- C++

```
// CPP program for above approach

#include <iostream>
#include <regex>
#include <string>
#include <vector>

/***
 * @brief Tokenize the given vector
 * according to the regex
 * and remove the empty tokens.
 *
 * @param str
 * @param re
 * @return std::vector<std::string>
 */

std::vector<std::string> tokenize(
    const std::string str,
    const std::regex re)
{
    std::sregex_token_iterator it{ str.begin(),
                                str.end(), re, -1 };
    std::vector<std::string> tokenized{ it, {} };

    // Additional check to remove empty strings
    tokenized.erase(
        std::remove_if(tokenized.begin(),
                      tokenized.end(),
```

```

    [](std::string const& s) {
        return s.size() == 0;
    }),
    tokenized.end());
}

return tokenized;
}

// Driver Code

int main()
{
    const std::string str = "Break string
                            a,spaces,and,commas";
    const std::regex re(R"([\s|,]+)");
}

// Function Call

const std::vector<std::string> tokenized =
    tokenize(str, re);

for (std::string token : tokenized)
    std::cout << token << std::endl;
return 0;
}

```

Output

Break

string

a

spaces

and

commas

14.6.strrchr() function in C/C++

strrchr() function in C/C++

strrchr() function

In C++, strrchr() is a predefined function used for string handling. cstring is the header file required for string functions.

This function returns a pointer to the last occurrence of a character in a string. The character whose last occurrence we want to find is passed as the second argument to the function and the string in which we have to find the character is passed as the first argument to the function.

Syntax

```
char *strrchr(const char *str, int c)
```

Here, str is the string and c is the character to be located. It is passed as its int promotion, but it is internally converted back to char.

Application

Given a string in C++, we need to find the last occurrence of a character, let's say 'a'.

Examples:

Input : string = 'This is a string'

Output :9

Input :string = 'My name is Ayush'

Output :12

Algorithm

1. Pass the given string in the strchr() function and mention the character you need to point to.
2. The function returns a value, print the value.

[Recommended: Please try your approach on {IDE} first, before moving on to the solution.](#)

- CPP

```
// C++ program to demonstrate working strrchr()  
#include <iostream>
```

```

#include <cstring>
using namespace std;

int main()
{
    char str[] = "This is a string";
    char * ch = strrchr(str, 'a');
    cout << ch - str + 1;
    return 0;
}

```

Output:

9

C Examples :

- C

```

// C code to demonstrate the working of
// strrchr()

```

```
#include <stdio.h>
```

```
#include <string.h>
```

```
// Driver function
```

```
int main()
```

```
{
```

```
// initializing variables
```

```
char st[] = "GeeksforGeeks";
```

```
char ch = 'e';
```

```
char* val;
```

```

// Use of strrchr()
// returns "ks"
val = strrchr(st, ch);

printf("String after last %c is :  %s \n", ch, val);

char ch2 = 'm';

// Use of strrchr()
// returns null
// test for null
val = strrchr(st, ch2);

printf("String after last %c is :  %s ", ch2, val);

return (0);
}

```

Output:

```

String after last e is :  eks
String after last m is :  (null)

```

Practical Application: Since it returns the entire string after the last occurrence of a particular character, it can be used to **extract the suffix of a string**. For e.g to know the entire leading zeroes in a denomination when we know the first number. This example is demonstrated below.

- C

```

// C code to demonstrate the application of
// strrchr()

```

```

#include <stdio.h>
#include <string.h>

// Driver function
int main()
{
    // initializing the denomination
    char denom[] = "Rs 10000000";

    // Printing original string
    printf("The original string is : %s", denom);

    // initializing the initial number
    char first = '1';
    char* entire;

    // Use of strrchr()
    // returns entire number
    entire = strrchr(denom, first);

    printf("\nThe denomination value is : %s ", entire);

    return (0);
}

```

Output:

```

The original string is : Rs 10000000
The denomination value is : 10000000

```

14.7. stringstream in C++ and its applications

A [stringstream](#) associates a string object with a stream allowing you to read from the string as if it were a stream (like cin). To use stringstream, we need to include **sstream** header file. The stringstream class is extremely useful in parsing input.

Basic methods are:

1. **clear()**- To clear the stream.
2. **str()**- To get and set string object whose content is present in the stream.
3. **operator <<**- Add a string to the stringstream object.
4. **operator >>**- Read something from the stringstream object.

Examples:

1. Count the number of words in a string-

Examples-

Input: Asipu Pawan Kumar

Output: 3

Input: Geeks For Geeks Ide

Output: 4

Below is the C++ program to implement the above approach-

- C++

```
// C++ program to count words in
// a string using stringstream.

#include <iostream>
#include <sstream>
#include<string>

using namespace std;

int countWords(string str)
{
    // Breaking input into word
    // using string stream

    // Used for breaking words
```

```

stringstream s(str);

// To store individual words
string word;

int count = 0;
while (s >> word)
    count++;
return count;
}

// Driver code
int main()
{
    string s = "geeks for geeks geeks "
               "contribution placements";
    cout << " Number of words are: " << countWords(s);
    return 0;
}

```

Output

Number of words are: 6

2. Print frequencies of individual words in a string- Examples-

Input: Geeks For Geeks Quiz Geeks Quiz Practice Practice

Output: For -> 1

Geeks -> 3

Practice -> 2

Quiz -> 2

Input: Word String Frequency String

Output: Frequency -> 1

String -> 2

Word -> 1

Below is the C++ program to implement the above approach-

- C++

```
// C++ program to demonstrate use
// of stringstream to count
// frequencies of words.

#include <bits/stdc++.h>
#include <iostream>
#include <sstream>
#include<string>
using namespace std;

void printFrequency(string st)
{
    // Each word it mapped to
    // it's frequency
    map<string, int> FW;

    // Used for breaking words
    stringstream ss(st);

    // To store individual words
    string Word;

    while (ss >> Word)
        FW[Word]++;
}

map<string, int>::iterator m;
for (m = FW.begin(); m != FW.end(); m++)
    cout << m->first << "-> "
        << m->second << "\n";
}
```

```
// Driver code  
int main()  
{  
    string s = "Geeks For Geeks Ide";  
    printFrequency(s);  
    return 0;  
}
```

Output

For-> 1

Geeks-> 2

Ide-> 1

3. [Removing spaces from a string using Stringstream](#)

4. [Converting Strings to Numbers in C/C++](#)

15. Functions in C/C++

A function is a set of statements that take inputs, do some specific computation and produces output.

The idea is to put some commonly or repeatedly done task together and make a function so that instead of writing the same code again and again for different inputs, we can call the function.

The general form of a function is:

```
return_type function_name([ arg1_type arg1_name, ... ]) { code }
```

Example:

Below is a simple C/C++ program to demonstrate functions.

- C
- C++

```
#include <iostream>

using namespace std;

int max(int x, int y)
{
    if (x > y)
        return x;
    else
        return y;
}

int main() {
    int a = 10, b = 20;

    // Calling above function to find max of 'a' and 'b'
    int m = max(a, b);
```

```

cout << "m is " << m;
return 0;
}

```

Output:

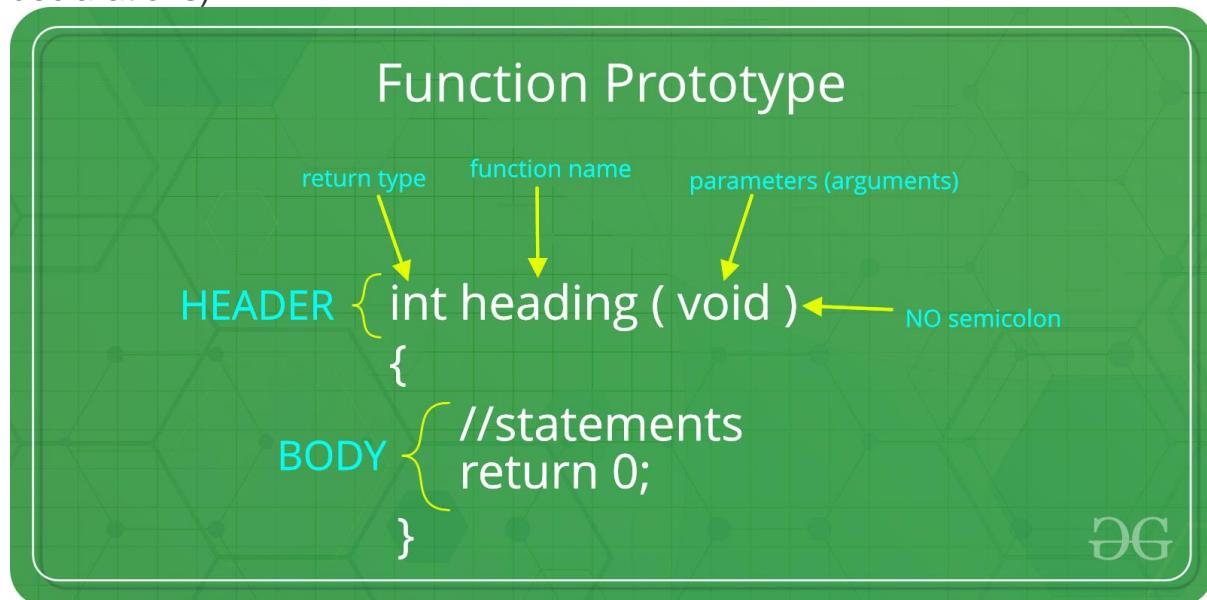
m is 20

Why do we need functions?

- Functions help us in reducing code redundancy. If functionality is performed at multiple places in software, then rather than writing the same code, again and again, we create a function and call it everywhere. This also helps in maintenance as we have to change at one place if we make future changes to the functionality.
- Functions make code modular. Consider a big file having many lines of code. It becomes really simple to read and use the code if the code is divided into functions.
- Functions provide abstraction. For example, we can use library functions without worrying about their internal working.

Function Declaration

A function declaration tells the compiler about the number of parameters function takes, data-types of parameters, and return type of function. Putting parameter names in function declaration is optional in the function declaration, but it is necessary to put them in the definition. Below are an example of function declarations. (parameter names are not there in below declarations)



```

// A function that takes two integers as parameters
// and returns an integer
int max(int, int);

// A function that takes an int pointer and an int variable as parameters
// and returns a pointer of type int
int *swap(int*,int);

// A function that takes a char as parameters
// and returns an reference variable
char *call(char b);

// A function that takes a char and an int as parameters
// and returns an integer
int fun(char, int);

```

It is always recommended to declare a function before it is used
 (See [this](#), [this](#) and [this](#) for details)

In C, we can do both declaration and definition at the same place, like done in the above example program.

C also allows to declare and define functions separately, this is especially needed in the case of library functions. The library functions are declared in header files and defined in library files. Below is an example declaration.

Parameter Passing to functions

The parameters passed to function are called ***actual parameters***. For example, in the above program 10 and 20 are actual parameters.

The parameters received by function are called ***formal parameters***. For example, in the above program x and y are formal parameters.

There are two most popular ways to pass parameters.

Pass by Value: In this parameter passing method, values of actual parameters are copied to function's formal parameters and the two types of parameters are stored in different memory locations. So any changes made inside functions are not reflected in actual parameters of caller.

Pass by Reference Both actual and formal parameters refer to same locations, so any changes made inside the function are actually reflected in actual parameters of caller.

Parameters are always passed by value in C. For example. in the below code, value of x is not modified using the function fun().

- C
- C++

```
#include <iostream>

using namespace std;

void fun(int x) {
    x = 30;
}

int main() {
    int x = 20;
    fun(x);
    cout << "x = " << x;
    return 0;
}
```

Output:

x = 20

However, in C, we can use pointers to get the effect of pass-by reference. For example, consider the below program. The function fun() expects a pointer ptr to an integer (or an address of an integer). It modifies the value at the address ptr. The dereference operator * is used to access the value at an address. In the statement ‘*ptr = 30’, value at address ptr is changed to 30. The address operator & is used to get the address of a variable of any data type. In the function call statement ‘fun(&x)’, the address of x is passed so that x can be modified using its address.

- C
- C++

```
#include <iostream>

using namespace std;
```

```

void fun(int *ptr)
{
    *ptr = 30;
}

int main() {
    int x = 20;
    fun(&x);
    cout << "x = " << x;

    return 0;
}

```

Output:

x = 30

Following are some important points about functions in C.

- 1)** Every C program has a function called main() that is called by operating system when a user runs the program.
- 2)** Every function has a return type. If a function doesn't return any value, then void is used as a return type. Moreover, if the return type of the function is void, we still can use return statement in the body of function definition by not specifying any constant, variable, etc. with it, by only mentioning the 'return;' statement which would symbolize the termination of the function as shown below:

```

void function name(int a)
{
    ....... //Function Body
    return; //Function execution would get terminated
}

```

- 3)** In C, functions can return any type except arrays and functions. We can get around this limitation by returning pointer to array or pointer to function.

4) Empty parameter list in C means that the parameter list is not specified and function can be called with any parameters. In C, it is not a good idea to declare a function like fun(). To declare a function that can only be called without any parameter, we should use “void fun(void)”.

As a side note, in C++, an empty list means a function can only be called without any parameter. In C++, both void fun() and void fun(void) are same.

5) If in a C program, a function is called before its declaration then the C compiler automatically assumes the declaration of that function in the following way:

```
int function name();
```

And in that case, if the return type of that function is different than INT, compiler would show an error.

Main Function:

The main function is a special function. Every C++ program must contain a function named main. It serves as the entry point for the program. The computer will start running the code from the beginning of the main function.

Types of main Function:

1) The first type is – main function without parameters :

```
// Without Parameters

int main()
{
    ...
    return 0;
}
```

2) Second type is main function with parameters :

```
// With Parameters

int main(int argc, char * const argv[])
{
    ...
    return 0;
}
```

The reason for having the parameter option for the main function is to allow input from the command line.

When you use the main function with parameters, it saves every group of characters (separated by a space) after the program name as elements in an array named argv.

Since the main function has the return type of int, the programmer must always have a return statement in the code. The number that is returned is used to inform the calling program what the result of the program's execution was. Returning 0 signals that there were no problems.

15.1.Default Arguments in C++

A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the caller of the function doesn't provide a value for the argument with a default value. In case any value is passed the default value is overridden.

1) Following is a simple C++ example to demonstrate the use of default arguments. Here, we don't have to write 3 sum functions, only one function works by using default values for 3rd and 4th arguments.

- CPP

```
// CPP Program to demonstrate Default Arguments
#include <iostream>
using namespace std;

// A function with default arguments,
// it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z = 0, int w = 0) //assigning default values to z,w as 0
{
    return (x + y + z + w);
}

// Driver Code
int main()
{
    // Statement 1
    cout << sum(10, 15) << endl;

    // Statement 2
    cout << sum(10, 15, 25) << endl;

    // Statement 3
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}
```

Output

25

50

80

Explanation: In statement 1, only two values are passed, hence the variables z and w take the default values as 0. In statement 2, three values are passed, so the value of z is over-ridden with 25. In statement 3, four

values are passed, so the value of z and w are over-ridden with 25 and 30 respectively.

2) When Function overloading is done along with default values. Then we need to make sure it will not be ambiguous.

The compiler will throw an error, if ambiguous. Following is the modified version of above program,

- CPP

```
// CPP Program to demonstrate Function overloading in
// Default Arguments
#include <iostream>
using namespace std;

// A function with default arguments, it can be called with
// 2 arguments or 3 arguments or 4 arguments.
int sum(int x, int y, int z = 0, int w = 0)
{
    return (x + y + z + w);
}
int sum(int x, int y, float z = 0, float w = 0)
{
    return (x + y + z + w);
}
// Driver Code
int main()
{
    cout << sum(10, 15) << endl;
    cout << sum(10, 15, 25) << endl;
    cout << sum(10, 15, 25, 30) << endl;
    return 0;
}
```

Error:

```
prog.cpp: In function 'int main()':
prog.cpp:17:20: error: call of overloaded
'sum(int, int)' is ambiguous
    cout << sum(10, 15) << endl;
                           ^
prog.cpp:6:5: note: candidate:
int sum(int, int, int, int)
    int sum(int x, int y, int z=0, int w=0)
                           ^
prog.cpp:10:5: note: candidate:
```

```

int sum(int, int, float, float)
int sum(int x, int y, float z=0, float w=0)
^

```

3) A constructor can contain default parameters as well. A default constructor can either have no parameters or have parameters with default arguments.

- C++

```

// CPP code to demonstrate use of default arguments in
// Constructors

#include <iostream>
using namespace std;
class A {
public:
    int sum = 0;
    A(); // default constructor with no argument
    A(int x = 0); // default constructor with one
                   // arguments

};


```

Explanation: Here, we see a default constructor with no arguments and a default constructor with default argument. The default constructor with arguments has a default parameter x which has been assigned a value 0.

Key Points:

- Default arguments are different from constant arguments as constant arguments can't be changed whereas default arguments can be overwritten if required.
- Default arguments are overwritten when calling function provides values for them. For example, calling of function sum(10, 15, 25, 30) overwrites the value of z and w to 25 and 30 respectively.
- During the calling of function, arguments from calling function to called function are copied from left to right. Therefore, sum(10, 15, 25) will assign 10, 15, and 25 to x, y, and z. Therefore, the default value is used for w only.
- Once the default value is used for an argument in the function definition, all subsequent arguments to it must have a default value. It can also be stated as default arguments are assigned from right to left. For example, the following function definition is invalid as the subsequent argument of default variable z is not default.

```

// Invalid because z has default value, but w after it doesn't have
a default value

int sum(int x, int y, int z = 0, int w).

```

Advantages of Default Arguments:

- Default arguments are useful when we want to increase the capabilities of an existing function by adding another default arguments.
- It helps in reducing the size of program.
- It provides a simple and effective programming approach.
- Default arguments improves consistency of program.

Disadvantages of Default Arguments:

- It increase the execution time as compiler need to replace the omitted arguments by there default values in the function call.

15.2. Inline Functions in C++

Inline function is one of the important feature of C++. So, let's first understand why inline functions are used and what is the purpose of inline function?

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register and returns control to the calling function. This can become overhead if the execution time of function is less than the switching time from the caller function to called function (callee). For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because execution time of small function is less than the switching time.

C++ provides an inline functions to reduce the function call overhead. Inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of inline function call. This substitution is performed by the C++ compiler at compile time. Inline function may increase efficiency if it is small. The syntax for defining the function inline is:

```
inline return-type function-name(parameters)
{
    // function code
}
```

Remember, inlining is only a request to the compiler, not a command. Compiler can ignore the request for inlining. Compiler may not perform inlining in such circumstances like:

- 1) If a function contains a loop. (for, while, do-while)
- 2) If a function contains static variables.
- 3) If a function is recursive.
- 4) If a function return type is other than void, and the return statement doesn't exist in function body.
- 5) If a function contains switch or goto statement.

Inline functions provide following advantages:

- 1) Function call overhead doesn't occur.
- 2) It also saves the overhead of push/pop variables on the stack when function is called.
- 3) It also saves overhead of a return call from a function.

4) When you inline a function, you may enable compiler to perform context specific optimization on the body of function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of calling context and the called context.

5) Inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function call preamble and return.

Inline function disadvantages:

1) The added variables from the inlined function consumes additional registers, After in-lining function if variables number which are going to use register increases than they may create overhead on register variable resource utilization. This means that when inline function body is substituted at the point of function call, total number of variables used by the function also gets inserted. So the number of register going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause an overhead on register utilization.

2) If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of same code.

3) Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory.

4) Inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because compiler would require to replace all the code once again to reflect the changes, otherwise it will continue with old functionality.

5) Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed.

6) Inline functions might cause thrashing because inlining might increase size of the binary executable file. Thrashing in memory causes performance of computer to degrade.

The following program demonstrates the use of use of inline function.

```
#include <iostream>

using namespace std;

inline int cube(int s)

{
    return s*s*s;
}

int main()
```

```

{
    cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
} //Output: The cube of 3 is: 27

```

Inline function and classes:

It is also possible to define the inline function inside the class. In fact, all the functions defined inside the class are implicitly inline. Thus, all the restrictions of inline functions are also applied here. If you need to explicitly declare inline function in the class then just declare the function inside the class and define it outside the class using inline keyword.

For example:

```

class S
{
public:
    inline int square(int s) // redundant use of inline
    {
        // this function is automatically inline
        // function body
    }
};

```

The above style is considered as a bad programming style. The best programming style is to just write the prototype of function inside the class and specify it as an inline in the function definition.

For example:

```

class S
{
public:
    int square(int s); // declare the function
};

inline int S::square(int s) // use inline prefix

```

{

}

The following program demonstrates this concept:

```
#include <iostream>

using namespace std;

class operation

{
    int a,b,add,sub,mul;
    float div;

public:
    void get();
    void sum();
    void difference();
    void product();
    void division();

};

inline void operation :: get()
{
    cout << "Enter first value:";

    cin >> a;

    cout << "Enter second value:";

    cin >> b;
}

inline void operation :: sum()
{
    add = a+b;

    cout << "Addition of two numbers: " << a+b << "\n";
}
```

```

}

inline void operation :: difference()
{
    sub = a-b;
    cout << "Difference of two numbers: " << a-b << "\n";
}

inline void operation :: product()
{
    mul = a*b;
    cout << "Product of two numbers: " << a*b << "\n";
}

inline void operation ::division()
{
    div=a/b;
    cout<<"Division of two numbers: "<<a/b<<"\n" ;
}

int main()
{
    cout << "Program using inline function\n";
    operation s;
    s.get();
    s.sum();
    s.difference();
    s.product();
    s.division();
    return 0;
}

```

```
}
```

Output:

```
Enter first value: 45
Enter second value: 15
Addition of two numbers: 60
Difference of two numbers: 30
Product of two numbers: 675
Division of two numbers: 3
```

What is wrong with macro?

Readers familiar with the C language knows that C language uses macro. The preprocessor replace all macro calls directly within the macro code. It is recommended to always use inline function instead of macro. According to Dr. Bjarne Stroustrup the creator of C++ that macros are almost never necessary in C++ and they are error prone. There are some problems with the use of macros in C++. Macro cannot access private members of class. Macros looks like function call but they are actually not.

Example:

```
#include <iostream>

using namespace std;

class S
{
    int m;

public:
#define MAC(S)::m) // error
};
```

C++ compiler checks the argument types of inline functions and necessary conversions are performed correctly. Preprocessor macro is not capable for doing this. One other thing is that the macros are managed by preprocessor and inline functions are managed by C++ compiler.

Remember: It is true that all the functions defined inside the class are implicitly inline and C++ compiler will perform inline call of these functions, but C++ compiler cannot perform inlining if the function is virtual. The reason is call to a virtual function is resolved at runtime instead of compile time. Virtual means wait until runtime and inline means during compilation, if the compiler doesn't know which function will be called, how it can perform inlining?

One other thing to remember is that it is only useful to make the function inline if the time spent during a function call is more compared to the function body execution time. An example where inline function has no effect at all:

```
inline void show()  
{  
    cout << "value of S = " << S << endl;  
}
```

The above function relatively takes a long time to execute. In general function which performs input output (I/O) operation shouldn't be defined as inline because it spends a considerable amount of time. Technically inlining of show() function is of limited value because the amount of time the I/O statement will take far exceeds the overhead of a function call.

Depending upon the compiler you are using the compiler may show you warning if the function is not expanded inline. Programming languages like Java & C# doesn't support inline functions.

