LAB MANUAL

JUNE 2, 2021

LAHORE GARRISON UNIVERSOTY

Table of Contents

LAB NO. 01

LOOP, IF ELSE AND SWITCH

Lab Objectives

Following are the lab objectives:

1. Refresh loop concepts
2. If else in C++
3. Switch in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

1. Introduction

Simple program in C++

```
#include<iostream.h>/*Header File*/
int main()/*Main Function*/
{
cout<<"\n*HELLO*\n";
/*Output Statements*/
}
```

C++ DATA TYPES

primary data type	**int, float, char, void**
user defined data type	**structure, union, class, enumerate on**
derived data type	**array, function, pointer, reference**

C++ VARIABLES SCOPE

- A scope is a region of the program and broadly speaking there are three places, where variables can be declared –
- Inside a function or a block which is called local variables,
- In the definition of function parameters which is called
- Formal parameters.
- Outside of all functions which is called global variables.

Local Variable

```
#include <iostream>
using namespace std;
int main() {
       // Local variable declaration:
       int a, b;
       int c;
       // actual initialization
       a = 10;
       b = 20;
       c = a + b;
       cout << c;
       return 0;
}
```

Global Variable

```
#include <iostream>
using namespace std;
// Global variable declaration:
int g;
int main() {
	// Local variable declaration:
	int a, b;
	// actual initialization
	a = 10;
	b = 20;
	g = a + b;
	cout << g;
	return 0;
}
```

C++ for Loops

In computer programming, loops are used to repeat a block of code.

For example, let's say we want to show a message 100 times. Then instead of writing the print statement 100 times, we can use a loop.

That was just a simple example; we can achieve much more efficiency and sophistication in our programs by making effective use of loops.

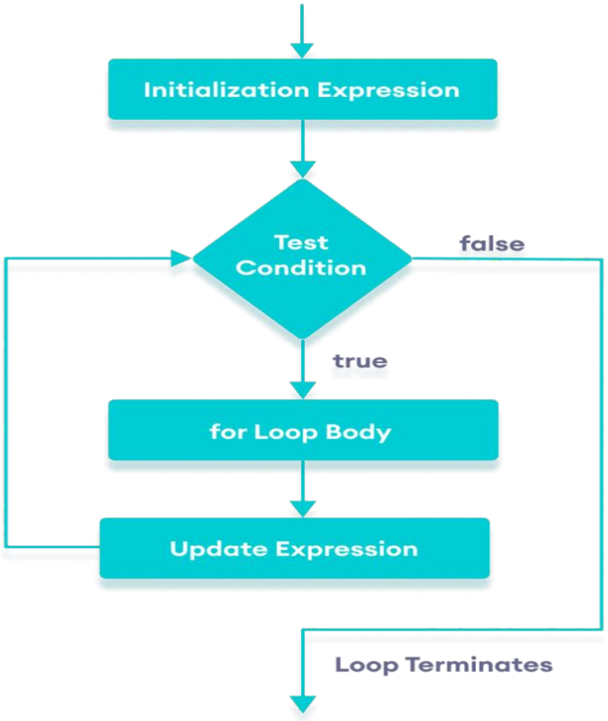
There are 3 types of loops in C++.

1. for loop
2. while loop
3. do...while loop

The syntax of for-loop is:

```
for (initialization; condition; update) {
 // body of-loop
}
```

Flowchart of for Loop in C++



Example 1: Printing Numbers From 1 to 5

```
#include <iostream>

using namespace std;

int main() {
        for (int i = 1; i <= 5; ++i) {
        cout << i  << " ";
    }
    return 0;
}
```

Output

```
1 2 3 4 5
```

Example 2: Display a text 5 times

```
// C++ Program to display a text 5 times

#include <iostream>

using namespace std;

int main() {
    for (int i = 1; i <= 5; ++i) {
        cout <<  "Hello World! " << endl;
    }
    return 0;
}
```

Output

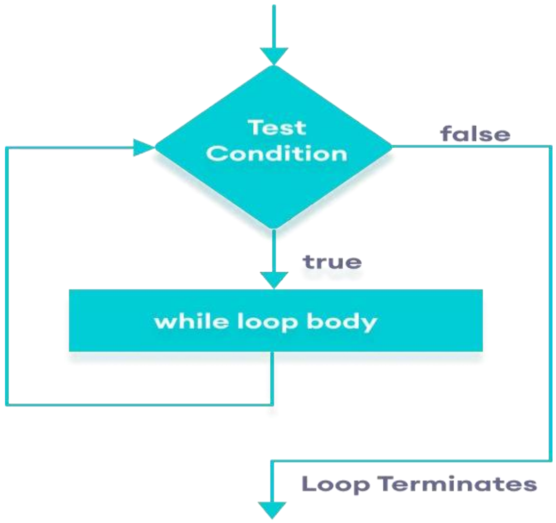
```
Hello World!
Hello World!
Hello World!
Hello World!
Hello World!
```

C++ while Loop

The syntax of the `while` loop is:

```
while (condition) {
    // body of the loop
}
```

Flowchart of while Loop



Example 1: Display Numbers from 1 to 5

```
// C++ Program to print numbers from 1 to 5

#include <iostream>

using namespace std;

int main() {
    int i = 1;

    // while loop from 1 to 5
    while (i <= 5) {
        cout << i  << " ";
        ++i;
    }

    return 0;
}
```

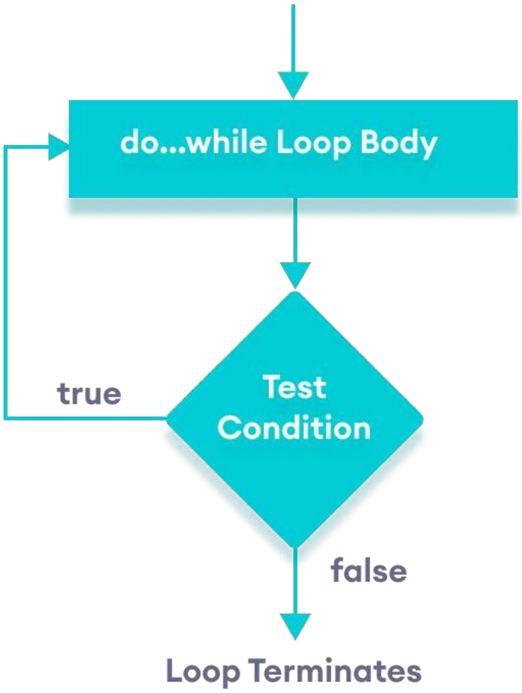
Output

```
1 2 3 4 5
```

C++ do...while Loop

```
do {
   // body of loop;
}
while (condition);
```

Flowchart of do...while Loop



Display Numbers from 1 to 5

```
// C++ Program to print numbers from 1 to 5

#include <iostream>

using namespace std;

int main() {
    int i = 1;
    // do...while loop from 1 to 5
    do {
        cout << i  << " ";
        ++i;
    }
    while (i <= 5);

    return 0;
}
```

Output

```
1 2 3 4 5
```

C++ if, if...else and Nested if...else

In computer programming, we use the if statement to run a block code only when a certain condition is met.

For example, assigning grades (A, B, C) based on marks obtained by a student.

if the percentage is above 90, assign grade A

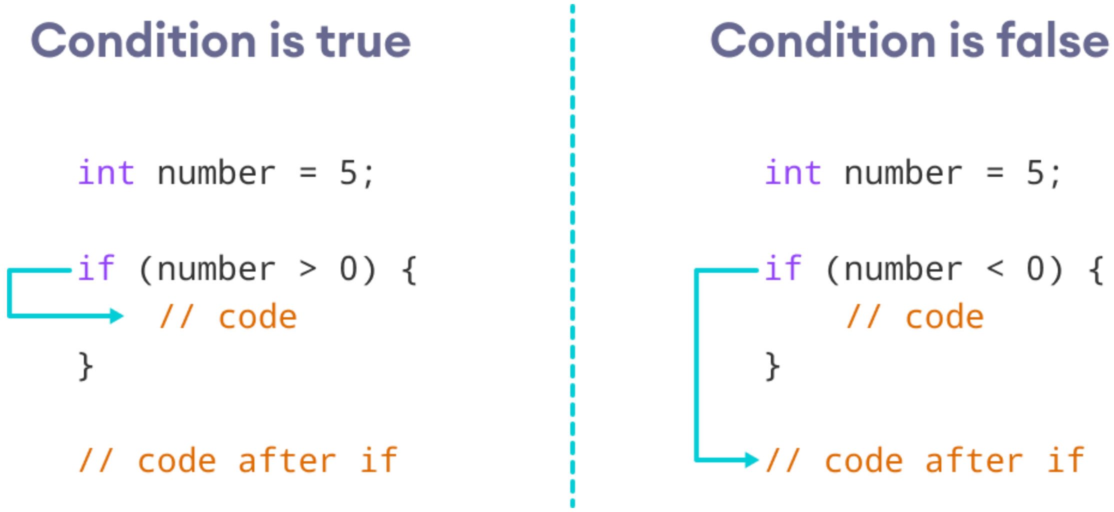
if the percentage is above 75, assign grade B

if the percentage is above 65, assign grade C

C++ if Statement

The syntax of the `if` statement is:

```
if (condition) {
   // body of if statement
}
```



```
int number = 5;

if (number > 0) {
    // code
}

// code after if
```

```
int number = 5;

if (number < 0) {
    // code
}

// code after if
```

C++ if Statement

```
// Program to print positive number entered by the user
// If the user enters a negative number, it is skipped

#include <iostream>
using namespace std;

int main() {
    int number;

    cout << "Enter an integer: ";
    cin >> number;

    // checks if the number is positive
    if (number > 0) {
        cout << "You entered a positive integer: " << number << endl;
    }
    cout << "This statement is always executed.";
    return 0;
}
```

Output

```
Enter an integer: 5
```

```
You entered a positive number: 5
This statement is always executed.
```

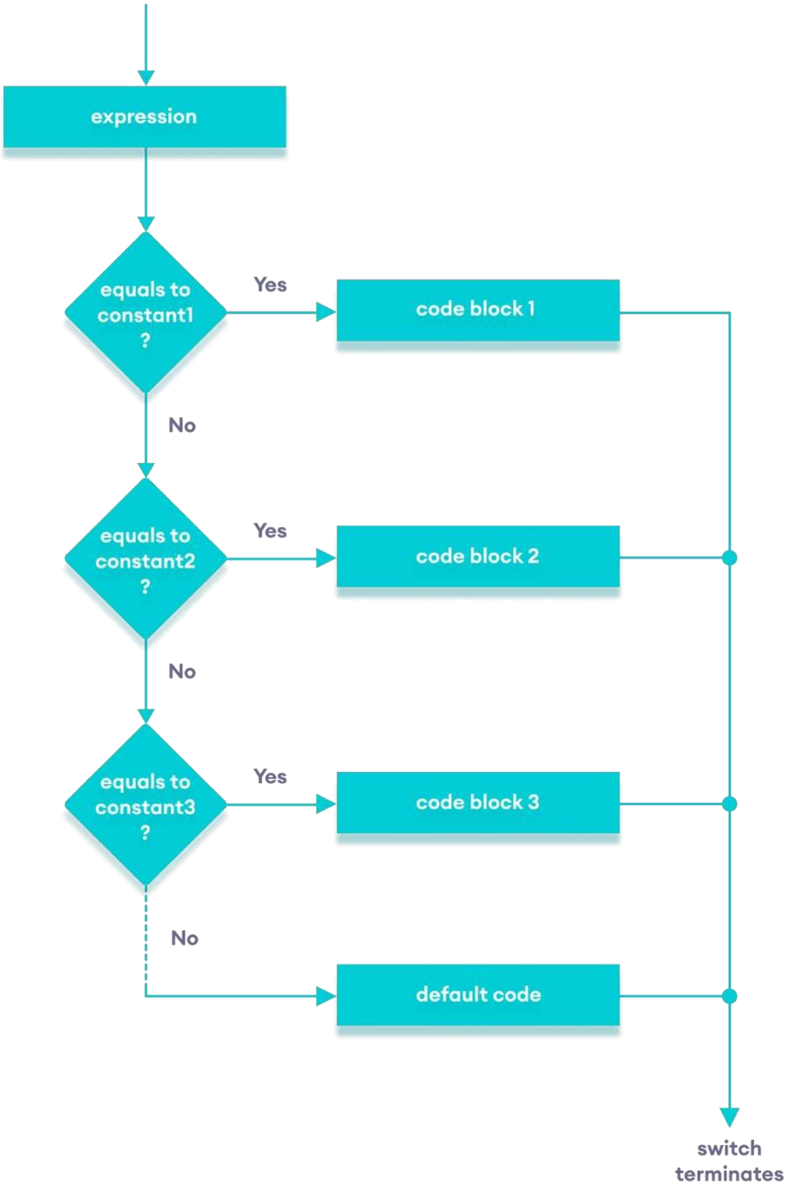
C++ switch..case Statement

The `switch` statement allows us to execute a block of code among many alternatives.

