OOPS with C++

UNIT-4

➢ STANDARD C++ CLASSES:

Standard C++ provides a rich set of classes in its Standard Template Library (STL) to facilitate common programming tasks. Some key classes include:

Vector (std::vector): Dynamic array that can resize itself.
List (std::list): Doubly-linked list for efficient insertions and removals.
Deque (std::deque): Double-ended queue with fast insertion/removal at both ends.
Queue (std::queue): Adapter class for a queue, typically implemented with std::deque or std::list.
Stack (std::stack): Adapter class for a stack, typically implemented with std::deque or std::list.
Map (std::map): Associative container that stores key-value pairs in a sorted order.
Set (std::set): Container that stores unique elements in a sorted order.
Unordered Map (std::unordered_map): Associative container like std::map but without ordering.
Unordered Set (std::unordered_set): Container like std::set but without ordering.
String (std::string): Sequence of characters with various utility functions.

These are just a few examples. The C++ Standard Library offers many more classes and algorithms to simplify programming tasks.

➢ PERSISTANT OBJECT

In C++, a persistent object typically refers to an object whose lifetime extends beyond the scope or duration of the function or block in which it is created. There are different ways to create persistent objects in C++, depending on the desired scope and storage duration. Here are a few options:

1. **Static Storage Duration:**

- Objects with static storage duration are created and initialized only once during the program's lifetime.
- They persist throughout the entire program execution.
- You can create a static object using the static keyword outside of any function or class.

Example of a static object with global scope

```
#include <iostream>

class PersistentObject {
public:
    PersistentObject() {
        std::cout << "PersistentObject constructor\n";
    }

    ~PersistentObject() {
        std::cout << "PersistentObject destructor\n";
    }
};

// Static object with global scope
static PersistentObject globalObject;

int main() {
    std::cout << "Inside main function\n";
    // The global Object will persist throughout the program
    return 0;
}
```

2. Dynamic Storage Duration (Heap Allocation):

- Objects created using new keyword on the heap have dynamic storage duration.
- They persist until explicitly deallocated using delete.

Example of a dynamically allocated object

```
#include <iostream>

class PersistentObject {
public:
    PersistentObject() {
        std::cout << "PersistentObject constructor\n";
    }

    ~PersistentObject() {
        std::cout << "PersistentObject destructor\n";
    }
};
```

```
int main() {
    std::cout << "Inside main function\n";
    // Dynamic object created on the heap
    PersistentObject *dynamicObject = new PersistentObject();

    // Explicitly deallocate the object to avoid memory leaks
    delete dynamicObject;
    return 0;
}
```

3. File Storage:

- You can use files to persistently store data between program executions.
- Read and write data to a file to achieve persistence.

Example of using files for persistence

```
#include <iostream>
#include <fstream>

class PersistentObject {
public:
    PersistentObject() {
        std::cout << "PersistentObject constructor\n";
    }

    ~PersistentObject() {
        std::cout << "PersistentObject destructor\n";
    }

    void saveToFile() {
        std::ofstream file("persistent_data.txt");
        file << "Some data related to the object\n";
        file.close();
    }

    void loadFromFile() {
        std::ifstream file("persistent_data.txt");
        if (file.is_open()) {
            std::string data;
            std::getline(file, data);
            std::cout << "Loaded data: " << data << "\n";
            file.close();
        }
    }
};
```

```
int main() {
   std::cout << "Inside main function\n";
   PersistentObject persistentObject;
   persistentObject.saveToFile();
   persistentObject.loadFromFile();
   return 0;
}
```

➢ STREAMS AND FILES

So far, we have been using the iostream standard library, which provides **cin** and **cout** methods for reading from standard input and writing to standard output respectively.

This topic will teach you how to read and write from a file. This requires another standard C++ library called **fstream**, which defines three new data types −

Sr.No	Data Type & Description
1	**ofstream** This data type represents the output file stream and is used to create files and to write information to files.
2	**ifstream** This data type represents the input file stream and is used to read information from files.
3	**fstream** This data type represents the file stream generally, and has the capabilities of both ofstream and ifstream which means it can create files, write information to files, and read information from files.

To perform file processing in C++, header files <iostream> and <fstream> must be included in your C++ source file.

Opening a File

A file must be opened before you can read from it or write to it. Either ofstream or fstream object may be used to open a file for writing. And ifstream object is used to open a file for reading purpose only.

Following is the standard syntax for open() function, which is a member of fstream, ifstream, and ofstream objects.

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

Here, the first argument specifies the name and location of the file to be opened and the second argument of the open() member function defines the mode in which the file should be opened.

Sr.No	Mode Flag & Description
1	**ios::app** Append mode. All output to that file to be appended to the end.
2	**ios::ate** Open a file for output and move the read/write control to the end of the file.
3	**ios::in** Open a file for reading.
4	**ios::out** Open a file for writing.
5	**ios::trunc** If the file already exists, its contents will be truncated before opening the file.

You can combine two or more of these values by ORing them together. For example if you want to open a file in write mode and want to truncate it in case that already exists, following will be the syntax –

ofstream outfile;

outfile.open("file.dat", ios::out | ios::trunc);

Similar way, you can open a file for reading and writing purpose as follows −

fstream afile;

afile.open("file.dat", ios::out | ios::in);

Closing a File

When a C++ program terminates it automatically flushes all the streams, release all the allocated memory and close all the opened files. But it is always a good practice that a programmer should close all the opened files before program termination.

