Agenda

- STL
- Operator Overloading
- ~~Nested and Local class~~

STL

- We can not divide template code into multiple files.
- Standard Template Library(STL) is a collection of readymade template data structure classes and algorithms.
- 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.
- STL has 4 components:

    1. Containers
    2. Algorithms
    3. Function Objects
    4. Iterators

1. Container

- Containers or container classes to store objects and data.
- The C++ container library categorizes containers into four types:

1. Sequence containers
2. Associative containers
3. Unordered associative containers
4. Sequence container adapters

```
STL
│
├── Containers
   ├── Sequence Containers
   │   ├── vector
   │   ├── deque
   │   ├── list
   │   └── forward_list
   ├── Associative Containers
   │   ├── set
   │   ├── multiset
```

```
    │   ├── map
    │   └── multimap
    ├── Unordered Associative Containers
    │   ├── unordered_set
    │   ├── unordered_multiset
    │   ├── unordered_map
    │   └── unordered_multimap
    └── Container Adapters
        ├── stack
        ├── queue
        └── priority_queue
```

2. Algorithm

- They act on containers and provide means for various operations for the contents of the containers.
    - Sorting
    - Searching

3. Functions

- The STL includes classes that overload the function call operator.
- Instances of such classes are called function objects or functors.
- Functors allow the working of the associated function to be customized with the help of parameters to be passed.

4. Iterators

- As the name suggests, iterators are used for working upon a sequence of values.
- They are the major feature that allows generality in STL.
- Iterators are used to point at the memory addresses of STL containers.
- They are primarily used in sequences of numbers, characters etc.
- They reduce the complexity and execution time of the program.

Sequence Containers

- Sequence containers are used for data structures that store objects of the same type in a linear manner.
- The STL Sequence Container types are:
- vector: A dynamic array that can grow and shrink in size. It provides fast random access to elements and efficient insertion and deletion at the end.
- deque: A double-ended queue that supports efficient insertion and deletion at both ends. It provides similar functionality to vector but may have better performance for inserting and deleting elements at the beginning.
- list: A doubly-linked list that allows for efficient insertion and deletion of elements at any position. It does not provide random access to elements and has slower traversal compared to vector and deque.

- forward_list: A singly-linked list that provides similar functionality to list but with reduced memory overhead. It allows for efficient insertion and deletion at the beginning and after an element.

Associative Containers

- Associative containers implement sorted data structures that can be quickly searched.
- set : collection of unique keys, sorted by keys
- map : collection of key-value pairs, sorted by keys, keys are unique
- multiset : collection of keys, sorted by keys
- multimap : collection of key-value pairs, sorted by keys

Container adaptors

- Container adaptors provide a different interface for sequential containers.
- stack : adapts a container to provide stack (LIFO data structure)
- queue : adapts a container to provide queue (FIFO data structure)

Vector

- 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.
- When the vector needs to grow beyond its current capacity, it typically doubles its capacity.
- Inserting and erasing at the beginning or in the middle is linear in time.
- the iterator in the vector is a Random Access Iterator that Supports all iterator operations including arithmetic (e.g., +, -), comparison (<, >, etc.), and dereferencing (*, []).

map

- map is a sorted associative container that contains key-value pairs with unique keys.
- Keys are sorted by using the comparison function Compare.
- Search, removal, and insertion operations have logarithmic complexity.
- Maps are usually implemented as Red–black trees
- Iterators of map iterate in ascending order of keys, where ascending is defined by the comparison that was used for construction.
- Iterator of map are bidirectional iterator that supports dereferencing (*, ->) and bidirectional movement (++, --).

iterator

- An iterator in C++ is an object that enables traversal over the elements of a container (such as std::vector, std::list, std::map, etc.) and provides access to these elements.

- Iterators are a fundamental part of the Standard Template Library (STL) and are designed to abstract the concept of element traversal, making it possible to work with different containers in a consistent

manner.

- Key Characteristics of Iterators:

1. Traversal: Iterators allow you to move through the elements of a container, one element at a time.
2. Access: Iterators provide access to the element they point to, typically through the dereference operator (*).
3. Type-Specific: Iterators are strongly typed, meaning that an iterator for an std::vector will be different from an iterator for an std::list.

