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

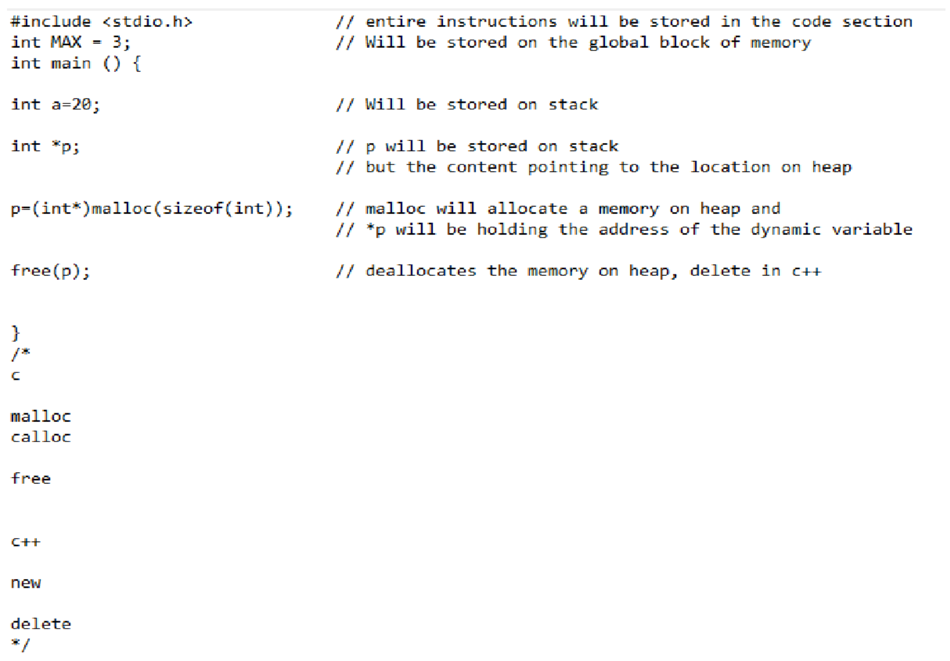
Stack

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.



```
#include <stdio.h>                // entire instructions will be stored in the code section
int MAX = 3;                      // Will be stored on the global block of memory
int main () {

int a=20;                         // Will be stored on stack

int *p;                           // p will be stored on stack
                                  // but the content pointing to the location on heap

p=(int*)malloc(sizeof(int));      // malloc will allocate a memory on heap and
                                  // *p will be holding the address of the dynamic variable

free(p);                          // deallocates the memory on heap, delete in c++


}
/*
c

malloc
calloc

free


c++

new

delete
*/
```

Global variables: they are store in a section above text segment.

2 types initialized ones and un-initialized

All global variables assigned the values is stored in global initialized block of memory.

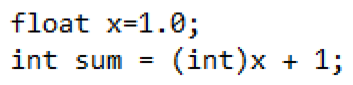
All global variables un initialized are stored in global un initialized space of memory.

Link: https://www.youtube.com/watch?v=0jhQBQcGnuM

Type Casting

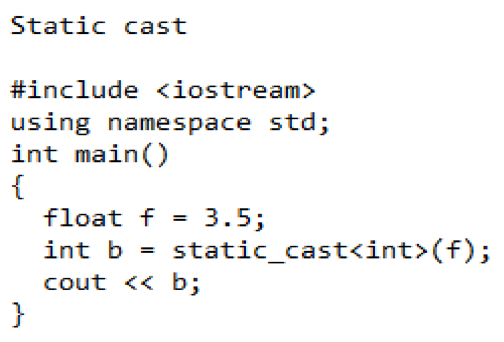
Implicit Conversion: Done by compiler on its own

Explicit conversion: forcefully done



```
float x=1.0;
int sum = (int)x + 1;
```

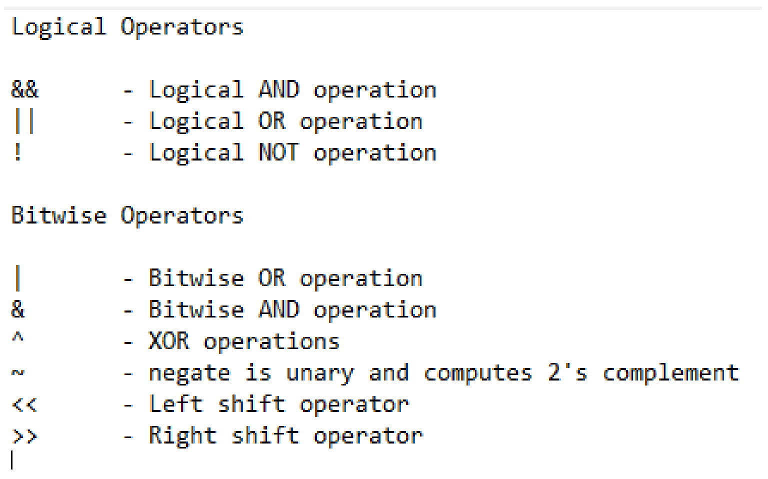
Static cast



```
Static cast

#include <iostream>
using namespace std;
int main()
{
  float f = 3.5;
  int b = static_cast<int>(f);
  cout << b;
}
```

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

Bitwise Operators in C++



```
Logical Operators

&&      - Logical AND operation
||      - Logical OR operation
!       - Logical NOT operation

Bitwise Operators

|       - Bitwise OR operation
&       - Bitwise AND operation
^       - XOR operations
~       - negate is unary and computes 2's complement
<<      - Left shift operator
>>      - Right shift operator
```

Array

Array Size: int arr_size = sizeof(a)/sizeof(a[0]);

Initializing methods

```
int ages[]={23,24,25,26}  or int ages[4]

either initialize the size or with elements

To pass entire array to a function -> declare with [] and size is mandatory

example:

void sum(int a[], int size);

Implicitly arrays are always passsed by reference

to keep the parameters const indicate with const

void sum(const int a[], int size);
```

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.

Void* means the pointer can store address of any type.

Three types of using pointers

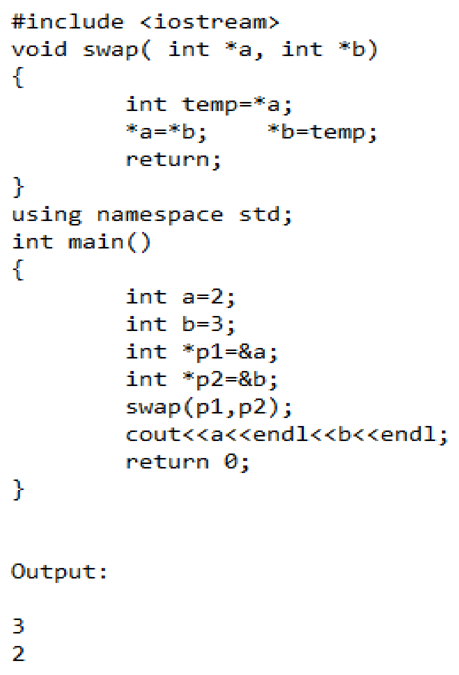
Type 1

```
Type 1

#include <iostream>
void swap( int *a, int *b)
{
        int temp=*a;
        *a=*b;
        *b=temp;
        return;
}
using namespace std;
int main()
{
        int a=2;
        int b=3;
        swap(&a,&b);
        cout<<a<<endl<<b<<endl;
        return 0;
}

Output:

3
2
```

Type 2

```
#include <iostream>
void swap( int &a, int &b)
{
        int temp=a;
        a=b;
        b=temp;
        return;
}
using namespace std;
int main()
{
        int a=2;
        int b=3;
        swap(a,b);
        cout<<a<<endl<<b<<endl;
        return 0;
}

Output:

3
2
```

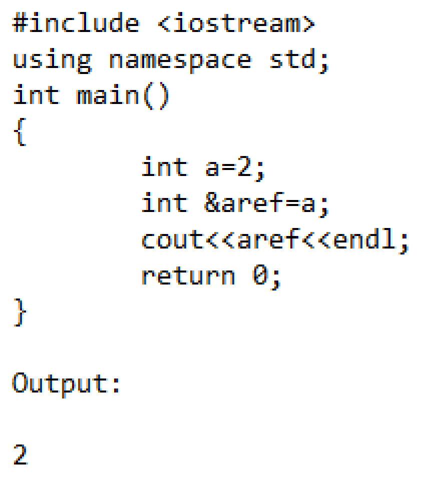
Type 3



```
#include <iostream>
void swap( int *a, int *b)
{
        int temp=*a;
        *a=*b;    *b=temp;
        return;
}
using namespace std;
int main()
{
        int a=2;
        int b=3;
        int *p1=&a;
        int *p2=&b;
        swap(p1,p2);
        cout<<a<<endl<<b<<endl;
        return 0;
}
```

```
Output:

3
2
```

Reference example



```
#include <iostream>
using namespace std;
int main()
{
        int a=2;
        int &aref=a;
        cout<<aref<<endl;
        return 0;
}
```

```
Output:

2
```

Pointer Arithmetic

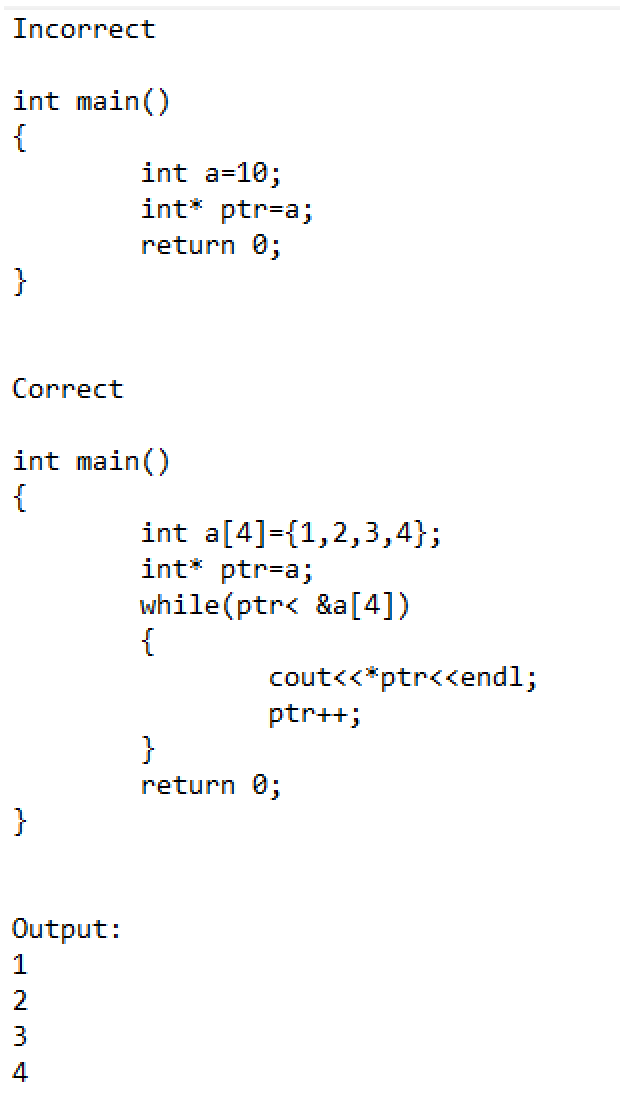
Consider ptr → points to integer address 1000 ptr++ → 1004

