**Links**

<https://beginnersbook.com/2017/09/cpp-encapsulation/>

<https://www.geeksforgeeks.org/how-does-a-c-program-executes/>

<https://www.geeksforgeeks.org/memory-leak-in-c-and-how-to-avoid-it/>

<https://www.tutorialspoint.com/cplusplus/cpp_multithreading.htm>

**Software Tool Chain**

C code 🡪 Compiler 🡪 Assembler 🡪 Linker 🡪 Debugger 🡪 Processor

Compiler – Converts high level code in to low level code.

Assembler – Converts assembly code into object code or machine code.

Linker – Link all pre-defined libraries used in the code.

**Allocation of program variables on memory stack**

4 parts of memory from top to bottom

**Heap**

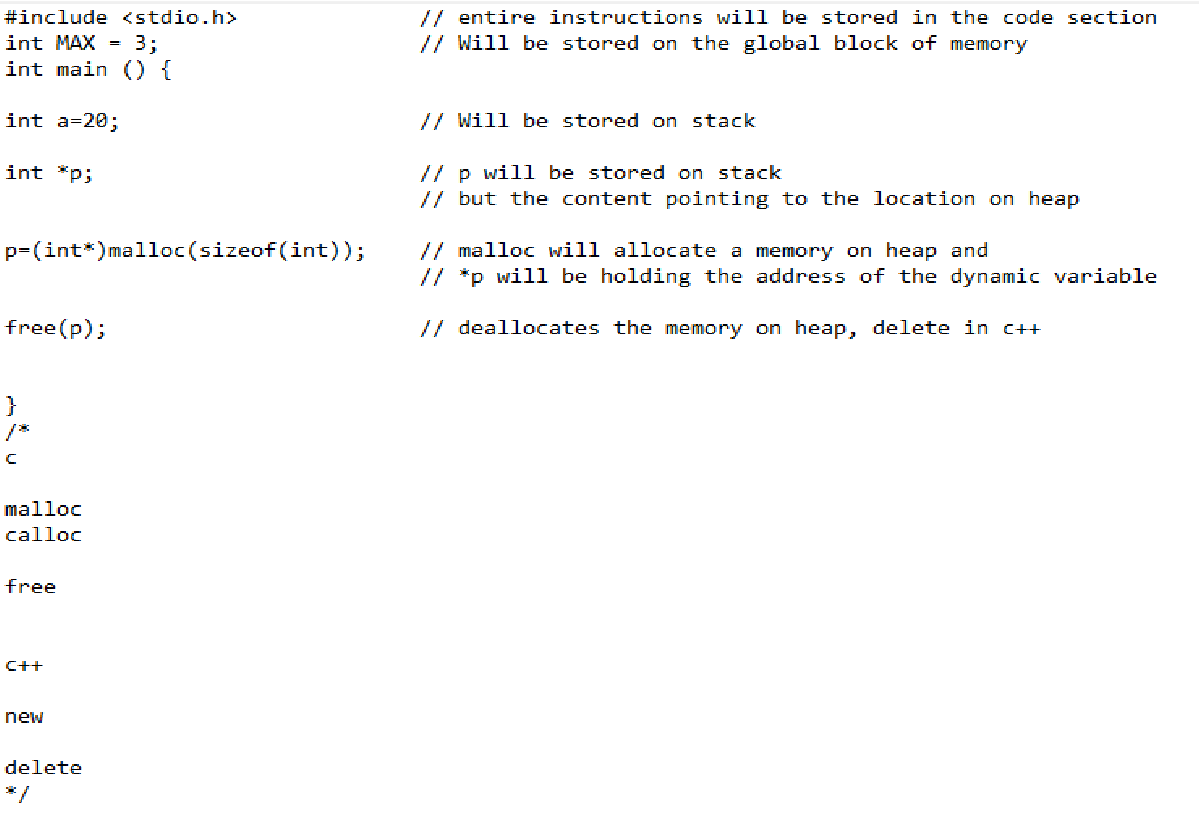
**Stack**

**Static/global**

**Code section**

The memory allocated for heap does not extend after the end address that is why it decrements the counter.

The memory for stack is limited and fixed and program cannot request above its allocated space.



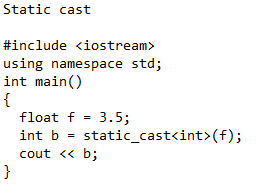
**Type Casting**

**Implicit Conversion**: Done by compiler on its own

**Explicit conversion**: forcefully done



**Static cast**

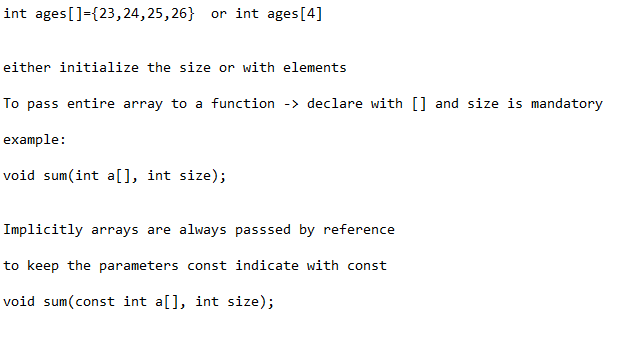


**Dynamic Cast**: will pass during compile time but fail at run time.

**Array**

**Array Size**: int arr\_size = sizeof(a)/sizeof(a[0]);

Initializing methods



**Vectors**

Comparison between array

1. Vector lengths can dynamically grow and shrink.

vector<data type> var\_name (size,default value);

