SEE “C NOTES” BEFORE CHECKING THIS FOR BETTER UNDERSTANDING

(note: always cut copy the code in vs code to understand it and not read it in word file )

1. Multi line comment

Use /\*

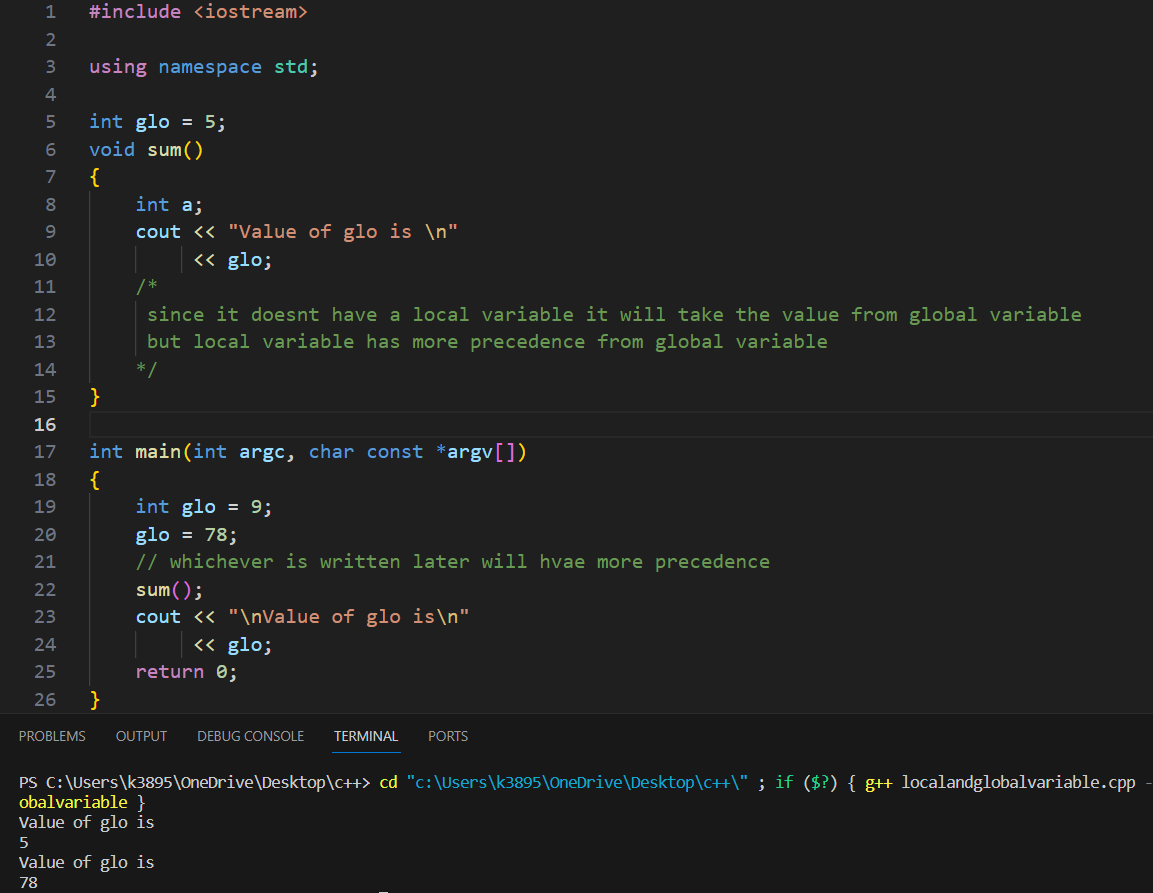
\*/

1. To write a statement we use “ ” but for simply printing a variable

We can do it without putting “ ”



1. Local and global variable



1. Boolean data type

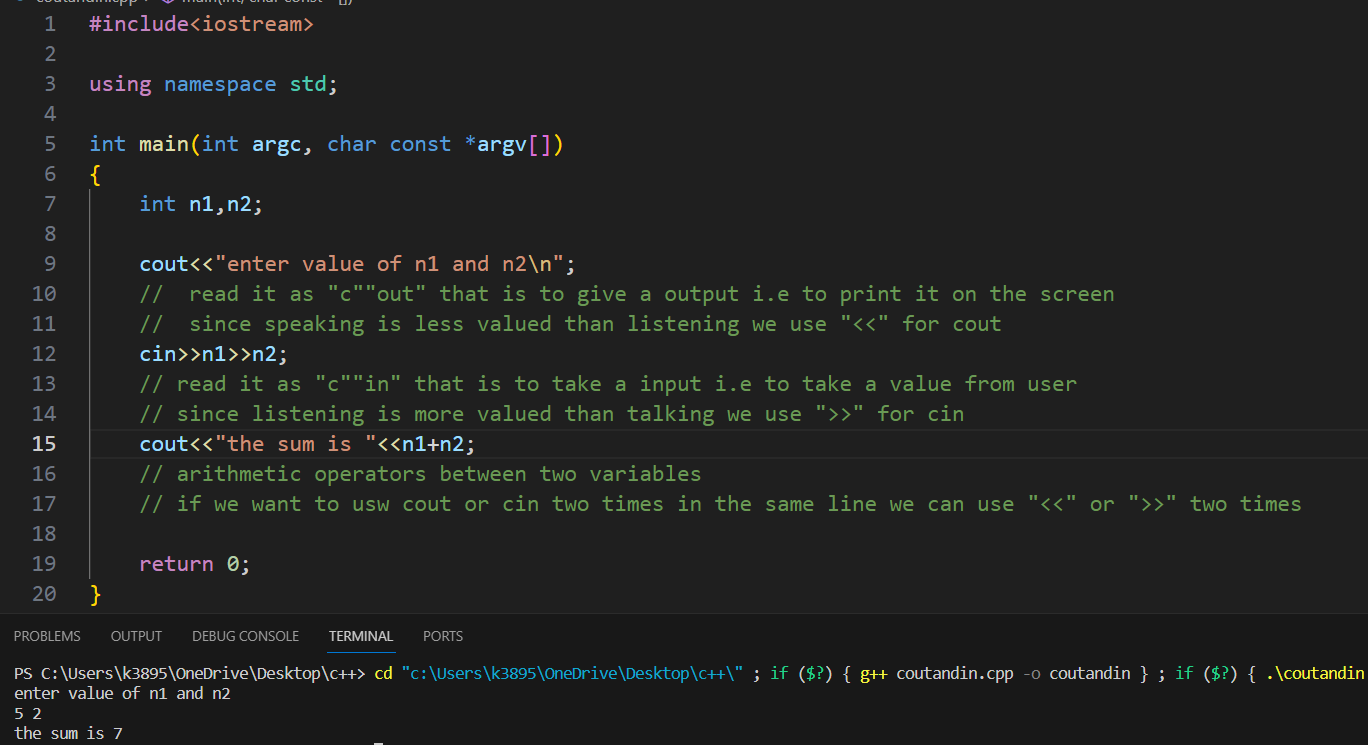


Since, “h” is true printing it will give “1”

And

“k” is false printing it will give “0”

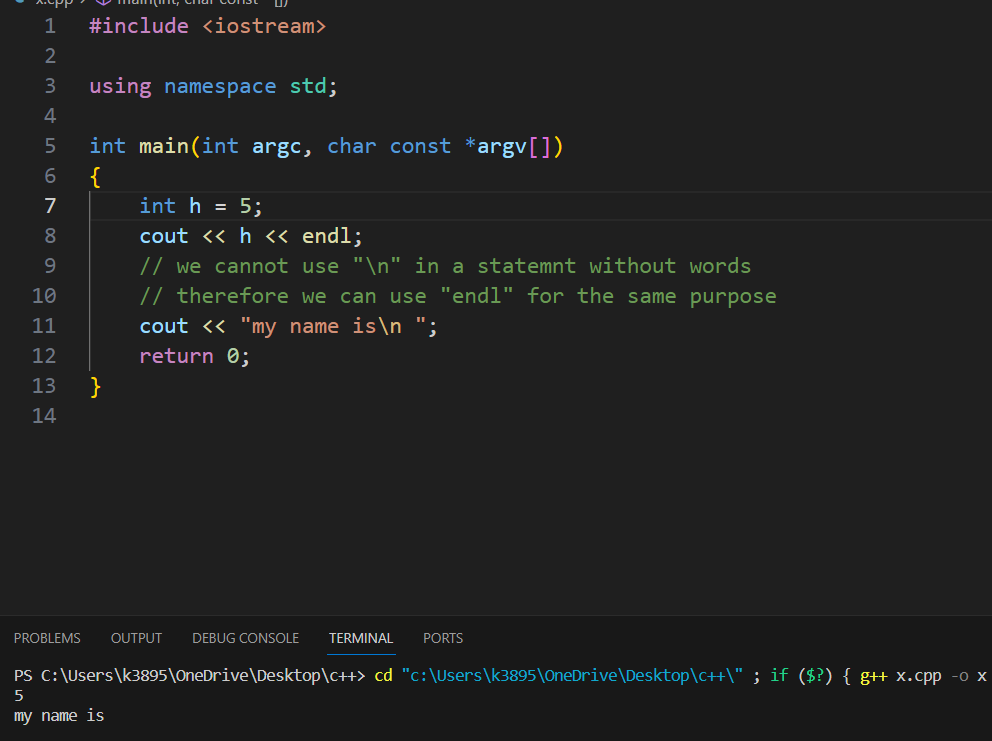
1. COUT AND CIN



(Cpp reference for telling which header files do what)

1. MANIPULATORS

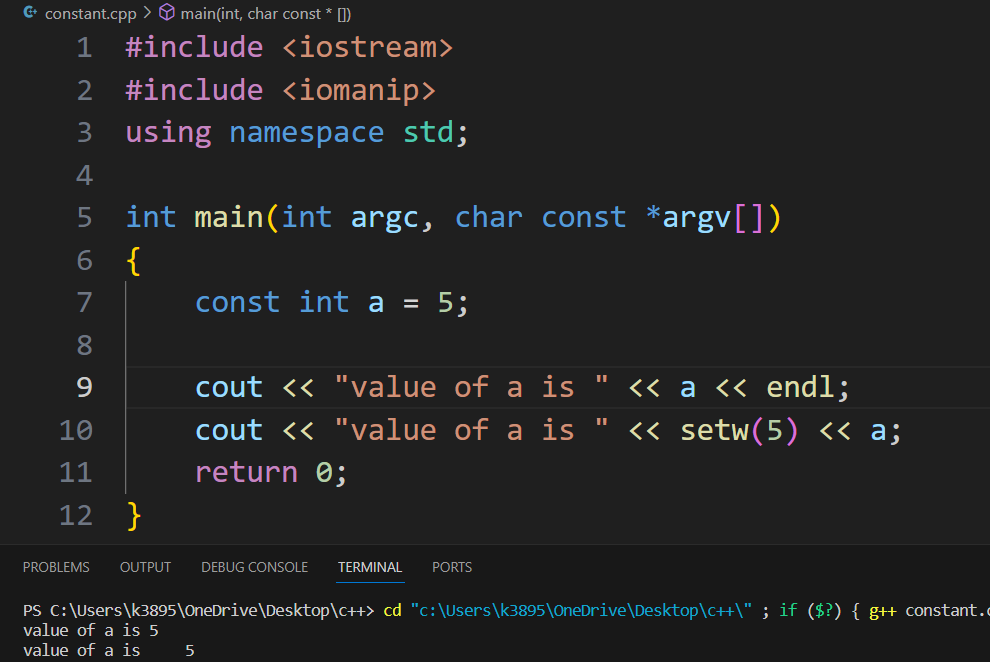
ENDL or \n



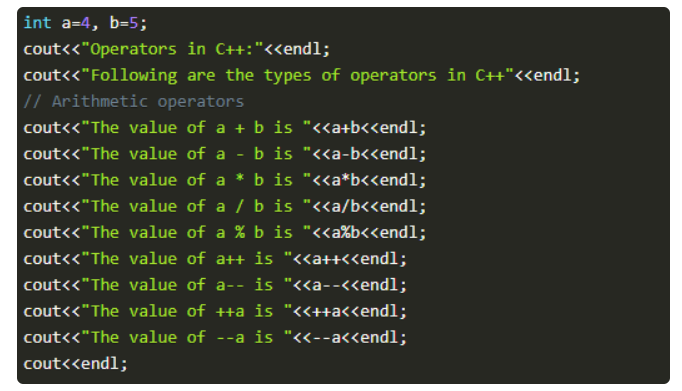
SETW (no. of space to be given) or \t

To use “setw” we need to add a header file <iomanip>

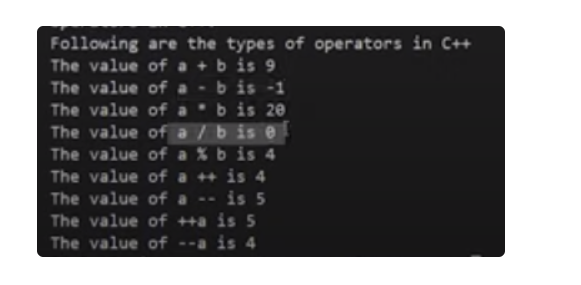
Whatever is written after setw(space to be given) will get that space



