Variables:

1. Int a=10;
2. Int a {10}; // same as assignment;

Arrays:

1. Int arr[]={0};
2. Int arr[5]={0};
3. Int arr[] {0}; //same as assignment;

* Traversal using for each loop:

For(int x:arr){}//here for each array element a copy is created and given to x

To change something in array:

For(int &x:arr){x=x\*2;} // here we don’t create a copy for each array element instead use alias using reference variable.This is faster way

Character Arrays:

1. Char a[] ={‘a’,’b’,’c,’};
2. Char a[]={‘a’,’b’,’c’,’\0’}; // necessary to terminate with ‘\0’ so that no garbage values are stored after ‘c’.
3. Char a[]=”abc”;
4. Char a[10]; cin>>a;

🡪cout<<a; \\ prints the content of the array and not the first character address. unlike to integer arrays.

🡪to take sentence as input use cin.getline(chararrayname,maxsize,delimit); //i.e cin.getline(a,100,’\n’) .

//’\n’ is by default we can give any delimit where we want to terminate the input like’$’.

🡪another way: getline(cin,a,’\n’);

Strings:

🡪 String is a class in STL;

* string s1=”abc”;
* string s1(“abc”);
* String s2=”def”; String s1=s2;
* String s1(s2);
* Char a[]={‘a’,’b’,’c’,,’\0’}; String s1=a;

🡪string functions;

* String s1;
* S1.append(“def”);
* S1.length();
* S1.compare(s2);
* S1.find(“de”); //gives the index of first occurrence of the searched substring if found else gives a constant called ‘string::npos’ . // s1.find(pattern,start); to find pattern from index start.
* S1.erase(index,length);

🡪strings support operator overloading eg: s1<s2;

🡪tokenize string using strtok() cstring function : first call-strtok(s,delimit); remaining calls-strtok(NULL,delimit);// delimit : “ “ or “;”.etc

🡪Iteration

* For(int i=0;I<str.length();i++);
* For(auto st:str);
* For(auto it=str.begin();it!=str.end();it++);

2D Character arrays:

🡪char a[][]={{‘a’,’b’,’\0’},{‘d’,’e’,’\0’}};

🡪char b[][]={“ab”,”cd”};

🡪char c[5][10];

C[0][0]=’a’; //single initialization

🡪here cout<<a[0]; // prints entire array at row 0;

🡪 for(int i=0;i<n;i++)

cin.getline(a[i],maxsize,delimit); //taking list of strings as input and storing in 2D array;

REFERENCES:

1. Create an alias,must be assigned when declared,cannot be NULL, are constant pointers therefore cannot refer to another variable.
2. Int a=10;

Int &b=a;// reference to a

Pointers:

🡪in char a=’A’; cout<<&a; // it prints A i.e content of the character and not the address of the character. To avoid this use explicitly typecasting form char\* to void\* or int\*.==> cout<<(void \*)&a;// here cout doesn’t know the data type of a.

🡪size of pointer of any data type is same. Int\*,char\*.etc all sizes are same but they point to different datatypes.

🡪char a=’A’; int b=12; char\* ptr=&b; // avoid this because there will be confusion how many bytes to read as int takes 4 bytes and it its pointed by char\* which takes only one byte.

🡪It is necessary to initialize the pointers to null initially else it will be wild pointer.

🡪NULL is used to show that it is an invalid value.it is defined as it is having a value of 0. It can be assigned to any data type. Also helpful in function return values instead of -1 we can return NULL.

FUNCTION POINTERS:

#function pointers stores the address of the text section or the address of the function(set of instructions) are used,like normal pointers we can pass a function pointer to other functions.It is used in sort() like functions where we pass function parameter for compare. It is used in implementation of virtual functions where vtable stores functions pointers to achieve dynamic polymorphism.

* Void fun(){}

Int main(){

Void (\*fun\_ptr)()=&fun;// or (\*fun\_ptr)()=fun;

// like array name, function name also stores the address of function.

// here we are pointing the fun\_ptr to the address of fun.

Fun\_ptr();// or (\*fun\_ptr)(); // calling the function.

}

* Int fun(int a,int b){ return (a+b);}

Int main(){

Int (\*fun\_ptr)(int ,int )=fun;// or auto fun\_ptr=fun; // we use auto keyword.

Cout<<Fun\_ptr(10,20);// or (\*fun\_ptr)(10,20);}

* Passing function pointers as parameters: eg compare function parameter is passed in sort function.

Bool compare(int a,int b){ return abs(a)<abs(b) ;}

Iint main(){

Int arr={-1,10,2,-2}; int n=sizeof(arr)/sizeof(arr[0]);

sort(arr,arr+n,compare); // here sorting by absolute value is done}

LAMBDA FUNCTIONS:

# These are anonymous functions or functions without name introduced in c++ 11. Used to pass functionality to a function (like function pointers). Used for small works that are to be done by functions .We do not need to create a separate function to pass its functionality to other function.

#synatx: []()->.{} where [] is capture list, () is parameter, ->. Is return type but since lambda function is used for small tasks compiler will guess the return type.Thus it is optional to pass ->. {} is body of lambda function.

🡪eg:

Int main(){ int n=4,x=1;

Int arr={10,-2,2,-1};

sort(arr,arr+n,[](int a,int b){return abs(a)<abs(b); }); //we can write more lines in body also, not compulsory to have only one line

// suppose want to use a local variable from the main function in the lambda function we need to pass it in the capture list.

//Here if we need to use x thus pass x in capture list [x](int a,int b){return abs(a)<abs(b); here it is pass by value. other way is to use reference &x [](int a,int b){return abs(a)<abs(b);

🡪eg:

Int main(){int n=4,x=1;

Int arr={10,-2,2,-1};

auto mycmp=[](int a,int b){return abs(a)<abs(b);}; // storing it in a variable mycmp and passing this variable to sort function.

Sort(arr,arr+n,mycmp); }

🡪eg: for\_each() function: //#include <algorithms>

Int main(){

Vector<int> v;

for\_each(v.begin(),v.end(),[](int &x){return x=x\*2;}); // for\_each() function takes a container’s start ,end and work to be done and does the work on each element. Here we are passing by reference in order to reflect changes in original data (of container).

}

🡪eg: count\_if() functions: //#include <algorithms>

Int main(){

Int res;

Vector<int> v{10,3,20,30};

Res=count\_if(v.begin(),v.end(),[](int &x){return x>10;}); // count\_if() function takes a container’s start ,end and work to be done (it should return a bool value and according to true condition of this bool value it counts the no of elements satisfying this true condition) and does the work on each element. Here we are passing by reference in order to reflect changes in original data (of container). Here it will show true for 20,30 and return 2 as count.

}

🡪 eg: find\_if() functions: //#include <algorithms>

Int main(){

Vector<int> v{10,3,20,2,30};

auto it=find\_if(v.begin(),v.end(),[](int &x){return x<10;}; // find\_if() function takes a container’s start ,end and work to be done. It returns iterator pointing to the first element that satisfies the given condition. Here it will return iterator pointing to the first element less than 10 i.e 3

cout<<\*it;

}

🡪 eg: accumulate() function: //#include <algorithms>

Int main(){

Vector<int> v{10,3,20,30};

Int res1,res2;

Res1=accumulate(v.begin(),v.end(),0); // accumulate() function by default calculate the sum of all elements with initial sum as given in parameter(here 0); It can also do other arithmetic functionality operations. It takes a container’s start ,end and initial sum and work to be done

Res2=accumulate(v.begin(),v.end(),1,[](int x,int y){return x\*y;}); // here accumulates does multiplication of the x with 1 (and y is the elements in the container one by one) and pass its result as parameter x and repeats for all elements. And returns final result to res.

