**Week 1(01/04/2022)**

1. a) Write a program to generate the following sequence

1

1 2

1 2 3

1 2 3 4

Program:

#include<iostream>

using namespace std;

int main()

{

int i,j,n;

cout<<"enter the number";

cin>>n;

for(i=1;i<=n;i++)

{

for(j=1;j<i;j++)

{

cout<<j;

}

cout<<"\n";

}

}

1. Write a program which uses function to swap two integers and two float numbers by using reference variable

#include<iostream>

using namespace std;

void swap1(int &x, int &y)

{

int temp;

temp = x;

x = y;

y = temp;

}

void swap2(float &x, float &y)

{

float temp;

temp = x;

x = y;

y = temp;

}

int main()

{

int a, b;

float c,d;

cout<<"Enter the value of a: ";

cin>>a;

cout<<"Enter the value of b: ";

cin>>b;

cout<<"Enter the value of c: ";

cin>>c;

cout<<"Enter the value of d: ";

cin>>d;

swap1(a, b);

swap(c,d);

cout<<endl<<"After swapping: ";

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

cout<<endl<<"c= "<<c<<" and d= "<<d;

return 0;

}

1. Write a program that demonstrates default arguments:

**Default arguments**: A default argument is a value in the function declaration automatically assigned by the compiler if the calling function does not pass any value to that argument.

**Following are the rules of declaring default arguments –**

* The values passed in the default arguments are not constant. These values can be overwritten if the value is passed to the function. If not, the previously declared value retains.
* During the calling of function, the values are copied from left to right.
* All the values that will be given default value will be on the right.

**Example**

* **void function(int x, int y, int z = 0)**Explanation - The above function is valid. Here z is the value that is predefined as a part of the default argument.
* **Void function(int x, int z = 0, int y)**  
  Explanation - The above function is invalid. Here z is the value defined in between, and it is not accepted.

Program:

#include<iostream>

using namespace std;

int sum(int x, int y, int z=0, int w=0)

{

return (x + y + z + w);

}

int main()

{

cout << sum(10, 15) << endl;

cout << sum(10, 15, 25) << endl;

cout << sum(10, 15, 25, 30) << endl;

return 0;

}

**Explanation**

In the above program, we have called the sum function three times.

* Sum(10,15):  
  When this function is called, it reaches the definition of the sum. There it initializes x to 10 y to 15, and the rest values are zero by default as no value is passed. And all the values after sum give 25 as output.
* Sum(10, 15, 25):  
  When this function is called, x remains 10, y remains 15, the third parameter z that is passed is initialized to 25 instead of zero. And the last value remains 0. The sum of x, y, z, w, is 50 which is returned as output.
* Sum(10, 15, 25, 30):  
  In this function call, there are four parameter values passed into the function with x as 10, y as 15, z is 25, and w as 30. All the values are then summed up to give 80 as the output.

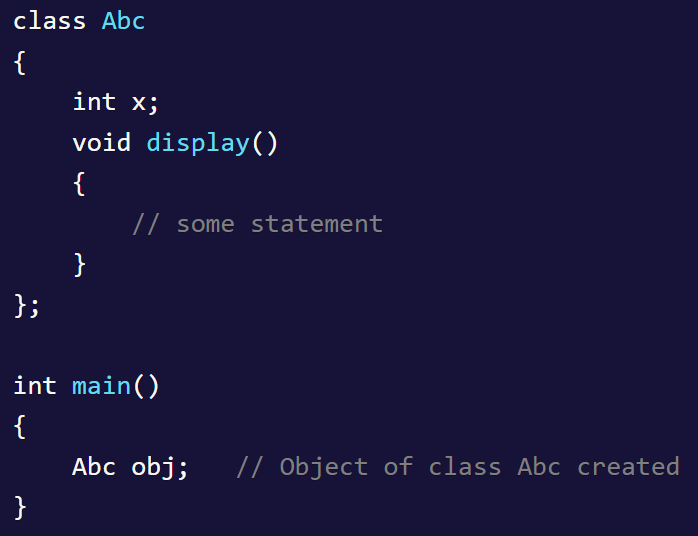
**WEEK -2(01/04/2022)**

2. a) Write a program Illustrating Class Declarations, Definition, and Accessing Class Members.

Class is a user defined data type, which holds its own [data members](https://www.studytonight.com/cpp/accessing-data-members.php) and member functions, which can be accessed and used by creating instance of that class.The [variables](https://www.studytonight.com/cpp/variables-scope-details.php) inside class definition are called as data members and the functions are called member functions.

1. Class name must start with an uppercase letter(Although this is not mandatory). If class name is made of more than one word, then first letter of each word must be in uppercase. *Example*, class Study, class StudyTonight etc
2. Classes contain, data members and member functions, and the access of these data members and variable depends on the [access specifiers](https://www.studytonight.com/cpp/access-control.php) (discussed in next section).
3. Class's member functions can be defined inside the class definition or outside the class definition.
4. Class in C++ are similar to structures in [C](https://www.studytonight.com/c/overview-of-c.php), the only difference being, class defaults to private access control, where as structure defaults to public.
5. All the features of OOPS, revolve around classes in C++. Inheritance, Encapsulation, Abstraction etc.

Example:



Program:

#include <iostream>

using namespace std;

class Sample

{

public:

int a;

int b;

void display()

{

cout << "a =: " << a<<endl;

cout << "b =: " << b;

}

};

int main() {

Sample s;

s.a = 100;

s.b = 200;

s.display();

return 0;

}

b)Write a program to illustrate default constructor, parameterized constructor and copy constructor, destructors for a class

**Constructor**: In C++, constructor is a special method which is invoked automatically at the time of object creation. It is used to initialize the data members of new object generally. The constructor in C++ has the same name as class or structure.

There can be two types of constructors in C++.

* Default constructor: A constructor which has no argument is known as default constructor. It is invoked at the time of creating object.