But in Java, the compiler can perform inlining when the small final method is called, because final methods can't be overridden by sub classes and call to a final method is resolved at compile time. In C# JIT compiler can also optimize code by inlining small function calls (like replacing body of a small function when it is called in a loop).

Last thing to keep in mind that inline functions are the valuable feature of C++. An appropriate use of inline function can provide performance enhancement but if inline functions are used arbitrarily then they can't provide better result. In other words don't expect better performance of program. Don't make every function inline. It is better to keep inline functions as small as possible.

References:

- 1) [Effective C++ , Scott Meyers](#)
- 2) <http://www.parashift.com/c++-faq/inline-and-perf.html>
- 3) <http://www.cplusplus.com/forum/articles/20600/>
- 4) [Thinking in C++, Volume 1, Bruce Eckel.](#)
- 5) [C++ the complete reference, Herbert Schildt](#)

15.3.Return From Void Functions in C++

Void functions are known as **Non-Value Returning functions**. They are “void” due to the fact that they are not supposed to return values. True, but not completely. We cannot return values but there is something we can surely return from void functions. **Void functions do not have a return type, but they can do return values.** Some of the cases are listed below:

1) A Void Function Can Return: We can simply write a return statement in a void fun(). In fact, it is considered a good practice (for readability of code) to write a return; statement to indicate the end of the function.

- CPP

```
// CPP Program to demonstrate void functions
#include <iostream>
using namespace std;

void fun()
{
    cout << "Hello";

    // We can write return in void
    return;
}

// Driver Code
int main()
{
    fun();
    return 0;
}
```

Output

Hello

2) A void fun() can return another void function: A void function can also call another void function while it is terminating. For example,

- CPP

```
// C++ code to demonstrate void()
// returning void()
#include <iostream>
using namespace std;
```

```

// A sample void function
void work()
{
    cout << "The void function has returned "
        " a void() !!! \n";
}

// Driver void() returning void work()
void test()
{
    // Returning void function
    return work();
}

// Driver Code
int main()
{
    // Calling void function
    test();
    return 0;
}

```

Output

The void function has returned a void() !!!

The above code explains how void() can actually be useful to return void functions without giving errors.

3) A void() can return a void value: A void() cannot return a value that can be used. But it can return a value that is void without giving an error. For example,

- CPP

```

// C++ code to demonstrate void()
// returning a void value
#include <iostream>
using namespace std;

// Driver void() returning a void value
void test()
{
    cout << "Hello";

    // Returning a void value
    return (void)"Doesn't Print";
}

```

```
}
```

```
// Driver Code
int main()
{
    test();
    return 0;
}
```

Output

Hello

15.4. Functors in C++

Please note that the title is **Functors** (Not Functions)!!

Consider a function that takes only one argument. However, while calling this function we have a lot more information that we would like to pass to this function, but we cannot as it accepts only one parameter. What can be done?

One obvious answer might be global variables. However, good coding practices do not advocate the use of global variables and say they must be used only when there is no other alternative.

Functors are objects that can be treated as though they are a function or function pointer. Functors are most commonly used along with STLs in a scenario like following:

Below program uses [transform\(\) in STL](#) to add 1 to all elements of arr[].

```
// A C++ program uses transform() in STL to add
// 1 to all elements of arr[]

#include <bits/stdc++.h>

using namespace std;

int increment(int x) { return (x+1); }

int main()
{
    int arr[] = {1, 2, 3, 4, 5};
    int n = sizeof(arr)/sizeof(arr[0]);

    // Apply increment to all elements of
    // arr[] and store the modified elements
    // back in arr[]
    transform(arr, arr+n, arr, increment);

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

```

```

    cout << arr[i] <<" ";
}


```

Output:

2 3 4 5 6

This code snippet adds only one value to the contents of the arr[]. Now suppose, that we want to add 5 to contents of arr[].

See what's happening? As transform requires a unary function(a function taking only one argument) for an array, we cannot pass a number to increment(). And this would, in effect, make us write several different functions to add each number. What a mess. This is where functors come into use.

A functor (or function object) is a C++ class that acts like a function. Functors are called using the same old function call syntax. To create a functor, we create a object that overloads the *operator()*.

The line,
`MyFunctor(10);`

Is same as

`MyFunctor.operator()(10);`

Let's delve deeper and understand how this can actually be used in conjunction with STLs.

```

// C++ program to demonstrate working of
// functors.

#include <bits/stdc++.h>

using namespace std;

// A Functor

class increment
{
private:
    int num;

public:

```

```

increment(int n) : num(n) { }

// This operator overloading enables calling
// operator function () on objects of increment

int operator () (int arr_num) const {

    return num + arr_num;

}

};

// Driver code

int main()

{

    int arr[] = {1, 2, 3, 4, 5};

    int n = sizeof(arr)/sizeof(arr[0]);

    int to_add = 5;

    transform(arr, arr+n, arr, increment(to_add));

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

        cout << arr[i] << " ";

}

```

Output:

6 7 8 9 10

Thus, here, Increment is a functor, a C++ class that acts as a function.

The line,
`transform(arr, arr+n, arr, increment(to_add));`

is the same as writing below two lines,
`// Creating object of increment`
`increment obj(to_add);`

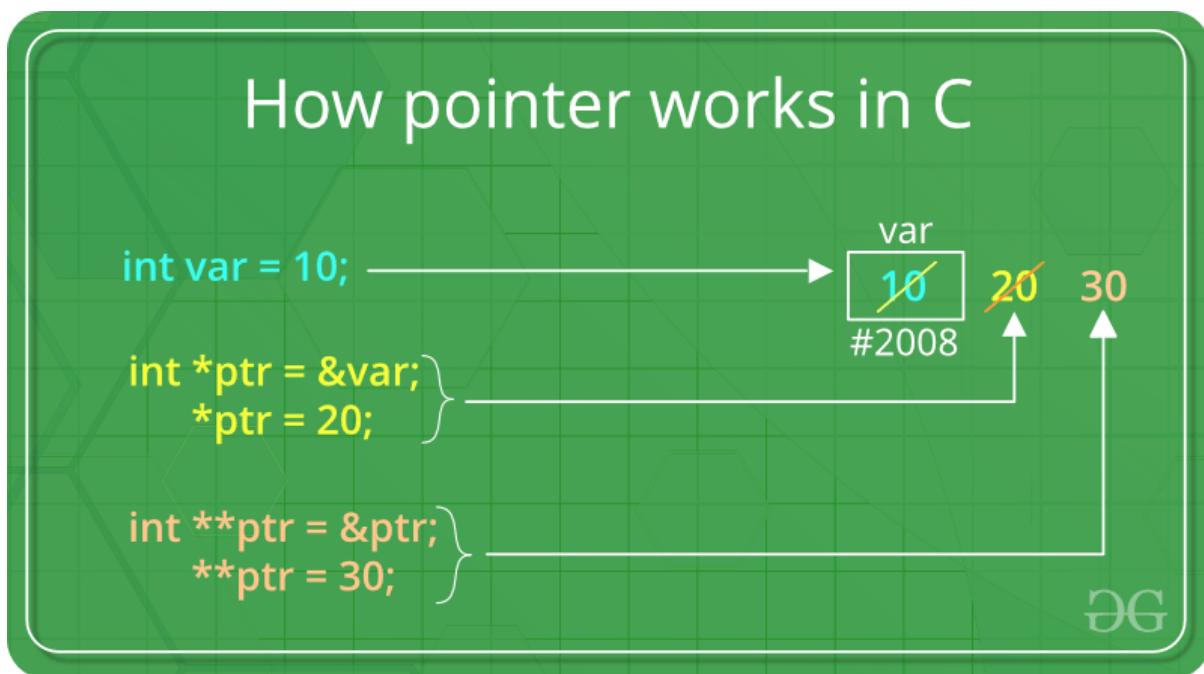
`// Calling () on object`
`transform(arr, arr+n, arr, obj);`

Thus, an object *a* is created that overloads the *operator()*. Hence, functors can be used effectively in conjunction with C++ STLs.

16. Pointers in C and C++ | Set 1 (Introduction, Arithmetic and Array)

Pointers store address of variables or a memory location.

```
// General syntax  
datatype *var_name;  
  
// An example pointer "ptr" that holds  
// address of an integer variable or holds  
// address of a memory whose value(s) can  
// be accessed as integer values through "ptr"  
int *ptr;  
Using a Pointer:
```



To use pointers in C, we must understand below two operators.

- To access address of a variable to a pointer, we use the unary operator **&** (ampersand) that returns the address of that variable. For example `&x` gives us address of variable x.

- C

```

// The output of this program can be different
// in different runs. Note that the program
// prints address of a variable and a variable
// can be assigned different address in different
// runs.
#include <stdio.h>

int main()
{
    int x;

    // Prints address of x
    printf("%p", &x);

    return 0;
}

```

- One more operator is **unary *** (Asterisk) which is used for two things :
 - To declare a pointer variable: When a pointer variable is declared in C/C++, there must be a * before its name.

• C

```

// C program to demonstrate declaration of
// pointer variables.
#include <stdio.h>
int main()
{
    int x = 10;

    // 1) Since there is * in declaration, ptr
    // becomes a pointer variable (a variable
    // that stores address of another variable)
    // 2) Since there is int before *, ptr is
    // pointer to an integer type variable
    int *ptr;

    // & operator before x is used to get address
    // of x. The address of x is assigned to ptr.
    ptr = &x;

    return 0;
}

```

- To access the value stored in the address we use the unary operator (*) that returns the value of the variable located at the address specified by its operand. This is also called **Dereferencing**.

- C++
- C

```
// C++ program to demonstrate use of * for pointers in C++
#include <iostream>
using namespace std;

int main()
{
    // A normal integer variable
    int Var = 10;

    // A pointer variable that holds address of var.
    int *ptr = &Var;

    // This line prints value at address stored in ptr.
    // Value stored is value of variable "var"
    cout << "Value of Var = " << *ptr << endl;

    // The output of this line may be different in different
    // runs even on same machine.
    cout << "Address of Var = " << ptr << endl;

    // We can also use ptr as lvalue (Left hand
    // side of assignment)
    *ptr = 20; // Value at address is now 20

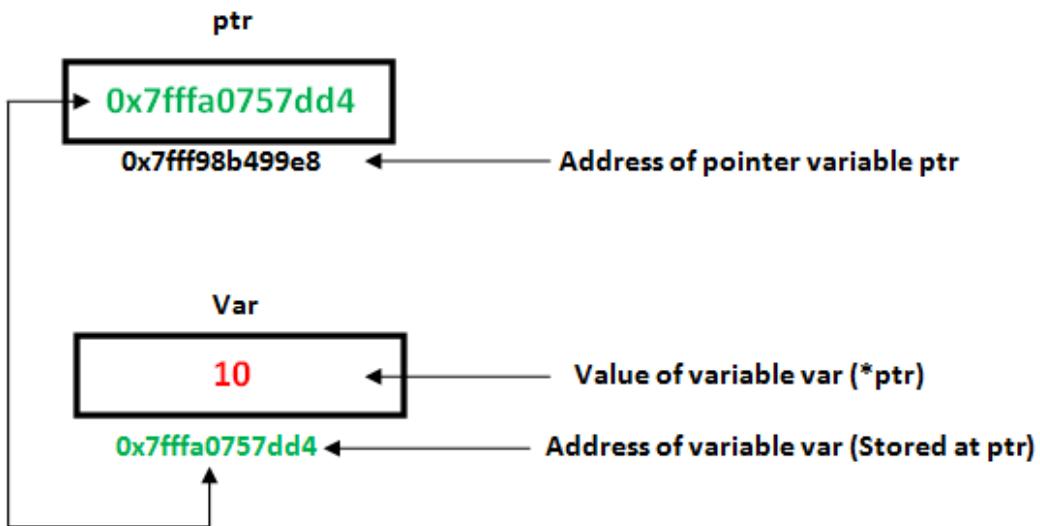
    // This prints 20
    cout << "After doing *ptr = 20, *ptr is " << *ptr << endl;

    return 0;
}

// This code is contributed by
// shubhamsingh10

    • Output :
Value of Var = 10
Address of Var = 0x7ffffa057dd4
After doing *ptr = 20, *ptr is 20
```

- Below is pictorial representation of above program:



Pointer Expressions and Pointer Arithmetic

A limited set of arithmetic operations can be performed on pointers. A pointer may be:

- incremented (`++`)
- decremented (`--`)
- an integer may be added to a pointer (`+` or `+=`)
- an integer may be subtracted from a pointer (`-` or `-=`)

Pointer arithmetic is meaningless unless performed on an array.

Note : Pointers contain addresses. Adding two addresses makes no sense, because there is no idea what it would point to. Subtracting two addresses lets you compute the offset between these two addresses.

- CPP

```
// C++ program to illustrate Pointer Arithmetic
// in C/C++
#include <bits/stdc++.h>

// Driver program
int main()
{
```

```

// Declare an array

int v[3] = {10, 100, 200};

// Declare pointer variable

int *ptr;

// Assign the address of v[0] to ptr

ptr = v;

for (int i = 0; i < 3; i++)
{
    printf("Value of *ptr = %d\n", *ptr);
    printf("Value of ptr = %p\n\n", ptr);

    // Increment pointer ptr by 1

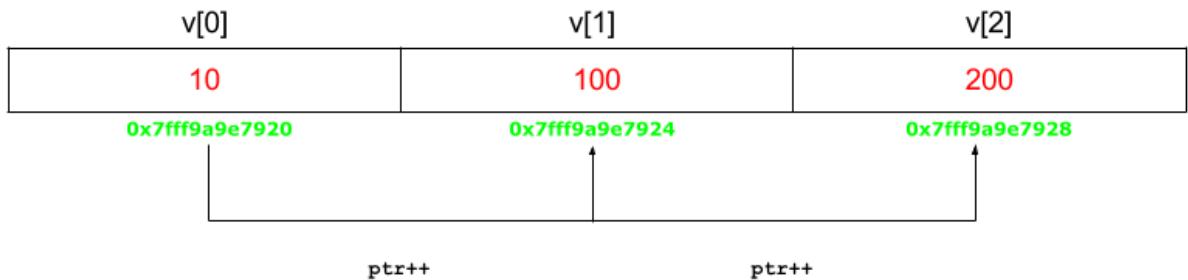
    ptr++;
}

```

Output:Value of *ptr = 10
Value of ptr = 0x7ffcae30c710

Value of *ptr = 100
Value of ptr = 0x7ffcae30c714

Value of *ptr = 200
Value of ptr = 0x7ffcae30c718



Array Name as Pointers

An array name acts like a pointer constant. The value of this pointer constant is the address of the first element.

For example, if we have an array named val then **val** and **&val[0]** can be used interchangeably.

- CPP

```
// C++ program to illustrate Array Name as Pointers in C++
#include <bits/stdc++.h>
using namespace std;

void geeks()
{
    // Declare an array
    int val[3] = { 5, 10, 15};

    // Declare pointer variable
    int *ptr;

    // Assign address of val[0] to ptr.
    // We can use ptr=&val[0];(both are same)
    ptr = val ;
    cout << "Elements of the array are: ";
    cout << ptr[0] << " " << ptr[1] << " " << ptr[2];

    return;
}
```

```
}
```

```
// Driver program
```

```
int main()
```

```
{
```

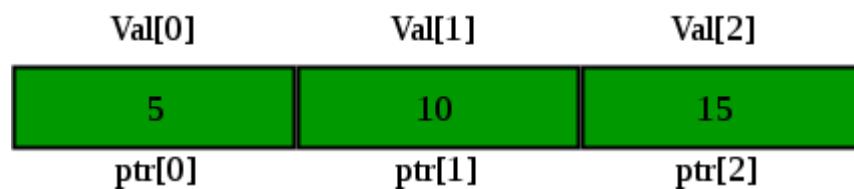
```
    geeks();
```

```
    return 0;
```

```
}
```

Output:

Elements of the array are: 5 10 15



Now if this ptr is sent to a function as an argument then the array val can be accessed in a similar fashion.

Pointers and Multidimensional Arrays

Consider pointer notation for the two-dimensional numeric arrays. consider the following declaration

```
int nums[2][3] = { {16, 18, 20}, {25, 26, 27} };
```

In general, **nums[i][j]** is equivalent to ***(*(nums+i)+j)**

Pointer Notation	Array Notation	Value
------------------	----------------	-------

*(*nums)	nums[0][0]	16
-----------------	-------------------	-----------

*(*nums + 1)	nums[0][1]	18
---------------------	-------------------	-----------

*(*nums + 2)	nums[0][2]	20
---------------------	-------------------	-----------

Pointer Notation	Array Notation	Value
<code>*(*(nums + 1))</code>	<code>nums[1][0]</code>	25
<code>*(*(nums + 1) + 1)</code>	<code>nums[1][1]</code>	26
<code>*(*(nums + 1) + 2)</code>	<code>nums[1][2]</code>	27

Related Articles:

[Applications of pointers in C/C++.](#)

Prerequisite : [Pointers in C/C++, Memory Layout of C Programs.](#)

- **To pass arguments by reference.** Passing by reference serves two purposes

(i) **To modify variable of function in other.** Example to swap two variables;

- C
- C++

```
// C++ program to demonstrate that we can change
// local values of one function in another using
// pointers.
```

```
#include <iostream>

using namespace std;

void swap(int* x, int* y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}
```

```

int main()
{
    int x = 10, y = 20;
    swap(&x, &y);
    cout << x << " " << y << endl;
    return 0;
}

```

Output :

20 10

(ii) **For efficiency purpose.** Example passing large structure without reference would create a copy of the structure (hence wastage of space).

Note : The above two can also be achieved through [References in C++](#).

- **For accessing array elements.** Compiler internally uses pointers to access array elements.

- C
- C++

```

// C++ program to demonstrate that compiler
// internally uses pointer arithmetic to access
// array elements.

#include <iostream>

using namespace std;

```

```

int main()
{
    int arr[] = { 100, 200, 300, 400 };

    // Compiler converts below to *(arr + 2).

    cout << arr[2] << " ";
}

```

```

// So below also works.

cout << *(arr + 2) << " ";

return 0;
}

```

Output :

300 300

- **To return multiple values.** Example returning square and square root of numbers.

• C

• C++

```

// C++ program to demonstrate that using a pointer
// we can return multiple values.

#include <bits/stdc++.h>

using namespace std;

void fun(int n, int* square, double* sq_root)
{
    *square = n * n;
    *sq_root = sqrt(n);
}

int main()
{
    int n = 100;
    int* sq = new int;

```

```

double* sq_root = new double;
fun(n, sq, sq_root);
cout << *sq << " " << *sq_root;
return 0;
}

```

Output :

10000 10

- **Dynamic memory allocation** : We can use pointers to dynamically allocate memory. The advantage of dynamically allocated memory is, it is not deleted until we explicitly delete it.

- C

- C++

```

// C++ program to dynamically allocate an
// array of given size.
#include <iostream>

using namespace std;

int* createArr(int n)
{
    return new int[n];
}

int main()
{
    int* pt = createArr(10);
    return 0;
}

```

Some Questions Regarding Pointers:

1. *What are the uses of a pointer?*

Ans. Pointer is used in the following cases

- i) It is used to access array elements
- ii) It is used for dynamic memory allocation.
- iii) It is used in Call by reference
- iv) It is used in data structures like trees, graph, linked list etc.

2. *Are pointers integer?*

Ans. No, pointers are not integers. A pointer is an address and a positive number.

3. *What does the error 'Null Pointer Assignment' means and what causes this error?*

Ans. As null pointer points to nothing so accessing a uninitialized pointer or invalid location may cause an error.

4. *How pointer variables are initialized?*

Ans. Pointer variables are initialized by one of the following ways.

- I. Static memory allocation
- II. Dynamic memory allocation

5. *What is pointer to a pointer?*

Ans. If a pointer variable points another pointer value. Such a situation is known as a pointer to a pointer.

Example:

```
int *p1,**p2,v=10;  
P1=&v; p2=&p1;  
Here p2 is a pointer to a pointer
```

6. *What is an array of pointers?*

Ans: If the elements of an array are addresses, such an array is called an array of pointers.

- **To implement data structures.**

Example [linked list](#), [tree](#), etc. We cannot use [C++ references](#) to implement these data structures because references are fixed to a location (For example, we can not traverse a linked list using references)

- **To do system level programming where memory addresses are useful.** For example shared memory used by multiple threads. For more examples, see [IPC through shared memory](#), [Socket Programming in C/C++](#), etc

16.1.Opaque Pointer

What is an opaque pointer?

Opaque as the name suggests is something we can't see through. e.g. wood is opaque. Opaque pointer is a pointer which points to a data structure whose contents are not exposed at the time of its definition.

Following pointer is opaque. One can't know the data contained in STest structure by looking at the definition.

```
struct STest* pSTest;
```

It is safe to assign NULL to an opaque pointer.

```
pSTest = NULL;
```

Why Opaque pointer?

There are places where we just want to hint the compiler that "Hey! This is some data structure which will be used by our clients. Don't worry, clients will provide its implementation while preparing compilation unit". Such type of design is robust when we deal with shared code. Please see below example: Let's say we are working on an app to deal with images. Since we are living in a world where everything is moving to cloud and devices are very affordable to buy, we want to develop apps for windows, android and apple platforms. So, it would be nice to have a good design which is robust, scalable and flexible as per our requirements. We can have shared code which would be used by all platforms and then different end-point can have platform specific code.

To deal with images, we have a CImage class exposing APIs to deal with various image operations (scale, rotate, move, save etc).

Since all the platforms will be providing same operations, we would define this class in a header file. But the way an image is handled might differ across platforms. Like Apple can have different mechanism to access pixels of an image than Windows does. This means that APIs might demand different set of info to perform operations. So to work on shared code, this is what we would like to do:

Image.h : A header file to store class declaration.

```
// This class provides API to deal with various
// image operations. Different platforms can
// implement these operations in different ways.
class CImage
{
public:
    CImage();
    ~CImage();
    struct SImageInfo* pImageInfo;
    void Rotate(double angle);
```

```

    void Scale(double scaleFactorX,
               double scaleFactorY);
    void Move(int toX, int toY);
private:
    void InitImageInfo();
};


```

Image.cpp : Code that will be shared across different end-points

```

// Constructor and destructor for CImage
CImage::CImage()
{
    InitImageInfo();
}

CImage::~CImage()
{
    // Destroy stuffs here
}

```

Image_windows.cpp : Code specific to Windows will reside here

```

struct SImageInfo
{
    // Windows specific DataSet
};

void CImage::InitImageInfo()
{
    pImageInfo = new SImageInfo;
    // Initialize windows specific info here
}

void CImage::Rotate()
{
    // Make use of windows specific SImageInfo
}

```

Image_apple.cpp : Code specific to Apple will reside here

```

struct SImageInfo
{
    // Apple specific DataSet
};
void CImage::InitImageInfo()
{
    pImageInfo = new SImageInfo;
}

```

```
// Initialize apple specific info here  
}  
void CImage::Rotate()  
{  
    // Make use of apple specific SImageInfo  
}
```

As it can be seen from the above example, **while defining blueprint of the CImage class we are only mentioning that there is a SImageInfo data structure.**

The content of SImageInfo is unknown. Now it is the responsibility of clients(windows, apple, android) to define that data structure and use it as per their requirement. If in future we want to develop app for a new end-point 'X', the design is already there. We only need to define SImageInfo for end-point 'X' and use it accordingly.