Pointer with array

```
#include <iostream>
using namespace std;
int main()
{
    int a[3]={1,2,3};
    int* ptr;
    ptr=a;
    while(ptr<=&a[2])
    {
        cout<<*ptr<<endl;
        ptr++;
    }
    return 0;
}

/* Output
1
2
3
*/
```

Arrays

```
#include <iostream>
using namespace std;
const int MAX = 3;
int main () {
   int  var[MAX] = {10, 100, 200};
   cout<<*(var)<<endl;
   cout<<*(var+1)<<endl;
   cout<<*(var+2);
   return 0;
}

/*Output
10
100
200
*/
```



```
Incorrect

int main()
{
        int a=10;
        int* ptr=a;
        return 0;
}


Correct

int main()
{
        int a[4]={1,2,3,4};
        int* ptr=a;
        while(ptr< &a[4])
        {
                cout<<*ptr<<endl;
                ptr++;
        }
        return 0;
}


Output:
1
2
3
4
```

Containers

Containers are holder objects that store collection of other objects.

main functions

1. Container manage storage spaces.
2. And provides member function to access the objects.

Iterators

Declaration:

Vector<int>::iterator <iterator_name>;

Value: *<iterator_name>

Increments: ++<iterator_name>

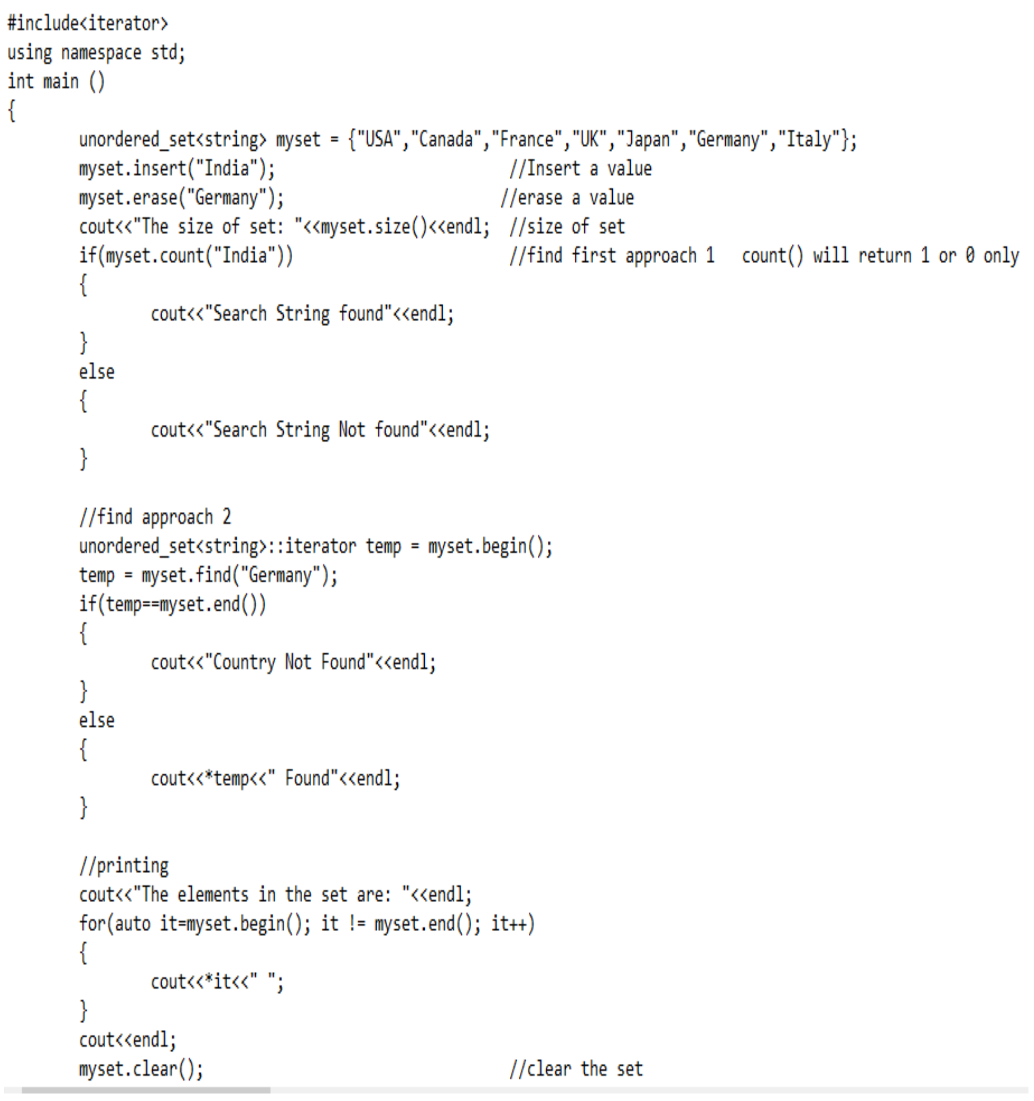
Initialize: <iterator_name>=vector.begin() or vector.end()

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

```
Input:

#include<iostream>
#include<vector>
using namespace std;
int main()
{
        vector<int> myvector (3,100);
        vector<int>::iterator it;
        it = myvector.begin();
        it = myvector.insert ( it , 200 );
        myvector.insert (it,2,300);

        for (it=myvector.begin(); it<myvector.end(); it++)
        {
                cout << ' ' << *it;
        }
        return 0;
}
```

```
Output

300 300 200 100 100
```

Unordered Set



```
#include<iterator>
using namespace std;
int main ()
{
        unordered_set<string> myset = {"USA","Canada","France","UK","Japan","Germany","Italy"};
        myset.insert("India");                          //Insert a value
        myset.erase("Germany");                        //erase a value
        cout<<"The size of set: "<<myset.size()<<endl;  //size of set
        if(myset.count("India"))                        //find first approach 1   count() will return 1 or 0 only
        {
                cout<<"Search String found"<<endl;
        }
        else
        {
                cout<<"Search String Not found"<<endl;
        }

        //find approach 2
        unordered_set<string>::iterator temp = myset.begin();
        temp = myset.find("Germany");
        if(temp==myset.end())
        {
                cout<<"Country Not Found"<<endl;
        }
        else
        {
                cout<<*temp<<" Found"<<endl;
        }

        //printing
        cout<<"The elements in the set are: "<<endl;
        for(auto it=myset.begin(); it != myset.end(); it++)
        {
                cout<<*it<<" ";
        }
        cout<<endl;
        myset.clear();                                  //clear the set
```

Unordered Map

Difference between a map and unordered map

Element is a combination of Key and Value

Map: Elements in a map are sorted by its key.

Unordered Map: Elements are not ordered by key or value. But they are organized in to buckets that depending on their hash values.

```
                  | map             | unordered_map
---------------------------------------------------------
Ordering        | increasing  order   | no ordering
                | (by default)        |

Implementation  | Self balancing BST  | Hash Table
                | like Red-Black Tree |

search time     | log(n)              | O(1) -> Average
                |                     | O(n) -> Worst Case

Insertion time  | log(n) + Rebalance  | Same as search

Deletion time   | log(n) + Rebalance  | Same as search
```

```cpp
#include <iostream>
#include <string>
#include <unordered_map>
using namespace std;
int main ()
{
    unordered_map<string,double> myrecipe = {{"milk",2.0}};

    //Insert Operations
    pair<string,double> temp ("flour",3);
    myrecipe.insert(temp);

    myrecipe.insert({{"oil",2},{"chilli",4}});

    myrecipe["salt"]=1.0;
    myrecipe["salt"]++;

    //size and empty operations
    cout<<"my recipe size :"<<myrecipe.size()<<endl<<endl;
    cout<<"recipe empty or not?  "<<(myrecipe.empty()?"empty":"Not empty")<<endl<<endl;

    //Printing
    cout << "myrecipe contains:" << endl<<endl;
    for (auto& x: myrecipe)
        cout << x.first << ": " << x.second << endl;

    //Erase
    myrecipe.erase("salt");
    //Finding Operations
    cout <<endl<< "Finding Milk Quantity" << endl<<endl;
    auto it = myrecipe.find("milk");
    if(it!=myrecipe.end())
    {
        cout<<"Milk quantity is :"<<myrecipe["milk"]<<endl<<endl;
    }
    else
    {
        cout<<"Not Found"<<endl<<endl;
    }
    //finding second approach
    cout << "Finding Milk Quantity approach 2" << endl<<endl;
    if(myrecipe.count("milk"))
    {
        cout<<"Milk quantity is :"<<myrecipe["milk"]<<endl<<endl;
    }
    else
    {
        cout<<"Not Found"<<endl<<endl;
    }

    cout <<endl<<"myrecipe contains:" << endl;
    for (auto& x: myrecipe)
        cout << x.first << ": " << x.second << endl;

    //clear operation
    myrecipe.clear();
    cout <<endl;
  return 0;
}
```

```
Output

my recipe size :5

recipe empty or not?  Not empty

myrecipe contains:

salt: 2
milk: 2
flour: 3
chilli: 4
oil: 2

Finding Milk Quantity

Milk quantity is :2

Finding Milk Quantity approach 2

Milk quantity is :2


myrecipe contains:
milk: 2
flour: 3
chilli: 4
oil: 2
```

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.

Memset

Used to fill a block a memory with a particular value.

Syntax:

void memset(*ptr, int x, size_t n);

when it comes to integer it can set only 0 or -1

no other values permitted. Since it can do only byte by byte.

sample:

```
#include <cstring>
#include <iostream>
using namespace std;
int main()
{
    char str[] = "jeevan***babu";
    memset(str+6, ' ', 3*sizeof(char));
    cout <<str<<endl;

    int a[3]={1,2,3};
    memset(a,-1,sizeof(a));
    cout<<"{ ";
    for(int i=0; i<sizeof(a)/sizeof(a[0]); i++)
    {
        cout<<a[i]<<" ";
    }
    cout<<"}";
    return 0;
}
```

```
Output:

jeevan   babu
{ -1 -1 -1 }
```

Abstract Data Type

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

Examples:

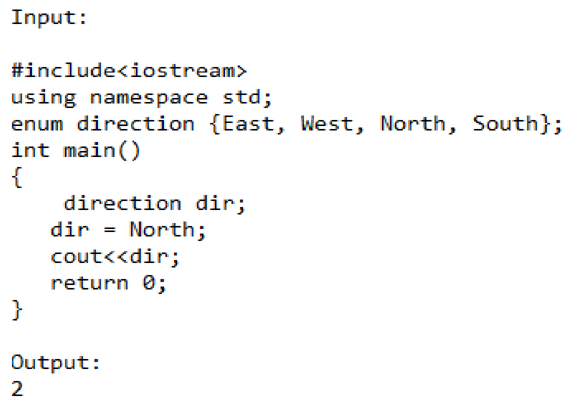
ListADT

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

QueueADT

Enumerated data type

Example:



```
Input:

#include<iostream>
using namespace std;
enum direction {East, West, North, South};
int main()
{
    direction dir;
   dir = North;
   cout<<dir;
   return 0;
}

Output:
2
```

```
#include <iostream>
using namespace std;
enum direction {East=11, West=22, North=33, South=44};
int main(){
   direction dir;
   dir = South;
   cout<<dir;
   return 0;
}

Output
44
```