The syntax of the `switch` statement in C++ is:

```
switch (expression)  {
    case constant1:
        // code to be executed if
        // expression is equal to constant1;
        break;

    case constant2:
        // code to be executed if
        // expression is equal to constant2;
        break;
        .
        .
        .
    default:
        // code to be executed if
        // expression doesn't match any constant
}
```

Flowchart of switch Statement



Create a Calculator using the switch Statement

```
// Program to build a simple calculator using switch Statement
#include <iostream>
using namespace std;

int main() {
    char oper;
    float num1, num2;
    cout << "Enter an operator (+, -, *, /): ";
    cin >> oper;
    cout << "Enter two numbers: " << endl;
    cin >> num1 >> num2;

    switch (oper) {
        case '+':
            cout << num1 << " + " << num2 << " = " << num1 + num2;
            break;
        case '-':
            cout << num1 << " - " << num2 << " = " << num1 - num2;
            break;
        case '*':
            cout << num1 << " * " << num2 << " = " << num1 * num2;
            break;
        case '/':
            cout << num1 << " / " << num2 << " = " << num1 / num2;
            break;
        default:
            // operator is doesn't match any case constant (+, -, *, /)
            cout << "Error! The operator is not correct";
            break;
    }

    return 0;
}
```

Output

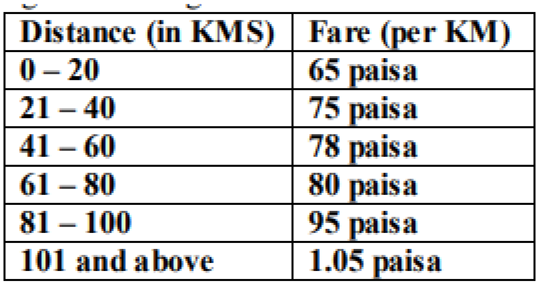
```
Enter an operator (+, -, *, /): +
Enter two numbers:
2.3
4.5
2.3 + 4.5 = 6.8
```

Lab Tasks

i. Write a C++ program to check whether the given number is even or odd.

```
/*
	Task 1: Number is even or odd
*/
#include <iostream>
using namespace std;

int main(){
	int num;
	cout << "Enter number: ";
	cin >> num;

	if(num%2==0)
		cout << "This number is even." << endl;
	else
		cout << "This number is odd." << endl;

	return 0;
}
```

ii. Write a Program to calculate the fare for the passengers traveling in a bus. When a Passenger enters the bus, the conductor asks "What distance will you travel?" On knowing distance from passenger (as an approximate integer), the conductor mentions the fare to the passenger according to following criteria.



Distance (in KMS)	Fare (per KM)
0 – 20	65 paisa
21 – 40	75 paisa
41 – 60	78 paisa
61 – 80	80 paisa
81 – 100	95 paisa
101 and above	1.05 paisa

```
/*
    Task 1: Number is even or odd
*/
#include <iostream>
using namespace std;

int main(){
    float distance,fees;
    cout << "Enter Distance: ";
    cin >> distance;

    if(distance <= 20){
        fees = distance*0.65;
    }
    else if(distance <= 40){
        fees = distance*0.75;
    }
    else if(distance <= 60){
        fees = distance*0.78;
    }
    else if(distance <= 80){
        fees = distance*0.80;
    }
    else if(distance <= 100){
        fees = distance*0.95;
    }
    else if(distance > 100){
        fees = distance*1.05;
    }

    cout << "Fees: " << fees << endl;

    return 0;
}
```

LAB NO. 02

FUNCTIONS

Lab Objectives

Following are the lab objectives:

1. Functions pass by value
2. Function pass by reference

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

C++ Functions

A function is a block of code that performs a specific task. Suppose we need to create a program to create a circle and color it. We can create two functions to solve this problem:

- a function to draw the circle
- a function to color the circle

Dividing a complex problem into smaller chunks makes our program easy to understand and reusable.
There are two types of function:

1. Standard Library Functions: Predefined in C++
2. User-defined Function: Created by users

C++ User-defined Function

C++ allows the programmer to define their own function. A user-defined function groups code to perform a specific task and that group of code is given a name (identifier). When the function is invoked from any part of the program, it all executes the codes defined in the body of the function.

C++ Function Declaration

The syntax to declare a function is:

```
  returnType functionName (parameter1, parameter2,...) {
// function body
  }
```

Here's an example of a function declaration.

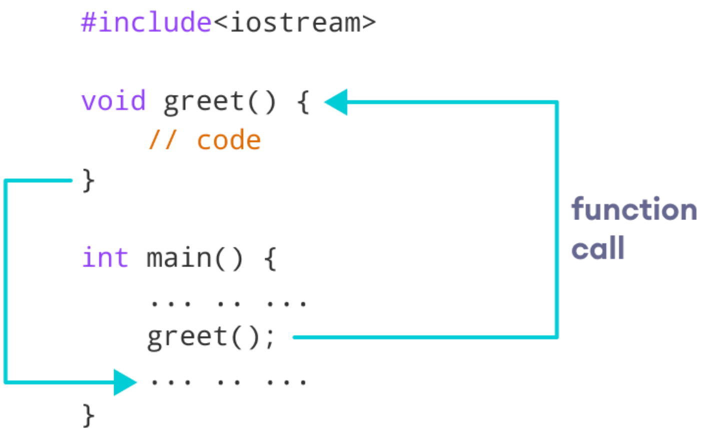
```
  // function declaration
  void greet() {
cout << "Hello World";
  }
```

Calling a Function

In the above program, we have declared a function named greet(). To use the greet() function, we need to call it.

Here's how we can call the above greet() function.

```
int main() {
    // calling a function
    greet();  }
```



```
#include<iostream>

void greet() {
    // code
}

int main() {
    ... .. ...
    greet();
    ... .. ...
}
```

Example 1: Display a Text

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

  // declaring a function
  void greet() {
cout << "Hello there!";
  }

  int main(){

// calling the function
greet();

return 0;
  }
```

Output

```
Hello there!
```

Example 2: Function with Parameters

```
// program to print a text

#include <iostream>
using namespace std;

// display a number
displayNum(int n1, float n2) { cout <<
"The int number is " << n1;
cout << "The double number is " << n2;
}

int main() {

      int num1 = 5;
      double num2 = 5.5;

// calling the function displayNum(num1,
num2);

return 0;
}
```

Output

```
The int number is 5
The double number is 5.5
```

Types of User-defined Functions in C++

For better understanding of arguments and return in functions, user-defined functions can be categorised as:

i. Function with no argument and no return value
ii. Function with no argument but return value
iii. Function with argument but no return value
iv. Function with argument and return value

Example 1: No arguments passed and no return value

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

  void prime();

  int main()
  {
// No argument is passed to prime()
prime();
return 0;
  }


  // Return type of function is void because value is not returned.
  void prime()
  {

int num, i, flag = 0;

cout << "Enter a positive integer enter to check: "; cin >>
num;

for(i = 2; i <= num/2; ++i)
{
         if(num % i == 0)
         {
             flag = 1;
             break;
         }
}

if (flag ==1)
{
         cout << num << " is not a prime number.";
}
else
{
         cout << num << " is a prime number.";
}
```

```
  }
```

Example 2: No arguments passed but a return value

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

  int prime();

  int main()
  {
int num, i, flag = 0;

// No argument is passed to prime() num =
prime();
for (i = 2; i <= num/2; ++i)
{
          if (num%i == 0)
          {
              flag = 1;
              break;
          }
}

if (flag ==1)
{
          cout<<num<<" is not a prime number.";
}
else
{
          cout<<num<<" is a prime number.";
}
return 0;
  }

  // Return type of function is int
  int prime()
  {
int n;

printf("Enter a positive integer to check: ");
cin >> n; return

n;
   }
```

Example 3: Arguments passed but no return value

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

   void prime(int n);

   int main()
   {
int num;
cout << "Enter a positive integer to check: "; cin >>
num;

// Argument num is passed to the function prime()
prime(num);
return 0;
   }

   // There is no return value to calling function. Hence, return type of function
   is void. */
   void prime(int n)
   {
int i, flag = 0;
for (i = 2; i <= n/2; ++i)
{
             if (n%i == 0)
             {
                  flag = 1;
                  break;
             }
}

if (flag ==1)
{
             cout << n << " is not a prime number.";
}
else {
            cout << n << " is a prime number.";
}
   }
```

Example 4: Arguments passed and a return value.

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

   int prime(intn);

   int main()
   {
int num, flag = 0;
cout << "Enter positive integer to check: "; cin >>
num;

// Argument num is passed to check() function flag =
prime(num);

if(flag == 1)
     cout << num << " is not a prime number."; else
     cout<< num << " is a prime number.";
return 0;
   }

   /* This function returns integer value.  */
   int prime(int n)
   {
int i;
for(i = 2; i <= n/2; ++i)
{
            if(n % i  == 0)
                 return 1;
}

return 0;
   }
```

C++ Function Overloading

In C++, two functions can have the same name if the number and/or type of arguments passed is different.

These functions having the same name but different arguments are known as overloaded functions. For example:

```
 // same name different arguments
 int test() { }
 int test(int a) { } float
 test(double a) { }
 int test(int a, double b) { }
```

Function Overloading using Different Number of Parameters

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

 // function with 2 parameters
display(int var1, double var2) { cout <<
"Integer number: " << var1;
cout << " and double number: " << var2 << endl;
 }

 // function with double type single parameter
 void display(double var) {
cout << "Double number: " << var << endl;
 }

 // function with int type single parameter
 void display(int var) {
cout << "Integer number: " << var << endl;
 }

 int main() {

int a = 5; double b =
5.5;
// call function with int type parameter display(a);

// call function with double type parameter display(b);

// call function with 2 parameters
display(a, b);

return 0;
  }
```

Output

```
Integer number: 5
Float number: 5.5
Integer number: 5 and double number: 5.5
```

Lab Tasks

i. Write a program which will ask the user to enter his/her marks (out of 100). Define a function that will display grades according to the marks entered as below:

Marks	Grade
91-100	AA
81-90	AB
71-80	BB
61-70	BC
51-60	CD
41-50	DD
<=40	Fail

```
/*
Grading Program
*/
#include <iostream>
using namespace std;

string grading(float marks)
{
string grade;
if(marks<=100 && marks>=91){
     grade = "A";
}
else if(marks<=90 && marks>=81){
     grade = "AB";
```

```
   }
   else if(marks<=80 && marks>=71){
        grade = "BB";
   }
   else if(marks<=70 && marks>=61){
        grade = "BC";
   }
   else if(marks<=60 && marks>=51){
        grade = "CD";
   }
   else if(marks<=50 && marks>=41){
        grade = "DD";
   }
   else if(marks<=40){
        grade="Fail";
   }
   return grade;
   }
   int main(){
   int marks;
   cout << "Enter Makrs: ";
   cin >> marks;
   cout << "Grade is " << grading(marks) << endl;

   return 0;
   }
```

ii. Write a program that performs arithmetic division. The program will use two integers, a and b (obtained by the user) and will perform the division a/b, store the result in another integer c and show the result of the division using cout. In a similar way, extend the program to add, subtract, multiply, do modulo and power using integers a and b. Modify your program so that when it starts, it asks the user which type of calculation it should do, then asks for the 2 integers, then runs the user selected calculation and outputs the result in a user friendly formatted manner.

```
/*
    Calculator
*/
#include <iostream>
using namespace std;

int add(int a, int b){
    return a+b;
}
int mul(int a, int b){
    return a*b;
}
int sub(int a, int b){
    return a-b;
}
int divide(int a, int b){
    return a/b;
}
int mod(int a, int b){
    return a%b;
}
int pow(int a, int b){
    int ans=1;
    for(int i=1;i<=b;i++)
        ans*=a;
    return ans;
}

int main(){
    char symb;
    int a , b;
    cout << "Operation: ";
    cin >> symb;

    switch(symb){
        case '+':
            cout << "Enter A: ";
            cin >> a;
            cout << "Enter B: ";
            cin >> b;
            cout << "Sum:" << add(a,b);
            break;
        case '-':
            cout << "Enter A: ";
            cin >> a;
            cout << "Enter B: ";
            cin >> b;
            cout << "Sub:" << sub(a,b);
            break;
        case '*':
            cout << "Enter A: ";
            cin >> a;
            cout << "Enter B: ";
            cin >> b;
            cout << "Mul:" << mul(a,b);
            break;
        case '/':
            cout << "Enter A: ";
            cin >> a;
            cout << "Enter B: ";
            cin >> b;
            cout << "Divide:" << divide(a,b);
            break;
        case '%':
            cout << "Enter A: ";
            cin >> a;
            cout << "Enter B: ";
            cin >> b;
            cout << "Mod:" << mod(a,b);
            break;
        case '^':
            cout << "Enter A: ";
            cin >> a;
            cout << "Enter B: ";
            cin >> b;
            cout << "Pow:" << pow(a,b);
            break;
    }
    return 0;
}
```