Following is the standard syntax for close() function, which is a member of fstream, ifstream, and ofstream objects.

void close();

Writing to a File

While doing C++ programming, you write information to a file from your program using the stream insertion operator (<<) just as you use that operator to output information to the screen. The only difference is that you use an ofstream or fstream object instead of the cout object.

Reading from a File

You read information from a file into your program using the stream extraction operator (>>) just as you use that operator to input information from the keyboard. The only difference is that you use an ifstream or fstream object instead of the cin object.

Read and Write Example

Following is the C++ program which opens a file in reading and writing mode. After writing information entered by the user to a file named afile.dat, the program reads information from the file and outputs it onto the screen –

```
#include <fstream>
#include <iostream>
using namespace std;
int main () {
  char data[100];
  // open a file in write mode.
  ofstream outfile;
  outfile.open("afile.dat");
  cout << "Writing to the file" << endl;
  cout << "Enter your name: ";
  cin.getline(data, 100);

  // write inputted data into the file.
  outfile << data << endl;

  cout << "Enter your age: ";
  cin >> data;
  cin.ignore();
```

```
   // again write inputted data into the file.
   outfile << data << endl;

   // close the opened file.
   outfile.close();

   // open a file in read mode.
   ifstream infile;
   infile.open("afile.dat");

   cout << "Reading from the file" << endl;
   infile >> data;

   // write the data at the screen.
   cout << data << endl;

   // again read the data from the file and display it.
   infile >> data;
   cout << data << endl;

   // close the opened file.
   infile.close();

   return 0;
}
```

When the above code is compiled and executed, it produces the following sample input and **output −**

$./a.out

Writing to the file

Enter your name: XYZ

Enter your age: 9

Reading from the file

XYZ

9

File Position Pointers

Both istream and ostream provide member functions for repositioning the file-position pointer. These member functions are seekg ("seek get") for istream and seekp ("seek put") for ostream.

The argument to seekg and seekp normally is a long integer. A second argument can be specified to indicate the seek direction. The seek direction can be ios::beg (the default) for positioning relative to the beginning of a stream, ios::cur for positioning relative to the current position in a stream or ios::end for positioning relative to the end of a stream.

The file-position pointer is an integer value that specifies the location in the file as a number of bytes from the file's starting location. Some examples of positioning the "get" file-position pointer are −

// position to the nth byte of fileObject (assumes ios::beg)

fileObject.seekg(n);

// position n bytes forward in fileObject

fileObject.seekg(n, ios::cur);

// position n bytes back from end of fileObject

fileObject.seekg(n, ios::end);

// position at end of fileObject

fileObject.seekg(0, ios::end);

➢ NAMESPACES

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:

You can also avoid prepending of namespaces with the using namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace.
The namespace is thus implied for the following code:

```
#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;
}
```

Output:

Inside first_space

Names introduced in a using directive obey normal scope rules. The name is visible from the point of the using directive to the end of the scope in which the directive is found. Entities with the same name defined in an outer scope are hidden.

Nested Namespaces:

Namespaces can be nested where you can define one namespace inside another namespace as follows:

SYNTAX:

```
   namespace namespace_name1
   {
        // code declarations
        namespace namespace_name2
        {
          // code declarations
        }
   }
```

You can access members of nested namespace by using resolution operators as follows:

// to access members of namespace_name2
using namespace namespace_name1::namespace_name2;

// to access members of namespace_name1

using namespace namespace_name1;

In the above statements if you are using namespace_name1, then it will make elements of namespace_name2 available in the scope as follows:

```
#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::second_space;
int main ()
{
    // This calls function from second name space.
      func();

      return 0;
}
```

Output:

Inside second_space

Let us see how namespace scope the entities including variable and functions:

```
#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;
  }
}

int main ()
{
      // Calls function from first name space.
      first_space :: func();

    // Calls function from second name space.
      second_space :: func();
      return 0;
}
```

Output:

Inside first_space
Inside second_space

Consider the following program:

// A program to demonstrate need of namespace

```
int main()
{
   int value;
```

```
    value = 0;
    double value; // Error here
    value = 0.0;
}
```

Output:

Compiler Error:
'value' has a previous declaration as 'int value'

In each scope, a name can only represent one entity. So, there cannot be two variables with the same name in the same scope. Using namespaces, we can create two variables or member functions having the same name.

// Here we can see that more than one variable
// are being used without reporting any error.
// That is because they are declared in the
// different namespaces and scopes.