16.2. References in C++

When a variable is declared as a reference, it becomes an alternative name for an existing variable. A variable can be declared as a reference by putting ‘&’ in the declaration.

- CPP

```
#include <iostream>

using namespace std;

int main()
{
    int x = 10;

    // ref is a reference to x.

    int& ref = x;

    // Value of x is now changed to 20
    ref = 20;
    cout << "x = " << x << '\n';

    // Value of x is now changed to 30
    x = 30;
    cout << "ref = " << ref << '\n';

    return 0;
}
```

Output:

x = 20

ref = 30

Applications :

1. **Modify the passed parameters in a function:** If a function receives a reference to a variable, it can modify the value of the

variable. For example, the following program variables are swapped using references.

- CPP

```
#include <iostream>

using namespace std;

void swap (int& first, int& second)

{

    int temp = first;

    first = second;

    second = temp;

}

int main()

{

    int a = 2, b = 3;

    swap( a, b );

    cout << a << " " << b;

    return 0;

}
```

Output:

3 2

1. **Avoiding a copy of large structures:** Imagine a function that has to receive a large object. If we pass it without reference, a new copy of it is created which causes wastage of CPU time and memory. We can use references to avoid this.

- CPP

```
struct Student {

    string name;

    string address;

    int rollNo;
```

```

}

// If we remove & in below function, a new
// copy of the student object is created.
// We use const to avoid accidental updates
// in the function as the purpose of the function
// is to print s only.

void print(const Student &s)
{
    cout << s.name << " " << s.address << " " << s.rollNo << '\n';
}

```

- 1. In For Each Loops to modify all objects :** We can use references in for each loops to modify all elements.

- CPP

```

#include <bits/stdc++.h>

using namespace std;

int main()
{
    vector<int> vect{ 10, 20, 30, 40 };

    // We can modify elements if we
    // use reference

    for (int &x : vect)
    {
        x = x + 5;
    }

    // Printing elements

```

```

for (int x : vect)
{
    cout << x << " ";
}
cout << '\n';

return 0;
}

```

- For Each Loop to avoid the copy of objects:** We can use references in each loop to avoid a copy of individual objects when objects are large.

- CPP

```

#include <bits/stdc++.h>

using namespace std;

int main()
{
    vector<string> vect{"geeksforgeeks practice",
                        "geeksforgeeks write",
                        "geeksforgeeks ide"};

    // We avoid copy of the whole string
    // object by using reference.

    for (const auto& x : vect)
    {
        cout << x << '\n';
    }

    return 0;
}

```

16.3. References vs Pointers:

Both references and pointers can be used to change local variables of one function inside another function. Both of them can also be used to save copying of big objects when passed as arguments to functions or returned from functions, to get efficiency gain. Despite the above similarities, there are the following differences between references and pointers.

1. A pointer can be declared as void but a reference can never be void For example

```
int a = 10;  
void* aa = &a; // it is valid  
void& ar = a; // it is not valid
```

2. The pointer variable has n-levels/multiple levels of indirection i.e. single-pointer, double-pointer, triple-pointer. Whereas, the reference variable has only one/single level of indirection. The following code reveals the mentioned points:

3. Reference variable cannot be updated.
4. Reference variable is an internal pointer .
5. Declaration of Reference variable is preceded with ‘&’ symbol (but do not read it as “address of”).

- C++

```
#include <iostream>  
  
using namespace std;  
  
  
int main() {  
    int i = 10; // simple or ordinary variable.  
    int *p = &i; // single pointer  
    int **pt = &p; // double pointer  
    int ***ptr = &pt; // triple pointer  
  
    // All the above pointers differ in the value they store or point to.  
  
    cout << "i = " << i << "\t" << "p = " << p << "\t"  
        << "pt = " << pt << "\t" << "ptr = " << ptr << '\n';  
  
    int a = 5; // simple or ordinary variable  
    int &s = a;
```

```

int &S0 = S;
int &S1 = S0;
cout << "a = " << a << "\t" << "S = " << S << "\t"
     << "S0 = " << S0 << "\t" << "S1 = " << S1 << '\n';
// All the above references do not differ in their values
// as they all refer to the same variable.

}

```

- **References are less powerful than pointers**

- 1) Once a reference is created, it cannot be later made to reference another object; it cannot be reset. This is often done with pointers.
- 2) References cannot be NULL. Pointers are often made NULL to indicate that they are not pointing to any valid thing.
- 3) A reference must be initialized when declared. There is no such restriction with pointers.

Due to the above limitations, references in C++ cannot be used for implementing data structures like Linked List, Tree, etc. In Java, references don't have the above restrictions and can be used to implement all data structures. References being more powerful in Java is the main reason Java doesn't need pointers.

- **References are safer and easier to use:**

- 1) **Safer:** Since references must be initialized, wild references like wild pointers are unlikely to exist. It is still possible to have references that don't refer to a valid location (See questions 5 and 6 in the below exercise)
- 2) **Easier to use:** References don't need a dereferencing operator to access the value. They can be used like normal variables. '&' operator is needed only at the time of declaration. Also, members of an object reference can be accessed with dot operator ('.'), unlike pointers where arrow operator ('->') is needed to access members.

- Together with the above reasons, there are few places like the copy constructor argument where pointer cannot be used. Reference must be used to pass the argument in the copy constructor. Similarly, references must be used for overloading some operators like `++`.

Exercise:

Predict the output of the following programs. If there are compilation errors, then fix them.

Question 1

- CPP

```

#include <iostream>

using namespace std;

int& fun()
{
    static int x = 10;
    return x;
}

int main()
{
    fun() = 30;
    cout << fun();
    return 0;
}

```

Question 2

- CPP

```

#include <iostream>

using namespace std;

int fun(int& x)
{
    return x;
}

int main()
{
    cout << fun(10);
    return 0;
}

```

```
}
```

Question 3

- CPP

```
#include <iostream>

using namespace std;

void swap(char * &str1, char * &str2)
{
    char *temp = str1;
    str1 = str2;
    str2 = temp;
}

int main()
{
    char *str1 = "GEEKS";
    char *str2 = "FOR GEEKS";
    swap(str1, str2);
    cout << "str1 is " << str1 << '\n';
    cout << "str2 is " << str2 << '\n';
    return 0;
}
```

Question 4

- CPP

```
#include <iostream>

using namespace std;

int main()
```

```
{  
    int x = 10;  
    int *ptr = &x;  
    int &ptr1 = ptr;  
}
```

Question 5

- CPP

```
#include <iostream>  
using namespace std;  
  
int main()  
{  
    int *ptr = NULL;  
    int &ref = *ptr;  
    cout << ref << '\n';  
}
```

Question 6

- CPP

```
#include <iostream>
```

```
using namespace std;
```

```
int& fun()  
{  
    int x = 10;  
    return x;  
}
```

```
int main()
```

```
{  
    fun() = 30;  
    cout << fun();  
    return 0;  
}
```

16.4. ‘this’ pointer in C++

To understand ‘this’ pointer, it is important to know how objects look at functions and data members of a class.

1. Each object gets its own copy of the data member.
2. All-access the same function definition as present in the code segment.

Meaning each object gets its own copy of data members and all objects share a single copy of member functions.

Then now question is that if only one copy of each member function exists and is used by multiple objects, how are the proper data members are accessed and updated?

The compiler supplies an implicit pointer along with the names of the functions as ‘this’.

The ‘this’ pointer is passed as a hidden argument to all nonstatic member function calls and is available as a local variable within the body of all nonstatic functions. ‘this’ pointer is not available in static member functions as static member functions can be called without any object (with class name).

For a class X, the type of this pointer is ‘X*’. Also, if a member function of X is declared as const, then the type of this pointer is ‘const X *’ (see [this GFact](#))

In the early version of C++ would let ‘this’ pointer to be changed; by doing so a programmer could change which object a method was working on. This feature was eventually removed, and now this in C++ is an r-value.

C++ lets object destroy themselves by calling the following code :

```
delete this;
```

As Stroustrup said ‘this’ could be the reference than the pointer, but the reference was not present in the early version of C++. If ‘this’ is implemented as a reference then, the above problem could be avoided and it could be safer than the pointer.

Following are the situations where ‘this’ pointer is used:

1) When local variable’s name is same as member’s name

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

/* local variable is same as a member's name */
```

```

class Test
{
private:
    int x;
public:
    void setX (int x)
    {
        // The 'this' pointer is used to retrieve the object's x
        // hidden by the local variable 'x'
        this->x = x;
    }
    void print() { cout << "x = " << x << endl; }
};

int main()
{
    Test obj;
    int x = 20;
    obj.setX(x);
    obj.print();
    return 0;
}

```

Output:

x = 20

For constructors, [initializer list](#) can also be used when parameter name is same as member's name.

2) To return reference to the calling object

```
/* Reference to the calling object can be returned */
```

```

Test& Test::func ()
{
    // Some processing
    return *this;
}

```

When a reference to a local object is returned, the returned reference can be used to **chain function calls** on a single object.

```

#include<iostream>

using namespace std;

class Test
{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }
    Test &setX(int a) { x = a; return *this; }
    Test &setY(int b) { y = b; return *this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj1(5, 5);

    // Chained function calls. All calls modify the same object
    // as the same object is returned by reference
    obj1.setX(10).setY(20);
}

```

```
    obj1.print();  
    return 0;  
}
```

Output:

```
x = 10 y = 20
```

Exercise:

Predict the output of following programs. If there are compilation errors, then fix them.

Question 1

```
#include<iostream>  
  
using namespace std;  
  
  
class Test  
{  
  
private:  
    int x;  
  
public:  
    Test(int x = 0) { this->x = x; }  
    void change(Test *t) { this = t; }  
    void print() { cout << "x = " << x << endl; }  
};  
  
  
int main()  
{  
    Test obj(5);  
    Test *ptr = new Test (10);  
    obj.change(ptr);  
    obj.print();
```

```
    return 0;  
}
```

Question 2

```
#include<iostream>  
using namespace std;  
  
class Test  
{  
private:  
    int x;  
    int y;  
public:  
    Test(int x = 0, int y = 0) { this->x = x; this->y = y; }  
    static void fun1() { cout << "Inside fun1()"; }  
    static void fun2() { cout << "Inside fun2()"; this->fun1(); }  
};  
  
int main()  
{  
    Test obj;  
    obj.fun2();  
    return 0;  
}
```

Question 3

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

```

class Test
{
private:
    int x;
    int y;
public:
    Test (int x = 0, int y = 0) { this->x = x; this->y = y; }
    Test setX(int a) { x = a; return *this; }
    Test setY(int b) { y = b; return *this; }
    void print() { cout << "x = " << x << " y = " << y << endl; }
};

int main()
{
    Test obj1;
    obj1.setX(10).setY(20);
    obj1.print();
    return 0;
}

```

Question 4

```

#include<iostream>
using namespace std;

class Test
{
private:
    int x;

```

```
int y;

public:

Test(int x = 0, int y = 0) { this->x = x; this->y = y; }

void setX(int a) { x = a; }

void setY(int b) { y = b; }

void destroy() { delete this; }

void print() { cout << "x = " << x << " y = " << y << endl; }

};

int main()

{

    Test obj;

    obj.destroy();

    obj.print();

    return 0;

}
```

16.5.Smart Pointers in C++ and How to Use Them

In this article, we will be discussing smart pointers in C++. What are Smart Pointers, why, and how to use them properly?

Pointers are used for accessing the resources which are external to the program – like heap memory. So, for accessing the heap memory (if anything is created inside heap memory), pointers are used. When accessing any external resource we just use a copy of the resource. If we make any change to it, we just change it in the copied version. But, if we use a pointer to the resource, we'll be able to change the original resource.

Problems with Normal Pointers

Take a look at the code below.

- C++

```
#include <iostream>

using namespace std;

class Rectangle {
private:
    int length;
    int breadth;
};

void fun()
{
    // By taking a pointer p and
    // dynamically creating object
    // of class rectangle
    Rectangle* p = new Rectangle();
}

int main()
```

```

{
    // Infinite Loop
    while (1) {
        fun();
    }
}

```

In function *fun*, it creates a pointer that is pointing to the *Rectangle* object. The object *Rectangle* contains two integers, *length* and *breadth*. When the function *fun* ends, *p* will be destroyed as it is a local variable. But, the memory it consumed won't be deallocated because we forgot to use *delete p*; at the end of the function. That means the memory won't be free to be used by other resources. But, we don't need the variable anymore, but we need the memory. In function *main*, *fun* is called in an infinite loop. That means it'll keep creating *p*. It'll allocate more and more memory but won't free them as we didn't deallocate it. The memory that's wasted can't be used again. Which is a memory leak. The entire *heap* memory may become useless for this reason. C++11 comes up with a solution to this problem, [Smart Pointer](#).

Introduction of Smart Pointers

As we've known unconsciously not deallocating a pointer causes a memory leak that may lead to crash of the program. Languages Java, C# has *Garbage Collection Mechanisms* to smartly deallocate unused memory to be used again. The programmer doesn't have to worry about any memory leak. C++11 comes up with its own mechanism that's *Smart Pointer*. When the object is destroyed it frees the memory as well. So, we don't need to delete it as Smart Pointer does will handle it.

A *Smart Pointer* is a wrapper class over a pointer with an operator like * and -> overloaded. The objects of the smart pointer class look like normal pointers. But, unlike *Normal Pointers* it can deallocate and free destroyed object memory.

The idea is to take a class with a pointer, [destructor](#) and [overloaded operators](#) like * and ->. Since the destructor is automatically called when an object goes out of scope, the dynamically allocated memory would automatically be deleted (or reference count can be decremented). Consider the following simple *SmartPtr* class.

- C++

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

```

class SmartPtr {
    int* ptr; // Actual pointer

public:
    // Constructor: Refer https://www.geeksforgeeks.org/g-fact-93/
    // for use of explicit keyword
    explicit SmartPtr(int* p = NULL) { ptr = p; }

    // Destructor
    ~SmartPtr() { delete (ptr); }

    // Overloading dereferencing operator
    int& operator*() { return *ptr; }

};

int main()
{
    SmartPtr ptr(new int());
    *ptr = 20;
    cout << *ptr;

    // We don't need to call delete ptr: when the object
    // ptr goes out of scope, the destructor for it is automatically
    // called and destructor does delete ptr.

    return 0;
}

```

Output:

20

This only works for *int*. So, we'll have to create Smart Pointer for every object? No, there's a solution, *Template*. In the code below as you can see *T* can be of any type. Read more about [Template](#) here.

- C++

```
#include <iostream>

using namespace std;

// A generic smart pointer class
template <class T>
class SmartPtr {

    T* ptr; // Actual pointer

public:
    // Constructor
    explicit SmartPtr(T* p = NULL) { ptr = p; }

    // Destructor
    ~SmartPtr() { delete (ptr); }

    // Overloading dereferencing operator
    T& operator*() { return *ptr; }

    // Overloading arrow operator so that
    // members of T can be accessed
    // like a pointer (useful if T represents
    // a class or struct or union type)
    T* operator->() { return ptr; }

};

int main()
{
    SmartPtr<int> ptr(new int());
}
```

```

    *ptr = 20;
    cout << *ptr;
    return 0;
}

```

Output:

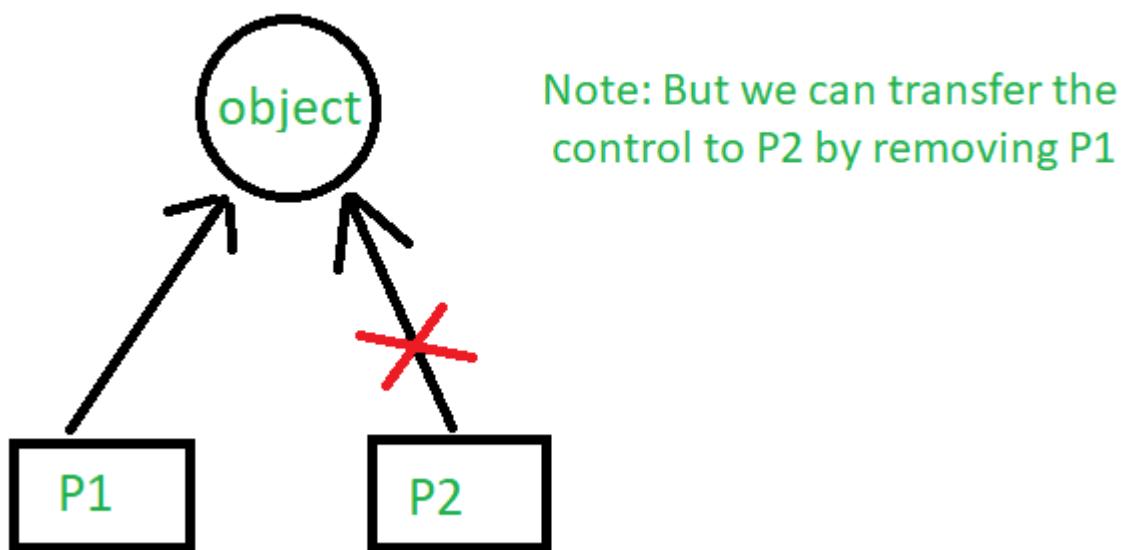
20

Note: Smart pointers are also useful in the management of resources, such as file handles or network sockets.

Types of Smart Pointers

1. `unique_ptr`

`unique_ptr` stores one pointer only. We can assign a different object by removing the current object from the pointer. Notice the code below. First, the `unique_pointer` is pointing to `P1`. But, then we remove `P1` and assign `P2` so the pointer now points to `P2`.



- C++14

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

```

class Rectangle {
    int length;
    int breadth;

public:
    Rectangle(int l, int b){
        length = l;
        breadth = b;
    }

    int area(){
        return length * breadth;
    }
};

int main(){
    unique_ptr<Rectangle> P1(new Rectangle(10, 5));
    cout << P1->area() << endl; // This'll print 50

    // unique_ptr<Rectangle> P2(P1);
    unique_ptr<Rectangle> P2;
    P2 = move(P1);

    // This'll print 50
    cout << P2->area() << endl;

    // cout<<P1->area()<<endl;
    return 0;
}

```

}

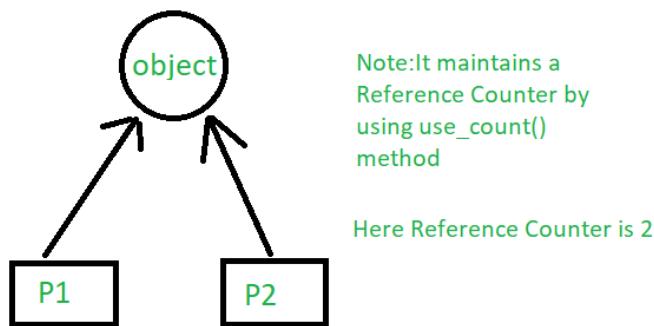
Output:

50

50

2. shared_ptr

By using *shared_ptr* more than one pointer can point to this one object at a time and it'll maintain a **Reference Counter** using *use_count()* method.



- C++14

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

#include <memory>

class Rectangle {
    int length;
    int breadth;

public:
    Rectangle(int l, int b)
```

```

{

    length = l;
    breadth = b;
}

int area()
{
    return length * breadth;
}

};

int main()
{
    shared_ptr<Rectangle> P1(new Rectangle(10, 5));
    // This'll print 50
    cout << P1->area() << endl;

    shared_ptr<Rectangle> P2;
    P2 = P1;

    // This'll print 50
    cout << P2->area() << endl;

    // This'll now not give an error,
    cout << P1->area() << endl;

    // This'll also print 50 now
    // This'll print 2 as Reference Counter is 2
    cout << P1.use_count() << endl;
}

```

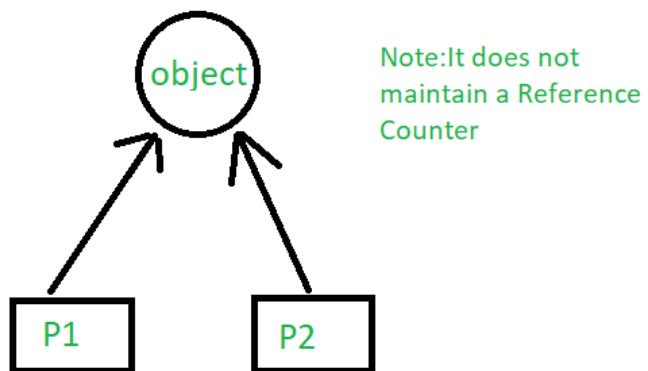
```
    return 0;  
}
```

Output:

```
50  
50  
50  
2
```

3. weak_ptr

It's much more similar to shared_ptr except it'll not maintain a **Reference Counter**. In this case, a pointer will not have a stronghold on the object. The reason is if suppose pointers are holding the object and requesting for other objects then they may form a **Deadlock**.



C++ libraries provide implementations of smart pointers in the form of [auto_ptr](#), [unique_ptr](#), [shared_ptr](#) and [weak_ptr](#)

16.6. Pointers vs References in C++

Prerequisite: [Pointers](#), [References](#)

C and C++ support pointers which are different from most of the other programming languages. Other languages including C++, Java, Python, Ruby, Perl and PHP support references.

On the surface, both references and pointers are very similar, both are used to have one variable provide access to another. With both providing lots of the same capabilities, it's often unclear what is different between these different mechanisms. In this article, I will try to illustrate the differences between pointers and references.

[Pointers](#): A pointer is a variable that holds memory address of another variable. A pointer needs to be dereferenced with * operator to access the memory location it points to.

[References](#) : A reference variable is an alias, that is, another name for an already existing variable. A reference, like a pointer, is also implemented by storing the address of an object.

A reference can be thought of as a constant pointer (not to be confused with a pointer to a constant value!) with automatic indirection, i.e the compiler will apply the * operator for you.

```
int i = 3;
```

```
// A pointer to variable i (or stores  
// address of i)  
int *ptr = &i;
```

```
// A reference (or alias) for i.  
int &ref = i;
```

Differences :

1. Initialization: A pointer can be initialized in this way:

```
int a = 10;  
int *p = &a;  
  
OR  
  
int *p;  
p = &a;
```

we can declare and initialize pointer at same step or in multiple line.

2. While in references,

```
int a=10;  
int &p=a; //it is correct  
but  
int &p;  
p=a; // it is incorrect as we should declare and initialize  
references at single step.
```

3. NOTE: This differences may vary from compiler to compiler. The above differences is with respect to turbo IDE.

4. Reassignment: A pointer can be re-assigned. This property is useful for implementation of data structures like linked list, tree, etc. See the following examples:

```
int a = 5;  
int b = 6;  
int *p;  
p = &a;  
p = &b;
```

5. On the other hand, a reference cannot be re-assigned, and must be assigned at initialization.

```
int a = 5;  
int b = 6;  
int &p = a;  
int &p = b; //At this line it will show error as "multiple  
declaration is not allowed".
```

However it is valid statement,

```
int &q=p;
```

6. Memory Address: A pointer has its own memory address and size on the stack whereas a reference shares the same memory address (with the original variable) but also takes up some space on the stack.

```
int &p = a;  
cout << &p << endl << &a;
```

7. NULL value: Pointer can be assigned NULL directly, whereas reference cannot. The constraints associated with references (no NULL, no reassignment) ensure that the underlying operations do not run into exception situation.