LAB NO. 03

Basics of STRUCTURES IN C++

Lab Objectives

Following are the lab objectives:

1. Structures in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

Structure in C++

Structure is a collection of variables of different data types under a single name. It is similar to a class in that, both holds a collection of data of different data types.
For example: You want to store some information about a person: his/her name, citizenship number and salary. You can easily create different variables name, citNo, salary to store these information separately.
However, in the future, you would want to store information about multiple persons. Now, you'd need to create different variables for each information per person: name1, citNo1, salary1, name2, citNo2, salary2
You can easily visualize how big and messy the code would look. Also, since no relation between the variables (information) would exist, it's going to be a daunting task.
A better approach will be to have a collection of all related information under a single name Person, and use it for every person. Now, the code looks much cleaner, readable and efficient as well.

This collection of all related information under a single name Person is a structure.

How to declare a structure in C++ programming?

The struct keyword defines a structure type followed by an identifier (name of the structure).
Then inside the curly braces, you can declare one or more members (declare variables inside curly braces) of that structure. For example:

```
struct Person
{
    char name[50];
    int age;
    float salary;
};
```

How to define a structure variable?

Once you declare a structure person as above. You can define a structure variable as:

```
Person bill;
```

Here, a structure variable bill is defined which is of type structure Person.

How to access members of a structure?

The members of structure variable is accessed using a **dot (.)** operator.

Suppose, you want to access age of structure variable bill and assign it 50 to it. You can perform this task by using following code below:

```
bill.age = 50;
```

Lab Tasks

Q1. C++ Program to assign data to members of a structure variable and display it.

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

struct Person
{
    char name[50];
    int age;
    float salary;
};

int main()
{
    Person p1;

    cout << "Enter Full name: ";
    cin.get(p1.name, 50);
    cout << "Enter age: ";
    cin >> p1.age;
    cout << "Enter salary: ";
    cin >> p1.salary;

    cout << "\nDisplaying Information." << endl;
    cout << "Name: " << p1.name << endl;
    cout <<"Age: " << p1.age << endl;
    cout << "Salary: " << p1.salary;

    return 0;
}
```

Output

```
Enter Full name: Magdalena Dankova
Enter age: 27
Enter salary: 1024.4

Displaying Information.
Name: Magdalena Dankova
Age: 27
Salary: 1024.4
```

Q2. Example for Nested Structure

```
#include<iostream.h>
        struct Address
        {
                char HouseNo[25];
                char City[25];
                char PinCode[25];
        };
        struct Employee
        {
            int Id;
            char Name[25];
            float Salary;
            struct Address Add;
        };
        void main()
        {
                int i;
                Employee E;
                cout << "\n\tEnter Employee Id : ";
                cin >> E.Id;
                cout << "\n\tEnter Employee Name : ";
                cin >> E.Name;
                cout << "\n\tEnter Employee Salary : ";
                cin >> E.Salary;
                cout << "\n\tEnter Employee House No : ";
                cin >> E.Add.HouseNo;
                cout << "\n\tEnter Employee City : ";
                cin >> E.Add.City;
                cout << "\n\tEnter Employee House No : ";
                cin >> E.Add.PinCode;
                cout << "\nDetails of Employees";
                cout << "\n\tEmployee Id : " << E.Id;
                cout << "\n\tEmployee Name : " << E.Name;
                cout << "\n\tEmployee Salary : " << E.Salary;
                cout << "\n\tEmployee House No : " << E.Add.HouseNo;
                cout << "\n\tEmployee City : " << E.Add.City;
                cout << "\n\tEmployee House No : " << E.Add.PinCode;
        }
    Output :
                Enter Employee Id : 101
                Enter Employee Name : Suresh
                Enter Employee Salary : 45000
                Enter Employee House No : 4598/D
                Enter Employee City : Delhi
                Enter Employee Pin Code : 110056
                Details of Employees
                Employee Id : 101
                Employee Name : Suresh
                Employee Salary : 45000
                Employee House No : 4598/D
                Employee City : Delhi
                      Employee Pin Code : 110056
```

Q3. Write a program that declares a structure to store date. Declare an instance of this structure to represent date of birth. The program should read the day, month and year values of birth date and display date of birth in dd/mm/yy format.

```
#include<iostream>
using namespace std;
struct Date
{
        int day;
        int month;
        int year;
};
void main()
{
        Date birthDate;
        cout << "Enter day of birth"; //enter day
        cin >> birthDate.day; //display day
        cout << "Enter month of birth";      //enter month
        cin >> birthDate.month;      //display month
        cout << "Enter year of birth"; //enter year
        cin >> birthDate.year; //display year
```

```
          cout << "your date of birth is: " << birthDate.day << "/" << birthDate.month <<
    "/"<< birthDate.year << "\n";
          system("pause");
    }
```

Q4. Write a C++ program that declares a structure to store the distance covered by a player along with the minutes and seconds taken to cover the distance. The program should input the records of two players and then display the record of the winner player.

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

struct Player{
    float distance=0;
    float minutes=0, seconds=0;
};

int main(){
    Player p1,p2;
    cout << "####### Player1 ########" << endl;
    cout << "Enter Distance: ";
    cin >> p1.distance;
    cout << "Enter minutes: ";
    cin >> p1.minutes;
    cout << "Enter second: ";
    cin >> p1.seconds;
    float t1 = p1.minutes + p1.seconds/10;

    cout << "####### Player2 ########" << endl;
    cout << "Enter Distance: ";
    cin >> p2.distance;
    cout << "Enter minutes: ";
    cin >> p2.minutes;
    cout << "Enter second: ";
    cin >> p2.seconds;
    int t2 = p2.minutes + p2.seconds/10;

    if(p1.distance>=p2.distance && p1.minutes < p2.minutes ){
        cout << "Player 1 is winner!" << endl;
        cout << "Distance: " << p1.distance << endl;
        cout << "Time taken: " << t1;
    }
    else if(p1.distance==p2.distance && p1.minutes == p2.minutes){
        cout << "Draw!" << endl;
    }
    else{
        cout << "Player 2 is winner!" << endl;
        cout << "Distance: " << p1.distance << endl;
        cout << "Time taken: " << t2;

    }
    return 0;
}
```

Home Activity

Q5. There is a structure called employee that holds information like employee code, name, date of joining. Write a program to create an array of the structure and enter some data into it. Then ask the user to enter current date. Display the names of those employees whose tenure is 3 or more than 3 years according to the given current date.

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

struct Employee{
    int code;
    int year;
    string name;
};

int main(){
    Employee emp_arr[3];
    int year_inp;
    for(int index=0; index<3; index++){
        cout << "### EMPLOYEE " << index+1 << " ###" << endl;
        cout << "ID: ";
        cin >> emp_arr[index].code;
        cout << "Enter Name: ";
        cin >> emp_arr[index].name;
        cout << "Joing Year: ";~
        cin >> emp_arr[index].year;
    }
    cout << "###############################"
    cout << "Enter Year: ";
    cin >> year_inp;
    for(int i=0; i<3; i++){
        if((year_inp - emp_arr[i].year)>=3){
            cout << "ID: " << emp_arr[i].code << endl;
            cout << "Name: " << emp_arr[i].name << endl;
            cout << "Year of Joining: " << emp_arr[i].year << endl;
        }
    }
    return 0;
}
```

Q6. Write a C++ program that declare a structure to store income, tax rate and tax of a person. The program defines an array of structure to store the record of five persons. It inputs income and tax rate of five person and then displays the tax payable.

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

struct Tax{
    float income,taxRate, totalTax;
};

int main(){
    Tax taxArray[5];

    for(int index=0; index<5; index++){
        cout << "Income: ";
        cin >> taxArray[index].income;
        cout << "Tax Rate: ";
        cin >> taxArray[index].taxRate;
        taxArray[index].totalTax = taxArray[index].income * taxArray[index].taxRate
/100;
    }

    for(int index=0; index<5; index++){
        cout << "Tax " << index+1 << ": " << taxArray[index].totalTax << endl;
    }

    return 0;
}
```

LAB NO. 04

STRUCTURES IN C++

Lab Objectives

Following are the lab objectives:

1. Array of Structures
2. Structure to functions
3. Pointer to structure

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

Array of Structure

Structure is collection of different data type. An object of structure represents a single record in memory, if we want more than one record of structure type, we have to create an array of structure or object. As we know, an array is a collection of similar type, therefore an array can be of structure type.

Syntax for declaring structure array

```
struct struct-name
{
        datatype var1;
        datatype var2;
        - - - - - - - - - -
        - - - - - - - - - -
        datatype varN;
};
struct-name obj [ size ];
```

Example for declaring structure array

```
               #include<iostream.h>
              struct Employee
              {
int Id;
char Name[25]; int Age;
long Salary;
              };
              void main()
              {
int i;
Employee Emp[ 3 ];                              //Statement 1
for(i=0;i<3;i++)
{
            cout << "\nEnter details of " << i+1 << " Employee";
               cout << "\n\tEnter Employee Id : ";
               cin >> Emp[i].Id;
               cout << "\n\tEnter Employee Name : ";
               cin >> Emp[i].Name;
               cout << "\n\tEnter Employee Age : ";
               cin >> Emp[i].Age;
               cout << "\n\tEnter Employee Salary : ";
               cin >> Emp[i].Salary;
}
cout << "\nDetails of Employees"; for(i=0;i<3;i++)
cout << "\n"<< Emp[i].Id <<"\t"<< Emp[i].Name <<"\t"
                << Emp[i].Age <<"\t"<< Emp[i].Salary;
        }
```

Array within Structure

As we know, structure is collection of different data type. Like normal data type, It can also store an array as well.

Syntax for array within structure

```
struct struct-name
{
    datatype var1;                  // normal variable
    datatype array [size];          // array variable
    - - - - - - - - - -
    - - - - - - - - - -
    datatype varN;
};
```

```
        struct-name obj;
```

Example for array within structure

```
        struct Student
        {
int Roll;
char Name[25];
int Marks[3];                    //Statement 1 : array of marks
int Total;
float Avg;
        };
        void main()
        {
int i;
Student S;
cout << "\n\nEnter Student Roll : "; cin >>
S.Roll;
cout << "\n\nEnter Student Name : "; cin >>
S.Name;
S.Total = 0; for(i=0;i<3;i++)
{
                cout << "\n\nEnter Marks " << i+1 << " : ";
                cin >> S.Marks[i];
                S.Total = S.Total + S.Marks[i];
}
S.Avg = S.Total / 3;

cout << "\nRoll : " << S.Roll;
cout << "\nName : " << S.Name; cout <<
"\nTotal : " << S.Total; cout << "\nAverage : "
<< S.Avg;
            }
```

Structure and Function

Using function we can pass structure as function argument and we can also return structure from function.

Passing structure as function argument

Structure can be passed to function through its object therefore passing structure to function or passing structure object to function is same thing because structure object represents the structure. Like normal variable, structure variable (structure object) can be pass by value or by references/addresses.

Passing Structure by Value

In this approach, the structure object is passed as function argument to the definition of function, here object is representing the members of structure with their values.