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

// Variable created inside namespace
namespace first {
int val = 500;
}

// Global variable
int val = 100;

int main()
{
    // Local variable
    int val = 200;

    // These variables can be accessed from
    // outside the namespace using the scope
    // operator ::
    cout << first::val << '\n';

    return 0;
}
```

Output:

500

Definition and Creation:

Namespaces allow us to group named entities that otherwise would have global scope into narrower scopes, giving them namespace scope. This allows organizing the elements of programs into different logical scopes referred to by names. Namespaces provide the space where we can define or declare identifiers i.e. names of variables, methods, classes, etc.

A namespace is a feature added in C++ and is not present in C.
A namespace is a declarative region that provides a scope to the identifiers (names of functions, variables or other user-defined data types) inside it.
Multiple namespace blocks with the same name are allowed. All declarations within those blocks are declared in the named scope.
A namespace definition begins with the keyword namespace followed by the namespace name as follows:

```
namespace namespace_name
{
   int x, y;
    // code declarations where
    // x and y are declared in
    // namespace_name's scope
}
```

Namespace declarations appear only at global scope.
Namespace declarations can be nested within another namespace.
Namespace declarations don't have access specifiers (Public or Private).
No need to give a semicolon after the closing brace of the definition of namespace.
We can split the definition of namespace over several units.

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 a function or a variable, prepend the namespace name as follows:

namespace_name: :code; **// code could be a variable, function or class.**

Let us see how namespace scope the entities including variable and functions:

```
#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;
    }
}

int main ()
{
    // Calls function from first name space.
    first_space::func();

    // Calls function from second name space.
    second_space::func();
    return 0;
}
```

// If we compile and run above code, this would produce the following result:
// Inside first_space
// Inside second_space

Output
Inside first_space
Inside second_space

The using directive:

You can avoid prepending of namespaces with the using namespace directive. This directive tells the compiler that the subsequent code is making use of names in the specified namespace. The namespace is thus implied for the following code:

```
#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;
}
```

// If we compile and run above code, this would produce the following result:
// Inside first_space

Output:
Inside first_space

Instead of accessing the whole namespace, another option (known as using declaration) is to access a particular item within a namespace. For example, if the only part of the std namespace that you intend to use is cout, you can refer to it as follows:

using std::cout;

Subsequent code can refer to cout without prepending the namespace, but other items in the std namespace will still need to be explicit as follows:

```
#include <iostream>
using std::cout;

int main ()
{
   cout << "std::endl is used with std!" << std::endl;
     return 0;
}
```

Output:

std::endl is used with std!

Names introduced in a using directive obey normal scope rules, i.e., they are visible from the point the using directive occurs to the end of the scope in which the directive is found. Entities with the same name defined in an outer scope are hidden.

Creating namespaces

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

// namespace ns1
namespace ns1 {
int value() { return 5; }
}

// namespace ns2
namespace ns2 {
const double x = 100;
double value() {
return 2 * x;
}
}

int main()
{
   // Access value function within ns1
   cout << ns1::value() << '\n';
```

```
    // Access value function within ns2
    cout << ns2::value() << '\n';

    // Access variable x directly
    cout << ns2::x << '\n';

    return 0;
}
```

Output:

5
200
100

Classes and Namespace:

The following is a simple way to create classes in a namespace:

A program to demonstrate use of class in a namespace

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

namespace ns
{
    // A Class in a namespace
    class geek
    {
    public:
        void display()
        {
            cout<<"ns::geek::display()"<<endl;;
        }
    };
}

int main()
{
    // Creating Object of geek Class
    ns::geek obj;

    obj.display();

    return 0;
}
```

Output

ns::geek::display()

A class can also be declared inside namespace and defined outside namespace using the following syntax:

A program to demonstrate use of class in a namespace

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

// namespace ns
namespace ns {
// Only declaring class here
 class geek;
}

// Defining class outside
class ns::geek {
public:
    void display() { cout << "ns::geek::display()\n"; }
};

int main()
{
    // Creating Object of geek Class
    ns::geek obj;
    obj.display();
    return 0;
}
```

Output

ns::geek::display()

We can define methods as well outside the namespace. The following is an example code:
A code to demonstrate that we can define methods outside namespace.

```
#include <iostream>
using namespace std;
// Creating a namespace
// namespace ns
namespace ns {
void display();
class geek {
```

```
public:
    void display();
};
}

// Defining methods of namespace
void ns::geek::display()
{
    cout << "ns::geek::display()\n";
}
void ns::display() { cout << "ns::display()\n"; }

// Driver code
int main()
{
    ns::geek obj;
    ns::display();
    obj.display();
    return 0;
}
```

Output:

ns::display()
ns::geek::display():

Nested Namespaces:

Namespaces can be nested, i.e., you can define one namespace inside another namespace as follows:

```
namespace namespace_name1 {
  // code declarations
  namespace namespace_name2 {
     // code declarations
  }
}
```

You can access the members of a nested namespace by using the resolution operator (::) as follows:

```
// to access members of namespace_name2
using namespace namespace_name1::namespace_name2;

// to access members of namespace:name1
```

using namespace namespace_name1;

In the above statements, if you are using namespace_name1, then it will make the elements of namespace_name2 available in the scope as follows:

```
#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::second_space;
int main ()
{

  // This calls function from second name space.
  func();

  return 0;
}
```

// If we compile and run above code, this would produce the following result:
// Inside second_space

Output:
Inside second_space

Namespace provides the advantage of avoiding name collision:-

For example, you might be writing some code that has a function called xyz() and there is another library available in your code which also has the 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;

- **EXCEPTION HANDLING**

One of the advantages of C++ over C is Exception Handling. Exceptions are runtime anomalies or abnormal conditions that a program encounters during its execution. There are two types of exceptions: a)Synchronous, b)Asynchronous (i.e., exceptions which are beyond the program's control, such as disc failure, keyboard interrupts etc.). C++ provides the following specialized keywords for this purpose:

Why Exception Handling?