Types of Iterators:

1. Input Iterators:

- Can read elements from a container. Only allow single-pass access (i.e., you can only move forward through the container).

2. Output Iterators:

- Can write elements to a container.
- Also allow only single-pass access.

3. Forward Iterators:

- Can read and write elements.
- Support multi-pass traversal, meaning you can go through the container multiple times.

4. Bidirectional Iterators:

- Can move both forward and backward in a container.
- Support all operations of forward iterators, with the additional ability to decrement the iterator.

5. Random Access Iterators:

- Provide all the capabilities of bidirectional iterators.
- Allow direct access to any element in the container using arithmetic operations like addition and subtraction.

Common Operations on Iterators:

- Dereferencing (*): Access the element the iterator points to.
- Incrementing (++): Move the iterator to the next element.
- Decrementing (--): Move the iterator to the previous element (not supported by input or output iterators).
- Equality/Inequality (==, !=): Compare iterators to check if they point to the same position.
- Addition/Subtraction (+, -): For random access iterators, allows moving the iterator by a specific number of elements.

Operator Overloading

- operator is token in C/C++.
- It is used to generate expression.

- operator is keyword in C++.
- Types of operator:
    1. Unary operator
    2. Binary Operator
    3. Ternary operator
- Unary Operator:
    - If operator require only one operand then it is called unary operator.
    - example : Unary(+,-,*) , &, !, ~, ++, --, sizeof, typeid etc.
- Binary Operator:
    - If operator require two operands then it is called binary operator.
    - Example:
        1. Arithmetic operator
        2. Relational operator
        3. Logical operator
        4. Bitwise operator
        5. Assignment operator
- Ternary operator:
    - If operator require three operands then it is called ternary operator.
    - Example:
        - Conditional operator(? :)
- In C/C++, we can use operator with objects of fundamental type directly.(No need to write extra code).

```
int num1 = 10; //Initialization
int num2 = 20; //Initialization
int num3 = num1 + num2; //OK
```

- In C++, also we can not use operator with objects of user defined type directly.
- If we want to use operator with objects of user defined type then we should overload operator.

```
class Point
{
    int x;
    int y;
};
int main( void )
{
    struct Point pt1 = { 10,20};
    struct Point pt2 = { 30,40};
    struct Point pt3;
    pt3 = pt1 + pt2; //Not OK
    //pt3.x = pt1.x + pt2.x;
    //pt3.y = pt1.y + pt2.y;
return 0;
}
```

- If we want to use operator with objects of user defined type then we should overload operator.
- To overload operator, we should define operator function.
- We can define operator function using 2 ways
    1. Using member function
    2. Using non member function.
- By defining operator function, it is possible to use operator with objects of user defined type. This process of giving extension to the meaning of operator is called operator overloading.
- Using operator overloading we can not define user defined operators rather we can increase capability of existing operators.

Limitations of operator overloading

- We can not overloading following operator using member as well as non member function:

    1. dot/member selection operator(.)
    2. Pointer to member selectiion operator(.*)
    3. Scope resolution operator(::)
    4. Ternary/conditional operator(? :)
    5. sizeof() operator
    6. typeid() operator
    7. static_cast operator
    8. dynamic_cast operator
    9. const_cast operator
    10. reinterpret_cast operator

- We can not overload following operators using non member function:

    1. Assignment operator(=)
    2. Subscript / Index operator([])
    3. Function Call operator[()]
    4. Arrow / Dereferencing operator(->)

- Using operator overloading, we can change meaning of operator.

- Using operator overloading, we can not change number of parameters passed to the operator function.

Operator overloading using member function(operator function must be member function)

- If we want to overload, binary operator using member function then operator function should take only one parameter.
- Using operator overloading, we can not change, precedance and associativity of the operator.
- If we want to overload unary operator using member function then operator function should not take any parameter.

```
c3 = c1 + c2; //c3 = c1.operator+(c2);

c4 = c1 + c2 + c3; //c4 = c1.operator+( c2 ).operator+( c3 );
```

Operator overloading using non member function(operator function must be global function)

- If we want to overload binary operator using non member function then operator function should take two parameters.
- If we want to overload unary operator using non member function then operator function should take only one parameters.

```
c3 = c1 + c2; //c3 = operator+(c1,c2);

c4 = c1 + c2 + c3; //c4 = operator+(operator+(c1,c2),c3);

c2 = ++ c1; //c2=operator++( c1 );
```

Overloading Insertion Operator(<<)

-cout is an external object of ostream class which is declared in std namespace.