Example for passing structure object by value

```
            #include<iostream.h>
            struct Employee
            {
int Id;
char Name[25]; int Age;
long Salary;
            };
            void Display(struct Employee);
            void main()
            {
Employee Emp = {1,"Kumar",29,45000};
Display(Emp);
            }
            void Display(struct Employee E)
            {
                cout << "\n\nEmployee Id : " << E.Id;
                cout << "\nEmployee Name : " << E.Name;
                cout << "\nEmployee Age : " << E.Age;
                cout << "\nEmployee Salary : " << E.Salary;
            }
```

Output :

```
Employee Id : 1 Employee Name :
Kumar Employee Age : 29 Employee
Salary : 45000
```

Passing Structure by Reference

In this approach, the reference/address structure object is passed as function argument to the definition of function.

Example for passing structure object by reference

```
    #include<iostream.h>
    struct Employee
    {
int Id;
char Name[25]; int Age;
long Salary;
    };
    void Display(struct Employee*);
    void main()
```

```
        {
Employee Emp = {1,"Kumar",29,45000};
Display(&Emp);
        }

        void Display(struct Employee *E)
        {
               cout << "\n\nEmployee Id : " << E->Id;
               cout << "\nEmployee Name : " << E->Name;
               cout << "\nEmployee Age : " << E->Age;
               cout << "\nEmployee Salary : " << E->Salary;
        }
```

      Output :

```
Employee Id : 1 Employee Name :
Kumar Employee Age : 29 Employee
Salary : 45000
```

Function Returning Structure

Structure is user-defined data type, like built-in data types structure can be return from function.

Example for passing structure object by reference

```
         #include<iostream.h>
        struct Employee
        {
int Id;
char Name[25]; int Age;
long Salary;
        };
        Employee Input();       //Statement 1
        void main()
        {
Employee Emp;
Emp = Input();
               cout << "\n\nEmployee Id : " << E.Id;
               cout << "\nEmployee Name : " << E.Name;
               cout << "\nEmployee Age : " << E.Age;
               cout << "\nEmployee Salary : " << E.Salary;
        }
        Employee Input()
        {
Employee E;
               cout << "\nEnter Employee Id : ";
               cin >> E.Id;
               cout << "\nEnter Employee Name : ";
               cin >> E.Name;
               cout << "\nEnter Employee Age : ";
               cin >> E.Age;
               cout << "\nEnter Employee Salary : ";
               cin >> E.Salary;
return E;                      //Statement  2
        }
```

Output :

```
Enter Employee Id : 10
Enter Employee Name : Ajay Enter
Employee Age : 25 Enter Employee Salary :
15000
```

Employee Id : 10 Employee

Name : Ajay Employee Age : 25

Employee Salary : 15000

Pointers to Structure

It's possible to create a pointer that points to a structure. It is similar to how pointers pointing to native data types like int, float, double, etc. are created. Note that a pointer in C++ will store a memory location.

Example:

```
#include
<iostream>
using
namespace std;
struct Length
{
        int meters;
        float centimeters;
};
int main()
{
        Length
        *ptr, l;
        ptr = &l;
        cout << "Enter
        meters: "; cin >>
        (*ptr).meters;
        cout << "Enter centimeters:
        "; cin >>
        (*ptr).centimeters;
        cout << "Length = " << (*ptr).meters << " meters " << (*ptr).centimeters << "
centimeters";

        return 0;
}
```

Lab Tasks

Q1. Write a structure to store the roll no, name and age (between 10 to 15) of students (more than 5). Store the information of the students.

1 - Write a function to print the names of all the students having age 14.
2 - Write another function to print the names of all the students having even roll no. 3 - Write another function to display the details of the student whose roll no is given

(i.e. roll no. entered by the user).

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

struct Student{
    string name;
    int age;
    int roll_no;
};

void age_14(Student st[],int size);
void even_roll(Student st[], int size);
void search(Student st[], int size, int search_rn);

int main(){
    Student st[6];
    // Data
    st[0] = {"Zafeer",14,50};
    st[1] = {"Abdullah",11,51};
    st[2] = {"Kainat",14,52};
    st[3] = {"Armghan",14,54};
    st[4] = {"Ahsan",12,58};
    st[5] = {"Mahad",10,65};

    // Function Calls
    age_14(st,6);
    even_roll(st,6);
    search(st,6,54);

    return 0;
}
// Function Definition
void age_14(Student st[],int size){
    cout << "Age 14" << endl;
    for(int index=0; index<size; index++){
            if(st[index].age==14)
                cout << st[index].name << endl;
    }
}
void even_roll(Student st[], int size){
    cout << "Even Roll Number" << endl;
    for(int index=0; index<size; index++){
        if(st[index].age%2==0)
```

```
            cout << st[index].name << endl;
    }
}
void search(Student st[], int size, int search_rn){
    cout << "Search Student using roll number." << endl;
    for(int index=0; index<size; index++){
        if(st[index].roll_no==search_rn){
            cout << "Name: " << st[index].name << endl;
            cout << "Roll No.: " << st[index].roll_no << endl;
            cout << "Age: " << st[index].age << endl;
        }
    }
}
```

Q2. Write a structure to store the name, account number and balance of customers (more than 10) and store their information.

1 - Write a function to print the names of all the customers having balance less than 20k.
2 - Write a function to add 1000k in the balance of all the customers having more than 30k in their balance and then print the incremented value of their balance.

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

struct Customer{
    string name;
    int account_no;
    long balance;
};
void customer_balance(Customer ct[], int size);
void add_cash(Customer ct[], int size);

int main(){
    Customer ct[11];
    ct[0] ={"Zafeer", 25554112, 50000};
    ct[1] ={"Mahad", 25554116, 65000};
    ct[2] ={"Armghan", 25554142, 50000};
    ct[3] ={"Zoya", 25554522, 1000};
    ct[4] ={"Nayab", 25554422, 300};
    ct[5] ={"Norain", 25557112, 50000};
    ct[6] ={"Bunti", 25552112, 600};
    ct[7] ={"Rohit", 25559112, 7};
    ct[8] ={"Sher Khan", 25554212, 5};
    ct[9] ={"Langor Khan", 25554712, 144500};
    ct[10] ={"Zaheer ul Deen", 25554112, 9000};
    customer_balance(ct,11);
    add_cash(ct,11);
    return 0;
}
```

```
void customer_balance(Customer ct[], int size){
    cout << "Name of Customers having balance < 20,000" << endl;
    for(int i=0; i<size; i++){
        if(ct[i].balance<20000)
            cout << ct[i].name << endl;
    }
}
void add_cash(Customer ct[], int size){
    cout << "Add 1000k" << endl;
    for(int i=0; i<size; i++){
        if(ct[i].balance<30000){
            cout << ct[i].balance << endl;
            ct[i].balance+=1000000;
            cout << ct[i].balance << endl;
        }
    }
}
```

Home Activity

Q3. Let us work on the menu of a library. Create a structure containing book information like accession number, name of author, book title and flag to know whether book is issued or not.

Create a menu in which the following can

be done. 1 - Display book information
2 - Add a new book
3 - Display all the books in the library of a particular
author 4 - Display the number of books of a particular
title
5 - Display the total number of books in
the library 6 - Issue a book
(If we issue a book, then its number gets decreased by 1 and if we add a book, its number gets increased by 1)

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

// For using it in a function's argument as flag.
#define AUTHOR 1
#define TITLE 2

// Book Structure
struct Book{
    string author_name;
    string book_name;
    bool is_issued;
    // Default value is 0 to avoid unnecessory iteration
    // while working with its array.
    int accession_number=0;
};
// Function Prototype
void display_menu();
void display_info(Book books[]);
int find_book(Book books[], int filter);
void add_new(Book books[]);
void issue_book(Book books[]);
int count_by_title(Book books[]);
void add_demo_books(Book books[]);
int book_counter(Book books[]);

int main(){
    // Our Library can store 1000 books
    Book books[1000];
    int user_choice;
    // This will add some demo books(Hard Coded)
    add_demo_books(books);

    do{
        display_menu();
        cout << "Choice: ";
        cin >> user_choice;

        switch(user_choice){
            case 1:
                display_info(books);
                break;
            case 2:
                add_new(books);
                break;
            case 3:
                find_book(books, AUTHOR);
                break;
            case 4:
                cout << "Count: " << count_by_title(books);
                break;
            case 5:
                book_counter(books);
                break;
            case 6:
                issue_book(books);
                break;
            default:
                cout << "Enter a valid choice!" << endl;
        }
        }while(user_choice!=0);
        return 0;
    }
    void display_menu(){
        cout << "1 - Display book information" << endl;
        cout << "2 - Add a new book" << endl;
        cout << "3 - Display all the books in the library of a particular author" <
< endl;
        cout << "4 - Display the number of books of a particular title" << endl;
        cout << "5 - Display the total number of books in the library" << endl;
        cout << "6 - Issue a book" << endl;
        cout << "0 - Exit" << endl;
    }
    void display_info(Book books[]){
        for(int i=0; i<book_counter(books); i++){
            cout << "------------------------------------------" << endl;
            cout << "Author: " << books[i].author_name << endl;
            cout << "Book Name: " << books[i].book_name << endl;
            cout << "Issued: " << ((books[i].is_issued)?"Yes":"No") << endl;
            cout << "Accession Number: " << books[i].accession_number << endl;
        }
        cout << "------------------------------------------" << endl;

    }
    // By using filters
    int find_book(Book books[], int filter){
        string search_inp;
        int book_index;
        cout << "Enter " << ((filter==AUTHOR)?"Auther":"Title") << " Name: ";
        getline(cin,search_inp);

        for(int i=0; i<book_counter(books); i++){
            if(((filter==AUTHOR)?books[i].author_name:books[i].book_name)==search_i
np){
                cout << "Author: " << books[i].author_name << endl;
                cout << "Book Name: " << books[i].book_name << endl;
                cout << "Issued: " << ((books[i].is_issued)?"Yes":"No") << endl;
                cout << "Accession Number: " << books[i].accession_number << endl;
                book_index = i;
            }
        }
        return book_index;
    }
    void add_new(Book books[]){
        int index=book_counter(books);
        cout << "Book Name: ";
        cin >> books[index].book_name;
        cout << "Author Name: ";
        cin >> books[index].author_name;
        cout << "Issued (1/0): ";
        cin >> books[index].is_issued;
        books[index].accession_number=index+1;
    }
    int count_by_title(Book books[]){
        string user_inp;
        cout << "Enter book title: ";
        cin >> user_inp;
        int counter=0;
        for(int i=0; i<book_counter(books); i++) if(user_inp==books[i].book_name) c
ounter++;
        return counter;
    }
    void issue_book(Book books[]){
        int index = find_book(books,TITLE);
        if(books[index].accession_number!=0){
            cout << "Issue (1/0): ";
            cin >> books[index].is_issued;
            cout << "Book issued!" << endl;
        }
    }
    int book_counter(Book books[]){
        int counter=0;
        while(books[counter].accession_number!=0) counter++;

        return counter;
    }
    void add_demo_books(Book books[]){
        books[0] = {"Jon Erickson", "Art of Exploitation", 1, 1};
        books[1] = {"Justin Seitz", "Black Hat Python", 1, 2};
        books[2] = {"Ben Clark", "Rtfm: Red Team Field Manual", 1, 3};
        books[3] = {"Don Murdoch", "Blue Team Handbook", 1, 4};
    }
```

Q4. Write a structure to store the names, salary and hours of work per day of 10 employees in a company. Write a program to increase the salary depending on the number of hours of work per day as follows and then print the name of all the employees along with their final salaries.

Hours of work per day	**8**	**10**	**>=12**
Increase in salary	**1k**	**2k**	**3k**

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

struct Employee{
  string name;
  // Base salary is 90k
  float salary=90000;
```

```
    int work_hours;
};
int main(){
    Employee emp[10];
    for(int i=0;i<10; i++){
        cout << "Name: ";
        cin >> emp[i].name;
        cout << "Working hours: ";
        cin >> emp[i].work_hours;
        if(emp[i].work_hours>=12){
            emp[i].salary += 3000;
        }
        else if(emp[i].work_hours>=10){
            emp[i].salary += 2000;
        }
        else if(emp[i].work_hours>=8){
            emp[i].salary += 1000;
        }
        cout << "Salary: " << emp[i].salary << endl;
    }

    return 0;
}
```

LAB NO. 05

CLASSES IN C++

Lab Objectives

Following are the lab objectives:

1. C++ Class
2. C++ Objects
3. Class Member Functions
   - Inside the class definition
   - Outside the class definition

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

Object Oriented Programming

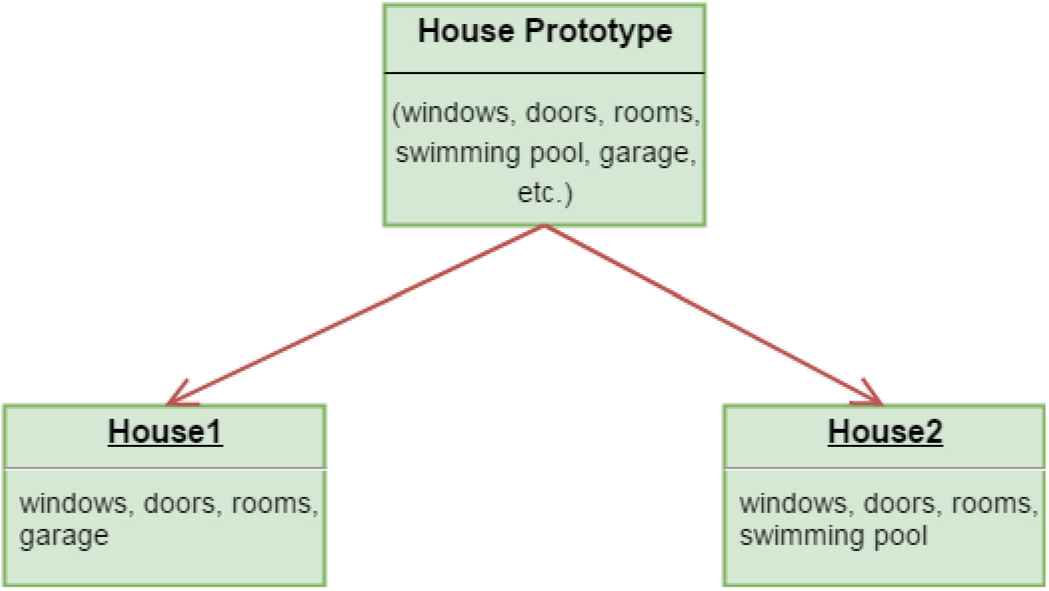
Suppose, we need to store the length, breadth, and height of a rectangular room and calculate its area and volume. To handle this task, we can create three variables, say, length, breadth, and height along with the functions calculateArea() and calculateVolume().

However, in C++, rather than creating separate variables and functions, we can also wrap these related data and functions in a single place (by creating objects). This programming paradigm is known as object-oriented programming.

C++ Class

A C++ class combines data and methods for manipulating the data into one. Classes also determine the forms of objects. The data and methods contained in a class are known as class members. A class is a user-defined data type. To access the class members, we use an instance of the class. You can see a class as a blueprint for an object.

A class be a prototype for a house. It shows the location and sizes of doors, windows, floors, etc. From these descriptions, we can construct a house. The house becomes the object. It's possible to create many houses from the prototype. Also, it's possible to create many objects from a class.


House Prototype
(windows, doors, rooms, swimming pool, garage, etc.)
House1
windows, doors, rooms, garage
House2
windows, doors, rooms, swimming pool

In the above figure, we have a single house prototype. From this prototype, we have created two houses with different features.

Class Declaration

In C+, a class is defined using the class keyword. This should be followed by the class name. The class body is then added between curly braces { }.

Syntax:

Class class-name

```
{
// data
// functions
};
```

- The class-name is the name to assign to the class.
- The data is the data for the class, normally declared as variables.
- The functions are the class functions.

Private and Public Keywords

You must have come across these two keywords. They are access modifiers.

Private:

When the private keyword is used to define a function or class, it becomes private. Such are only accessible from within the class.

Public:

The public keyword, on the other hand, makes data/functions public. These are accessible from outside the class.

Object Definition

Objects are created from classes. Class objects are declared in a similar way as variables are declared. The class name must start, followed by the object name. The object of the class type.

Syntax:

```
class-name object-name;
```

- The class-name is the name of the class from which an object is to be created.
- The object-name is the name to be assigned to the new object.

This process of creating an object from a class is known as instantiation.

Accessing Data Members

To access public members of a class, we use the (.)dot operator. These are members marked with public access modifier.

Example 1: #include

```
<iostream> using
namespace std; class
Phone {
```

```
public:
        double cost;
        int slots;
};

int main() {
        Phone Y6;
        Phone Y7;
        Y6.cost = 100.0;
        Y6.slots = 2;
        Y7.cost = 200.0;
        Y7.slots = 2;
        cout << "Cost of Huawei Y6 : " << Y6.cost << endl;
        cout << "Cost of Huawei Y7 : " << Y7.cost << endl;
        cout << "Number of card slots for Huawei Y6 : " << Y6.slots << endl;
        cout << "Number of card slots for Huawei Y7 : " << Y7.slots << endl;
        return 0;
}
```

Class Member Functions

Functions help us manipulate data. Class member functions can be defined in two ways:

- Inside the class definition
- Outside the class definition

If a function is to be defined outside a class definition, we must use the scope resolution operator (::). This should be accompanied by the class and function names.

Example 2: #include

```
<iostream> #include
<string> using
namespace std; class
Name
{
public:
    string tutorial_name;
    int id;
    void printname();
    void printid()
    {
        cout << "class id is: "<< id;
    }
};
void Name::printname()
{
    cout << "class name is: " << class_name;
}
int main() {
    Name n1;
    n1.class_name = "C++";
    n1.id = 1001;
    n1.printname();
    cout << endl;
    n1.printid();
    return 0;
}
```

Example 3: Using public and private in C++ Class

```
#include   <iostream>
using  namespace  std;
class Room {
  private: double
   length;
   double  breadth;
   double height;
  public:
   // function to initialize private variables
   void getData(double len, double brth, double hgt) {
       length = len;
       breadth = brth;
       height = hgt;
   }
   double calculateArea() {
       return length * breadth;
   }
   double calculateVolume() {
       return length * breadth * height;
   }
};
int main() {
   // create object of Room class
   Room room1;
   // pass the values of private variables as arguments
   room1.getData(42.5, 30.8, 19.2);
   cout << "Area of Room = " << room1.calculateArea() << endl;
   cout << "Volume of Room = " << room1.calculateVolume() << endl;
   return 0;
}
```

Lab Tasks

Q1). Write a C++ program to create student class that contains attributes of the student name, roll no, and total-marks. Write two functions to get and display these attributes.

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

class Student{
    private:
        string name;
        int rollNum;
        float totalMarks;
    void getData(){
        cout << "Enter Name: ";
        cin >> name;
        cout << "Enter Roll Number: ";
        cin >> rollNum;
        cout << "Enter Total Marks: ";
        cin >> totalMarks;
    }
    void printInfo(){
        cout << "Name: " << name << endl;
        cout << "Roll Number: " << rollNum << endl;
        cout << "Total Marks: " << totalMarks << endl;
    }
};
int main(){
    Student s1;
    s1.getData();
    s1.printInfo();
}
```

Write a C++ program to create a class name as arithmetic_operations with two attributes (number 1 and number 2). Write four functions addition, subtraction, multiplication and division. Program menu should be user friendly.

```
#include <iostream>
#include <float.h>
using namespace std;

class ArithmeticOperations{

    private:
        double num1, num2;
    public:
        char choice;
        void menu(){
            cout << "+ for addition." << endl;
            cout << "- for Subtraction." << endl;
            cout << "* for Multiplication." << endl;
            cout << "+ for addition." << endl;
            cout << "q for quite." << endl;
        }
        void getData(){
            cout << "Enter Number 1: ";
            cin >> num1;
            cout << "Enter Number 2: ";
            cin >> num2;
        }
        double addition(){
            return num1+num2;
        }
        double subtraction(){
            return num1-num2;
        }
        double multiplication(){
            return num1*num2;
        }
        double division(){
            return (num2!=0)?num1/num2:DBL_MAX;
        }
};

int main(){
    ArithmeticOperations a1;
    char choice;
    do{
        a1.menu();
        cout << "Choose: ";
        cin >> a1.choice;
        a1.getData();

        switch(a1.choice){
            case '+':
                cout << "Result: " << a1.addition() << endl;
                break;
            case '-':
                cout << "Result: " << a1.subtraction() << endl;
                break;
            case '*':
                cout << "Result: " << a1.multiplication() << endl;
                break;
            case '/':
                cout << "Result: " << a1.division() << endl;
                break;
            default:
                cout << "Invalid option." << endl;
        }
    }while(a1.choice!='q');

    return 0;
}
```

Home Activity

Q.3. Create a class named “Student” with a string variable “name” and an integer variable “roll_no”. Assign the value of roll_no as “101” and that of name as "Ali" by creating an object of the class Student.

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

class Student{
    public:
        string name;
        int rollNo;
};
int main(){
    Student s1 = {"Ali", 101};
    cout << s1.name << endl;
    cout << s1.rollNo << endl;
    return 0;
}
```

Q4. Write a C++ program to print the area of a rectangle by creating a class named 'Area' having two functions. First function named as “Set_Dim” takes the length and breadth of the rectangle as parameters and the second function named as 'Get_Area' returns the area of the rectangle. Length and breadth of the rectangle are entered through user.

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

class Area{
    public:
        double length,breadth;
    void setDim(double length,double breadth){
        Area::length =  length;
        Area::breadth = breadth;
    }
    double getArea(){
        return length*breadth;
    }
};
int main(){
    Area a1;
    a1.setDim(20.25,65.214);
    cout << "Area: " << a1.getArea() << endl;
    return 0;
}
```

LAB NO. 06

CONSTRUCTORS AND DESTRUCTORS IN C++

Lab Objectives

Following are the lab objectives:

1. Constructors
   - Default constructor
   - Parameterize constructor
   - Copy constructor
2. Destructors

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

1. Constructors and Destructors In C++

1.1 Constructor

A Constructor is a member function of a class. It is mainly used to initialize the objects of the class. It has the same name as the class. When an object is created, the constructor is automatically called. It is a special kind of member function of a class.

Difference between Constructor and Other Member Functions:

i. The Constructor has the same name as the class name.
ii. The Constructor is called when an object of the class is created.
iii. A Constructor does not have a return type.
iv. When a constructor is not specified, the compiler generates a default constructor which does nothing.

There are 3 types of constructors:

1. Default Constructor
2. Parameterized Constructor
3. Copy constructor

A constructor can also be defined in the private section of a class.

Default Constructor

A Default constructor is a type of constructor which doesn't take any argument and has no parameters.

Example 1

```
#include   <iostream>
using  namespace  std;
class sum {
public:
int  y,  z;
sum()
{
y = 7;
z = 13;
}
};
int main()
{
```

```
sum a;
cout <<"the sum is: "<< a.y+a.z;
return 0;
}
```

Example 2

```
#include   <iostream>
using  namespace  std;
class Line {
   public:
     void setLength( double len );
     double getLength( void );
     Line(); // This is the constructor
   private:
     double length;
};
// Member functions definitions including constructor
Line::Line(void) {
   cout << "Object is being created" << endl;
}
  void Line::setLength( double len ) {
   length = len;
}
  double Line::getLength( void ) {
   return length;
}
// Main function for the program
int main() {
   Line line;
   // set line length
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

Parameterized Constructor

Passing of parameters to the constructor is possible. This is done to initialize the value using these passed parameters. This type of constructor is called a parameterized constructor.

The constructor is defined as follows:

```
test(int x1)
{
x = x1;
}
```

There is a parameter that is passed to the constructor. The value is passed when the object is created in the main function as shown below.

```
test t(10);
```

Inside the main function, we create an object of class test and pass the value of the variable.