OOPs (Object Oriented Programming)

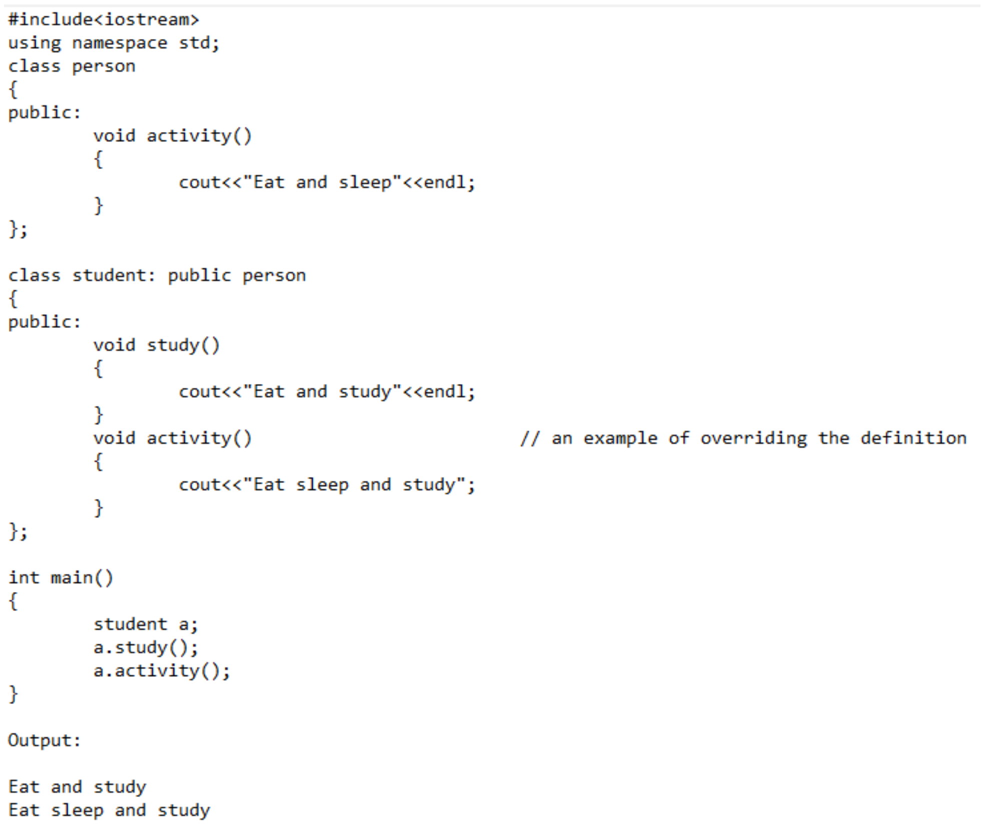
1. Encapsulation
2. Inheritance
3. Polymorphism
4. Abstraction

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.

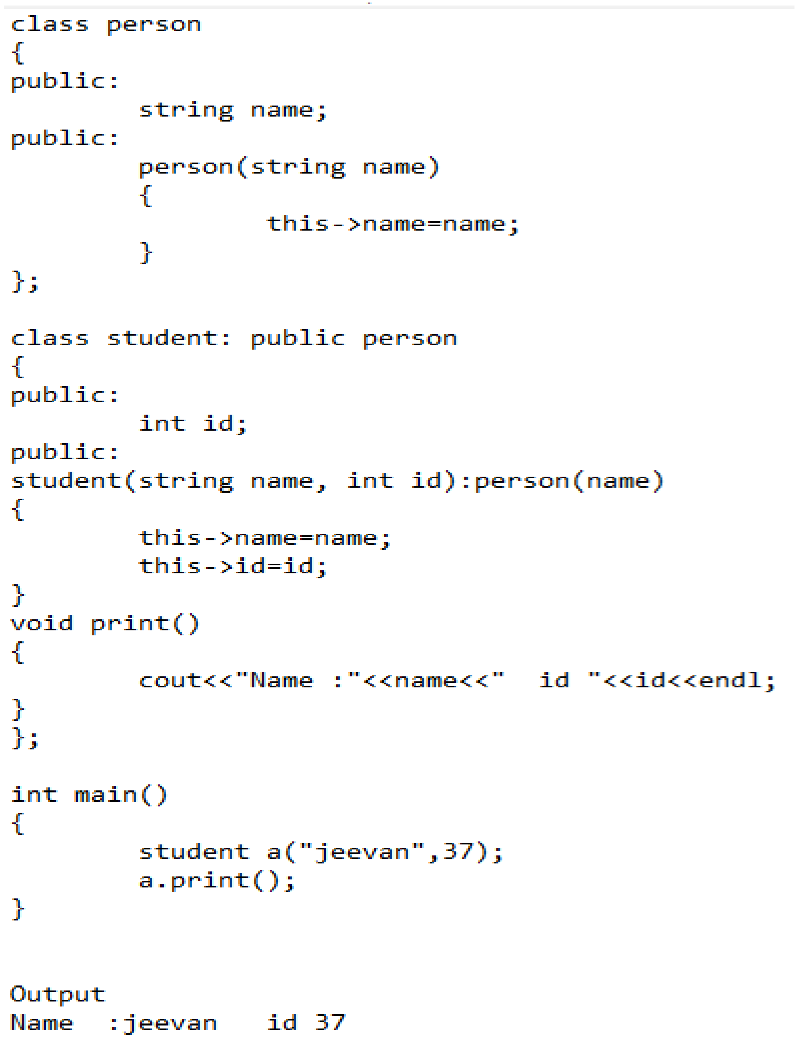
Examples

Simple Inheritance



```
#include<iostream>
using namespace std;
class person
{
public:
        void activity()
        {
                cout<<"Eat and sleep"<<endl;
        }
};

class student: public person
{
public:
        void study()
        {
                cout<<"Eat and study"<<endl;
        }
        void activity()                         // an example of overriding the definition
        {
                cout<<"Eat sleep and study";
        }
};

int main()
{
        student a;
        a.study();
        a.activity();
}

Output:

Eat and study
Eat sleep and study
```

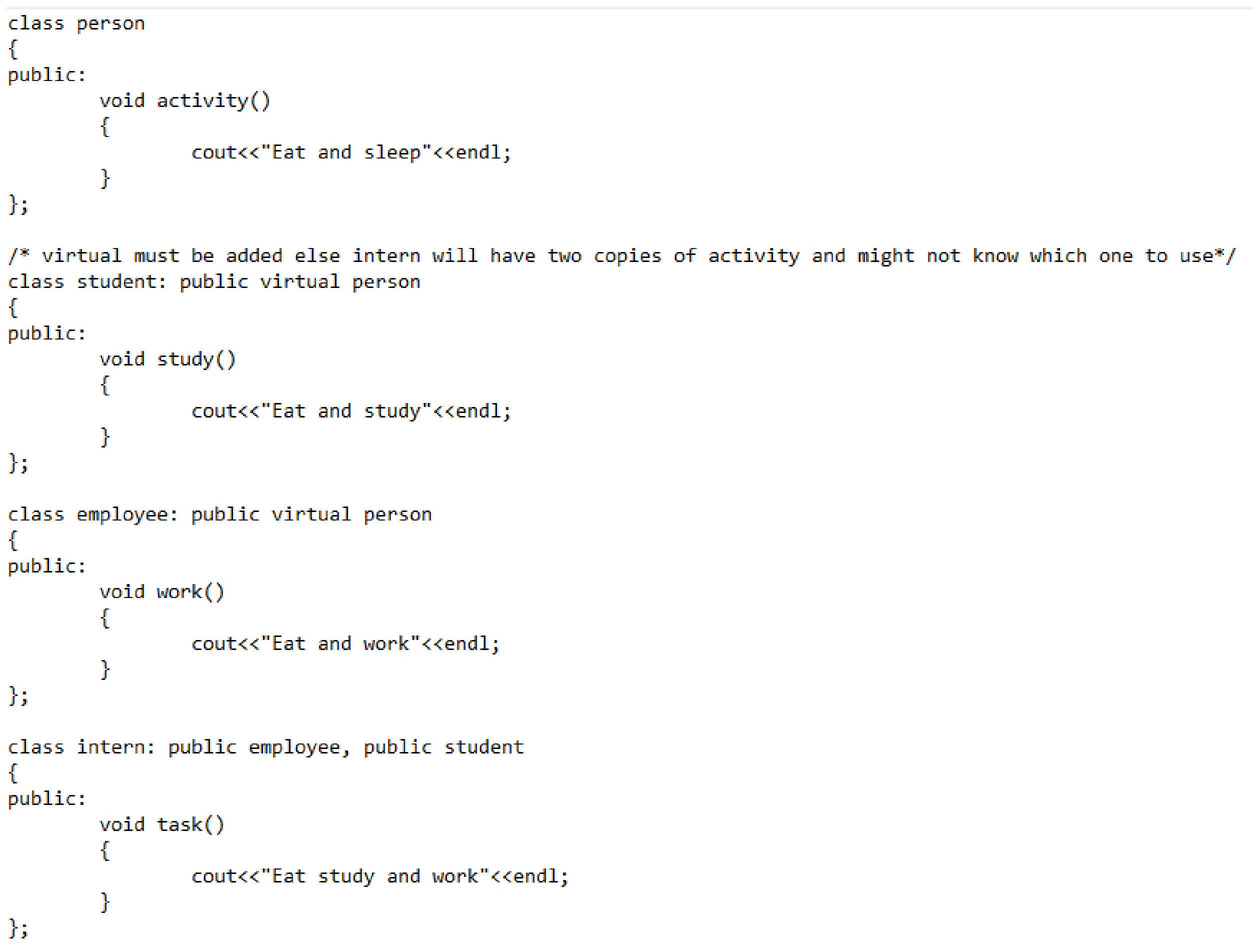
Inheritance setting parent class from child class



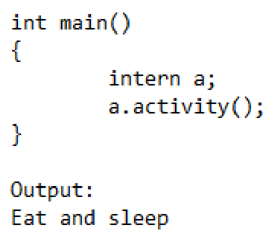
```
class person
{
public:
        string name;
public:
        person(string name)
        {
                this->name=name;
        }
};

class student: public person
{
public:
        int id;
public:
student(string name, int id):person(name)
{
        this->name=name;
        this->id=id;
}
void print()
{
        cout<<"Name :"<<name<<"  id "<<id<<endl;
}
};

int main()
{
        student a("jeevan",37);
        a.print();
}
```

```
Output
Name  :jeevan   id 37
```

