Day 2 C++

- C++ gives flexibility to write a program with or without a class and its member functions definitions.
- A simple C++ program will include comments, namespace, headers, main(), and input/output statements

Comments

- It is useful for documentation
- C++ structure supports two comment styles
    - Single line comment

/ / An example to show single line comment
It can also be written as
/ / An example to demonstrate
/ / single line comment

- Multiline comment

/* An example to demonstrate multiline comment */
for(int i = 0; i<10; //loop runs 10 times i++) --- This is wrong... i++ also gets commented
for(int i = 0; i<10; /*loop runs 10 times */ i++)

Headers

- Usually, a program includes different programming elements such as built-in functions, classes, keywords, constants, operators and more which are already defined in the standard C++ library
- For using such pre-defined elements in a program, an appropriate header must be included in the program. Moreover, the standard headers comprise information such as prototype, definition and return type of library functions, the data type of constants and more.
- Consequently, programmers do not have to declare (or define) the predefined programming elements explicitly. Further, standard headers specify in the program through the preprocessor directive" #include.

Namespace

- Namespace allows the grouping of different entities such as classes, objects, functions and a variety of C++ tokens, etc., under a single name.
- The C++ Standards Committee has rearranged the entities of the standard library under a namespace known as std.
    - **using namespace std;**
      cout<<"Hello World";
    - **std: :cout<<"Hello World";**
- The using statement informs the compiler that you want to use the std namespace
- The purpose of namespace is to avoid name collision it is done by localising the names of identifiers
- It creates a declarative region and defines the scope
- Anything defined within a namespace is in the scope of that namespace

- Here std is the namespace in which entire standard C++ library is declared

Main Function

- The main () is a startup function that starts the execution of a c++ program. All C++ statements which are to be executed are written within main ( ). The compiler executes all the instructions written within the opening and closing curly braces' {}' that enclose the body of the main ( ).
- As soon as all the instructions in main () execute, the control passes out of main ( ), and terminates the whole program and return a value to the operating system.
- By default, main () in C++ returns an int value to the operating system. Thus, main () must end with the return 0 statement. A return value zero denotes success and a non-zero value denotes failure or error.
- Every program should have one and only one main function, otherwise the compiler will not be able to locate the begining of the program

Tokens in C++

- Token is a generic word for keywords, Data types, Variables, Constants and Identifiers

```
int main() {
        int a =2;
        double const b = 4;
        float c = 1.5;
        char d = 'A';
```

…..

Keywords, Constants, Datatype

- Keywords have fixed meaning that cannot be changed.
- Keywords cannot be used as variable names
- There are 32 keywords in C eg. Auto, break, case, char, enum, extern, etc
- There are additional 30 reserved words in C++
  new, catch, namespace, bool, class, friend etc
- Constants are fixed values
- They do not change during the execution of program
- There are two types of constants
  - Numeric constants
  - Character constants
- Constant can be defined by the use of keyword 'const'
- Another way is by using '#define' preprocessor directive

Data Type

- Data type is a finite set of values along with a set of rules
  in the above eg. int, double, float, char are data types
- a, c, d are variables

- Variable is a data name
- It may be used to store a data value
- The values can change when a program runs

- Before using a variable it must be declared
- We should try to give meaningful names to variables

Identifiers

- Identifiers are user defined names
- An identifier consists of letters, digits and underscore
- Both uppercase and lowercase letters are permitted
- First character must be an alphabet or underscore
- Identifiers are case sensitive
- Variable name, function name, structure name etc are known as identifiers

Operators

- Operator is a symbol that we use for performing mathematical or logical manipulations
    - Arithmetic operators
    - Increment and Decrement operators
    - Relational operators
    - Logical operators
    - Bitwise operators
    - Assignment operators
    - Misc. operators

Typecasting

- Typecasting is used to make one type variable to act like another type
- Typecasting is done by enclosing the data type you want within parenthesis
- (float) -- this is the cast that needs to be put in front of he variable you want to cast
- (float)varname
- This cast is valid for one single operation

Structure of object oriented program

```
#include <iostream>
using namespace std;
class square
{
   int x;   // data member
   public:
   int area(int);  //member function
};
int square :: area(int a){      //:: is a scope resolution operator. It specifies that area is not global function but it is member function of class
   x = a;
   return x*x;
}
main(){
   square sqr;     // sqr is the object of class square
   cout << "Area of square is " << sqr.area(4) << endl;     //here we are calling function area using the object sqr and dot operator
   return 0;
```

```
}
```

Class

- Class is a user-defined data type
- It is a template of an object..
    - Eg. Animal is a class and dog is the object of class animal
- Class is created using a keyword class
- It holds data and functions
- Class links the code and data
- The data and the functions are called as the members of the class

Object

- Objects are variables
- They are the copy (instances) of a class
- Each of them has property and behaviour
- Properties are defined through data elements
- Behaviour is defined through member functions called methods
- The class itself is just a template that is not allocated any memory

When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated.

- To use the data and methods defined in the class, we have to create an object of that class

Class_name object_name;

- We use (.) dot operator for accessing data and methods of an object

Eg. Obj.number()

- The public data members are also accessed in the same way, however the private data members are not allowed to be accessed directly by the object.

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 the class name and function name.

Constructors

- Constructors are special class members which are called by the compiler every time an object of that class is instantiated. Constructors have the same name as the class and may be defined inside or outside the class definition. There are 3 types of constructors:
    - Default Constructor
    - Parameterized Constructor
    - Copy Constructor – it creates a new object, which is an exact copy of the existing object. The compiler provides a default Copy Constructor to all the classes.
        - Class-name (class-name &){}

Destructors

- Destructor is another special member function that is called by the compiler when the scope of the object ends.

Syntax of a class
Class class-name
{
public/private/protected:
Data members
Member functions
};

Access specifier

- Public Specifier
  - The public specifier allows the data to be accessed outside the class
  - A public member can be used anywhere in the program
- Private Specifier
  - The members declared private cannot be accessed outside the class
  - Private members can be used only by the members of the class
- Protected Specifier
  - Protected members cannot be accessed from outside the class
  - They can be accessed by a derived class

Scope Resolution Operator

- It is used to access hidden data
- To access the variable or function with the same name we use :: operator
- Suppose the local variable and global variable has same name, the local variable gets the priority
- We can access the global variable using :: operator

Encapsulation, Data Abstraction

- So far we have seen data and function combined together in a class
- Class is a single unit in which the data and function using them are grouped, this mechanism is called as **encapsulation**
- We have also seen that the data of the class is a private member, which cannot be accessed outside the class.
- The private data is hidden and cannot be accessed outside the class, this mechanism is called **data abstraction**
- The interface is seen but the implementation is hidden

Static Members

- Static variables are initialized to zero before the first object is created
- Only one copy of the static variable exists for the whole program
- All the objects will share that variable
- It will remain in the memory till the end of the program
- A Static Function may be called by itself without depending on any object
- To access the static function we use,

Classname :: staticfunction();

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

class stat1 {
    int x;  // private member
public:
  static int sum;

  stat1(){    //Constructor
    x = sum++;
  }
  static void statdis(){
    cout << "Result is : " << sum << "\n";
  }
  void number(){
    cout << "Number is : " <<  x << "\n";
  }
};
int stat1 :: sum;  //to declare the static variable globally we use :: //now the storage is allocated to the variable sum and is initialized to zero
int main(){
    stat1 obj1, obj2, obj3;
    obj1.number();
    obj2.number();
    obj3.number();
    stat1::statdis();
    cout << " Now static var sum is" << obj1.sum << "\n";
    return 0;
}
```

Local Variables

A variable defined within a block or method or constructor is called a local variable.

- These variables are created when entered into the block 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.

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.

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.

Properties of Arrays in C++

- An Array is a collection of data of the same data type, stored at a contiguous memory location.
- Indexing of an array starts from 0. It means the first element is stored at the 0th index, the second at 1st, and so on.
- Elements of an array can be accessed using their indices.
- Once an array is declared its size remains constant throughout the program.
- An array can have multiple dimensions.
- The number of elements in an array can be determined using the sizeof operator.
- We can find the size of the type of elements stored in an array by subtracting adjacent addresses.
- There are 3 types of arrays
  - Single Dimensional Array
  - Two Dimensional Array
  - Multi Dimensional Array
- In C++, we can declare an array by simply specifying the data type first and then the name of an array with its size.
  data_type array_name[size_of_array]
  int arr[5];

Initialization of Array in C++

- In C++, we can initialize an array in many ways but we will discuss some most common ways to initialize an array. We can initialize an array at the time of declaration or after declaration.
  - int arr[5] = {1, 2, 3, 4, 5};
  - int arr[ ] = {1, 2, 3, 4, 5};
  - int arr[N];

```
for(int i =0; i<N; i++)  {
        arr[i] = value;
}
```

- Initialize array partially
    - int partialArr[5] = {1, 2};
- Initialize array to 0
    - int zeroArr[5] = {0}; //all elements will be 0, this will happen only for 0
- Acessing an element of an Array
    - Elements of an array can be accessed by specifying the name of the array, then the index of the element enclosed in the array subscript operator []. For example, arr[i].

Relation between Arrays and Pointers in C++

- In C++, arrays and pointers are closely related to each other. The array name is treated as a pointer that stored the memory address of the first element of the array.
- As we have discussed earlier, In array, elements are stored at contiguous memory locations that's why we can access all the elements of an array using the array name.
- cout << "first element: " << *arr << endl;
- cout << "Second element: " << *(arr + 1) << endl;
- cout << "Third element: " << *(arr + 2) << endl;
- cout << "fourth element: " << *(arr + 3) << endl;
- In the above code, we first declared an array "arr" with four elements. After that, we are printing the array elements.
- Array name is a pointer that stores the address of the first element of an array so, to print the first element we have dereferenced that pointer (*arr) using dereferencing operator (*) which prints the data stored at that address.
- To print the second element of an array we first add 1 to arr which is equivalent to (address of arr + size_of_one_element *1) that takes the pointer to the address just after the first one and after that, we dereference that pointer to print the second element. Similarly, we print rest of the elements of an array without using indexing.

Multidimensional Arrays in C++

- Arrays declared with more than one dimension are called multidimensional arrays.
- The most widely used multidimensional arrays are 2D arrays and 3D arrays.
- These arrays are generally represented in the form of rows and columns.
    - Data_type Array_name[Size1][Size2]….[SizeN];

Two Dimensional Array in C++

- A two-dimensional array is a grouping of elements arranged in rows and columns.
- The left index indicates the row, and right index indicates column
- Starting index of matrix or array is always 0.
- Each element is accessed using two indices: one for the row and one for the column, which makes it easy to visualize as a table or grid.

  data_type array_name[n][m];

  where n – is number of rows

  m – is number of columns

Functions in C++

- A function is a set of statements that takes input, does some specific computation, and produces output.
- The idea is to put some commonly or repeatedly done tasks together to make a function, so that instead of writing the same code again and again for different inputs, we can call this function.
- In simple terms, a function is a block of code that runs only when it is called.

data_type func_name(data_type var1, data_type var2);

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 make changes in only one place if we make changes to the functionality in future.
- 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 work.

Function Declaration

- A function declaration tells the compiler about the number of parameters, data types of parameters, and returns type of function. Writing parameter names in the function declaration is optional but it is necessary to put them in the definition.