8. Indirection: You can have pointers to pointers offering extra levels of indirection. Whereas references only offer one level of indirection.I.e,
In Pointers,

```
int a = 10;  
int *p;  
int **q; //it is valid.  
p = &a;  
q = &p;
```

Whereas in references,

```
int &p = a;  
int &&q = p; //it is reference to reference, so it is an error.
```

9. Arithmetic operations: Various arithmetic operations can be performed on pointers whereas there is no such thing called Reference Arithmetic.(but you can take the address of an object pointed by a reference and do pointer arithmetics on it as in `&obj + 5`.)

When to use What

The performances are exactly the same, as references are implemented internally as pointers. But still you can keep some points in your mind to decide when to use what :

- Use references
 - In function parameters and return types.
- Use pointers:
 - Use pointers if pointer arithmetic or passing NULL-pointer is needed. For example for arrays (Note that array access is implemented using pointer arithmetic).
 - To implement data structures like linked list, tree, etc and their algorithms because to point different cell, we have to use the concept of pointers.

[Quoted in C++ FAQ Lite](#) : Use references when you can, and pointers when you have to. References are usually preferred over pointers whenever you don't need "reseating". This usually means that references are most useful in a class's public interface. References typically appear on the skin of an object, and pointers on the inside.

The exception to the above is where a function's parameter or return value needs a "sentinel" reference — a reference that does not refer to an object. This is usually best done by returning/taking a pointer, and giving the NULL pointer this special significance (references must always alias objects, not a dereferenced null pointer).

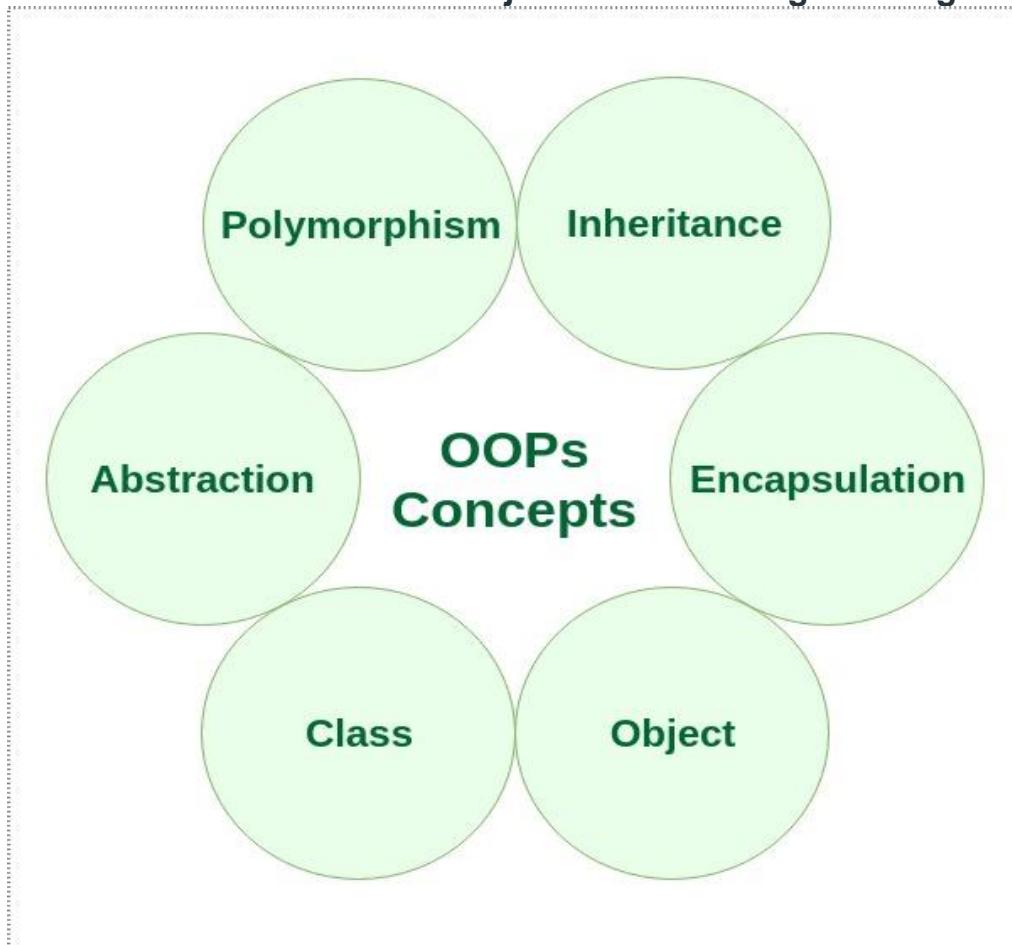
17. Object Oriented Programming in C++

TABLE OF CONTENT:

1. [Introduction](#)
2. [Class](#)
3. [Objects](#)
4. [Encapsulation](#)
5. [Abstraction](#)
6. [Polymorphism](#)
7. [Inheritance](#)
8. [Dynamic Binding](#)
9. [Message Passing](#)

Object-oriented programming – As the name suggests uses objects in programming. Object-oriented programming aims to implement real-world entities like inheritance, hiding, polymorphism, etc in programming. The main aim of OOP is to bind together the data and the functions that operate on them so that no other part of the code can access this data except that function.

Characteristics of an Object Oriented Programming language



17.1. **Class**: The building block of C++ that leads to Object-Oriented programming is a Class. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A class is like a blueprint for an object. For Example: Consider the Class of Cars. There may be many cars with different names and brand but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user-defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions define the properties and behaviour of the objects in a Class.
- In the above example of class Car, the data member will be speed limit, mileage etc and member functions can apply brakes, increase speed etc.

We can say that a **Class in C++** is a blue-print representing a group of objects which shares some common properties and behaviours.

17.2. **Object**: An Object is an identifiable entity with some characteristics and behaviour. An Object is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

```
class person
{
    char name[20];
    int id;

public:
    void getdetails(){}
};

int main()
{
    person p1; // p1 is a object
}
```

Object take up space in memory and have an associated address like a record in pascal or structure or union in C.

When a program is executed the objects interact by sending messages to one another.

Each object contains data and code to manipulate the data. Objects can interact without having to know details of each other's data or code, it is sufficient to know the type of message accepted and type of response returned by the objects.

17.3. **Encapsulation**: In normal terms, Encapsulation is defined as wrapping up of data and information under a single unit. In Object-Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulate them.

Consider a real-life example of encapsulation, in a company, there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keeps records of all the data related to finance. Similarly, the sales section handles all the sales-related activities and keeps records of all the sales. Now there may arise a situation when for some reason an official from the finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of the sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of the sales section and the employees that can manipulate them are wrapped under a single name "sales section".

Encapsulation in C++



Encapsulation in C++

Encapsulation also leads to *data abstraction or hiding*. As using encapsulation also hides the data. In the above example, the data of any of the section like sales, finance or accounts are hidden from any other section.

17.4. **Abstraction**: Data abstraction is one of the most essential and important features of object-oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real-life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of the car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the

inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

- *Abstraction using Classes:* We can implement Abstraction in C++ using classes. The class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to the outside world and which is not.
- *Abstraction in Header files:* One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate the power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating the power of numbers.

17.5. **Polymorphism:** The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form.

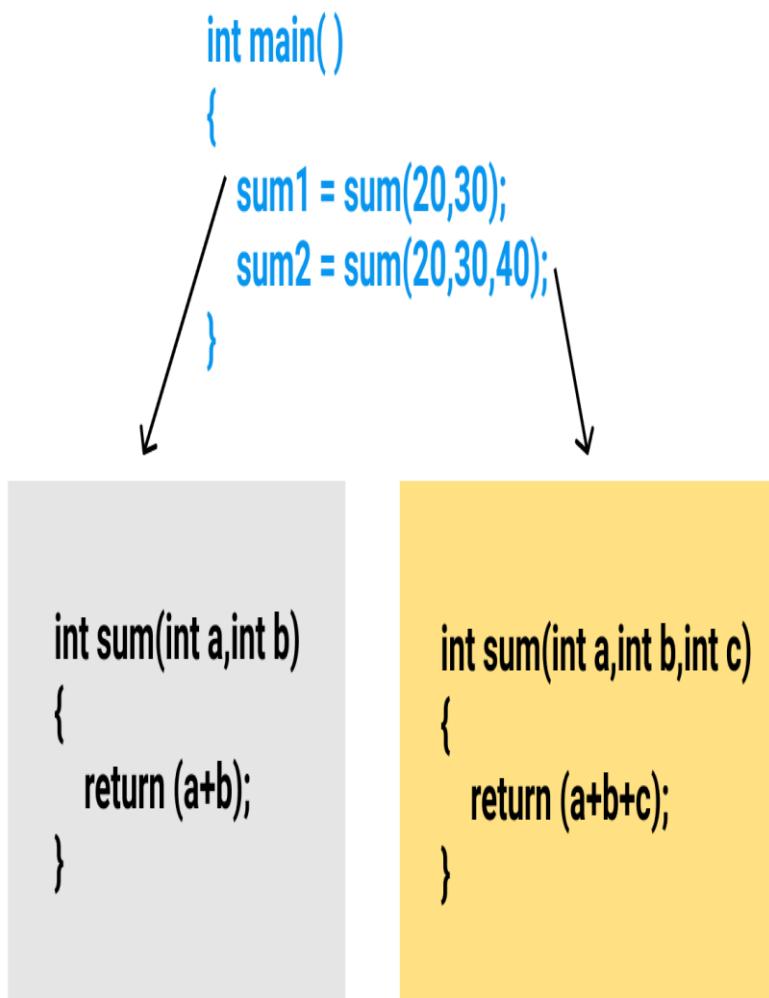
A person at the same time can have different characteristic. Like a man at the same time is a father, a husband, an employee. So the same person posses different behaviour in different situations. This is called polymorphism.

An operation may exhibit different behaviours in different instances. The behaviour depends upon the types of data used in the operation.

C++ supports operator overloading and function overloading.

- *Operator Overloading:* The process of making an operator to exhibit different behaviours in different instances is known as operator overloading.
- *Function Overloading:* Function overloading is using a single function name to perform different types of tasks.
Polymorphism is extensively used in implementing inheritance.

Example: Suppose we have to write a function to add some integers, sometimes there are 2 integers, sometimes there are 3 integers. We can write the Addition Method with the same name having different parameters, the concerned method will be called according to parameters.

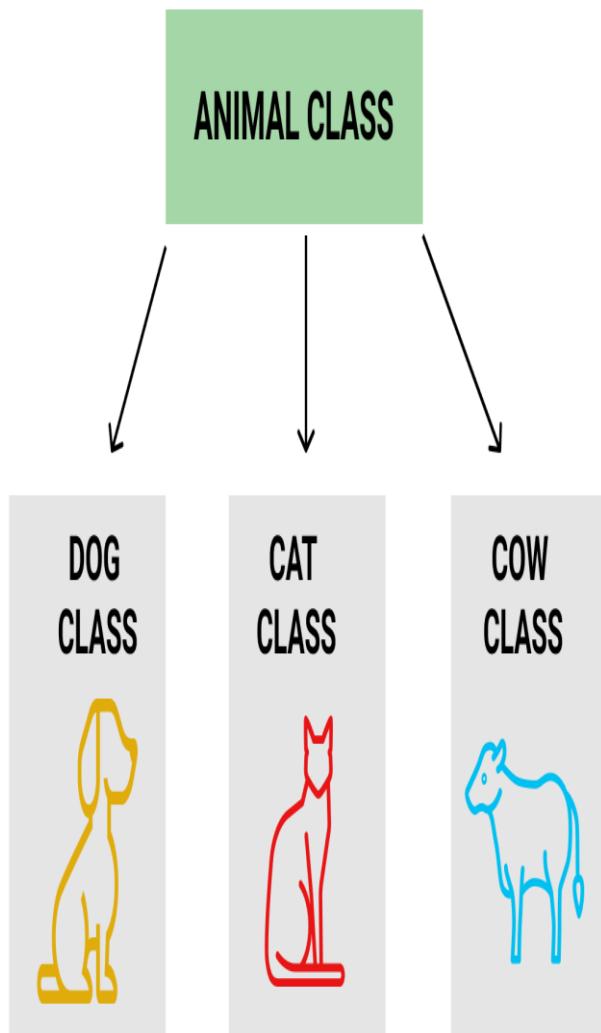


17.6. [Inheritance](#): The capability of a class to derive properties and characteristics from another class is called Inheritance. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class:** The class that inherits properties from another class is called Sub class or Derived Class.
- **Super Class:** The class whose properties are inherited by sub class is called Base Class or Super class.
- **Reusability:** Inheritance supports the concept of “reusability”, i.e. when we want to create a new class and there is already a class

that includes some of the code that we want, we can derive our new class from the existing class. By doing this, we are reusing the fields and methods of the existing class.

Example: Dog, Cat, Cow can be Derived Class of Animal Base Class.



17.7.Dynamic Binding: In dynamic binding, the code to be executed in response to function call is decided at runtime. C++ has [virtual functions](#) to support this.

17.8.Message Passing: Objects communicate with one another by sending and receiving information to each other. A message for an object is a request for execution of a procedure and therefore will invoke a function in the receiving object that generates the desired results. Message passing involves specifying the name of the object, the name of the function and the information to be sent.

18.C++ Classes and Objects

Class: A class in C++ is the building block that leads to Object-Oriented programming. It is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class. A C++ class is like a blueprint for an object.

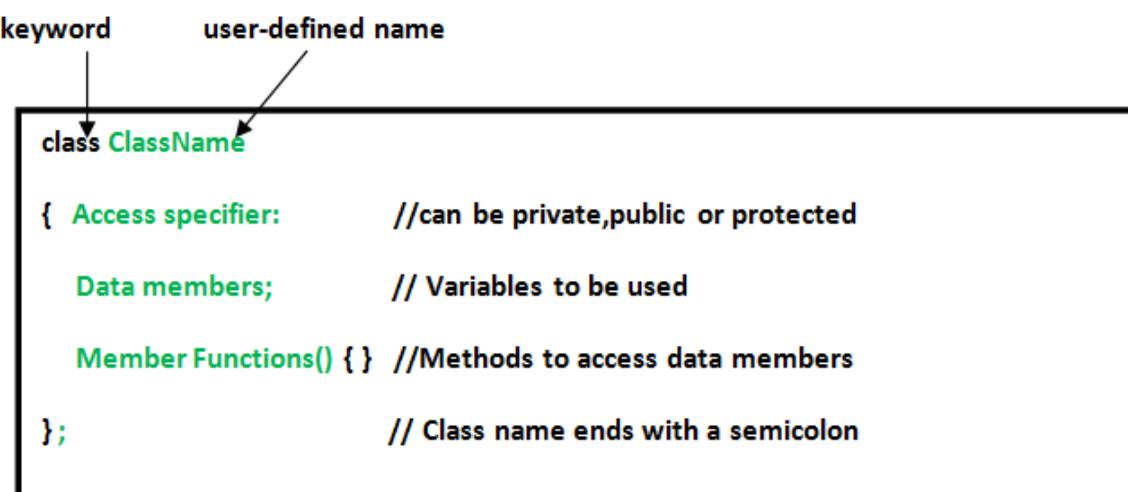
For Example: Consider the Class of **Cars**. There may be many cars with different names and brand but all of them will share some common properties like all of them will have *4 wheels*, *Speed Limit*, *Mileage range* etc. So here, Car is the class and wheels, speed limits, mileage are their properties.

- A Class is a user defined data-type which has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables and together these data members and member functions defines the properties and behavior of the objects in a Class.
- In the above example of class *Car*, the data member will be *speed limit*, *mileage* etc and member functions can be *apply brakes*, *increase speed* etc.

An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

Defining Class and Declaring Objects

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.



Declaring Objects: When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax:

```
ClassName ObjectName;
```

Accessing data members and member functions: The data members and member functions of class can be accessed using the dot(‘.’) operator with the object. For example if the name of object is *obj* and you want to access the member function with the name *printName()* then you will have to write *obj.printName()* .

Accessing Data Members

The public data members are also accessed in the same way given however the private data members are not allowed to be accessed directly by the object. Accessing a data member depends solely on the access control of that data member.

This access control is given by [Access modifiers in C++](#). There are three access modifiers : **public, private and protected**.

```
// C++ program to demonstrate  
// accessing of data members  
  
#include <bits/stdc++.h>  
  
using namespace std;  
  
class Geeks  
{  
    // Access specifier  
public:  
  
    // Data Members  
    string geekname;  
  
    // Member Functions()  
    void printname()  
    {  
        cout << "Geekname is: " << geekname;  
    }  
};
```

```

int main() {

    // Declare an object of class geeks
    Geeks obj1;

    // accessing data member
    obj1.geekname = "Abhi";

    // accessing member function
    obj1.printname();
    return 0;
}

```

Output:

Geekname is: Abhi

18.1.Member Functions in Classes

There are 2 ways to define a member function:

- Inside class definition
- Outside class definition

To define a member function outside the class definition we have to use the scope resolution :: operator along with class name and function name.

```

// C++ program to demonstrate function

// declaration outside class

#include <bits/stdc++.h>

using namespace std;

class Geeks

{
public:
    string geekname;
    int id;
}

```

```

// printname is not defined inside class definition

void printname();

// printid is defined inside class definition

void printid()
{
    cout << "Geek id is: " << id;
}

};

// Definition of printname using scope resolution operator ::

void Geeks::printname()
{
    cout << "Geekname is: " << geekname;
}

int main() {

    Geeks obj1;

    obj1.geekname = "xyz";
    obj1.id=15;

    // call printname()
    obj1.printname();
    cout << endl;

    // call printid()
    obj1.printid();

    return 0;
}

```

Output:

Geekname is: xyz

Geek id is: 15

Note that all the member functions defined inside the class definition are by default **inline**, but you can also make any non-class function inline by using keyword **inline** with them. Inline functions are actual functions, which are copied everywhere during compilation, like pre-processor macro, so the overhead of function calling is reduced.

Note: Declaring a [friend function](#) is a way to give private access to a non-member function.

19. Constructors in C++

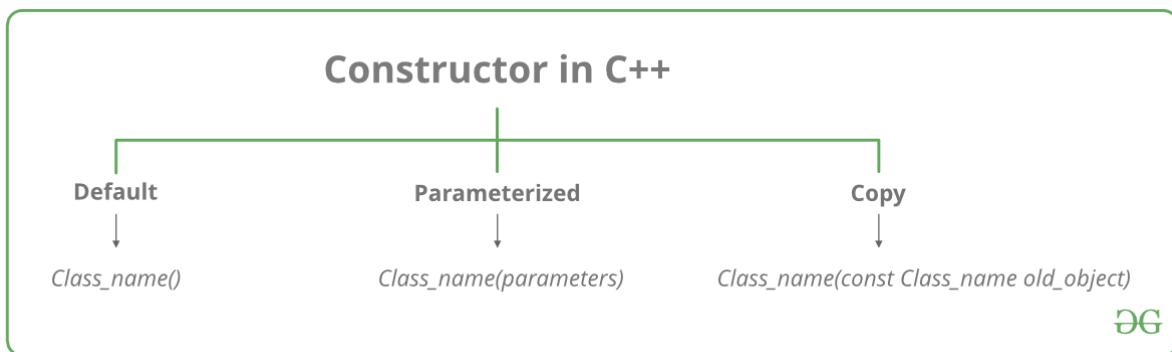
What is constructor?

A constructor is a special type of member function of a class which initializes objects of a class. In C++, Constructor is automatically called when object(instance of class) is created. It is special member function of the class because it does not have any return type.

How constructors are different from a normal member function?

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Default Constructors don't have input argument however, Copy and Parameterized Constructors have input arguments
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- It must be placed in public section of class.
- If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).



Let us understand the types of constructors in C++ by taking a real-world example. Suppose you went to a shop to buy a marker. When you want to buy a marker, what are the options? The first one you go to a shop and say give me a marker. So just saying give me a marker mean that you did not set which brand name and which color, you didn't mention anything just say you want a marker. So when we said just I want a marker so whatever the frequently sold marker is there in the market or in his shop he will simply hand over that. And this is what a default constructor is! The second method you go to a shop and say I want a marker a red in color and XYZ brand. So you are mentioning this and he will give you that marker. So in this case you have given the parameters. And this is what a parameterized constructor is! Then the third one you go to a shop and say I want a marker like this(a physical marker on your hand). So the shopkeeper will see that marker. Okay, and he will give a new marker for you. So copy of that marker. And that's what copy constructor is!

Types of Constructors

19.1. Default Constructors: Default constructor is the constructor which doesn't take any argument. It has no parameters.

- CPP

```
// Cpp program to illustrate the
// concept of Constructors

#include <iostream>

using namespace std;

class construct
{
public:
    int a, b;

    // Default Constructor
    construct()
    {
        a = 10;
        b = 20;
    }

};

int main()
{
    // Default constructor called automatically
    // when the object is created
    construct c;
    cout << "a: " << c.a << endl
        << "b: " << c.b;
    return 1;
}
```

Output:

a: 10

b: 20

Note: Even if we do not define any constructor explicitly, the compiler will automatically provide a default constructor implicitly.

19.2. Parameterized Constructors: It is possible to pass arguments to constructors. Typically, these arguments help initialize an object when it is created. To create a parameterized constructor, simply add parameters to it the way you would to any other function. When you define the constructor's body, use the parameters to initialize the object.

- CPP

```
// CPP program to illustrate  
// parameterized constructors  
  
#include <iostream>  
  
using namespace std;  
  
  
class Point  
{  
  
private:  
    int x, y;  
  
  
public:  
    // Parameterized Constructor  
    Point(int x1, int y1)  
    {  
        x = x1;  
        y = y1;  
    }  
  
  
    int getX()  
    {  
        return x;  
    }
```

```

    }

    int getY()
    {
        return y;
    }

};

int main()
{
    // Constructor called
    Point p1(10, 15);

    // Access values assigned by constructor
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();

    return 0;
}

```

Output:

p1.x = 10, p1.y = 15

When an object is declared in a parameterized constructor, the initial values have to be passed as arguments to the constructor function. The normal way of object declaration may not work. The constructors can be called explicitly or implicitly.

Example e = Example(0, 50); // Explicit call

Example e(0, 50); // Implicit call

- **Uses of Parameterized constructor:**

0. It is used to initialize the various data elements of different objects with different values when they are created.
1. It is used to overload constructors.

- **Can we have more than one constructor in a class?**

Yes, It is called Constructor Overloading.

19.3. Copy Constructor: A copy constructor is a member function which initializes an object using another object of the same class. Detailed article on [Copy Constructor](#).

Whenever we define one or more non-default constructors(with parameters) for a class, a default constructor(without parameters) should also be explicitly defined as the compiler will not provide a default constructor in this case. However, it is not necessary but it's considered to be the best practice to always define a default constructor.

- CPP

```
// Illustration

#include <iostream>

using namespace std;

class point
{
private:
    double x, y;

public:
    // Non-default Constructor &
    // default Constructor
    point (double px, double py)
    {
        x = px, y = py;
    }

};

int main(void)
{
    // Define an array of size
```

```
// 10 & of type point  
// This line will cause error  
point a[10];  
  
// Remove above line and program  
// will compile without error  
point b = point(5, 6);  
}
```

Output:

Error: point (double px, double py): expects 2 arguments, 0 provided

20.Destructors in C++

What is a destructor?

Destructor is an instance member function which is invoked automatically whenever an object is going to be destroyed. Meaning, a destructor is the last function that is going to be called before an object is destroyed.

The thing is to be noted here, if the object is created by using new or the constructor uses new to allocate memory which resides in the heap memory or the free store, the destructor should use delete to free the memory.