}

🡪CAPTURE LIST IN LAMBDA EXPRESSIONS:

[] : Nothing from local environment can be accessed by lambda exp;

[=]:all local items can be accessed by value

[&]: all local items can be accessed by reference

[=,&x] : all local items can be accessed by value except x which is accessed by reference

[&,x] : all local items can be accessed by reference except x which is accessed by value

Static and global variables are always captured or allowed access in lambda exps,no need to write them in capture list.

🡪eg: int main(){

Int x=10,y=20;

auto lambda\_exp=[&](int a){ x=x+a;y=y=a;} //all local items are accessed by reference and original values can be modified.

Lambda\_exp(5);//passing value of a in lambda\_exp

Cout<<x<<y;} // change in original values

//o/p: 15 25

🡪 eg: int main(){

Int x=10,y=20;

auto lambda\_exp=[=](int a)mutable{ x=x+a;y=y=a;} //all local items are accessed by value,here mutable is used because we can’t even modify these copies if not written mutable also the original values can’t be modified.

Lambda\_exp(5);//passing value of a in lambda\_exp

Cout<<x<<y;} // no change in original values

//o/p: 10 20

🡪 eg: int main(){

Static Int x=10,y=20;

auto lambda\_exp=[](int a){ x=x+a;y=y=a;} //since static all local items are by default accessed no need to capture them.

Lambda\_exp(5);//passing value of a in lambda\_exp

Cout<<x<<y;} // change in original static values

//o/p: 15 25

SMART POINTERS:

#We need to free the memory created dynamically else will create problems if delete part not written. For such cases smart pointers are used.

🡪user defined smart pointer classes:

Class SP{ // it wraps an int pointer within it.

Int \*ptr;

Public: SP (int \*p=NULL){ptr=p;} //if not allocated memory parameter then point null. Set the prt pointer this dynamic created memory.

~SP(){delete ptr;}

Int &operator\*(){return\*ptr;} // we pass the value of ptr by reference here

};

Int main(){

SP spobj(new int());

\*spobj=20; //we have overloaded \* operator therefore the reference is received as lvalue and is assigned value 20

Cout<<\*spobj;

return 0;}//after this destructor of spobj is called and performs delete operation of ptr memory.

🡪eg:

Class test{

Int x,y;

Public: test(int a=0,int b=0){x=a;y=b; cout<<”constructor”;}

~test(){cout<<”destructor”;}

};

Int main(){cout<<”main starts”;

{Test \*t=new test(10,20);

}cout<<”main ends”;

}

//o/p: main starts constructor main ends //here object is of special type i.e pointer so it is not automatically deleted or destructed .we need to write delete part for it.

So we use the SOLUTION of smart pointers:

Class sp{

test ptr;

Public: sp(test \*p=null){ptr=p;}

~sp(){delete ptr;}

Test &operator\*(){return \*ptr;} // overloading operator \* to refer them.

Test \*operator->(){return ptr;} // overload -> to access elements

};

Int main(){cout<<”main starts”;

Sp spobj(new test(10,20); cout<<”main ends”;} //sp class destructor will delete the dynamic memory of test class.

//o/p: main starts constructor destructor main ends

* Template smart pointers can be used for any datatype

Template<class t>

Class sp{

t ptr;

Public: sp(t \*p=null){ptr=p;}

~sp(){delete ptr;}

t &operator\*(){return \*ptr;} // overloading operator \* to refer them.

t \*operator->(){return ptr;} // overload -> to access elements

};

Int main(){cout<<”main starts”;

Sp<int> spobj(new int(10,20); cout<<”main ends”;} // sp object of type int

* Problem with smart pointer:

//assume if main function is like:

Int main(){

Int \*ptr1=new int(10);

{

Int \*ptr2=ptr1; //same memory pointed by ptr1,ptr2

Sp<int> spobj(ptr2); // passing address of ptr 2

} // here spobj destructor deletes the memory of ptr2 but that memory was also pointed by ptr1.so ptr1 loses access to it.

Cout<<\*ptr1; // this has no access to that memory ,therefore error is created. To solve this issue we need to check the reference count for the dynamic memory and if it its 0 then only delete it else do not delete that memory.

Return 0;

}

🡪c++ library implementation of smart pointers:

# smart pointers are useful to delete the objects pointers and memory locations incase if we miss the delete code.

1. Unique\_ptr:

# It is simple and similar to smart pointer we created above .it deletes the memory for the dynamically created ones after its scope has ended. It allows pointing only one pointer to a memory since if more than one pointer are pointing to same location and once the object destructor is called for first pointer the memory is deleted and the other pointer also loses access to that memory since no reference count is maintained heres. So it points only one pointer to a memory, therefore there is no copy constructor inside this class to pass a pointer type and point to the same memory. It causes very less overhead since it points only one pointer so no reference count maintained.

Eg:

Class test{

Int x;

Public: test(int a){x=a; cout<<”constructor”;}

~test(){cout<<”destructor”;}

Void fun(){cout<<x;}};

Int main(){

unique\_ptr<test> ptr = make\_unique<test>(10);// or Unique\_ptr<test> ptr (new test(10);

This way of declaration is not recommended.

Ptr->fun();

Unique\_ptr<test> ptr2=ptr;//this is not allowed since unique\_ptr can point only one pointer to any memory. Therefore there is no copy constructor or assignment operator like thing allowed . will give compiler error.

Unique\_ptr<test> ptr3= move(ptr);

//but to change the ownership of the memory from one pointer to another we can do it }

//o/p: constructor 10 destructor

1. Shared\_ptr:

# multiple shared pointers can hold and point to same memory or can have co-ownership to same memory. Here the memory is deleted only when reference count is 0. Else though the objects or pointers are deleted but the memory remains.

Eg: class test{ int x;

Public: test(int a){x=a;cout<<”constructor”;}

~test(){cout<<”destructor”;}

Void fun(){cout<<x;}};

Int main(){ cout<<”main starts”;

shared\_ptr<test> ptr; //just declaration , no object creation so no constructor called

{

Shared\_ptr<test> ptr1=make\_shared<test>(10); // creation and initialization of object. This prints “constructor”

// or shared\_ptr<test> ptr1 (new test(10));

Ptr=ptr1; //memory is shared by both now.

Cout<<ptr.use\_count(); // returns reference count (total objects or pointers pointing) here 2

Cout<<ptr1.use\_count();// returns reference count (total objects or pointers pointing) here 2

} // now object of ptr1 is deleted so its destructor is called and ptr1 is deleted and reference count becomes 1

Cout<<ptr.use\_count(); //here 1

Cout<<”main ends”;

} // now ptr is also deleted and reference count is 0 so memory is also freed.

//o/p: main starts constructor 2 2 1 main ends destructor

1. Weak\_ptr:

# These are a kind of temporary pointers. These are used with shared pointer but they do not have the ownership of the memory so they do not increase the reference count. We can access the objects using this weak pointer but we do not increase the reference count.

# weak pointers are also used to overcome cyclic problems for smart pointer classes. If there are 2 classes and both has a shared pointer of each other’s class so we cannot delete any of this memory pointed since reference count never becomes 0. For this we can make one of the shared pointer as weak pointer . now on deleting the only one remaining shared pointer will make reference count to 0.

Eg:

class test{ int x;

Public: test(int a){x=a;cout<<”constructor”;}

~test(){cout<<”destructor”;}

Void fun(){cout<<x;}};

Int main(){

Weak\_ptr<test> ptr1; //we do not initialize it with any value since it doesn’t have ownership to that memory.

{shared\_ptr<test> ptr2=make\_shared<test>(10); // create a shared ptr

Ptr1=ptr2; // assign weak pointer to this shared ptr

// Weak\_ptr<test> ptr3= lock(ptr2); //if we want to upgrade to shared pointer and give ownership to this weak ptr so reference count increases.

Cout<<ptr2.use\_count();

}

Cout<<ptr2.use\_count();

Cout<<ptr1.expire(); // checks if the memory pointed by weak object is deleted or not. This checks if object is valid or not.