#include <iostream>

**using** **namespace** std;

**class** Employee

 {

**public**:

        Employee()

        {

            cout<<"Default Constructor Invoked"<<endl;

        }

};

**int** main()

{

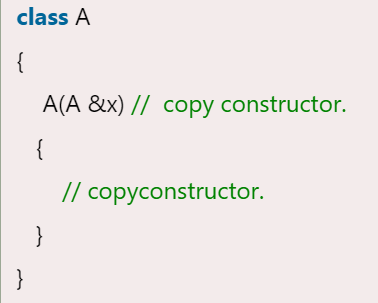
    Employee e1; //creating an object of Employee

    Employee e2;

**return** 0;

}

* Parameterized constructor: A constructor which has parameters is called parameterized constructor. It is used to provide different values to distinct objects.
* Copy constructor: A Copy constructor is an **overloaded** constructor used to declare and initialize an object from another object.



program:

#include<iostream>

using namespace std;

class Sample

{

private:

int x;

public:

Sample()

{

x=10;

cout<<x<<endl;

}

Sample (int a)

{

x = a;

}

Sample( Sample &b)

{

x = b.x;

}

~ Sample()

{

cout<<"Destructor called"<<endl;

}

int display( )

{

return x;

}

};

int main()

{

Sample c;

Sample c1(2);

Sample c2(c1);

cout << "c1 = " << c1.display() << endl;

cout << "c2 = " << c2.display() << endl;

return 0;

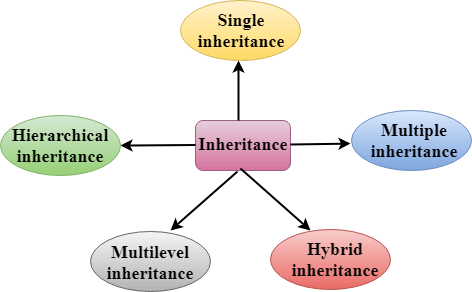
}

**WEEK-3 (08/04/2022)**

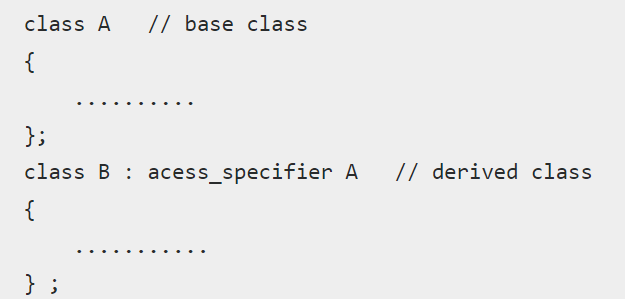
**3. a) Write a program that illustrates the following forms of inheritances**

**Ī) Single II) Multiple III) Multilevel IV) Hierarchical**

**Inheritance**: In C++, inheritance is a process in which one object acquires all the properties and behaviours of its parent object automatically. In such way, you can reuse, extend or modify the attributes and behaviours which are defined in other class.



**Single inheritance: Single inheritance** is defined as the inheritance in which a derived class is inherited from the only one base class.



**Program:**

#include<iostream>

using namespace std;

class Animal{

public:

void eat()

{

cout<<"Animal can eat anything"<<"\n";

}

public:

void sleep()

{

cout<<"Animal can sleep for long time"<<"\n";

}

};

class Der\_ani : public Animal

{

public:

void bark()

{

cout<<"Animal can bark"<<"\n";

}

};

int main()

{

Der\_ani da;

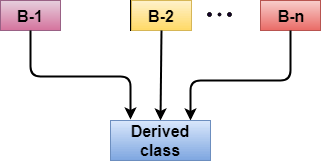
da.bark();

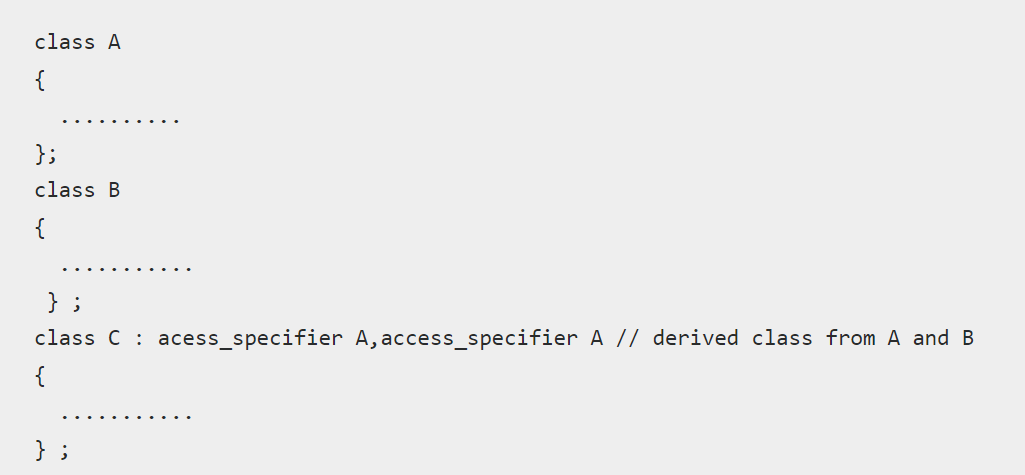
da.sleep();

da.eat();

}

**Multiple Inheritance**: **Multiple inheritance** is the process of deriving a new class that inherits the attributes from two or more classes.