Diamond Problem



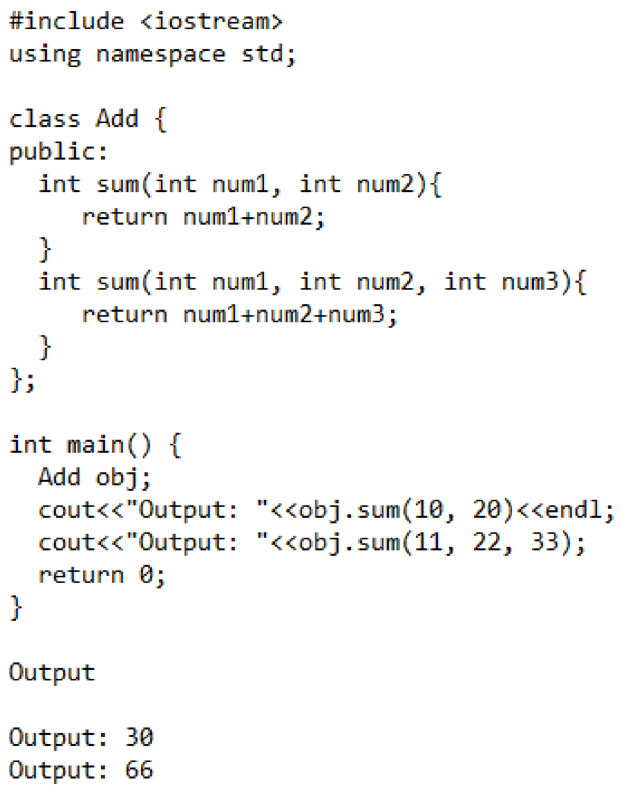
```
class person
{
public:
        void activity()
        {
                cout<<"Eat and sleep"<<endl;
        }
};

/* virtual must be added else intern will have two copies of activity and might not know which one to use*/
class student: public virtual person
{
public:
        void study()
        {
                cout<<"Eat and study"<<endl;
        }
};

class employee: public virtual person
{
public:
        void work()
        {
                cout<<"Eat and work"<<endl;
        }
};

class intern: public employee, public student
{
public:
        void task()
        {
                cout<<"Eat study and work"<<endl;
        }
};
```



```
int main()
{
        intern a;
        a.activity();
}

Output:
Eat and sleep
```

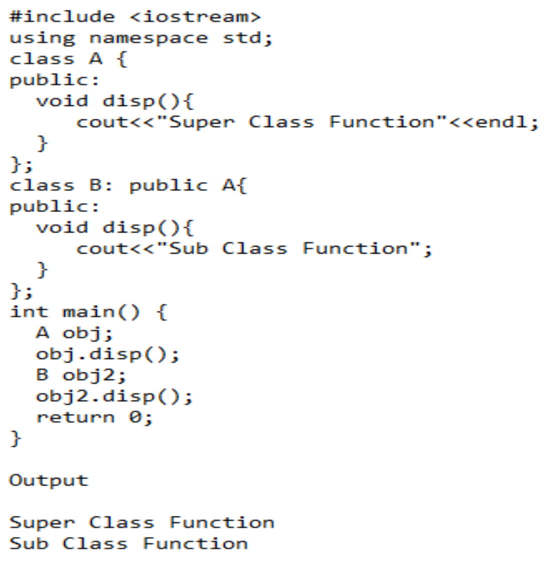
Polymorphism: Ability of objects to behave differently in different conditions

Compile time polymorphism: Function overloading and Operator overloading.



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

class Add {
public:
  int sum(int num1, int num2){
     return num1+num2;
  }
  int sum(int num1, int num2, int num3){
     return num1+num2+num3;
  }
};

int main() {
  Add obj;
  cout<<"Output: "<<obj.sum(10, 20)<<endl;
  cout<<"Output: "<<obj.sum(11, 22, 33);
  return 0;
}

Output

Output: 30
Output: 66
```

Runtime Polymorphism: Function overriding



```
#include <iostream>
using namespace std;
class A {
public:
  void disp(){
     cout<<"Super Class Function"<<endl;
  }
};
class B: public A{
public:
  void disp(){
     cout<<"Sub Class Function";
  }
};
int main() {
  A obj;
  obj.disp();
  B obj2;
  obj2.disp();
  return 0;
}

Output

Super Class Function
Sub Class Function
```

Runtime Polymorphism: Problem

Non-Virtual function

```
Input:

#include<iostream>
using namespace std;
class A{
public:
   void activity(){
      cout<<"This is a parent class";
   }
};
//child class or sub class or derived class
class D : public A{
public:
   void activity(){
      cout<<"This is sub class";
   }
};
int main(){
   A *obj;
   obj = new D();
   obj->activity();
   return 0;
}

Output:
This is a parent class          //print parent function even though the
                                        object points to base class.
```

Virtual function

```
Input:

#include<iostream>
using namespace std;
class A{
public:
   virtual void activity(){                     // compiler determines the type of object
                                                        and calls appropriate function.
      cout<<"This is a parent class";
   }
};
//child class or sub class or derived class
class D : public A{
public:
   void activity(){
      cout<<"This is sub class";
   }
};
int main(){
   A *obj;
   obj = new D();
   obj->activity();
   return 0;
}

Output:
This is a sub class
```

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.

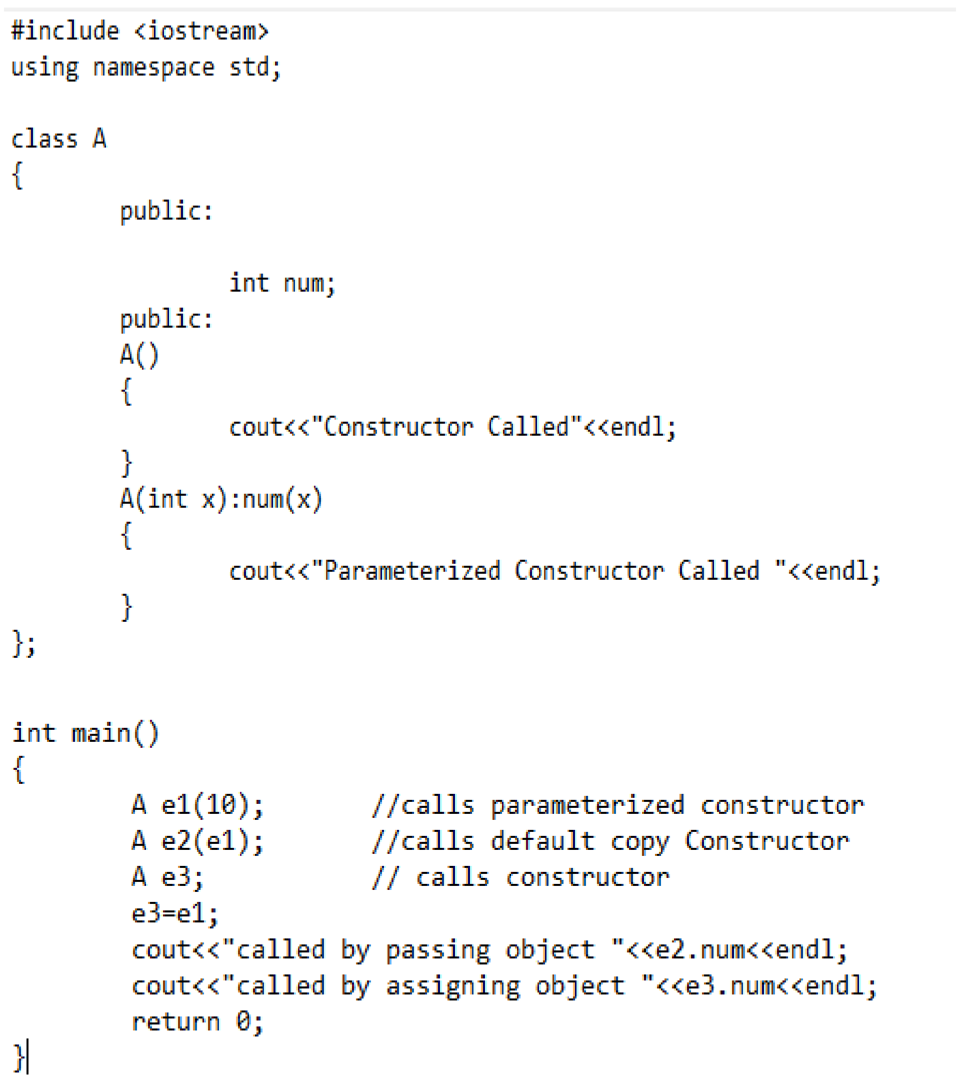
Example:

```
Input:

#include <iostream>
using namespace std;
class AbstractionExample{
private:
   int num;
   char ch;

public:
   void setMyValues(int n, char c) {
      num = n; ch = c;
   }
   void getMyValues() {
      cout<<"Numbers is: "<<num<< endl;
      cout<<"Char is: "<<ch<<endl;
   }
};
int main(){
   AbstractionExample obj;
   obj.setMyValues(100, 'X');
   obj.getMyValues();                                  --> No need to send data again the object contains data
   return 0;
}

Output

Numbers is: 100
Char is: X
```

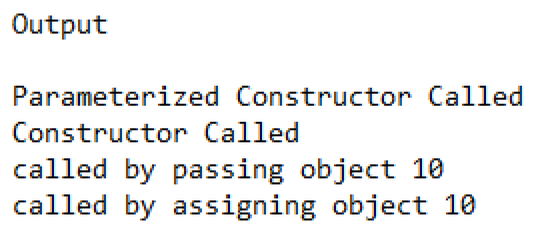
Static inside a class

Copy Constructor

By default compiler provides a copy constructor. And also assignment operator



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

class A
{
        public:

                int num;
        public:
        A()
        {
                cout<<"Constructor Called"<<endl;
        }
        A(int x):num(x)
        {
                cout<<"Parameterized Constructor Called "<<endl;
        }
};

int main()
{
         A e1(10);       //calls parameterized constructor
         A e2(e1);       //calls default copy Constructor
         A e3;           // calls constructor
         e3=e1;
         cout<<"called by passing object "<<e2.num<<endl;
         cout<<"called by assigning object "<<e3.num<<endl;
         return 0;
}
```



```
Output

Parameterized Constructor Called
Constructor Called
called by passing object 10
called by assigning object 10
```

Defining an own copy constructor

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

class A
{
        public:

                int num;
        public:
        A()
        {
                cout<<"Constructor Called"<<endl;
        }
        A(int x):num(x)
        {
                cout<<"Parameterized Constructor Called "<<endl;
        }