}

//o/p:

Memory Allocation:

Static/compile time allocation: The memory that will be required is defined ,the size of the memory will be fixed for the total no of variables in the program, by the compiler using the virtual memory mapping where the symbol table stores the memory sizes required by the variables (including pointer variables) and just before the program starts execution the memory gets allocated from the static part of the RAM. Allocation + deallocation is defined by compile time.

Adv:fast;

Disadv: less flexible;

Dynamic/runtime allocation: The memory gets allocated during the execution of the program from the DYNAMIC/Heap part of the RAM. The pointer variables sizes are allocated during the compile time and after dynamic allocation of the memory is done these pointer variables points to these dynamic memory. These pointers can point to another memory dynamically allocated after the previous memory is freed. The code for this type of memory allocation needs to be written by the programmer who defines the sizes of the memory by himself as compared to compile time allocation where the size of memory is defined during compile time and code also not written by user.

#new is an operator whereas malloc,calloc etc are functions. New is used for dynamic memory allocation,initialization and returns to pointer to the memory allocated, also it calls constructors for objects class/structs.

Adv:more flexible(size of memory can be defined and adjusted by user);

Disadv: non freed memory cannot be used creating memory leaks.

Comparison between static and dynamic allocation:

🡪int a[100]={0}; // static allocation;

Cout<<sizeof(a); // gives size of complete array since it points to start of the array allocated in static part. With all elements initialized to 0.

Cout<<a; //prints address of first element.

🡪 int \*a= new int[100]{0}; // dynamic allocation; here pointer size(4B) is allocated during runtime in the static part of the RAM but the memory of 100 integers(array) is allocated during runtime from heap and its address is passed to the pointer. With all elements initialized to 0.

Cout<<sizeof(a);// here doing sizeof(a) will give 4B which is actually size of int pointer variable.

Cout<<a; //prints address of first element ;

delete [] a;// to free array memory; (for normal dynamic variable deletion : delete a; for array: delete [] a;)

2D Dynamic array:

🡪the 2d array is not directly created like int \*a=new int [][]; we create a 1D array of pointers to pointers. where each element points to an array.

int \*\*a[]=new int[size]; // array for rows

for (int i=0;i<n;i++) a[i]=new int[size]; //each row points to another array which will be the elements in that row.

STRUCTURES:

🡪declaration:

* Struct A{

Int x;

Int y;

};

Int main(){ struct A a;// or A a; // in c++ struct is optional but in c it is mandatory.}

* Typedef struct A{

Int x;

Int y;

}newA;

Int main(){ newA a;}

🡪initialization:

* Struct A a={10,20}; // or A a={10,20;
* A a;

a.x=10;

a.y=20;

🡪 designated initialization:

Struct A a={.y=10,.x=20}; // here order is not necessary. Also if only one value is initialized others are assigned value 0. // this is allowed in c but in c++ it may give error.

🡪structures vs class:

* By default access in class is private but in structures it is public. We can make it private by using private keyword.
* By default inheritance is private in class if not specified but it is public in structues. We can make it private by using private keyword.

Struct a{};

Struct b:a{};//it is public by default. // struct b:private a{};

Class a{};

Class b:a{};//it is private by default. // class b:public a{};

* We can use constructors, destructors, functions, access modifiers in structures same like in classes.

Struct a{

Private: int a;

int b;

public:

A(int a,intb){

X=a;y=b;

}

}

* We use structures when we need to bundle data and functions specific to it. We use class for object oriented programming and data hiding.

🡪array of structures:

Struct A{ int x;};

Int main(){

A arr[10];

For(int i=0;i<10;i++){

arr[i].x=i; // each array element is a structure having element x}

}

🡪passing structures to functions:

Struct Aa{};

Fun1(A a){}

Fun2(A &a){}

Fun3(A \*a){}

Int main(){ A b={10,20}; Fun1(b);Fun2(b);Fun3(&b);}

🡪alignment in structures:

* A structure has alignment requirements or padding according to the member with the largest datatype size.It is recommended to declare the member variables in the structures either in ascending or in descending order so as to reduce padding.
* Padding is done because though the memory is byte addressable but the cpu reads the memory in form of words of 4bytes at a time in 32 bit processor, ,8bytes for a 64 bit processor .etc for a single read cycle.
* Struct a{double d1;char c1;char c2;} //To read a double of size 8, cpu may need 2 cycles( 2 bytes for char + 6 bytes for double and another cycle for remaining 2 bytes of double) if not aligned properly. But if all items are aligned according to size 8 than this can be done in one read cycle.
* Compilers do required alignment according to the machine.
* We can do compact representation of structures :

Struct a{double d1;char c1;char c2;}\_ \_attribute\_ \_((packed)); int main(){ sizeof(a);}

UNIONS:

🡪syntax same as structures. Not necessary to use union keyword inside main fun In c++ but is must in c.

🡪memory is allocated according to largest member size and shared among all elements such that all elements point to the same memory location.

🡪applications:type punning, using same code for binary search tree as well as doubly linked list.

* Type punning:convert one type to another type without explicit typecasting and we get the internal representation of that particular type.

1)Union A{ int x;float y;}; int main(){A b;

b.y=1.1; cout<<b.x;}// op:1066192077 we get the decimal value equivalent to the binary representation of 1.1

2)union A{int x;char arr[4];}; int main(){ A b; b.x=512; cout<<b.arr[0]<<b.arr[1]<<b.arr[2]<<b.arr[3];} //op:0 2 0 0 we get the decimal value equivalent to the binary representation of 512. Means 512 requires 4B for its internal representation and first Byte value is 0(in decimal),second Byte value is 2(in decimal) and so on.

🡪 struct node{

Int data;

Union {

Struct {node \*left,\*right;};//a node structure that can be used for both binary tree and

doubly linked list.

Struct {node \*prev,\*next;};

};

}

Generic programming:

🡪 template<typename T,typename U> or template<class T,class U> //T , U can be of any datatype ,container

U funname(T start,U end){ //this function can take any datatype or container or iterator and

// can return any of these.

return something;

}

Templates:

# Used when same functionality has to be provided for various datatypes.write once,use for any datatype. These are like macros as the passed parameter replaces all instances of templatename, also are processed by compiler,but better than macros as typechecking is done. Two kinds of templates: 1) function templates(sort,binary\_search,linear search) 2) class templates(stack,queue,deque)

🡪function templates:

Eg: template<typename T,int limit> //here we are passing non-datatype parameter(will not change for any data type passed)-int limit // this has to be a const parameter as its execution is done during preprocessing phase by compiler. So if we pass <int,x> then it will give error as during preprocessing compiler doesn’t have the const value of x.

//here we are also providing limit upto where we want to process the array.

T arrmax(T arr[],int n){

If(n<=limit){//code for max of array}}

Int main(){int arr[]={1,12,2};

Cout<<arrmax<int,3>(arr,3); // passing functionality for int and limit is 3

Float arr[]={1.1,5.2,2.1,2.0};

Cout<<arrmax<float,3>(arr,4); // passing functionality for float and limit is 3}

🡪class templates:

Eg: template<typename T>

Struct pair{

T x,y;

Pair(T i,Tj){x=i;y=j;}

T getfirst(); // just declaration

T getsecond(){ return y;}

};

//defining body of getfirst()

template<typename T>

Pair<T> ::getfirst(){

return x;}

Int main(){

Pair<int> p1(10,20),p2(30,39);

Cout<<p1.getfirst()<<p2.getsecond();}

//o/p: 10 39

VECTORS:

#insert,erase,resize are O(n); rest-- push\_back,pop\_back(user defined may take more time),front,back,empty,size,begin,end,rbegin,rend,cbegin,cend,crbegin,crend,size() takes O(1);

#avg time to push\_back an element in vector is O(1);

🡪initialization:

* Vector<int> v1;

# by default all values are assigned to 0 in vectors unlike in arrays having random values.

* Vector<int> v2(5,10);
* Vector<int> v3{1,2,3,4,5};
* Vector<int> v4(v2);
* Int arr[]={1,2,3}; int n=sizeof(arr)/sizeof(arr[0]); vector<int> v5(arr,arr+n) // now v5 has 1,2,3 as elements. Here we created a container using another container.
* Vector<vector<int>> mat(n,vector<int> (m,0)); // 2-d array matrix using vectors
* V.at[i] is same as v[i] but it does array index out of bound checking (accessing index more than size of arrays) and gives exception. Whereas v[i] doesn’t do this. V[i] returns the reference to the ith element so we can change it.