The following are the main advantages of exception handling over traditional error handling:

1) **Separation of Error Handling code from Normal Code:**

   In traditional error handling codes, there are always if-else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try/catch blocks, the code for error handling becomes separate from the normal flow.

2) **Functions/Methods can handle 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.
   A function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).

3) **Grouping of Error Types:**

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.

C++ Exceptions:

When executing C++ code, different errors can occur: coding errors made by the programmer, errors due to wrong input, or other unforeseeable things.

When an error occurs, C++ will normally stop and generate an error message. The technical term for this is: C++ will throw an exception (error).

C++ try and catch:

Exception handling in C++ consists of three keywords: try, throw and catch:

The try statement allows you to define a block of code to be tested for errors while it is being executed.

The throw keyword throws an exception when a problem is detected, which lets us create a custom error.

The catch statement allows you to define a block of code to be executed if an error occurs in the try block.

The try and catch keywords come in pairs:

We use the try block to test some code: If the value of a variable "age" is less than 18, we will throw an exception, and handle it in our catch block.

In the catch block, we catch the error if it occurs and do something about it. The catch statement takes a single parameter. So, if the value of age is 15 and that's why we are throwing an exception of type int in the try block (age), we can pass "int myNum" as the parameter to the catch statement, where the variable "myNum" is used to output the value of age.

If no error occurs (e.g. if age is 20 instead of 15, meaning it will be greater than 18), the catch block is skipped.

Exception Handling in C++

1) The following is a simple example to show exception handling in C++. The output of the program explains the flow of execution of try/catch blocks.

```
#include <iostream>

using namespace std;

int main()

{

int x = -1;
```

```
// Some code
cout << "Before try \n";
try {
        cout << "Inside try \n";
        if (x < 0)
        {
                throw x;
                cout << "After throw (Never executed) \n";
        }
}
catch (int x ) {
        cout << "Exception Caught \n";
}
cout << "After catch (Will be executed) \n";
return 0;
}
```

- **GENERIC CLASSES:**

  Generics is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces. For example, classes like an array, map, etc, which can be used using generics very efficiently. We can use them for any type.

  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.

  The advantages of Generic Programming are:
  Code Reusability
  Avoid Function Overloading
  Once written it can be used for multiple times and cases.
  Generics can be implemented in C++ using Templates. Template is a simple and yet very powerful tool in C++. The simple idea is to pass 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 sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter.

Generic Functions using Template:
We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray()

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

// One function works for all data types.
// This would work even for user defined types
// if operator '>' is overloaded
template <typename T>

T myMax(T x, T y)
{
    return (x > y) ? x : y;
}

int main()
{

    // Call myMax for int
    cout << myMax<int>(3, 7) << endl;

    // call myMax for double
    cout << myMax<double>(3.0, 7.0) << endl;

    // call myMax for char
    cout << myMax<char>('g', 'e') << endl;

    return 0;
}
```

Output:

```
7
7
g
```

Generic Class using Template:
Like function templates, class templates are useful when a class defines something that is independent of data type. Can be useful for classes like LinkedList, binary tree, Stack, Queue, Array, etc.

Following is a simple example of template Array class.

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

template <typename T>
```

```
class Array {
private:
	T* ptr;
	int size;

public:
	Array(T arr[], int s);
	void print();
};

template <typename T>
Array<T>::Array(T arr[], int s)
{
	ptr = new T[s];
	size = s;
	for (int i = 0; i < size; i++)
		ptr[i] = arr[i];
}

template <typename T>
void Array<T>::print()
{
	for (int i = 0; i < size; i++)
		cout << " " << *(ptr + i);
	cout << endl;
}

int main()
{
	int arr[5] = { 1, 2, 3, 4, 5 };
	Array<int> a(arr, 5);
	a.print();
	return 0;
}
```

Output:

1 2 3 4 5

Working with multi-type Generics:

We can pass more than one data types as arguments to templates. The following example demonstrates the same.

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

template <class T, class U>
class A {
```

```
    T x;
    U y;

public:
    A()
    {
        cout << "Constructor Called" << endl;
    }
};