        A(const A& b):num(b.num)           //copy constructor
        {
                cout<<"Copy constructor called"<<endl;
        }
};

int main()
{
        A e1(10);       //calls parameterized constructor
        A e2(e1);       //calls default copy Constructor
        A e3;           // calls constructor
        e3=e1;
        cout<<"called by passing object "<<e2.num<<endl;
        cout<<"called by assigning object "<<e3.num<<endl;
        return 0;
}
```

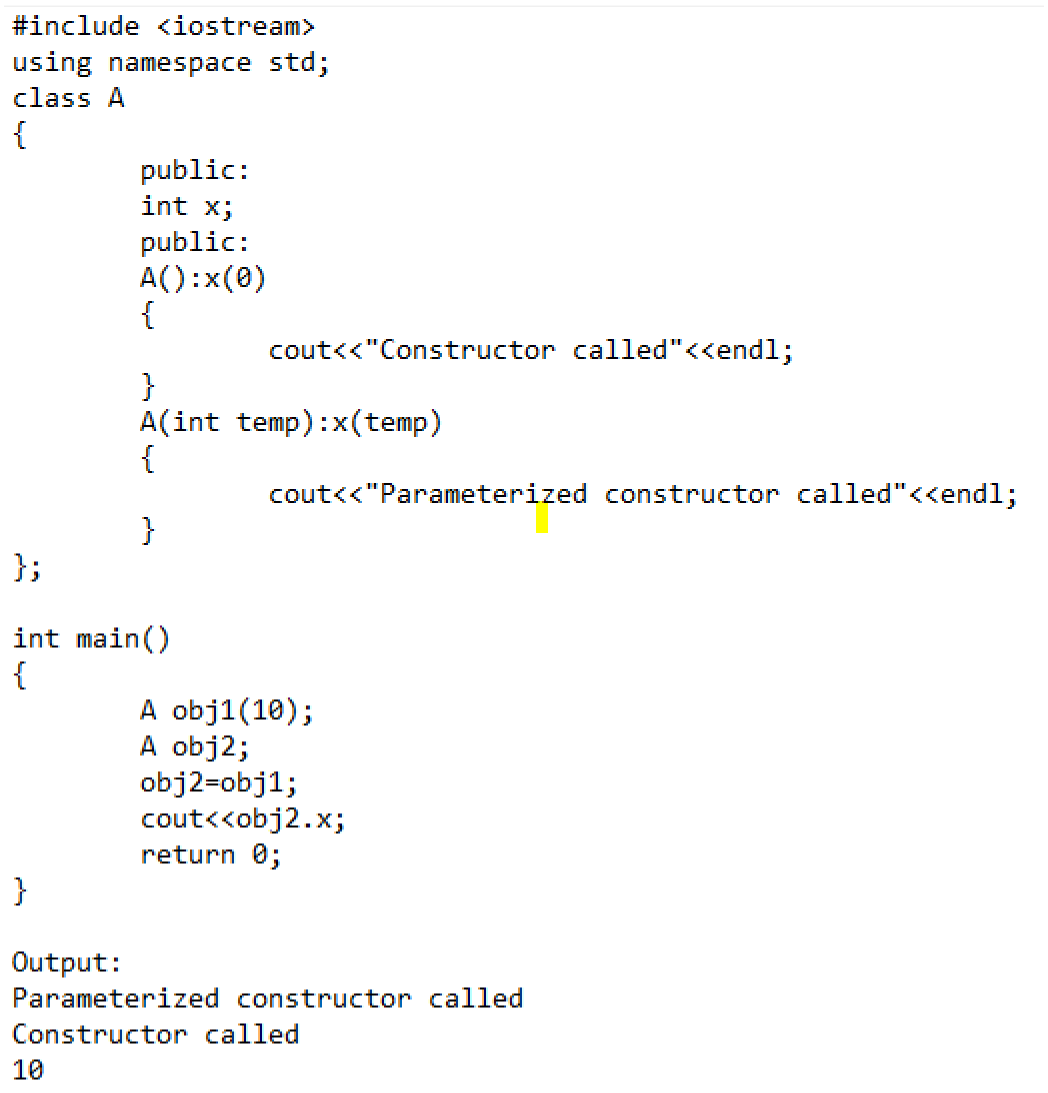
Output

```
Parameterized Constructor Called
Copy constructor called
Constructor Called

called by passing object 10
called by assigning object 10
```

https://www.geeksforgeeks.org/copy-constructor-in-cpp/

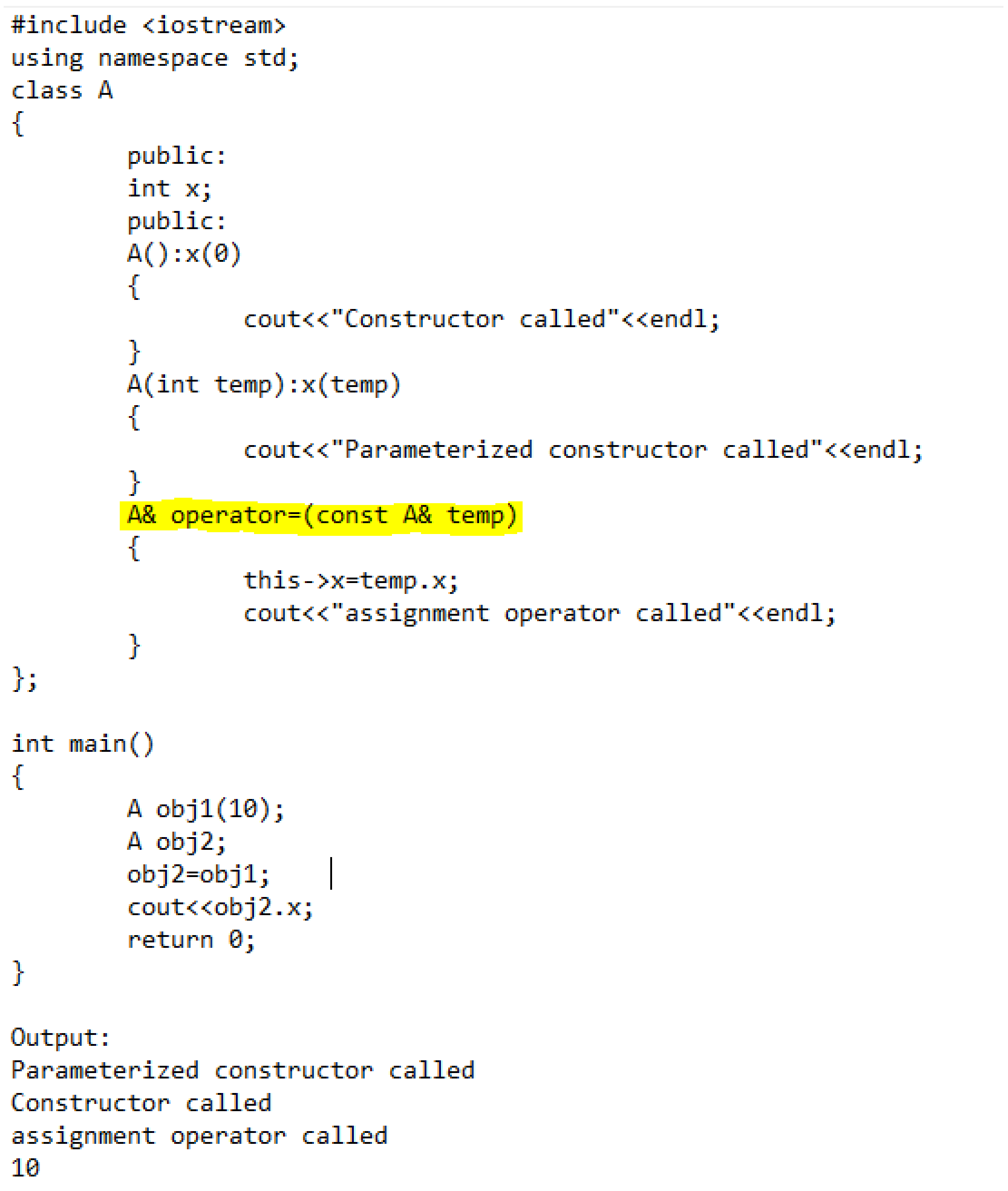
Assignment Operator

Compiler provided



```
#include <iostream>
using namespace std;
class A
{
        public:
        int x;
        public:
        A():x(0)
        {
                cout<<"Constructor called"<<endl;
        }
        A(int temp):x(temp)
        {
                cout<<"Parameterized constructor called"<<endl;
        }
};

int main()
{
        A obj1(10);
        A obj2;
        obj2=obj1;
        cout<<obj2.x;
        return 0;
}

Output:
Parameterized constructor called
Constructor called
10
```

Explicitly defined



```
#include <iostream>
using namespace std;
class A
{
        public:
        int x;
        public:
        A():x(0)
        {
                cout<<"Constructor called"<<endl;
        }
        A(int temp):x(temp)
        {
                cout<<"Parameterized constructor called"<<endl;
        }
        A& operator=(const A& temp)
        {
                this->x=temp.x;
                cout<<"assignment operator called"<<endl;
        }
};

int main()
{
        A obj1(10);
        A obj2;
        obj2=obj1;
        cout<<obj2.x;
        return 0;
}

Output:
Parameterized constructor called
Constructor called
assignment operator called
10
```

Why we need virtual functions in C++?

When a parent class pointer and is assigned a new instance of child class.

Call to the methods of the will result in calling methods in parent class only.

To eliminate this and to ensure correct function is called we need virtual function.

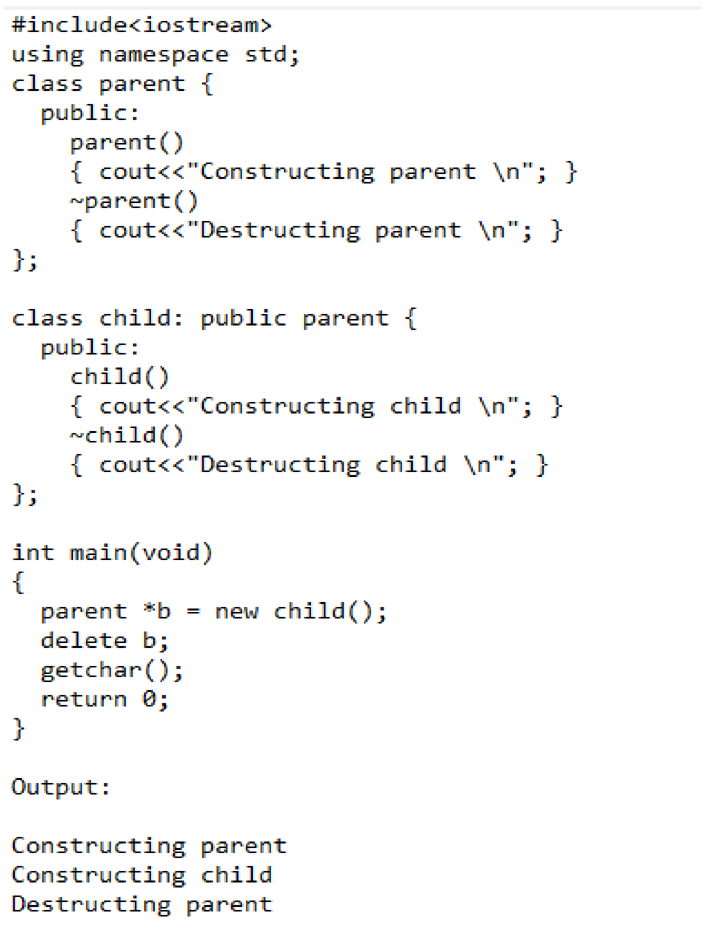
What is a virtual destructor?

When we have a parent class pointer and it is assigned a instance of child class.

Delete the parent pointer will result in free only the parent object.

Virtual keyword to the parent destructor will first delete the child instance and the delete the parent.

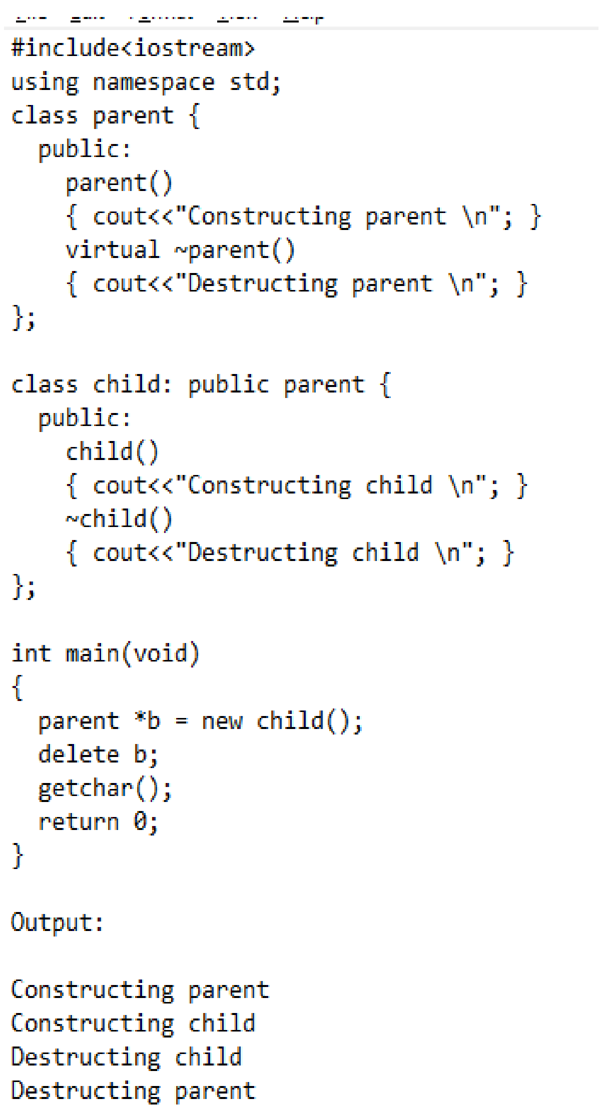
Without Virtual