🡪insertion:

* Vector<int> v2(5,10); // initializes 5 elements to 10;
* Vector<int> v3{1,2,3,4,5}; // initializes according to given numbers;
* V1.push\_back(1); //single insertion, it doubles the capacity if it becomes full and new element has to be inserted.
* V1.insert(v.begin()+2,91); // inserts 91 at 3rd index i.e at 2nd after 2nd position.
* V1.insert(v.begin()+3,4,31); //inserts 4 times 31 after 3rd position i.e from index 4;

🡪deletion:

* V1.pop\_back();// deletes last element
* V1.erase(v1.begin()+4); // deletes 4th index element
* V1.erase(v1.begin+2,v1.begin+6); // erases form range 2nd index to 5th index
* V1.clear(); //deletes all elements but vector capacity memory is still there.

🡪iteration:

* For(i=0;I<n;i++);
* For(auto it=v1.begin();it!=v1.end();it++);
* For(auto it=v1.rbegin();it!=v1.rend();it++); // there is also rbegin,rend,cbegin,cend,crbegin,crend functions to iterate the vector. //in cbegin and all similar functions the elements are constant and cannot be changed during iteration of the container.
* For(auto i:v1);

🡪functions:

* V1.push\_back();//inserts element at last positon
* V1.pop\_back();//deletes last element
* V1.insert(v1.begin()+1,100);//inserts element 100 and returns pointer of that location index.

V1.insert(v1.begin()+4,2,20);//inserts 20 2 times

V1.insert(v1.begin(),v2.begin+3,v2.begin+8);//inserting elements from vector v2 (starting from v2.begin+3 till v2.begin+8-1)into an empty vector v1

* V1.front(); // returns and gives front element by reference.
* V1.back();//gives last element by reference.
* V1.empty(); //checks if vector is empty or not.
* V1.clear();//clears all vector elements.
* V1.size();// gives no of elements
* V1.resize(3);//changes size of vector to 3; if original size was greater than it will delete the extra size elements and the size of vector (Thus it is not good to use it to reduce size). If original size was small and we are increasing the size than the extra elements are set to default value(0 for int).

V1.resize(3,100);//we specify the value 100 to set to the extra elements.

* V1.capacity(); // gives total memory capacity of vector
* V1.max\_size(); // gives max no of values that can be inserted in worst case in available system memory.
* V1.reserve(100); //fixes 100 size capacity so that no time waste in doubling the vector capacity for size full.

PAIRS:

pair<int,int> p;

p.first=1;p.second=2;

pair<int,int>p1(p2);

pair<int,string>p3=make\_pair(1,”aab”);

🡪declaration and initialization:

pair<int,string> p1(10,”aab”),p2,p3,p4;

p2={10,”aab”};

p3=make\_pair(10,”dgf”);

p4.first=21;p4.second=”ggg”;

🡪comparison of pairs:

Cout<<(p1==p2)<<(p1!=p2)<<(p2<p3)<<(p3>p1);

o/p:1 0 1 1 //for == and != both values are compared . for < ,> by default comparison is done on first value,if same then second value comparison is done.

FORWARD LIST:

#It is singly linked list. Include header file <forward\_list>

🡪initialization and operations:

<forward\_list> l={10,20,30};

🡪l.push\_front(5); // adds item at front of list.

🡪l.pop\_front(5); // deletes item at front of list.

🡪l.assign({10,20,30,10});// assigns the list with the elements given

l.assign(l2.begin(),l2.end());// assigns l with all items of l2 from start to end.

l.assign(5,10); // creates a linked list of size 5 with 5 instances of 10. Used to create a linked list with multiple instance values. Here all elements are 10.

🡪l.remove(10); // removes all instances of 10 from list.

🡪auto it=l.insert\_after(l.begin(),8); // inserts 8 after begin and returns the pointer to the element inserted

auto it= l.insert\_after(it,{2,3,5}); //inserts the elements 2,3,5 after pointer it. And returns pointer to last element inserted.

🡪auto it=emplace\_after(it,40); // same as insert but are optimized. Better for user defined large objects.. And returns pointer to last element inserted.

🡪auto it=erase\_after(it); // deletes the item after the element pointed by pointer it. And returns pointer to the element after pointer it. (element at pointer it doesn’t change. Value after that is deleted )

🡪l.clear(); //clears content of list

🡪l.empty(); // returns true if empty else returns false.

🡪l.reverse(); // reverses all the elements

🡪l.merge(l2); // merges items of two sorted lists ( list l2 with list l ) in sorted way and makes list l2 empty.

🡪l.sort(); // sorts the list l

🡪 time complexities:

Insert\_after(),erase\_after() o(1) for 1 item and o(m) for m items

Push\_front(),pop\_front() o(1)

reverse() -O(1)

sort() – o(nlogn)

remove() – o(n)

assign() – o(1) for 1 item and o(m) for m items

# forward lists are used for inserting,deleting from middle or creating x instances of y type problems since we do not need to create auxiliary space to copy the items (like in vectors), we can store the items in the list and modify them in same list.

#merge is faster in lists. Since in other d.s like vectors we need to create auxiliary space of m+n for arrays of size m,n . But here We can modify in same list after inserting elements so no need for extra space.

LIST:

# it is doubly linked list. Include header <list>

🡪initialization and operations:

* list<int> l={1,2,3};
* l.push\_back(10); // inserts into back
* l.push\_front(20);// inserts in front
* l.pop\_back(); // removes last element
* l.pop\_front(); // removes front element
* auto it=l.insert(l.begin(),10); // or auto it=l.begin(); it=l.insert(it,10);// inserts 10 in front of element pointed by pointer it.And returns pointer to element inserted.

l.insert(it,2,7);//here we insert element 7 , 2 times in front of element pointed by pointer it. And returns pointer to first occurrence of element inserted.

🡪 l.front(); // gives reference of first element

* l.back(); // gives reference of last element
* l.size(); // gives size (or no of elements)
* auto it=l.erase(it); // deletes element pointed by pointer it and returns pointer to element next to deleted one.
* l.remove(40); // removes all occurrences of 40
* l.merge(l2); // merges list l with list l2 (both are sorted lists and the result is sorted list) also result is stored in list l and list l2 becomes empty. We do not need extra space to merge.
* l.unique(); //removes only the consecutive occurrences of the elements. (like {10,20,20,30,30,10} will result in {10,20,30,10} we do not check all occurrences of the elements just check consecutive occurrences.)
* l.sort(); // sorts
* l.reverse(); // reverses all elements

🡪Time complexities:

* front(),back(),size(),begin(),end(),erase(it),push\_front(),pop\_front(),push\_back(),pop\_back() -o(1)
* reverse(),unique(),remove() -o(n)
* sort -o(nlogn)

DEQUE:

#allows front and back delete, insert and random access in o(1). Circular arrays also provide this feature but their size is fixed so we need to create double size array and copy data which is costly . In doubly linked lists all the features are there ,also size is not fixed but it doesn’t provide random access.

🡪initialization and operations:

* deque<int> dq={1,2,3};
* dq.push\_back(4); // inserts 4 in end
* dq.push\_front(0);//inserts 0 in front
* dq.front();// gives reference to first element
* dq.back(); // gives reference to last element
* dq.pop\_front(); // deletes first element
* dq.pop\_back(); // deletes last element
* auto it=dq.begin(); dq.insert(it,10);//inserts 10 in front of element pointed by pointer it

🡪time complexities:

* push\_front,push\_back,pop\_front,pop\_back -o(1)
* insert and erase -o(n)

STACK:

#LIFO OR FILO D.S. INCLUDE header <stack>

#it is also called container adapter since it can be implemented on any underlying data structure that provides operations like : push\_back,pop\_backs,top,empty,size. Hence it is like an interface and It can be implemented using lists,deque,vectors.