var\_name.push\_back(value) 🡪 Insert an element

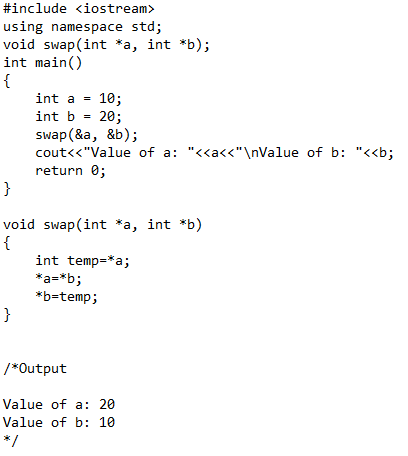
var\_name.size() 🡪 length of vector

var\_name.capacity() 🡪 Number of elements for which memory is currently allocated

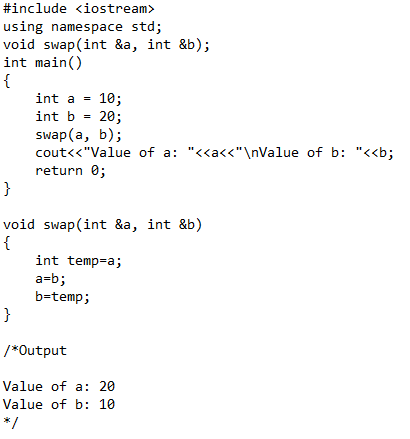
**Pointers**

Pointer is a variable whose value is address of another variable.

**Pointers Call by reference**



**Method – 2**

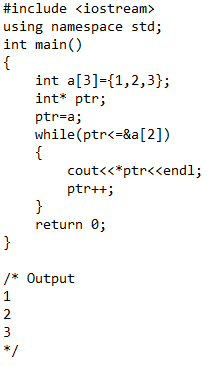


**Pointer Arithmetic**

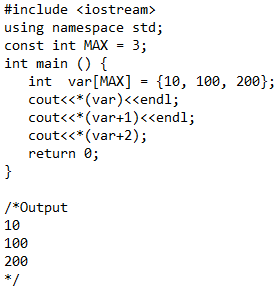
Consider ptr 🡪 points to integer address 1000

ptr++ 🡪 1004

**Pointer with array**



**Arrays**



**Iterators**

**Declaration**:

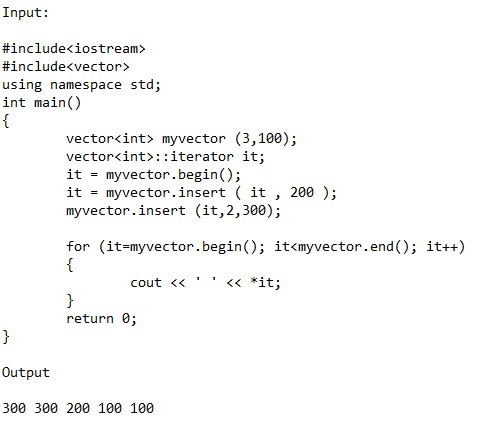
Vector<int>::iterator <iterator\_name>;