```
#include<iostream>
using namespace std;
class parent {
  public:
    parent()
    { cout<<"Constructing parent \n"; }
    ~parent()
    { cout<<"Destructing parent \n"; }
};

class child: public parent {
  public:
    child()
    { cout<<"Constructing child \n"; }
    ~child()
    { cout<<"Destructing child \n"; }
};

int main(void)
{
  parent *b = new child();
  delete b;
  getchar();
  return 0;
}
```

```
Output:

Constructing parent
Constructing child
Destructing parent
```

With Virtual



```
#include<iostream>
using namespace std;
class parent {
  public:
    parent()
    { cout<<"Constructing parent \n"; }
    virtual ~parent()
    { cout<<"Destructing parent \n"; }
};

class child: public parent {
  public:
    child()
    { cout<<"Constructing child \n"; }
    ~child()
    { cout<<"Destructing child \n"; }
};

int main(void)
{
  parent *b = new child();
  delete b;
  getchar();
  return 0;
}

Output:

Constructing parent
Constructing child
Destructing child
Destructing parent
```

Classes vs Struct

1. Members of class are private by default, Public in struct.
2. By default, when deriving a struct default base class access specifier is public.

Dynamic Arrays

1 D Array

Example:

```
1D array

int* a;
cin>>n;
a=new int[n];

for(int i=0; i<n; i++)
cin>>a[i];

delete [] a;
```

2D array

```
int main()
{
        char** list;
        cin>>n;
        list = new char*[n];    //define number of rows

        for(int i=0; i<n; i++)
        {
                list[i] = new char[len];        //define the no of columns. len is the max length of a string
                cin>>list[i];                   //make an entry for each row
        }

        print(list,n);
        list[i][j] --> gives the character at i,j subscript


        for(int i=0; i<n; i++)
        delete[] list[i];               //deallocate each colums
        delete[] list;                  //delete rows
}
```

Printing

```
void print(char** list, int n)
{
        for(int i=0; i<n; i++)
        {
                cout<<list[i];
        }
}
```

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.

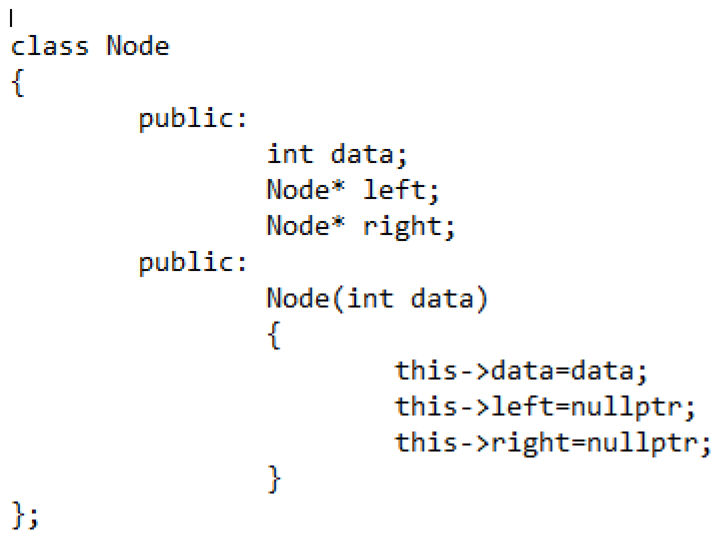
Tree Traversal can be of 2 types

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.

Example: To find height and also print level order



```
class Node
{
        public:
                int data;
                Node* left;
                Node* right;
        public:
                Node(int data)
                {
                        this->data=data;
                        this->left=nullptr;
                        this->right=nullptr;
                }
};
```

```
int height(Node* root)
{
        if(root==nullptr)
        {
                return 0;
        }
        else
        {
                int lheight=height(root->left);
                int rheight=height(root->right);
                if(lheight>rheight)
                {
                        return(lheight+1);
                }
                else
                {
                        return(rheight+1);
                }
        }
}


void printLevelOrder(Node* root, int level)
{
        if(root==nullptr)return;
        if(level==1)
        {
                cout<<root->data;
                return;
        }
        else
        {
                printLevelOrder(root->left,level-1);
                printLevelOrder(root->right,level-1);
        }
}

void printLevel(Node* root)
{
        int height1=height(root);
        for(int i=0; i<=height1; i++)
        {
                printLevelOrder(root,i);
                cout<<endl;
        }
}
```

Depth First

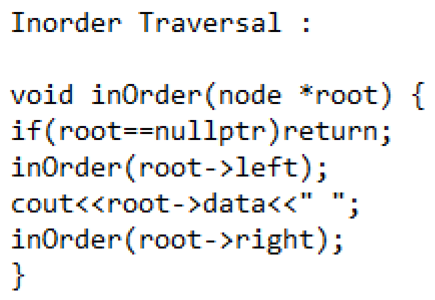
First we move to child only after completing the whole subtree, the next child is accessed.

In Depth first there are 3 types of traversal

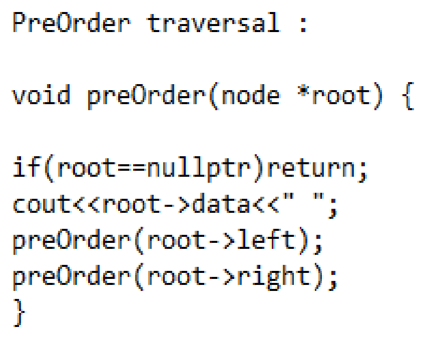
Inorder

Preorder

Postorder



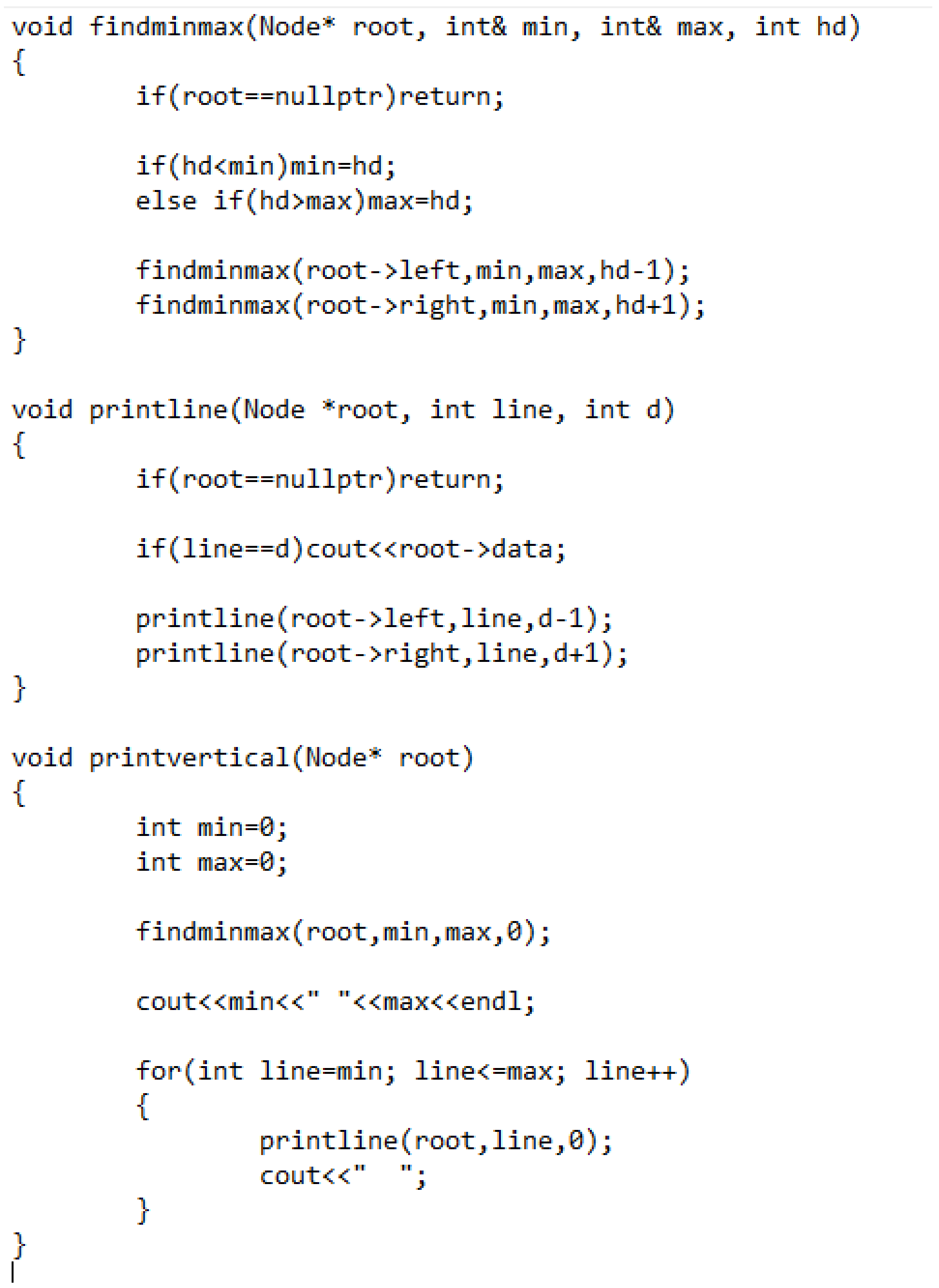
```
Inorder Traversal :

void inOrder(node *root) {
if(root==nullptr)return;
inOrder(root->left);
cout<<root->data<<" ";
inOrder(root->right);
}
```



```
PreOrder traversal :

void preOrder(node *root) {

if(root==nullptr)return;
cout<<root->data<<" ";
preOrder(root->left);
preOrder(root->right);
}
```

```
PostOrder traversal :

void postOrder(node *root) {

    if(root == nullptr)return;

    postOrder(root->left);
    postOrder(root->right);
    cout<<root->data<<" ";
}
```

Print Tree Vertical Order