Syntax:

```
~constructor-name();
```

Properties of Destructor:

- Destructor function is automatically invoked when the objects are destroyed.
- It cannot be declared static or const.
- The destructor does not have arguments.
- It has no return type not even void.
- An object of a class with a Destructor cannot become a member of the union.
- A destructor should be declared in the public section of the class.
- The programmer cannot access the address of destructor.

When is destructor called?

A destructor function is called automatically when the object goes out of scope:

- (1) the function ends
- (2) the program ends
- (3) a block containing local variables ends
- (4) a delete operator is called

How are destructors different from a normal member function?

Destructors have same name as the class preceded by a tilde (~)

Destructors don't take any argument and don't return anything

- CPP

```
class String {  
  
private:  
    char* s;  
    int size;  
  
public:  
    String(char*); // constructor
```

```

~String(); // destructor
};

String::String(char* c)
{
    size = strlen(c);
    s = new char[size + 1];
    strcpy(s, c);
}
String::~String() { delete[] s; }

```

Can there be more than one destructor in a class?

No, there can only one destructor in a class with classname preceded by ~, no parameters and no return type.

When do we need to write a user-defined destructor?

If we do not write our own destructor in class, compiler creates a default destructor for us. The default destructor works fine unless we have dynamically allocated memory or pointer in class. When a class contains a pointer to memory allocated in class, we should write a destructor to release memory before the class instance is destroyed. This must be done to avoid memory leak.

Can a destructor be virtual?

Yes, In fact, it is always a good idea to make destructors virtual in base class when we have a virtual function. See [virtual destructor](#) for more details.

20.1. Private Destructor in C++

Destructors with the [access modifier](#) as private are known as Private Destructors. Whenever we want to prevent the destruction of an object, we can make the destructor private.

What is the use of private destructor?

Whenever we want to control the destruction of objects of a class, we make the destructor private. For dynamically created objects, it may happen that you pass a pointer to the object to a function and the function deletes the object. If the object is referred after the function call, the reference will become dangling.

Predict the Output of the Following Programs:

- CPP

```
// CPP program to illustrate  
// Private Destructor  
  
#include <iostream>  
  
using namespace std;  
  
  
class Test {  
private:  
    ~Test() {}  
};  
  
int main() {}
```

The above program compiles and runs fine. Hence, we can say that: It is **not** a compiler error to create private destructors.

Now, What do you say about the below program?

- CPP

```
// CPP program to illustrate  
// Private Destructor  
  
#include <iostream>  
  
using namespace std;  
  
  
class Test {
```

```
private:
    ~Test() {}

};

int main() { Test t; }
```

Output

```
prog.cpp: In function ‘int main()’:
prog.cpp:8:5: error: ‘Test::~Test()’ is private
    ~Test() {}
    ^
prog.cpp:10:19: error: within this context
int main() { Test t; }
```

The above program fails in the compilation. The compiler notices that the local variable ‘t’ cannot be destructed because the destructor is private.

Now, What about the Below Program?

- CPP

```
// CPP program to illustrate
// Private Destructor

#include <iostream>

using namespace std;

class Test {

private:
    ~Test() {}

};

int main() { Test* t; }
```

The above program works fine. There is no object being constructed, the program just creates a pointer of type “Test *”, so nothing is destructed.

Next, What about the below program?

- CPP

```
// CPP program to illustrate
```

```

// Private Destructor

#include <iostream>

using namespace std;

class Test {
private:
    ~Test() {}
};

int main() { Test* t = new Test; }

```

The above program also works fine. When something is created using dynamic memory allocation, it is the programmer's responsibility to delete it. So compiler doesn't bother.

In the case where the destructor is declared private, an instance of the class can also be created using the malloc() function. The same is implemented in the below program.

- CPP

```

// CPP program to illustrate

// Private Destructor


#include <bits/stdc++.h>

using namespace std;

class Test {
public:
    Test() // Constructor
    {
        cout << "Constructor called\n";
    }

private:

```

```

~Test() // Private Destructor
{
    cout << "Destructor called\n";
}
};

int main()
{
    Test* t = (Test*)malloc(sizeof(Test));
    return 0;
}

```

The above program also works fine. However, The below program fails in the compilation. When we call delete, destructor is called.

- CPP

```

// CPP program to illustrate
// Private Destructor

#include <iostream>

using namespace std;

class Test {
private:
    ~Test() {}
};

// Driver Code

int main()
{
    Test* t = new Test;
    delete t;
}

```

}

We noticed in the above programs when a class has a private destructor, only dynamic objects of that class can be created. Following is a way to **create classes with private destructors and have a function as a friend of the class**. The function can only delete the objects.

- CPP

```
// CPP program to illustrate  
// Private Destructor  
  
#include <iostream>  
  
// A class with private destructor  
  
class Test {  
  
private:  
    ~Test() {}  
  
public:  
    friend void destructTest(Test*);  
};  
  
// Only this function can destruct objects of Test  
void destructTest(Test* ptr) { delete ptr; }  
  
  
int main()  
{  
    // create an object  
    Test* ptr = new Test;  
    // destruct the object  
    destructTest(ptr);  
    return 0;  
}
```

21. Inheritance in C++

The capability of a [class](#) to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important features of Object-Oriented Programming.

- **Sub Class:** The class that inherits properties from another class is called Subclass or Derived Class.
- **Super Class:** The class whose properties are inherited by a subclass is called Base Class or Superclass.

The article is divided into the following subtopics:

- Why and when to use inheritance?
- Modes of Inheritance
- Types of Inheritance

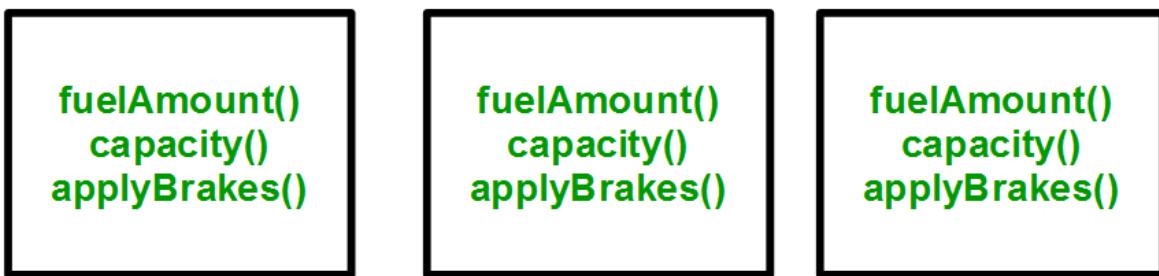
Why and when to use inheritance?

Consider a group of vehicles. You need to create classes for Bus, Car, and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be the same for all three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown below figure:

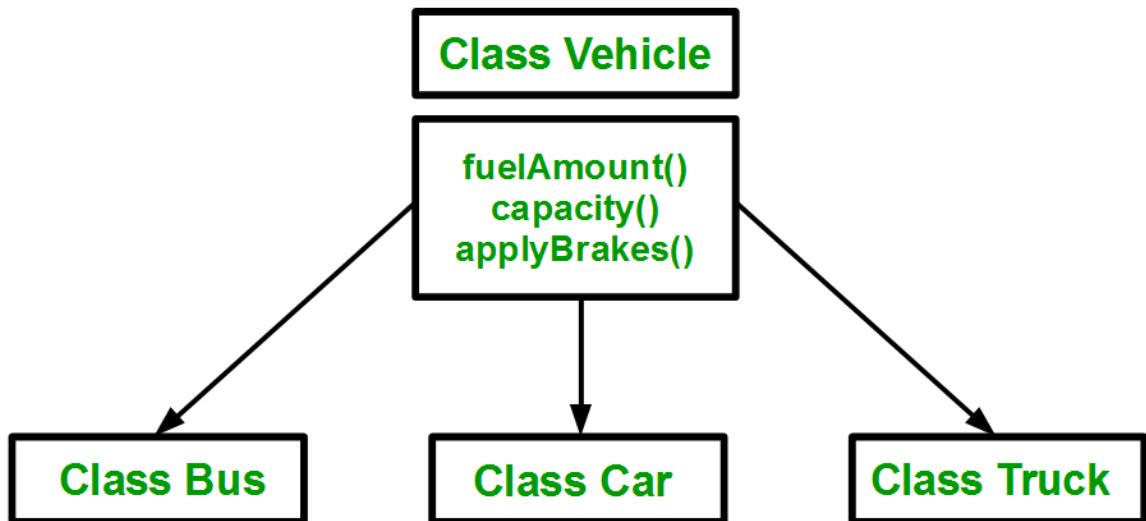
Class Bus

Class Car

Class Truck



You can clearly see that the above process results in duplication of the same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Look at the below diagram in which the three classes are inherited from vehicle class:



Using inheritance, we have to write the functions only one time instead of three times as we have inherited the rest of the three classes from the base class (Vehicle).

Implementing inheritance in C++: For creating a sub-class that is inherited from the base class we have to follow the below syntax.

Syntax:

```

class subclass_name : access_mode base_class_name
{
    // body of subclass
};
  
```

Here, **subclass_name** is the name of the subclass, **access mode** is the mode in which you want to inherit the subclass for example public, private, etc. and **base_class_name** is the name of the base class from which you want to inherit the subclass.

Note: A derived class doesn't inherit **access** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

- CPP

```

// C++ program to demonstrate implementation
// of Inheritance
  
```

```

#include <bits/stdc++.h>
using namespace std;
  
```

```

// Base class
  
```

```

class Parent {
public:
    int id_p;
};

// Sub class inheriting from Base Class(Parent)
class Child : public Parent {
public:
    int id_c;
};

// main function
int main()
{
    Child obj1;

    // An object of class child has all data members
    // and member functions of class parent
    obj1.id_c = 7;
    obj1.id_p = 91;
    cout << "Child id is: " << obj1.id_c << '\n';
    cout << "Parent id is: " << obj1.id_p << '\n';

    return 0;
}

```

Output:

Child id is: 7

Parent id is: 91

In the above program, the ‘Child’ class is publicly inherited from the ‘Parent’ class so the public data members of the class ‘Parent’ will also be inherited by the class ‘Child’.

Modes of Inheritance: There are 3 modes of inheritance.

1. **Public Mode:** If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.
2. **Protected Mode:** If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.
3. **Private Mode:** If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.

Note: The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed. For example, Classes B, C, and D all contain the variables x, y, and z in the below example. It is just a question of access.

- CPP

```
// C++ Implementation to show that a derived class
// doesn't inherit access to private data members.
// However, it does inherit a full parent object.

class A {
public:
    int x;

protected:
    int y;

private:
    int z;
};

class B : public A {
    // x is public
    // y is protected
    // z is not accessible from B
};
```

```

class C : protected A {

    // x is protected
    // y is protected
    // z is not accessible from C

};

class D : private A // 'private' is default for classes

{
    // x is private
    // y is private
    // z is not accessible from D

};

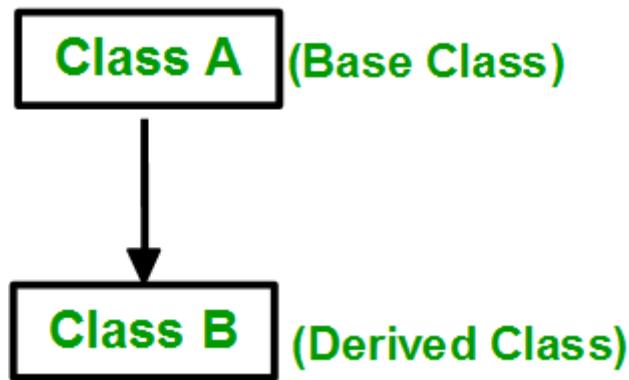
```

The below table summarizes the above three modes and shows the access specifier of the members of the base class in the subclass when derived in public, protected and private modes:

Base class member access specifier	Type of Inheritance		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

Types of Inheritance in C++

21.1. Single Inheritance: In single inheritance, a class is allowed to inherit from only one class. i.e. one subclass is inherited by one base class only.



Syntax:

```

class subclass_name : access_mode base_class
{
    // body of subclass
};

```

- CPP

```

// C++ program to explain
// Single inheritance

#include<iostream>
using namespace std;

// base class
class Vehicle {
public:
    Vehicle()
    {
        cout << "This is a Vehicle\n";
    }
};


```

```

// sub class derived from a single base classes
class Car : public Vehicle {

```

```

};

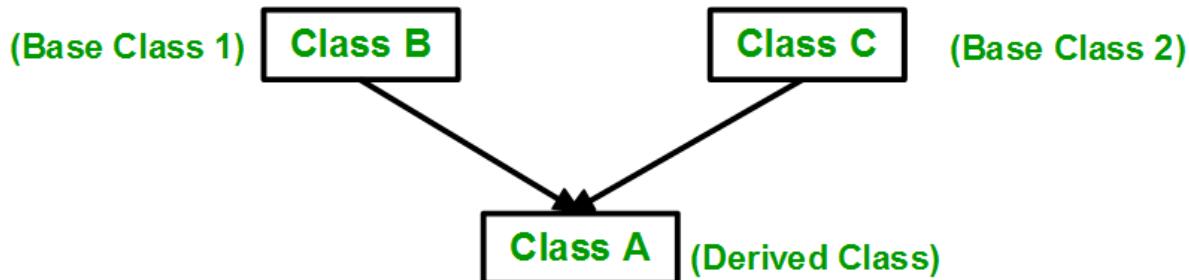
// main function
int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes
    Car obj;
    return 0;
}

```

Output

This is a Vehicle

21.2. Multiple Inheritance: Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one **subclass** is inherited from more than one **base class**.



Syntax:

```

class subclass_name : access_mode base_class1, access_mode
base_class2, ....
{
    // body of subclass
}

```

Here, the number of base classes will be separated by a comma (‘,’) and the access mode for every base class must be specified.

- CPP

```

// C++ program to explain
// multiple inheritance

```

```

#include <iostream>

using namespace std;

// first base class

class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }

};

// second base class

class FourWheeler {
public:
    FourWheeler()
    {
        cout << "This is a 4 wheeler Vehicle\n";
    }
};

// sub class derived from two base classes

class Car : public Vehicle, public FourWheeler {
};

// main function

int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes.

    Car obj;
    return 0;
}

```

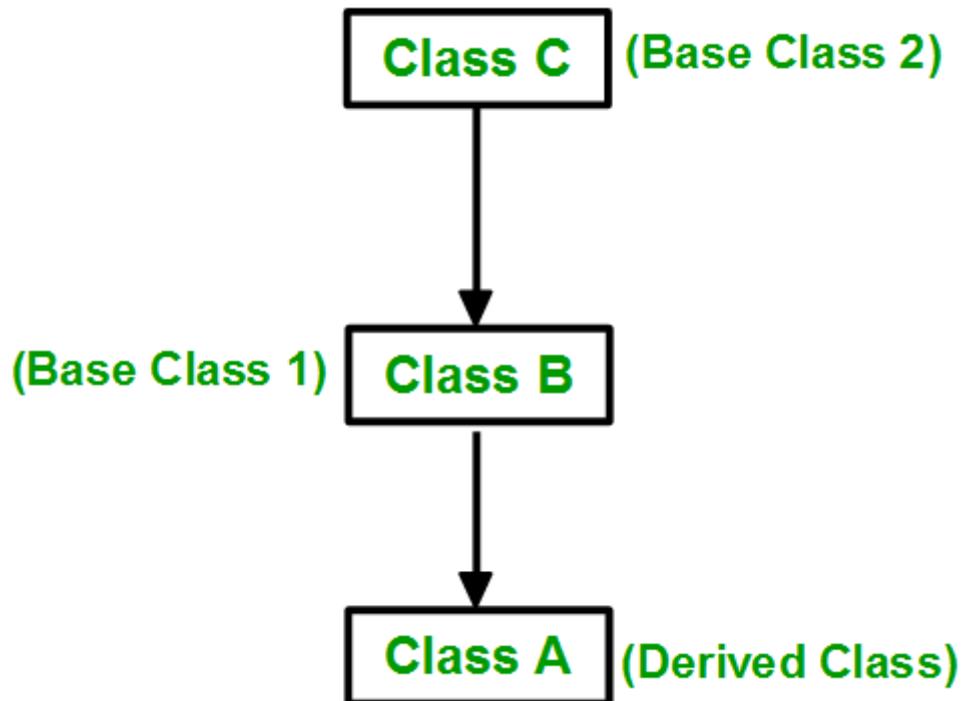
Output

This is a Vehicle

This is a 4 wheeler Vehicle

To know more about it, please refer to the article [Multiple Inheritances](#).

21.3. Multilevel Inheritance: In this type of inheritance, a derived class is created from another derived class.



- CPP

```
// C++ program to implement  
// Multilevel Inheritance  
  
#include <iostream>  
  
using namespace std;  
  
  
// base class  
class Vehicle {  
public:  
    Vehicle() { cout << "This is a Vehicle\n"; }  
};
```

```

// first sub_class derived from class vehicle

class fourWheeler : public Vehicle {

public:
    fourWheeler()
    {
        cout << "Objects with 4 wheels are vehicles\n";
    }
};

// sub class derived from the derived base class fourWheeler

class Car : public fourWheeler {

public:
    Car() { cout << "Car has 4 Wheels\n"; }

};

// main function

int main()
{
    // Creating object of sub class will
    // invoke the constructor of base classes.
    Car obj;

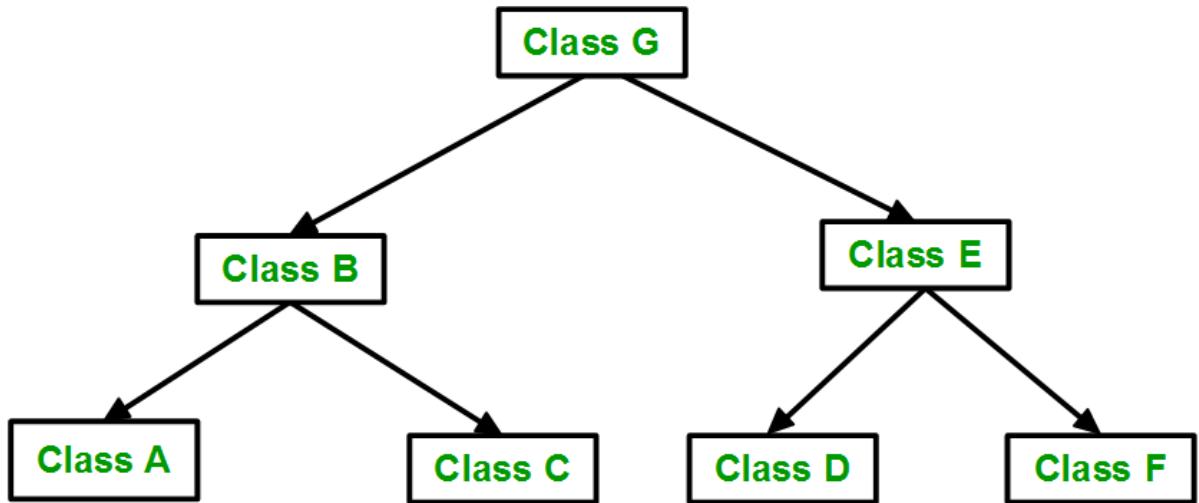
    return 0;
}

```

Output

This is a Vehicle
 Objects with 4 wheels are vehicles
 Car has 4 Wheels

21.4. Hierarchical Inheritance: In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.



- CPP

```

// C++ program to implement
// Hierarchical Inheritance

#include <iostream>

using namespace std;

// base class
class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {
};

// second sub class
class Bus : public Vehicle {
};

// main function

```

```

int main()
{
    // Creating object of sub class will
    // invoke the constructor of base class.

    Car obj1;
    Bus obj2;

    return 0;
}

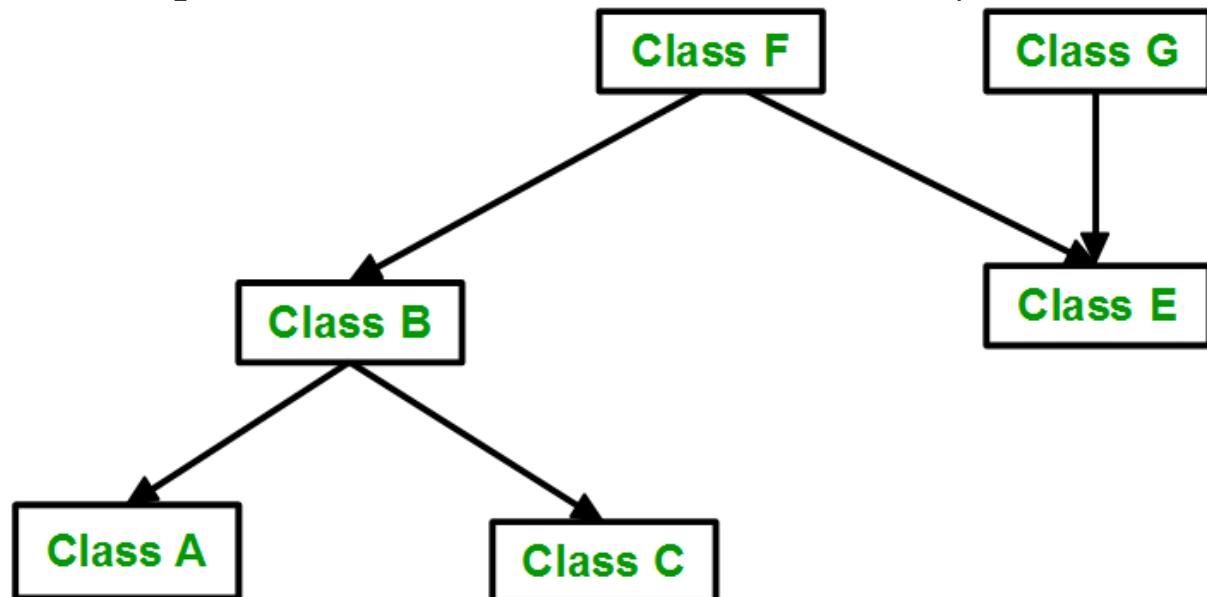
```

Output

This is a Vehicle

This is a Vehicle

21.5. Hybrid (Virtual) Inheritance: Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance. Below image shows the combination of hierarchical and multiple inheritances:



- CPP

```
// C++ program for Hybrid Inheritance
```

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

```

// base class

class Vehicle {
public:
    Vehicle() { cout << "This is a Vehicle\n"; }

};

// base class

class Fare {
public:
    Fare() { cout << "Fare of Vehicle\n"; }

};

// first sub class

class Car : public Vehicle {

};

// second sub class

class Bus : public Vehicle, public Fare {

};

// main function

int main()
{
    // Creating object of sub class will
    // invoke the constructor of base class.

    Bus obj2;

    return 0;
}

```

Output

This is a Vehicle

Fare of Vehicle

21.6. A special case of hybrid inheritance: Multipath inheritance:
A derived class with two base classes and these two base classes have one common base class is called multipath inheritance. Ambiguity can arise in this type of inheritance.