Value: \*<iterator\_name>

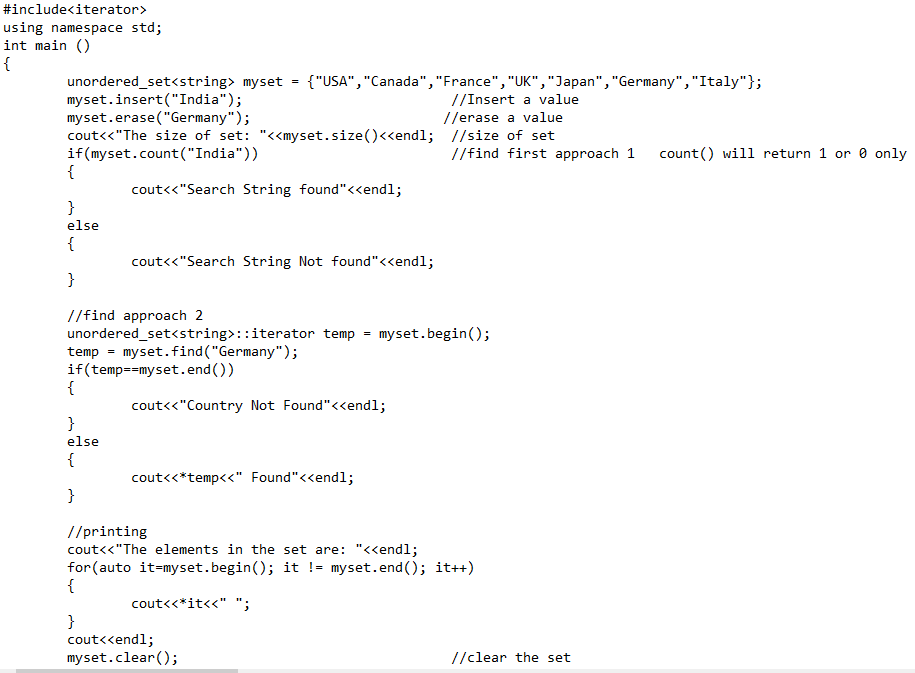
Increments: ++<iterator\_name>

Initialize: <iterator\_name>=vector.begin() or vector.end()

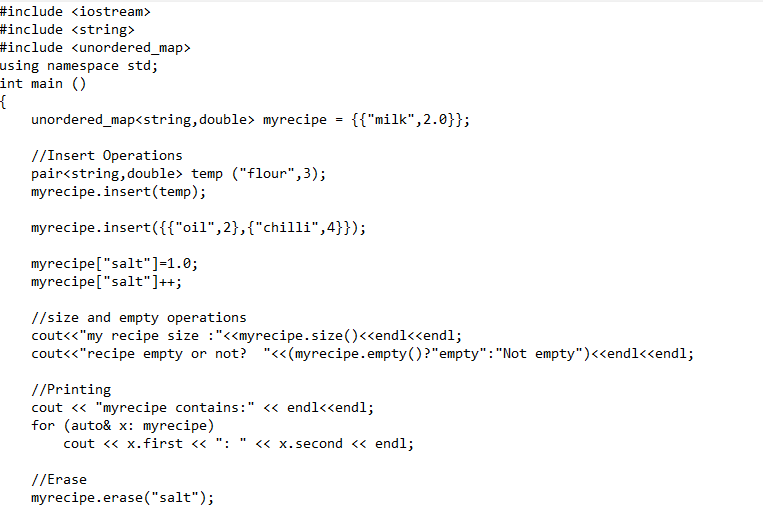
**Inserting elements in middle indexes of a vector same works for string**

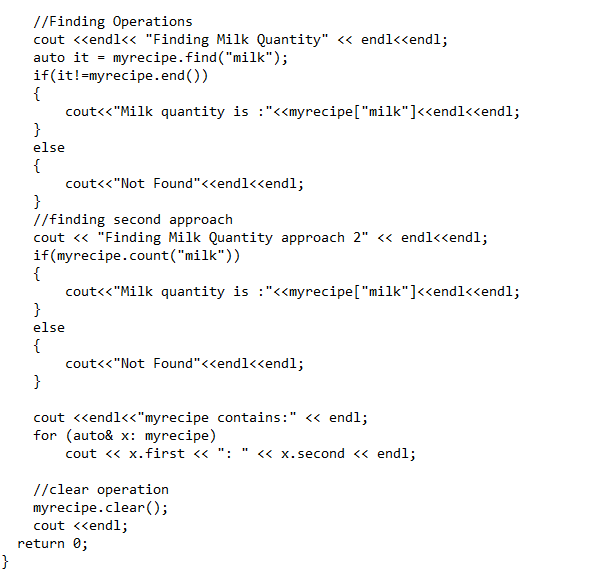


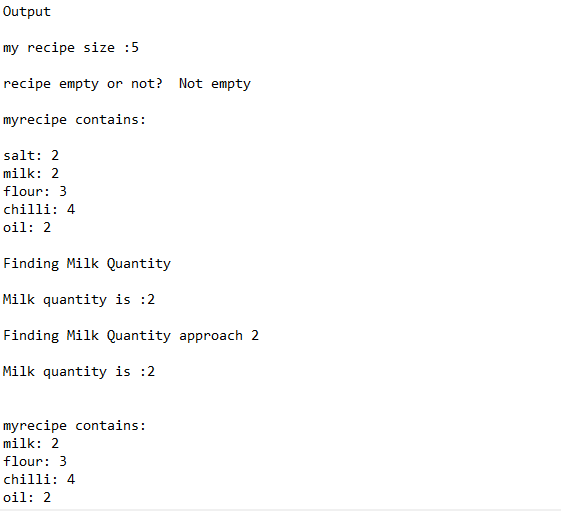
**Unordered Set**



**Unordered Map**







**Auto**

The auto keyword specifies that the type of the variable that is being declared will be automatically deducted from its initializer. In case of functions, if their return type is auto then that will be evaluated by return type expression at runtime.

**Abstract Data Type**

Is a type of class or objects who behavior is defined by set of values and a set of operations.