int main()
{
    A<char, char> a;
    A<int, double> b;
    return 0;
}
```

Output:

Constructor Called
Constructor Called

- **STANDARD TEMPLATE LIBRARY**

  The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized. Working knowledge of template classes is a prerequisite for working with STL.

  The C++ Standard Template Library (STL) is a collection of algorithms, data structures, and other components that can be used to simplify the development of C++ programs. The STL provides a range of containers, such as vectors, lists, and maps, as well as algorithms for searching, sorting and manipulating data.

  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

1. 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
- valarray class

2. Containers

Containers or container classes store objects and data. There are in total seven standards "first-class" container classes and three container adaptor classes and only seven header files that provide access to these containers or container adaptors.

Sequence Containers: implement data structures that can be accessed in a sequential manner.

- vector
- list
- deque
- arrays
- forward_list

Container Adaptors: provide a different interface for sequential containers.

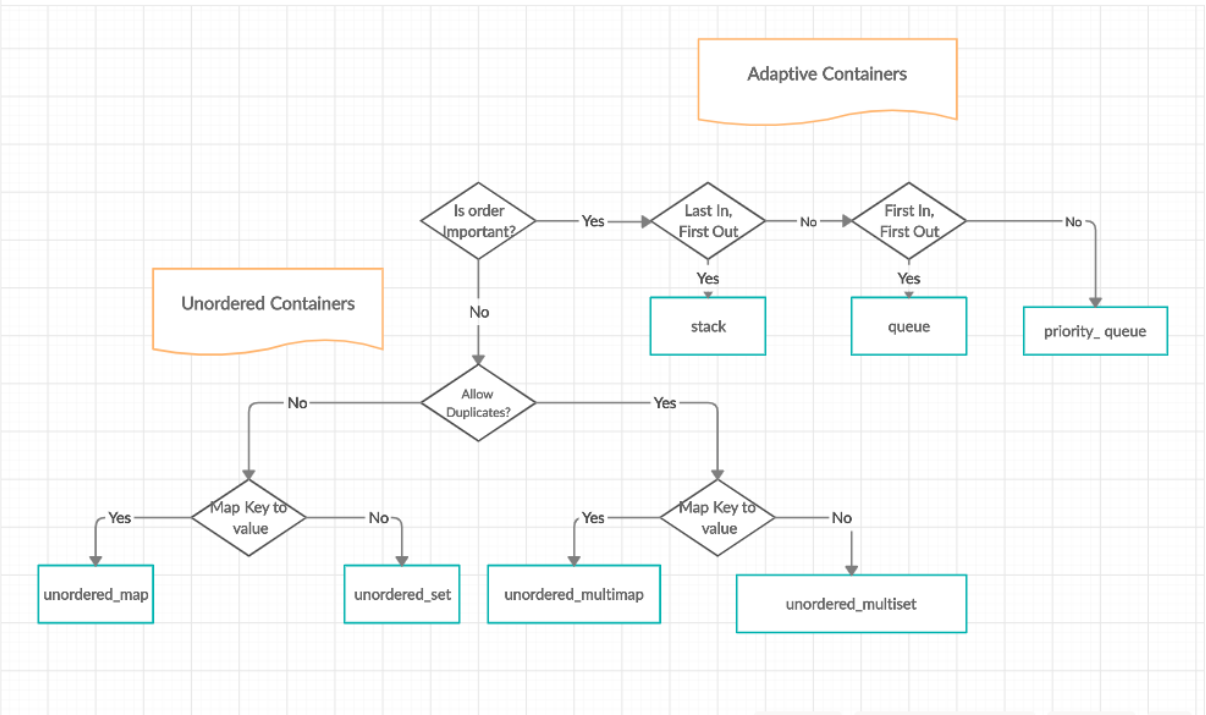
- queue
- priority_queue
- stack

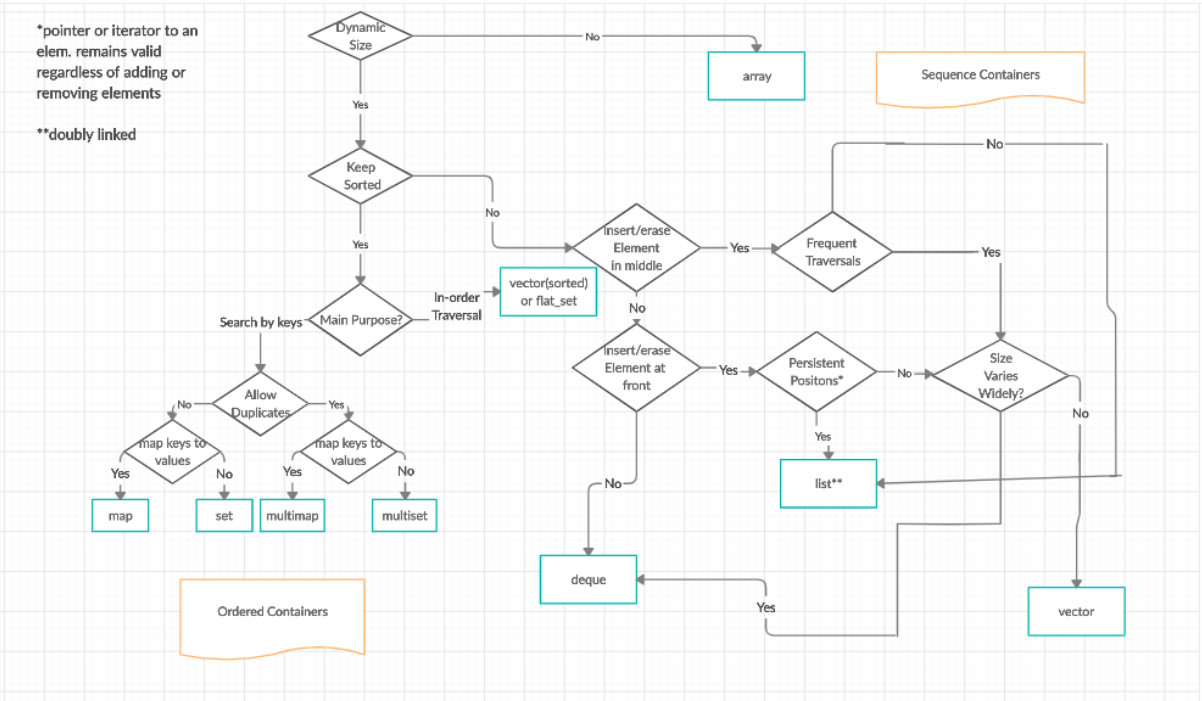
Associative Containers: implement sorted data structures that can be quickly searched (O(log n) complexity).

- set
- multiset
- map
- multimap

Unordered Associative Containers: implement unordered data structures that can be quickly searched

- unordered_set
- unordered_multiset
- unordered_map
- unordered_multimap





3. 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.

4. 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.

Utility Library

Defined in header <utility>.

Advantages of the C++ Standard Template Library (STL):

- **Reusability:** One of the key advantages of the STL is that it provides a way to write generic, reusable code that can be applied to different data types. This can lead to more efficient and maintainable code.
- **Efficient algorithms:** 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.
- **Improved code readability:** The STL provides a consistent and well-documented way of working with data, which can make your code easier to understand and maintain.
- **Large community of users:** The STL is widely used, which means that there is a large community of developers who can provide support and resources, such as tutorials and forums.

Disadvantages of the C++ Standard Template Library (STL):

- **Learning curve:** The STL can be difficult to learn, especially for beginners, due to its complex syntax and use of advanced features like iterators and function objects.
- **Lack of control:** When using the STL, you have to rely on the implementation provided by the library, which can limit your control over certain aspects of your code.
- **Performance:** In some cases, using the STL can result in slower execution times compared to custom code, especially when dealing with small amounts of data.

➢ MANIPULATORS

Manipulators are helping functions that can modify the input/output stream. It does not mean that we change the value of a variable, it only modifies the I/O stream using insertion (<<) and extraction (>>) operators.

Manipulators are special functions that can be included in the I/O statement to alter the format parameters of a stream.
Manipulators are operators that are used to format the data display.
To access manipulators, the file iomanip.h should be included in the program.

For example, if we want to print the hexadecimal value of 100 then we can print it as:

```
cout<<setbase(16)<<100
```

Types of Manipulators

There are various types of manipulators:

1. **Manipulators without arguments: The most important manipulators defined by the IOStream library are provided below.**

endl: It is defined in ostream. It is used to enter a new line and after entering a new line it flushes (i.e. it forces all the output written on the screen or in the file) the output stream.
ws: It is defined in istream and is used to ignore the whitespaces in the string sequence.
ends: It is also defined in ostream and it inserts a null character into the output stream. It typically works with **std::ostrstream,** when the associated output buffer needs to be null-terminated to be processed as a C string.
flush: It is also defined in ostream and it flushes the output stream, i.e. it forces all the output written on the screen or in the file. Without flush, the output would be the same, but may not appear in real-time. **Examples:**

```
#include <iostream>
#include <istream>
#include <sstream>
#include <string>
```

```
using namespace std;

int main()
{
  istringstream str("                Programmer");
  string line;
  // Ignore all the whitespace in string
  // str before the first word.
  getline(str >> std::ws, line);

  // you can also write str>>ws
  // After printing the output it will automatically
  // write a new line in the output stream.
  cout << line << endl;

  // without flush, the output will be the same.
  cout << "only a test" << flush;

  // Use of ends Manipulator
  cout << "\na";

  // NULL character will be added in the Output
  cout << "b" << ends;
  cout << "c" << endl;

  return 0;
}
```

Output:
Programmer
only a test
abc

2.Manipulators with Arguments: Some of the manipulators are used with the argument like setw (20), setfill (‘*’), and many more. These all are defined in the header file. If we want to use these manipulators then we must include this header file in our program. **For Example**, you can use following manipulators to set minimum width and fill the empty space with any character you want: std::cout << std::setw (6) << std::setfill (’*’);

Some important manipulators in <iomanip> are:
setw (val): It is used to set the field width in output operations.
setfill (c): It is used to fill the character ‘c’ on output stream.
setprecision (val): It sets val as the new value for the precision of floating-point values.
setbase(val): It is used to set the numeric base value for numeric values.
setiosflags(flag): It is used to set the format flags specified by parameter mask.

resetiosflags(m): It is used to reset the format flags specified by parameter mask.

Some important manipulators in <ios> are:
showpos: It forces to show a positive sign on positive numbers.
noshowpos: It forces not to write a positive sign on positive numbers.
showbase: It indicates the numeric base of numeric values.
uppercase: It forces uppercase letters for numeric values.
nouppercase: It forces lowercase letters for numeric values.
fixed: It uses decimal notation for floating-point values.
scientific: It uses scientific floating-point notation.
hex: Read and write hexadecimal values for integers and it works same as the setbase(16).
dec: Read and write decimal values for integers i.e. setbase(10).
oct: Read and write octal values for integers i.e. setbase(10).
left: It adjusts output to the left.
right: It adjusts output to the right.

There are two types of manipulators used generally:
1] Parameterized and
2] Non-parameterized

1] Parameterized Manipulators:-

Manipulator	->	Meaning
setw (int n)	->	To set field width to n
setprecision (int p)	->	The precision is fixed to p
setfill (Char f)	->	To set the character to be filled
setiosflags (long l)	->	Format flag is set to l
resetiosflags (long l)	->	Removes the flags indicated by l
Setbase(int b)	->	To set the base of the number to b

setw () is a function in Manipulators in C++:

The setw() function is an output manipulator that inserts whitespace between two variables. You must enter an integer value equal to the needed space.

Syntax:

```
setw ( int n)
As an example,
int a=15;  int b=20;
cout << setw(10) << a << setw(10) << b << endl;
```

setfill() is a function in Manipulators in C++:
It replaces setw(whitespaces)'s with a different character. It's similar to setw() in that it manipulates output, but the only parameter required is a single character.
Syntax:

```
setfill(char ch)
```

Example:

```
int a,b;
a=15;   b=20;
cout<< setfill('*') << endl;
        cout << setw(5) << a << setw(5) << a << endl;
```

- **setprecision()** is a function in Manipulators in C++:

It is an output manipulator that controls the number of digits to display after the decimal for a floating point integer.

Syntax:

```
setprecision (int p)
```

Example:

```
float A = 1.34255;

cout <<fixed<< setprecision(3) << A << endl;
```

setbase() is a function in Manipulators in C++:

The setbase() manipulator is used to change the base of a number to a different value. The following base values are supported by the C++ language:

- hex (Hexadecimal = 16)
- oct (Octal = 8)
- dec (Decimal = 10)

The manipulators hex, oct, and dec can change the basis of input and output numbers.

// Example:

```
#include <iostream>
#include <iomanip>

using namespace std;
main()
{

int number = 100;
cout << "Hex Value =" << " " << hex << number << endl;
cout << "Octal Value=" << " " << oct << number << endl;
cout << "Setbase Value=" << " " << setbase(8) << number << endl;
cout << "Setbase Value=" << " " << setbase(16) << number << endl;
return 0;

}
```

Output

Hex Value = 64

Octal Value= 144

Setbase Value= 144

Setbase Value= 64

2] Non-parameterized
Examples are endl, fixed, showpoint and flush.
• endl – Gives a new line
• ends – Adds null character to close an output string
• flush – Flushes the buffer stream
• ws – Omits the leading white spaces present before the first field
• hex, oct, dec – Displays the number in hexadecimal or octal or in decimal format

// **Example:** ws – Omits the leading white spaces present before the first field

```
#include<iostream>

using namespace std;

int main()

{

        char name[125];

        cout << "Enter your name" << endl;

        cin >> ws;

        cin.getline(name,125);

        cout << name << endl;

        return 0;

}
```

Output:

```
Enter your name

ram

ram
```

- **VECTORS**
  Vectors are the same as dynamic arrays with the ability to resize itself automatically when an element is inserted or deleted, with their storage being handled automatically by the container. Vector elements are placed in contiguous storage so that they can be accessed and traversed using iterators. In vectors, data is inserted at the end. Inserting at the end takes differential time, as sometimes the array may need to be extended. Removing the last element takes only constant time because no resizing happens. Inserting and erasing at the beginning or in the middle is linear in time.

  What is std::vector in C++?
  std::vector in C++ is the class template that contains the vector container and its member functions. It is defined inside the <vector> header file. The member functions of std::vector class provide various functionalities to vector containers. Some commonly used member functions are written below:

Iterators

begin() – Returns an iterator pointing to the first element in the vector
end() – Returns an iterator pointing to the theoretical element that follows the last element in the vector
rbegin() – Returns a reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
rend() – Returns a reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)
cbegin() – Returns a constant iterator pointing to the first element in the vector.
cend() – Returns a constant iterator pointing to the theoretical element that follows the last element in the vector.
crbegin() – Returns a constant reverse iterator pointing to the last element in the vector (reverse beginning). It moves from last to first element
crend() – Returns a constant reverse iterator pointing to the theoretical element preceding the first element in the vector (considered as reverse end)

```
// C++ program to illustrate the
// iterators in vector
#include <iostream>
#include <vector>

using namespace std;

int main()
{
	vector<int> g1;

	for (int i = 1; i <= 5; i++)
		g1.push_back(i);

	cout << "Output of begin and end: ";
	for (auto i = g1.begin(); i != g1.end(); ++i)
		cout << *i << " ";

	cout << "\nOutput of cbegin and cend: ";
	for (auto i = g1.cbegin(); i != g1.cend(); ++i)
		cout << *i << " ";

	cout << "\nOutput of rbegin and rend: ";
	for (auto ir = g1.rbegin(); ir != g1.rend(); ++ir)
		cout << *ir << " ";

	cout << "\nOutput of crbegin and crend : ";
	for (auto ir = g1.crbegin(); ir != g1.crend(); ++ir)
		cout << *ir << " ";

	return 0;
}
```

Output:

Output of begin and end: 1 2 3 4 5

Output of cbegin and cend: 1 2 3 4 5
Output of rbegin and rend: 5 4 3 2 1
Output of crbegin and crend : 5 4 3 2 1

Capacity

size() – Returns the number of elements in the vector.
max_size() – Returns the maximum number of elements that the vector can hold.
capacity() – Returns the size of the storage space currently allocated to the vector expressed as number of elements.
resize(n) – Resizes the container so that it contains ‘n’ elements.
empty() – Returns whether the container is empty.
shrink_to_fit() – Reduces the capacity of the container to fit its size and destroys all elements beyond the capacity.
reserve() – Requests that the vector capacity be at least enough to contain n elements.