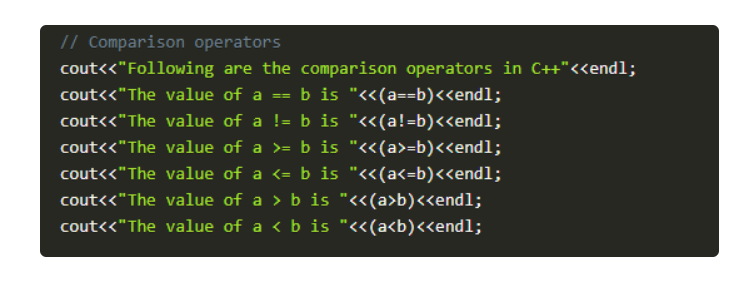
1. OPERATORS (a=4,b=5)

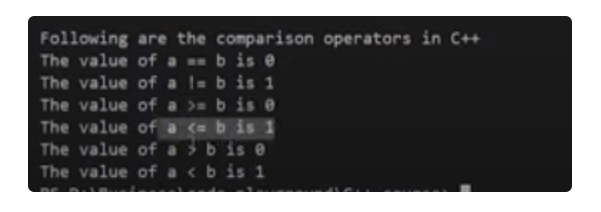


1. The function "**a++"** will first print the value of a and then increment it by 1(not print it just increment).
2. Similarly for a--
3. The function "**++a"**, will first increment it by one and then print its value.(pehle increment kara a ki value ko then print kara
4. Similar for --a

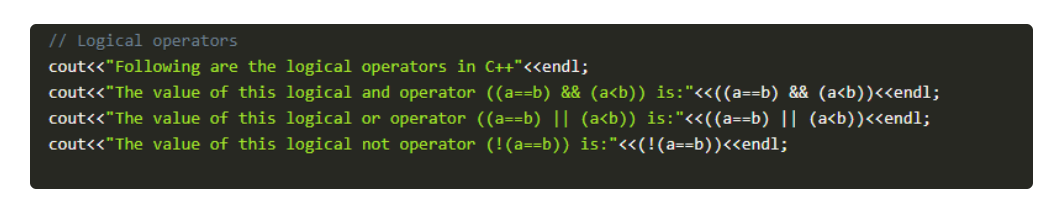


This is because operations between integers will only return integers and not float,thus the float part(decimal part) is removed and only int part is presented.

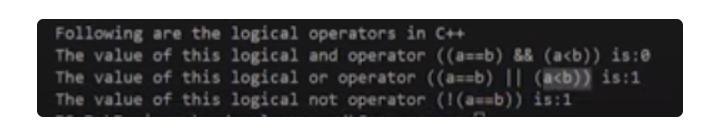
IT ONLY ANSWERS IN 0 AND 1 (FALSE AND TRUE)



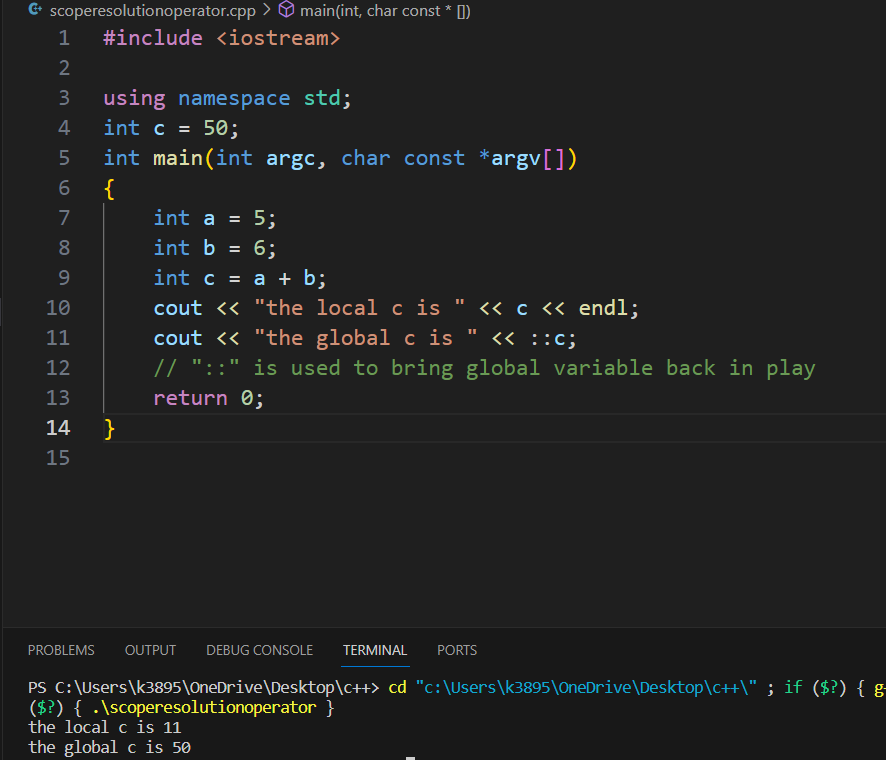
AGAIN, IT ONLY ANSWERS IN 0 AND 1 (FALSE AND TRUE)



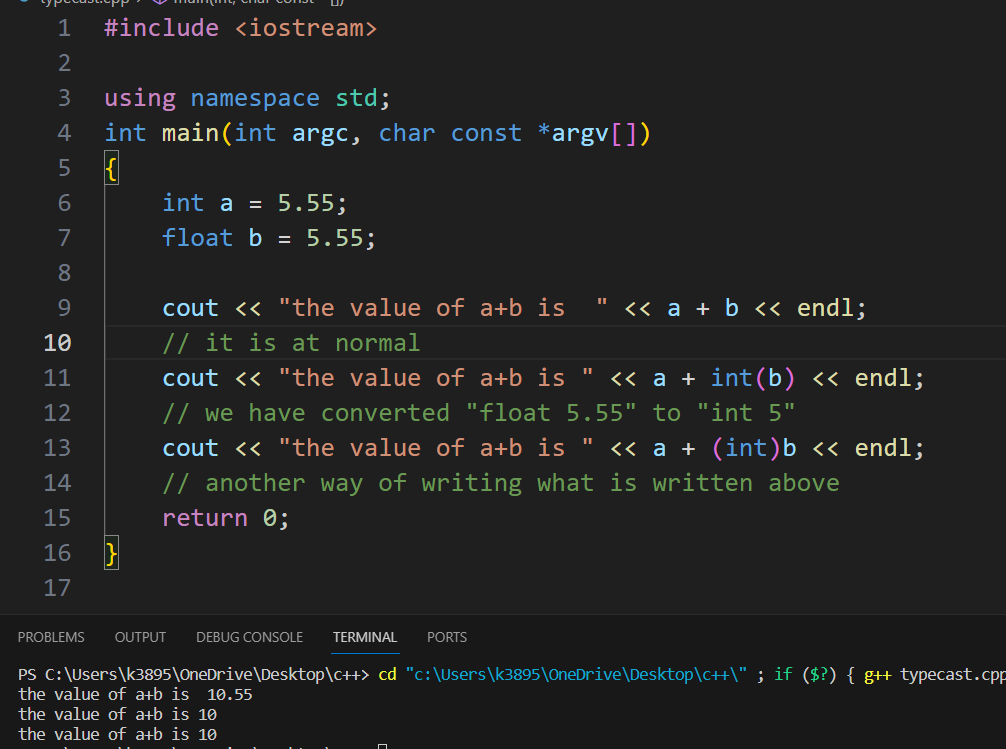
1. The function "(**(a==b)&& (a<b))"** will first compare a and b values and check if they are equal or not; if they are equal, the next expression will check whether "a" is smaller than "b". The output will be one if both expressions are true and 0 if not.
2. The function "(**(a==b) || (a<b))"**, will first compare a and b values and check if they are equal or not, even if they are not equal it will still check the next expression ie whether "a" is smaller than "b" or not. The output will be one if any one of the expressions is true and 0 if both are false.
3. The function "(!**(a==b))"**, will first compare a and b values and check if they are equal or not. The output will be inversed ie if "a" and "b" are equal; the output will be 0 and 1 otherwise.

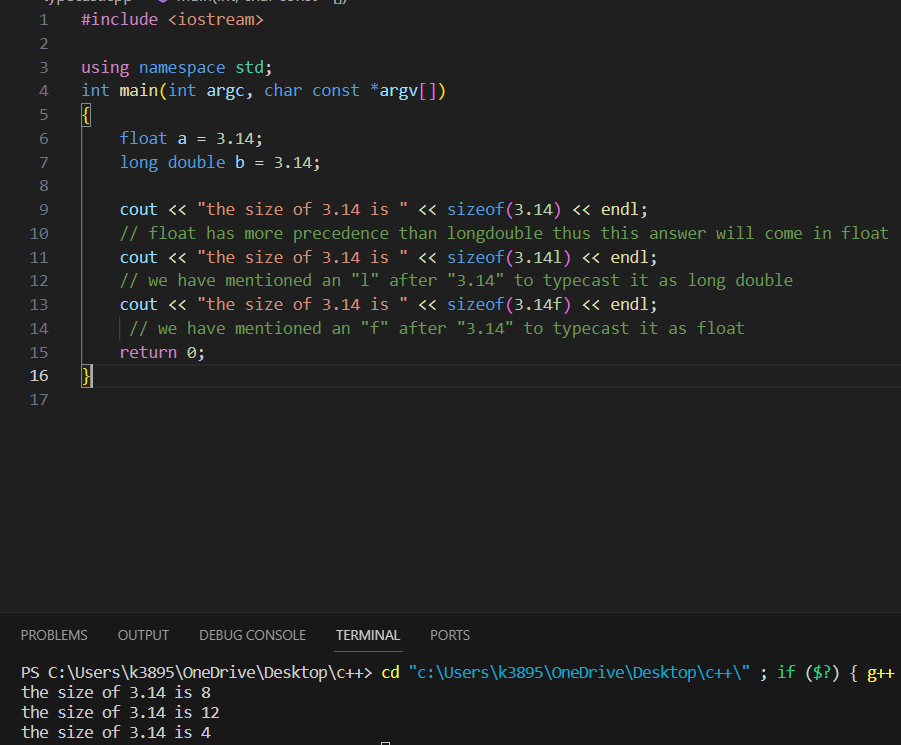


1. Scope resolution operator

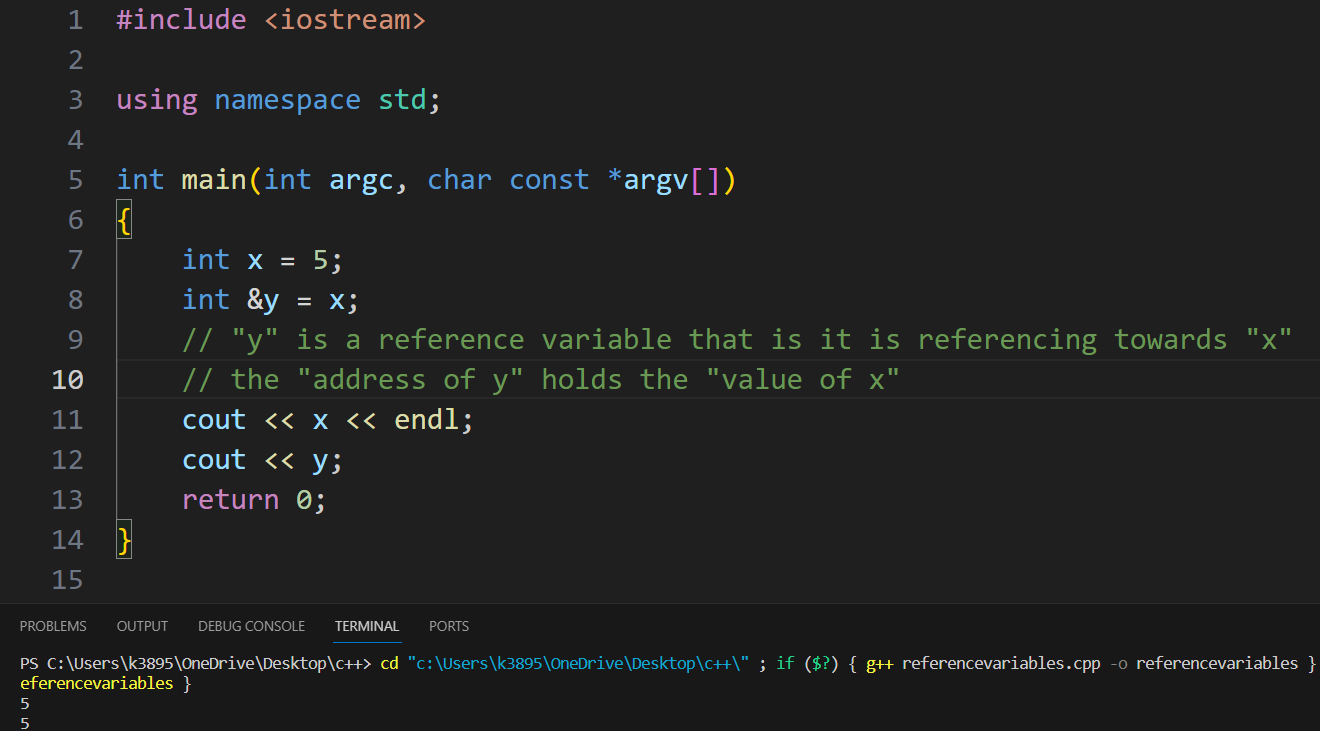


1. TYPECASTING



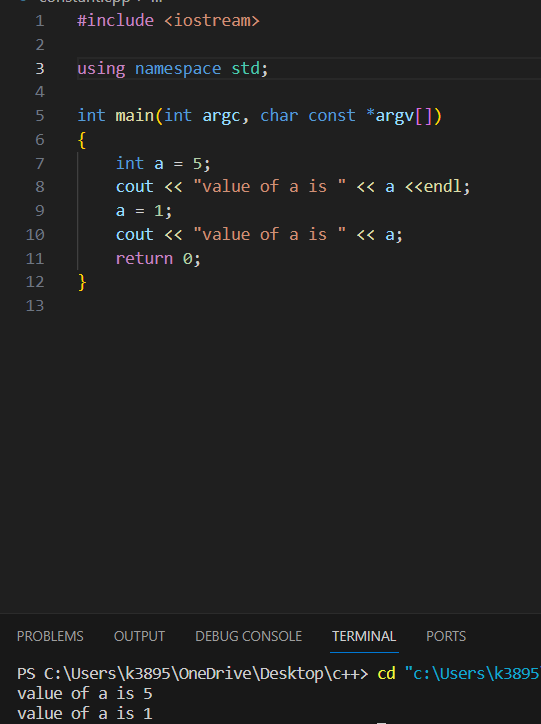
this is not exactly typecasting but a kind of it

1. REFERENCING VARIABLE



1. CONSTANTS

Usually,



But after applying constant (const)



1. CONTROL STRUCTURES

(basically if-else statements,loops etc.)