- ostream class is typdef of basic_ostream class.
- If we want print state of object on console(monitor) then we should use cout object and insertion operator(<<).
- Copy constructor of ostream class is private hence we can not copy of cout object inside our program
- If we want to avoid copy then we should use reference.
- If we want to print state of object(of structure/class) on console then we should overload insertion operator.

```
//ostream out = cout; // NOT OK
ostream &out = cout; //OK
```

```
1. cout<<c1; //cout.operator<<( c1 );
2. cout<<c1; //operator<<(cout, c1 );
```

- According to first statement, to print state of c1 on console, we should define operator<<( ) function inside ostream class. But ostream class is library defined class hence we should not modify its implementation.
- According to second statement, to print state of c1 on console, we should define operator<<() function globally. Which possible for us. Hence we should overload operator<<( ) using non member function.

```
class ClassName
{
  friend ostream& operator<<( ostream &cout, ClassName &other );
};
```

```
ostream& operator<<( ostream &cout, ClassName &other )
{
  //TODO : print state of object using other
  return cout;
}
```

Overloading Extraction Operator(>>)

- cin stands for character input. It represents keyboard.
- cin is external object of istream class which is declared in std namespace.
- istream class is typedef of basic_istream class.
- If we want to accept data/state of the variable/object from console/keyboard then we should use cin object and extraction operator.
- Copy constructor of istream class is private hence, we can not create copy of cin object in out program.
- To avoid copy, we should use reference.

```
istream in = cin; // NOT OK
istream &in = cin; // OK
```

- If we want to accept state of object (of structure/class) from console(keyboard) then we should overload extraction operator.

```
1. cin>>c1; //cin.operator>>( c1 )

2. cin>>c1;//operator>>( cin, c1 );
```

- According to first statement, to accept state of c1 from console, we should define operator>>( ) function inside istream class. But istream class is library defined class hence we should not modify its implementation.
- According to second statement, to accept state of c1 from console, we should define operator>>( ) function globally. Which possible for us. Hence we should overload operator>>( ) using non member function.

```
class ClassName
{
  friend istream& operator>>( istream &cin, ClassName &other );
};
istream& operator>>( istream &cin, ClassName &other )
{
  //TODO : accept state of object using other
  return cin;
}
```

Index/Subscript Operator Overloading

- If we want to overcome limitations of array then we should encapsulate array inside class and we should perform operations on object by considering it array.
- If we want to consider object as a array then we should overload sub script/index operator.

```
//Array *const this = &a1
int& operator[]( int index )throw( ArrayIndexOutOfBoundsException )
{
  if( index >= 0 && index < SIZE )
    return this->arr[ index ];
  throw ArrayIndexOutOfBoundsException("ArrayIndexOutOfBoundsException");
}

//If we use subscript operator with object at RHS of assignment operator then
expression must return value from array.

Array a1;
cin>>a1; //operator>>( cin, a1 );
cout<<a1; //opeator<<( cout, a1 );
int element = a1[ 2 ]; //int element = a1.operator[]( 1 );

// If we want to use sub script operator with object at LHS of assignment operator
then expression should not return a value rather it should return either address /
reference of memory location.

Array a1;
cin>>a1; //operator>>( cin, a1 );
a1[ 1 ] = 200; //a1.operator[]( 1 ) = 200;
cout<<a1; //opeator<<( cout, a1 );
```

Overloading assignment operator.

- If we initialize newly created object from existing object of same class then copy constructor gets called.
- If we assign, object to the another object then assignment operator function gets called.

```
Complex c1(10,20);
Complex c2 = c1; //On c2 copy ctor will call

Complex c1(10,20);
Complex c2;
c2 = c1; //c2.operator=( c1 )
```

```
class ClassName
{
public:
  ClassName& operator=( const ClassName &other )
{
  //TODO : Shallow/Deep Copy
```

```
    return *this;
}
};
```

- If we do not define assignment operator function inside class then compiler generates default assignment operator function for the class. By default it creates shallow Copy.
- During assignment, if there is need to create deep copy then we should overload assignment operator function.