Examples:

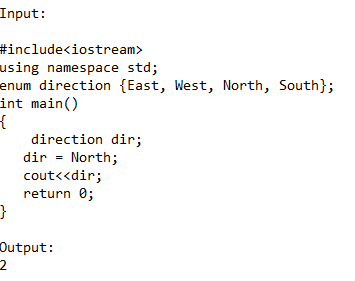
ListADT

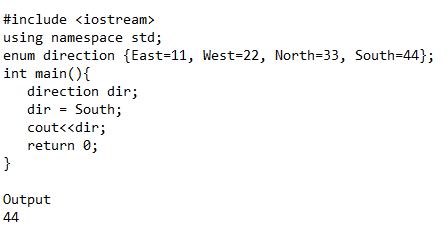
StackADT – The operations push and pop on stack are different that on queue

QueueADT

**Enumerated data type**

**Example:**





**OOPs (Object Oriented Programming)**

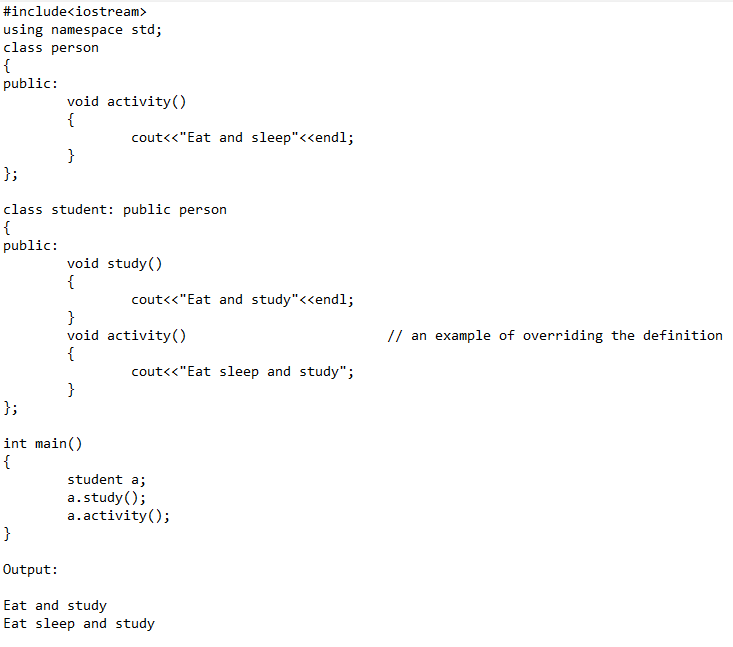
1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

**Encapsulation**: Combining data members and functions in a single unit called class.

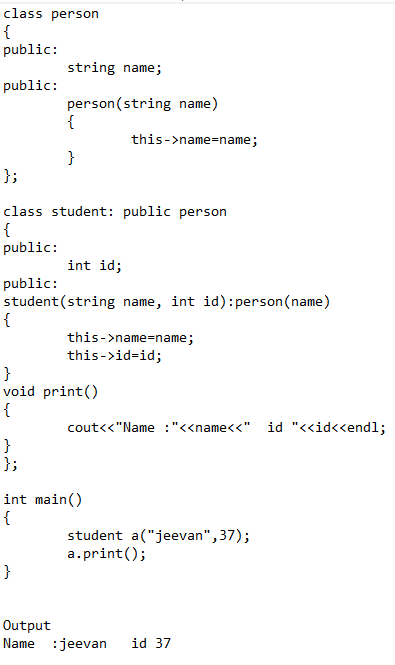
**Inheritance**: is one of the features of Object-Oriented Programming System (OOPs), it allows the child class to acquire the properties (the data members) and functionality (the member functions) of parent class.