It is of 3 type

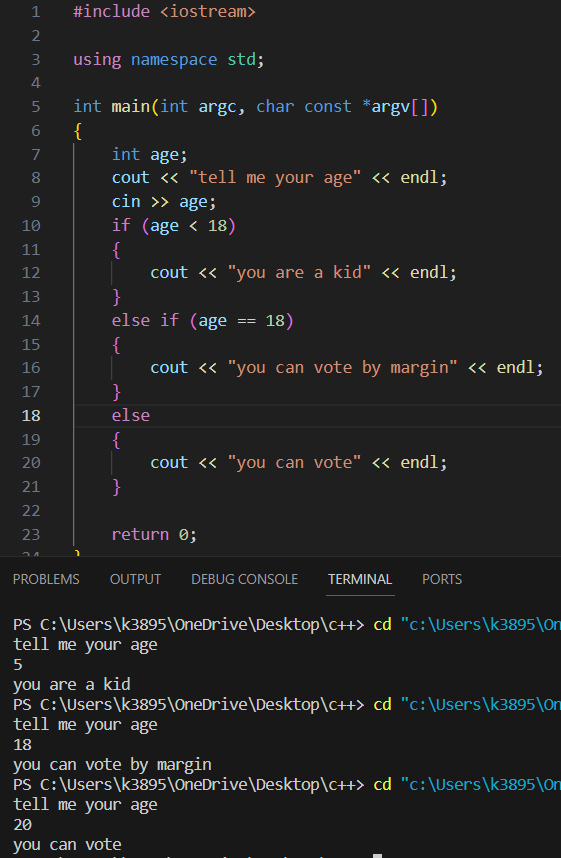
* SEQUENCE STRUCTURE

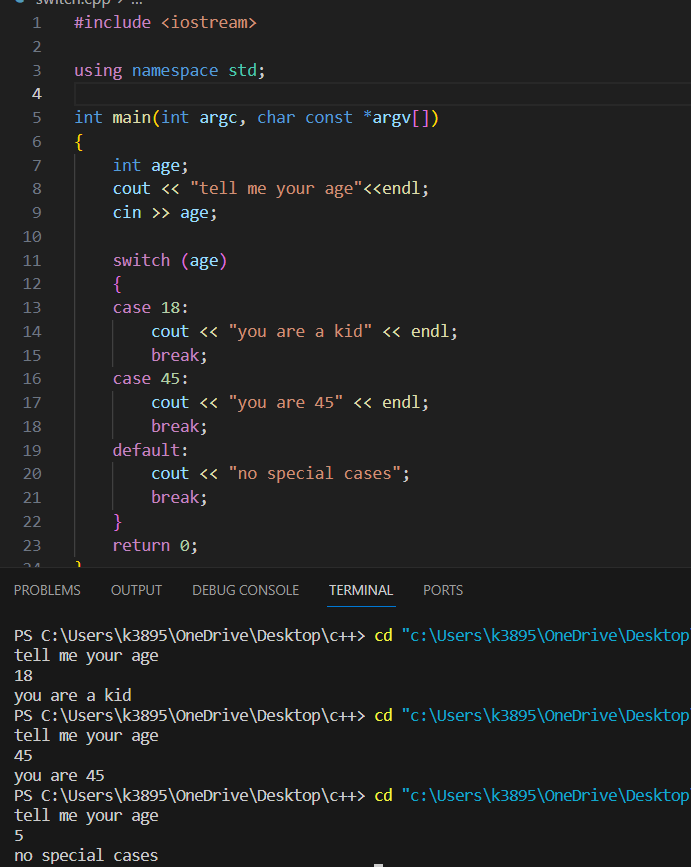
It refers to the sequence in which program execute instructions one after another.(without checking any condition)

* SELECTION STRUCTURE

It refers to the execution of code according to the selected condition (true or false)

Eg. IF ELSE statements and SWITCH statements



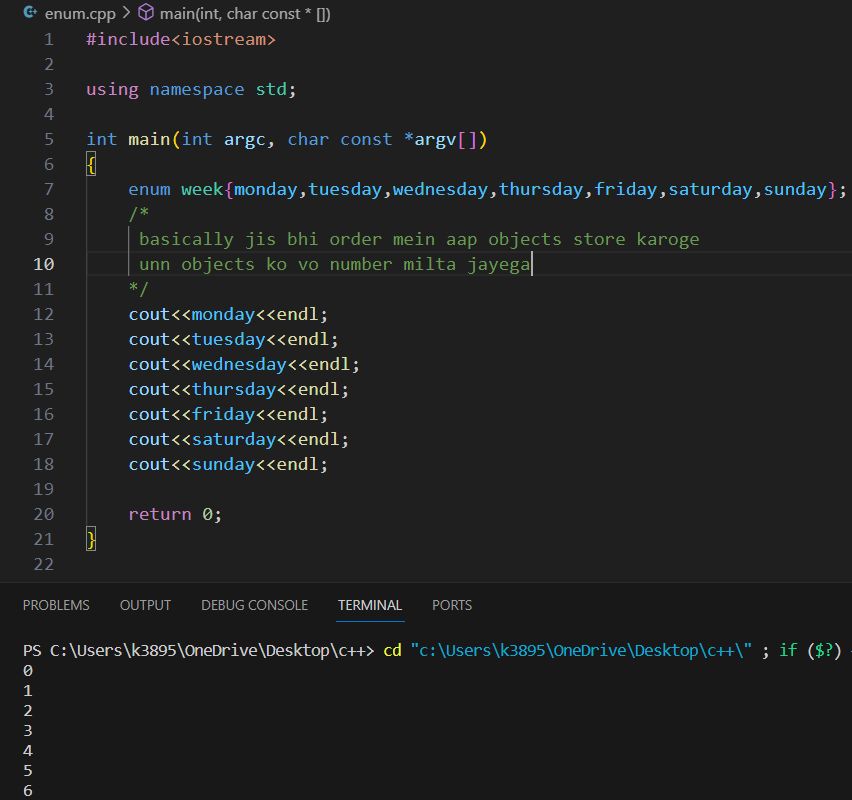


* LOOP STRUCTURE

It refers to the execution of an instruction until the condition gets false

Eg. FOR,WHILE,DO WHILE loops

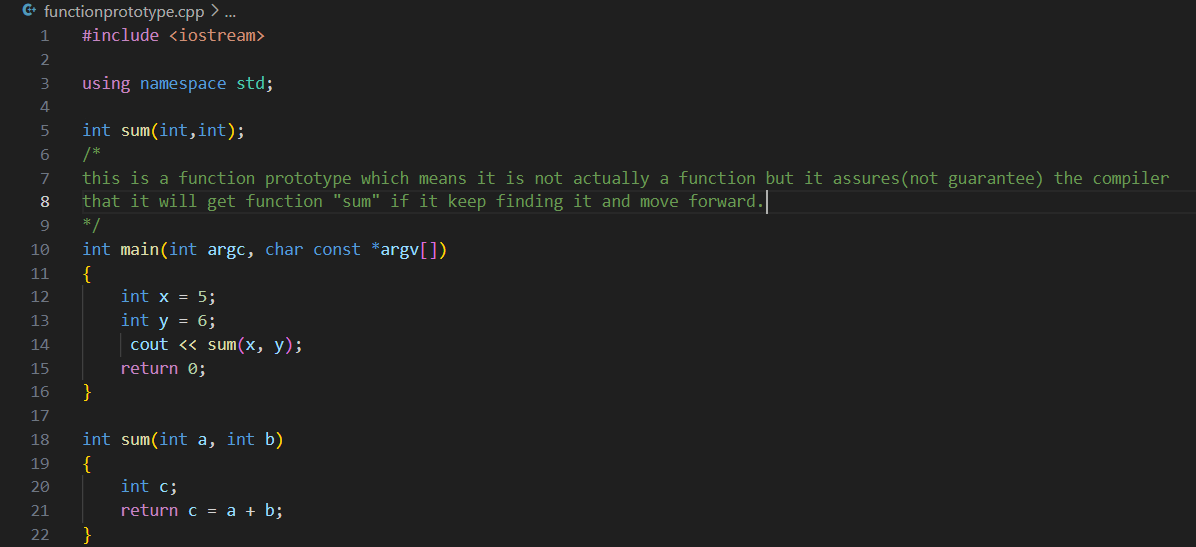
1. ENUM

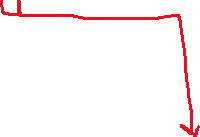


1. FUNCTION PROTOTYPE

It makes the “main” block of code work even when the “desired function” is present after the “main” function as it assures the main function that the “desired function” will be present somewhere

Desired function = sum (in this case)



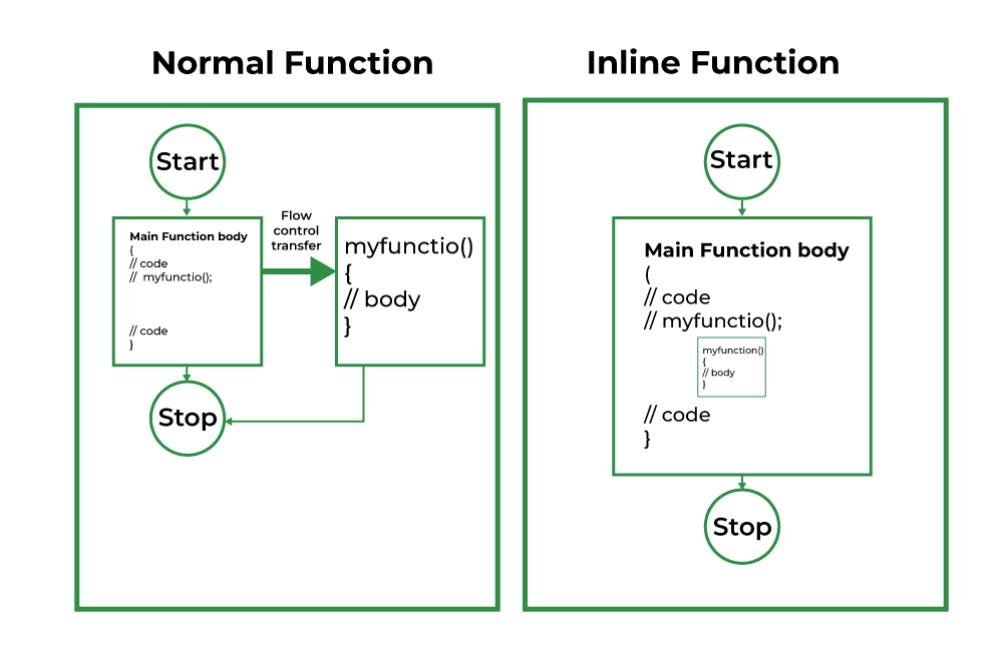


Formal parameters Actual parameter\

1. INLINE FUNCTIONS

In C++, an inline function is a function that is expanded in-line by the compiler at the point of call, rather than being called as a separate function. This means that the compiler replaces the function call with the actual function body

NOTE:**Avoid using inline functions for recursive functions**



1. CLASSES

Classes are extensions of Structures

Structures can have more than 1 data type

Similarly, Classes have more than 1 function in it

(methods are functions inside class)

#include <iostream>

using namespace std;

class employee

{

private:

    int a, b, c;

    // it is private and can only be accessed from here(that is if we want to give them a

    // integer value we can only give it from within the function)

public:

    int d, e;

    void bata(int a1, int b2, int c3)

    {

        a = 1;

        b = 2;

        c = 3;

        // we have given the value to the private variable from here (within the function)

    };

    void data()

    {

        cout << a << endl;

        cout << b << endl;

        cout << c << endl;

        cout << d << endl;

        cout << e << endl;

    };

};

int main(int argc, char const \*argv[])

{

    employee happy;

    happy.d = 4;

    happy.e = 5;

    // we cannot attribute the value of "a","b","c" similarly because they are in the private domain

    // they can only be accessed within the function

    happy.bata(0, 0, 0);

    // since "a","b","c" all have been given values here as well as from "bata" function

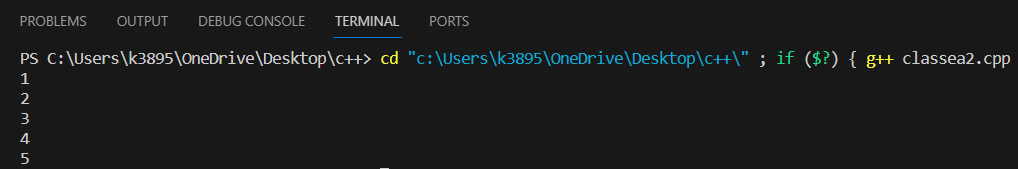
    // it will only accept value from "bata" function

    // because local>>>>global

    happy.data();

    return 0;

}



1. NESTED FUNCTION

#include <iostream>

#include <string>

// used for using string functions

using namespace std;

class Binary

{

private:

    string s;

    // string here is a data type which is basicially a group of characters

public:

    void read(void);

    void CheckBinary(void);

    void Onescompliment(void);

    void display(void);

  // only function declaration here (no function defination)

};

// function defination here

void Binary::read(void)

// since it will return nothing it is of “void” type

// jiss class ka yeh function hai uska naam likha hai “binary”

Tabhi scope resolution ka use kara hai “::” so that we can know yeh function kahan se aaya

{

    cout << "Enter a binary number" << endl;

    cin >> s;

// “read” function ka itna hi role hai ki ek binary number le user se

}

void Binary::CheckBinary(void)

{

    read();

    // nested function

This basically means that humne same “class ka function” doosre function ke andar use kara

    for (int i = 0; i < s.length(); i++)

    // another string function s.length which basically says throughout the length of string….