🡪initialization and operations:

* stack<int> s;
* s.push(5); // inserts 5
* s.pop(); // deletes top element
* s.top(); //returns top element
* s.empty(); //returns true if stack is empty else false
* s.size(); // returns size(no of elements) of stack

🡪time complexities:

Push,pop,top,empty,size -o(1);

QUEUE:  
# FIFO or LILO d.s . include header <queue>

#it is also called container adapter since it can be implemented on any underlying data structure that provides operations like : push\_back,pop\_front ,front,back,empty,size. Hence it is like an interface and It can be implemented using lists,deque.(vectors doesn’t provide pop\_front in o(1) thus we don’t use them).

🡪initialization and operations:

* queue<int> q;
* q.push(10); // inserts 10 in queue at back
* q.pop(); //deletes front element
* q.front(); //returns front element
* q.back(); //returns back element
* q.empty(); //returns true if empty else false
* q.size(); //returns size(no of elements) in queue

🡪time complexities:

* push,pop,empty,size,front,back -o(1)

PRIORITY QUEUE:

# it is implementation of heap d.s but is implemented using dynamic arrays (vectors).By default it is max heap in c++ whereas in java it is min heap.

Use header # include <queue>

🡪initialization and operations:

* priority\_queue<int> pq; // just declaration and then to insert push the elements.
* priority\_queue<int> pq(arr,arr+n); // initialization of max heap from elements of an existing array using constructor of priority queue. This way is better than normal declaration and has less time complexity.
* Priority\_queue<int,vector<int>,greater<int>> pq; // min heap creation
* Other way to create min heap is just create normal max heap and while inserting make the element negative.this will make it min heap. Eg:arr[i]=-arr[i];
* Struct A{int age;int height;A(int a,int h){age=a;height=h;}}; //user defined datatype

Struct mycmp{bool operator() (A const &obj1, A const &obj2){return obj1.height<obj2.height}}; // we make a constructor for comparison in decreasing order(min heap) for user defined datatype.

Priority\_queue<A,vector<A>,mycmp> pq; // min heap for user defined data type

* Pq.push(10); //inserts 10
* Pq.pop(); //deletes top (current max) for max heap and current min for min heap
* Pq.top(); //gives or returns current max element for max heap and current min for min heap
* Pq.empty(); //returns true if empty else false
* Pq.size(); //returns size of pq

🡪Time complexities:

Priority\_queue<int> pq; // in creating a priority queue- o(n) (n:no of elements in array)

Push,pop -o(log n)

top,size,empty -o(1)

🡪applications:

Dijkstra,prims,huffman,heap sort,and any other place where is heap used

SET:

#set is implemented using red black trees ,it is a self balancing tree. Use header #include<set>

🡪initialization and operations:

* Set<int> s; // increasing order set

Set<int,greater<int>> s; //decreasing order set

* S.insert(2); // inserts 2 if not present earlier (it doesn’t inserts duplicates)
* S.clear(); // deletes all elements
* S.find(20); // finds 20 if present returns pointer to element found else returns pointer to s.end()
* S.lower\_bound(10);// (>=) if present returns pointer to element found else if not present returns pointer to element just greater than the searched. If searched element is largest than the largest one in the set then the pointer to s.end() is returned.
* S.upper\_bound(10);//(>) it returns pointer to next greater element whether found or not . If searched element is largest than the largest one in the set then the pointer to s.end() is returned.
* S.count(); // returns 1 if element is present else returns 0. Here in set count() works same as find() as in set no duplicates are found so count will always be either 1 or 0.
* S.empty(); //returns true if empty else false
* S.size(); // returns no of elements
* S.erase(10); //deletes 10 if present in set else ignores it, here we pass value in erase()

Auto it =s.find(2); //it finds 2 in o(log n)

s.erase(it); //deletes value at pointer it , this is amortized erase(passing address in erase fun) done in o(1)

s.erase(it,s.end()); //deletes range of values from pointer it to end

🡪Time complexities:

Begin(),end(),rbegin(),rend(),cbegin(),cend(),crbegin(),crend() ,erase(it) (--amortized) -o(1)

Insert,find,count,lower\_bound,upper\_bound,erase(val) - o(logn);

🡪user defined data type with set:

Struct A{ int x; bool operator< (const A &obj){ return(this->x<obj.x) ;}};

Int main(){set<A> s;

s.insert({10}); s.insert({20}); // we insert using {} for user defined datatypes

for(A i:s){cout<<i.x;}}

// o/p: 10 20

🡪Applications:

* Sorted stream of data: wherever we need sorted data we use set since time complexity is o(log n).
* Doubly ended priority queue: using normal priority queue we can either do max or min heap at a time. But using set we can do both functions on set. For min heap we can traverse from set begin and for max heap we can traverse form set end() -1. Both in o(1)

MULTISET:

#it is same as set but here we can add multiple instances of same element(duplicates). It is also implemented using red black trees.

🡪initialization and declaration:

* Multiset<int> ms;
* #all functions of set are there in multiset.
* Ms.erase(10); // deletes all instances of 10

Ms.erase(it);//deletes only the instance of 10 where pointer it points. Ms.erase(it,ms.end()); // deletes the range of values from it to end.

* Ms.count(10); // returns the no of instances of 10
* Ms.lower\_bound(10); //this returns pointer to first occurrence of 10 if found else returns pointer to next greater element and if searched element is largest then it returns end pointer.
* Ms.upper\_bound(10); //this returns pointer first occurrence of next greater element and if searched element is largest then it returns end pointer.
* Auto it=Ms.equal\_range(20); //this returns pair of pointers: lower bound and upper bound for range where instances of 20 are present

Cout<<(\*it).first<<(\*it).second; // first gives lower bound i.e first occurrence of 20 if found and second gives next greater element of 20.

MAP:

#it is same as set just here we have a key value pair instead of single set items. It is also implemented using red black trees. Use header # include<map>

🡪initialization and operations:

* Map<int,int> m;
* M.insert({10,2}); // inserts 10 if not present earlier else ignores it
* For(auto &i: m){cout<<i.first<<i.second;} // accessing key using first and value using second
* M[20]=13; // inserts 20 in map if it is not there. We can also access the value for key 20. Also we can modify or change the values using this syntax.
* M.at(20); //it doesn’t adds in map if the element is not present . it does range check and throws exception if we are accessing a key out of range from map.
* M.find(10);//returns pointer to element if found else returns pointer to end
* Auto it=M.lower\_bound(10);// (>=)returns pointer to element if found else pointer to next greater element(key).

if(it!=m.end()){cout<<(\*it).first<<(\*it).second;} // accessing key using first and value using second

* M.upper\_bound(10);//(>) returns pointer to next greater element(key).
* M.count(10);//returns 1 or 0 for instance of element present or not
* M.size(); //returns size
* M.empty(); // returns true if empty else false
* M.clear();//clears all elements
* M.erase(10); // erases key and value for key= 10

M.erase(it); //deletes the element pointed by pointer it

M.erase(m.find(10),m.end()); // deletes the range of values from start address (the element found) here to end address.

🡪Time complexities:

Begin,end,rbegin,rend,cbegin,cend,crbegin,crend,size,empty,clear - o(1)

erase(it) (amortized) -o(1)

Insert(),m[key],at(),count,find,erase(key), -o(log n)

🡪Applications:

* Sorted stream of data of items with (key ,value) pair
* Doubly priority queue of items with (key ,value) pair

UNORDERED SET:

#it is implemented using hashing data structure. It is similar to set but doesn’t have any order of storing the elements. Include header #include<unordered\_set>

🡪initialization and operations:

* Unordered\_set<int> s;
* S.insert(10); // inserts 10 if not present. It doesn’t allow duplicate entries

For(auto i:s){cout<<i;} // prints any random permutation of all values inserted in unordered set.it doesn’t have fixed order and depends on machines how they are stored.