Examples

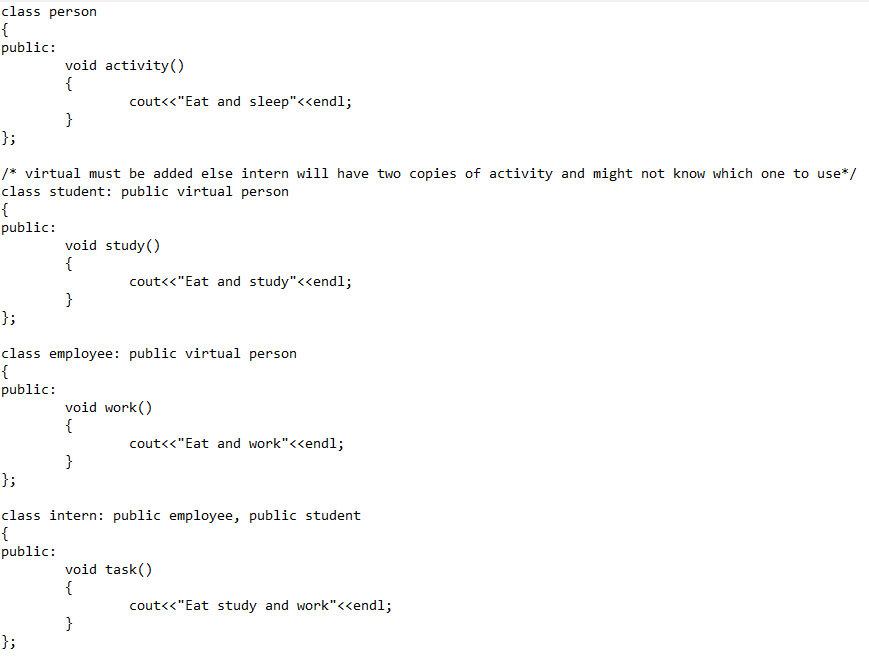
**Simple Inheritance**



**Inheritance setting parent class from child class**



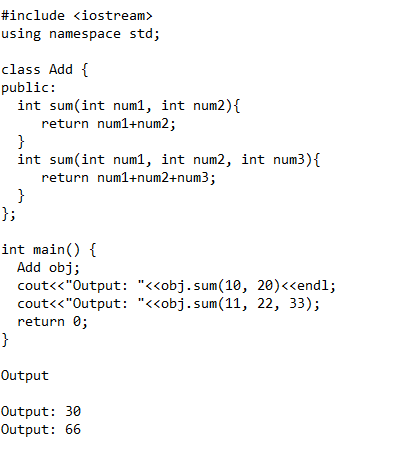
**Diamond Problem**

****

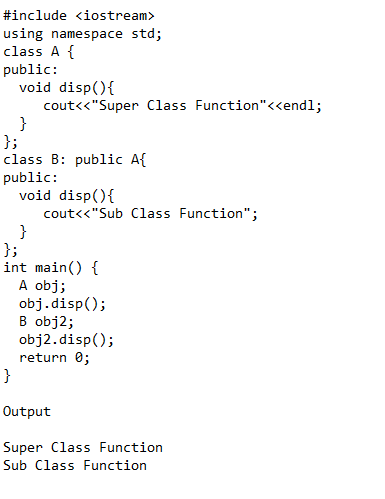


**Polymorphism**: Ability of objects to behave differently in different conditions

**Compile time polymorphism**: Function overloading and Operator overloading.

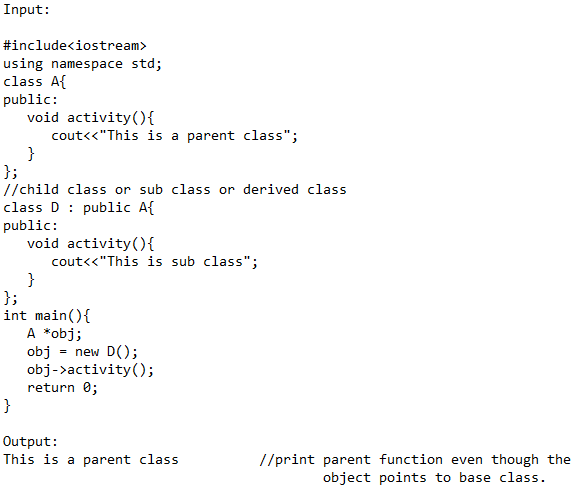


**Runtime Polymorphism**: Function overriding

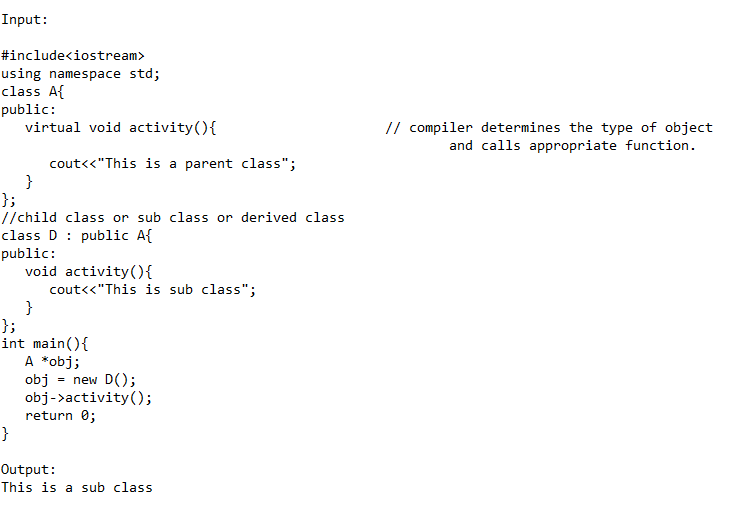


**Runtime Polymorphism: Problem**

Non-Virtual function

****

Virtual function

****

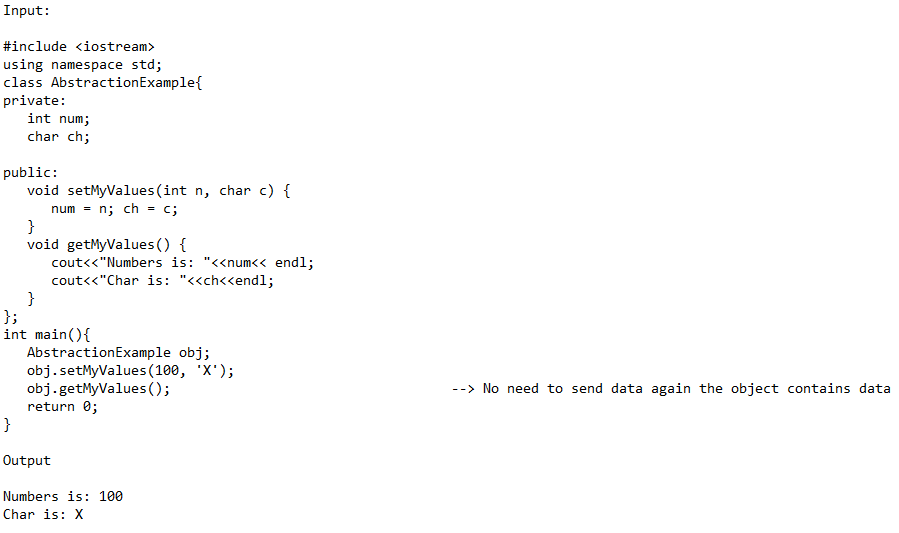
**Abstract Class**

An abstract class is a class defined to be specifically used as base class.

It must contain at least one pure virtual function.

**Abstraction**: is one of the features of Object Oriented Programming, where you show only relevant details to the user and hide irrelevant details.

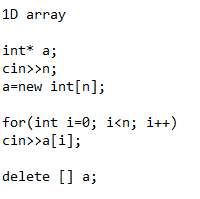
Example:



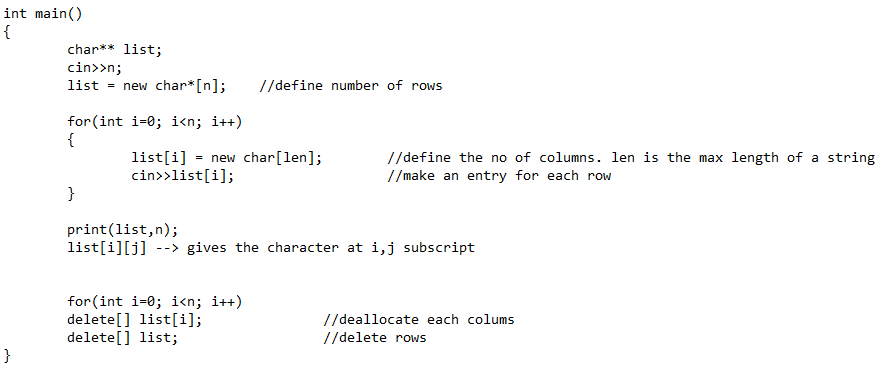
**Dynamic Arrays**

**1 D Array**

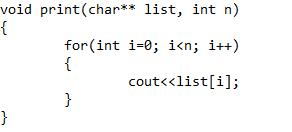
Example:



**2D array**



Printing



**Data Structures**

Is a way of organizing data so that data can be used efficiently.

**Linear Data Structures**: If elements from a sequence or a linear list.

Example: Array, Linked Lists, Stacks and Queues.