    {

        if (s.at(i) != '0' && s.at(i) != '1')

//….if the character at that position is not “0” or “1” then …

        // as member of string they have single quotation

        // another string function which says "at" that position of the string

        {

            cout << "This is not a binary number" << endl;

//……

          exit(0);

            // this basically means end of the program

        }

    }

//this function basically says that jo binary number input aaya hai usse check karo ki vo binary hai ya nahi

}

void Binary::Onescompliment(void)

{

    for (int i = 0; i < s.length(); i++)

    {

        if (s.at(i) == '0')

//agar wahan “0” hai toh “1” kardo

        {

            s.at(i) = '1';

        }

        Else

//nahi toh “1” hai toh “0” kardo

        {

            s.at(i) = '0';

        }

    }

//basically inverse of binary if “1” then “0” and “0” then “1”

}

void Binary::display(void)

{

    cout << "Displaying the binary number" << endl;

    for (int i = 0; i < s.length(); i++)

    {

        cout << s.at(i);

    }

    cout << endl;

}

int main(int argc, char const \*argv[])

{

    Binary b;

    b.CheckBinary();

//humne “read” function kahin bhi use nahi kara because vo nested hai “checkbinary’ mein

    b.display();

    b.Onescompliment();

    b.display();

    return 0;

}

1. Arrays in class

#include <iostream>

using namespace std;

class shop

{

private:

    int Id[10];

    int price[10];

public:

    int index = 0;

    void setprice()

    {

        cout << "Enter Id of your item number " << index + 1 << endl;

        cin >> Id[index];

        /\* we directly cannot write the value of an array

         we need to write the value of the nth memeber of array \*/

        cout << "Enter price of your item " << endl;

        cin >> price[index];

        index++;

/\* the value of index and the index itself

remains same for both the functions present in “public” \*/

    }

    void displayprice()

    {

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

        {

            cout << "For Item number " << i + 1 << endl;

            cout << "The price is " << price[i] << endl;

        }

    }

};

int main(int argc, char const \*argv[])

{

    shop dukaan;

    int n;

    cout << "How many Items do you want?" << endl;

    cin >> n;

    for (int i = 0; i < n; i++)

    {

        dukaan.setprice();

    }

    dukaan.displayprice();

    return 0;

}

**OUTPUT**

How many Items do you want?

2

Enter Id of your item number 1

55

Enter price of your item

100

Enter Id of your item number 2

56

Enter price of your item

200

For Item number 1

The price is 100

For Item number 2

The price is 200

1. STATIC

#include <iostream>

using namespace std;

class employee

{

private:

    int id;

    static int count /\*=1000(will be invalid here but)\*/;

    /\*

    by adding "static" we make the value of "count" variable continous for

    each "object(in this case happy,tushu,harsh)" of the class

     \*/

public:

    void setdata(void)

    {

        cout << "enter the id" << endl;

        cin >> id;

        count++;

    }

    void getdata(void)

    {

        cout << "the Id of the employee is " << id <<" and this is employee number "<<count<<endl;

    }

    static void getcount(void)

    // static function can only access static variables

    {

        cout<<"the value of count is "<<count<<endl;

        // cout<<id;

        // ( since id isn’t static function thus this would be invalid )

    }

};

int employee::count/\*=1000(would be valid here)\*/;

int main(int argc, char const \*argv[])

{

    employee happy, tushu, harsh;

    happy.setdata();

    happy.getdata();

    employee::getcount;

    tushu.setdata();

    tushu.getdata();

    employee::getcount;

    harsh.setdata();

    harsh.getdata();

    employee::getcount;

    return 0;

}

1. Arrays as Objects

#include<iostream>

using namespace std;

class employee

{

    int id;

    int salary;

    public:

    void setid(void)

    {

        salary=122;

        cout<<"enter the id of employee"<<endl;

        cin>>id;

    }

    void getid(void)

    {

        cout<<"the id of the employee is "<<id<<endl;

    }

};

int main(int argc, char const \*argv[])

{

    employee array[4];

    for (int i = 0; i < 4; i++)

    {

        array[i].setid();

        array[i].getid();

    }

    return 0;

}

1. PASSING OBJECTS AS ARGUMNETS OF FUNCTION

#include<iostream>

using namespace std;

class complex

{

    int a;

    int b;

    // private by default

public:

void setdata(int v1,int v2)

{

    a=v1;

    b=v2;

}

void setdatabysum(complex o1,complex o2)

// here the arguments are objects of the class

{

    a=o1.a+o2.a;

    // "a of c3"="o1 ka a"+"o2 ka a"

    b=o1.b+o2.b;

}

void printnumber()

{

    cout<<"your complex number is "<<a<<"+"<<b<<"i"<<endl;

}

}

int main(int argc, char const \*argv[])

{

    complex c1,c2,c3;

    c1.setdata(1,2);

    c1.printnumber();

    c2.setdata(3,4);

    c2.printnumber();

    c3.setdatabysum();

    c3.printnumber();

    return 0;

}

1. FRIEND CLASSES

#include<iostream>

using namespace std;

// 1 + 4i

// 5 + 8i

// -------

// 6 + 12i

class Complex{

    int a, b;

    friend Complex sumComplex(Complex o1, Complex o2);

    public:

        void setNumber(int n1, int n2){

            a = n1;

            b = n2;

        }

        // Below line means that non member - sumComplex funtion is allowed to do anything with my private parts (members)

        void printNumber(){

            cout<<"Your number is "<<a<<" + "<<b<<"i"<<endl;

        }

};

Complex sumComplex(Complex o1, Complex o2){

    Complex o3;

    o3.setNumber((o1.a + o2.a), (o1.b+o2.b))

    ;

    return o3;

}

int main(){

    Complex c1, c2, sum;

    c1.setNumber(1, 4);

    c1.printNumber();

    c2.setNumber(5, 8);

    c2.printNumber();

    sum = sumComplex(c1, c2);

    sum.printNumber();

    return 0;

}

/\* Properties of friend functions

1. Not in the scope of class

2. since it is not in the scope of the class, it cannot be called from the object of that class. c1.sumComplex() == Invalid

3. Can be invoked without the help of any object

4. Usually contans the objects as arguments

5. Can be declared inside public or private section of the class

6. It cannot access the members directly by their names and need object\_name.member\_name to access any member.

\*/

#include<iostream>

using namespace std;

class Y;

// …… therefore yahan mention(declare) kara gya hai class Y

// this is called forward declaration

class X{

    int data;

    public:

        void setValue(int value){

            data = value;

        }

    friend void add(X, Y);

 //here compiler doesn't know what is y therefore (upar)...

};

class Y{

    int num;

    public:

        void setValue(int value){

            num = value;

        }

    friend void add(X, Y);

};

void add(X o1, Y o2){

    cout<<"Summing data of X and Y objects gives me "<< o1.data + o2.num;

}

int main(){

    X a1;

    a1.setValue(3);

    Y b1;

    b1.setValue(15);

    add(a1, b1);

    return 0;

}

1. CONSTRUCTORS

#include <iostream>

using namespace std;

class complex

{

    int a, b;

public:

    complex(void)

    {

        a = 10;

        b = 20;

        cout << "Hello World" << endl;

    }

    // constructor >>> normal function (in this case "setnumber")

    void setnumber()

    {

        a = 50;

        b = 60;

    }

    void printnumber()

    {

        cout << "Your number is " << a << "+" << b << "i" << endl;

        /\* "a","b" same rhega within the class i.e

         jo bhi a and b ki value milegi upar se vo yeh use kr lega\*/

“a” and “b” ki value constructor se aayega na ki normal function se

    }

};

int main(int argc, char const \*argv[])

{

    complex c1, c2, c3;

    c1.printnumber();

jab bhi hum printnumber karenge toh “a” and “b” ke two values possible hai

but

kyuki ek constructor se aari hai toh vo autmotacially invoke hojayegi

    c2.printnumber();

    c3.printnumber();

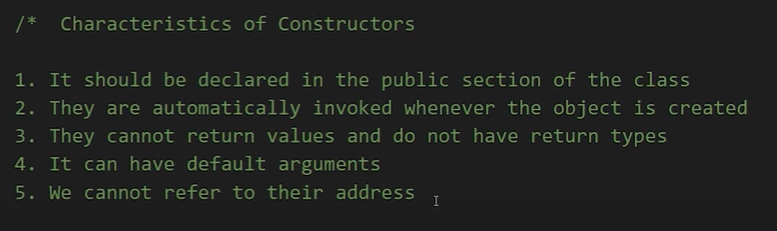
    return 0;

}

// Constructor is a special member function with the same name as of the class.

//It is used to initialize the objects of its class

//It is automatically invoked whenever an object is created



#include <iostream>

using namespace std;

class complex

{

    int a, b;

    // private “variable” of the class “complex”

public:

    complex(int x, int y)

    // this will take arguments of "int" type

    {

        a = x;

        // private "variable" "a"="x" jo humein function call hote time milega

        b = y;

        // private "variable" "b"="y" jo humein function call hote time milega

    };

    void printnumber()

    {

        cout << "your number is " << a << "+" << b << "i" << endl;

        // idhar jo "a" and "b" hai vo private "variable" se aayenge

    }

};

int main(int argc, char const \*argv[])

{

    complex a(4,6);

    // implicit call

    /\*

     kyuki yeh ek "constructor" hai

     isliye

     ek saaath object bhi bana liya and function ko call bhi kardiya(i.e when the object is made it is invoked automatically)

     which is the function of "constructor"

     \*/

    a.printnumber();

    complex b = complex(5, 7);

    // explicit call

    /\*this simply means that humne ab constructor ko normal function ki tarah treat kara hai

    that is

    ab automatically constructor invoke nahi kare object banate hi

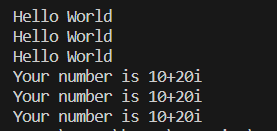
    but "object" alag se bana rahe hai and "function call" alag se karre hai

    \*/

    b.printnumber();

    return 0;

}

OUTPUT: 

1. PARAMETERIZED CONSTRUCTOR

Implicit type conversion is done automatically by the compiler, while explicit type conversion is done manually by the programmer

1. CONSTRUCTOR OVERLOADING

If there is constructor overloading (more than 1 constructor of the same name) then the one matching the parameters will be used

#include <iostream>

using namespace std;

class Complex

{

    int a, b;

public:

    //  1.

     Complex(){

        a = 0;

        b =0;

    }

// 2.

    Complex(int x, int y)

    {

        a = x;

        b = y;

    }

// 3.

    Complex(int x){

        a = x;

        b = 0;

    }

    void printNumber()

    {

        cout << "Your number is " << a << " + " << b << "i" << endl;

    }

};

int main()

{

    Complex c1(4, 6);

    // parameters matches 2.complex

    c1.printNumber();

    Complex c2(5);

    // parameter matches 1.complex

    c2.printNumber();

    Complex c3;

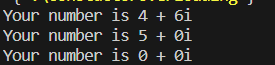
    // parameter matches 3.complex

    c3.printNumber();

    return 0;

}

Output



1. DEFAULT CONSTRUCTOR

In the default constructor if the values are given in “main function”

Then well and good and if not then the default value is also provided

#include<iostream>

using namespace std;

class Simple{

    int data1;

    int data2;

    int data3;

    public:

//here “b=9” acts as a default constructor as “b” ki value given nahi hoti aage jaakr toh “9” use hojata but aage given hai

        Simple(int a, int b=9, int c=8){

            data1 = a;

            data2 = b;

            data3 = c;

        }

        void printData();

};

void Simple :: printData(){

    cout<<"The value of data1, data2 and data3 is "<<data1<<", "<< data2<<" and "<< data3<<endl;

}

int main(){

    Simple s(12, 13);

    s.printData();

    return 0;

}

Output:



We need to have a blank constructor for supporting another constructor (having input values as int,char,float,etc..) because in case if you miss providing any input code will automatically run as it has a default constructor which can run even on providing no input

This is a good coding practice

1. Copy Constructor

The copy constructor is a special member function in C++ that initializes an object using another object of the same class.