Example:

- CPP

```
// C++ program demonstrating ambiguity in Multipath  
// Inheritance  
  
#include <iostream>  
  
using namespace std;  
  
class ClassA {  
  
public:  
    int a;  
};  
  
class ClassB : public ClassA {  
  
public:  
    int b;  
};  
  
class ClassC : public ClassA {  
  
public:  
    int c;  
};  
  
class ClassD : public ClassB, public ClassC {  
  
public:
```

```

    int d;

};

int main()
{
    ClassD obj;

    // obj.a = 10;                                // Statement 1, Error
    // obj.a = 100;                               // Statement 2, Error

    obj.ClassB::a = 10; // Statement 3
    obj.ClassC::a = 100; // Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << " a from ClassB : " << obj.ClassB::a;
    cout << "\n a from ClassC : " << obj.ClassC::a;

    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}

```

Output:

```

a from ClassB : 10
a from ClassC : 100
b : 20
c : 30
d : 40

```

In the above example, both ClassB and ClassC inherit ClassA, they both have a single copy of ClassA. However Class-D inherits both ClassB and ClassC, therefore Class-D has two copies of ClassA, one from ClassB and another from ClassC.

If we need to access the data member of ClassA through the object of Class-D, we must specify the path from which a will be accessed, whether it is from ClassB or ClassC, bcoz compiler can't differentiate between two copies of ClassA in Class-D.

21.7. There are 2 Ways to Avoid this Ambiguity:

1) Avoiding ambiguity using the scope resolution operator: Using the scope resolution operator we can manually specify the path from which data member a will be accessed, as shown in statements 3 and 4, in the above example.

- CPP

```
obj.ClassB::a = 10;           // Statement 3  
obj.ClassC::a = 100;          // Statement 4
```

Note: Still, there are two copies of ClassA in Class-D.

2) Avoiding ambiguity using the virtual base class:

- CPP

```
#include<iostream>  
  
class ClassA  
{  
    public:  
        int a;  
};  
  
class ClassB : virtual public ClassA  
{  
    public:  
        int b;  
};
```

```

class ClassC : virtual public ClassA
{
    public:
        int c;
};

class ClassD : public ClassB, public ClassC
{
    public:
        int d;
};

int main()
{
    ClassD obj;

    obj.a = 10;           // Statement 3
    obj.a = 100;          // Statement 4

    obj.b = 20;
    obj.c = 30;
    obj.d = 40;

    cout << "\n a : " << obj.a;
    cout << "\n b : " << obj.b;
    cout << "\n c : " << obj.c;
    cout << "\n d : " << obj.d << '\n';
}

```

Output:

a : 100

b : 20

c : 30

d : 40

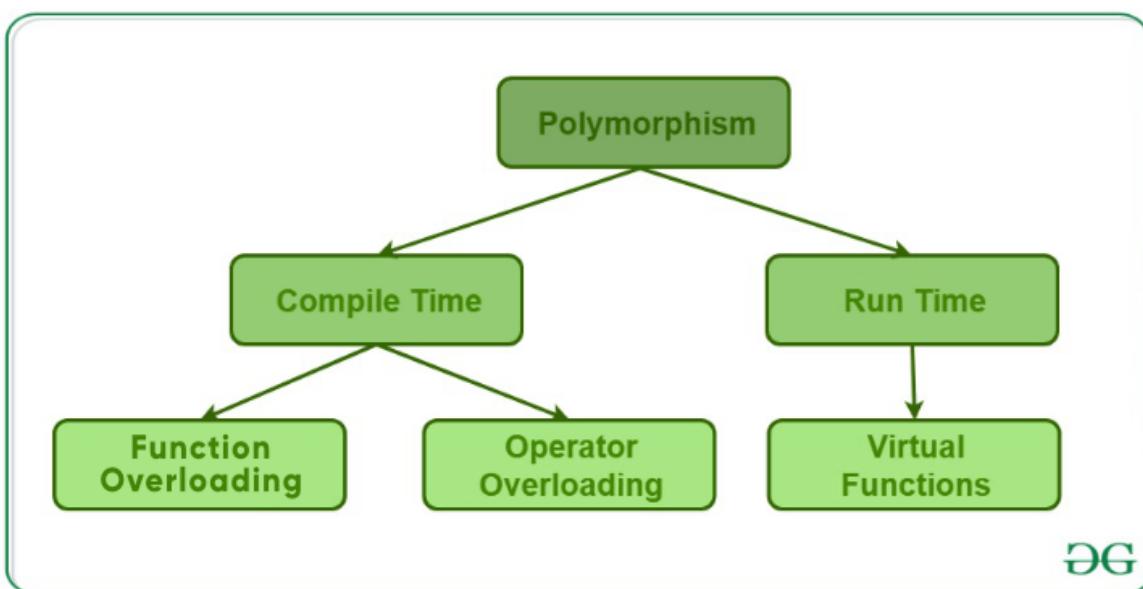
According to the above example, Class-D has only one copy of ClassA, therefore, statement 4 will overwrite the value of a, given in statement 3.

22. Polymorphism in C++

The word polymorphism means having many forms. In simple words, we can define polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism, a person at the same time can have different characteristics. Like a man at the same time is a father, a husband, an employee. So the same person posses different behavior in different situations. This is called polymorphism. Polymorphism is considered as one of the important features of Object Oriented Programming.

In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism



Types of Polymorphism

22.1. Compile time polymorphism: This type of polymorphism is achieved by function overloading or operator overloading.

- Function Overloading: When there are multiple functions with same name but different parameters then these functions are said to be **overloaded**. Functions can be overloaded by **change in number of arguments** or/and **change in type of arguments**.

Rules of Function Overloading

```
// C++ program for function overloading  
#include <bits/stdc++.h>
```

```

using namespace std;

class Geeks

{

public:

    // function with 1 int parameter

    void func(int x)

    {

        cout << "value of x is " << x << endl;

    }

    // function with same name but 1 double parameter

    void func(double x)

    {

        cout << "value of x is " << x << endl;

    }

    // function with same name and 2 int parameters

    void func(int x, int y)

    {

        cout << "value of x and y is " << x << ", " << y << endl;

    }

};

int main() {

    Geeks obj1;

    // Which function is called will depend on the parameters passed
}

```

```

// The first 'func' is called
obj1.func(7);

// The second 'func' is called
obj1.func(9.132);

// The third 'func' is called
obj1.func(85,64);
return 0;
}

```

- **Output:**
- value of x is 7
- value of x is 9.132
- value of x and y is 85, 64
- In the above example, a single function named *func* acts differently in three different situations which is the property of polymorphism.
- **Operator Overloading:** C++ also provide option to overload operators. For example, we can make the operator ('+') for string class to concatenate two strings. We know that this is the addition operator whose task is to add two operands. So a single operator '+' when placed between integer operands, adds them and when placed between string operands, concatenates them.

Example:

```

// CPP program to illustrate
// Operator Overloading
#include<iostream>
using namespace std;

class Complex {
private:
    int real, imag;

```

```

public:

    Complex(int r = 0, int i = 0) {real = r; imag = i;}

    // This is automatically called when '+' is used with
    // between two Complex objects

    Complex operator + (Complex const &obj) {

        Complex res;

        res.real = real + obj.real;
        res.imag = imag + obj.imag;

        return res;
    }

    void print() { cout << real << " + i" << imag << endl; }

};

int main()
{
    Complex c1(10, 5), c2(2, 4);

    Complex c3 = c1 + c2; // An example call to "operator+"

    c3.print();
}

```

- Output:
- 12 + i9
- In the above example the operator ‘+’ is overloaded. The operator ‘+’ is an addition operator and can add two numbers(integers or floating point) but here the operator is made to perform addition of two imaginary or complex numbers. To learn operator overloading in details visit [this](#) link.

22.2. Runtime polymorphism: This type of polymorphism is achieved by Function Overriding.

- [Function overriding](#) on the other hand occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be **overridden**.

```
// C++ program for function overriding
```

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
class base
```

```
{
```

```
public:
```

```
    virtual void print ()
```

```
    { cout<< "print base class" << endl; }
```

```
    void show ()
```

```
    { cout<< "show base class" << endl; }
```

```
};
```

```
class derived:public base
```

```
{
```

```
public:
```

```
    void print () //print () is already virtual function in derived  
    could also declared as virtual void print () explicitly
```

```
    { cout<< "print derived class" << endl; }
```

```
    void show ()
```

```
    { cout<< "show derived class" << endl; }
```

```
};
```

```
//main function
```

```
int main()
```

```
{
```

```
    base *bptr;
```

```
    derived d;  
    bptr = &d;  
  
    //virtual function, binded at runtime (Runtime polymorphism)  
    bptr->print();  
  
    // Non-virtual function, binded at compile time  
    bptr->show();  
  
    return 0;  
}
```

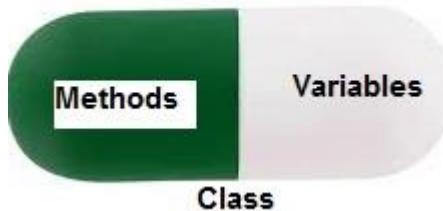
- Output:
- print derived class
- show base class

23. Encapsulation in C++

In normal terms **Encapsulation** is defined as wrapping up of data and information under a single unit. In Object Oriented Programming, Encapsulation is defined as binding together the data and the functions that manipulates them.

Consider a real life example of encapsulation, in a company there are different sections like the accounts section, finance section, sales section etc. The finance section handles all the financial transactions and keep records of all the data related to finance. Similarly the sales section handles all the sales related activities and keep records of all the sales. Now there may arise a situation when for some reason an official from finance section needs all the data about sales in a particular month. In this case, he is not allowed to directly access the data of sales section. He will first have to contact some other officer in the sales section and then request him to give the particular data. This is what encapsulation is. Here the data of sales section and the employees that can manipulate them are wrapped under a single name “sales section”.

Encapsulation in C++



Encapsulation also lead to data abstraction or hiding. As using encapsulation also hides the data. In the above example the data of any of the section like sales, finance or accounts is hidden from any other section.

In C++ encapsulation can be implemented using Class and access modifiers. Look at the below program:

```
// c++ program to explain  
// Encapsulation
```

```
#include<iostream>  
  
using namespace std;  
  
  
class Encapsulation  
{
```

```

private:

    // data hidden from outside world

    int x;

public:

    // function to set value of

    // variable x

    void set(int a)

    {

        x =a;

    }

    // function to return value of

    // variable x

    int get()

    {

        return x;

    }

};

// main function

int main()

{

    Encapsulation obj;

    obj.set(5);

    cout<<obj.get();

    return 0;

}

```

output:

5

In the above program the variable **x** is made private. This variable can be accessed and manipulated only using the functions `get()` and `set()` which are present inside the class. Thus we can say that here, the variable **x** and the functions `get()` and `set()` are binded together which is nothing but encapsulation.

23.1.Role of access specifiers in encapsulation

As we have seen in above example, access specifiers plays an important role in implementing encapsulation in C++. The process of implementing encapsulation can be sub-divided into two steps:

1. The data members should be labeled as private using the **private** access specifiers
2. The member function which manipulates the data members should be labeled as public using the **public** access specifier

24. Abstraction in C++

Data abstraction is one of the most essential and important feature of object oriented programming in C++. Abstraction means displaying only essential information and hiding the details. Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

Consider a real life example of a man driving a car. The man only knows that pressing the accelerators will increase the speed of car or applying brakes will stop the car but he does not know about how on pressing accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of accelerator, brakes etc in the car. This is what abstraction is.

Abstraction using Classes: We can implement Abstraction in C++ using classes. Class helps us to group data members and member functions using available access specifiers. A Class can decide which data member will be visible to outside world and which is not.

Abstraction in Header files: One more type of abstraction in C++ can be header files. For example, consider the pow() method present in math.h header file. Whenever we need to calculate power of a number, we simply call the function pow() present in the math.h header file and pass the numbers as arguments without knowing the underlying algorithm according to which the function is actually calculating power of numbers.

Abstraction using access specifiers

Access specifiers are the main pillar of implementing abstraction in C++. We can use access specifiers to enforce restrictions on class members. For example:

- Members declared as **public** in a class, can be accessed from anywhere in the program.
- Members declared as **private** in a class, can be accessed only from within the class. They are not allowed to be accessed from any part of code outside the class.

We can easily implement abstraction using the above two features provided by access specifiers. Say, the members that defines the internal implementation can be marked as private in a class. And the important information needed to be given to the outside world can be marked as public. And these public members can access the private members as they are inside the class.

Example:

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

```

class implementAbstraction
{
    private:
        int a, b;

    public:

        // method to set values of
        // private members
        void set(int x, int y)
        {
            a = x;
            b = y;
        }

        void display()
        {
            cout<<"a = " <<a << endl;
            cout<<"b = " << b << endl;
        }
};

int main()
{
    implementAbstraction obj;
    obj.set(10, 20);
    obj.display();
    return 0;
}

```

Output:

```

a = 10
b = 20

```

You can see in the above program we are not allowed to access the variables a and b directly, however one can call the function set() to set the values in a and b and the function display() to display the values of a and b.

Advantages of Data Abstraction:

- Helps the user to avoid writing the low level code
- Avoids code duplication and increases reusability.
- Can change internal implementation of class independently without affecting the user.
- Helps to increase security of an application or program as only important details are provided to the user.

25.Function Overloading in C++

- Difficulty Level : [Easy](#)
- Last Updated : 17 Jun, 2021

Function overloading is a feature of object oriented programming where two or more functions can have the same name but different parameters.

When a function name is overloaded with different jobs it is called Function Overloading.

In Function Overloading “Function” name should be the same and the arguments should be different.

Function overloading can be considered as an example of polymorphism feature in C++.

Following is a simple C++ example to demonstrate function overloading.

- CPP

```
#include <iostream>

using namespace std;

void print(int i) {
    cout << " Here is int " << i << endl;
}

void print(double f) {
    cout << " Here is float " << f << endl;
}

void print(char const *c) {
    cout << " Here is char* " << c << endl;
}

int main() {
    print(10);
    print(10.10);
    print("ten");
    return 0;
}
```

}

Output:

Here is int 10

Here is float 10.1

Here is char* ten

How Function Overloading works?

- *Exact match:-* (Function name and Parameter)
- *If a not exact match is found:-*
 - >Char, Unsigned char, and short are promoted to an int.
 - >Float is promoted to double
- *If no match found:*
 - >C++ tries to find a match through the standard conversion.
- *ELSE ERROR*

26. Operator Overloading in C++

In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big Integer, etc.

A simple and complete example

- CPP

```
#include<iostream>

using namespace std;

class Complex {
private:
    int real, imag;
public:
    Complex(int r = 0, int i = 0) {real = r;    imag = i;}

    // This is automatically called when '+' is used with
    // between two Complex objects

    Complex operator + (Complex const &obj) {
        Complex res;
        res.real = real + obj.real;
        res.imag = imag + obj.imag;
        return res;
    }

    void print() { cout << real << " + i" << imag << '\n'; }
};
```

```
int main()
{
    Complex c1(10, 5), c2(2, 4);
    Complex c3 = c1 + c2;
    c3.print();
}
```

Output

12 + i9

Output:

12 + i9

What is the difference between operator functions and normal functions?

Operator functions are the same as normal functions. The only differences are, the name of an operator function is always the operator keyword followed by the symbol of the operator and operator functions are called when the corresponding operator is used.

Following is an example of global operator function.

Output

12 + i9

Can we overload all operators?

Almost all operators can be overloaded except a few. Following is the list of operators that cannot be overloaded.

sizeof

typeid

Scope resolution (::)

Class member access operators (.(dot), .* (pointer to member operator))

Ternary or conditional (?:)

Why can't the above-stated operators be overloaded?

1. sizeof – This returns the size of the object or datatype entered as the operand. This is evaluated by the compiler and cannot be evaluated during runtime. The proper incrementing of a pointer in an array of objects relies on the sizeof operator implicitly. Altering its meaning using overloading would cause a fundamental part of the language to collapse.

2. typeid: This provides a CPP program with the ability to recover the actual derived type of the object referred to by a pointer or reference. For this operator, the whole point is to uniquely identify a type. If we want to make a user-defined type to 'look' like another type, polymorphism can be used but the meaning of the typeid operator must remain unaltered, or else serious issues could arise.

3. Scope resolution (::): This helps identify and specify the context to which an identifier refers by specifying a namespace. It is completely evaluated at runtime and works on names rather than values. The operands of scope resolution are note expressions with data types and CPP has no syntax for capturing them if it were overloaded. So it is syntactically impossible to overload this operator.

4. Class member access operators (.(dot), .* (pointer to member operator)): The importance and implicit use of class member access operators can be understood through the following example:

- C++

```
#include <iostream>

using namespace std;

class ComplexNumber{

private:
    int real;
    int imaginary;

public:
    ComplexNumber(int real, int imaginary){
        this->real = real;
        this->imaginary = imaginary;
    }

    void print(){
        cout<<real<< " + i" <<imaginary;
    }

    ComplexNumber operator+ (ComplexNumber c2){
        ComplexNumber c3(0,0);
        c3.real = this->real+c2.real;
```

```

c3.imaginary = this->imaginary + c2.imaginary;
return c3;
}
};

int main() {
    ComplexNumber c1(3,5);
    ComplexNumber c2(2,4);
    ComplexNumber c3 = c1 + c2;
    c3.print();
    return 0;
}

```

Output

5 + i9

The statement `ComplexNumber c3 = c1 + c2;` is internally translated as `ComplexNumber c3 = c1.operator+ (c2);` in order to invoke the operator function. The argument `c1` is implicitly passed using the ‘.’ operator. The next statement also makes use of the dot operator to access the member function `print` and pass `c3` as an argument. Thus, in order to ensure a reliable and non-ambiguous system of accessing class members, the predefined mechanism using class member access operators is absolutely essential. Besides, these operators also work on names and not values and there is no provision (syntactically) to overload them.

5. Ternary or conditional (?:): The ternary or conditional operator is a shorthand representation of an if-else statement. In the operator, the true/false expressions are only evaluated on the basis of the truth value of the conditional expression.

`conditional statement ? expression1 (if statement is TRUE) : expression2 (else)`

A function overloading the ternary operator for a class say ABC using the definition

`ABC operator ?: (bool condition, ABC trueExpr, ABC falseExpr);`

would not be able to guarantee that only one of the expressions was evaluated. Thus, ternary operator cannot be overloaded.

Important points about operator overloading

- 1) For operator overloading to work, at least one of the operands must be a

user-defined class object.

2) Assignment Operator: Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of the right side to the left side and works fine in most cases (this behaviour is the same as the copy constructor). See [this](#) for more details.

3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type.

- CPP

```
#include <iostream>

using namespace std;

class Fraction

{

private:

    int num, den;

public:

    Fraction(int n, int d) { num = n; den = d; }

    // Conversion operator: return float value of fraction
    operator float() const {

        return float(num) / float(den);

    }

};

int main() {

    Fraction f(2, 5);

    float val = f;

    cout << val << '\n';

    return 0;

}
```

Output

0.4

Overloaded conversion operators must be a member method. Other operators can either be the member method or the global method.

4) Any constructor that can be called with a single argument works as a conversion constructor, which means it can also be used for implicit conversion to the class being constructed.

- CPP

```
#include <iostream>

using namespace std;

class Point
{
private:
    int x, y;
public:
    Point(int i = 0, int j = 0) {
        x = i; y = j;
    }
    void print() {
        cout << "x = " << x << ", y = " << y << '\n';
    }
};

int main() {
    Point t(20, 20);
    t.print();
    t = 30; // Member x of t becomes 30
    t.print();
    return 0;
}
```

Output

x = 20, y = 20

x = 30, y = 0

27.Exception Handling in C++

One of the advantages of C++ over C is Exception Handling. Exceptions are run-time anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions: a)Synchronous, b)Asynchronous(Ex:which are beyond the program's control, Disc failure etc). C++ provides following specialized keywords for this purpose.

try: represents a block of code that can throw an exception.

catch: represents a block of code that is executed when a particular exception is thrown.

throw: Used to throw an exception. Also used to list the exceptions that a function throws, but doesn't handle itself.

Why Exception Handling?

Following are main advantages of exception handling over traditional error handling.

1) Separation of Error Handling code from Normal Code: In traditional error handling codes, there are always if else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try catch blocks, the code for error handling becomes separate from the normal flow.

2) Functions/Methods can handle any exceptions they choose: A function can throw many exceptions, but may choose to handle some of them. The other exceptions which are thrown, but not caught can be handled by caller. If the caller chooses not to catch them, then the exceptions are handled by caller of the caller.

In C++, a function can specify the exceptions that it throws using the *throw* keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it)

3) Grouping of Error Types: In C++, both basic types and objects can be thrown as exception. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, categorize them according to types.

C++ Exceptions:

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (throw an error).

C++ try and catch:

Exception handling in C++ consists of three keywords: *try*, *throw* and *catch*:

The *try* statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed, if an error occurs in the try block.

The try and catch keywords come in pairs:

We use the try block to test some code: If the age variable is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error and do something about it. The catch statement takes a parameter: in our example we use an int variable (myNum) (because we are throwing an exception of int type in the try block (age)), to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be greater than 18), the catch block is skipped:

Exception Handling in C++

1) Following is a simple example to show exception handling in C++. The output of program explains flow of execution of try/catch blocks.

- CPP

```
#include <iostream>

using namespace std;

int main()
{
    int x = -1;

    // Some code
    cout << "Before try \n";
    try {
        cout << "Inside try \n";
        if (x < 0)
        {
            throw x;
            cout << "After throw (Never executed) \n";
        }
    }
```

```

}

catch (int x) {

    cout << "Exception Caught \n";

}

cout << "After catch (Will be executed) \n";
return 0;

}

```

Output:

Before try

Inside try

Exception Caught

After catch (Will be executed)

2) There is a special catch block called ‘catch all’ `catch(...)` that can be used to catch all types of exceptions. For example, in the following program, an int is thrown as an exception, but there is no catch block for int, so `catch(...)` block will be executed.

- CPP

```

#include <iostream>

using namespace std;

int main()
{
    try  {

        throw 10;

    }

    catch (char *excp)  {

        cout << "Caught " << excp;

    }

    catch (...)  {

```

```

        cout << "Default Exception\n";
    }
    return 0;
}

```

Output:

Default Exception

- 3) Implicit type conversion doesn't happen for primitive types. For example, in the following program 'a' is not implicitly converted to int

- CPP

```

#include <iostream>

using namespace std;

int main()
{
    try {
        throw 'a';
    }
    catch (int x) {
        cout << "Caught " << x;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
    return 0;
}

```

Output:

Default Exception

- 4) If an exception is thrown and not caught anywhere, the program terminates abnormally. For example, in the following program, a char is thrown, but there is no catch block to catch a char.

- CPP

```

#include <iostream>

using namespace std;

int main()
{
    try {
        throw 'a';
    }

    catch (int x) {
        cout << "Caught ";
    }

    return 0;
}

```

Output:

terminate called after throwing an instance of 'char'

This application has requested the Runtime to terminate it in an unusual way. Please contact the application's support team for more information.

We can change this abnormal termination behavior by [writing our own unexpected function](#).

5) A derived class exception should be caught before a base class exception. See [this](#) for more details.

6) Like Java, C++ library has a [standard exception class](#) which is base class for all standard exceptions. All objects thrown by components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type

7) Unlike Java, in C++, all exceptions are unchecked. Compiler doesn't check whether an exception is caught or not (See [this](#) for details). For example, in C++, it is not necessary to specify all uncaught exceptions in a function declaration. Although it's a recommended practice to do so. For example, the following program compiles fine, but ideally signature of fun() should list unchecked exceptions.

- CPP

```

#include <iostream>

using namespace std;

// This function signature is fine by the compiler, but not recommended.
// Ideally, the function should specify all uncaught exceptions and function
// signature should be "void fun(int *ptr, int x) throw (int *, int)"
void fun(int *ptr, int x)