```
// C++ Program to show 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 parameter and
// returns a 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);
```

User Defined Function

- User-defined functions are user/customer-defined blocks of code specially customized to reduce the complexity of big programs.
- They are also commonly known as "*tailor-made functions*" which are built only to satisfy the condition in which the user is facing issues meanwhile reducing the complexity of the whole program.

Library Function

- Library functions are also called "*built-in Functions*".
- These functions are part of a compiler package that is already defined and consists of a special function with special and different meanings.
- Built-in Function gives us an edge as we can directly use them without defining them whereas in the user-defined function we have to declare and define a function before using them.
  For Example: sqrt(), setw(), strcat(), etc.

Parameter Passing to Functions

- The parameters passed to the function are called *actual parameters*.

- The parameters received by the function are called *formal parameters*.

Passing 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 the function's formal parameters. The actual and formal parameters are stored in different memory locations so any changes made in the functions are not reflected in the actual parameters of the caller.

- *Pass by Reference:* Both actual and formal parameters refer to the same locations, so any changes made inside the function are reflected in the actual parameters of the caller.

Pass by value

```
// C++ Program to demonstrate function definition
#include <iostream>
using namespace std;

void fun(int x)
{
    // definition of function
    x = 30;
}

int main()
{
    int x = 20;
    fun(x);
    cout << "x = " << x;   //the value of x is not modified using the function fun().
    return 0;
}
```

Functions Using Pointers

- 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', the 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++ Program to demonstrate working of
// function using pointers
#include <iostream>
using namespace std;

void fun(int* ptr) { *ptr = 30; }

int main()
{
    int x = 20;
    fun(&x);
    cout << "x = " << x;

    return 0;
```

}

Difference between call by value and call by reference

Call by Value
A copy of the value is passed to the function
Changes made inside the function are not reflected on other functions
Actual and formal arguments will be created at different memory location

Call by Reference
An address of value is passed to the function
Changes made inside the function are reflected outside the function as well
Actual and formal arguments will be created at same memory location.

Points to Remember About Functions

- Most C++ program has a function called main() that is called by the operating system when a user runs the program.
- 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 the return statement in the body of the 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
- 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.

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 Functions

- Without parameters:
  - int main() { …. Return 0;}
- With parameters:
  - int main(int argc, char* const argv[]) { …. Return 0;}

- argc – Non negative value representing the number of arguments passed to the program from the environment in which the program is run
- argv – pointers to the first element of an array

- 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.

C++ Passing Array to Function

- In C++, to reuse the array logic, we can create a function. To pass an array to a function in C++, we need to provide only the array name.
  function_name(array_name[]); //passing array to a function

C++ Overloading (Function)

- If we create two or more members having the same name but different in number or type of parameters, it is known as C++ overloading. In C++, we can overload:
  - *methods,*
  - *constructors and*
  - *indexed properties*
- Types of overloading in C++ are:
  - *Function overloading*
  - *Operator overloading*

C++ Function Overloading

- Function Overloading is defined as the process of having two or more functions with the same name, but different parameters.
- In function overloading, the function is redefined by using either different types or number of arguments.
- It is only through these differences a compiler can differentiate between the functions.
- The advantage of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

Function Overloading and Ambiguity

- When the compiler is unable to decide which function is to be invoked among the overloaded function, this situation is known as *function overloading ambiguity.*
- When the compiler shows the ambiguity error, the compiler does not run the program

Function with Default Arguments

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int, int);
void fun(int i) { cout << "Value of i is : " << i << endl; }
void fun(int a, int b = 9)
{
   cout << "Value of a is : " << a << endl;
   cout << "Value of b is : " << b << endl;
}
int main()
{
   fun(12);

   return 0;
}
```

- shows an error "*call of overloaded 'fun(int)' is ambiguous*".
- The fun(int i) function is invoked with one argument.
- The fun(int a, int b=9) can be called in two ways:
    - first is by calling the function with one argument, i.e., fun(12) and
    - another way is calling the function with two arguments, i.e., fun(4,5).
- Therefore, the compiler could not be able to select among fun(int i) and fun(int a,int b=9).

Function with Pass By Reference

```
#include <iostream>
using namespace std;
void fun(int);
void fun(int&);
int main()
{
   int a = 10;
   fun(a); // error, which fun()?
   return 0;
}
void fun(int x) { cout << "Value of x is : " << x << endl; }
void fun(int& b)
{
   cout << "Value of b is : " << b << endl;
}
```

error "*call of overloaded 'fun(int&)' is ambiguous*"

- The first function takes one integer argument and the second function takes a reference parameter as an argument.
- In this case, the compiler does not know which function is needed by the user as there is no syntactical difference between the fun(int) and fun(int &).

Inline Functions in C++

- C++ provides inline functions to reduce the function call overhead.
- An 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 the inline function call.
- This substitution is performed by the C++ compiler at compile time.
- An inline function may increase efficiency if it is small.

```
inline return_type function_name(parameters)
        {…….}
```

- Remember, inlining is only a request to the compiler, not a command. The compiler can ignore the request for inlining.
- The compiler may not perform inlining in such circumstances as
    - If a function contains a loop. (*for, while and do-while*)
    - If a function contains static variables.
    - If a function is recursive.
    - If a function return type is other than void, and the return statement doesn't exist in a function body.
    - If a function contains a switch or goto statement.

Why Inline Functions are Used?

- 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 the execution time of a small function is less than the switching time.

Inline functions Advantages

- Function call overhead doesn't occur.
- It also saves the overhead of push/pop variables on the stack when a function is called.
- It also saves the overhead of a return call from a function.
- When you inline a function, you may enable the compiler to perform context-specific optimization on the body of the function. Such optimizations are not possible for normal function calls. Other optimizations can be obtained by considering the flows of the calling context and the called context.
- An inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function called preamble and return.

Inline function Disadvantages

1. The added variables from the inlined function consume additional registers, After the in-lining function if the variable number which is going to use the register increases then they may create overhead on register variable resource utilization. This means that when the inline function body is substituted at the point of the function call, the total number of variables used by the function also gets inserted. So the number of registers 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 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 the 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. The 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 the compiler would be required 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 the size of the binary executable file. Thrashing in memory causes the performance of the computer to degrade. The following program demonstrates the use of the inline function.

Default Arguments in C++

- A default argument is a value provided in a function declaration that is automatically assigned by the compiler if the calling function doesn't provide a value for the argument. In case any value is passed, the default value is overridden.

eg.

```
int sum(int x, int y, int z = 0, int w = 0)
```

Pointers and References in C++

- In C++ pointers and references both are mechanisms used to deal with memory, memory address, and data in a program.
- Pointers are used to store the memory address of another variable whereas references are used to create an alias for an already existing variable.

- Pointers are symbolic representation of addresses. Pointers store the address of variables or a memory location.
- They enable programs to simulate call-by-reference and create and manipulate dynamic data structures.

Datatype *var_name;

For eg. int *ptr // ptr points to an address that holds int data

- Note that the * sign can be confusing here, as it does two different things in our code:
- When used in declaration (string* ptr), it creates a pointer variable.
- When not used in declaration, it act as a dereference operator.
- A pointer however, is a variable that stores the memory address as its value. It is basically an integer, a number which stores a memory address.
- It also gives value stored at that address
- There are 3 ways to create pointer variables

string* mystring; // Preferred

string *mystring;

string * mystring;

- References and Pointers are important in C++, because they give you the ability to manipulate the data in the computer's memory - which can reduce the code and improve the performance.
- These two features are one of the things that make C++ stand out from other programming languages

How to use a pointer?

- Define a pointer variable
- Assigning the address of a variable to a pointer using the unary operator (&) which returns the address of that variable.
- Accessing the value stored in the address using unary operator (*) which returns the value of the variable located at the address specified by its operand.
- The reason we associate data type with a pointer is that it knows how many bytes the data is stored in. When we increment a pointer, we increase the pointer by the size of the data type to which it points.

Pointer Expressions and Pointer Arithmetic

- A limited set of arithmetic operations can be performed on pointers which are:
- incremented (++)
- decremented (—)
- an integer may be added to a pointer (+ or +=)
- an integer may be subtracted from a pointer (– or -=)
- difference between two pointers (p1-p2)

(Note: Pointer arithmetic is meaningless unless performed on an array.)

Advanced Pointer Notation

- 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
((nums + 1))	nums[1] [0]	25
((nums + 1)+1)	nums[1] [1]	26
((nums + 1)+2)	nums[1] [2]	27

Pointers to pointers

- In C++, we can create a pointer to a pointer that in turn may point to data or another pointer. The syntax simply requires the unary operator (*) for each level of indirection while declaring the pointer.

```
char a;
char *b;
char ** c;
a = ‘g’;
b = &a;
c = &b;
```

Here b points to a char that stores ‘g’ and c points to the pointer b.

Void Pointers

- This is a special type of pointer available in C++ which represents the absence of type.
- Void_pointers_are pointers that point to a value that has no type (and thus also an undetermined length and undetermined dereferencing properties).
- This means that void pointers have great flexibility as they can point to any data type.
- There is a payoff for this flexibility. These pointers cannot be directly dereferenced.
- They have to be first transformed into some other pointer type that points to a concrete data type before being dereferenced.
- Invalid pointers
    - A pointer should point to a valid address but not necessarily to valid elements (like for arrays). These are called invalid pointers. Uninitialized pointers are also invalid pointers.

```
int *ptr1;
int arr[10];
int *ptr = arr +20;
```

Here, ptr1 is uninitialized so it becomes an invalid pointer and ptr2 is out of bounds of arr so it also becomes an invalid pointer. (Note: invalid pointers do not necessarily raise compile errors)

- NULL Pointers
- A null pointer is a pointer that point nowhere and not just an invalid address. Following are 2 methods to assign a pointer as NULL

```
int *ptr1 =0;
Int *ptr2 = NULL;
```

Application of Pointers in C++

- To pass arguments by reference: Passing by reference serves two purposes

- For accessing array elements: The Compiler internally uses pointers to access array elements.
- To return multiple values: For example in returning square and the square root of numbers.
- Dynamic memory allocation: We can use pointers to dynamically allocate memory. The advantage of dynamically allocated memory is, that it is not deleted until we explicitly delete it.
- To implement data structures.
- To do system-level programming where memory addresses are useful.

```
void swap(int *x, int *y)
{
   int z = *x;
   *x = *y;
   *y = z;
}
Main(){swap(&a, &b);….}
```

Passing by Pointer	Passing By Reference

void swap(int *x, int *y)
{
int z = *x;
*x = *y;
*y = z;
}
Main(){swap(&a, &b); ….}
	void swap(int& x, int& y)
	{
	int z = x;
	x = y;
	y = z;
	}
	Main(){swap(a, b); ….}

We pass the address of arguments in the function call.	We pass the arguments in the function call.
The value of the arguments is accessed via the dereferencing operator *	The reference name can be used to implicitly reference a value
Passed parameters can be moved/reassigned to a different memory location.	Parameters can't be moved/reassigned to another memory address.
Pointers can contain a NULL value, so a passed argument may point to a NULL or even a garbage value.	References cannot contain a NULL value, so it is guaranteed to have some value.

Difference Between Reference Variable and Pointer Variable

- A reference is the same object, just with a different name and a reference must refer to an object. Since references can't be NULL, they are safer to use
- A pointer can be re-assigned while a reference cannot, and must be assigned at initialization only.
- The pointer can be assigned NULL directly, whereas the reference cannot.
- Pointers can iterate over an array, we can use increment/decrement operators to go to the next/previous item that a pointer is pointing to.
- A pointer is a variable that holds a memory address. A reference has the same memory address as the item it references.
- A pointer needs to be dereferenced with * to access the memory location it points to, whereas a reference can be used directly.

'this' Pointer in C++

- this is a keyword that refers to the current instance of the class.
- 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). Even if only one member of each function exists which is used by multiple objects, the compiler supplies an implicit pointer along with the names of the functions as ‘this’.
- this->x = x;
- To understand ‘this’ pointer, it is important to know how objects look at functions and data members of a class.
- Each object gets its own copy of the data member.
- 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 *’

Class pointer

- A pointer to a C++ class is done exactly the same way as a pointer to a structure and to access members of a pointer to a class you use the member access operator -> operator, just as you do with pointers to structures. Also as with all pointers, you must initialize the pointer before using it.

Structures

- The structure is a user-defined data type that is available in C++.
- Structures are used to combine different types of data types, just like an array is used to combine the same type of data types.
- A structure is declared by using the keyword “struct“. When we declare a variable of the structure we need to write the keyword “struct in C language but for C++ the keyword is not mandatory
- The are same as class only difference is that their members are by default public

```
struct
{
  // Declaration of the struct
}
```

Structure using typedef

- typedef is a keyword that is used to assign a new name to any existing data-type.
- Below is the C++ program illustrating use of struct using typedef:

```
typedef struct myStruct {
   int s1;
   char s2;
   float s3;
}str1;
int main()
{
   // Declaring a Structure
   //struct myStruct hello;
        str1 hello;
```

```
hello.s1 = 85;
hello.s2 = 'G';
```

- In the above code, the keyword “typedef” is used before struct and after the closing bracket of structure, “str1” is written.
- Now create structure variables without using the keyword “struct” and the name of the struct.
- A structure instance has been created named “hello” by just writing “str1” before it.

Enumeration

- Enums: Enums are user-defined types that consist of named integral constants.
- It helps to assign constants to a set of names to make the program easier to read, maintain and understand.
- An Enumeration is declared by using the keyword “enum“.

Dynamic Memory Allocation in C using malloc(), calloc(), free() and realloc()

- Dynamic Memory Allocation can be defined as a procedure in which the size of a data structure (like Array) is changed during the runtime.
- C provides some functions to achieve these tasks. There are 4 library functions provided by C defined under <stdlib.h> header file to facilitate dynamic memory allocation in C programming.
    - malloc()
    - calloc()
    - free()
    - realloc()
- The “malloc” or “memory allocation” method in C is used to dynamically allocate a single large block of memory with the specified size. It returns a pointer of type void which can be cast into a pointer of any form. It doesn’t Initialize memory at execution time so that it has initialized each block with the default garbage value initially.
- *ptr = (int*) malloc(100 * sizeof(int));*
  Since the size of int is 4 bytes, this statement will allocate 400 bytes of memory. And, the pointer ptr holds the address of the first byte in the allocated memory.