**Non-Linear**: If traversal of node is non-linear

Example: Trees and Graphs

**Traversal**: Accessing each element only once in some order.

**Stack**: FIFO

**Applications**:

Infix to Postfix conversion using Stack

Reverse a string -- Push all elements and pop all elements.

Implement 2 stacks in a array – have 2 indexes 1 starting from (0 and incrementing) and (size and decrementing)

Queue is used for Breadth first search.

Stack is used for Depth First Search.

**Trees**

**Inorder**: sorted order in non-decreasing. (This also confirms whether

**Preorder**: Create a copy of the tree. also used to get prefix expression on of an expression tree.

**Postorder**: Delete the tree. Also to get postfix expression on an expression tree.

Tree Traversal can be of 2 types

1. Breadth First
2. Depth First

**Breadth First – (Level Order)**

Will print out values in each level first, and then increment and write out the next level values.

**Depth First**

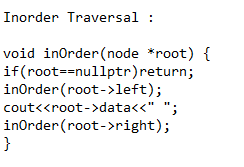
First we move to child only after completing the whole subtree, the next child is accessed.

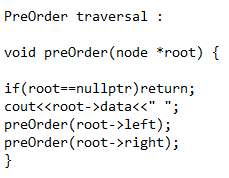
In Depth first there are 3 types of traversal

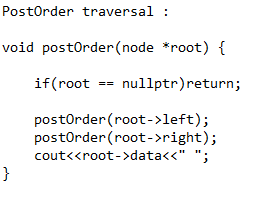
Inorder

Preorder

Postorder







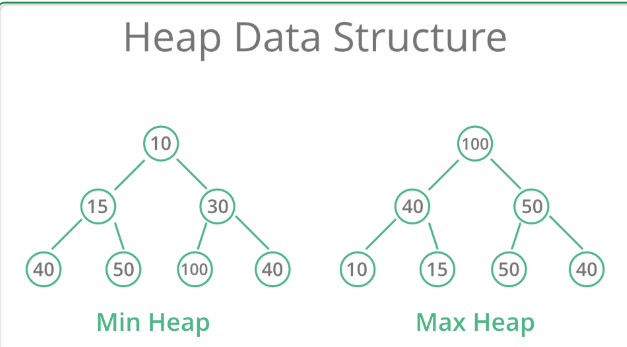
Heap

Is a tree based data structure in which tree is a complete binary tree.