Example 3 #include

```
<iostream> using
namespace std; class
test {
public:
int x;
test(int x1)
{
x = x1;
}
int getX()
{
return x;
}
};
int main()
{
test a(10);
cout << "a.x = " << a.getX() ;
return 0;
}
```

Example 4

```
#include <iostream>
```

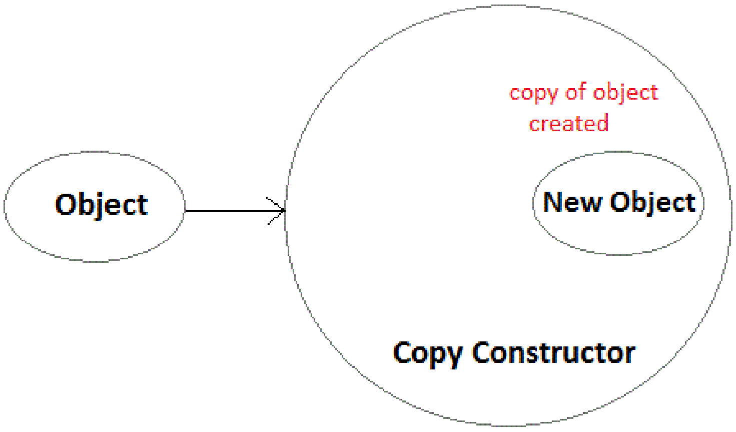
```
using namespace std;
class Line {
    public:
       void setLength( double len );
       double getLength( void );
       Line(double len); // This is the constructor
    private:
       double length;
};
// Member functions definitions including constructor
Line::Line( double len) {
    cout << "Object is being created, length = " << len << endl;
    length = len;
}
  void Line::setLength( double len ) {
    length = len;
}
  double Line::getLength( void ) {
    return length;
}
// Main function for the program
int main() {
    Line line(10.0);
    // get initially set length.
    cout << "Length of line : " << line.getLength() <<endl;
    // set line length again
    line.setLength(6.0);
    cout << "Length of line : " << line.getLength() <<endl;
    return 0;
}
```

Copy Constructor

A Copy Constructor is a Constructor which initializes an object of a class using another object of the same class.

Syntax of Copy Constructor

```
Classname(const classname & objectname)
{
    . . . .
}
```



Example 5

```
#include<iostream>
using namespace std;
class test
{
private:
int x;
public:
test(int x1)
{
x = x1;
}
test(test &t2)
{
x = t2.x;
}
int getX()
{
return x;
}
};
int main()
{
test t1(7); // Normal constructor is called here
test t2 = t1; // Copy constructor is called here
cout << "t1.x = " << t1.getX();
cout << "nt2.x = " << t2.getX();
return 0;
}
```

2. Constructor Overloading in C++

In C++, We can have more than one constructor in a class with same name, as long as each has a different list of arguments. This concept is known as Constructor Overloading and is quite similar to function overloading.

- Overloaded constructors essentially have the same name (name of the class) and different number of arguments.
- A constructor is called depending upon the number and type of arguments passed.
- While creating the object, arguments must be passed to let compiler know, which constructor needs to be called.

Example 6

```
#include   <iostream>
using  namespace  std;
class construct
{
public:
        float area;
        // Constructor with no parameters
        construct()
        {
                area = 0;
        }
        // Constructor with two parameters
        construct(int a, int b)
        {
                area = a * b;
        }
        void disp()
        {
                cout<< area<< endl;
        }
};
int main()
{
        construct o;
        construct o2( 10, 20);
        o.disp();
        o2.disp();
        return 0;
}
```

Destructors in C++

Destructor is a special class function which destroys the object as soon as the scope of object ends. The destructor is called automatically by the compiler when the object goes out of scope.

The syntax for destructor is same as that for the constructor, the class name is used for the name of destructor, with a tilde ~ sign as prefix to it.

```
class A
{
    public:
    // defining destructor for class
    ~A()
    {
      // statement
    }
};
```

Example to see how Constructor and Destructor are called

Below we have a simple class A with a constructor and destructor. We will create object of the class and see when a constructor is called and when a destructor gets called.

```
class A
{
      // constructor
      A()
      {
          cout << "Constructor called";
      }

      // destructor
      ~A()
      {
          cout << "Destructor called";
      }
};
int main()
{
      A obj1; // Constructor Called
      int x = 1
      if(x)
      {
           A obj2; // Constructor Called
        } // Destructor Called for obj2
} // Destructor called for obj1
```

```
Constructor called Constructor
called
Destructor called
Destructor called
```

Lab Tasks

Q1). Write a program to print the names of employee by creating an Employee class. If no name is passed while creating an object of the Employee class, then the name should be "Unknown", otherwise the name should be equal to the String value passed while creating the object of the Employee class.

```
#include <iostream>
using namespace std;
class Employee{
    private:
        string name;
    public:
        Employee(){
            name="Unknown";
        }
        Employee(string name){
            Employee::name=name;
        }
        void printName(){
            cout << "Employee Name: " << name << endl;
        }
};
int main(){
    Employee em1, em2("Bunti");
    em1.printName();
    em2.printName();
}
```

Q2). Suppose you have an account in HBL Bank with an initial amount of 10k and you have to add some more amount to it. Create a class 'AddAmount' with a data member named 'amount' with an initial value of 10k. Now make two constructors of this class as follows:

```
#include <iostream>
using namespace std;
class AddAmount{
    private:
        double amount=10000;
    public:
        AddAmount(){
            cout << "Ammount: " << amount << endl;

        }
        AddAmount(double amount){
            AddAmount::amount += amount;
            cout << "Ammount: " << AddAmount::amount << endl;
        }
};
```

```
int main(){
    AddAmount a1,a2(500);
}
```

LAB NO. 07

CONSTRUCTORS MEMBER INITIALIZATION LIST IN C++

Lab Objectives

Following are the lab objectives:

1. Initializer list in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

Initializer List in C++

In the previous lab, we learned about how classes and their objects can be created and the different ways their members can be accessed. We also saw how data members are initialized in the constructor of a class as shown below.

```
class Rectangle
{
    int length;
    int breadth;
    public:
      Rectangle()
      {
        length = 7;
        breadth = 4;
      }
};
```

Syntax:

```
Constructorname(datatype value1, datatype value2):datamember(value1),datamember(value2)
{
    ...
}
```

In the above constructor, we assigned the values 7 and 4 to the data members length and breadth respectively. Note that, here we assigned the values to these variables, not initialized.

In the case of constructors having parameters, we can directly initialize our data members using initialization lists.

Using initialization list, we can write the above code as follows.

```
class Rectangle
{
    int length;
    int breadth;
    public:
      Rectangle() : length(7), breadth(4) // initializing data members
      {
        // no need to assign anything here
      }
};
```

An initializer list starts after the constructor name and its parameters and begins with a colon (:) followed by the list of variables which are to be initialized separated by a comma with their values in curly brackets.

Let's see an example for the above code.

```
#include   <iostream>
using  namespace  std;
class Rectangle
{
    int length;
    int breadth;
    public:
        Rectangle() : length(7), breadth(4)
        {

        }
        int printArea()
        {
            return length * breadth;
        }
};
int main()
{
    Rectangle rt;
    cout << rt.printArea() << endl;
    return 0;
}
```

Let's see another example of initialization list in case of parameterized constructor.

```
#include   <iostream>
using  namespace  std;
class Rectangle
{
    int length;
    int breadth;
    public:
        Rectangle( int l, int b ) : length(l), breadth(b)
```

```
                {

                }
                int printArea()
                {
                        return length * breadth;
                }
};
int main()
{
         Rectangle rt( 7, 4 );
         cout << rt.printArea() << endl;
         return 0;
}
```

In this example, the initializer list is directly initializing the variables length and breadth with the values of l and b which are 7 and 4 respectively.

This is the same as the following code with the only difference that in the above example, we are directly initializing the variables while in the following code, the variables are being assigned the parameterized values.

```
class Rectangle
{
    int length;
    int breadth;
    public:
       Rectangle( int l, int b )
       {
          length = l;
          breadth = b;
       }
};
```

Need for Initialization List

1) For initializing const data member

Though we can use initialization list anytime as we did in the above examples, but there are certain cases where we have to use initialization list otherwise the code won't work.

If we declare any variable as const, then that variable can be initialized, but not assigned.

Any variable declared with const keyword before its data type is a const variable.

To initialize a const variable, we use initialization list. Since we are not allowed to assign any value to a const variable, so we cannot assign any value in the body of the constructor. So, this is the case where we need initialization list.

Following is an example initializing a const data member with initialization list.

```
#include <iostream>
using namespace std;
class Rectangle
{
        const int length;
        const int breadth;
        public:
              Rectangle( int l, int b ) : length(l), breadth(b)
              {

              }
              int printArea()
              {
                    return length * breadth;
              }
};
int main()
{
        Rectangle rt( 7, 4 );
        cout << rt.printArea() << endl;
        return 0;
}
```

2) When data member and parameter have same name

```
#include<iostream>
using namespace std;
class Base
{
    private:
    int value;
    public:
    Base(int value):value(value)
    {
      cout << "Value is " << value;
    }
};
int main()
{
    Base il(10); return 0;
}
```

Lab Tasks

Q1). Suppose you have an account in HBL Bank with an initial amount of 10k and you have to add some more amount to it. Create a class 'AddAmount' with a data member named 'amount' with an initial value of 10k. Now make two constructors of this class as follows:

1. Create a default constructor that uses a member initializer list that allows the user to initialize initial value in account.
2. having a parameter which is the amount that will be added to the account.

3) without any parameter - no amount will be added to the account.
4) having a parameter which is the amount that will be added to the

Create an object of the 'AddAmount' class and display the final amount in the account.

```
#include <iostream>
using namespace std;
class AddAmount{
    private:
        int initAmount=10000;
        double amount;
    public:
        AddAmount():amount(initAmount) {}
        AddAmount(double addAmount):amount(initAmount+addAmount) {

        }
        void printAmount(){
            cout << "Amount: " << amount << endl;
        }
};
int main(){
    AddAmount a1(500), a2;
    a1.printAmount();
    a2.printAmount();
}
```

Q2). Declare a class **Area** which perform the following tasks using constructor chaining:

1. Create a default constructor that uses a member initializer list that initialize the data members (length, width and breadth)
2. Calculate area of rectangle when length and width are passed.
3. Calculate area of circle when radius is passed.
4. Calculate area of cube when length, width and breadth is passed.
5. Define destructor at the end of Area class

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

#define PI 3.14

class Area{
    public:
        int radius, length, width, breadth;

        Area(int r):radius(r){
            cout << "Area of Circle: " << PI*radius*radius << endl;
        }
        Area(int l, int w):length(l), width(w){
            cout << "Area of rectangle: " << length*width << endl;
        }
        Area(int l, int w, int b):length(l),width(w), breadth(b){
            cout << "Area of cube: " << length*width*breadth << endl;
        }
};

int main(){
    Area a1(5);
    Area a2(6,5);
    Area a3(2,8,9);
    return 0;
}
```

LAB NO. 09

INHERITANCE IN C++

Lab Objectives

Following are the lab objectives:

1. Inheritance in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

C++ Inheritance

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

In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

Access Control and Inheritance

A derived class can access all the non-private members of its base class. Thus base-class members that should not be accessible to the member functions of derived classes should be declared private in the base class. We can summarize the different access types according to - who can access them in the following way –

Access	**public**	**protected**	**privat e**
Same class	yes	yes	yes
Derived classes	yes	yes	no
Outside classes	yes	no	no

A derived class inherits all base class methods with the following exceptions −

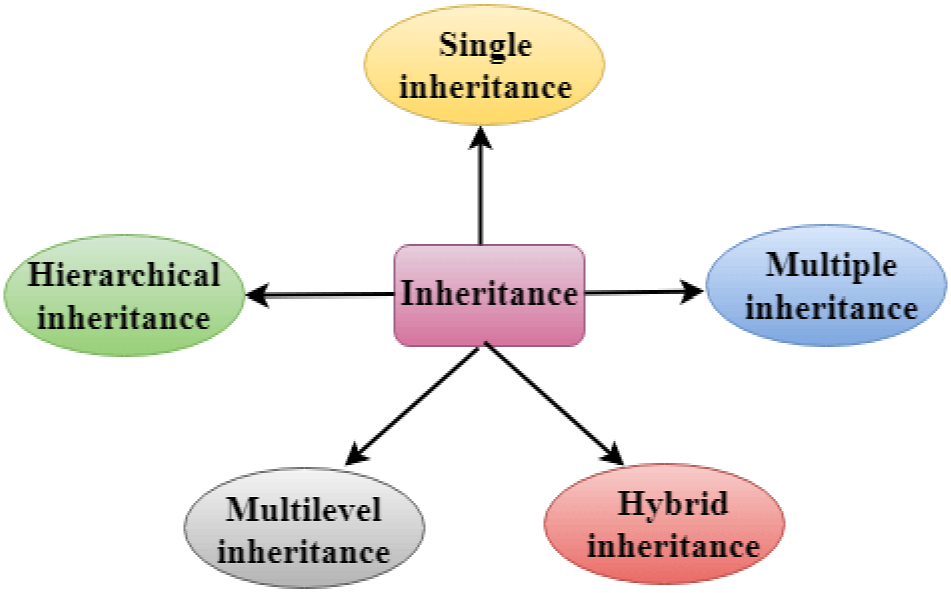
- Constructors, destructors and copy constructors of the base class.
- Overloaded operators of the base class.
- The friend functions of the base class.

Types of Inheritance

C++ supports five types of inheritance:

- Single inheritance
- Multiple inheritance
- Hierarchical inheritance
- Multilevel inheritance

- Hybrid inheritance



Derived Classes

A Derived class is defined as the class derived from the base class.

The Syntax of Derived class:

```
class derived_class_name:visibility-mode base_class_name
{
   // body of the derived class.
}
```

Where,

derived_class_name: It is the name of the derived class.

visibility mode: The visibility mode specifies whether the features of the base class are publicly inherited or privately inherited. It can be public or private.

Type of Inheritance

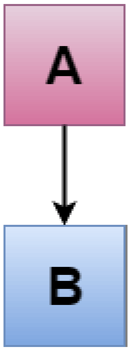
When deriving a class from a base class, the base class may be inherited through **public, protected** or **private** inheritance. The type of inheritance is specified by the access-specifier as explained above.

We hardly use **protected** or **private** inheritance, but **public** inheritance is commonly used. While using different type of inheritance, following rules are applied −

- **Public Inheritance** − When deriving a class from a **public** base class, **public** members of the base class become **public** members of the derived class and **protected** members of the base class become **protected** members of the derived class. A base class's **private** members are never accessible directly from a derived class, but can be accessed through calls to the **public** and **protected** members of the base class.
- **Protected Inheritance** − When deriving from a **protected** base class, **public** and **protected** members of the base class become **protected** members of the derived class.
- **Private Inheritance** − When deriving from a **private** base class, **public** and **protected** members of the base class become **private** members of the derived class.

C++ Single Inheritance

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



Where 'A' is the base class, and 'B' is the derived class.

C++ Single Level Inheritance Example: Inheriting Fields

When one class inherits another class, it is known as single level inheritance. Let's see the example of single level inheritance which inherits the fields only.

```
#include
<iostream> using
namespace std;
class Account {
public:
   float salary = 60000;
 };
   class Programmer: public Account {
public:
  float bonus = 5000;
  };
int main(void) {
    Programmer
    p1;
    cout<<"Salary: "<<p1.salary<<endl;
   cout<<"Bonus:  "<<p1.bonus<<endl;
   return 0;
}
```

Output:

```
Salary: 60000
Bonus: 5000
```

In the above example, Employee is the **base** class and Programmer is the **derived** class.

C++ Single Level Inheritance Example: Inheriting Methods

```
#include
<iostream> using
namespace std;
class A
{
   int a =
   4; int b
   = 5;
   public:
   int mul()
   {
      int c = a*b;
      return c;
   }
};

class B : private A
{
   public:
   void display()
   {
        int result = mul();
        std::cout <<"Multiplication of a and b is : "<<result<< std::endl;
    }
};
int main()
{
   B b;
   b.display();

    return 0;
}
```

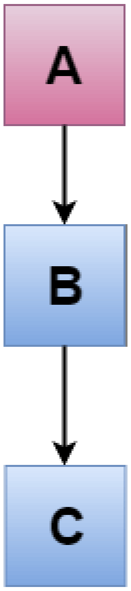
Output:

```
Multiplication of a and b is : 20
```

In the above example, class A is privately inherited. Therefore, the mul() function of class 'A' cannot be accessed by the object of class B. It can only be accessed by the member function of class B.

C++ Multilevel Inheritance

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



C++ Multi Level Inheritance Example

When one class inherits another class which is further inherited by another class, it is known as multi level inheritance in C++. Inheritance is transitive so the last derived class acquires all the members of all its base classes.

Let's see the example of multi level inheritance in C++.

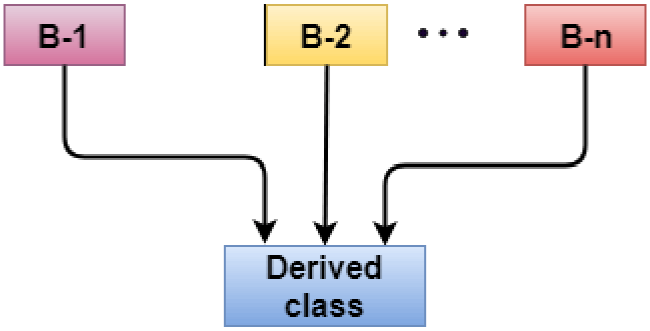
```
#include <iostream>
using namespace std;
class Animal {
   public:
 void eat() {
    cout<<"Eating..."<<endl;
 }
   };
   class Dog: public Animal
   {
       public:
     void bark(){
    cout<<"Barking..."<<endl;
     }
   };
   class BabyDog: public Dog
   {
       public:
     void weep() {
    cout<<"Weeping...";
     }
   };
int main(void) {
    BabyDog d1;
    d1.eat();
    d1.bark();
     d1.weep();
     return 0;
}
```

Output:

```
Eating...
Barking...
Weeping...
```

C++ Multiple Inheritance

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



Syntax of the Derived class:

```
class D : visibility B-1, visibility B-2, ?
{
    // Body of the class;
}
```

Let's see a simple example of multiple inheritance.

```
#include <iostream>
using namespace std;
class A
{
    protected:
     int a;
    public:
    void get_a(int n)
    {
        a = n;
    }
};
class B
{
    protected:
    int b;
    public:
    void get_b(int n)
    {
        b = n;
    }
};
class C : public A,public B
{
   public:
    void display()
    {
        std::cout << "The value of a is : " <<a<< std::endl;
        std::cout << "The value of b is : " <<b<< std::endl;
        cout<<"Addition of a and b is : "<<a+b;
    }
};
int main()
{
   C c;
   c.get_a(10);
   c.get_b(20);
   c.display();

    return 0;
}
```

Output:

```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

In the above example, class 'C' inherits two base classes 'A' and 'B' in a public mode.

Ambiquity Resolution in Inheritance

Ambiguity can be occurred in using the multiple inheritance when a function with the same name occurs in more than one base class.

Let's understand this through an example:

```
#include <iostream>
using namespace std;
class A
{
    public:
    void display()
    {
        cout << "Class A" <<endl;
    }
};
class B
{
    public:
    void display()
    {
        cout << "Class B" <<endl;
    }
};
class C : public A, public B
{
    void view()
    {
        display();
    }
};
int main()
{
    C c;
    c.display();
    return 0;
}
```

Output:

```
error: reference to 'display' is ambiguous
        display();
```

- The above issue can be resolved by using the class resolution operator with the function. In the above example, the derived class code can be rewritten as:

```
class C : public A, public B
{
    void view()
    {
        A :: display();         // Calling the display() function of class A.
        B :: display();         // Calling the display() function of class B.

    }
};
```

An ambiguity can also occur in single inheritance.

Consider the following situation:

```
class A
{
   public:
void display()
{
   cout<<?Class A?;
}
} ;
class B
{
  public:
```

```
 void display()
{
 cout<<?Class B?;
}
} ;
```

In the above case, the function of the derived class overrides the method of the base class. Therefore, call to the display() function will simply call the function defined in the derived class. If we want to invoke the base class function, we can use the class resolution operator.

```
int main()
{
    B b;
   b.display();              // Calling the display() function of B class.
   b.B :: display();       // Calling the display() function defined in B class.
}
```

Lab Tasks

Q1). Write a class Employee that contains attributes of employee id and his salary. The class contains member functions to input and show the attribute. Write a child class Manager that inherits from Employee class. The child class has attributes of manager id and his department. It also contains the member functions to input and show its attribute.

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

class Employee{
    public:
        int empId, empSalary;

        void input(){
            cout << "Enter Employee ID: "; cin >> empId;
            cout << "Enter Employee Salary: "; cin >> empSalary;
        }
        void show(){
            cout << "Employee ID: " << empId << endl;
            cout << "Employee Salary: " << empSalary << endl;
        }
};
class Manager: public Employee{
    public:
        int managerId;
        void input(){
            cout << "Enter Manager ID: "; cin >> managerId;
            cout << "Enter Manager Salary: "; cin >> empSalary;
        }
        void show(){
            cout << "Manager ID: " << managerId << endl;
            cout << "Manager Salary: " << empSalary << endl;
        }
};

int main(){
    Manager m1;
    Employee::m1.input();
    Employee::m1.show();

    Manager::m1.input();
    Manager::m1.show();
    return 0;
}
```

Q2). Create two classes named Mammals and MarineAnimals. Create another class named BlueWhale which inherits both the above classes. Now, create a function in each of these classes which prints "I am mammal", "I am a marine animal" and "I belong to both the categories: Mammals as well as Marine Animals" respectively. Now, create an object for each of the above class and try calling

1 - function of Mammals by the object of Mammal

2 - function of MarineAnimal by the object of

MarineAnimal 3 - function of BlueWhale by the object of

BlueWhale

3 - function of each of its parent by the object of BlueWhale

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

class Mammals{
    public:
        void mammalShow(){
            cout << "I am mammal." << endl;
        }
};
class MarineAnimals{
    public:
        void marnineShow(){
            cout << "I am a marine animal" << endl;
        }
};
class BlueWhales: public Mammals, public MarineAnimals{
    public:
        void blueWhaleShow(){
            cout << "I belong to both the categories: \nMammals as well as Marine Anim
als" << endl;
        }
};

int main(){
    Mammals Mammal;
    Mammal.mammalShow();

    MarineAnimals MarineAnimal;
    MarineAnimal.marnineShow();

    BlueWhales BlueWhale;
    BlueWhale.blueWhaleShow();

    return 0;
}
```

Q3). Create a C++ program in which we can calculate the total marks of each student of a class in OOP, data structure and database and the average marks of the class. The number of students in the class are entered by the user. Create a class named Marks with data members for roll number, name and marks. Create three other classes inheriting the Marks class, OOP, data structure and database, which are used to define marks in individual subject of each student.

```
#include <iostream>
using namespace std;
class Marks{
    public:
        int rollNo,marks=-1;
        string name;
};
class OOP: public Marks{
    public:
        OOP(string stName,int stRoll, int stMarks):
            name(stName), rollNo(stRoll), marks(stMarks){}
};
class DataStructure: public Marks{
    public:
        DataStructure(string stName,int stRoll, int stMarks):
            name(stName), rollNo(stRoll), marks(stMarks){}
};
class Database: public Marks{
    public:
        Database(string stName,int stRoll, int stMarks):
            name(stName), rollNo(stRoll), marks(stMarks){}
};
// Not Completed
int main(){
    OOP oop;
    DataStructure ds;
    Database db;
    int numStudents;

    cout  << "Enter tudents: " << endl; cin >> numStudents;

    for(int i=0; i<numStudents; i++){
        cout << "Enter Name: " <<
    }
    return 0;
}
```

LAB NO. 10

INHERITANCE IN C++

Lab Objectives

Following are the lab objectives:

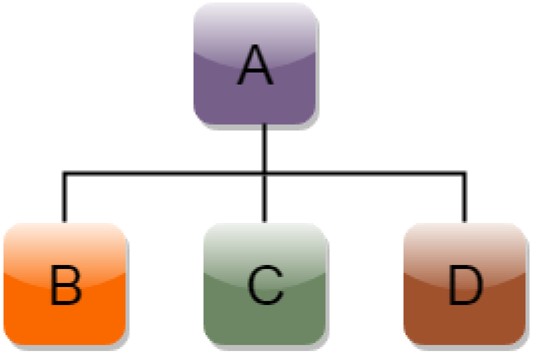
1. Inheritance in C++
2. Constructors and destructors calling order

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

C++ Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class. Example: Computer, Civil, Mechanical, Electrical are derived from Engineer. Natural Language and Programming Language derived from Language.