```
int main()
{

    // This pointer will hold the
    // base address of the block created
    int* ptr;
    int n, i;

    // Get the number of elements for the array
    printf("Enter number of elements:");
    scanf("%d",&n);
    printf("Entered number of elements: %d\n", n);

    // Dynamically allocate memory using malloc()
    ptr = (int*)malloc(n * sizeof(int));
  // Check if the memory has been successfully
    // allocated by malloc or not
    if (ptr == NULL) {
        printf("Memory not allocated.\n");
```

```
        exit(0);
    }
    else {

        // Memory has been successfully allocated
        printf("Memory successfully allocated using malloc.\n");

        // Get the elements of the array
        for (i = 0; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }
    }
   free(ptr);
    return 0;
}
```

C calloc() method

- “calloc” or “contiguous allocation” method in C is used to dynamically allocate the specified number of blocks of memory of the specified type. it is very much similar to malloc() but has two different points and these are:
- It initializes each block with a default value ‘0’.
- It has two parameters or arguments as compare to malloc().
- *ptr = (float*) calloc(25, sizeof(float));*
  This statement allocates contiguous space in memory for 25 elements each with the size of the float.

C free() method

- “free” method in C is used to dynamically de-allocate the memory. The memory allocated using functions malloc() and calloc() is not de-allocated on their own. Hence the free() method is used, whenever the dynamic memory allocation takes place. It helps to reduce wastage of memory by freeing it.
- free(ptr);

C realloc() method

- “realloc” or “re-allocation” method in C is used to dynamically change the memory allocation of a previously allocated memory. In other words, if the memory previously allocated with the help of malloc or calloc is insufficient, realloc can be used to dynamically re-allocate memory. re-allocation of memory maintains the already present value and new blocks will be initialized with the default garbage value.
- ptr = realloc(ptr, newSize);

```
// Dynamically re-allocate memory using realloc()
        ptr = (int*)realloc(ptr, n * sizeof(int));

        // Memory has been successfully allocated
        printf("Memory successfully re-allocated using realloc.\n");
```

```
        // Get the new elements of the array
        for (i = 5; i < n; ++i) {
            ptr[i] = i + 1;
        }

        // Print the elements of the array
        printf("The elements of the array are: ");
        for (i = 0; i < n; ++i) {
            printf("%d, ", ptr[i]);
        }

        free(ptr);
    }

    return 0;
```

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

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.
- 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
- pointer-variable = new data-type;
- Here, the 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.

```
// 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);
int* p = new int(25);
float* q = new float(75.25);

// Custom data type
struct cust
{
    int p;
    cust(int q) : p(q) {}
    cust() = default;
    //cust& operator=(const cust& that) = default;
};
int main()
{
    // Works fine, doesn't require constructor
    cust* var1 = new cust;

    //OR

    // Works fine, doesn't require constructor
    var1 = new cust();

    // Notice error if you comment this line
    cust* var = new cust(25);
    return 0;
}
```

- Allocate a block of memory: a 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.
- 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 top(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
- 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

- Therefore, it may be a good idea to check for the pointer variable produced by the new before using its program.
- delete operator -Since it is the programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator in C++ language.
- delete pointer-variable;
- Here, the pointer variable is the pointer that points to the data object created by *new*.

new vs malloc() and free() vs delete in C++

- We use new and delete operators in C++ to dynamically allocate memory whereas malloc() and free() functions are also used for the same purpose in C and C++.
- The functionality of the new or malloc() and delete or free() seems to be the same but they differ in various ways.
- The behavior with respect to constructors and destructors calls differ in the following ways:
  malloc() vs new():
- malloc(): It is a C library function that can also be used in C++, while the "new" operator is specific for C++ only.

- Both malloc() and new are used to allocate the memory dynamically in heap. But "new" does call the constructor of a class whereas "malloc()" does not.
- free() vs delete:
- free() is a C library function that can also be used in C++, while "delete" is a C++ keyword.
- free() frees memory but doesn't call Destructor of a class whereas "delete" frees the memory and also calls the Destructor of the class.

What is Memory Leak?

- A memory leak occurs when programmers create a memory in a heap and forget to delete it.
- The consequence of the memory leak is that it reduces the performance of the computer by reducing the amount of available memory. Eventually, in the worst case, too much of the available memory may become allocated, all or part of the system or device stops working correctly, the application fails, or the system slows down vastly.
- Memory leaks are particularly serious issues for programs like daemons and servers which by definition never terminate.

```
void f()
{
    int* ptr = (int*)malloc(sizeof(int));

    /* Do some work */

    /* Return without freeing ptr*/
    return;
}
```

How to avoid memory leaks?

- To avoid memory leaks, memory allocated on the heap should always be freed when no longer needed.

```
/* Function without memory leak */
#include <stdlib.h>
```

```
void f()
{
   int* ptr = (int*)malloc(sizeof(int));

   /* Do some work */

   /* Memory allocated by malloc is released */
   free(ptr);
   return;
}
```

Object Oriented Programming in C++

- Procedural programming is about writing procedures or functions that perform operations on the data, while object-oriented programming is about creating objects that contain both data and functions.
- Object-oriented programming has several advantages over procedural 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.
- There are some basic concepts that act as the building blocks of OOPs i.e.
    - Class
    - Objects
    - Encapsulation
    - Abstraction
    - Polymorphism
    - Inheritance
    - Dynamic Binding
    - Message Passing

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 brands but all of them will share some common properties like all of them will have 4 wheels, Speed Limit, Mileage range, etc.
- So here, the Car is the class, and wheels, speed limits, and mileage are their properties.
- A Class is a user-defined data type that has data members and member functions.
- Data members are the data variables and member functions are the functions used to manipulate these variables together these data members and member functions define 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 apply brakes, increase speed, etc.

We can say that a **Class in C++** is a blueprint representing a group of objects which shares some common properties and behaviors.

Object

- An Object is an identifiable entity with some characteristics and behavior. 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.

```
/ C++ Program to show the syntax/working of Objects as a
// part of Object Oriented PProgramming
#include <iostream>
using namespace std;

class person {
   char name[20];
   int id;

public:
   void getdetails() {}
};

int main()
{

   person p1; // p1 is a object
   return 0;
}
```

- Objects take up space in memory and have an associated address
- 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 the type of response returned by the objects.

Encapsulation

- In normal terms, Encapsulation is defined as wrapping up 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. Now,
- 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".
- **Two Important property of Encapsulation**
- **Data Protection:** Encapsulation protects the internal state of an object by keeping its data members private. Access to and modification of these data members is restricted to the class's public methods, ensuring controlled and secure data manipulation.
- **Information Hiding:** Encapsulation hides the internal implementation details of a class from external code. Only the public interface of the class is accessible, providing abstraction and simplifying the usage of the class while allowing the internal implementation to be modified without impacting external code.

Features of Encapsulation

- We can not access any function from the class directly. We need an object to access that function that is using the member variables of that class.
- The function which we are making inside the class must use only member variables, only then it is called *encapsulation*.
- If we don't make a function inside the class which is using the member variable of the class then we don't call it encapsulation.
- Encapsulation improves readability, maintainability, and security by grouping data and methods together.
- It helps to control the modification of our data members.

Encapsulation also leads to data abstraction.

Role of Access Specifiers in Encapsulation

- Access specifiers facilitate Data Hiding in C++ programs by restricting access to the class member functions and data members. There are three types of access specifiers in C++:
- **Private:** Private access specifier means that the member function or data member can only be accessed by other member functions of the same class.
- **Protected: A protected** access specifier means that the member function or data member can be accessed by other member functions of the same class or by derived classes.
- **Public:** Public access specifier means that the member function or data member can be accessed by any code.
- By **default**, all data members and member functions of a class are made **private** by the compiler.
- The process of implementing encapsulation can be sub-divided into two steps:
- Creating a class to encapsulate all the data and methods into a single unit.
- Hiding relevant data using access specifiers.

Abstraction in C++

- 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 accelerator will increase the speed of the car or applying brakes will stop the car but he does not know how on pressing the accelerator the speed is actually increasing, he does not know about the inner mechanism of the car or the implementation of the accelerator, brakes, etc in the car. This is what abstraction is.

Types of Abstraction

- **Data abstraction –** This type only shows the required information about the data and hides the unnecessary data.
- **Control Abstraction –** This type only shows the required information about the implementation and hides unnecessary information.

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.
- **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 the code outside the class.

C++ 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 real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee.
- So the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

Types of Polymorphism

- **Compile-time Polymorphism**
- **Runtime Polymorphism**

Compile-Time Polymorphism

- This type of polymorphism is achieved by function overloading or operator overloading.
- **Function Overloading**
- When there are multiple functions with the same name but different parameters, then the functions are said to be **overloaded,** hence this is known as Function Overloading.
- Functions can be overloaded by **changing the number of arguments** or/and **changing the type of arguments**.
- In simple terms, it is a feature of object-oriented programming providing many functions that have the same name but distinct parameters when numerous tasks are listed under one function name.
- There are certain Rules of Function Overloading that should be followed while overloading a function.
- **Operator Overloading**
- 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 make use of the addition operator (+) for string class to concatenate two strings.
- We know that the task of this operator is to add two operands. So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

Runtime Polymorphism

- This type of polymorphism is achieved by **Function Overriding**.
- Late binding and dynamic polymorphism are other names for runtime polymorphism.
- The function call is resolved at runtime in runtime polymorphism. In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

Virtual Function

- A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.
- **Some Key Points About Virtual Functions:**
- Virtual functions are Dynamic in nature.
- They are defined by inserting the keyword "**virtual**" inside a base class and are always declared with a base class and overridden in a child class
- A virtual function is called during Runtime

- What we do want is the selection of the function to be called at any given point in the program to be based on the kind of object for which it is called. This sort of operation is referred to as **dynamic linkage**, or **late binding**.
- Pure Virtual Functions
- It is possible that you want to include a virtual function in a base class so that it may be redefined in a derived class to suit the objects of that class, but that there is no meaningful definition you could give for the function in the base class.