{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}

int main()
{
    try {
        fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}

```

Output:

Caught exception from fun()

A better way to write above code

- CPP

```

#include <iostream>

using namespace std;

// Here we specify the exceptions that this function
// throws.

void fun(int *ptr, int x) throw (int *, int) // Dynamic Exception specification
{
    if (ptr == NULL)
        throw ptr;
    if (x == 0)
        throw x;
    /* Some functionality */
}

int main()
{
    try {
        fun(NULL, 0);
    }
    catch(...) {
        cout << "Caught exception from fun()";
    }
    return 0;
}

```

(Note : The use of Dynamic Exception Specification has been deprecated after C++11, one of the reason maybe because it can randomly abort your program. This can happen when you throw an exception of an another type which is not mentioned in the dynamic exception specification, your program will abort itself, because in that scenario program calls(indirectly) terminate(), and which is by default call abort()).

Output:

Caught exception from fun()

8) In C++, try-catch blocks can be nested. Also, an exception can be re-thrown using “throw; ”

- CPP

```
#include <iostream>

using namespace std;

int main()
{
    try {
        try {
            throw 20;
        }

        catch (int n) {
            cout << "Handle Partially ";
            throw; // Re-throwing an exception
        }
    }

    catch (int n) {
        cout << "Handle remaining ";
    }

    return 0;
}
```

Output:

Handle Partially Handle remaining

A function can also re-throw a function using same “throw; ”. A function can handle a part and can ask the caller to handle remaining.

9) When an exception is thrown, all objects created inside the enclosing try block are destructed before the control is transferred to catch block.

- CPP

```
#include <iostream>

using namespace std;
```

```

class Test {
    public:
        Test() { cout << "Constructor of Test " << endl; }
        ~Test() { cout << "Destructor of Test " << endl; }
};

int main()
{
    try {
        Test t1;
        throw 10;
    }
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}

```

Output:

Constructor of Test

Destructor of Test

Caught 10

10) You may like to try [Quiz on Exception Handling in C++](#).

Please write comments if you find anything incorrect, or you want to share more information about the topic discussed above.

27.1.Stack Unwinding in C++

Stack Unwinding is the process of removing function entries from function call stack at run time. The local objects are destroyed in reverse order in which they were constructed.

Stack Unwinding is generally related to [Exception Handling](#). In C++, when an exception occurs, the function call stack is linearly searched for the exception handler, and all the entries before the function with exception handler are removed from the function call stack. So, exception handling involves Stack Unwinding if an exception is not handled in the same function (where it is thrown). Basically, Stack unwinding is a process of calling the destructors (whenever an exception is thrown) for all the automatic objects constructed at run time.

For example, the output of the following program is:

- CPP

```
// CPP Program to demonstrate Stack Unwinding

#include <iostream>

using namespace std;

// A sample function f1() that throws an int exception
void f1() throw(int)
{
    cout << "\n f1() Start ";
    throw 100;
    cout << "\n f1() End ";
}

// Another sample function f2() that calls f1()
void f2() throw(int)
{
    cout << "\n f2() Start ";
    f1();
    cout << "\n f2() End ";
}
```

```

// Another sample function f3() that calls f2() and handles
// exception thrown by f1()

void f3()
{
    cout << "\n f3() Start ";

    try {
        f2();
    }

    catch (int i) {
        cout << "\n Caught Exception: " << i;
    }

    cout << "\n f3() End";
}

// Driver Code

int main()
{
    f3();

    getchar();
    return 0;
}

```

Output

```

f3() Start
f2() Start
f1() Start
Caught Exception: 100
f3() End

```

Explanation:

- When f1() throws exception, its entry is removed from the function call stack, because f1() doesn't contain exception handler for the thrown exception, then next entry in call stack is looked for exception handler.
- The next entry is f2(). Since f2() also doesn't have a handler, its entry is also removed from the function call stack.
- The next entry in the function call stack is f3(). Since f3() contains an exception handler, the catch block inside f3() is executed, and finally, the code after the catch block is executed.

Note that the following lines inside f1() and f2() are not executed at all.

```
cout<<"\n f1() End "; // inside f1()
```

```
cout<<"\n f2() End "; // inside f2()
```

If there were some local class objects inside f1() and f2(), destructors for those local objects would have been called in the Stack Unwinding process.

Note: *Stack Unwinding also happens in Java when exception is not handled in same function.*

27.2.Catching Base and Derived Classes as Exceptions in C++ and Java

An Exception is an unwanted error or hurdle that a program throws while compiling. There are various methods to handle an exception which is termed [exceptional handling](#).

Let's discuss what is Exception Handling and how we catch base and derived classes as an exception in C++:

- If both base and derived classes are caught as exceptions, then the catch block of the derived class must appear before the base class.
- If we put the base class first then the derived class catch block will never be reached. For example, the following **C++** code prints “**Caught Base Exception**”

- C++

```
// C++ Program to demonstrate a
// Catching Base Exception

#include <iostream>

using namespace std;

class Base {

};

class Derived : public Base {

};

int main()
{
    Derived d;
    // Some other functionalities
    try {
        // Monitored code
        throw d;
    }
}
```

```

catch (Base b) {
    cout << "Caught Base Exception";
}

catch (Derived d) {
    // This 'catch' block is NEVER executed
    cout << "Caught Derived Exception";
}

getchar();
return 0;
}

```

The output of the above C++ code:

```

prog.cpp: In function ‘int main()’:
prog.cpp:20:5: warning: exception of type ‘Derived’ will be caught
    catch (Derived d) {
        ^
prog.cpp:17:5: warning:     by earlier handler for ‘Base’
    catch (Base b) {

```

In the above C++ code, if we change the order of catch statements then both catch statements become reachable.

Following is the modified program and it prints “Caught Derived Exception”

- C++

```

// C++ Program to demonstrate a catching of
// Derived exception and printing it successfully
#include <iostream>
using namespace std;

class Base {};
class Derived : public Base {};
int main()

```

```

{
    Derived d;
    // Some other functionalities
    try {
        // Monitored code
        throw d;
    }
    catch (Derived d) {
        cout << "Caught Derived Exception";
    }
    catch (Base b) {
        cout << "Caught Base Exception";
    }
    getchar(); // To read the next character
    return 0;
}

```

Output:

Caught Derived Exception

In **Java**, catching a base class exception before derived is not allowed by the compiler itself. In **C++**, the compiler might give a warning about it but compiles the code.

For example, the following Java code fails in compilation with the error message “exception Derived has already been caught”

- Java

```

// Java Program to demonstrate
// the error filename Main.java
class Base extends Exception {
}
class Derived extends Base {
}

```

```
public class Main {  
    public static void main(String args[])  
    {  
        try {  
            throw new Derived();  
        }  
        catch (Base b) {  
        }  
        catch (Derived d) {  
        }  
    }  
}
```

Error:

```
prog.java:11: error: exception Derived has already been caught  
    catch(Derived d) {}
```

27.3.Catch block and type conversion in C++

Predict the output of following C++ program.

- C++

```
#include <iostream>

using namespace std;

int main()
{
    try
    {
        throw 'x';
    }

    catch(int x)
    {
        cout << " Caught int " << x;
    }

    catch(...)
    {
        cout << "Default catch block";
    }
}
```

Output:

Default catch block

In the above program, a character 'x' is thrown and there is a catch block to catch an int. One might think that the int catch block could be matched by considering ASCII value of 'x'. But such conversions are not performed for catch blocks. Consider the following program as another example where conversion constructor is not called for thrown object.

- C++

```

#include <iostream>

using namespace std;

class MyExcept1 {};

class MyExcept2
{
public:

    // Conversion constructor
    MyExcept2 (const MyExcept1 &e )
    {
        cout << "Conversion constructor called";
    }

};

int main()
{
    try
    {
        MyExcept1 myexp1;
        throw myexp1;
    }
    catch(MyExcept2 e2)
    {
        cout << "Caught MyExcept2 " << endl;
    }
    catch(...)
    {
        cout << " Default catch block " << endl;
    }
}

```

```
    }  
  
    return 0;  
}
```

Output:

Default catch block

As a side note, the derived type objects are converted to base type when a derived object is thrown and there is a catch block to catch base type.

27.4.Exception Handling and Object Destruction in C++

An exception is termed as an unwanted error that arises during the runtime of the program. The practice of separating the anomaly-causing program/code from the rest of the program/code is known as [Exception Handling](#).

An object is termed as an instance of the class which has the same name as that of the class. A [destructor](#) is a member function of a class that has the same name as that of the class but is preceded by a ‘~’ (tilde) sign, also it is automatically called after the scope of the code runs out. The practice of pulverizing or demolition of the existing object memory is termed *object destruction*.

In other words, the class of the program never holds any kind of memory or storage, it is the object which holds the memory or storage and to deallocate/destroy the memory of created object we use destructors.

For Example:

- CPP

```
// C++ Program to show the sequence of calling  
// Constructors and destructors  
  
#include <iostream>  
  
using namespace std;  
  
  
// Initialization of class  
  
class Test {  
  
public:  
  
    // Constructor of class  
  
    Test()  
  
    {  
  
        cout << "Constructing an object of class Test "  
            << endl;  
  
    }  
  
  
    // Destructor of class  
  
    ~Test()
```

```

    {
        cout << "Destructing the object of class Test "
        << endl;
    }
};

int main()
{
    try {
        // Calling the constructor
        Test t1;
        throw 10;

    } // Destructor is being called here
    // Before the 'catch' statement
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}

```

Output:

Constructing an object of class Test

Destructing the object of class Test

Caught 10

When an exception is thrown, destructors of the objects (whose scope ends with the try block) are automatically called before the catch block gets executed. That is why the above program prints “**Destructing an object of Test**” before “**Caught 10**”.

What Happens When an Exception is Thrown From a Constructor?

Example:

- CPP

```

// C++ Program to show what really happens
// when an exception is thrown from
// a constructor

#include <iostream>
using namespace std;

class Test1 {
public:
    // Constructor of the class
    Test1()
    {
        cout << "Constructing an Object of class Test1"
            << endl;
    }
    // Destructor of the class
    ~Test1()
    {
        cout << "Destructuring an Object the class Test1"
            << endl;
    }
};

class Test2 {
public:
    // Following constructor throws
    // an integer exception
    Test2() // Constructor of the class
    {
        cout << "Constructing an Object of class Test2"
            << endl;
    }
};

```

```

        throw 20;
    }

    // Destructor of the class
    ~Test2()
    {
        cout << "Destructing the Object of class Test2"
        << endl;
    }

};

int main()
{
    try {
        // Constructed and destructed
        Test1 t1;

        // Partially constructed
        Test2 t2;

        // t3 is not constructed as
        // this statement never gets executed
        Test1 t3; // t3 is not called as t2 is
                    // throwing/returning 'int' argument which
                    // is not accepeted
                    // is the class test1'
    }

    catch (int i) {
        cout << "Caught " << i << endl;
    }
}

```

Output:

```
Constructing an Object of class Test1
Constructing an Object of class Test2
Destructuring an Object the class Test1
Caught 20
```

Destructors are only called for the completely constructed objects. When the constructor of an object throws an exception, the destructor for that object is not called.

Predict the output of the following program:

- CPP

```
// C++ program to show how many times
// Constructors and destructors are called

#include <iostream>

using namespace std;

class Test {
    static int count; // Used static to initialise the scope
                      // Of 'count' till lifetime
    int id;

public:
    // Constructor
    Test()
    {
        count++;
        id = count;
        cout << "Constructing object number " << id << endl;
        if (id == 4)
            throw 4;
    }
    // Destructor
```

```

~Test()
{
    cout << "Destructing object number " << id << endl;
}

};

int Test::count = 0;

// Source code

int main()
{
    try {
        Test array[5];
    }
    catch (int i) {
        cout << "Caught " << i << endl;
    }
}

```

Output :

```

Constructing object number 1
Constructing object number 2
Constructing object number 3
Constructing object number 4
Destructing object number 3
Destructing object number 2
Destructing object number 1
Caught 4

```

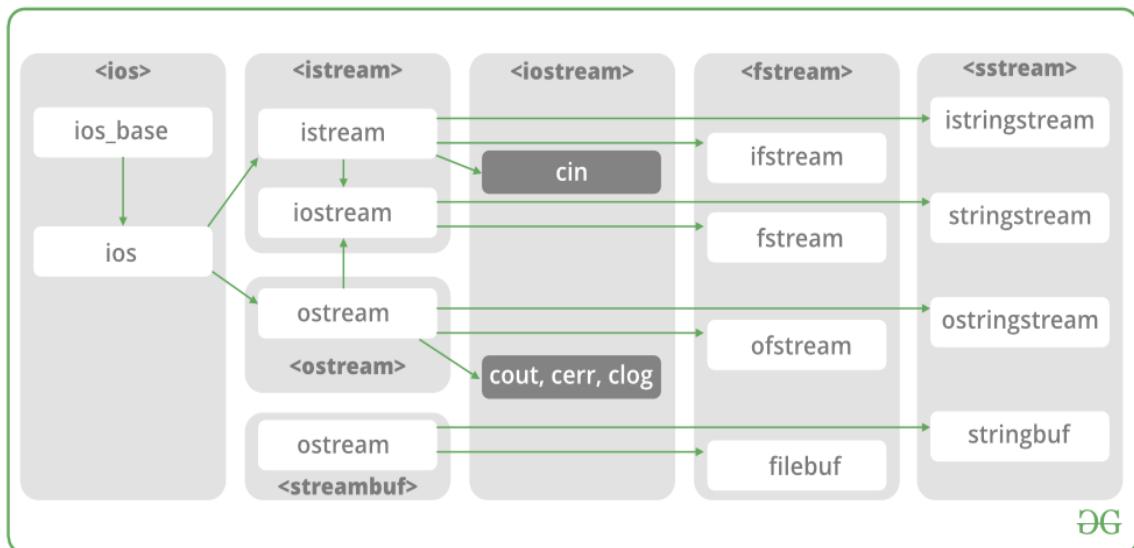
28.File Handling through C++ Classes

In C++, files are mainly dealt by using three classes `fstream`, `ifstream`, `ofstream` available in `fstream` headerfile.

ofstream: Stream class to write on files

ifstream: Stream class to read from files

fstream: Stream class to both read and write from/to files.



Now the first step to open the particular file for read or write operation. We can open file by

1. passing file name in constructor at the time of object creation
2. using the open method

For e.g.

Open File by using constructor

```
ifstream (const char* filename, ios_base::openmode mode = ios_base::in);
ifstream fin(filename, openmode) by default openmode = ios::in
ifstream fin("filename");
```

Open File by using open method

Calling of default constructor

```
ifstream fin;
fin.open(filename, openmode)
fin.open("filename");
```

Modes :

Member Constant	Stands For	Access
in *	input	File open for reading: the internal stream buffer supports input operations.
out	output	File open for writing: the internal stream buffer supports output operations.
binary	binary	Operations are performed in binary mode rather than text.
ate	at end	The output position starts at the end of the file.
app	append	All output operations happen at the end of the file, appending to its existing contents.
trunc	truncate	Any contents that existed in the file before it is open are discarded.

Default Open Modes :

ifstream ios::in

ofstream ios::out

fstream ios::in | ios::out

Problem Statement : To read and write a File in C++.

Examples:

Input :

Welcome in GeeksforGeeks. Best way to learn things.

-1

Output :

Welcome in GeeksforGeeks. Best way to learn things.

Recommended: Please try your approach on {IDE} first, before moving on to the solution.

Below is the implementation by using **ifstream & ofstream classes**.

- C++

```
/* File Handling with C++ using ifstream & ofstream class object*/  
/* To write the Content in File*/  
/* Then to read the content of file*/  
  
#include <iostream>  
  
/* fstream header file for ifstream, ofstream,  
   fstream classes */  
  
#include <fstream>  
  
using namespace std;  
  
// Driver Code  
int main()  
{  
    // Creation of ofstream class object  
    ofstream fout;  
  
    string line;  
  
    // by default ios::out mode, automatically deletes  
    // the content of file. To append the content, open in ios::app  
    // fout.open("sample.txt", ios::app)  
    fout.open("sample.txt");
```

```

// Execute a loop If file successfully opened

while (fout) {

    // Read a Line from standard input
    getline(cin, line);

    // Press -1 to exit
    if (line == "-1")
        break;

    // Write line in file
    fout << line << endl;
}

// Close the File
fout.close();

// Creation of ifstream class object to read the file
ifstream fin;

// by default open mode = ios::in mode
fin.open("sample.txt");

// Execute a loop until EOF (End of File)
while (fin) {

    // Read a Line from File
    getline(fin, line);
}

```

```

    // Print line in Console
    cout << line << endl;
}

// Close the file
fin.close();

return 0;
}

```

Below is the implementation by using **fstream class**.

- C++

```

/* File Handling with C++ using fstream class object */

/* To write the Content in File */

/* Then to read the content of file*/

#include <iostream>

/* fstream header file for ifstream, ofstream,
   fstream classes */

#include <fstream>

using namespace std;

// Driver Code

int main()
{
    // Creation of fstream class object
    fstream fio;

    string line;

```

```

// by default openmode = ios::in|ios::out mode

// Automatically overwrites the content of file, To append
// the content, open in ios::app
// fio.open("sample.txt", ios::in|ios::out|ios::app)

// ios::trunc mode delete all content before open

fio.open("sample.txt", ios::trunc | ios::out | ios::in);

// Execute a loop If file successfully Opened

while (fio) {

    // Read a Line from standard input
    getline(cin, line);

    // Press -1 to exit
    if (line == "-1")
        break;

    // Write line in file
    fio << line << endl;
}

// Execute a loop until EOF (End of File)
// point read pointer at beginning of file
fio.seekg(0, ios::beg);

while (fio) {

    // Read a Line from File
    getline(fio, line);
}

```

```
// Print line in Console  
cout << line << endl;  
}  
  
// Close the file  
fio.close();  
  
return 0;  
}
```

28.1.Read/Write Class Objects from/to File in C++

Given a file “Input.txt” in which every line has values same as instance variables of a class.

Read the values into the class’s object and do necessary operations.

Theory :

The data transfer is usually done using '>>' and '<<' operators. But if you have a class with 4 data members and want to write all 4 data members from its object directly to a file or vice-versa, we can do that using following syntax :

To write object's data members in a file :

```
// Here file_obj is an object of ofstream  
file_obj.write((char *) & class_obj, sizeof(class_obj));
```

To read file's data members into an object :

```
// Here file_obj is an object of ifstream  
file_obj.read((char *) & class_obj, sizeof(class_obj));
```

Examples:

Input :

Input.txt :

Michael 19 1806

Kemp 24 2114

Terry 21 2400

Operation : Print the name of the highest rated programmer.

Output :

Terry

- C++

```
// C++ program to demonstrate read/write of class  
// objects in C++.  
#include <iostream>  
#include <fstream>
```

```

using namespace std;

// Class to define the properties
class Contestant {
public:
    // Instance variables
    string Name;
    int Age, Ratings;

    // Function declaration of input() to input info
    int input();

    // Function declaration of output_highest_rated() to
    // extract info from file Data Base
    int output_highest_rated();
};

// Function definition of input() to input info
int Contestant::input()
{
    // Object to write in file
    ofstream file_obj;

    // Opening file in append mode
    file_obj.open("Input.txt", ios::app);

    // Object of class contestant to input data in file
    Contestant obj;

    // Feeding appropriate data in variables
    string str = "Michael";
    int age = 18, ratings = 2500;

    // Assigning data into object
    obj.Name = str;
    obj.Age = age;
    obj.Ratings = ratings;

    // Writing the object's data in file
    file_obj.write((char*)&obj, sizeof(obj));

    // Feeding appropriate data in variables
    str = "Terry";
    age = 21;
    ratings = 3200;
}

```

```

// Assigning data into object
obj.Name = str;
obj.Age = age;
obj.Ratings = ratings;

// Writing the object's data in file
file_obj.write((char*)&obj, sizeof(obj));

return 0;
}

// Function definition of output_highest_rated() to
// extract info from file Data Base
int Contestant::output_highest_rated()
{
    // Object to read from file
    ifstream file_obj;

    // Opening file in input mode
    file_obj.open("Input.txt", ios::in);

    // Object of class contestant to input data in file
    Contestant obj;

    // Reading from file into object "obj"
    file_obj.read((char*)&obj, sizeof(obj));

    // max to store maximum ratings
    int max = 0;

    // Highest_rated stores the name of highest rated contestant
    string Highest_rated;

    // Checking till we have the feed
    while (!file_obj.eof()) {
        // Assigning max ratings
        if (obj.Ratings > max) {
            max = obj.Ratings;
            Highest_rated = obj.Name;
        }
    }

    // Checking further
    file_obj.read((char*)&obj, sizeof(obj));
}

// Output is the highest rated contestant

```

```
    cout << Highest_rated;
    return 0;
}

// Driver code
int main()
{
    // Creating object of the class
    Contestant object;

    // Inputting the data
    object.input();

    // Extracting the max rated contestant
    object.output_highest_rated();

    return 0;
}
```

Output:

Terry

28.2.C++ program to create a file

Problem Statement: Write a C++ program to create a file using file handling and check whether the file is created successfully or not. If a file is created successfully then it should print “File Created Successfully” otherwise should print some error message.

Approach: Declare a stream class file and open that text file in writing mode. If the file is not present then it creates a new text file. Now check if the file does not exist or not created then return false otherwise return true.

Below is the program to create a file:

```
// C++ implementation to create a file
#include <bits/stdc++.h>
using namespace std;

// Driver code
int main()
{
    // fstream is Stream class to both
    // read and write from/to files.
    // file is object of fstream class
    fstream file;

    // opening file "Gfg.txt"
    // in out(write) mode
    // ios::out Open for output operations.
    file.open("Gfg.txt",ios::out);

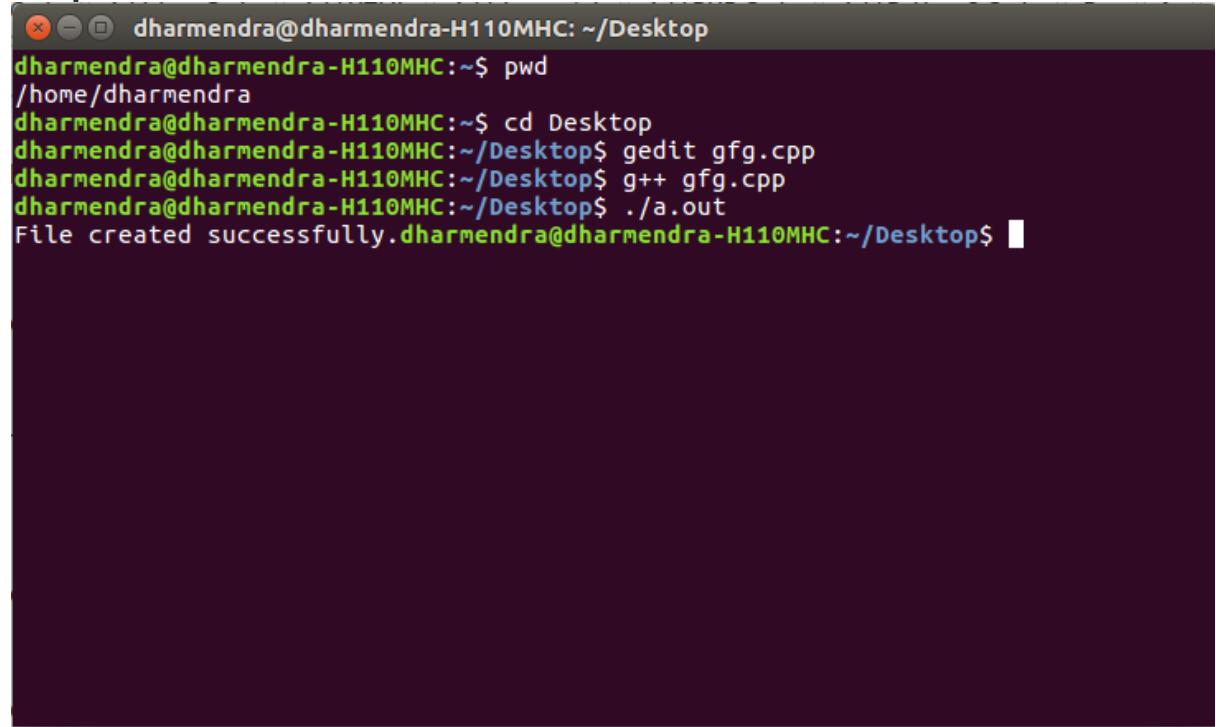
    // If no file is created, then
    // show the error message.
    if(!file)
    {
        cout<<"Error in creating file!!!";
        return 0;
    }

    cout<<"File created successfully.";