A copy constructor takes a reference to an object of the same class as an argument.

A copy constructor is a type of constructor that creates a copy of another object.

If we want one object to resemble another object we can use a copy constructor.

When no copy constructor is found, compiler supplies its own copy instructor.

#include <iostream>

using namespace std;

class number

{

    int a;

public:

    number()

    {

        a = 0;

    }

    number(int num)

    {

        a = num;

    }

    number(number &obj)

    {

        cout << "Copy constructor called!!" << endl;

        a = obj.a;

        /\*this simply states that "a" ko "obj ke a" ke barabar kardo

        and idhar "obj"="object of class called numbers" eg.. "x,y,z" \*/

    }

    void display()

    {

        cout << "the number for this object is " << a << endl;

    }

};

int main(int argc, char const \*argv[])

{

    number x, y, z(55), z2;

    // "z" jaayega "integer waale constructor" mein

    z.display();

    number z1(z);

     /\*copy constructor invoked because it is asking to take input "z" to copy. \*/

    z1.display();

    z2 = z;

    z2.display();

     /\*copy constructor NOT invoked because z2 is already an object that ahs void as input\*/

    return 0;

}

1. DESTRUCTOR ( run the below written code to get the output and match the output)

A destructor is called for a class object when that object passes out of scope or is explicitly deleted

Destructor is invoked automatically whenever an object is over(destructed)

    #include<iostream>

using namespace std;

int count=0;

class num

{

public:

num()

{

    count++;

    cout<<"this is the time when constuctor is called for object number "<<count<<endl;

}

~num()

{

    cout<<"this is the time when destructor is called for object number"<<count<<endl;

    count--;

}

};

int main(int argc, char const \*argv[])

{

    cout<<"we are inside our main function"<<endl;

    cout<<"creating first object n1"<<endl;

    num n1;

    // SINCE AN OBJECT IS MADE ,, CONSTRUCTOR WILL RUN

    {

        cout<<"Entering this block"<<endl;

        cout<<"creating two more objects"<<endl;

        num n2,n3;

        // SINCE OBJECTS ARE MADE ,, CONSTRUCTOR WILL RUN

        cout<<"exiting this block"<<endl;

    }

    /\*anything and everything that is created wihin the scope "{}" will end or destruct after the scope is over \*/

    // therefore N2 and N3 will end here and thus destructor will run

    cout<<"back to main"<<endl;

    return 0;

}

// since N1 will end here destructor will run for it here.

1. INHERITANCE

Inheritance is a process of inheriting attributes of the base class by a derived class.

// Derived Class syntax…

class {{derived-class-name}} : {{visibility-mode}} {{base-class-name}}

{

whatever is written in {} needs to be filled by our own like

class members/methods/etc...

}

* Default visibility mode is private
* Public Visibility Mode: Public members of the base class becomes Public members of the derived class
* Private Visibility Mode: Public members of the base class become private members of the derived class
* Private members are never inherited

#include <iostream>

using namespace std;

// Base Class

class Employee

{

public:

    int id;

    float salary;

    Employee(int inpId)

    {

        id = inpId;

        salary = 34.0;

    }

    Employee() {}

    // RATTA!!

    /\*whenever a derived class(in this case "programmer") is made ,

    default constructor is called from base class(in this case "employee") thus it is created\*/

};

// Creating a "Programmer" class derived from "Employee" Base class

class Programmer : public Employee

{

public:

    int languageCode;

    Programmer(int inpId)

    {

        id = inpId;

        // we didnt initialise "id" because we already initialised it in base class

        languageCode = 9;

    }

    void getData()

    {

        cout << id << endl;

    }

};

int main()

{

    Employee harry(1), rohan(2);

    cout << harry.salary << endl;

    cout << rohan.salary << endl;

    Programmer skillF(10);

    cout << skillF.languageCode << endl;

    cout << skillF.id << endl;

    // could only be used to print "id" when  it is public

    skillF.getData();

    /\*it is made because sometimes some variables(in this case  for eg it could have been "id") are not public

    thus to print them this method is used using a function\*/

    return 0;

}

OUTPUT



#include <iostream>

using namespace std;

class Base

{

protected:

    // kind of private but can be inherited

    int a;

private:

    int c;

};

class Derived : protected Base

{

};

 int main(int argc, char const \*argv[])

{

    Base b;

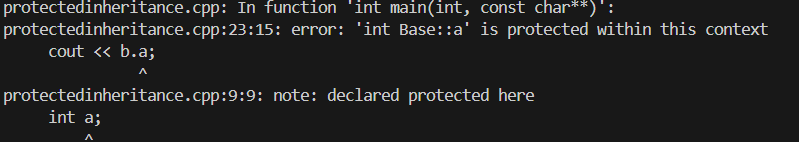
    Derived d;

    cout << b.a;

    // this cannot be printed directly because "protected" member can be inherited but cannot be printed

    return 0;

}



#include <iostream>

using namespace std;

class Base

{

public:

    int a, b;

    void input()

    {

        cout << "enter n1" << endl;

        cin >> a;

        cout<<&a<<endl;

        cout << "enter n2" << endl;

        cin >> b;

        cout<<&b<<endl;

    }

    void foo()

    {

        cout << a << "+" << b << "=" << a + b << endl;

        cout<<&a<<endl;

        cout<<&b<<endl;

    }

};

/\* the variable "a" and "b" used in the two "functions" above within

 the same "class" are same for both the "functions" \*/

class Derived : public Base

{

public:

    void bar()

    {

        foo(); // Call the foo function from the Base class

    }

};

/\* whereas when the same "foo" function is called from

 different "class" the "a" and "b" which are to be used

 to perform the operation are different than the ones

 used in the previous class \*/

int main()

{

    Base b;

    b.input();

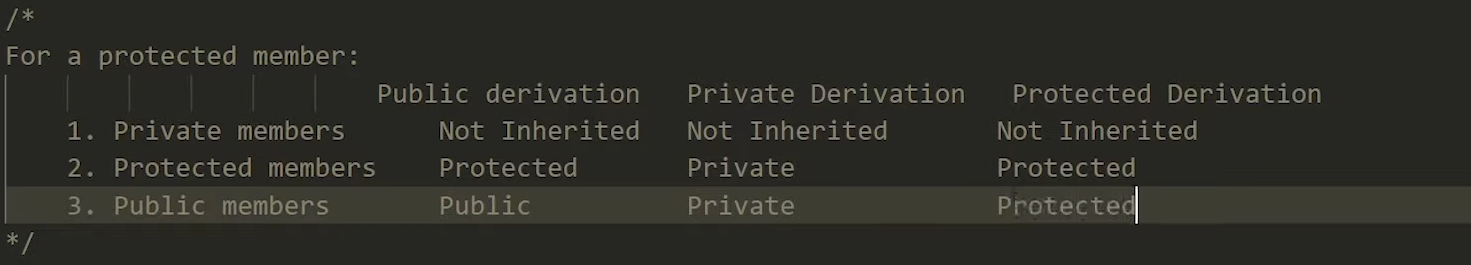
    b.foo();

    Derived d;

    d.bar();

    return 0;

}



1. Multilevel inheritance is a type of inheritance in which one derived class is inherited from another derived class. For example, we have three classes “animal”, “mammal” and “cow”. If the “mammal” class is inherited from the “animal” class and “cow” class is inherited from “mammal” which means that the “mammal” class can now implement the functionalities of “animal” and “cow” class can now implement the functionalities of “mammal” class.

#include <iostream>

using namespace std;

class Student

{

protected:

    int roll\_number;

public:

    void set\_roll\_number(int);

    void get\_roll\_number(void);

};

void Student::set\_roll\_number(int r)

{

    roll\_number = r;

    /\* since "roll number " is "protected" it cannot be given value in "main function"

    therefore we are assigning its value here \*/

};

void Student ::get\_roll\_number()

{

    cout << "the roll number is " << roll\_number << endl;

};

class Exam : public Student

{

protected:

    float maths;

    float physics;

public:

    void set\_marks(float, float);

    void get\_marks(void);

};

void Exam::set\_marks(float m1, float m2)

    /\* since "maths and physics " are "protected" it cannot be given value in "main function"

    therefore we are assigning its value here \*/

{

    maths = m1;

    physics = m2;

}

void Exam::get\_marks()

{

    cout << "the marks obtained in maths are :" << maths << endl;

    cout << "the marks obtained in physics are :" << physics << endl;

};

class Result : public Exam

{

    float percentage;

public:

    void display()

    {

        get\_roll\_number();

        get\_marks();

        /\*since the "RESULT" class is inherited "publically"

        we can use the function from its base class (and uski bhi base class)\*/

        cout << "your percentage is " << (maths + physics) / 2 << "%" << endl;

    }

};

int main(int argc, char const \*argv[])

{

    Result happy;

    happy.set\_roll\_number(5);

    happy.set\_marks(90,90);

    happy.display();

    return 0;

}

1. MULTIPLEINHERITANCE

Multiple inheritances are a type of inheritance in which one derived class is inherited with more than one base class. For example, we have three classes “employee”, “assistant” and “programmer”. If the “programmer” class is inherited from the “employee” and “assistant” class which means that the “programmer” class can now implement the functionalities of the “employee” and “assistant” class.

SYNTAX:

class Derived: visibility-mode base1, visibility-mode base2

{

Class body of class "DerivedC"

};

#include<iostream>

using namespace std;

class Base1

{

protected:

    int base1int;

public:

    void set\_base1int(int a)

    {

        base1int = a;

    }

};

class Base2

{

protected:

    int base2int;

public:

    void set\_base2int(int a)

    {

        base2int = a;

    }

};

class Base3

{

protected:

    int base3int;

public:

    void set\_base3int(int a)

    {

        base3int = a;

    }

};

class Derived : public Base1, public Base2, public Base3

{

    public:

        void show(){

            cout << "The value of Base1 is " << base1int<<endl;

            // we can use a "variable" from "base class" in "derived class"

            cout << "The value of Base2 is " << base2int<<endl;

            cout << "The value of Base3 is " << base3int<<endl;

            cout << "The sum of these values is " << base1int + base2int + base3int << endl;

        }

};

int main()

{

    Derived harry;

    harry.set\_base1int(25);

    /\* we can call "functions" of "base class" from "derived class object"

    if we want those functions for objetcs of derived class \*/

    harry.set\_base2int(5);

    harry.set\_base3int(15);

    harry.show();

    return 0;

}

1. VIRTUAL BASE CLASS

Virtual base classes in C++ are used to prevent multiple instances of a given class from appearing in an inheritance hierarchy when using multiple inheritances. A base class is the class from which other classes are derived. The derived(child) classes have access to the variables and methods/functions of a base(parent) class. The entire structure is known as the inheritance hierarchy.

When a class is specified as a virtual base class, it prevents duplication of its data members. If a virtual base class is not used, all the derived classes will get duplicated data members. In this case, the compiler cannot decide which one to execute.

#include <iostream>

using namespace std;

class Student

{

protected:

    int rollno;

public:

    void setnumber(int a)

    {

        rollno = a;

    }

    void printnumber(void)

    {

        cout << "your roll no is " << rollno << endl;

    };

};

class Test : virtual public Student

{

protected:

    float maths, physics;

public:

    void setmarks(float m1, float m2)

    {

        maths = m1;

        physics = m2;

    }

    void printmarks(void)

    {

        cout << "your result is here" << endl;

        cout << "maths " << maths << endl;

        cout << "physics " << physics << endl;

    }

};

class Sports : virtual  public Student

{

protected:

    float score;

public:

    void setscore(float sc)

    {

        score = sc;

    }

    void printscore(void)

    {

        cout << "your PT score is" << score << endl;

    }

};

class Result :  public Test,  public Sports, virtual public Student

{

    float total;

public:

    void display(void)

    {

        total = maths + physics + score;

        printnumber();

        printmarks();

        printscore();

        cout << "your total score is" << total << endl;

    }

};

int main(int argc, char const \*argv[])

{

    Result happy;

    happy.setnumber(4200);

    happy.setmarks(60, 99.6);

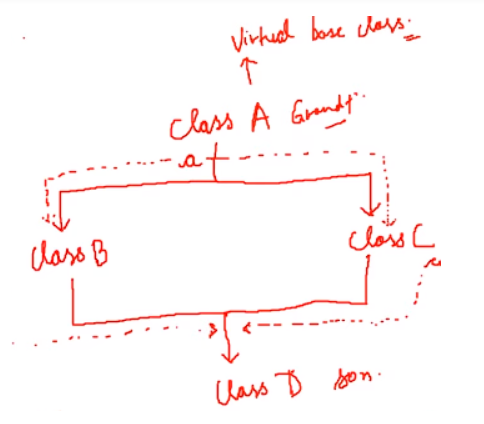
    happy.setscore(9);

    happy.display();

    return 0;

}

SPECIAL NOTE FOR VIRTUAL CLASS:



In cases where classes are being formed in such pattern as A,B,C,D

Even if visibly no “function” or “variable” is being inherited from the “base class” i.e “A”

We still use the keyword “virtual” in front of “base class” when making a “derived class” i.e “B” and “C”

Example:

Class B :: virtual public A

Class C :: virtual public A

This is done because even if visibly we cannot see functions and variables being inherited directly but they are being inherited when the classes are being derived

So when we create class “D” from classes “B” and “C” the functions and members of “A” are being inherited twice even if it cannot be seen directly they are being inherited

Therefore “virtual” keyword is used in front of the “base class” i.e “A” everytime

1. CONSTRUCTOR IN DERIVED CLASS

If the base class constructor does not have any arguments, there is no need for any constructor in the derived class

But if there are one or more arguments in the base class constructor, derived class need to pass argument to the base class constructor This is because the base class constructor is called first when an object of the derived class is created.