```
class Shape {
  protected:
    int width, height;
  public:
    Shape(int a = 0, int b = 0) {
      width = a;
      height = b;
    }

    // pure virtual function
    virtual int area() = 0;
};
```

Constructors in C++

- **Constructor in C++** is a special method that is invoked automatically at the time of object creation.
- It is used to initialize the data members of new objects generally.
- The constructor in C++ has the same name as the class or structure.
- It constructs the values i.e. provides data for the object which is why it is known as a constructor.
- Constructor is a member function of a class, whose name is the same as the class name.
- Constructor is a special type of member function that is used to initialize the data members for an object of a class automatically when an object of the same class is created.
- Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object that is why it is known as a constructor.
- Constructors do not return value, hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.
- Constructors can be overloaded.
- A constructor cannot be declared virtual.

- **Syntax for Defining the Constructor Within the Class**

```
<class-name> (list-of-parameters)
{
   // constructor definition
}
```

- **Syntax for Defining the Constructor Outside the Class**

```
<class-name>: :<class-name>(list-of-parameters)
{
   // constructor definition
}
```

Characteristics of Constructors in C++

- The name of the constructor is the same as its class name.
- Constructors are mostly declared in the public section of the class though they can be declared in the private section of the class.
- Constructors do not return values; hence they do not have a return type.
- A constructor gets called automatically when we create the object of the class.

- Constructors can be overloaded.
- A constructor can not be declared virtual.
- A constructor cannot be inherited.
- The addresses of the Constructor cannot be referred to.
- The constructor makes implicit calls to **new** and **delete** operators during memory allocation.
- Private constructors are typically used when the class is not intended to be used as a standalone object but rather as a base class for inheritance. In this case, the derived classes can call the private constructor of the base class using the base keyword to create instances of the base class. This allows the derived classes to control how and when the base class is instantiated and can be used to implement the Singleton pattern, where only one instance of a class is allowed to be created.

Types of Constructor in C++

- Constructors can be classified based on in which situations they are being used. There are 4 types of constructors in C++:
- **Default Constructor**
- **Parameterized Constructor**
- **Copy Constructor**
- **Move Constructor**
- **Default Constructor in C++**
- A constructor without any arguments or with the default value for every argument is said to be the **Default constructor**.
- A constructor that has zero parameter list or in other sense, a constructor that accept no arguments is called a zero argument constructor or default constructor.
- If default constructor is not defined in the source code by the programmer, then the compiler defined the default constructor implicitly during compilation.
- If the default constructor is defined explicitly in the program by the programmer, then the compiler will not defined the constructor implicitly, but it calls the constructor implicitly.

Parameterized Constructor in C++

- Parameterized Constructors make it 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.
- When the parameterized constructor is defined and no default constructor is defined explicitly, the compiler will not implicitly create the default constructor and hence create a simple object as: Student s; will flash an error
- ***Important Note:*** *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.*
- **Uses of Parameterized Constructor**
- It is used to initialize the various data elements of different objects with different values when they are created.
- It is used to overload constructors.
- **Default Arguments with C++ Parameterized Constructor**
- Just like normal functions, we can also define default values for the arguments of parameterized constructors. All the rules of the default arguments will be applied to these parameters.

Copy Constructor in C++

- A copy constructor is a member function that initializes an object using another object of the same class.
- Copy constructor takes a reference to an object of the same class as an argument.

```
ClassName (ClassName &obj)
{
  // body_containing_logic
}
```

- A **copy constructor** is a member function that initializes an object using another object of the same class. In simple terms, a constructor which creates an object by initializing it with an object of the same class, which has been created previously is known as a **copy constructor**.
- Copy constructor is used to initialize the members of a newly created object by copying the members of an already existing object.
- Copy constructor takes a reference to an object of the same class as an argument.
- The process of initializing members of an object through a copy constructor is known as copy initialization.
- It is also called member-wise initialization because the copy constructor initializes one object with the existing object, both belonging to the same class on a member by member copy basis.
- The copy constructor can be defined explicitly by the programmer. If the programmer does not define the copy constructor, the compiler does it for us.
- The copy constructor is used to −
- Initialize one object from another of the same type.
- Copy an object to pass it as an argument to a function.
- Copy an object to return it from a function.

Dynamic initialization of object in C++

- Dynamic initialization of object refers to initializing the objects at a run time i.e., the initial value of an object is provided during run time.
- It can be achieved by using constructors and by passing parameters to the constructors.
- This comes in really handy when there are multiple constructors of the same class with different inputs.
- The constructor used for allocating the memory at runtime is known as the dynamic constructor.
- The memory is allocated at runtime using a new operator and similarly, memory is deallocated at runtime using the delete operator.

```
class streg {
   const char* p;

public:
   // default constructor
   streg()
   {

      // allocating memory at run time
      p = new char[6];
      p = “Hello";
   }

   void display() { cout << p << endl; }
};

int main()
{
   streg obj;
   obj.display();
```

}

Destructors in C++

- A destructor is also a special member function as a constructor.
- Destructor destroys the class objects created by the constructor.
- Destructor has the same name as their class name preceded by a tilde (~) symbol.
- It is not possible to define more than one destructor.
- The destructor is only one way to destroy the object created by the constructor. Hence destructor can-not be overloaded.
- Destructor neither requires any argument nor returns any value. It is automatically called when the object goes out of scope.
- Destructors release memory space occupied by the objects created by the constructor.
- In destructor, objects are destroyed in the reverse of object creation
- The thing is to be noted here if the object is created by using new or the constructor uses new to allocate memory that resides in the heap memory or the free store, the destructor should use delete to free the memory.

Properties of Destructor

7. The destructor function is automatically invoked when the objects are destroyed.
8. It cannot be declared static or const.
9. The destructor does not have arguments.
10. It has no return type not even void.
11. An object of a class with a Destructor cannot become a member of the union.
12. A destructor should be declared in the public section of the class.
13. The programmer cannot access the address of the destructor.

When is the destructor called?

- A destructor function is called automatically when the object goes out of scope:
- The function ends
- The program ends
- A block containing local variables ends
- A delete operator is called
- ***Note: destructor*** *can also be called explicitly for an object.*
- We can call the destructors explicitly using the following statement:

  object_name.~class_name()

How are destructors different from normal member functions?

- Destructors have the same name as the class preceded by a tilde (~)
- Destructors don’t take any argument and don’t return anything

When do we need to write a user-defined destructor?

- If we do not write our own destructor in class, the 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 the class, we should write a destructor to release memory before the class instance is destroyed.
- This must be done to avoid memory leaks.

Can a destructor be virtual?

- Yes, In fact, it is always a good idea to make destructors virtual in the base class when we have a virtual function.
- Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.

Namespace

- Namespace provide the space where we can define or declare identifier i.e. variable, method, classes.
- Using namespace, you can define the space or context in which identifiers are defined i.e. variable, method, classes. In essence, a namespace defines a scope.
- **Advantage of Namespace to avoid name collision.**
- Example, you might be writing some code that has a function called xyz() and there is another library available which is also having same function xyz(). Now the compiler has no way of knowing which version of xyz() function you are referring to within your code.
- A namespace is designed to overcome this difficulty and is used as additional information to differentiate similar functions, classes, variables etc. with the same name available in different libraries.
- The best example of namespace scope is the C++ standard library (std) where all the classes, methods and templates are declared. Hence while writing a C++ program we usually include the directive using namespace std;

Defining a Namespace:

- A namespace definition begins with the keyword namespace followed by the namespace name as follows

```
namespace  namespace_name
{
   // code declarations i.e. variable  (int a;)
   method (void add();)
   classes ( class student{};)
}
```

- It is to be noted that, there is no semicolon (;) after the closing brace.
- To call the namespace-enabled version of either function or variable, prepend the namespace name as follows:
- namespace_name: :code; // code could be variable , function or class.

The using directive

```
#include <iostream>
using namespace std;
// first name space
namespace first_space
{
  void func()
```

```
  {
    cout << "Inside first_space" << endl;
  }
}
 // second name space
namespace second_space
{
  void func()
  {
    cout << "Inside second_space" << endl;
  }
}
using namespace first_space;
int main ()
{
   // This calls function from first name space.
  func();
  return 0;
}
```

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.
- Inheritance is a feature or a process in which, new classes are created from the existing classes.
- The new class created is called “derived class” or “child class” and the existing class is known as the “base class” or “parent class”.
- The derived class now is said to be inherited from the base class.
- When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own.
- These new features in the derived class will not affect the base class.
- The derived class is the specialized class for the base class.
- **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.

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:
- 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.

- **Derived Classes:** A Derived class is defined as the class derived from the base class.

```
class  <derived_class_name> : <access-specifier> <base_class_name>
{
      //body
}
```

Where

class — keyword to create a new class

derived_class_name — name of the new class, which will inherit the base class

access-specifier — either of private, public or protected. If neither is specified, PRIVATE is taken as default

base-class-name — name of the base class

- **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.

Example:

```
1. class ABC : private XYZ            //private derivation
         {            }
2. class ABC : public XYZ            //public derivation
         {            }
3. class ABC : protected XYZ            //protected derivation
         {            }
4. class ABC: XYZ                        //private derivation by default
{          }
```

- When a base class is privately inherited by the derived class, public members of the base class becomes the private members of the derived class and therefore, the public members of the base class can only be accessed by the member functions of the derived class. They are inaccessible to the objects of the derived class.
- On the other hand, when the base class is publicly inherited by the derived class, public members of the base class also become the public members of the derived class. Therefore, the public members of the base class are accessible by the objects of the derived class as well as by the member functions of the derived class.

Modes of Inheritance

- There are 3 modes of inheritance.
- **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.
- **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.
- **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.
- // 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
};
```

Base class member access specifier	Type of Inheritence		
	Public	Protected	Private
Public	Public	Protected	Private
Protected	Protected	Protected	Private
Private	Not accessible (Hidden)	Not accessible (Hidden)	Not accessible (Hidden)

A derived class inherits all base class methods with the following exceptions −

- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Types Of Inheritance

- Single inheritance
- Multilevel inheritance
- Multiple inheritance
- Hierarchical inheritance
- Hybrid inheritance

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

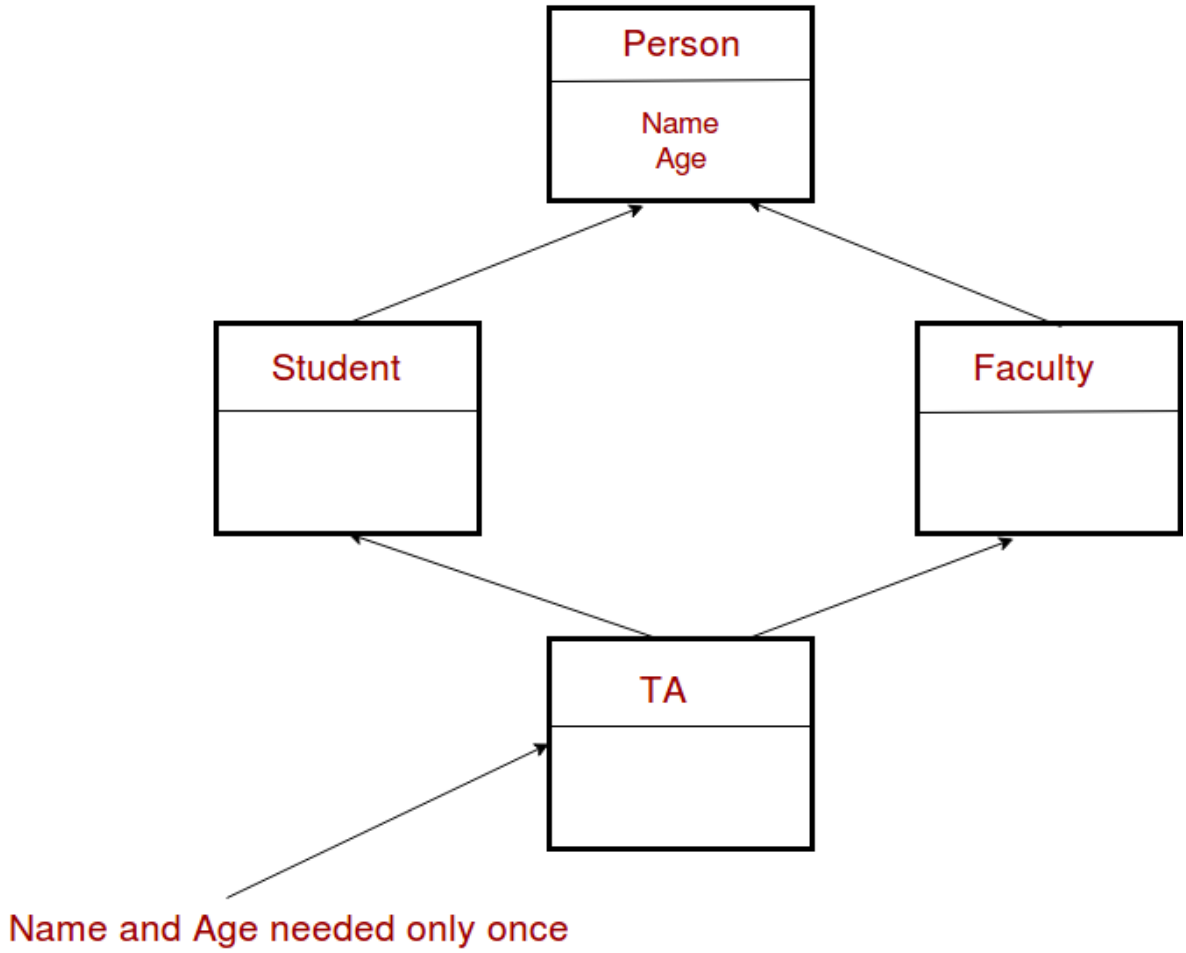
```
class subclass_name : access_mode base_class
{
```

```
  // body of subclass
};
```
OR
```
class A
{
... .. ...
};
class B: public A
{
... .. ...
};
```
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**.