**Program:**

#include <iostream>

using namespace std;

class A

{

public:

int x;

void getx()

{

cout << "enter value of x: "; cin >> x;

}

};

class B

{

public:

int y;

void gety()

{

cout << "enter value of y: "; cin >> y;

}

};

class C : public A, public B

{

public:

void sum()

{

cout << "Sum = " << x + y;

}

};

int main()

{

C obj1;

obj1.getx();

obj1.gety();

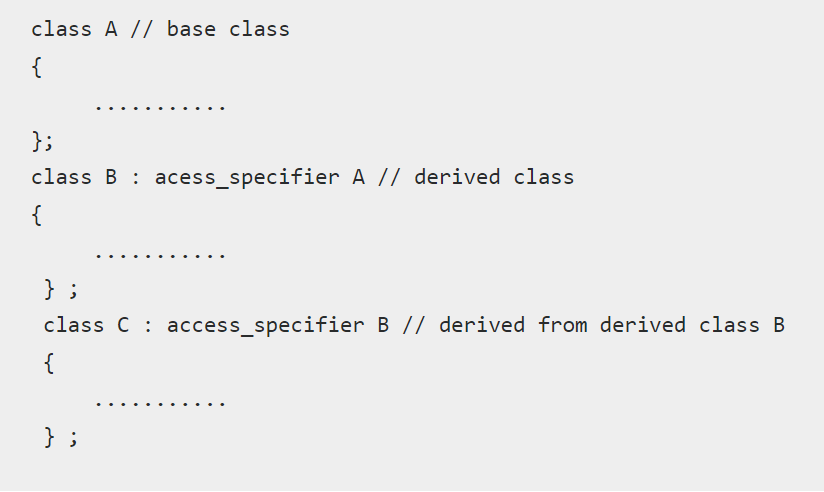
obj1.sum();

return 0;

}

**Multilevel**: **Multilevel inheritance** is a process of deriving a class from another derived class.

C++ Inheritance



**Program:**

#include <iostream>

using namespace std;

class base //single base class

{

public:

int x;

void getdata()

{

cout << "Enter value of x= "; cin >> x;

}

};

class derive1 : public base // derived class from base class

{

public:

int y;

void readdata()

{

cout << "\nEnter value of y= "; cin >> y;

}

};

class derive2 : public derive1 // derived from class derive1

{

private:

int z;

public:

void indata()

{

cout << "\nEnter value of z= "; cin >> z;

}

void product()

{

cout << "\nProduct= " << x \* y \* z;

}

};

int main()

{

derive2 a; //object of derived class

a.getdata();

a.readdata();

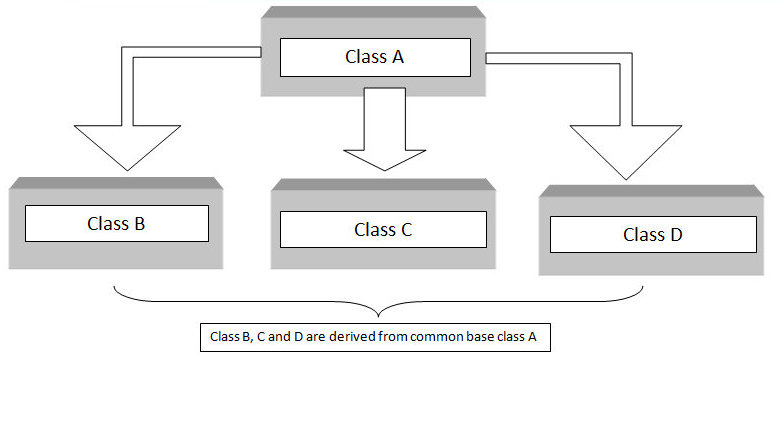
a.indata();

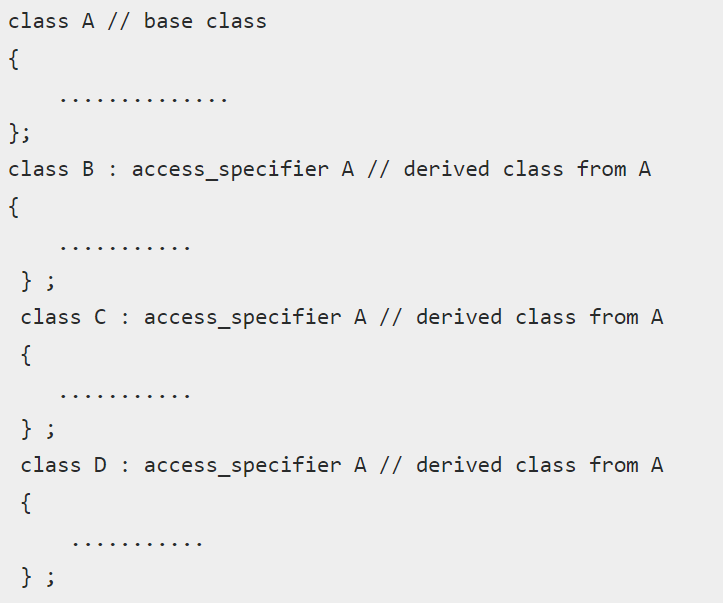
a.product();

return 0;

}

**Hierarchical** : more number of derived classes can be created from one base class





**Program:**

#include <iostream>

using namespace std;

class A

{

public:

int x, y;

void getdata()

{

cout << "\nEnter value of x and y:\n"; cin >> x >> y;

}

};

class B : public A

{

public:

void product()

{

cout << "\nProduct= " << x \* y;

}

};

class C : public A

{

public:

void sum()

{

cout << "\nSum= " << x + y;

}

};

int main()

{

B obj1;

C obj2;

obj1.getdata();

obj1.product();

obj2.getdata();

obj2.sum();

return 0;

}

**b) Create multiple objects for the class and observe the order in which constructors and destructors are called.**

**Program:**

#include<iostream>

using namespace std;

class Emp{

public:

Emp()

{

cout<<"constructor1 created \n";

}

~Emp()

{

cout<<"distructor1 \n";

}

};

class Sam{

public:

Sam()

{

cout<<"constructor2 created \n";

}

~Sam()

{

cout<<"destructor2\n";

}

};

int main()

{

Emp e1;

Emp e2;

Sam s1;

Sam s2;

}

WEEK 4 (08/04/2022)

**4 a) Write a program to use pointers for both base and derived classes and call the member functions.**

**PROGRAM:**

#include<iostream>

using namespace std;

class Base

{

public:

int var\_base;

void display(){

cout<<var\_base<<endl;

}

};

class Derived:public Base

{

public:

int var\_der;

void display(){

cout<<var\_base<<endl;

cout<<var\_der<<endl;

}

};

int main()

{

Base \*bp;

Derived dobj;

bp=&dobj;

bp->var\_base=34;

bp->display();

bp->var\_base=3400;

bp->display();

Derived \*dp;

dp=&dobj;

dp->var\_base=9448;

dp->var\_der=98;

dp->display();

bp->display();

}

**b) Write a program that demonstrates function overloading, operator overloading, overriding**

**Function overloading:**

* Function overloading is a feature of object oriented programming where two or more functions can have the same name but different parameters.
* When a function name is overloaded with different jobs it is called Function Overloading.
* In Function Overloading “Function” name should be the same and the arguments should be different.
* Function overloading can be considered as an example of polymorphism feature in C++.