    // closing the file.
    // The reason you need to call close()
    // at the end of the loop is that trying
    // to open a new file without closing the
    // first file will fail.
```

```
    file.close();  
  
    return 0;  
}
```

Output:



```
darmendra@darmendra-H110MHC: ~/Desktop  
darmendra@darmendra-H110MHC:~/Desktop$ pwd  
/home/darmendra  
darmendra@darmendra-H110MHC:~/Desktop$ cd Desktop  
darmendra@darmendra-H110MHC:~/Desktop$ gedit gfg.cpp  
darmendra@darmendra-H110MHC:~/Desktop$ g++ gfg.cpp  
darmendra@darmendra-H110MHC:~/Desktop$ ./a.out  
File created successfully.darmendra@darmendra-H110MHC:~/Desktop$
```

28.3.CSV file management using C++

CSV is a simple file format used to store tabular data such as a spreadsheet or a database. CSV stands for **Comma Separated Values**. The data fields in a CSV file are separated/delimited by a comma (‘,’) and the individual rows are separated by a newline (‘\n’). CSV File management in C++ is similar to text-type file management, except for a few modifications.

This article discusses about how to **create, update and delete records** in a CSV file:

Note: Here, a reportcard.csv file has been created to store the student’s roll number, name and marks in math, physics, chemistry and biology.

1. Create operation:

The create operation is similar to creating a text file, i.e. input data from the user and write it to the csv file using the file pointer and appropriate delimiters(‘,’) between different columns and ‘\n’ after the end of each row.

- o CREATE

```
void create()
{
    // file pointer
    fstream fout;

    // opens an existing csv file or creates a new file.
    fout.open("reportcard.csv", ios::out | ios::app);

    cout << "Enter the details of 5 students:"
        << " roll name maths phy chem bio";
    << endl;

    int i, roll, phy, chem, math, bio;
    string name;

    // Read the input
    for (i = 0; i < 5; i++) {

        cin >> roll
            >> name
            >> math
            >> phy
            >> chem
```

```

    >> bio;

    // Insert the data to file
    fout << roll << ","
        << name << ","
        << math << ","
        << phy << ","
        << chem << ","
        << bio
        << "\n";
}
}

```

Output:

The screenshot shows a code editor interface with a dark theme. At the top is a menu bar with File, Edit, Selection, View, Go, Debug, Terminal, and Help. Below the menu is a toolbar with icons for file operations like Open, Save, Find, and Run. The main area displays two files: gfg.cpp and reportcard.csv. The reportcard.csv file is open and shows the following content:

	Roll Number	Name	Math	Physics	Chemistry
1	rahul	100	90	75	100
2	kaushik	90	80	50	90
3	nayonika	90	70	95	86
4	simran	55	85	70	70
5	sumana	65	90	95	100
6					

2. Read a particular record:

In reading a CSV file, the following approach is implemented:-

- Using getline(), file pointer and '\n' as the delimiter, read an entire row and store it in a string variable.
- Using stringstream, separate the row into words.
- Now using getline(), the stringstream pointer and ',' as the delimiter, read every word in the row, store it in a string variable and push that variable to a string vector.
- Retrieve a required column data through row[index]. Here, row[0] always stores the roll number of a student, so compare row[0] with the roll number input by the user, and if it matches, display the details of the student and break from the loop.

Note: Here, since whatever data reading from the file, is stored in string format, so always convert string to the required datatype before comparing or calculating, etc.

- READ

```
void read_record()
```

```

{

    // File pointer
    fstream fin;

    // Open an existing file
    fin.open("reportcard.csv", ios::in);

    // Get the roll number
    // of which the data is required
    int rollnum, roll2, count = 0;
    cout << "Enter the roll number "
        << "of the student to display details: ";
    cin >> rollnum;

    // Read the Data from the file
    // as String Vector
    vector<string> row;
    string line, word, temp;

    while (fin >> temp) {

        row.clear();

        // read an entire row and
        // store it in a string variable 'line'
        getline(fin, line);

        // used for breaking words
        stringstream s(line);

        // read every column data of a row and
        // store it in a string variable, 'word'
        while (getline(s, word, ',')) {

            // add all the column data
            // of a row to a vector
            row.push_back(word);
        }

        // convert string to integer for comparision
        roll2 = stoi(row[0]);

        // Compare the roll number
        if (roll2 == rollnum) {
}

```

```

        // Print the found data
        count = 1;
        cout << "Details of Roll " << row[0] << " : \n";
        cout << "Name: " << row[1] << "\n";
        cout << "Maths: " << row[2] << "\n";
        cout << "Physics: " << row[3] << "\n";
        cout << "Chemistry: " << row[4] << "\n";
        cout << "Biology: " << row[5] << "\n";
        break;
    }
}
if (count == 0)
    cout << "Record not found\n";
}

```

Output:

```

OUTPUT PROBLEMS DEBUG CONSOLE TERMINAL

nayonika@nayonika-HP-Notebook:~/Documents/cpp lab/cpp project/profinal$ ./a.out
Enter the roll number of the student to display details: 2
Details of Roll 2 :
Name: kaushik
Maths: 90
Physics: 80
Chemistry: 50
Biology: 90
nayonika@nayonika-HP-Notebook:~/Documents/cpp lab/cpp project/profinal$ ./a.out
Enter the roll number of the student to display details: 8
Record not found
nayonika@nayonika-HP-Notebook:~/Documents/cpp lab/cpp project/profinal$ █

```

3. Update a record:

The following approach is implemented while updating a record:-

- Read data from a file and compare it with the user input, as explained under read operation.
- Ask the user to enter new values for the record to be updated.
- update row[index] with the new data. Here, index refers to the required column field that is to be updated.
- Write the updated record and all other records into a new file('reportcardnew.csv').
- At the end of operation, remove the old file and rename the new file, with the old file name, i.e. remove 'reportcard.csv' and rename 'reportcardnew.csv' with 'reportcard.csv'
- UPDATE

```

void update_recode()
{
    // File pointer
    fstream fin, fout;

    // Open an existing record
    fin.open("reportcard.csv", ios::in);

    // Create a new file to store updated data
    fout.open("reportcardnew.csv", ios::out);

    int rollnum, roll1, marks, count = 0, i;
    char sub;
    int index, new_marks;
    string line, word;
    vector<string> row;

    // Get the roll number from the user
    cout << "Enter the roll number "
        << "of the record to be updated: ";
    cin >> rollnum;

    // Get the data to be updated
    cout << "Enter the subject "
        << "to be updated(M/P/C/B): ";
    cin >> sub;

    // Determine the index of the subject
    // where Maths has index 2,
    // Physics has index 3, and so on
    if (sub == 'm' || sub == 'M')
        index = 2;
    else if (sub == 'p' || sub == 'P')
        index = 3;
    else if (sub == 'c' || sub == 'C')
        index = 4;
    else if (sub == 'b' || sub == 'B')
        index = 5;
    else {
        cout << "Wrong choice.Enter again\n";
        update_record();
    }

    // Get the new marks
    cout << "Enter new marks: ";
}

```

```

cin >> new_marks;

// Traverse the file
while (!fin.eof()) {

    row.clear();

    getline(fin, line);
    stringstream s(line);

    while (getline(s, word, ',', ',')) {
        row.push_back(word);
    }

    roll1 = stoi(row[0]);
    int row_size = row.size();

    if (roll1 == rollnum) {
        count = 1;
        stringstream convert;

        // sending a number as a stream into output string
        convert << new_marks;

        // the str() converts number into string
        row[index] = convert.str();

        if (!fin.eof()) {
            for (i = 0; i < row_size - 1; i++) {

                // write the updated data
                // into a new file 'reportcardnew.csv'
                // using fout
                fout << row[i] << ", ";
            }

            fout << row[row_size - 1] << "\n";
        }
    }
    else {
        if (!fin.eof()) {
            for (i = 0; i < row_size - 1; i++) {

                // writing other existing records
                // into the new file using fout.
                fout << row[i] << ", ";
            }
        }
    }
}

```

```

    }

        // the last column data ends with a '\n'
        fout << row[row_size - 1] << "\n";
    }
}

if (fin.eof())
    break;
}

if (count == 0)
    cout << "Record not found\n";

fin.close();
fout.close();

// removing the existing file
remove("reportcard.csv");

// renaming the updated file with the existing file name
rename("reportcardnew.csv", "reportcard.csv");
}

```

Output:

```

dit Selection View Go Debug Terminal Help
gfg.cpp reportcard.csv x
1 1,rahul,100,90,75,100
2 2,kaushik,90,80,50,90
3 3,nayonika,90,78,95,86
4 4,simran,55,85,70,70
5 5,sumana,65,90,95,100
6 6,shiv,78,75,84,85
7

OUTPUT PROBLEMS DEBUG CONSOLE TERMINAL
nayonika@nayonika-HP-Notebook:~/Documents/cpp lab/cpp project/profinal$ ./a.out
Enter the roll number of the record to be updated: 3
Enter the subject to be updated(M/P/C/B): p
Enter new marks: 78
nayonika@nayonika-HP-Notebook:~/Documents/cpp lab/cpp project/profinal$ █

```

4. Delete a record:

The following approach is implemented while deleting a record

- Read data from a file and compare it with the user input, as explained under read and update operation.
- Write all the updated records, except the data to be deleted, onto a new file(reportcardnew.csv).

- Remove the old file, and rename the new file, with the old file's name.

- DELETE

```
void delete_record()
{
    // Open FILE pointers
    fstream fin, fout;

    // Open the existing file
    fin.open("reportcard.csv", ios::in);

    // Create a new file to store the non-deleted data
    fout.open("reportcardnew.csv", ios::out);

    int rollnum, roll1, marks, count = 0, i;
    char sub;
    int index, new_marks;
    string line, word;
    vector<string> row;

    // Get the roll number
    // to decide the data to be deleted
    cout << "Enter the roll number "
        << "of the record to be deleted: ";
    cin >> rollnum;

    // Check if this record exists
    // If exists, leave it and
    // add all other data to the new file
    while (!fin.eof()) {

        row.clear();
        getline(fin, line);
        stringstream s(line);

        while (getline(s, word, ',', ',')) {
            row.push_back(word);
        }

        int row_size = row.size();
        roll1 = stoi(row[0]);

        // writing all records,
        // except the record to be deleted,
    }
}
```

```

        // into the new file 'reportcardnew.csv'
        // using fout pointer
        if (roll1 != rollnum) {
            if (!fin.eof()) {
                for (i = 0; i < row_size - 1; i++) {
                    fout << row[i] << ", ";
                }
                fout << row[row_size - 1] << "\n";
            }
        }
        else {
            count = 1;
        }
        if (fin.eof())
            break;
    }
    if (count == 1)
        cout << "Record deleted\n";
    else
        cout << "Record not found\n";

    // Close the pointers
    fin.close();
    fout.close();

    // removing the existing file
    remove("reportcard.csv");

    // renaming the new file with the existing file name
    rename("reportcardnew.csv", "reportcard.csv");
}

```

Output:

```

gfg.cpp reportcard.csv *
1 1,rahul,100,90,75,100
2 2,kaushik,90,80,50,90
3 3,nayonika,90,78,95,86
4 5,sumana,65,90,95,100
5 6,shiv,78,75,84,85
6

OUTPUT PROBLEMS DEBUG CONSOLE TERMINAL
nayonika@nayonika-HP-Notebook:~/Documents/cpp_lab/cpp project/profinal$ ./a.out
Enter the roll number of the record to be deleted: 4
Record deleted
nayonika@nayonika-HP-Notebook:~/Documents/cpp_lab/cpp project/profinal$ 

```

28.4.Four File Handling Hacks which every C/C++ Programmer should know

We will discuss about four file hacks listed as below-

1. Rename – Rename a file using C/C++
2. Remove – Remove a file using C/C++
3. File Size – Get the file size using C/C++
4. Check existence – Check whether a file exists or not in C/C++

```
// A C++ Program to demonstrate the
// four file hacks every C/C++ must know

// Note that we are assuming that the files
// are present in the same file as the program
// before doing the below four hacks
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>

// A Function to get the file size
unsigned long long int fileSize(const char *filename)
{
    // Open the file
    FILE *fh = fopen(filename, "rb");
    fseek(fh, 0, SEEK_END);
    unsigned long long int size = ftell(fh);
    fclose(fh);

    return (size);
}

// A Function to check if the file exists or not
bool fileExists(const char * fname)
{
    FILE *file;
    if (file = fopen(fname, "r"))
    {
        fclose(file);
        return (true);
    }
    return (false);
}

// Driver Program to test above functions
```

```

int main()
{
    printf("%llu Bytes\n", fileSize("Passwords.txt"));
    printf("%llu Bytes\n", fileSize("Notes.docx"));

    if (fileExists("OldData.txt") == true )
        printf("The File exists\n");
    else
        printf("The File doesn't exist\n");

    rename("Videos", "English_Videos");
    rename("Songs", "English_Songs");

    remove("OldData.txt");
    remove("Notes.docx");

    if (fileExists("OldData.txt") == true )
        printf("The File exists\n");
    else
        printf("The File doesn't exist\n");

    return 0;
}

```

Output:

```

C:\Users\BELWARIAR\Desktop\Test\Four_File_Hacks.exe
14 Bytes
9924 Bytes
The File exists
The File doesn't exist

Process exited after 0.09681 seconds with return value 0
Press any key to continue . . .

```

Screenshot before executing the program :

Name	Date modified	Type	Size
PythonPrograms	19-06-2016 08:09	File folder	
Songs	19-06-2016 08:07	File folder	
Videos	19-06-2016 08:07	File folder	
Four_File_Hacks	19-06-2016 08:19	C++ Source File	2 KB
Notes	19-06-2016 08:19	Microsoft Office ...	10 KB
OldData	19-06-2016 08:08	Notepad++ Docu...	1 KB
Passwords	19-06-2016 08:10	Notepad++ Docu...	1 KB

Screenshot after executing the program :

Name	Date modified	Type	Size
English_Songs	19-06-2016 08:07	File folder	
English_Videos	19-06-2016 08:07	File folder	
PythonPrograms	19-06-2016 08:09	File folder	
Four_File_Hacks	19-06-2016 08:33	C++ Source File	2 KB
Passwords	19-06-2016 08:10	Notepad++ Docu...	1 KB

29.The C++ Standard Template Library (STL)

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized. A working knowledge of [template classes](#) is a prerequisite for working with STL.

STL has four components

- Algorithms
- Containers
- Functions
- Iterators

Algorithms

The header algorithm defines a collection of functions especially designed to be used on ranges of elements. They act on containers and provide means for various operations for the contents of the containers.

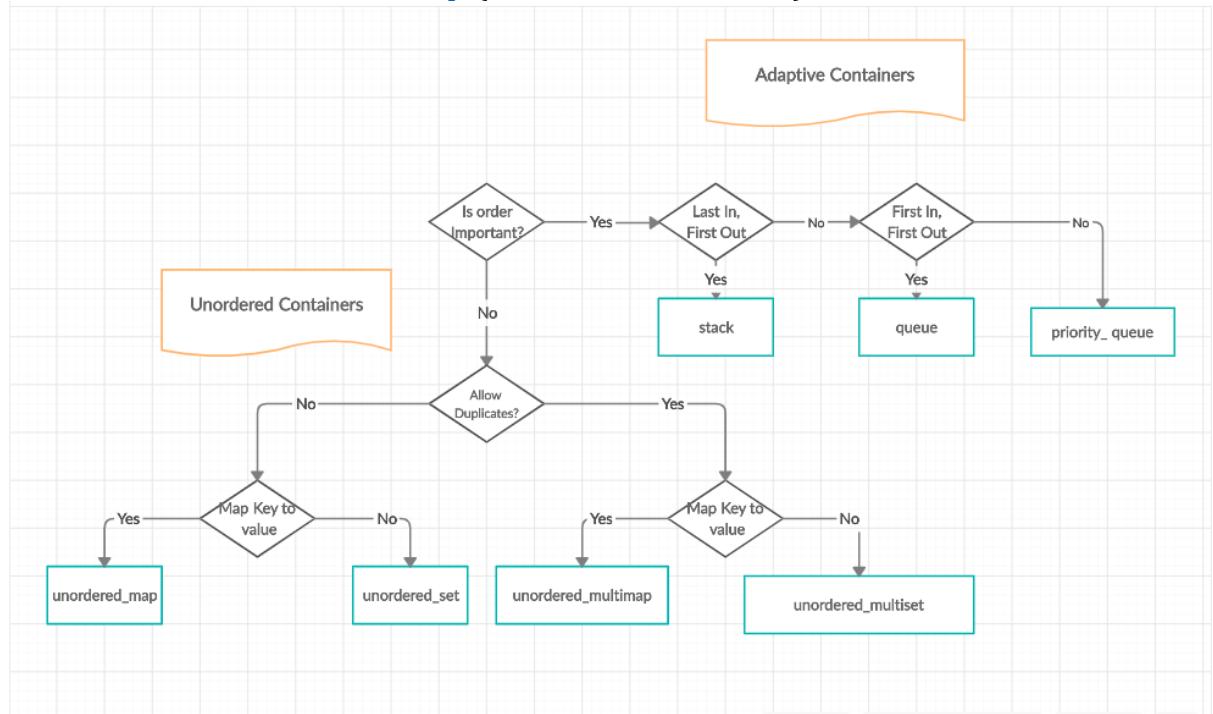
- Algorithm
 - [Sorting](#)
 - [Searching](#)
 - [Important STL Algorithms](#)
 - [Useful Array algorithms](#)
 - [Partition Operations](#)
- Numeric
 - [valarray class](#)

Containers

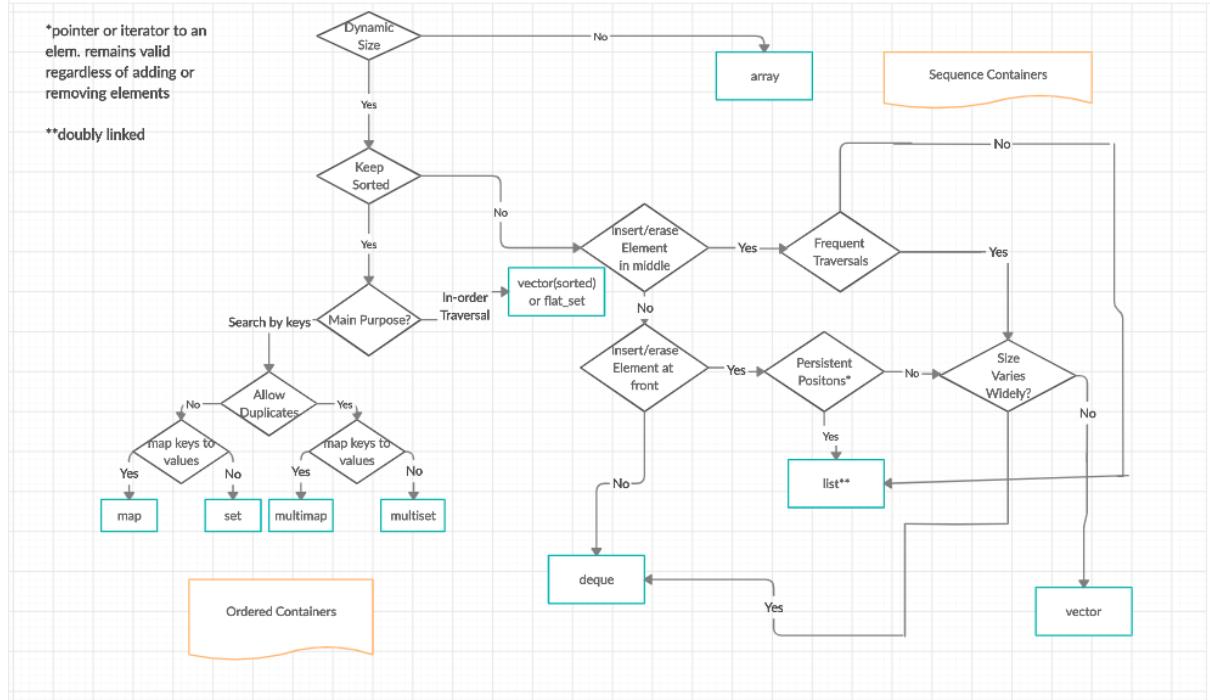
Containers or container classes store objects and data. There are in total seven standard “first-class” container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

- Sequence Containers: implement data structures which can be accessed in a sequential manner.
 - [vector](#)
 - [list](#)
 - [deque](#)
 - [arrays](#)
 - [forward list](#)(Introduced in C++11)
- Container Adaptors : provide a different interface for sequential containers.
 - [queue](#)
 - [priority queue](#)
 - [stack](#)

- **Associative Containers** : implement sorted data structures that can be quickly searched ($O(\log n)$ complexity).
 - [set](#)
 - [multiset](#)
 - [map](#)
 - [multimap](#)
- **Unordered Associative Containers** : implement unordered data structures that can be quickly searched
 - [unordered_set](#) (Introduced in C++11)
 - [unordered_multiset](#) (Introduced in C++11)
 - [unordered_map](#) (Introduced in C++11)
 - [unordered_multimap](#) (Introduced in C++11)



Flowchart of Adaptive Containers and Unordered Containers



Flowchart of Sequence containers and ordered containers

Functions

The STL includes classes that overload the function call operator. Instances of such classes are called function objects or functors. Functors allow the working of the associated function to be customized with the help of parameters to be passed.

- [Functors](#)

Iterators

As the name suggests, iterators are used for working upon a sequence of values. They are the major feature that allow generality in STL.

- [Iterators](#)

Utility Library

Defined in header <utility>.

- [pair](#)

To master **C++ Standard Template Library (STL)** in the most efficient and effective way, do check out this [C++ STL Online Course](#) by GeeksforGeeks. The course covers the basics of C++ and in-depth explanations to all C++ STL containers, iterators, etc along with video explanations of a few problems. Also, you'll learn to use STL inbuilt classes and functions in order to implement some of the complex data structures and perform operations on them conveniently.

References:

- <http://en.cppreference.com/w/cpp>
- <http://cs.stmarys.ca/~porter/csc/ref/stl/headers.html>
- <http://www.cplusplus.com/reference/stl/>