It is of 2 types:

Max-Heap: Key present in root node must be max of all its children. The same property for all subtress.

Min Heap: The root node key must the minimum of the tree.



**Hash Table:**

a) The array maps and index to the data value stored in the array. The mapping function is efficient as long as the index value is known or within range.

b) We can consider the index value to be the "key" to the corresponding data value.

c) A hash table also stores data value but use a key to obtain the corresponding data value.

d) The key need not be an integer value it can be of any data type or a class.

e) The hash code are limited in size and no

f) If the hash table’s mapping function maps a key value into an integer in the range 0 to Table Size– 1, then we can use this integer value as the index into underlying array.

Two approaches for collision resolution

1) Separate chaining

2) Open addressing

a) Linear Probing

b) Quadratic probing

Separate chaining:

a) Each cell in a hash table is a pointer to a linked list of all records that hash to that entry.

b) To retrieve a data we first hash to that cell.

c) Then we search in the associated linked list for the data record.

d) We can sort the link list to improve search performance.

Open addressing:

Linear Probing:

Insert: If the cell is filled look for the next empty cell.

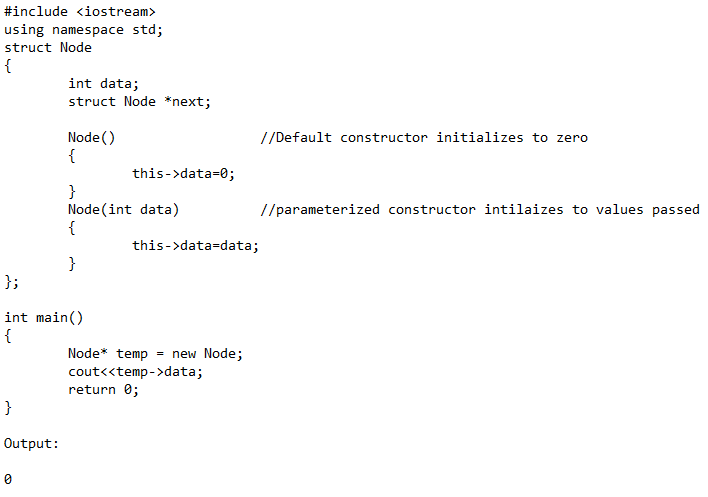
Search: Start searching at the home cell. keep looking at the next cell until the matching key is found. If you encounter an empty cell then there is no match.

Quadratic Probing:

Search at 1, 2^2, 3^2, 4^2 positions.

**Linked List**

**Singly Linked List**

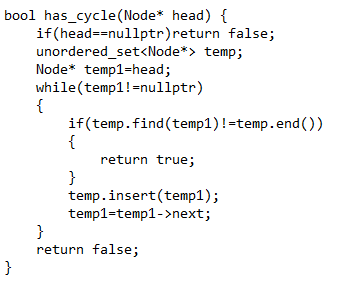


**Reverse a single linked list**

1. 3 Node\* pointers (prev, current, next)
2. Prev and next assigned to nullptr and current to head.
3. While current !=nullptr
4. Assign next to current->next
5. Assign current->next to prev
6. Assign prev to current
7. Assign current to next

**Detect a cycle**

1. Declare an unodered of node\*. (unordered\_set <Node\*> temp)
2. Iterate the linked list until nullptr is reached



**Complexities**

Order of complexities

O(1)

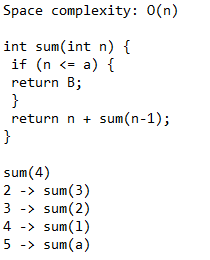
O(log n)

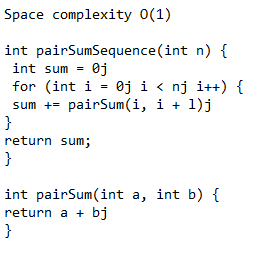
O(n)

O(n log n)

O( )

**Space Complexity**: amount of memory required by the algorithm





**Complexity of Binary Search:** O( log N )

**Complexity of recursive calls**: O( )

Branches = No of calls

Depth = No of counts

Page 68(solve all) – coding book

**Searching**

**Binary Search**

Search a sorted array.

On every search check only with half the elements.