```
void findminmax(Node* root, int& min, int& max, int hd)
{
        if(root==nullptr)return;

        if(hd<min)min=hd;
        else if(hd>max)max=hd;

        findminmax(root->left,min,max,hd-1);
        findminmax(root->right,min,max,hd+1);
}

void printline(Node *root, int line, int d)
{
        if(root==nullptr)return;

        if(line==d)cout<<root->data;

        printline(root->left,line,d-1);
        printline(root->right,line,d+1);
}

void printvertical(Node* root)
{
        int min=0;
        int max=0;

        findminmax(root,min,max,0);

        cout<<min<<" "<<max<<endl;

        for(int line=min; line<=max; line++)
        {
                printline(root,line,0);
                cout<<"  ";
        }
}
```

Multi Child Tree

```
class Node
{
	public:
		int data;
		vector<Node*> child;
	public:

	Node(int data)
	{
		this->data=data;
	}
};


void printLevelOrder(Node* root)
{
	while(root==nullptr)
	{
		return;
	}
	queue<Node*> q;
	q.push(root);
	while(!q.empty())
	{
		int n=q.size();
		while(n>0)
		{
			Node* temp1=q.front();
			q.pop();
			cout<<temp1->data<<" ";
			for(int i=0; i<(temp1->child).size(); i++)
			{
				q.push(temp1->child[i]);
			}
			n--;
		}
	}
}
```

```
int main()
{
        Node* root = new Node(10);
        (root->child).push_back(new Node(11));
        (root->child).push_back(new Node(12));
        (root->child).push_back(new Node(13));
        (root->child).push_back(new Node(14));
        (root->child[0]->child).push_back(new Node(15));
        printLevelOrder(root);    return 0;
}
```

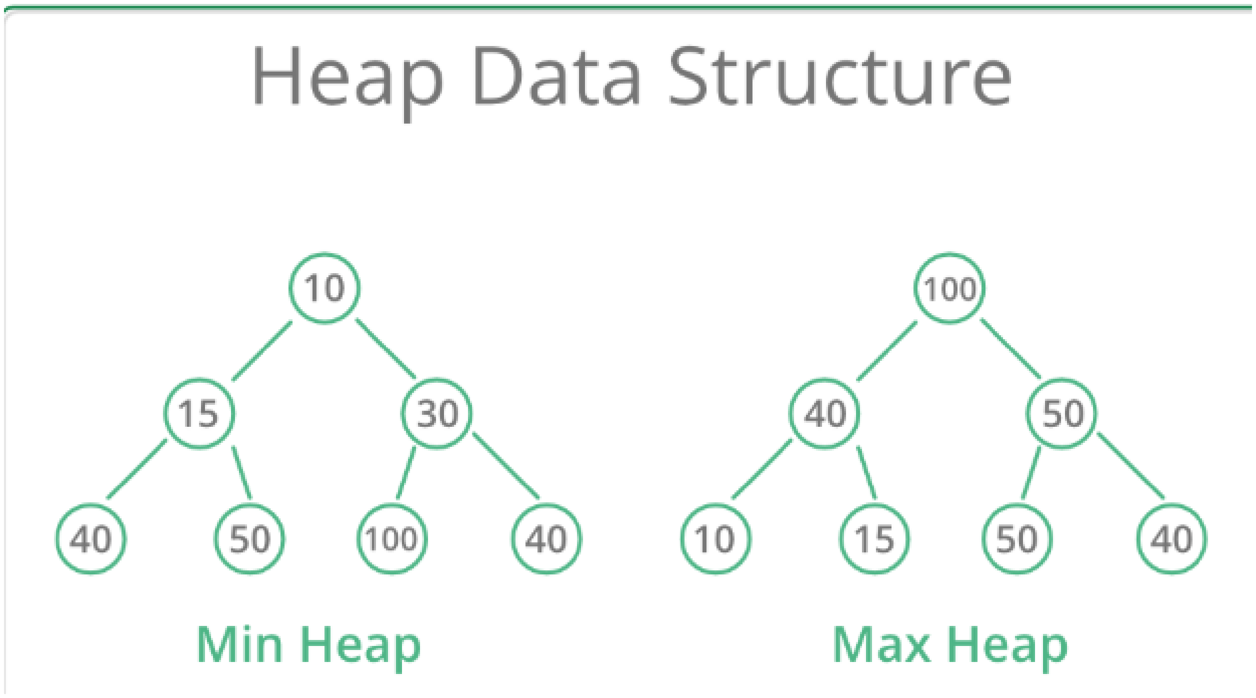
Heap

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

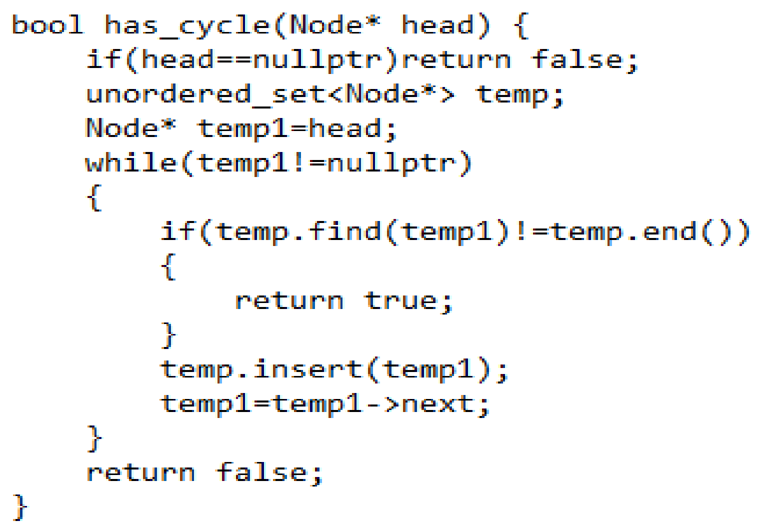
```
#include <iostream>
using namespace std;
struct Node
{
        int data;
        struct Node *next;

        Node()                  //Default constructor initializes to zero
        {
                this->data=0;
        }
        Node(int data)          //parameterized constructor intilaizes to values passed
        {
                this->data=data;
        }
};

int main()
{
        Node* temp = new Node;
        cout<<temp->data;
        return 0;
}

Output:

0
```

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
    a. Assign next to current->next
    b. Assign current->next to prev
    c. Assign prev to current
    d. 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



```
bool has_cycle(Node* head) {
    if(head==nullptr)return false;
    unordered_set<Node*> temp;
    Node* temp1=head;
    while(temp1!=nullptr)
    {
        if(temp.find(temp1)!=temp.end())
        {
            return true;
        }
        temp.insert(temp1);
        temp1=temp1->next;
    }
    return false;
}
```

Remove Duplicates from a Sorted Linked List

```
Node* remove_duplicated(Node* Head)
{
        Node* temp=Head;
        while(temp!=nullptr)
        {
                Node* temp2=temp->next;
                while(temp2!=nullptr && temp->data==temp2->data )
                {
                        Node* temp3=temp2;
                        if(temp2->next==nullptr)
                        {
                                temp->next=nullptr;
                                delete temp3;
                                break;
                        }
                        temp2=temp2->next;
                        delete temp3;
                }
                if(!(temp->next==nullptr))
                {
                        temp->next=temp2;
                }
                temp=temp->next;
        }
        return Head;
}
```

Complexities

Order of complexities

O(1)

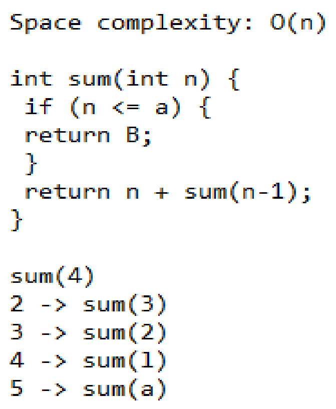
O(log n)

O(n)

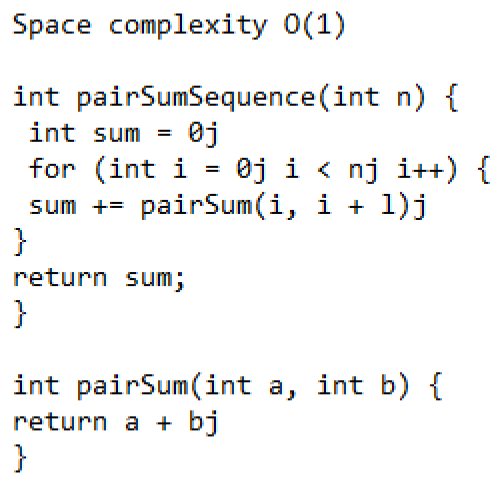
O(n log n)

O(n^2)

Space Complexity: amount of memory required by the algorithm



```
Space complexity: O(n)

int sum(int n) {
 if (n <= a) {
 return B;
 }
 return n + sum(n-1);
}

sum(4)
2 -> sum(3)
3 -> sum(2)
4 -> sum(1)
5 -> sum(a)
```



```
Space complexity O(1)

int pairSumSequence(int n) {
 int sum = 0j
 for (int i = 0j i < nj i++) {
 sum += pairSum(i, i + l)j
}
return sum;
}

int pairSum(int a, int b) {
return a + bj
}
```

Complexity of Binary Search: O(log N)

Complexity of recursive calls: O($Branches^{Depth}$)

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.

```
#include <iostream>
using namespace std;
void bubble_sort(int* a,int len)
{
    for(int i=0;i<len;i++)
    {
        for(int j=1;j<len-i;j++)
        {
            if(a[j-1]>a[j])
            {
                int temp=a[j-1];
                a[j-1]=a[j];
                a[j]=temp;
            }
        }
    }
}

int main()
{
    int a[10]={2,5,3,4,1,9,8,7,6,0};
    bubble_sort(a,10);
    for(int i=0; i<10; i++)
    {
        cout<<a[i];
    }
    return 0;
}
```

Insertion Sort

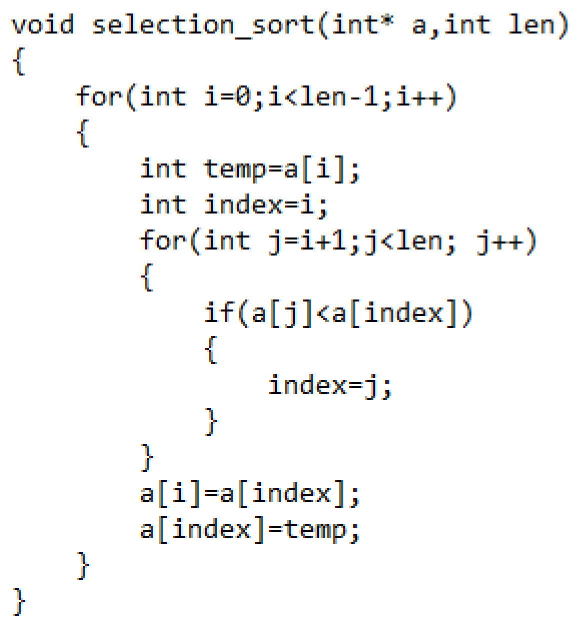
Sorting cards.

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