* S.empty(); //returns true if empty
* S.clear(); //clears all elements
* Auto it=S.count(10); // returns 1 or 0 if element is there or not
* Auto it=S.find(10); //returns pointer to element searched
* S.erase(20);//deletes 20 if present

s.erase(it); //deletes element at address pointed by pointer it

s.erase(s.begin(),s.end()); //deletes range of values between start pointer address and end address

🡪 s.size(); returns no of elements

🡪Time complexities:

* Insert,erase(it),erase(val),find,count,size,empty -o(1) on average

🡪Applications:

We use them when we need following operations(or a subset of them): search,insert,delete to be done in constant time.

UNORDERED MAP:

#Same like maps but uses hashing. use header #include<unordered\_map>

🡪Application:

* Used to store key value pairs,
* uses hashing,
* no order of keys is required.

🡪initialization and operations:

* unordered\_map<string,int> mp;
* mp.insert({“ab”,10}); // inserts key,value pair if not present else ignores for duplicate
* mp[“ab”]=10; //inserts ab key with value 10 if not present
* mp.at(“ab”); //access key ab but does range checking
* for(auto i:mp){cout<<i.first<<i.second;} // prints any permutation of elements in map. No fixed order
* mp.erase(“ab”); //deletes key ab

mp.erase(it); //deletes value pointed by pointer it

mp.erase(mp.begin(),mp.end()); //erases range b/w start add and end add

* mp.size();//returns size
* mp.count(10);//returns 1 or 0 if element is found or not
* mp.find(10);//returns pointer to element if found else points end
* mp.empty();//returns true if empty

🡪time complexities:

Size,empty,Count,find,erase,insert,m[key],at() -o(1) on average.

Iterators:

🡪begin(),end(), // returns address of first and end+1

🡪prev()-eg: auto i=v.begin();//first point I to begin then-> i=prev(i);//moves and points i to prev position and returns its address;//eg: i=prev(i,3 );//moves and points i to 3rd previous position

🡪next()-eg:i=next(i);//moves and points i to next position and returns its address;//eg: i=next(i,3 );//moves and points i to 3rd next position,

🡪advance() - eg: advance(i,3); // just moves ahead and points I to next 3rd position but doesn’t returns address.

🡪auto it=find(start,end,val); // returns pointer to element if found else points end. // it has time complexity -o(n) since it increments the value from start add one by one and then compares it with val. // for sets,maps,unorderedsets,unorderedmaps use their own find functions as they have low time complexities than this general find function.

Algorithms:

🡪finding elements: for all these it is necessary to have sorted data

1)binary\_search(start,end,key); //checks if key element found or not; //time complexity is o(log n) for random access containers and o(n) for non random access containers.

🡪Binary\_search on User defined datatype :

Struct point{int x,inty;point(int x1,int y1){x=x1;y=y1;}};

Bool mycmp(point p1,point p2){return p1.x< p2.x;}

Int main(){vector<point> v={{10,10},{2,20},{50,100}};

Sort(v.begin(),v.end(),mycmp);

Point p(2,20);//creating an object p of point datatype

If(binary\_search(v.begin(),v.end(),p,mycmp)//finding object p using binary search

Cout<<”found”;

Else cout<<”not found”;

}

//o/p: found. //here since comparison is done for only x values of point data type so even if we pass a value like (2,22) which is not there in vector we will get true since comparison is done on x value and there is an element with x value 2 in vector.

Binary\_search(v.begin(),v.end(),p,mycmp)

2)lower\_bound(start,end,key); //gives first occurrence of an element >= key element; this is implemented using advance(it,val) function so has time complexity of o(log n). good for random access containers like arrays,vectors.etc but not good for non random access containers like sets,maps,unorderedsets,unorderedmaps.etc for such it is good to use their own lower\_bound function. Using the general lower bound function will call advance() which will take more time for sets,maps.etc.

3)upper\_bound(start,end,key); //gives first occurrence of an element strictly >key element; this is implemented using advance(it,val) function so has time complexity of o(log n). good for random access containers like arrays,vectors.etc but not good for non random access containers like sets,maps,unorderedsets,unorderedmaps.etc for such it is good to use their own upper\_bound function. Using the general lower bound function will call advance() which will take more time for sets,maps.etc.

🡪count(v.begin(),v.end(),10);//returns the counts of no of occurrences of 10. good for random access containers like arrays,vectors.etc but not good for non random access containers like sets,maps,unorderedsets,unorderedmaps.etc for such it is good to use their own count() function. Using the general count() function will take more time for sets,maps.etc.

🡪 rotate(start,mid,end); // rotates the container in clockwise . here mid becomes first element. time complexity is o(n)

eg: int arr[]={1,2,3,4,5}; rotate(arr,arr+2,arr+n); // gives 3,4,5,1,2

🡪reverse(start,mid,end); // reverses the container

Eg: int arr[] ={1,2,3,4,5}; reverse(arr,arr+3); // gives 4,3,2,1,5

🡪is\_permutation(v1.begin(),v1.end(),v2.begin());// compares v1 and v2 and counts the occurrences of each item and checks if same then it is a permutation and returns true(1). Else returns false(0). It increments v2 one by one and checks it with v1 from start to end.

🡪next\_permutation(start,end);//gives next permutation which is greater than the current number.

eg: int arr[]={1,2,3}; next\_permutation(arr,arr+n); //gives next permutation like 1,3,2;

🡪swap(a,b);

🡪min(a,b);

🡪max(a,b);

🡪max\_element(v.begin(),v.end());//returns pointer to max element in range given. It has time complexity – o(n)

🡪min\_element(v.begin(),v.end());//returns pointer to min element in range given. It has time complexity – o(n)

# for user defined datatypes using min\_element and max\_element

Struct A{int x,y; A(int x1,int y1){x=x1;y=y1} }; bool mycmp(A obj1,A obj2){return obj1.x<obj2.x;}

Int main(){vector<A> v={{4,40},{1,10},{2,20},{3,30}}; auto it=max\_element(v.begin(),v.end(),mycmp; auto it=min\_element(v.begin(),v.end(),mycmp;}

//o/p gives max as 4,40 and min as 1,10

🡪fill(v.begin()+1,v.end()-2,10); // fills the vector (form given start add to end add-1)with 10 as element. Time complexity-o(n);

🡪 accumulate(v.begin(),v.end(),initial\_result);//returns sum of all elements from start add and end add-1.

eg: accumulate() function: //#include <algorithms>

Int main(){

Vector<int> v{10,3,20,30};

Int res1,res2;

Res1=accumulate(v.begin(),v.end(),0); // accumulate() function by default calculate the sum of all elements with initial sum as given in parameter(here 0); It can also do other arithmetic functionality operations. It takes a container’s start ,end and initial sum and work to be done

Res2=accumulate(v.begin(),v.end(),1,[](int x,int y){return x\*y;}); // here accumulates does multiplication of the x with 1 (and y is the elements in the container one by one) and pass its result as parameter x and repeats for all elements. And returns final result to res.

}

User defined task and comparison:

Int mycmp(int x,int y){return (x\*y);}

accumulate(v.begin(),v.end(),initial\_result,mycmp); //returns the results of mycmp fun (here multipled value).

🡪rand(); // use header<cstdlib> . it generates a random number in range 0 to 32767 (0 to RAND\_MAX). same finite random numbers are generated since seed is not set.

🡪srand(); //use header <cstdlib> and <ctime>. Different seed is used to generate different random numbers. It uses time function to set seed. Eg: srand(time(NULL));

Synax : 1)srand(unsigned int);//it takes seed as unsigned int.