**Example :**

**int test ( )**

**int test ( int a)**

**float test(double a)**

**int test(int a, double b)**

**program:**

#include <iostream>

using namespace std;

// function with 2 parameters

void display(int a, double b)

{

cout << "Integer number: " << a;

cout << " and double number: " << b << endl;

}

// function with double type single parameter

void display(double a)

{

cout << "Double number: " << a << endl;

}

// function with int type single parameter

void display(int a)

{

cout << "Integer number: " << a << endl;

}

int main()

{

int a ;

cin>>a;

double b;

cin>>b;

display(a);

display(b);

display(a, b);

return 0;

}

**Operator overloading:**

* Using **operator overloading** in C++, you can specify more than one meaning for an operator in one scope. The purpose of operator overloading is to provide a special meaning of an operator for a user-defined data type.
* With the help of operator overloading, you can redefine the majority of the C++ operators. You can also use operator overloading to perform different operations using one operator.

The advantage of Operators overloading is to perform different operations on the same operand.

**Operator that cannot be overloaded are as follows:**

* Scope operator (::)
* Sizeof
* member selector(.)
* member pointer selector(\*)
* ternary operator(?:)

Syn:

return\_type class\_name  : : operator  symbol(argument\_list)

{

     // body of the function.

}

**Rules for Operator Overloading**

C++ has tried to bridge-in the gap between the user-defined data-type variables and built-in data-type variables, by allowing us to work on both type of variables in a same way by the concept of **operator overloading**.   
  
**Operator overloading** is to provide a *special meaning or redefinition* to an operator(*unary or binary*), so that it could work on *user-defined* data-type objects just in the same way as it works on *built-in* data-type type variables.

For example, we could add two *built-in* data type **int** variables by just using the **+** operator between these two int variables.

**Ex:**

**int a=10,b=20;**

**int result=a+b;**

And, C++ also allows us to use **+** operator on two objects of *user-defined* data-type by using the concept of operator overloading using ***member function* or a *non-member*** **friend function.**

**Ex: ob3=obj1+obj2;**

* Existing operators can only be overloaded, but the new operators cannot be overloaded.
* The overloaded operator contains atleast one operand of the user-defined data type.
* We cannot use friend function to overload certain operators. However, the member function can be used to overload those operators.
* When unary operators are overloaded through a member function take no explicit arguments, but, if they are overloaded by a friend function, takes one argument.
* When binary operators are overloaded through a member function takes one explicit argument, and if they are overloaded through a friend function takes two explicit arguments.

**Rules to be followed for operator overloading:-**

1. Only existing operators can be overloaded.
2. Overloaded operators must have at least one operand that is of user defined operators
3. 3.We cannot change basic meaning of an operator.
4. Overloaded operator must follow minimum characteristics that of original operator
5. When using binary operator overloading through member function, the left hand operand must be an object of relevant class.

The number of arguments in the overloaded operator’s arguments list depends

* 1. Operator function must be either member function or friend function.
  2. If operator function is a friend function then it will have one argument for unary operator & two arguments for binary operator
  3. If operator function is a member function then it will have Zero argument for unary operator & one arguments for binary operator

**Example:**

#include<iostream>

using namespace std;

class Sample

{

int a;

public:

Sample()

{

a=10;

}

void display()

{

cout<<"Number is="<<a;

}

void operator ++( )

{

++a;

}

};

int main()

{

Sample s;

++ s;

s.display();

return 0;

}

Operator Overloading can be done by using **three approaches**, they are 

1. Overloading unary operator.
2. Overloading binary operator.
3. **Unary Operator Overloading** : An unary operator means, an operator which works on single operand. For example, ++ is an unary operator, it takes single operand (c++). So, when overloading an unary operator, it takes no argument (because object itself is considered as argument).

**Syntax for Unary Operator (Inside a class)**

return-type operator operatorsymbol( )

{

//body of the function

}

**Syntax for Unary Operator definition (Outside a class)**

return-type classname::operator operatorsymbol( )

{

//body of the function

}

#include<iostream>

using namespace std;

class Sample

{

int a;

public:

Sample()

{

a=10;

}

void display()

{

cout<<"Number is="<<a;

}

void operator ++( )

{

++a;

}

};

int main()

{

Sample s;

++ s;

s.display();

return 0;

}

1. **Binary Operator Overloading** : In binary operator overloading function, there should be one argument to be passed. It is overloading of an operator operating on two operands(one is object itself and other one is passed argument).

**Syntax for Binary Operator (Inside a class)**

return-type operator operatorsymbol(argument)

{

//body of the function

}

**Example:**

#include<iostream>

using namespace std;

class Sample

{

int n;

public:

Sample()

{

}

Sample(int a)

{

n=a;

}

void display()

{

cout<<"Number="<<n;

}

Sample operator +(Sample s2)

{

Sample s3;

s3.n=n+s2.n;

return s3;

}

};

int main()

{

Sample s1(10);

Sample s2(20);

Sample s3;

s3=s1+s2;

s3.display();

return 0;

}

**Ex: implement overloading assignment operator**

#include<iostream>

using namespace std;

class Sample

{

int a;

public:

Sample(int x)

{

a=x;

}

Sample operator =(Sample s1)

{

return Sample(s1.a);

}

void display()

{

cout<<"a value="<<a;

}

};

int main()

{

//clrscr();

Sample s1(100);

Sample s2=s1;

s2.display();

return 0;