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  // body of subclass
};
class B
{
... .. ...
};
class C
{
... .. ...
};
class A: public B, public C
{
... ... ...
};
```

- Multiple Inheritance is a feature of C++ where a class can inherit from more than one classes.
- The constructors of inherited classes are called in the same order in which they are inherited.
- The destructors are called in reverse order of constructors
- **The diamond problem** The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



- Constructor of 'Person' is called two times. Destructor of 'Person' will also be called two times when object 'ta1' is destructed.
- So object 'ta1' has two copies of all members of 'Person', this causes ambiguities
- *The solution to this problem is 'virtual' keyword*.
- We make the classes 'Faculty' and 'Student' as virtual base classes to avoid two copies of 'Person' in 'TA' class.
- One important thing to note in the above output is, *the default constructor of 'Person' is called*.
- When we use 'virtual' keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

Multilevel Inheritance

- In this type of inheritance, a derived class is created from another derived class.

```
class C
{
... .. ...
};
class B:public C
{
... .. ...
};
class A: public B
{
... ... ...
};
```

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.

```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

Hybrid (Virtual) Inheritance

- Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance.

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.
- **There are 2 Ways to Avoid this Ambiguity:**
    - Using the scope resolution operator we can manually specify the path from which data member a will be accessed. (obj.ClassB::a = 10;)
    - ***Note:*** Still, there are two copies of ClassA in Class-D.
    - **Avoiding ambiguity using the virtual base class:**

Virtual base class in C++

- Virtual base classes are used in virtual inheritance in a way of preventing multiple “instances” of a given class appearing in an inheritance hierarchy when using multiple inheritances.

Constructors of Derived class

- Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.
- If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke, i.e the order of invocation is that the base class’s default constructor will be invoked first and then the derived class’s default constructor will be invoked.
- **Why the base class’s constructor is called on creating an object of derived class?**
- What happens when a class is inherited from other? The data members and member functions of base class comes automatically in derived class based on the access specifier but the definition of these members exists in base class only. So when we create an object of derived class, all of the members of derived class must be initialized but the inherited members in derived class can only be initialized by the base class’s constructor as the definition of these members exists in base class

only. This is why the constructor of **base class is called first to initialize all the inherited members.**

Inheritance and Friendship in C++

- **Inheritance in C++:** This is an OOPS concept. It allows creating classes that are derived from other classes so that they automatically include some of the functionality of its base class and some functionality of its own.
- **Friendship in C++:** Usually, private and protected members of a class cannot be accessed from outside the same class in which they are declared. However, a friend class has the access to the protected and private members of the first one. Classes that are 'friends' can access not just the public members, but the private and protected members too.
- In C++, friendship is not inherited. If a base class has a friend function, then the function doesn't become a friend of the derived class(es).
- If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
- By using the keyword friend compiler knows the given function is a friend function.
- For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.
- The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

Const member functions in C++

- Constant member functions are those functions that are denied permission to change the values of the data members of their class. To make a member function constant, the keyword const is appended to the function prototype and also to the function definition header.
- Like member functions and member function arguments, the objects of a class can also be declared as const. An object declared as const cannot be modified and hence, can invoke only const member functions as these functions ensure not to modify the object. A const object can be created by prefixing the const keyword to the object declaration. Any attempt to change the data member of const objects results in a compile-time error.

C++ Operators Overloading

- Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type.
- Operator overloading is used to overload or redefines most of the operators available in C++.
- It is used to perform the operation on the user-defined data type.
- **Operator that cannot be overloaded are as follows:**
- Scope operator (::)
- Sizeof
- member selector(.)
- member pointer selector(*)
- ternary operator(?:)

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

- Where the **return type** is the type of value returned by the function.
- **class_name** is the name of the class.

- **operator op** is an operator function where op is the operator being overloaded, and the operator is the keyword.

Runtime Polymorphism

- the compiler resolves the object at run time and then it decides which function call should be associated with that object.
- It is also known as dynamic or late binding polymorphism.
- This type of polymorphism is executed through virtual functions and function overriding.
- the compiler resolves the object at run time and then it decides which function call should be associated with that object.
- It is also known as dynamic or late binding polymorphism.
- This type of polymorphism is executed through virtual functions and function overriding.

```
class Animal {
   public:
void eat(){cout<<"Eating...";}
};
class Dog: public Animal
{public:
  void eat(){ cout<<"Eating bread...";}
};
int main(void) {
Animal *a;
   Dog d;
   a = &d;
   a->eat();
}
```

Output → "Eating..."

- The output is "Eating…" which is from the base class 'Animal'
- So irrespective of what type object the base pointer is holding, the program outputs the contents of the function of the class whose base pointer is the type of. In this case, also static linking is carried out.
- In order to make the base pointer output, correct contents and proper linking, we go for dynamic binding of functions. This is achieved using Virtual functions mechanism.

Virtual Functions

- A virtual function (also known as virtual methods) is a member function that is declared within a base class and is re-defined (overridden) by a derived class.
- When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the method.
- Virtual functions ensure that the correct function is called for an object, regardless of the type of reference (or pointer) used for the function call.
- They are mainly used to achieve Runtime polymorphism.
- Functions are declared with a **virtual** keyword in a base class.
- The resolving of a function call is done at runtime.

Rules for Virtual Functions

- Virtual functions cannot be static.
- A virtual function can be a friend function of another class.
- Virtual functions should be accessed using a pointer or reference of base class type to achieve runtime polymorphism.

- The prototype of virtual functions should be the same in the base as well as the derived class.
- They are always defined in the base class and overridden in a derived class. It is not mandatory for the derived class to override (or re-define the virtual function), in that case, the base class version of the function is used.
- A class may have a virtual destructor but it cannot have a virtual constructor.
-
- Runtime polymorphism is achieved only through a pointer (or reference) of the base class type. Also, a base class pointer can point to the objects of the base class as well as to the objects of the derived class.
- Late binding (Runtime) is done in accordance with the content of the pointer (i.e. location pointed to by pointer) and Early binding (Compile-time) is done according to the type of pointer since the print() function is declared with the virtual keyword so it will be bound at runtime (output is *print derived class* as the pointer is pointing to object of derived class) and show() is non-virtual so it will be bound during compile time (output is *show base class* as the pointer is of base type).
- ***Note:*** *If we have created a virtual function in the base class and it is being overridden in the derived class then we don't need a virtual keyword in the derived class, functions are automatically considered virtual functions in the derived class.*

Working of Virtual Functions (concept of VTABLE and VPTR)

- if a class contains a virtual function then the compiler itself does two things.
- If an object of that class is created then a **virtual pointer (VPTR)** is inserted as a data member of the class to point to the VTABLE of that class. For each new object created, a new virtual pointer is inserted as a data member of that class.
- Irrespective of whether the object is created or not, the class contains as a member **a static array of function pointers called VTABLE**. Cells of this table store the address of each virtual function contained in that class.
- Explanation(runpoly4): Initially, we create a pointer of the type base class and initialize it with the address of the derived class object. When we create an object of the derived class, the compiler creates a pointer as a data member of the class containing the address of VTABLE of the derived class.
- A similar concept of Late and Early Binding is used as in the above example. For the fun_1() function call, the base class version of the function is called, fun_2() is overridden in the derived class so the derived class version is called, fun_3() is not overridden in the derived class and is a virtual function so the base class version is called, similarly fun_4() is not overridden so base class version is called.
- Note: fun_4(int) in the derived class is different from the virtual function *fun_4() in the base class as prototypes of both functions are different.*

Limitations of Virtual Functions

- **Slower:** The function call takes slightly longer due to the virtual mechanism and makes it more difficult for the compiler to optimize because it does not know exactly which function is going to be called at compile time.
- **Difficult to Debug:** In a complex system, virtual functions can make it a little more difficult to figure out where a function is being called from.

```
class Base {
public:
   // Initialization of virtual function
   virtual void fun(int x = 0)
   {
      cout << "Base::fun(), x = " << x << endl;
```

```
        }
    };
     // Initialization of Derived class
    class Derived : public Base {
    public:
        // NOTE this virtual function will take an argument
        // But haven't initialized yet
        virtual void fun(int x)
        {
            cout << "Derived::fun(), x = " << x << endl;
        }
    };
// Driver Code
int main()
{
    Derived d1; // Constructor

    // Base class pointer which will
    // Edit value in memory location of
    // Derived class constructor
    Base* bp = &d1;

    bp->fun(); // Calling a derived class member function

    return 0; // Returning 0 means the program
              // Executed successfully
}
```

- Output →Derived::fun(), x = 0
- If we take a closer look at the output, we observe that the '*fun()*' function of the derived class is called, and the default value of the base class '*fun()*' function is used.
- Default arguments do not participate in the signature(function's name, type, and order) of functions. So signatures of the '*fun()*' function in the base class and derived class are considered the same, hence the '*fun()*' function of the base class is **overridden**. Also, the default value is used at compile time. When the compiler sees an argument missing in a function call, it substitutes the default value given. Therefore, in the above program, the value of **x** is substituted at compile-time, and at run-time derived class's '*fun()*' function is called.
- **Can virtual functions be inlined?**
- Whenever a virtual function is called using a base class reference or pointer it cannot be inlined because the call is resolved at runtime, but whenever called using the object (without reference or pointer) of that class, can be inlined because the compiler knows the exact class of the object at compile time.

Virtual Destructor

- Deleting a derived class object using a pointer of base class type that has a non-virtual destructor results in undefined behavior. To correct this situation, the base class should be defined with a virtual destructor.
- Making base class destructor virtual guarantees that the object of derived class is destructed properly, i.e., both base class and derived class destructors are called.
- As a guideline, any time you have a virtual function in a class, you should immediately add a virtual destructor (even if it does nothing). This way, you ensure against any surprises later.

Pure Virtual Functions

- A **pure virtual function** (or abstract function) in C++ is a virtual function for which we can have an implementation, But we must override that function in the derived class, otherwise, the derived class will also become an abstract class.
- A pure virtual function is declared by assigning 0 in the declaration.

```
// An abstract class
class Test {
   // Data members of class
public:
   // Pure Virtual Function
   virtual void show() = 0;
    /* Other members */
};
```

- The class which has at least one pure virtual function that is called an "**abstract class**". We can never instantiate the abstract class i.e. we cannot create an object of the abstract class.
- This is because we know that an entry is made for every virtual function in the VTABLE (virtual table). But in case of a pure virtual function, this entry is without any address thus rendering it incomplete. So the compiler doesn't allow creating an object for the class with incomplete VTABLE entry.