```
#include <iostream>
using namespace std;
void insertion_sort(int* a,int len)
{
    for(int i=1;i<len;i++)
    {
        int j=i-1;
        int key=a[i];
        while( a[j]>key && j>=0)
        {
            a[j+1]=a[j];
            j--;
        }
        a[j+1]=key;
    }
}
```

Selection Sort

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



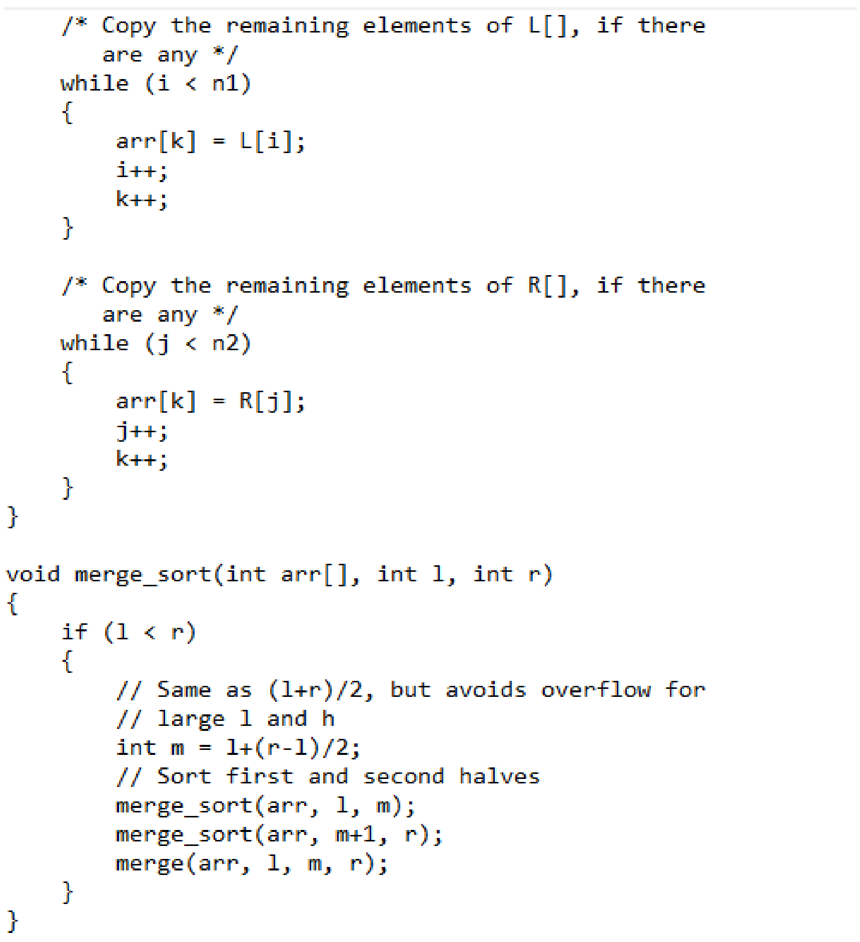
```
void selection_sort(int* a,int len)
{
    for(int i=0;i<len-1;i++)
    {
        int temp=a[i];
        int index=i;
        for(int j=i+1;j<len; j++)
        {
            if(a[j]<a[index])
            {
                index=j;
            }
        }
        a[i]=a[index];
        a[index]=temp;
    }
}
```

Merge Sort

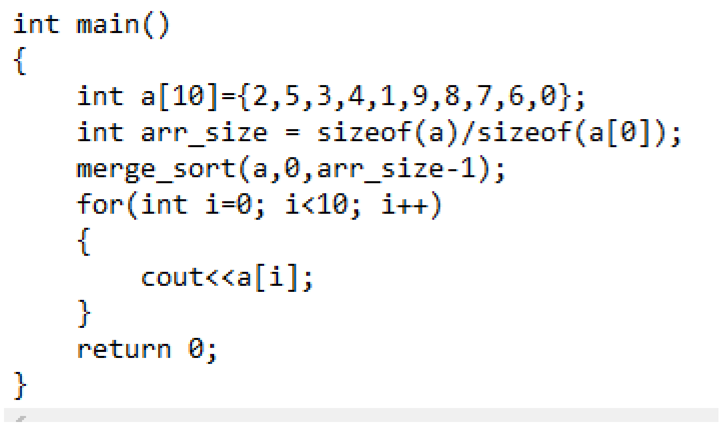
Continuously split the array into two halves.

Merge after it is split in to single elements.

```
void merge(int arr[], int l, int m, int r)
{
    int i, j, k;
    int n1 = m - l + 1;
    int n2 =  r - m;
    /* create temp arrays */
    int L[n1], R[n2];
    /* Copy data to temp arrays L[] and R[] */
    for (i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (j = 0; j < n2; j++)
        R[j] = arr[m + 1+ j];
    /* Merge the temp arrays back into arr[l..r]*/
    i = 0; // Initial index of first subarray
    j = 0; // Initial index of second subarray
    k = l; // Initial index of merged subarray
    while (i < n1 && j < n2)
    {
        if (L[i] <= R[j])
        {
            arr[k] = L[i];
            i++;
        }
        else
        {
            arr[k] = R[j];
            j++;
        }
        k++;
    }
```

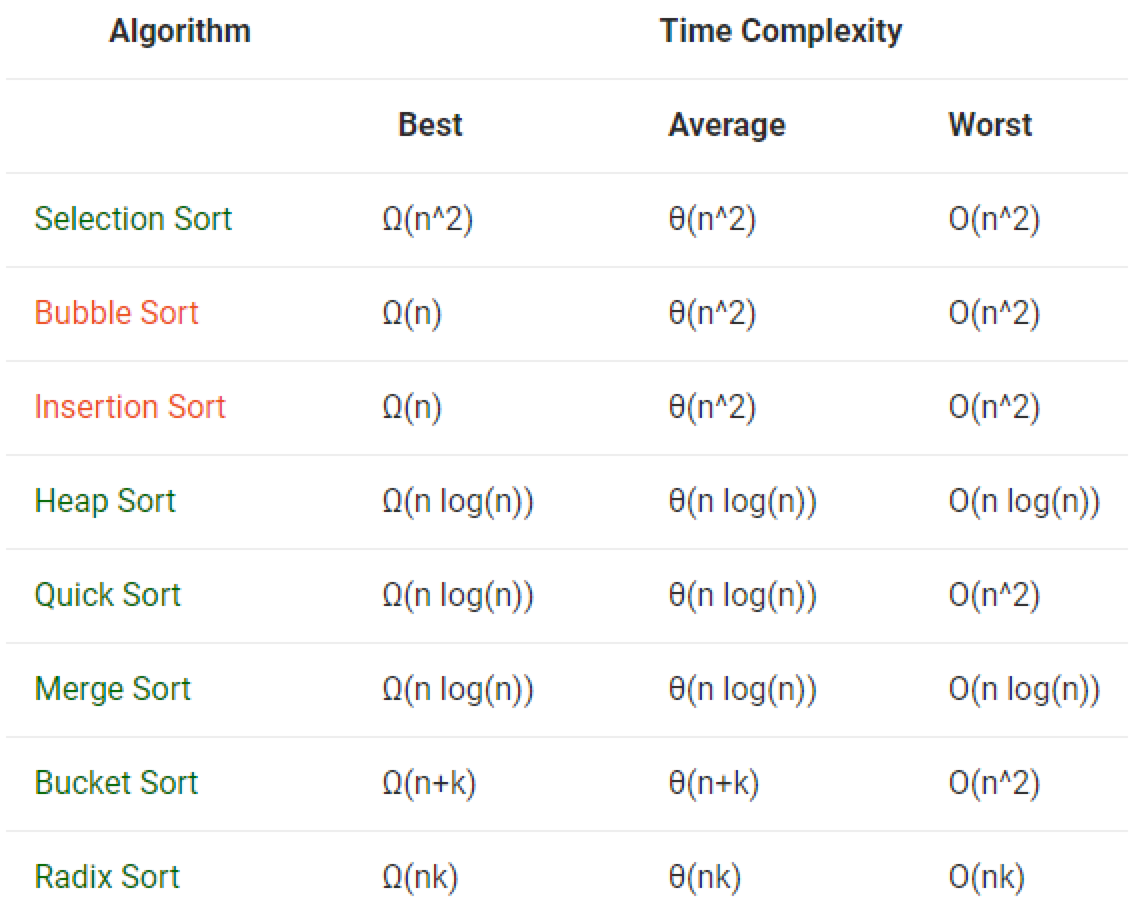


```
    /* Copy the remaining elements of L[], if there
       are any */
    while (i < n1)
    {
        arr[k] = L[i];
        i++;
        k++;
    }

    /* Copy the remaining elements of R[], if there
       are any */
    while (j < n2)
    {
        arr[k] = R[j];
        j++;
        k++;
    }
}

void merge_sort(int arr[], int l, int r)
{
    if (l < r)
    {
        // Same as (l+r)/2, but avoids overflow for
        // large l and h
        int m = l+(r-l)/2;
        // Sort first and second halves
        merge_sort(arr, l, m);
        merge_sort(arr, m+1, r);
        merge(arr, l, m, r);
    }
}
```



```
int main()
{
    int a[10]={2,5,3,4,1,9,8,7,6,0};
    int arr_size = sizeof(a)/sizeof(a[0]);
    merge_sort(a,0,arr_size-1);
    for(int i=0; i<10; i++)
    {
        cout<<a[i];
    }
    return 0;
}
```

Complexities of Sorting Techniques



Algorithm	Time Complexity		
	Best	Average	Worst
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)
Heap Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Quick Sort	Ω(n log(n))	θ(n log(n))	O(n^2)
Merge Sort	Ω(n log(n))	θ(n log(n))	O(n log(n))
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)
Radix Sort	Ω(nk)	θ(nk)	O(nk)

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

Smart Pointers

Pointers which are basically used to remove the process of using New and Delete.

1. Unique Pointer
2. Shared Pointer
3. Weak Pointer

Library:

#include<memory>

Unique Pointer:

1. When the pointer goes out of scope. The pointer frees up the memory.
2. Cannot copy a unique pointer. Because one of the pointers dies the other becomes dangling.

Shared Pointer:

1. Creates a pointer to the referred datatype. But keeps a count of all pointers referencing it.
2. The Pointer when goes out of scope. Does not delete until all pointers referencing the memory location goes out of scope.

Weak Pointer:

1. Same as shared pointer. But does not increment the ref count.
2. Can be copied or assigned.

Common class to demonstrate smart pointers

```
#include<iostream>
#include<memory>
using namespace std;
class Entity
{
        public:
        Entity()
        {
                cout<<"Constructor is called"<<endl;
        }
        ~Entity()
        {
                cout<<"destructor is called";
        }
};
```

Unique Pointer

```
int main()
{
        {
                unique_ptr<Entity> temp(new Entity());
                cout<<"hello"<<endl;
        }
}

Output:

Constructor is called
hello
destructor is called
```

Shared Pointer

```
int main()
{
        {
                shared_ptr<Entity> temp(new Entity());
                {
                        shared_ptr<Entity> temp2;
                        temp2=temp;
                        cout<<"I the pointer dont die here"<<endl;
                }
                cout<<"Not even here"<<endl;
        }
}
```

```
Output:

Constructor is called
I the pointer dont die here
Not even here
destructor is called
```

Weak Pointer

```
int main()
{
        {
                weak_ptr<Entity> temp;
                {
                        shared_ptr<Entity> temp2(new Entity());
                        temp=temp2;
                        cout<<"I the pointer dont die here"<<endl;
                }
                cout<<"Doesnot wait for me weak pointer to go out of scope"<<endl;
        }
}
```

```
Output:

Constructor is called
I the pointer dont die here
destructor is called
Doesnot wait for me weak pointer to go out of scope
```