}

**Function overriding:**

// C++ program to demonstrate function overriding

#include <iostream>

using namespace std;

class Base {

public:

void print() {

cout << "Base Function" << endl;

}

};

class Derived : public Base {

public:

void print() {

cout << "Derived Function" << endl;

}

};

int main() {

Derived derived1;

derived1.print();

return 0;

}

### # C++ Access Overridden Function to the Base Class

#include <iostream>

using namespace std;

class Base {

public:

void print() {

cout << "Base Function" << endl;

}

};

class Derived : public Base {

public:

void print() {

cout << "Derived Function" << endl;

}

};

int main() {

Derived derived1, derived2;

derived1.print();

// access print() function of the Base class

derived2.Base::print();

return 0;

}

WEEK – 5

5 a) Write a program that demonstrates friend functions, inline functions,

**Friend function:**

* If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.
* By using the keyword friend compiler knows the given function is a friend function.
* For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Syntax:

Class class\_name{

Friend returntype functionname(arguments)

}

**Example 1:**

#include<iostream>

using namespace std;

class Sample

{

int a,b;

public:

friend void display(Sample);

};

void display(Sample S)

{

S.a=10;

S.b=20;

cout<<S.a<<endl;

cout<<S.b;

}

int main()

{

Sample S;

display(S);

return 0;

}

**Example 2:**

#include<iostream>

using namespace std;

class Test2; //forward declaration

class Test1

{

int a;

public:

void geta()

{

cout<<"Enter a value";

cin>>a;

}

friend void big(Test1,Test2);

};

class Test2

{

int b;

public:

void getb()

{

cout<<"Enter b value";

cin>>b;

}

friend void big(Test1,Test2);

};

void big(Test1 t1, Test2 t2)

{

if(t1.a>t2.b)

{

cout<<"a is big"<<endl;

}

else{

cout<<"b is big";

}

}

int main()

{

Test1 t1;

Test2 t2;

t1.geta();

t2.getb();

big(t1,t2);

return 0;

}

**Friend class:**

#include<iostream>

using namespace std;

class Test1

{

int a,b,c;

public:

friend class Test2;

void getab()

{

cout<<"Enter a,b value";

cin>>a>>b;

}

};

class Test2

{

public:

void putab(Test1 t)

{

cout<<"a="<<t.a<<endl;

cout<<"b="<<t.b<<endl;

t.c=t.a+t.b;

cout<<"c is="<<t.c;

}

};

int main()

{

Test1 t;

Test2 t1;

t.getab();

t1.putab(t);

return 0;

}

**Inline function:**

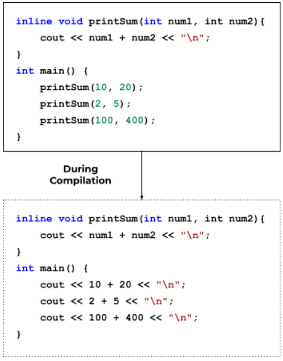
* If make a function as inline, then the compiler replaces the function calling location with the definition of the inline function at compile time.
* Any changes made to an inline function will require the inline function to be recompiled again because the compiler would need to replace all the code with a new code; otherwise, it will execute the old functionality.

Syntax:

Inline returntype functionname(arguments){

//code;

}



Example:

//normal function

#include<iostream>

using namespace std;

int addition(int,int);

int main()

{

int a,b;

cout<<"Enter a and b values";

cin>>a>>b;

cout<<"addition is ="<<addition(a,b);

return 0;

}

int addition(int a,int b)

{

return a+b;

}

**Inline example:**

#include<iostream>

using namespace std;

inline int addition(int,int);

int main()

{

int a,b;

cout<<"Enter a and b values";

cin>>a>>b;

cout<<"addition is ="<<addition(a,b);

return 0;

}

inline int addition(int a,int b)

{

return a+b;

}

**b) Write a program that demonstrates virtual, static functions**

**virtual functions:**

#include<iostream>

using namespace std;

class Base {

public:

virtual void print() {

cout<<"hai";

}

};

class Derived : public Base {

public:

void print() {

cout<<"hello";

}

};

int main() {

Derived derived1;

Base\* base1 = &derived1;

base1->print();

return 0;

}

**Example2:**

#include<iostream>

using namespace std;

class Base

{

int x=15;

public:

virtual void display()

{

cout<<"value of x is"<<x<<endl;

}

};

class Derv:public Base

{

int y=122;

public:

void display()

{

cout<<"value of y is"<<y<<endl;

}

};

int main()

{

Base \*b;

Derv d;

b=&d;

b->display();

return 0;

}

\* b is the base class pointer. The pointer can only access the base class members but not the members of the derived class. Although C++ permits the base pointer to point to any object derived from the base class, it cannot directly access the members of the derived class. Therefore, there is a need for virtual function which allows the base pointer to access the members of the derived class.

**Static functions:**

#include <iostream>

using namespace std;

void Test(){

static int x = 1;

x = ++x;

int y = 1;

y = ++y;

cout<<"x = "<<x<<"\n";

cout<<"y = "<<y<<"\n";

}

int main()

{

Test();

Test();

return 0;

}

**Example2:**

#include <iostream>

using namespace std;

class Example{

static int x;

public:

void function1(){

x++;

}

void function2(){

cout<<"x = "<<x<<"\n";

}

};

int Example :: x=10;

int main()

{

Example obj1, obj2, obj3;

cout<<"Initial value of x" <<"\n";

obj1.function2();

obj2.function2();

obj3.function2();

obj1.function1();

obj2.function1();

obj3.function1();

cout<<"Value of x after calling function1"<<"\n";

obj1.function2();

obj2.function2();

obj3.function2();

return 0;

}

**Example2:**

#include <iostream>

using namespace std;

class Example{

static int Number;

int n;

public:

void set\_n(){

n = ++Number;

}

void show\_n(){

cout<<"value of n = "<<n<<endl;

}

static void show\_Number(){

cout<<"value of Number = "<<Number<<endl;

}

};

int Example:: Number=99;

int main()

{

Example example1, example2;

example1.set\_n();

example2.set\_n();

example1.show\_n();

example2.show\_n();

example1.show\_Number();

example2.show\_Number();

return 0;