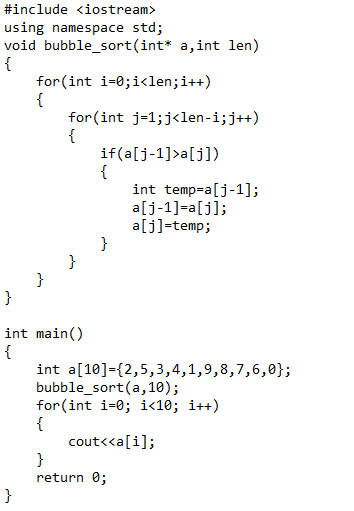
Search based on the element in the middle index.

**Sorting**

**Bubble Sort**

Compare with consecutive elements.

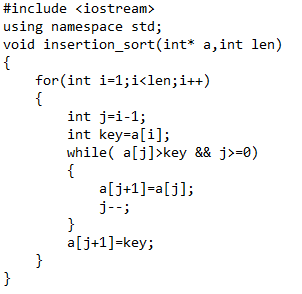
On every round the highest element is found and put at the end.



**Insertion Sort**

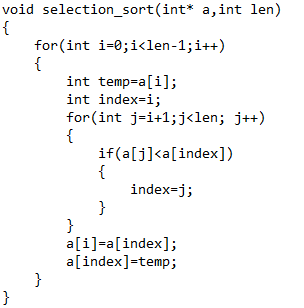
Sorting cards.

Compare with previous index and replace if the index element is small.



**Selection Sort**

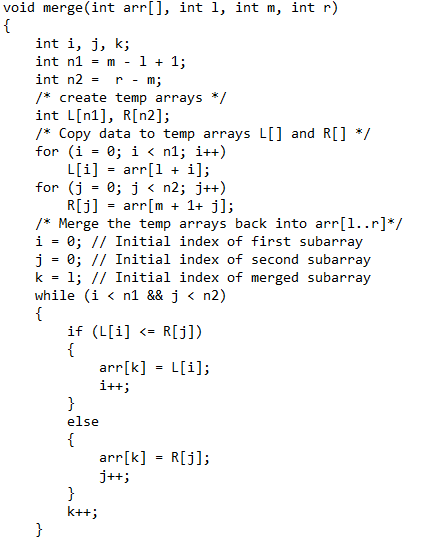
Find the minimum element from the unsorted part and put at the beginning.

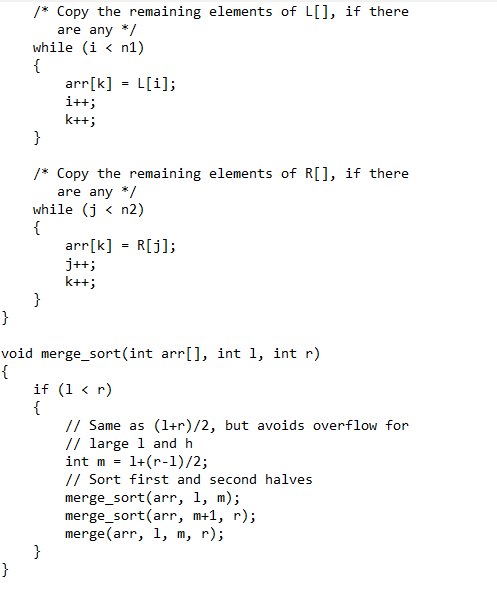


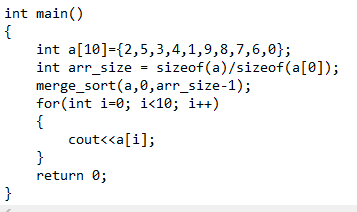
**Merge Sort**

Continuously split the array into two halves.

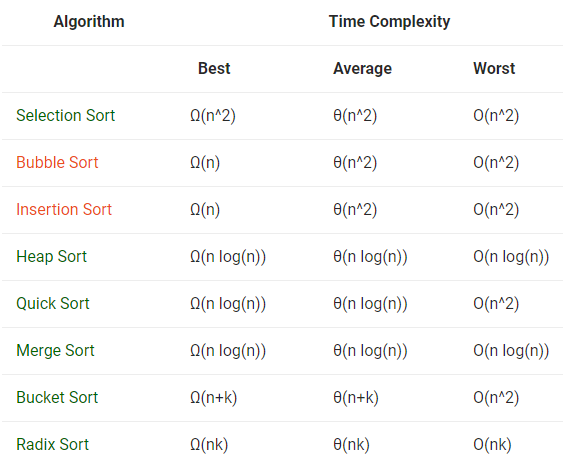
Merge after it is split in to single elements.







**Complexities of Sorting Techniques**



**Important Notes**

Merge Sort and Heap Sort are the best one for worst case data. With O(n log(n))

Merge Sort, Quick Sort and Heap Sort are the best one for average case data. With O(n log(n))

TRIES

**Exceptions**

|  |  |
| --- | --- |
| exception | Generalized exception |
| bad\_alloc | Thrown if error in new |
| bad\_cast | Error thrown by dynamic cast |
| bad\_exception | Handle unexpected exceptions in C++ |
| logic\_error | domain\_error  invalid\_arguement  length\_error |
| runtime\_error | overflow\_error  range\_error  underflow\_error |
|  |  |