If both base and derived classes have constructors, base class constructor is executed first

In multiple inheritances, base classes are constructed in the order in which they appear in the class deceleration. For example if there are three classes “A”, “B”, and “C”, and the class “C” is inheriting classes “A” and “B”. If the class “A” is written before class “B” then the constructor of class “A” will be executed first. But if the class “B” is written before class “A” then the constructor of class “B” will be executed first.

In multilevel inheritance, the constructors are executed in the order of inheritance. For example if there are three classes “A”, “B”, and “C”, and the class “B” is inheriting classes “A” and the class “C” is inheriting classes “B”. Then the constructor will run according to the order of inheritance such as the constructor of class “A” will be called first then the constructor of class “B” will be called and at the end constructor of class “C” will be called.

**Special Syntax**

C++ supports a special syntax for passing arguments to multiple base classes

The constructor of the derived class receives all the arguments at once and then will pass the call to the respective base classes

The body is called after the constructors is finished executing

**Special Case of Virtual Base Class**

The constructors for virtual base classes are invoked before a non-virtual base class

If there are multiple virtual base classes, they are invoked in the order declared

Any non-virtual base class are then constructed before the derived class constructor is executed

#include <iostream>

using namespace std;

class Base1

{

    int data1;

public:

    Base1(int i)

    /\*

    "base1" constructor isn't called ( as there is no object made for its class )

    still it gets it variable(int i) from the "main function"

    which is impossible normally according to our previous knowledge

    the answer to this query is

    As "Derived" class is inhereting "Base1" therefore

    if the "base class constructor" needs a parameter it will be provided by the

    "derived class constructor" (which we have done on line no. 59)

    while making "derived class constructor"

    This is because the base class constructor is called first when an object of the derived class is created.

    \*/

    {

        data1 = i;

        cout << "base1 class constructor called" << endl;

    }

    /\*

    although the "base1 constructor" isn't called

    it is still working when the constructor "derived"

    is made as it passes some of its arguments to

    constructor "Base1" and "Base2"

    that is why function "printdata1" and "printdata2"

    can get its value

    \*/

    void printdata1(void)

    {

        cout << "the value of data1 is " << data1 << endl;

    }

};

class Base2

{

    int data2;

public:

    Base2(int i)

    {

        data2 = i;

        cout << "base2 class constructor called" << endl;

    }

    void printdata2(void)

    {

        cout << "the value of data2 is " << data2 << endl;

    }

};

class Derived : public Base1, public Base2

{

    int derived1, derived2;

public:

    Derived(int a, int b, int c, int d) : Base1(a), Base2(b)

    /\*

    if the "base class constructor" needs a parameter it will be provided by the

    "derived class constructor"

    This is because the "base class constructor" is called first when an "object" of the "derived class" is created.

    \*/

    {

        derived1 = c;

        derived2 = d;

        cout << "Derived class constructor is called" << endl;

    }

    void printdataderived(void)

    {

        cout << "the value of derived1 is " << derived1 << endl;

        cout << "the value of derived2 is " << derived2 << endl;

    }

};

int main(int argc, char const \*argv[])

{

    Derived happy(1,2,3,4);

    happy.printdata1();

    happy.printdata2();

    happy.printdataderived();

    return 0;

}

1. INITIALIZATION LIST IN CONSTRUCTORS

The initialization list in constructors is another concept of initializing the data members of the class.

#include<iostream>

using namespace std;

class Test

{

    int a;

    int b;

public:

    Test(int i, int j) : a(i), b(j)

    // it can be done in such fashion or the usual one

    // int a=i;

    // int b=j;

    {

        cout << "Constructor executed"<<endl;

        cout << "Value of a is "<<a<<endl;

        cout << "Value of b is "<<b<<endl;

    }

};

int main()

{

    Test t(4, 6);

    return 0;

}

The main thing to note here is that if we use the code shown below to initialize data members the compiler will throw an error because the data member “a” is being initialized first and the “b” is being initialized second so we have to assign the value to “a” data member first.

Test(int i, int j) : b(j), a(i+b)

But if we use the code shown below to initialize data members the compiler will not throw an error because the data member “a” is being initialized first and we are assigning the value to the data member “a” first.

Test(int i, int j) : a(i), b(a + j)

1. DYNAMICALLY USE OF POINTER

The “new” and “delete” operators are used in C++ to dynamically allocate and deallocate memory. They are essential for creating and managing objects, arrays, and other data structures in C++.

int main(int argc, char const \*argv[])

{

    int \*h= new int(5);

    cout<<"the value stored at address h is "<<\*h<<endl;

    // dynamically used pointer

int \*arr=new int[4];

    arr[0]=10;

    arr[1]=20;

    arr[2]=30;

    cout<<"the value of arr[0] is "<<arr[0]<<endl;

    cout<<"the value of arr[1] is "<<arr[1]<<endl;

    cout<<"the value of arr[2] is "<<arr[2]<<endl;

   // dyanmically used the pointer to make array

When you use the new operator, it allocates memory on the heap and returns a pointer to that memory. The memory is not initialized, so you need to initialize it manually.

When you use the delete operator, it deallocates the memory that was previously allocated using the new operator. The memory is returned to the heap, and the pointer becomes invalid.

int \*arr=new int[4];

    arr[0]=10;

    arr[1]=20;

    arr[2]=30;

    delete[] arr;

    cout<<"the value of arr[0] is "<<arr[0]<<endl;

    cout<<"the value of arr[1] is "<<arr[1]<<endl;

    cout<<"the value of arr[2] is "<<arr[2]<<endl;

   // dyanmically used the pointer to make array and deleted it using "delete"

\*(arr+1)=20;

    // "\*"-->value of , (arr+1)--> arr[0]+1 =arr[1]

    // thus the whole thing is read as

    // the value of arr[1]=20

1. POINTERTOOBJECT

#include<iostream>

using namespace std;

class Complex

{

    int real,imaginary;

    public:

    void GetData()

    {

        cout<<"the real part is: "<<real<<endl;

        cout<<"the imaginary part is: "<<imaginary<<endl;

    }

    void SetData(int a,int b)

    {

        real=a;

        imaginary=b;

    }

};

int main(int argc, char const \*argv[])

{

    // one way to do it

    Complex c1;

    c1.SetData(1,54);

    c1.GetData();

    // another way to do it

    Complex \*ptr=&c1;

    /\*since "c1" ek complex class ka object hai isi liye

    uska data type bhi complex class walaa hai

    thus "ptr" naam ke "pointer" ka data type hai "complex" \*/

    (\*ptr).SetData(1,54);

    /\* kyuki "ptr" naam ka "pointer" point kara hai towards object c1

    that is ptr is storing the address of c1

    iss vajah se using "dereferencing (\*)" "(\*ptr)=c1"\*/

    (\*ptr).GetData();

    // another way to it

    Complex \*p= new Complex;

    /\*

     we can create a new "object" using this method

    (only "\*p" can be changed in this statement

    except it everything will stay the same

    that is "Complex","new" will stay in place

    \*/

    (\*p).SetData(1,54);

    (\*p).GetData();

    return 0;

}

Note: Arrowpointer “->” is mentioned in the DSA notes but it is taught here

1. ARRAY OF OBJECTS

#include<iostream>

using namespace std;

class Shop

{

    int id;

    float price;

    public:

    void SetData(int a,float b)

    {

       id = a;

       price =b;

    }

    void GetData()

    {

     cout<<"code of this item is "<<id<<endl;

     cout<<"price of this item is "<<price<<endl;

    }

};

int main(int argc, char const \*argv[])

{

    int size=2;

    // array of objects

    Shop \*ptr= new Shop[size];

    /\*

    "Shop" name ka array of objects hai

    jiske andar "2 objects" honge because

    upar it is mentioned "size"=2

     \*/

    /\*

    idhar basically "object" kisi naam se nahi banaya infact

    idhar "objects" toh hai but unka naam nahi hai

    unn "objects" ko banaya gaya hai and access kara jaaraa hai

    using "pointer" (in this case "ptr")

    \*/

    Shop \*ptrtemp=ptr;

    /\*

    this basically means that the "pointer" "ptrtemp"

    is pointing towards "ptr" and further more

    "ptr" is pointing towards the " array of objects of the class"

    thus it is a way to access the "objects of the class" (without

    being totally dependent on "ptr" by which I mean ki

    if "ptr" would be running for the "object number 2"

    it isnt necessary for "ptrtemp" to also run for "object number 2"

    just because "ptr" is running for "object 2" as it is only pointing

    "ptr(indirectly the objects)" thus it can run for "object number 1"

  even if the "ptr" is running for "object number 2" or so on....)

    \*/

    int p;

    float q;

    for (int i = 0; ththisi < size; i++)

    {

        cout<<"Enter id and price of item "<<i+1<<endl;

        cin>>p>>q;

        ptr->SetData(p,q);

        /\*

         in "main" it doesnt know what is "SetData" therefore

         we have mentioned jo "ptr" hai of the data type

         " class shop " uske andar hai ek function

         "SetData" (basically we gave reference of "SetData" function

         to "main" because iss reference se pehle "main" nahi jaanta tha

         "SetData" ko)

         \*/

        ptr++;

        /\*

        basically jab iske upar "SetData function" run hua tha

        vo run hua the for "object number 1" i.e for "object number 1"

        ke liye humne "id and price" pata kar liya using the function

        "SetData"

        now humein ye same kaam "next object" ke liye bhi karna hai

        so that is why we have done "ptr++" which means

        jo upar likha hai "ptr->setdata()" ab vo "setdata" run hoga

        for the "next object" (basically it will run for (ptr+1)

        which is ultimately pointing the next object)

        \*/

    }

    for (int i = 0; i < size; i++)

    {

        cout<<"Item number : "<<i+1<<endl;

        ptrtemp->GetData();

        /\*

         in "main" it doesnt know what is "GetData" therefore          we have mentioned jo "ptrtemp" hai of the data type

         " class shop " uske andar hai ek function

         "GetData" (basically we gave reference of "GetData" function

         to "main" because iss reference se pehle "main" nahi jaanta tha

         "GetData" ko)

         \*/

        ptrtemp++;

        /\*

        basically jab iske upar "GetData function" run hua tha

        vo run hua the for "object number 1" i.e for "object number 1"

        ke liye humne "code and price" ko print kar liya using the function

        "GetData"

        now humein ye same kaam "next object" ke liye bhi karna hai

        so that is why we have done "ptrtemp++" which means

        jo upar likha hai "ptrtemp->getdata()" ab vo "getdata" run hoga

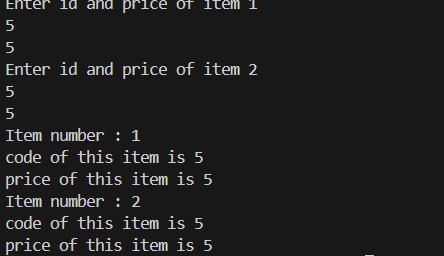
        for the "next object" (basically it will run for (ptrtemp+1)

        which is ultimately pointing the next object)

        \*/

    }

    return 0;





Red line area is governed by the pointer “ptr” whereas

The blue line area is governed by the pointer “ptrtemp”

It is done so because we can clearly see “ptr”

(red line area) has already reached item 1 and 2

And now it is difficult and hectic to again bring it back to object 1 and 2 again

Therefore we have created another pointer “ptrtemp” which will again start from item 1 and 2 independently from the pointer “ptr”

pointer



1. THIS POINTER

“this” pointer points to the object which calls the member function/variable.

#include<iostream>

using namespace std;

class A

{

    int a;

    public:

        void setData(int a)

        {

            this->a = a;

            /\*

            if we dont use "this ->" then

            compiler would be confused because

            it wont be able to differentiate between both the "a"

            but to make the compiler know which is local "a" and which "a" isnt

            local , we use "this->" ,"this->a" refers to the "not local a"

            \*/

        }

        void getData()

        {

            cout<<"The value of a is "<<a<<endl;

        }

};

int main()

{

    A b;

    b.setData(4);

    b.getData();

    return 0;

}

1. POLYMORPHISM

“Poly” means several and “morphism” means form. So we can say that polymorphism is something that has several forms or we can say it as one name and multiple forms. There are two types of polymorphism:

* Compile-time polymorphism
* Run time polymorphism

In compile-time polymorphism, it is already known which function will run. Compile-time polymorphism is also called early binding, which means that you are already bound to the function call and you know that this function is going to run.