}

WEEK 6 (02/05/2022)

**6 a) Write a program which uses the concept of pass and return objects to functions.**

#include <iostream>

using namespace std;

class Student

{

public:

double marks;

Student(double m)

{

marks = m;

}

};

void calculateAverage(Student s1, Student s2)

{

double average = (s1.marks + s2.marks) / 2;

cout << "Average Marks = " << average;

}

int main()

{

Student s1(88.0), s2(56.0);

calculateAverage(s1, s2);

return 0;

}

**Return object from function:**

#include<iostream>

using namespace std;

class Student

{

public:

double marks1, marks2;

};

Student createStudent( )

{

Student s;

s.marks1 = 96.5;

s.marks2 = 75.0;

cout << "Marks 1 = " << s.marks1 << endl;

cout << "Marks 2 = " << s.marks2 << endl;

return s;

}

int main()

{

Student s1;

s1 = createStudent();

return 0;

}

**b) Write a program to create an array of objects.**

#include<iostream>

using namespace std;

class MyClass

{

int x;

public:

void set(int i)

{

x=i;

}

int get()

{

return x;

}

};

int main()

{

MyClass obs[4];

int i;

for(i=0;i<4; i++)

obs[i].set(i);

for(i=0;i< 4;i++)

cout <<"obs[" << i << "]: " << obs[i].get() << "\n";

return 0;

}

Week -7

An exception is a problem that arises during the execution of a program. A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C++ exception handling is built upon three keywords: **try, catch,** and **throw**.

* **throw** − A program throws an exception when a problem shows up. This is done using a **throw** keyword.
* **catch** − A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The **catch** keyword indicates the catching of an exception.
* **try** − A **try** block identifies a block of code for which particular exceptions will be activated. It's followed by one or more catch blocks.

Assuming a block will raise an exception, a method catches an exception using a combination of the **try** and **catch** keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch as follows –

try {

// protected code

} catch( ExceptionName e1 ) {

// catch block

} catch( ExceptionName e2 ) {

// catch block

} catch( ExceptionName eN ) {

// catch block

}

You can list down multiple **catch** statements to catch different type of exceptions in case your **try** block raises more than one exception in different situations.

**a) Write a program that handles Exceptions. Use a Try Block to Throw it and a Catch Block to Handle it Properly**

#include <iostream>

using namespace std;

int division(int a, int b)

{

if( b == 0 )

{

throw b;

}

return (a/b);

}

int main ()

{

int x,y,z;

cout<<"Enter x";

cin>>x;

cout<<"Enter y";

cin>>y;

try

{

z = division(x, y);

cout << z << endl;

}

catch (int x)

{

cout<< "Division by zero condition!"<< endl;

}

return 0;

}

**b) Write a Program to Demonstrate the Catching of All Exceptions**

**program:**

#include <iostream>

using namespace std;

void test(int x)

{

try

{

if(x==1)

throw x;

else

if(x==0)

throw 'x';

else

if(x==-1)

throw 1.0;

cout<<"end of try block"<<endl;

}

catch(char c)

{

cout<<"caught a character"<<endl;

}

catch(int m)

{

cout<<"caught an integer"<<endl;

}

catch(double d)

{

cout<<"caught a double"<<endl;

}

}

int main()

{

test(1);

test(0);

test(-1);

test(2); return 0;

}

(OR)

#include<iostream>

using namespace std;

void test(int x)

{

try

{

if(x==0) throw x; if(x==0) throw 'x'; if(x==-1) throw 1.0;

}

catch(...)

{

cout<<"caught exception"<<endl;

}

}

int main()

{

test(-1);

test(0);

test(1);

return 0;

}

**Week -8**

8. Write a Program to demonstrates user defined exceptions

We have to remember certain rules in order to write user-defined exception :

1. Always include exception header using pre-processor directive at the very first step.
2. The function which will return an exception string should have a return type of char followed by **\***,

**char\* what()**

**{**

**// codes here**

**}**

char is as return type because we will return a string.

1. Should have a try and catch block.

#include <iostream>

#include <exception>

using namespace std;

class newException : public exception

{

public:

const char \* what() const throw()

{

return "Division is not possible";

}

};

int main ()

{

int x, y;

cout << "Enter x: ";

cin >> x;

cout << "Enter y: ";

cin >> y;

try

{

if( y == 0 )

{

newException err;

throw err;

}

else

cout << x/y <<endl;

}

catch(exception& e)

{

cout << e.what();

}

return 0;

}

**Week 9**

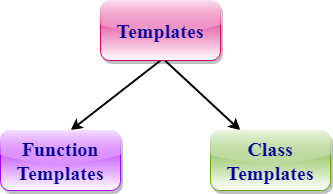
**9. Write a program to create a generic template for adding two integers and two float values and make use of the template to perform addition.**

**What is a template:**

A C++ template is a powerful feature added to C++. It allows you to define the generic classes and generic functions and thus provides support for generic programming. Generic programming is a technique where generic types are used as parameters in algorithms so that they can work for a variety of data types.

Templates can be represented in two ways:

* Function templates
* Class templates



**Function Templates:**

We can define a template for a function. For example, if we have an add() function, we can create versions of the add function for adding the int, float or double type values.

**Class Template:**

We can define a template for a class. For example, a class template can be created for the array class that can accept the array of various types such as int array, float array or double array.

## **Function Template**

* Generic functions use the concept of a function template. Generic functions define a set of operations that can be applied to the various types of data.
* The type of the data that the function will operate on depends on the type of the data passed as a parameter.
* For example, Quick sorting algorithm is implemented using a generic function, it can be implemented to an array of integers or array of floats.
* A Generic function is created by using the keyword template. The template defines what function will do.

### **Syntax of Function Template**

template < class Ttype>

ret\_type func\_name(parameter\_list)

{

    // body of function.

}

Where **Ttype**: It is a placeholder name for a data type used by the function. It is used within the function definition. It is only a placeholder that the compiler will automatically replace this placeholder with the actual data type.