Abstract Classes in C++

- Sometimes implementation of all functions cannot be provided in a base class because we don't know the implementation. Such a class is called an **abstract class**.
- For example, let Shape be a base class. We cannot provide the implementation of function draw() in Shape, but we know every derived class must have an implementation of draw().
- Similarly, an Animal class doesn't have the implementation of move() (assuming that all animals move), but all animals must know how to move.
- We cannot create objects of abstract classes.

```
class Test {
   // private member variable
   int x;

public:
   // pure virtual function
   virtual void show() = 0;

   // getter function to access x
   int getX() { return x; }
};
int main(void)
{
   // Error: Cannot instantiate an abstract class
   Test t;

   return 0;
}
```

- **A class is abstract if it has at least one pure virtual function.**
- **We can have pointers and references of abstract class type.**
- **If we do not override the pure virtual function in the derived class, then the derived class also becomes an abstract class.**
- **An abstract class can have constructors.**

- An abstract class can have some implementation like properties and methods along with pure virtual functions.

```
class Base {
public:
   // pure virtual function
   virtual void show() = 0;
};
class Derived : public Base {
};
int main(void)
{
   // creating an object of Derived class
   Derived d;
   return 0;
}
```

Compiler Error: cannot declare variable 'd' to be of abstract type

Interfaces

- **Interfaces** are nothing but a way to describe the behavior of a class without committing to the implementation of the class.
- In C++ programming there is no built-in concept of interfaces
- The C++ interfaces are implemented using **abstract classes** and these abstract classes should not be confused with data abstraction which is a concept of keeping implementation details separate from associated data.
- A class is made abstract by declaring at least one of its functions as **pure virtual** function. A pure virtual function is specified by placing "= 0" in its declaration
- **Importance of Interfaces**
- Any class derived from the pure abstract class (Interface) must implement all of the methods of the base class i.e. Interface.
- Interface pointers can be passed to functions and classes thereby we can call the functions of the derived class from there itself.

Type Casting

- Casting is a technique by which one data type to another data type. The operator used for this purpose is known as the **cast operator**. It is a **unary operator** which forces one data type to be converted into another data type.

```
int a, b;

   float c;
    a = 7;
   b = 2;
    // Typecasting
   c = (float)a / b;
```

- **C++ supports four types of casting:**

Static Cast
Dynamic Cast
Const Cast
Reinterpret Cast

Static Cast

- This is the simplest type of cast that can be used. It is a compile-time cast. It does things like implicit conversions between types (such as int to float, or pointer to void*), and it can also call explicit conversion functions.

- static_cast <dest_type> (source);
- The return value of static_cast will be of **dest_type.**

```
int main()
{
   float f = 3.5;

   // Implicit type case
   // float to int
   int a = f;
   cout << "The Value of a: " << a;

   // using static_cast for float to int
   int b = static_cast<int>(f);
   cout << "\nThe Value of b: " << b;
}
```

Dynamic Cast

- A cast is an operator that converts data from one type to another type.
- In C++, dynamic casting is mainly used for safe downcasting at run time.
- To work on **dynamic_cast** there must be one virtual function in the base class.
- A **dynamic_cast** works only polymorphic base class because it uses this information to decide safe downcasting.
- dynamic_cast <new_type>(Expression)
- **Downcasting:** Casting a base class pointer (or reference) to a derived class pointer (or reference) is known as downcasting. In figure 1 casting from the Base class pointer/reference to the “derived class 1” pointer/reference showing downcasting (Base ->Derived class).
- **Upcasting:** Casting a derived class pointer (or reference) to a base class pointer (or reference) is known as upcasting. In figure 1 Casting from Derived class 2 pointer/reference to the “Base class” pointer/reference showing Upcasting (Derived class 2 -> Base Class).
- As we mention above for dynamic casting there must be one Virtual function. Suppose If we do not use the virtual function, then what is the result?
- In that case, it generates an error message “Source type is not polymorphic”.

const_cast

- The **const_cast** operator is used to modify the const or volatile qualifier of a variable.
- It allows programmers to temporarily remove the constancy of an object and make modifications.
- Caution must be exercised when using const_cast, as modifying a const object can lead to undefined behavior.
- const_cast <new_type> (expression);

reinterpret_cast

- The reinterpret_cast operator is used to convert the pointer to any other type of pointer. It does not perform any check whether the pointer converted is of the same type or not.
- reinterpret_cast <new_type> (expression);

```
int number = 10;
   // Store the address of number in numberPointer
   int* numberPointer = &number;
   // Reinterpreting the pointer as a char pointer
   char* charPointer
      = reinterpret_cast<char*>(numberPointer);
   // Printing the memory addresses and values
```

```
    cout << "Integer Address: " << numberPointer << endl;
    cout << "Char Address: "
        << reinterpret_cast<void*>(charPointer) << endl;
```

- In the example, we have defined an int variable **'number'** and then store the address of 'number' in **'numberPointer'** of the int type after that we have converted the **'numberPointer'** of the int type into char pointer and then store it into **'charPointer'** variable. To verify that we have printed the address of both numberPointer and charPointer. To print the address stored in 'charPointer' **reinterpret_cast<void*>** is used to bypass the type-checking mechanism of C++ and allow the pointer to be printed as a generic memory address without any type-specific interpretation.
- ***Note:*** *const_cast and reinterpret_cast are generally not reccommended as they vulnerable to different kinds of errors.*

Exception Handling in C++

- In C++, exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution.
- The process of handling these exceptions is called exception handling.
- Using the exception handling mechanism, the control from one part of the program where the exception occurred can be transferred to another part of the code.
- So basically using exception handling in C++, we can handle the exceptions so that our program keeps running.

What is a C++ Exception?

- An exception is an unexpected problem that arises during the execution of a program our program terminates suddenly with some errors/issues. Exception occurs during the running of the program (runtime).
- **Types of C++ Exception**
- **Synchronous:** Exceptions that happen when something goes wrong because of a mistake in the input data or when the program is not equipped to handle the current type of data it's working with, such as dividing a number by zero.
- **Asynchronous**: Exceptions that are beyond the program's control, such as disc failure, keyboard interrupts, etc.
- In C++, exception is an event or object which is thrown at runtime.
- All exceptions are derived from std::exception class.
- It is a runtime error which can be handled.
- If we don't handle the exception, it prints exception message and terminates the program.
- In C++ standard exceptions are defined in <exception> class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:

C++ try and catch

- C++ provides an inbuilt feature for Exception Handling. It can be done using the following specialized keywords: try, catch, and throw with each having a different purpose.

```
try {
    // Code that might throw an exception
    throw SomeExceptionType("Error message");
 }
catch( ExceptionName e1 )  {
    // catch block catches the exception that is thrown from try block
 }
```

- The try keyword represents a block of code that may throw an exception placed inside the try block. It's followed by one or more catch blocks. If an exception occurs, try block throws that exception.
- The catch statement represents a block of code that is executed when a particular exception is thrown from the try block. The code to handle the exception is written inside the catch block.
- An exception in C++ can be thrown using the throw keyword. When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the thrown exception.
- Note: Multiple catch statements can be used to catch different type of exceptions thrown by try block.
- The try and catch keywords come in pairs: We use the try block to test some code and If the code throws an exception we will handle it in our catch block.
- **Why do we need Exception Handling in C++?**
- The following are the main advantages of exception handling over traditional error handling:
- **Separation of Error Handling Code from Normal Code***:* There are always if-else conditions to handle errors in traditional error handling codes. 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.
- **Functions/Methods can handle only the 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 the caller. If the caller chooses not to catch them, then the exceptions are handled by the 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).
- **Grouping of Error Types***:* In C++, both basic types and objects can be thrown as exceptions. We can create a hierarchy of exception objects, group exceptions in namespaces or classes, and categorize them according to their types.

Properties of Exception Handling in C++

- **Property 1 -** ***There is a special catch block called the 'catch-all' block, written as catch(…), that can be used to catch all types of exceptions.***
- **Property 2 -** ***Implicit type conversion doesn't happen for primitive types.***
- **Property 3 -** ***If an exception is thrown and not caught anywhere, the program terminates abnormally.***
- ***Note****: A derived class exception should be caught before a base class exception.*
- Like Java, the C++ library has a standard exception class which is the base class for all standard exceptions. All objects thrown by the components of the standard library are derived from this class. Therefore, all standard exceptions can be caught by catching this type.
- **Property 4 -** ***In C++, all exceptions are unchecked, i.e., the compiler doesn't check whether an exception is caught or not. So, it is not necessary to specify all uncaught exceptions in a function declaration. However,exception-handling it's a recommended practice to do so.***

Limitations of Exception Handling in C++

- Exceptions may break the structure or flow of the code as multiple invisible exit points are created in the code which makes the code hard to read and debug.
- If exception handling is not done properly can lead to resource leaks as well.
- It's hard to learn how to write Exception code that is safe.
- There is no C++ standard on how to use exception handling, hence many variations in exception-handling practices exist.

- Exception handling in C++ is used to handle unexpected happening using “try” and “catch” blocks to manage the problem efficiently. This exception handling makes our programs more reliable as errors at runtime can be handled separately and it also helps prevent the program from crashing and abrupt termination of the program when error is encountered.
- Every program takes some data as input and generates processed data as an output following the familiar input process output cycle.
- It is essential to know how to provide the input data and present the results in the desired form.
- The use of the **cin** and **cout** is already known with the operator >> and << for the input and output operations.
- C++ supports a rich set of I/O functions and operations. These functions use the advanced features of C++ such as classes, derived classes, and virtual functions.
- It also supports all of C’s set of I/O functions and therefore can be used in C++ programs, but their use should be restrained due to two reasons.
- I/O methods in C++ support the concept of OOPs.
- I/O methods in C cannot handle the user-define data type such as class and object.
- It uses the concept of stream and stream classes to implement its I/O operations with the console and disk files.

C++ Stream

- The I/O system in C++ is designed to work with a wide variety of devices including terminals, disks, and tape drives.
- Although each device is very different, the I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as the **stream.**
- It 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:**
- **iostream**: iostream stands for standard input-output stream. This header file contains definitions of objects like cin, cout, cerr, etc.
- **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.
- **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.
- **bits/stdc++:** This header file includes every standard library. In programming contests, using this file is a good idea, when you want to reduce the time wasted in doing chores; especially when your rank is time sensitive. To know more about this header file refer this article.

- In C++ after the header files, we often use ‘*using namespace std;*‘. The reason behind it is that all of the standard library definitions are inside the namespace std. As the library functions are not defined at global scope, so in order to use them we use *namespace std*. So, that we don’t need to write STD:: at every line (eg. STD::cout etc.).
- 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.
- The C++ **cout** statement is the instance of the **ostream** class.

- cin statement is the instance of the class **istream**
- 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)

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

- **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.

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

- Although each device is very different, the I/O system supplies an interface to the programmer that is independent of the actual device being accessed. This interface is known as the **stream.**
- stream refers to the stream of characters that are transferred between the program thread and i/o.
- In C++ there are number of stream classes for defining various streams related with files and for doing input-output operations.
- All these classes are defined in the file **iostream.h**. Figure given below shows the hierarchy of these classes.
- **ios class** is topmost class in the stream classes hierarchy. It is the base class for **istream, ostream,** and **streambuf** class.

- **istream** and **ostream** serves the base classes for **iostream** class. The class **istream** is used for input and **ostream** for the output.

- Class **ios** is indirectly inherited to **iostream** class using **istream** and **ostream**. To avoid the duplicity of data and member functions of **ios** class, it is declared as virtual base class when inheriting in **istream** and **ostream** as

```
class istream: virtual public ios
{
};
class ostream: virtual public ios
{
};
```

- The **_withassign classes** are provided with extra functionality for the assignment operations that’s why **_withassign** classes.

Facilities provided by these stream classes.

- **The ios class:** The ios class is responsible for providing all input and output facilities to all other stream classes.

- **The istream class:** This class is responsible for handling input stream. It provides number of function for handling chars, strings and objects such as **get, getline, read, ignore, putback** etc..

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

int main()
{
   char x;

   // used to scan a single char
   cin.get(x);

   cout << x;
}
```

- **The ostream class:** This class is responsible for handling output stream. It provides number of function for handling chars, strings and objects such as **write, put** etc..
- #include <iostream>
- using namespace std;

```
int main()
{
   char x;

   // used to scan a single char
   cin.get(x);

   // used to put a single char onto the screen.
   cout.put(x);
}
```

- **The iostream:** This class is responsible for handling both input and output stream as both **istream class** and o**stream class** is inherited into it. It provides function of both **istream class** and o**stream class** for handling chars, strings and objects such as **get, getline, read, ignore, putback, put, write** etc..

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

int main()
{
   // this function display
   // ncount character from array
   cout.write(“Hello World", 7);   // Hello W
}
```

- **istream_withassign class:** This class is variant of **istream** that allows object assignment.
- The predefined object **cin** is an object of this class and thus may be reassigned at run time to a different **istream** object.
- **ostream_withassign class:** This class is variant of **ostream** that allows object assignment.