**Run Time Polymorphism**

In the run-time polymorphism, the compiler doesn’t know already what will happen at run time. Run time polymorphism is also called late binding. The run time polymorphism is considered slow because function calls are decided at run time. Run time polymorphism can be achieved from the virtual function.

Virtual Function

A function that is in the parent class but redefined in the child class is called a virtual function. “virtual” keyword is used to declare a virtual function.

1. POINTERS IN DERIVED CLASS

#include <iostream>

using namespace std;

class BaseClass

{

public:

    int varbase;

    void display()

    {

        cout << "displaying base class variable varbase " << varbase << endl;

    }

};

class DerivedClass : public BaseClass

{

public:

    int varderived;

    void display()

    {

        cout << "displaying base class variable varbase " << varbase << endl;

        cout << "displaying derived class variable varderived " << varderived << endl;

    }

};

int main(int argc, char const \*argv[])

{

    BaseClass \*baseclasspointer;

    BaseClass objbase;

    DerivedClass objderived;

    baseclasspointer = &objderived;

    /\*

    we are pointing towards "object of derived class" using the pointer

    of data type "baseclass"

   \*/

    baseclasspointer->varbase=34;

    baseclasspointer->display();

    /\*

    it will run "display" function of "baseclass"

    this is because

    although the "pointer" point towards the "object of derived class"

    but the "pointer" is of base class and hence it happens.

    \*/

//    baseclasspointer->varderived=50;

   /\*

   this wont happen because

   if we have made "baseclasspointer" then it can only access the

   "properties"(variable in class) and "methods"(functions in class)

   inherited from "baseclass".

   even though it points towards "object of derived class"

   still it cannot access it

   \*/

  baseclasspointer->varbase=5000;

  baseclasspointer->display();

  DerivedClass \*derivedclasspointer;

  derivedclasspointer=&objderived;

  derivedclasspointer->varbase=500;

  derivedclasspointer->varderived=98;

  derivedclasspointer->display();

    return 0;

}

1. VIRTUAL FUNCTIONS

#include <iostream>

#include <string>

using namespace std;

class CWH

{

protected:

    string title;

    //"string" is a type specifier that is used to declare a string object

    // it is not a data type

    float rating;

public:

    CWH(string s, float r)

    {

        title = s;

        rating = r;

    }

    virtual void display()

    {

    }

};

class CWHVideo : public CWH

{

    float videoLength;

public:

    CWHVideo(string s, float r , float vl) : CWH(s,r)

    {

        videoLength = vl;

    }

    void display()

    {

        cout << "This is an amazing video with title " << title << endl;

        cout << "Ratings: " << rating << " out of 5 stars" << endl;

        cout << "Length of this video is: " << videoLength << " minutes" << endl;

    }

};

class CWHText : public CWH

{

    int words;

public:

    CWHText(string s, float r, int wc):CWH(s,r)

    {

        words = wc;

    }

    void display()

    {

        cout << "This is an amazing text tutorial with title " << title << endl;

        cout << "Ratings of this text tutorial: " << rating << " out of 5 stars" << endl;

        cout << "No of words in this text tutorial is: " << words << " words" << endl;

    }

};

int main()

{

    string title;

    float rating, vlen;

    int words;

    /\*

    since we need to pass all the "variables" to the "constructors" made above

    we are initializing the "variables" here

    (another method to pass the "variables" is to directly pass them through

    while creating "object" of that "class"(indirectly calling constructor) )

    but here we are passing it

    by first initializing it

    and then

    giving it a value

    and then passing the value indirectly

    \*/

    // for Code With Harry Video

    title = "Django tutorial";

    vlen = 4.56;

    rating = 4.89;

    CWHVideo djVideo(title, rating, vlen);

    // for Code With Harry Text

    title = "Django tutorial Text";

    words = 433;

    rating = 4.19;

    CWHText djText(title, rating, words);

    CWH \*tuts[2];

    /\*

    since "tuts" ek "CWH class" ka "array of object" hai isi liye

    uska data type bhi "CWH" class walaa hai

    thus "tuts" naam ke "pointer" ka data type hai "CWH"

    \*/

/\*

since "CWH" ek "base class" hai

therefore

we can call "functions" of "derived class"

using pointers of the data type "CWH"

that is why we have used "CWH \*tuts"

\*/

    tuts[0] = &djVideo;

    tuts[1] = &djText;

    tuts[0]->display();

    tuts[1]->display();

/\*

The main thing to note here is that

if we don’t use the “virtual” keyword with the “display” function of the base class

then the “display” function of the base class will run.

\*/

/\*

so when the "display function" is called by using the "base class pointer"

the "display function" of the "derived class" will run because

the "base class pointer" is pointing to the "derived class object".

\*/

    return 0;

}

/\*

previously we faced a problem of

{

if we have made "baseclasspointer" then it can only access the

"properties"(variable in class) and "methods"(functions in class)

inherited from "baseclass".

even though it points towards "object of derived class"

still it cannot access it

}

this problem is solved in this "code" using keywrod "virtual"

\*/

1. PURE VIRTUAL FUNCTIONS

Pure virtual function is a function that doesn’t perform any operation and the function is declared by assigning the value 0 to it. Pure virtual functions are declared in abstract classes.

Abstract base class is a class that has at least one pure virtual function in its body. The classes which are inheriting the base class must need to override the virtual function of the abstract class

otherwise compiler will throw an error.

   #include <iostream>

#include <string>

using namespace std;

class CWH

{

protected:

    string title;

    //"string" is a type specifier that is used to declare a string object

    // it is not a data type

    float rating;

public:

    CWH(string s, float r)

    {

        title = s;

        rating = r;

    }

    virtual void display()=0; //do nothing function🡪pure virtual function

};

class CWHVideo : public CWH

{

    float videoLength;

public:

    CWHVideo(string s, float r , float vl) : CWH(s,r)

    {

        videoLength = vl;

    }

    void display()

    {

        cout << "This is an amazing video with title " << title << endl;

        cout << "Ratings: " << rating << " out of 5 stars" << endl;

        cout << "Length of this video is: " << videoLength << " minutes" << endl;

    }

};

class CWHText : public CWH

{

    int words;

public:

    CWHText(string s, float r, int wc):CWH(s,r)

    {

        words = wc;

    }

    void display()

    {

        cout << "This is an amazing text tutorial with title " << title << endl;

        cout << "Ratings of this text tutorial: " << rating << " out of 5 stars" << endl;

        cout << "No of words in this text tutorial is: " << words << " words" << endl;

    }

};

int main()

{

    string title;

    float rating, vlen;

    int words;

    /\*

    since we need to pass all the "variables" to the "constructors" made above

    we are initializing the "variables" here

    (another method to pass the "variables" is to directly pass them through

    while creating "object" of that "class"(indirectly calling constructor) )

    but here we are passing it

    by first initializing it

    and then

    giving it a value

    and then passing the value indirectly

    \*/

    // for Code With Harry Video

    title = "Django tutorial";

    vlen = 4.56;

    rating = 4.89;

    CWHVideo djVideo(title, rating, vlen);

    // for Code With Harry Text

    title = "Django tutorial Text";

    words = 433;

    rating = 4.19;

    CWHText djText(title, rating, words);

    CWH \*tuts[2];

    /\*

    since "tuts" ek "CWH class" ka "array of object" hai isi liye

    uska data type bhi "CWH" class walaa hai

    thus "tuts" naam ke "pointer" ka data type hai "CWH"

    \*/

/\*

since "CWH" ek "base class" hai

therefore

we can call "functions" of "derived class"

using pointers of the data type "CWH"

that is why we have used "CWH \*tuts"

\*/

    tuts[0] = &djVideo;

    tuts[1] = &djText;

    tuts[0]->display();

    tuts[1]->display();

/\*

The main thing to note here is that

if we don’t use the “virtual” keyword with the “display” function of the base class

then the “display” function of the base class will run.

\*/

/\*

so when the "display function" is called by using the "base class pointer"

the "display function" of the "derived class" will run because

the "base class pointer" is pointing to the "derived class object".

\*/

    return 0;

}

/\*

previously we faced a problem of

{

if we have made "baseclasspointer" then it can only access the

"properties"(variable in class) and "methods"(functions in class)

inherited from "baseclass".

even though it points towards "object of derived class"

still it cannot access it

}

this problem is solved in this "code" using keywrod "virtual"

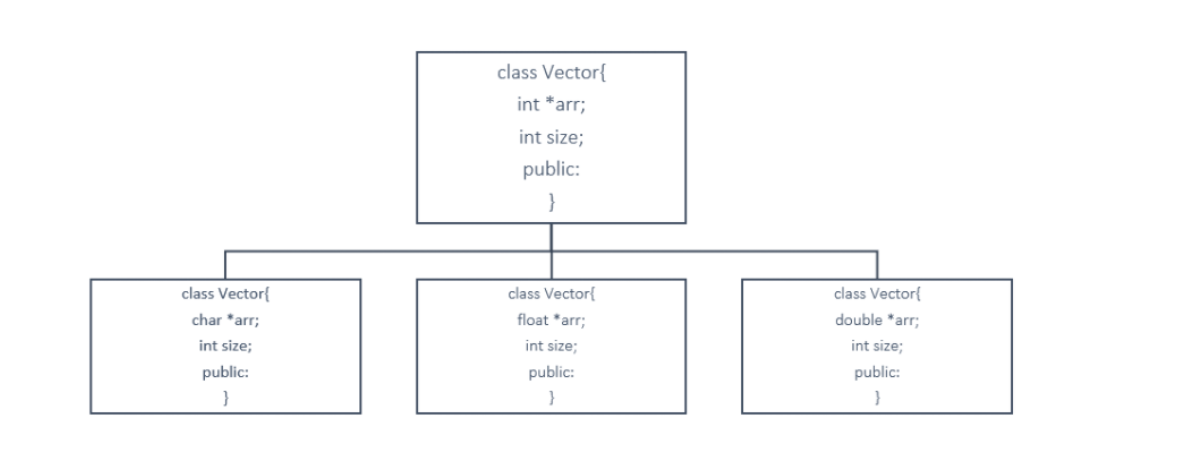
\*/

1. TEMPLATES

It is very analogous to when we said classes are the templates for objects, here templates itself are the templates of the classes. That is, what classes are for objects, templates are for classes.

 same declaration of classes for different data types. Declaring classes for every other data type(which if counted is way too much) in the very first place violates the DRY( Don’t Repeat Yourself) rule of programming

a template does the same thing to a class, what a class does to the objects. It parametrizes the data type hence making it easy for us to use different classes without having to write the whole thing again and again, violating the DRY rule. Templates furthermore give our program a generic view, where declaring one template suffices the task.



It is for “integer data type”

#include <iostream>

using namespace std;

class Vector

{

public:

    int \*arr;

    int size;

    Vector(int m)

    // m=3

    {

        size = m;

        // size=3

        arr = new int[size];

        // thus the array made is of (size of int)\*3

    }

    int dotproduct(Vector &v)

// "dotproduct" will return "integer value"

// as a "parameter" it is asking for "object" of "class Vector"

    {

        int d = 0;

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

        {

            d += this->arr[i] \* v.arr[i];

            /\*

            "d" + "local arr"(that is arr mentioned in the "constructor" i.e "v1 ka array") \* "v2 ka arr" (as v=v2,according to parameter)

            \*/

        }

        return d;

    }

};

int main(int argc, char const \*argv[])

{

    Vector v1(3);

    /\*

    the constructor named "Vector" requires parameter

    therefore we have provided "3"

   \*/

    v1.arr[0] = 0;

    v1.arr[1] = 1;

    v1.arr[2] = 2;

    /\*

    the "array" which we have created is created under the "public" domain of

    the "object v1" of the "class Vector"

    thus these are the values of the "array members" made under the "object v1"

    \*/

    Vector v2(3);

    /\*

    the constructor named "Vector" requires parameter

    therefore we have provided "3"

   \*/

    v2.arr[0] = 10;

    v2.arr[1] = 11;

    v2.arr[2] = 12;

    /\*

    the "array" which we have created is created under the "public" domain of

    the "object v2" of the "class Vector"

    thus these are the values of the "array members" made under the "object v2"

    \*/

    int a = v1.dotproduct(v2);

    /\*

    integer "a" = "v1 object" ke andar jo "function hai dotproduct" uski "return value"

    basically "a"="dotproduct function" ki jo bhi return value hogi

    \*/

    /\*

     since "dotproduct" requires an "object" of "class Vector" as  a "parameter"

     we have provided "v2"

    \*/

    cout << a << endl;

    return 0;

}

It is for “variable data type” can be int,float,double etc……

#include <iostream>

using namespace std;

template <class T>

// syntax : template <class "any name">

// we have given the name "T" right now

// and further on we can convert "T" into any data type

class Vector

{

public:

    T\*arr;

    // right now "arr"'s data type is "T" but we will further decide the data type of "T"

    int size;

    Vector(int m)

    {

        size = m;

        arr = new T[size];

    }

    T dotproduct(Vector &v)

    // right now "dotproduct" will return  data type "T" but we will further decide the data type of "T"