**class**: A class keyword is used to specify a generic type in a template declaration.

Example 1:

#include <iostream>

using namespace std;

template <class A>

void fun(A a, A b)

{

cout<<a<<endl;

cout<<b;

}

int main()

{

fun(15,34);

}

Example 2:

#include <iostream>

using namespace std;

template <class T>

T findmax(T x, T y)

{

return (x>y)?x:y;

}

int main()

{

int a,b;

cout<<"enter a value"<<endl;

cin>>a;

cout<<"enter b value"<<endl;

cin>>b;

cout<<"max of"<<findmax(a,b);

}

### **Function Templates with Multiple Parameters:**

We can use more than one generic type in the template function by using the comma to separate the list.

## **Syntax**

**template**<**class** T1, **class** T2,.....>

return\_type function\_name (arguments of type T1, T2....)

{

    // body of function.

}

In the above syntax, we have seen that the template function can accept any number of arguments of a different type.

Example:

#include <iostream>

using namespace std;

template <class A, class B>

void fun(A a, B b)

{

cout<<a<<endl;

cout<<b;

}

int main()

{

fun(15,34);

}

**Program with functions:**

#include <iostream>

using namespace std;

template <class T1, class T2>

void addition(T1 c,T2 d)

{

cout<<"addition of two number="<<c+d<<endl;

}

int main()

{

int a,b;

float c,d;

cout<<"Enter two integer variables";

cin>>a>>b;

cout<<"\n Enter tow float point variables";

cin>>c>>d;

addition(a,b);

addition(c,d);

addition(a,c);

}

**Class Template** can also be defined similarly to the Function Template. When a class uses the concept of Template, then the class is known as generic class.

## **Syntax**

**template**<**class** Ttype>

**class** class\_name

{

  .

  .

}

**Ttype** is a placeholder name which will be determined when the class is instantiated. We can define more than one generic data type using a comma-separated list. The Ttype can be used inside the class body.

Now, we create an instance of a class

class\_name<type> ob;

**where class\_name**: It is the name of the class.

**type**: It is the type of the data that the class is operating on.

**ob**: It is the name of the object.

Example:

#include <iostream>

using namespace std;

template<class T>

class A

{

public:

T n1;

T n2;

void add(T a,T b)

{

n1=a;

n2=b;

cout << "Addition of num1 and num2 : " << n1+n2;

}

};

int main()

{

A<int> d;

d.add(5,10);

return 0;

}

### **CLASS TEMPLATE WITH MULTIPLE PARAMETERS**

We can use more than one generic data type in a class template, and each generic data type is separated by the comma.

## **Syntax**

**template**<**class** T1, **class** T2, ......>

**class** class\_name

{

   // Body of the class.

}

Example:

#include <iostream>

using namespace std;

template<class T1,class T2>

class A

{

T1 a;

T2 b;

public:

A(T1 x,T2 y)

{

a=x;

b=y;

}

void add()

{

cout << "Addition of two numbers : " <<a+b<<endl ;

}

};

int main()

{

int a,b;

float x,y;

cout<<"Enter a and b values";

cin>>a>>b;

cout<<"Enter x and y values";

cin>>x>>y;

A<int,int>S(a,b);

A<float,float>B(x,y);

A<int,float>C(a,x);

A<float,int>D(y,b);

S.add();

B.add();

C.add();

D.add();

return 0;

}

**Week 10 (23/05/2022)**

10. Write a program to implement the matrix ADT using a class. The operations supported by this ADT are: a) Addition of two matrices. b) subtraction of two matrices. c) Multiplication of two matrices.

#include<iostream.h>

using namespace std;

struct matrixType{

    int matDimension;

    int matValues[10][10];

};

class MatrixADT{

    private:

        matrixType resultMatrix;

    public:

       //Member function declarations

        void intializeResultMatrix(int);

matrixType add(matrixType, matrixType);

        matrixType subtract(matrixType,matrixType);

        matrixType multiply(matrixType,matrixType);

        void printResult();

};

//Member functions of Matrix class to be defined here

matrixType MatrixADT::add(matrixType M1, matrixType M2)

{

            int i,j;

             for(i=0;i<M1.matDimension;i++)

     for(j=0;j<M2.matDimension;j++)

       {

                   resultMatrix.matValues[i][j]=M1.matValues[i][j]+M2.matValues[i][j];

                }

  return (M1);

}

matrixType MatrixADT::subtract(matrixType M1, matrixType M2)

{

            int i,j;

             for(i=0;i<M1.matDimension;i++)

     for(j=0;j<M2.matDimension;j++)

       {

                   resultMatrix.matValues[i][j]=M1.matValues[i][j]-M2.matValues[i][j];

                }

            return (M1);

}

matrixType MatrixADT::multiply(matrixType M1, matrixType M2)

{

            int i,j,k;

             for(i=0;i<M1.matDimension;i++)

     for(j=0;j<M2.matDimension;j++)

        for(k=0;k<M1.matDimension;k++)

       {

                   resultMatrix.matValues[i][j]+=M1.matValues[i][k]\*M2.matValues[k][j];

                }

             return (M1);

}

void MatrixADT::intializeResultMatrix(int dim)

{

int i,j;

resultMatrix.matDimension=dim;

             for(i=0;i<dim;i++)

     for(j=0;j<dim;j++)

       {

                   resultMatrix.matValues[i][j]=0;

                }

}

int main(){

    MatrixADT maX;

    matrixType M1, M2;

    char op;

    int dim;

    cin>>dim;

    cin>>op;

    M1.matDimension=M2.matDimension=dim;

    int i,j;

    for(i=0;i<dim;i++)

     for(j=0;j<dim;j++)

       cin>>M1.matValues[i][j];

    for(i=0;i<dim;i++)

     for(j=0;j<dim;j++)

       cin>>M2.matValues[i][j];

      maX.intializeResultMatrix(dim);

    switch(op)

    {

                 case '+' :  M1=maX.add(M1,M2);

                                                            break;

                 case '-' :  M1=maX.subtract(M1,M2);

                                                break;

                 case '\*' :   M1=maX.multiply(M1,M2);

                                                            cout<<"hello";

                                                            break;

             }

maX.printResult();

   getch();

            return (0);

}

void MatrixADT::printResult()

{

    int i,j;

    for (i=0;i<resultMatrix.matDimension;i++){

        for (j=0; j<resultMatrix.matDimension-1;j++){

            cout<<resultMatrix.matValues[i][j]<<" ";

        }

       cout <<resultMatrix.matValues[i][j]<<"\n";

    }

    cout <<"Done";

}

**WEEK – 11**

11. Accept two stacks as input from the user and perform operations on it using stack class available in Standard Template Library (STL).