- The predefined objects **cout, cerr, clog** are objects of this class and thus may be reassigned at run time to a different **ostream** object.
- Objects used to communicate through the standard input and output devices are declared in <iostream> file and are available in to std namespace. Objects cin, cout, cerr and clog are declared for narrow character streams; whereas objects wcin, wcout, wcerr and wclog for wide-character streams.

Unformatted input/output operations In C++

- In C++. Using objects **cin** and **cout** for the input and the output of data of various types is possible because of overloading of operator >> and << to recognize all the basic C++ types.
- The operator >> is overloaded in the **istream class** and operator << is overloaded in the ostream class.
- The general format for reading data from the keyboard:
- *cin >> var1 >> var2 >> .... >> var_n;*
- **put() and get() functions:**
- The class **istream** and **ostream** have predefined functions get() and put(), to handle single character input and output operations.
- The function **get()** can be used in two ways, such as **get(char*)** and **get(void)** to fetch characters including blank spaces, newline characters, and tab.
- The function **get(char*)** assigns the value to a variable and **get(void)** to return the value of the character.
- *char data;*
- *// get() return the character value and assign to data variable*
  data = cin.get();
- *// Display the value stored in data variable*
  cout.put(data);
- getline() and write() functions**:**
- the function getline() and write() provide a more efficient way to handle line-oriented inputs and outputs. **getline()** function reads the complete line of text that ends with the new line character. This function can be invoked using the **cin object**.
- *cin.getline(variable_to_store_line, size);*
- The reading is terminated by the **'\n' (newline) character**. The new character is read by the function, but it does not display it, instead, it is replaced with a NULL character. After reading a particular string the **cin** automatically adds the newline character at end of the string.
- The **write() function** displays the entire line in one go and its syntax is similar to the getline() function only that here **cout object** is used to invoke it.
- *cout.write(variable_to_store_line, size);*
- The key point to remember is that the **write() function** does not stop displaying the string automatically when a **NULL character** occurs. If the size is greater than the length of the line then, the **write() function** displays beyond the bound of the line.

Formatted I/O in C++

- C++ helps you to format the I/O operations like determining the number of digits to be displayed after the decimal point, specifying number base etc.
- If we want to add + sign as the prefix of out output, we can use the formatting to do so

stream.setf(ios::showpos)
If input=100, output will be +100

- If we want to add trailing zeros in out output to be shown when needed using the formatting:

stream.setf(ios::showpoint)
If input=100.0, output will be 100.000

- There are two ways to do formatting
- Using the ios class or various ios member functions.

- Using manipulators(special functions)
- **Formatting using the ios members:**
- **width():** The width method is used to set the required field width. The output will be displayed in the given width
- **precision():** The precision method is used to set the number of the decimal point to a float value
- **fill():** The fill method is used to set a character to fill in the blank space of a field
- **setf():** The setf method is used to set various flags for formatting output
- **unsetf():** The unsetf method is used To remove the flag setting

Formatting using Manipulators

- The second way you can alter the format parameters of a stream is through the use of special functions called manipulators that can be included in an I/O expression.
- To access manipulators that take parameters (such as setw( )), you must include "iomanip" header file in your program.
- The standard manipulators are shown below:
- boolalpha: The boolalpha manipulator of stream manipulators in C++ is used to turn on bool alpha flag
- dec: The dec manipulator of stream manipulators in C++ is used to turn on the dec flag
- endl: The endl manipulator of stream manipulators in C++ is used to Output a newline character.
- and: The and manipulator of stream manipulators in C++ is used to Flush the stream
- ends: The ends manipulator of stream manipulators in C++ is used to Output a null
- fixed: The fixed manipulator of stream manipulators in C++ is used to Turns on the fixed flag
- flush: The flush manipulator of stream manipulators in C++ is used to Flush a stream
- hex: The hex manipulator of stream manipulators in C++ is used to Turns on hex flag
- internal: The internal manipulator of stream manipulators in C++ is used to Turns on internal flag
- left: The left manipulator of stream manipulators in C++ is used to Turns on the left flag

File

- Files are used to store data in a storage device permanently. File handling provides a mechanism to store the output of a program in a file and to perform various operations on it. A stream is an abstraction that represents a device on which operations of input and output are performed

File Handling through C++ Classes

- File handling is used to store data permanently in a computer. Using file handling we can store our data in secondary memory (Hard disk).
  How to achieve the File Handling
  For achieving file handling we need to follow the following steps:-
  STEP 1-Naming a file
  STEP 2-Opening a file
  STEP 3-Writing data into the file
  STEP 4-Reading data from the file
  STEP 5-Closing a file.
- We give input to the executing program and the execution program gives back the output. The sequence of bytes given as input to the executing program and the sequence of bytes that comes as output from the executing program are called stream. In other words, streams are nothing but the flow of data in a sequence.
- The input and output operation between the executing program and the devices like keyboard and monitor are known as "console I/O operation". The input and output operation between the executing program and files are known as "disk I/O operation".

Classes for File stream operations

- The I/O system of C++ contains a set of classes which define the file handling methods.
- These include ifstream, ofstream and fstream classes.
- These classes are derived from fstream and from the corresponding iostream class.
- These classes, designed to manage the disk files, are declared in fstream and therefore we must include this file in any program that uses files.
- ios:- ios stands for input output stream.
- This class is the base class for other classes in this class hierarchy.
- This class contains the necessary facilities that are used by all the other derived classes for input and output operations.
- istream:- istream stands for input stream.
- This class is derived from the class 'ios'.
- This class handle input stream.
- The extraction operator(>>) is overloaded in this class to handle input streams from files to the program execution.
- This class declares input functions such as get(), getline() and read().
- ostream:- ostream stands for output stream.
- This class is derived from the class 'ios'.
- This class handle output stream.
- The insertion operator(<<) is overloaded in this class to handle output streams to files from the program execution.
- This class declares output functions such as put() and write().
- streambuf:- This class contains a pointer which points to the buffer which is used to manage the input and output streams.
- fstreambase:- This class provides operations common to the file streams. Serves as a base for fstream, ifstream and ofstream class.
- This class contains open() and close() function.
- ifstream:- This class provides input operations.
- It contains open() function with default input mode.
- Inherits the functions get(), getline(), read(), seekg() and tellg() functions from the istream.
- ofstream:- This class provides output operations.
- It contains open() function with default output mode.
- Inherits the functions put(), write(), seekp() and tellp() functions from the ostream.
- fstream:- This class provides support for simultaneous input and output operations.
- Inherits all the functions from istream and ostream classes through iostream.
- filebuf:- Its purpose is to set the file buffers to read and write.
- We can also use file buffer member function to determine the length of the file.
- 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");

Templates

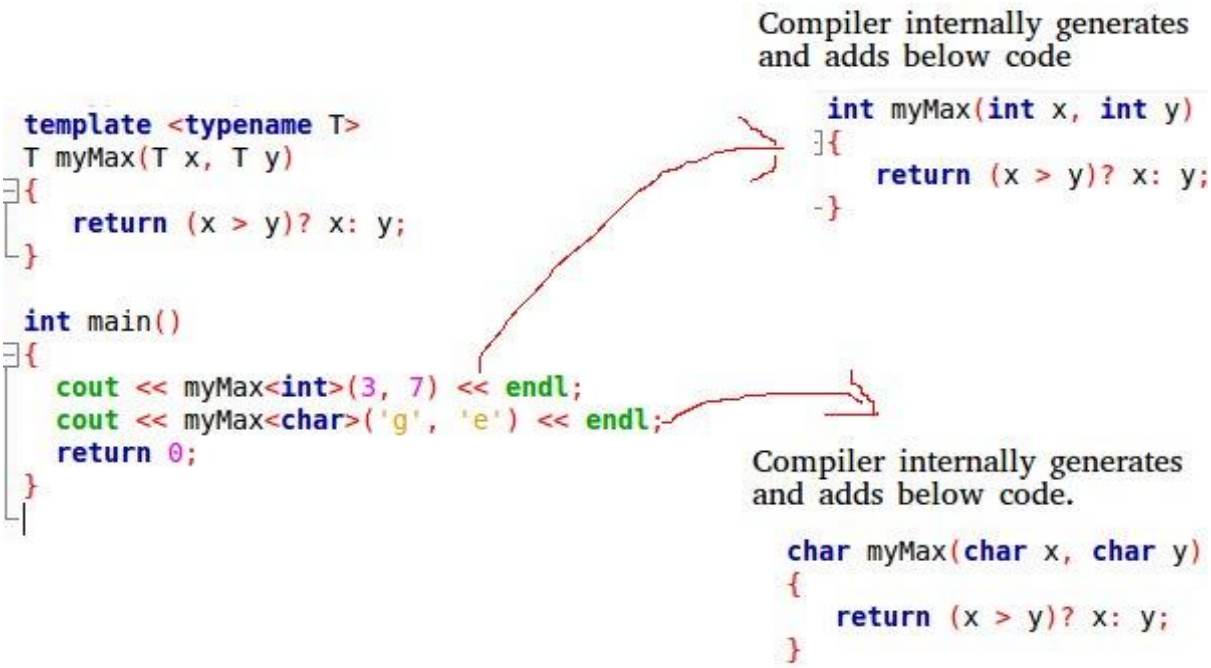
- A **template** is a simple yet very powerful tool in C++
- Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type.
- It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.
- A template is a blueprint or formula for creating a generic class or a function.
- The simple idea is to pass the data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need to sort() for different data types. Rather than writing and maintaining multiple codes, we can write one sort() and pass the datatype as a parameter.
- C++ adds two new keywords to support templates: ***'template'*** and ***'type name'***. The second keyword can always be replaced by the keyword **'class'**.

Templates can be represented in two ways:

- **Function templates**
- **Class templates**

How Do Templates Work?

- Templates are expanded at compile time.
- This is like macros.
- The difference is, that the compiler does type-checking before template expansion.
- The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of the same function/class.




```
template < class Ttype> ret_type func_name(parameter_list)
{
   // body of function.
}
```

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.
class: A class keyword is used to specify a generic type in a template declaration.

Function Templates

- We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray().
- Generics is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces.
- The method of Generic Programming is implemented to increase the efficiency of the code. Generic Programming enables the programmer to write a general algorithm which will work with all data types. It eliminates the need to create different algorithms if the data type is an integer, string or a character.
- Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
- The type of the data that the function will operate on depends on the type of the data passed as a parameter.
- For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
- A Generic function is created by using the keyword template. The template defines what function will do.

Function Templates with Multiple Parameters

- We can use more than one generic type in the template function by using the comma to separate the list.

```
template<class T1, class T2,.....>
return_type function_name (arguments of type T1, T2....)
{
   // body of function.
}
```

The typename and class are keywords used in templates in C++. There is no difference between the typename and class keywords. Both of the keywords are interchangeably used by the C++ developers as per their preference. There is no semantic difference between class and typename in a type-parameter-key

```
#include <iostream>
using namespace std;
template<class X,class Y> void fun(X a,Y b)
{
   std::cout << "Value of a is : " <<a<< std::endl;
   std::cout << "Value of b is : " <<b<< std::endl;
}
int main()
{
  fun(15,12.3);

  return 0;
}
```

Overloading a Function Template

- We can overload the generic function means that the overloaded template functions can differ in the parameter list.

E.g. fun(10);
fun(20,30.5);

Restrictions of Generic Functions

- Generic functions perform the same operation for all the versions of a function except the data type differs. Let's see a simple example of an overloaded function which cannot be replaced by the generic function as both the functions have different functionalities.

```
void fun(double a)
{
   cout<<"value of a is : "<<a<<'\n';
}
void fun(int b)
{
   if(b%2==0)
   {
      cout<<"Number is even";
   }
   else
   {
      cout<<"Number is odd";
   }
```

Class Templates

- **Class Template** can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.
- Class templates like function templates, class templates are useful when a class defines something that is independent of the data type.
- Can be useful for classes like LinkedList, BinaryTree, Stack, Queue, Array, etc.

```
template<class Ttype>
class class_name
{  ….
}
```

- **Ttype** is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.
- Now, we create an instance of a class
- class_name<type> ob;
- where **class_name**: It is the name of the class.
- **type**: It is the type of the data that the class is operating on.
- **ob**: It is the name of the object.