(it is based on linear congruential generator : xn+1 = (axn +b)%m here we generate next random number using previous random no. a,b are machine dependent constants and m is RAND\_MAX +1 .)

1. time\_t time(time\_t \*t);// it sets current time as seed if NULL is passed as parameter.

Else sets the time according to pointer t passed. It returns an integral value.

# if we want to generate random numbers in range we do :

srand(time(null));rand()%100; //(will generate in range 0-99)

# if we want to generate random numbers in range from high to low we do :

srand(time(NULL)); int high=100,int low=10; int range=high-low+1; rand()%range; //(will generate in range from high to low)

🡪sort(v.begin(),v.end(),mycmp); // time complexity -o(nlogn) uses intro sort(combination ofquick ,heap, insertion sort).

🡪make\_heap();

OOPS:

🡪classes and objects,encapsulation,abstraction,inheritance,polymorphism.

🡪ways to initialize members object:

* normal constructors:

class A{int x,y;

A(int x1,int x2){x=x1;y=x2;}};

* initializer list:

Class A{ int x;int y;

A():x(10),y(20){}

};

🡪advantage of initializer list:

Performance advantage because members are initialized only once.

//without initializer list: o/p: default parameterized

Class A{ A(){cout<<”default”;}

A(int x){cout<<”parameterized”;}};

Class B{A a;

B(){ a=A(10);} //firstly default constructor is called while member creation and then that member is initialized with parameterized constructor.

};

Int main(){

B b;}

// with initializer list: o/p: parameterized

Class A{ A(){cout<<”default”;}

A(int x){cout<<”parameterized”;}};

Class B{A a;

B():a(10){ } //here no need to call the default constructor of class A to initialize it with random values, we can directly initialize it with parameterized constructor.

};

Int main(){

B b;}

🡪constructors:

* Default
* Parameterized
* Copy

🡪copy constructors:

# whenever there is pointers in class and there is dynamic memory allocation use our own defined copy constructor to make deep copy rather than compiler generated default shallow copy.

1)Class Test{

Int\* ptr;

Public: Test(int x){ptr = new int(x);}

Void set(int x1){\*ptr=x1;}

Void get(){cout<<\*ptr;}

};

Int main(){

Test t1(10);

Test t2(t1); //same as Test t2=t1;

//but different from

Test t2;

T2=10;//this is not copy constructor ,instead this is first declaration and then assignment.

t2.set(20);

t1.print();t2.print()}

//o/p: 20 20 // here we are not using a user defined copy constructor but compiler generated default copy constructor which assigns member of one class to other one by one. Here a shallow copy is done meaning that both the pointers are pointing to same memory location. Thus changing the values at that location is reflected in both t1,t2.

2) Class Test{

Int\* ptr;

Public: Test(int x){ptr = new int(x);}

Test(const Test& t){int val=\*(t.ptr); // here we are defining our own copy constructor, firstly reading the value from passed parameter of type Test class and then pointing ptr to new dynamic created memory and assigning it with the value read. This is deep copy.

Ptr=new int(val);}

Void set(int x1){\*ptr=x1;}

Void get(){cout<<\*ptr;}

};

Int main(){

Test t1(10);

Test t2(t1);

t2.set(20);

t1.print();t2.print()}

//o/p: 10 20 // only t2 is changed

🡪DESTRUCTORS:

#Destructors are called in reverse order of creation for objects in same scope. Destructor is called when object goes out of scope. Like constructors there Is default destructor called. When we are allocating dynamic memory using new keyword we need to call the destructor using delete keyword else memory will not be free.

Class A{

Int x;

Public:

A(int x1):x(x1){Cout<<x<<” constructor”<<endl;}

~A(){cout<<x<<” destructor”<<endl;}

};

Int main(){

A a1(10);

A a2(20);}

//o/p: 10 constructor

20 constructor

20 destructor

10 destructor

🡪this keyword

# it is a constant pointer that points to object and access it members without name collision using this->memvalue. It cal be used in chaining of functions. Changing this pointer to point other object can give compiler error

Changing of function:

1)Int a=10,b=20;Cout<<a<<b; // here << operator is a function call and we can chain it. Here firstly cout a is evaluated and then it returns a reference of ostream type and next call is done to print b and so on.

2)Class A{

Int x,y;

Public:

A(int x,int y){

This->x=x;

This->y=y;}

A &setx(int x){ // return by reference so that we get same object.

This->x=x;

Return \*this} // returns an object of type A

A &sety(int y){

This->y=y;

Return \*this}}; // returns an object of type A

Int main(){

A a(10,20);

a.setx(11).sety(21);} //Chaining of functions. Here firstly a.setx is evaluated and the result is returned as same object and then sety is called on same object.

🡪STATIC MEMBERS:

# static data members are created once for a class and are shared among all the objects for that class.

#static functions can access only static data members and not others. whereas normal functions can call static data.

Class A{

Static Int count;

Public: A(){count++}

~A(){count--}

Static int getcount(){return count;}

};

Int A::count=0;

Int main(){ A a1;

Cout<<A::count; // static data members called using class name

{,A a2;} // a2 gets created=> count=2; a2 gets destructed outside the scope+>count=1;

Cout<<A::getcount;

}

//o/p:1 1

🡪INHERITANCE

* # private members of base or parent class are not accessed even by child class. Only the protected and private members can be accessed by child class.
* Access rules:

1. Private- protected and public of base class are inherited as private in derived class.
2. Public- protected and public of base class are inherited and remains same in derived class
3. Protected- protected and public of base class are inherited as protected in derived class

* # whenever a derived class object is created,it will call the base class constructor, if there is no explicit base class constructor call in the derived class then the default constructor of the base class is called,if there is no default constructor (suppose we have declared a parameterized constructor and removed the default one)in base class then it will give compiler error. To solve this we then need to call the base class constructor explicitly form the derived class using initializer list.
* MULTIPLE INHERITANCE:

# when there is multiple inheritance like a is base and b,c are derived classes and d is derived form b and c . suppose class a has member x. Here when we call the member x in derived class d it will give compiler error since it has two copies of x (from class a,b). Also when an object is created for class d ,on printing cout statements for x in constructor of class d will show that class a constructor is called 2 times( form path a-b and b-c).This problem is called DIAMOND PROBLEM.

Class a{

public:int x;};

Class b:public a{};

Class c:public a{};

Class d:public b,public a{}; //#here firstly constructor of base classes are called in same order of their inheritance. Here firstly constructor of b and then a is called.

Int main(){

D d1;

Cout<<d1.x;} // ambiguous call to x(since there are 2 copies form two paths)

# TO SOLVE THIS WE USE VIRTUAL KEY WORD:

Class a{

public:int x;};

Class b: virtual public a{};

Class c: virtual public a{};

Class d:public a,public b{}; //#here firstly constructor of base classes are called in same order of their inheritance. Here firstly constructor of b and then a is called.

Int main(){

D d1;

Cout<<d1.x;}

🡪VIRTUAL FUNCTIONS

#These are used to implement runtime polymorphism, when a base class pointer or reference is created and is pointing to derived class object.

# implementation of virtual functions: VPTR (points to vtable)for every object and VTABLE(stores addresses of all virtual functions of the class) for every class are used. Every time an object is created compiler adds some code in the constructor and sets this vptr to the vtable of this class and when overridden functions are called then Compiler goes to vptr and gets address of appropriate function from the vtable and call the appropriate function..

# Thus virtual functions does some extra task by performing extra cpu cycles for vtable and vptr but are useful.

# if virtual keyword is used for a function then it cannot have static keyword.

# virtual function cannot be a friend function of another class.

# a class cannot have a virtual constructor but can have a virtual destructor.

#VIRTUAL DESTRUCTOR:1) if we delete child class object through a pointer of parent class then it is undefined behaviour. If parent class doesn’t have virtual destructor. 2)If we fail to declare destructor as virtual in parent class then we end up having memory leak.