    {

        T d = 0;

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

        {

            d += this->arr[i] \* v.arr[i];

        }

        return d;

    }

};

int main(int argc, char const \*argv[])

{

    Vector <float> v1(3);

    // syntax : "class name" <data type> "object name"

    // while making the "object" we have mentioned the "data type" of "T"

    v1.arr[0] = 0;

    v1.arr[1] = 1.5;

    v1.arr[2] = 2.5;

    Vector <float> v2(3);

    v2.arr[0] = 10.5;

    v2.arr[1] = 11.5;

    v2.arr[2] = 12.5;

     float a  = v1.dotproduct(v2);

    /\*

    we cannot use "T" instead of "float" here to define the "data type"

    we need to mention the data type before hand only for an variable (according to my knowledge rn)

    \*/

    cout << a << endl;

    return 0;

}

It is for multiple parameters

#include <iostream>

using namespace std;

template <class T1, class T2>

/\*

we have given the name "T1" and "T2" right now to multiple parameters

and further on we can convert "T1" and "T2" into any data type

\*/

class MyClass

{

public:

    T1 data1;

    // right now "data1"'s data type is "T1" but we will further decide the data type of "T1"

    T2 data2;

    // right now "data2"'s data type is "T2" but we will further decide the data type of "T2"

    MyClass(T1 a, T2 b)

    // parameter mein "variable a" aayega of "datatype T1"

    // parameter mein "variable b" aayega of "datatype T2"

    {

        data1 = a;

        data2 = b;

    }

    void display()

    {

        cout << this->data1 << this->data2;

    }

};

int main(int argc, char const \*argv[])

{

    MyClass<int, char> obj(1, 'c');

     // while making the "object" we have mentioned the "data type" of "T1" and "T2"

    obj.display();

    return 0;

}

It is for default multiple parameters

#include <iostream>

using namespace std;

template <class T1 = int, class T2 = float, class T3 = char>

/\*

we have given the name "T1","T2" and "T3" right now to multiple parameters

and further on we can convert "T1","T2" and "T3" into any data type

\*/

/\*

we have mentioned "default data type" of "T1,T2 and T3"

which means if nothing is mentioned further in the code then these dataype will work

otherwise if something is mentioned in the code that datatype will work

\*/

class Happy

{

public:

    T1 a;

    T2 b;

    T3 c;

    Happy(T1 x, T2 y, T3 z)

    {

        a = x;

        b = y;

        c = z;

    }

    void display()

    {

        cout << "the value of a is " << a << endl;

        cout << "the value of b is " << b << endl;

        cout << "the value of c is " << c << endl;

    }

};

int main(int argc, char const \*argv[])

{

    Happy<> h(4, 6.4, 'c');

    // here we can see we haven't mentioned "data type" for "T1,T2,T3"

    h.display();

    cout << endl;

    Happy<float, char, char> g(4.5, 'o', 'c');

    // here we can see we have mentioned "data type" for "T1,T2,T3"

    g.display();

    return 0;

}

1. FUNCTION TEMPLATE

#include<iostream>

using namespace std;



float FuncAvg(int a,int b)

{

    float avg=(a+b)/2.0;

    /\*

     we are writing "2.0" and not "2" because it is of

    "float" data type



    \*/



    return avg;

}

float FuncAvg2(int a,float b)

{

    float avg=(a+b)/2.0;

    /\*

     we are writing "2.0" and not "2" because it is of

    "float" data type

    \*/

    return avg;



}

int main(int argc, char const \*argv[])

{

    float a;

    a=FuncAvg2(5,2.4);

    printf("The average of these number is %f",a);

    return 0;

}

Instead of creating “two functions” for the “same work” but “different data type”

As mentioned within the “red line”

We can create a “template function”

#include <iostream>

using namespace std;

template <class T1, class T2>

float FuncAvg(T1 a, T2 b)

{

    float avg = (a + b) / 2.0;

    return avg;

}

int main(int argc, char const \*argv[])

{

    float a;

    a = FuncAvg(5, 2.4);

    printf("The average of these number is %f", a);

    return 0;

}

#include <iostream>

using namespace std;

// "T" ek "class" hai jiska "datatype" aage decide hoga

template <class T>

/\*

we have given the name "T" right now

and further on we can convert "T" into any data type

\*/

void swapp(T &a, T &b)

/\*

since "T" is a class therefore as a "parameter"

hum "T" ke andar ke "objects" lenge

Thus using "&a,&b"

 \*/

{

    T temp = a;

    // "T datatype" ka variable "temp" create kiya hai

    a = b;

    b = temp;

    // basically swapping process

}

int main(int argc, char const \*argv[])

{

    int x = 5;

    int y = 7;

    // using "int" we have mentioned the "datatype of class T"

    swapp(x, y);

    // same "x and y" pass on honge "swapp" function mein

    /\*

    kyunki "parameter" is of "datatype of class T"

    and we have passed "x and y" which are "int"

    therefore we have decided the "datatype for class T"

    \*/

    cout << x << endl

         << y;

    return 0;

}

In case there are two functions present

One asks “specific data type” as a parameter

And the other asks for “T data type” as a parameter

which we will decide further (the data type of class T)

Then that function will run whose

Parameters value(data type) will match exactly with the parameters value(datatype)

Given when calling the function

#include <iostream>

using namespace std;

void func(int a){

cout<<"I am first func() "<<a<<endl;

}

template<class T>

void func(T a){

cout<<"I am templatised func() "<<a<<endl;

}

int main()

{

func(4); //in this case “int a” waala function will run (as the parameters matches)

return 0;

}

1. VECTORS

A vector in C++ is a sequence container that can store elements of the same data type, and it provides dynamic sizing, meaning its size can change dynamically.

Syntax: vector<data\_type> vector\_name;

#include<iostream>

#include<vector>

// header file used to use "vector"

using namespace std;

void display(vector <int> v)

// we cannot simply mention "v", we need to mention "vector<int>v"(according to my knowledge rn..)

{

    for (int i = 0; i < v.size(); i++)

    {

        cout<<v[i]<<" ";

        // we have written " " just to provide space between two elements in the output

    }

    // basically displaying the "elements of vector"

    cout<<endl;

};

int main(int argc, char const \*argv[])

{

    vector <int> vec1;

    int element,size;

    cout<<"Enter size of vector"<<endl;

    cin>>size;

    for (int i = 0; i < size; i++)

    {

        cout<<"Enter the element to add to this vector";

        cin>>element;

        vec1.push\_back(element);

        // Vectors have a method(function), "push\_back()", to insert elements in it from the rear end.

        /\*

        "vec1" ka jo "function" hai "push\_back" uss "function" ko humne

        "parameter" provide kiya hai "element"

        basically "push\_back" items add karta hai "vector" mein from rear end

        toh ab vo "element" add karega given as a "parameter"

        \*/

    }

    display(vec1);

    // "display function" wants "vector" as "parameter"

    return 0;

}

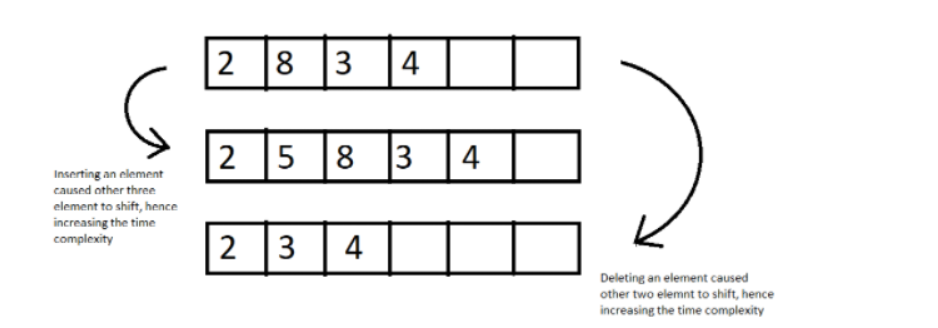
SIZE VECTOR

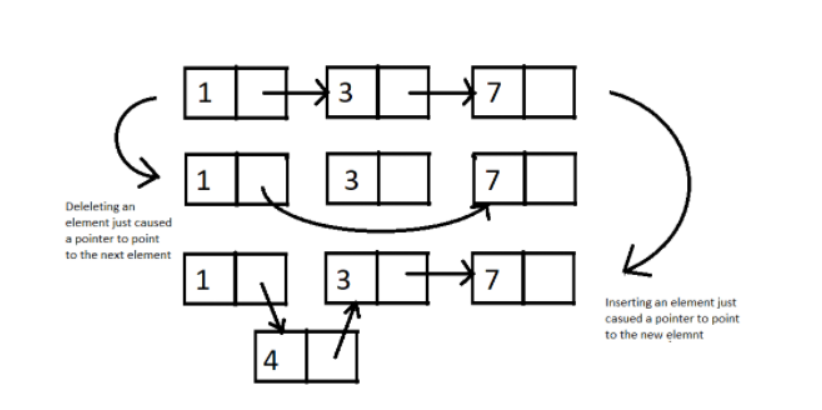
 vector <int> vec1(4)

in this we decide the “size of vector”

1. LIST

An array stores the elements in a contiguous manner in which inserting some element calls for a shift of other elements, which is time taking. But in a list, we can simply change the address the pointer is pointing to. I’ll show you how these work via an illustration.





We define a list iterator

syntax:

list<int> :: iterator it;

#include<iostream>

#include<list>

using namespace std;

int main(int argc, char const \*argv[])

{

    list <int> list1;

    list1.push\_back(5);

    list1.push\_back(7);

push\_back works the same for “list” as for “vector”

    list1.push\_back(9);

    list1.push\_back(11);

    list<int>::iterator iter;

    // it is kind of "pointer" which points(and thus iterates) "member of list"

    iter=list1.begin();

    // here it says the "pointer iter" points to the "beginning" of the "list l1"

    cout<<\*iter<<" ";

    // print whatever the "pointer iter" is storing

    iter++;

    // the "pointer iter" moves to the next member of "list"

    cout<<\*iter<<" ";

    iter++;

    cout<<\*iter<<" ";

    iter++;

    cout<<\*iter<<" ";

    return 0;

}

1. MAP(associative array)

a map stores a key of some data type and its corresponding values of some data type

 For example: a teacher wants to store the marks of students which in future can be accessed by their names. Here, keys are the student names, and their marks are the corresponding values

#include<iostream>

#include<map>

#include<string>

using namespace std;

int main(int argc, char const \*argv[])

{

    map <string,int> marks;

    marks["Happy"]=99;

    // "names" are the "key" and "marks" are the "corresponding value"

    marks["Harshi"]=95;

    // this is the way to make elements in "map"

    marks["Tushu"]=96;

    marks["Bawri"]=97;

 map <string,int> :: iterator iter;

//  iterator(same as list)

 for (iter=marks.begin(); iter!=marks.end(); iter++)

 /\*

 "iter" starting from "beginning of marks"

 will run till "iter" doesn't reach "end of marks"

 till then "iter" will keep adding by 1

 \*/

 {

    cout<<(\*iter).first<<" "<<(\*iter).second<<"\n";

    /\*

    print the "first item(key) of iter(i.e name here)"

    and then

    print the "second item(corresponding value) of iter(i.e marks here)"

    \*/

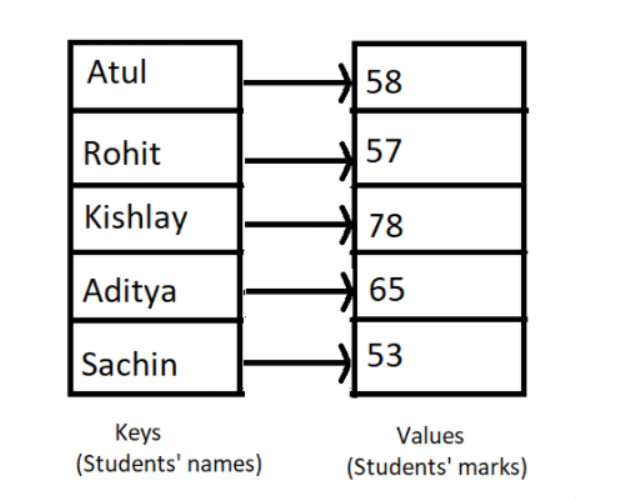
 }

    return 0;

}

(there is something wrong with the output

Specifically in the output the order of names of item)



 Syntax:

map <data\_type\_of\_key, data\_type\_of\_value> variable\_name;

1. FUNCTION OBJECTS

C plus plus .com

A function object is a function wrapped in a class so that it is available as an object.

That is, we can then use a function as an object.

Be sure to include the header file < functional> before you do anything else.

#include <iostream>

#include <functional>

#include <algorithm>

using namespace std;

int main(int argc, char const \*argv[])

/\*

FUNCTION OBJECTS(FUNCTOR):function wrapped in a class so that it is

available like an object

\*/

{

    int arr[] = {0, 1, 2, 3, 4, 5};

    sort(arr, arr + 6, greater<int>());

    // we don't need to understand rn how "sort" works

    /\*

     Suppose we want to sort this array in ascending order.

     So we’ll include a header file <algorithm>

     and write the syntax of the sort object

     syntax: sort(address of first element, address of last element);

     \*/

    for (int i = 0; i < 6; i++)

    {

        cout << arr[i] << endl;

    }

    return 0;

}