CLASS TEMPLATE WITH MULTIPLE PARAMETERS

- We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

```
template<class T1, class T2, ......>
class class_name
{
   // Body of the class.
}
```

Nontype Template Arguments

- The template can contain multiple arguments, and we can also use the non-type arguments In addition to the type T argument, we can also use other types of arguments such as strings, function names, constant expression and built-in types.

```
template<class T, int size>
class array
```

```
{
    T arr[size];          // automatic array initialization.
};
```

- Here the nontype template argument is size

Arguments are specified when the objects of a class are created

- Non-type parameters are mainly used for specifying max or min values or any other constant value for a particular instance of a template.
- The important thing to note about non-type parameters is, that they must be **const**.
- The compiler must know the value of non-type parameters at compile time. Because the compiler needs to create functions/classes for a specified non-type value at compile time.

default value for Template arguments

- **Can we specify a default value for template arguments?**
- Yes, like normal parameters, we can specify default arguments to templates. The following example demonstrates the same.
  template <class T, class U = char> class A
- **What is the difference between function overloading and templates?**
- Both function overloading and templates are examples of polymorphism features of OOP. Function overloading is used when multiple functions do quite similar (not identical) operations, templates are used when multiple functions do identical operations.

Template Specialization

- We write code once and use it for any data type including user defined data types.
- For example, sort() can be written and used to sort any data type items. A class stack can be created that can be used as a stack of any data type.
- *What if we want a different code for a particular data type?*
- Consider a big project that needs a function sort() for arrays of many different data types. Let Quick Sort be used for all datatypes except char. In case of char, total possible values are 256 and counting sort may be a better option. Is it possible to use different code only when sort() is called for char data type?
- *It is possible in C++ to get a special behavior for a particular data type. This is called template specialization.*

```
// A generic sort function
template <class T>
void sort(T arr[], int size)
{
   // code to implement Quick Sort
}

// Template Specialization: A function
// specialized for char data type
template <>
void sort<char>(char arr[], int size)
{
   // code to implement counting sort
}
```

- ***How does template specialization work?***
- When we write any template based function or class, compiler creates a copy of that function/class whenever compiler sees that being used for a new data type or new set of data types(in case of multiple template arguments).
- If a specialized version is present, compiler first checks with the specialized version and then the main template. Compiler first checks with the most specialized version by matching the passed parameter with the data type(s) specified in a specialized version.

Template Argument Deduction

- Template argument deduction automatically deduces the data type of the argument passed to the class or function templates. This allows us to instantiate the template without explicitly specifying the data type.
- For example, consider the below function template to multiply two numbers:

```
template <typename t>
t multiply (t num1,t num2) { return num1*num2; }
```

In general, when we want to use the multiply() function for integers, we have to call it like this:

- multiply<int> (25, 5);

But we can also call it:

- multiply(23, 5);
- We don't explicitly specify the type ie 1,3 are integers.
- The same is true for the template classes(since C++17 only).
- Suppose we define the template class as:

```
template<typename t>
class student{
   private:
     t total_marks;
   public:
     student(t x) : total_marks(x) {}
};
```

- If we want to create an instance of this class, we can use any of the following syntax:
- student<int> stu1(23);
- or
- student stu2(24);
- ***Note:*** *It is important to note the the template argument deduction for a classes is only available since C++17*

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 vectors, lists, stacks, arrays, etc.
- The STL provides a range of containers, such as vectors, lists, and maps, as well as algorithms for searching, sorting and manipulating data.
- It is a library of container classes, algorithms, and iterators.
- All the three components have a rich set of pre-defined functions which help us in doing complicated tasks in very easy fashion.
- It is a generalized library and so, its components are parameterized.
- One of the key benefits of the STL is that it provides a way to write generic, reusable code that can be applied to different data types. This means that you can write an algorithm once, and then use it with different types of data without having to write separate code for each type.
- The STL also provides a way to write efficient code. Many of the algorithms and data structures in the STL are implemented using optimized algorithms, which can result in faster execution times compared to custom code.
- **Some of the key components of the STL include:**
- Containers: The STL provides a range of containers, such as vector, list, map, set, and stack, which can be used to store and manipulate data.
- Algorithms: The STL provides a range of algorithms, such as sort, find, and binary_search, which can be used to manipulate data stored in containers.

- Iterators: Iterators are objects that provide a way to traverse the elements of a container. The STL provides a range of iterators, such as forward_iterator, bidirectional_iterator, and random_access_iterator, that can be used with different types of containers.
- Function Objects: Function objects, also known as functors, are objects that can be used as function arguments to algorithms. They provide a way to pass a function to an algorithm, allowing you to customize its behavior.
- Adapters: Adapters are components that modify the behavior of other components in the STL. For example, the reverse_iterator adapter can be used to reverse the order of elements in a container.
- By using the STL, you can simplify your code, reduce the likelihood of errors, and improve the performance of your programs.
- **STL has 4 components:**
- **Algorithms**
- **Containers**
- **Functors**
- **Iterators**

Algorithms

- The header algorithm defines a collection of functions specially designed to be used on a range 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
- Sorting - There is a builtin function in C++ STL by the name of sort().
    - sort(startaddress, endaddress)
- Searching - Binary search is a widely used searching algorithm that requires the array to be sorted before search is applied. The main idea behind this algorithm is to keep dividing the array in half (divide and conquer) until the element is found, or all the elements are exhausted.

binary_search(startaddress, endaddress, valuetofind)

- STL has an ocean of algorithms, Some of the most used algorithms on vectors and most useful one's in Competitive Programming are mentioned as follows :
- **Non-Manipulating Algorithms**
- sort(first_iterator, last_iterator) – To sort the given vector.
- sort(first_iterator, last_iterator, greater<int>()) – To sort the given container/vector in descending order
- reverse(first_iterator, last_iterator) – To reverse a vector. (if ascending -> descending OR if descending -> ascending)
- *max_element (first_iterator, last_iterator) – To find the maximum element of a vector.
- *min_element (first_iterator, last_iterator) – To find the minimum element of a vector.
- accumulate(first_iterator, last_iterator, initial value of sum) – Does the summation of vector elements

Containers

- A container is a holder object that stores a collection of other objects (its elements). They are implemented as class templates, which allows great flexibility in the types supported as elements.

- The container manages the storage space for its elements and provides member functions to access them, either directly or through iterators (reference objects with similar properties to pointers).
- **Sequence containers -** Sequence containers implement data structures that can be accessed sequentially
    - array: Static contiguous array (class template)
    - vector: Dynamic contiguous array (class template)
    - deque: Double-ended queue (class template)
    - forward_list: Singly-linked list (class template)
    - list: Doubly-linked list (class template)
- **Associative containers** -Associative containers implement sorted data structures that can be quickly searched (O(log n) complexity).
    - Set: Collection of unique keys, sorted by keys (class template)
    - Map: Collection of key-value pairs, sorted by keys, keys are unique (class template).
    - multiset: Collection of keys, sorted by keys (class template)
    - multimap: Collection of key-value pairs, sorted by keys (class template)
- **Unordered associative containers -** Unordered associative containers implement unsorted (hashed) data structures that can be quickly searched.
    - unordered_set: Collection of unique keys, hashed by keys. (class template)
    - unordered_map: Collection of key-value pairs, hashed by keys, keys are unique. (class template)
    - unordered_multiset: Collection of keys, hashed by keys (class template)
    - unordered_multimap: Collection of key-value pairs, hashed by keys (class template)
- **Container adapters -** Container adapters provide a different interface for sequential containers.
    - stack: Adapts a container to provide stack (LIFO data structure) (class template).
    - queue: Adapts a container to provide queue (FIFO data structure) (class template).
    - priority_queue: Adapts a container to provide priority queue (class template).

Functors

- 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.
- 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

Iterators

- As the name suggests, iterators are used for working on a sequence of values. They are the major feature that allows generality in STL
- An **iterator** is an object (like a pointer) that points to an element inside the container. We can use iterators to move through the contents of the container. They can be visualized as something similar to a pointer pointing to some location and we can access the content at that particular location using them. Iterators play a critical role in connecting algorithm with containers along with the manipulation of data stored inside the containers. The most obvious form of an iterator is a pointer.
- Now each one of these iterators are not supported by all the containers in STL, different containers support different iterators, like vectors support Random-access iterators, while lists support bidirectional iterators. The whole list is as given below:

- Types of iterators: Based upon the functionality of the iterators, they can be classified into five major categories:
- Input Iterators: They are the weakest of all the iterators and have very limited functionality. They can only be used in a single-pass algorithms, i.e., those algorithms which process the container sequentially, such that no element is accessed more than once.
- Output Iterators: Just like input iterators, they are also very limited in their functionality and can only be used in single-pass algorithm, but not for accessing elements, but for being assigned elements.
- Forward Iterator: They are higher in the hierarchy than input and output iterators, and contain all the features present in these two iterators. But, as the name suggests, they also can only move in a forward direction and that too one step at a time.
- Bidirectional Iterators: They have all the features of forward iterators along with the fact that they overcome the drawback of forward iterators, as they can move in both the directions, that is why their name is bidirectional.
- Random-Access Iterators: They are the most powerful iterators. They are not limited to moving sequentially, as their name suggests, they can randomly access any element inside the container. They are the ones whose functionality are same as pointers.

Benefits of Iterators

- There are certainly quite a few ways which show that iterators are extremely useful to us and encourage us to use it profoundly. Some of the benefits of using iterators are as listed below:
- **Convenience in programming:** It is better to use iterators to iterate through the contents of containers as if we will not use an iterator and access elements using [ ] operator, then we need to be always worried about the size of the container, whereas with iterators we can simply use member function end() and iterate through the contents without having to keep anything in mind.
- A vector container (a C++ Standard Template) which is similar to an array with an exception that it automatically handles its own storage requirements in case it grows −
- Functions used in eg contEg1 prog

The push_back( ) member function inserts value at the end of the vector, expanding its size as needed.
The size( ) function displays the size of the vector.
The function begin( ) returns an iterator to the start of the vector.
The function end( ) returns an iterator to the end of the vector.

- The C++ Standard Library can be categorized into two parts −
- The Standard Function Library − This library consists of general-purpose,stand-alone functions that are not part of any class. The function library is inherited from C.
- The Object Oriented Class Library − This is a collection of classes and associated functions.
- I/O,
- String and character handling,
- Mathematical,
- Time, date, and localization,
- Dynamic allocation,
- Miscellaneous,
- Wide-character functions,

The Object Oriented Class Library

- Standard C++ Object Oriented Library defines an extensive set of classes that provide support for a number of common activities, including I/O, strings, and numeric processing. This library includes the following −
- The Standard C++ I/O Classes
- The String Class
- The Numeric Classes
- The STL Container Classes

- The STL Algorithms
- The STL Function Objects
- The STL Iterators
- The STL Allocators
- The Localization library
- Exception Handling Classes
- Miscellaneous Support Library

RTTI (Run-Time Type Information) in C++

- In C++, **RTTI (Run-time type information)** is a mechanism that exposes information about an object's data type at runtime and is available only for the classes which have at least one virtual function. It allows the type of an object to be determined during program execution.
- **Runtime Casts -** checks that the cast is valid, is the simplest approach to ascertain the runtime type of an object using a pointer or reference. This is especially beneficial when we need to cast a pointer from a base class to a derived type. When dealing with the inheritance hierarchy of classes, the casting of an object is usually required. There are two types of casting:
- **Upcasting:** When a pointer or a reference of a derived class object is treated as a base class pointer.
- **Downcasting:** When a base class pointer or reference is converted to a derived class pointer.
- **Using '*dynamic_cast*':** In an inheritance hierarchy, it is used for downcasting a base class pointer to a child class. On successful casting, it returns a pointer of the converted type and, however, it fails if we try to cast an invalid type such as an object pointer that is not of the type of the desired subclass.