Eg: class base{ public: base(){cout<<”base constructor”;}

~base(){cout<<”base destructor”;}};

Class der:public base{public: der({cout<<”der constructor”;}

~der(){cout<<”der destructor”;}};

Int main(){

Base \*b =new der();

delete b;}

//o/p: base constructor der constructor base constructor // here since the compiler doesn’t know what type of object pointer b is holding therefore it doesn’t call the destructor of der classs so it is memory leak since not deleted. To solve this use virtual destructor in base class.

# virtual keyword should be used only with the (polymorphic) functions base class or higher order class in the hierarchy. Using it with the derived class functions (polymorphic) is optional.

# when all the functions in a class are pure virtual functions(having no body eg: virtual void fun()=0;) then it behaves like an abstract class thus its object cannot be created. Also when some other class inherits this class it needs to have the body for this class else the derived class objects cannot be created.

Class base{public :

Void print(){cout<<”base”;}};

Class der:public base{

Public:

Void print(){cout<<”derived”;};

Int main(){

Base b1;

Der d1;

B1.print();

D1.print();

Base \*ptr=&d1;

Ptr->print();} //here ideally it should call the object assigned print function but it calls print function of the pointer datatype.

//o/p: base derived base

# to avoid this we use virtual keyword in base class functions which are identical to derived class functions and there is situation like here.

Class base{public :

Virtual Void print(){cout<<”base”;}};

Class der:public base{

Public:

Void print(){cout<<”derived”;};

Int main(){

Base b1;

Der d1;

B1.print();

D1.print();

Base \*ptr=&d1;

Ptr->print();} //now it calls derived class object function print.

//o/p: base derived derived

🡪OVERRIDE KEYWORD: (introduced from c++11)used to make code more readable and inform that it is overriding some base class virtual functions also gives error when there is spelling mistake in function name or when it is used with a function which is not overriding a virtual function of base class.

Class b{

Virtual Void print(){}};

Class d{

Void print() override{}};

Int main(){

B \*ptr=new d();

Ptr->print();}

🡪OPERATOR OVERLOADING:

* class complex{

Private: int real,img;

Public:

complex operator+ (const complex &c){ complex res;

res.real = real + c.real;

res.img = img + c.img;

return res;}};

Int main(){

Complex c1,c2,c3;

C3=c1+c2; // here it means c1.operator+ (c2); i.e object c1’s operator function is called and parameter is c2.

}

# use:to avoid less readability of code ,like + does addition.

# precedence ,associativity, arity(arity:no of operands it take): same as + or other arithmetic symbols

#we cannot overload . , ::, ?: , sizeof

#Assignment operator: when there is a dynamic memory allocation in class and we are using assignment operator then it will call the default assignment overloaded function and will do assign work and will create shallow copy. To avoid this we need to define our own assignment operator code.

* Eg: Test t1,t2

t1=t2; // assignment operator default will create shallow copy thus on changing t1 will change t2 also.

🡪FRIEND CLASS AND FUNCTIONS:

# friendship is granted i.e it is declared by the class who is his friend. It is non-mutual friendship i.e if a is friend of b then b may or may not be friend of a, also there is no transitivity relation,also on inheritance there is no automatic friendship it needs to be declared.

* Friend function:

Class a;

Class b{

public:

void funb(){}};

Class a{ friend void b::funb();} // declaring that funb of class b is granted access to private and protected members of class a.

Int main(){

A a1;

B b1;

b.funb(); }

* 🡪 Friend function:

Class a;

Class b{

public:

void funb(){}};

Class a{ friend class b;} // declaring that all functions and members of class b are granted access to private and protected members of class a.

Int main(){

A a1;

B b1;

b.funb(); }

🡪ERRORS in c++:

1. Syntax error: spelling errors
2. Semantic error : when statement doesn’t make sense //eg: 16 = x; or x+y =2; //there should be a storage location of lvalue but here it is not there so semantic error.
3. Linker error: if function called declaration is not present //eg we declare the function but do not provide the body for that so compiler gives linking or Ld error
4. Runtime error: errors caused during program run like memory leaks //eg: There Is uninitialized pointer and we are trying to access the memory pointed.so may give segmentation faults.etc
5. Logical error: wrong outputs //eg using = in place of == for comparison

🡪 EXCEPTION HANDLING:

# try,throw,catch

* Int fun(int arr[],int n) throw(string),throw(int){// here we write throw to specify that this fun can throw some exceptions of string , int.so it is recommended to check them. Writing throw with function declaration is optional for functions but recommended way.

If n==0

throw string(“array is not of valid size”);

/\*

If n==0

throw 5; // throw int

If n==0

throw 10.1000 // throw double

\*/

Int sum=0;

for(int i=0;i<n;i++)

Sum+=arr[i];

Return sum/n;

}

Int main(){

Int arr[]={1,2,3};

Int n=0;

try{

int res=fun(arr,n);

cout<<”successful”;

}

Catch(string &s){

Cout<<”string”;

}

/\*

Catch(int &e){

Cout<<”int”;

}

Catch(…){ //three dots are used to catch all other datatypes which are not having catch block defined here.

Cout<<”other datatypes which are not having catch block defined here, error caught”;

}

\*/

}

* Int a()throw (int){

Cout<<”a starts”;

throw 100;

Cout<<”a ends”;

}

Int b() throw (int){

Cout<<”b starts”;

a();

Cout<<”b ends”;

}

Int c() throw (int){

Cout<<”c starts”;

b();

Cout<<”c ends”;

/\* here also catch block can be written

Catch(int e){cout<<”exception caught”;}

\*/

}

Int main(){

try{ c();}

Catch(int e){cout<<”exception caught”;}

}

# **stack unwinding in exception handling:** for every call the exception is throwed and then looked for the catch function, if found then catch part is executed and activation record for that function is deleted from stack Else the catch part is looked upon the caller function and the activation record for the function is deleted form the stack.

#Here firstly c is called it then calls b and b calls a where int exception is found and returned to the caller function i.e to b but there is no handler code in b,so it checks in the caller function of b i.e c and repeats the process and then finally checks in the main function.

# if there is no catch block found then it will show error and will crash. There has to be atleast one catch block either in the function or in the caller function or in main function.

🡪USER DEFINED EXCEPTION:

# create user classes for exceptions and throw error

Class Arrayzerosize{};

Class Arraynegativesize{};

Int avg(int \* arr,int n){

If n==0

throw Arrayzerosize(); //we throw an object of Arrayzerosize class

If n<0

throw Arraynegativesize(); //we throw an object of Arraynegativesize class

int sum=0; for(int i=0;i<n;i++){sum+=arr[i];} cout<<sum/n;

}

Int main(){

int n;

cin>>n;

int \*arr = new int[n];

try{

int res=avg(arr,n);

cout<<res;

}

catch(Arrayzerosize &e1){

cout<<”size is zero”;

}

catch(Arraynegativesize &e2){

cout<<”size is negative”;

}

}

# It is recommended in c++ to make non-primitive or user defined exception classes.There is also a primitive exception class .All standard library exceptions like bad\_alloc,bad\_cast. etc inherit from exception class directly or indirectly. In java there is no privilege to make non-primitive exception classes.

#use header file #include<exception>

Class myexception: public exception{

// myexception- userdefined inherits from primitive -exception class

Public:

Virtual const char\* what() const throw(){

// what is a virual function of exception class and we can override it. It returns a string literal and this function does not throw any exception inside it therefore const throw() is written. Here we have defined our own what function.

Return ”exception occurred”;}

};

Int main(){

try{

throw myexception(); //we throw an object of myexception class

}

Catch(exception &e){ // we can write or declare here- myexception &e also , but since our myexception class is inherited from the exception class we can use the concept of virtual function where a base class is referred to derived class object we can call the derived class function using virtual functions. here we call the what function of the user defined or derived class object. This type of declaration can be used to catch any type of exception with single catch block when the user defined class is inherited from exception class.

Cout<<e.what();}

}