```
// C++ program to illustrate the
// capacity function in vector
#include <iostream>
#include <vector>

using namespace std;
int main()
{
	vector<int> g1;

	for (int i = 1; i <= 5; i++)
		g1.push_back(i);

	cout << "Size : " << g1.size();
	cout << "\nCapacity : " << g1.capacity();
	cout << "\nMax_Size : " << g1.max_size();

	// resizes the vector size to 4
	g1.resize(4);

	// prints the vector size after resize()
	cout << "\nSize : " << g1.size();

	// checks if the vector is empty or not
	if (g1.empty() == false)
		cout << "\nVector is not empty";
	else
		cout << "\nVector is empty";

	// Shrinks the vector
	g1.shrink_to_fit();
	cout << "\nVector elements are: ";
	for (auto it = g1.begin(); it != g1.end(); it++)
		cout << *it << " ";
```

```
	return 0;
}
```

Output:
Size : 5
Capacity : 8
Max_Size : 4611686018427387903
Size : 4
Vector is not empty
Vector elements are: 1 2 3 4

Element access
reference operator [g] – Returns a reference to the element at position ‘g’ in the vector
at(g) – Returns a reference to the element at position ‘g’ in the vector
front() – Returns a reference to the first element in the vector
back() – Returns a reference to the last element in the vector
data() – Returns a direct pointer to the memory array used internally by the vector to store its owned elements.

```
// C++ program to illustrate the
// element access in vector
#include <bits/stdc++.h>
using namespace std;

int main()
{
	vector<int> g1;

	for (int i = 1; i <= 10; i++)
		g1.push_back(i * 10);

	cout << "\nReference operator [g] : g1[2] = " << g1[2];

	cout << "\nat : g1.at(4) = " << g1.at(4);

	cout << "\nfront() : g1.front() = " << g1.front();

	cout << "\nback() : g1.back() = " << g1.back();

	// pointer to the first element
	int* pos = g1.data();

	cout << "\nThe first element is " << *pos;
	return 0;
}
```

Output:
Reference operator [g] : g1[2] = 30
at : g1.at(4) = 50

```
front() : g1.front() = 10
back() : g1.back() = 100
The first element is 10
```

Modifiers

assign() – It assigns new value to the vector elements by replacing old ones
push_back() – It push the elements into a vector from the back
pop_back() – It is used to pop or remove elements from a vector from the back.
insert() – It inserts new elements before the element at the specified position
erase() – It is used to remove elements from a container from the specified position or range.
swap() – It is used to swap the contents of one vector with another vector of same type. Sizes may differ.
clear() – It is used to remove all the elements of the vector container
emplace() – It extends the container by inserting new element at position
emplace_back() – It is used to insert a new element into the vector container, the new element is added to the end of the vector

```
// C++ program to illustrate the
// Modifiers in vector
#include <bits/stdc++.h>
#include <vector>
using namespace std;

int main()
{
  // Assign vector
  vector<int> v;

  // fill the vector with 10 five times
  v.assign(5, 10);

  cout << "The vector elements are: ";
  for (int i = 0; i < v.size(); i++)
          cout << v[i] << " ";

  // inserts 15 to the last position
  v.push_back(15);
  int n = v.size();
  cout << "\nThe last element is: " << v[n - 1];

  // removes last element
  v.pop_back();

  // prints the vector
  cout << "\nThe vector elements are: ";
```

```
for (int i = 0; i < v.size(); i++)
	cout << v[i] << " ";

// inserts 5 at the beginning
v.insert(v.begin(), 5);

cout << "\nThe first element is: " << v[0];

// removes the first element
v.erase(v.begin());

cout << "\nThe first element is: " << v[0];

// inserts at the beginning
v.emplace(v.begin(), 5);
cout << "\nThe first element is: " << v[0];

// Inserts 20 at the end
v.emplace_back(20);
n = v.size();
cout << "\nThe last element is: " << v[n - 1];

// erases the vector
v.clear();
cout << "\nVector size after clear(): " << v.size();

// two vector to perform swap
vector<int> v1, v2;
v1.push_back(1);
v1.push_back(2);
v2.push_back(3);
v2.push_back(4);

cout << "\n\nVector 1: ";
for (int i = 0; i < v1.size(); i++)
	cout << v1[i] << " ";

cout << "\nVector 2: ";
for (int i = 0; i < v2.size(); i++)
	cout << v2[i] << " ";

// Swaps v1 and v2
v1.swap(v2);

cout << "\nAfter Swap \nVector 1: ";
for (int i = 0; i < v1.size(); i++)
```

```
        cout << v1[i] << " ";

  cout << "\nVector 2: ";
  for (int i = 0; i < v2.size(); i++)
        cout << v2[i] << " ";
}
```

Output:

```
The vector elements are: 10 10 10 10 10
The last element is: 15
The vector elements are: 10 10 10 10 10
The first element is: 5
The first element is: 10
The first element is: 5
The last element is: 20
Vector size after erase(): 0

Vector 1: 1 2
Vector 2: 3 4
After Swap
Vector 1: 3 4
Vector 2: 1 2
```

The time complexity for doing various operations on vectors is-

Random access – constant O(1)
Insertion or removal of elements at the end – constant O(1)
Insertion or removal of elements – linear in the distance to the end of the vector O(N)
Knowing the size – constant O(1)
Resizing the vector- Linear O(N)