**Standard Template Library:**

* Standard Template Library (STL) of C++ is a collection of template classes that provide data structures such as arrays, vectors, queue, etc.
* STL is a library consisting of containers, algorithms, functions and iterators.
* As STL consists of a collection of template classes, it’s a generalized library that is independent of data types.

**STL has three components:**

1. **Algorithms:**

* Algorithms are the methods or functions that act on containers.
* By using algorithms we can have methods to search, sort, modify, transform or initialize the contents of container class objects.
* Algorithms provided by STL have built-in functions that can directly operate on complex data structure instead of having to write the algorithms ourselves.

**For Example,** reverse() function in STL can be used to reverse the linked list.

**STL supports the following types of algorithms:**

* Searching algorithms
* Sorting algorithms
* Modifying or manipulating algorithms
* Non-modifying algorithms
* Numeric algorithms
* Min/Max algorithms

1. **Containers:**

* A container is a collection of objects of a particular type of data structure. In STL, we have various types of container classes like Array, vector, queue, deque, list, map, stack, set, etc.

The containers are subdivided into 3 types

* 1. **Sequence containers** : can be accessed in a sequential or linear manner

Example: Arrays, Vector, dequeue, List

* 1. **Associative containers**: Associative containers are containers that implement sorted data structures. These containers are fast to search.

Example: Map, MultiMap, Set, Multiset.

* 1. **Container Adopters**: Container adopters are sequential containers, however, they are implemented by providing a different interface.

Example: Stack, Queue, Priority queue

1. **Iterators**: Iterators are constructs that we use to traverse or step through containers in STL. Iterators are very important in STL as they act as a bridge between algorithms and containers.

**Iterators a re of following types:**

**Input Iterators:** Simplest and is used mostly in single-pass algorithms.

**Output Iterators:** Same as input iterators but not used for traversing.

**Bidirectional Iterators:** These iterators can move in both directions.

**Forward Iterators:** Can be used only in the forward direction, one step at a time.

**Random Access Iterators:** Same as pointers. Can be used to access any element randomly.

**Example1:**

# vectors:

#include <iostream>

#include <vector>

using namespace std;

int main()

{

vector<int> nums;

for (int a = 1; a <= 5; a++)

nums.push\_back(a);

cout << "Output from begin and end: ";

for (auto b = nums.begin(); b != nums.end(); ++b)

cout << \*b << " ";

cout << "\nOutput from cbegin and cend: ";

for (auto c = nums.rbegin(); c != nums.rend(); ++c)

cout << \*c << " ";

return 0;

}

**Example 2:**

**//stacks**

#include <iostream>

#include <stack>

using namespace std;

// function prototype for display\_stack utility

void display\_stack(stack<int> st);

int main() {

// create a stack of strings

stack<int> num;

for(int i=10;i<=15;i++)

{

num.push(i);

}

// push elements into the stack

cout << "Initial Stack: ";

// print elements of stack

display\_stack(num);

// removes "Blue" as it was inserted last

num.pop();

cout << "Final Stack: ";

// print elements of stack

display\_stack(num);

return 0;

}

// utility function to display stack elements

void display\_stack(stack<int> st)

{

while(!st.empty()) {

cout << st.top() << ", ";

st.pop();

}

cout << endl;

}

// stacks

#include <iostream>

#include<stack>

using namespace std;

int main()

{

stack<int> s;

for(int i=0;i<5;i++)

{

s.push(i);

}

cout<<"Size="<<s.size()<<"\n";

cout<<"Top="<<s.top()<<endl;

s.pop();

s.pop();

cout<<"Size="<<s.size()<<"\n";

cout<<"Top="<<s.top()<<endl;

if(s.empty())

cout<<"stack is empty";

else

cout<<"stack is not empty";

}

// Queue:

#include <iostream>

#include <queue>

using namespace std;

void display\_queue(queue<string> q);

int main() {

queue<string> animals;

animals.push("Cat");

animals.push("Dog");

animals.push("Fox");

cout << "Initial Queue: ";

display\_queue(animals);

animals.pop();

cout << "Final Queue: ";

display\_queue(animals);

return 0;

}

void display\_queue(queue<string> q) {

while(!q.empty()) {

cout << q.front() << ", ";

q.pop();

}

cout << endl;

}

Example:

//algorithms

#include <algorithm>

#include <iostream>

#include <vector>

#include <numeric> //For accumulate operation

using namespace std;

int main()

{

// Initializing vector with array values

int arr[] = {10, 20, 5, 23 ,42 , 15};

int n = sizeof(arr)/sizeof(arr[0]);

vector<int> vect(arr, arr+n);

cout << "Vector is: ";

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

cout << vect[i] << " ";

// Sorting the Vector in Ascending order

sort(vect.begin(), vect.end());

cout << "\nVector after sorting is: ";

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

cout << vect[i] << " ";

// Reversing the Vector

reverse(vect.begin(), vect.end());

cout << "\nVector after reversing is: ";

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

cout << vect[i] << " ";

cout << "\nMaximum element of vector is: ";

cout << \*max\_element(vect.begin(), vect.end());

cout << "\nMinimum element of vector is: ";

cout << \*min\_element(vect.begin(), vect.end());

// Starting the summation from 0

cout << "\nThe summation of vector elements is: ";

cout << accumulate(vect.begin(), vect.end(), 0);

return 0;

}