Syntax of Hierarchical inheritance:

```
    class A
{
      // body of the class A.
}
    class B : public A
{
      // body of class B.
}
    class C : public A
{
      // body of class C.
}
    class D : public A
{
      // body of class D.
}
```

Let's see a simple example:

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

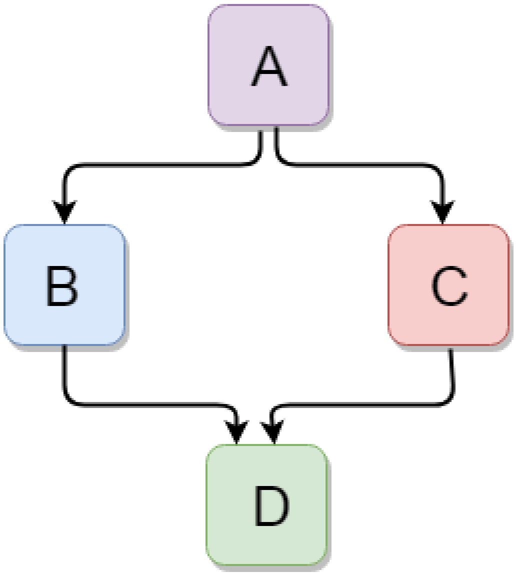
```
class Shape                      // Declaration of base class.
{
      public:
      int a;
      int b;
      void get_data(int n,int m)
      {
        a= n;
        b = m;
      }
};
    class Rectangle : public Shape // inheriting Shape class
{
      public:
      int rect_area()
      {
        int result = a*b;
        return result;
      }
};
    class Triangle : public Shape        // inheriting Shape class
{
      public:
      int triangle_area()
      {
        float result = 0.5*a*b;
        return result;
      }
};
    int main()
{
      Rectangle r;
      Triangle t;
      int length,breadth,base,height;
      cout << "Enter the length and breadth of a rectangle: " <<endl;
      cin>>length>>breadth;
      r.get_data(length,breadth);
      int m = r.rect_area();
      cout << "Area of the rectangle is : " <<m<<endl;
      cout << "Enter the base and height of the triangle: " <<endl;
      cin>>base>>height;
      t.get_data(base,height);
      float n = t.triangle_area();
      cout <<"Area of the triangle is : " << n<<endl;
      return 0;
}
```

Output:

```
Enter the length and breadth of a rectangle:
23
20
Area of the rectangle is : 460
Enter the base and height of the triangle:
2
5
Area of the triangle is : 5
```

C++ Hybrid Inheritance

Hybrid inheritance is a combination of more than one type of inheritance. Example of hybrid inheritance is combination of multilevel and hierarchical inheritance.



Let's see a simple example:

```
    #include
    <iostream> using
    namespace std;
    class A
{
      protected:
      int a;
      public:
      void
      get_a()
      {
        std::cout << "Enter the value of 'a' : " << std::endl;
        cin>>a;
      }
};

    class B : public A
{
      protected:
      int b;
      public:
      void
      get_b()
      {
        std::cout << "Enter the value of 'b' : " << std::endl;
       cin>>b;
      }
};
    class C
{
      protected:
      int c;
      public:
      void
      get_c()
      {
        std::cout << "Enter the value of c is : " << std::endl;
        cin>>c;
      }
};

    class D : public B, public C
{
      protecte
      d: int d;

      public:
      void
      mul()
      {
         get_a();
         get_b();
         get_c();
         std::cout << "Multiplication of a,b,c is : " <<a*b*c<< std::endl;
      }
};
    int main()
{
      D d;
```

```
        d.mul();
        return 0;
}
```

Output:

```
Enter the value of 'a' :
10
Enter the value of 'b' :
20
Enter the value of c is :
30
Multiplication of a,b,c is : 6000
```

Order of Constructor/ Destructor Call in Inheritance C++

Whenever we create an object of a class, the default constructor of that class is invoked automatically to initialize the members of the class.

If we inherit a class from another class and create an object of the derived class, it is clear that the default constructor of the derived class will be invoked but before that the default constructor of all of the base classes will be invoke, i.e the order of invokation is that the base class's default constructor will be invoked first and then the derived class's default constructor will be invoked.

Why the base class's constructor is called on creating an object of derived class?

To understand this you will have to recall your knowledge on inheritance. What happens when a class is inherited from other? The data members and member functions of base class comes automatically in derived class based on the access specifier but the definition of these members exists in base class only. So when we create an object of derived class, all of the members of derived class must be initialized but the inherited members in derived class can only be initialized by the base class's constructor as the definition of these members exists in base class only. This is why the **constructor of base class is called first to initialize all the inherited members**.

```
// C++ program to show the order of constructor call
// in single inheritance

#include <iostream>

using namespace std;

// base class

class Parent

 {
public:
// base class constructor
Parent()
{
                cout << "Inside base class" << endl;
}
 };
 // sub class
 class Child : public Parent
 {
public:

//sub class constructor

Child()

{
                cout << "Inside sub class" << endl;
}
 };
 // main function

 int main() {

// creating object of sub class

Child obj;

return 0;
 }
```

Output:

```
Inside base class Inside

sub class
```

Order of constructor call for Multiple Inheritance

For multiple inheritance order of constructor call is, the base class’s constructors are called in the order of inheritance and then the derived class’s constructor.

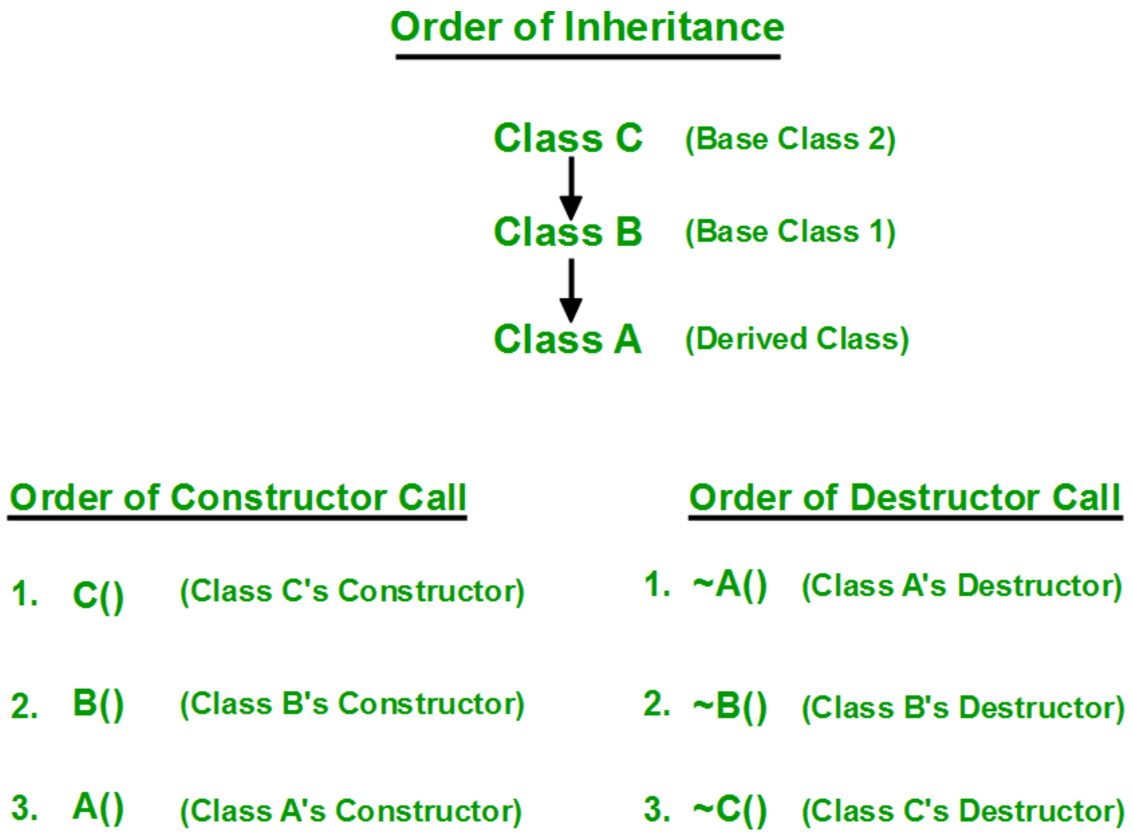
```
// C++ program to show the order of constructor calls
// in Multiple Inheritance

#include <iostream>

using namespace std;

// first base class

class Parent1
```

```
  {
public:
// first base class's Constructor

Parent1()

{
            cout << "Inside first base class" << endl;
}
  };
  // second base class

  class Parent2

  {
public:
// second base class's Constructor

Parent2()

{
            cout << "Inside second base class" << endl;
}
  };
  // child class inherits Parent1 and Parent2

  class Child : public Parent1, public Parent2

  {
public:
// child class's Constructor Child()

{
            cout << "Inside child class" << endl;
}
  };
  // main function

  int main() {

// creating object of class Child

Child obj1;

return 0;
}
```

Output:

```
Inside first base class Inside
second base class Inside
child  class
```

Order of constructor and Destructor call for a given order of Inheritance


Order of Inheritance
Class C
(Base Class 2)
Class B
(Base Class 1)
Class A
(Derived Class)
Order of Constructor Call
1. C() (Class C's Constructor)
2. B() (Class B's Constructor)
3. A() (Class A's Constructor)
Order of Destructor Call
1. ~A() (Class A's Destructor)
2. ~B() (Class B's Destructor)
3. ~C() (Class C's Destructor)

How to call the parameterized constructor of base class in derived class constructor?

To call the parameterized constructor of base class when derived class's parameterized constructor is called, you have to explicitly specify the base class's parameterized constructor in derived class as shown in below program:

```
#include<iostream>
using namespace std;
class parent
{
   int x;
   public:
   // parameterized constructor
```

```
    parent(int i)
    {
        x = i;
        cout << "Parent class Parameterized Constructor\n";
    }
};
class child: public parent
{
    int y;
    public:


    // parameterized constructor
    child(int j) : parent(j)        //Explicitly calling
    {
        y = j;
        cout << "Child class Parameterized Constructor\n";
    }
};
int main()
{
    child c(10);
    return 0;
}
```

Output

```
Parent class Parameterized Constructor

Child class Parameterized Constructor
```

Constructor call in multiple inheritance constructors

```
class C: public A, public B;
```

Constructors are called upon the order in which they are inherited
First class A constructors are executed followed by class B constructors, then class C constructors

Lab Tasks

Q1). Imagine a publishing company that markets both book and audio-cassette versions of its works. Create a class publication that stores the title and price of a publication. from this class derive two classes:

1. book, which adds a page count and
2. tape, which adds a playing time in minutes.
3. each of these three classes should have getdata() function to get its data from the user at the keyboard and a putdata() function to display its data.

Write a main() program to test the book and tape class by creating instances of them, asking the user to fill in their data with getdata() and then displaying the data with putdata().

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

class Publication{
    private:
        string title;
        double price;
    public:
        void getData(){
            cout << "Enter Title: "; cin >> title;
            cout << "Enter Price: "; cin >> price;
        }
        void putData(){
            cout << "Title: " << title << endl;
            cout << "Price: " << price << endl;
        }
};
class Book: public Publication{
    private:
        int pageCount;
    public:
        void getData(){
            cout << "Enter page count: "; cin >> pageCount;
        }
        void putData(){
            cout << "Page Count: " << pageCount << endl;
        }

};
class Tape: public Publication{
    private:
        int time;
    public:
        void getData(){
            cout << "Enter playing time: "; cin >> time;
        }
        void putData(){
            cout << "Playing Time: " << time << endl;
        }
};


int main(){
    Book b1;
    Tape t1;
    // Get Data
    cout << "####### BOOK DATA #######" << endl;
    b1.Publication::getData();
    b1.getData();

    cout << "####### TAPE DATA #######" << endl;
    t1.Publication::getData();
    t1.getData();

    // Put Data
    cout << "####### PRINT BOOK DATA #######" << endl;
    b1.putData();
    b1.Publication::putData();

    cout << "####### PRINT TAPE DATA #######" << endl;
    t1.putData();
    t1.Publication::putData();
    return 0;
}
```

Q2). Write a class Person that has attributes of id, name and address. It has a constructor to initialize, a member function to input and a member function to display data members. Create another class Student that inherits Person class. It has additional attributes of rollnumber and marks. It also has member function to input and display its data members.

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

class Person{
    private:
        int id;
        string name;
        string address;
    public:
        Person(): id(0), name(""), address("") { }

        void input(){
            cout << "Enter ID: "; cin >> id;
            cout << "Enter Name: "; cin >> name;
            cout << "Enter Address: "; cin >> address;
        }
        void output(){
            cout << "ID: " << id << endl;
            cout << "Name: " << name << endl;
            cout << "Address: " << address << endl;
        }
};
class Student: public Person{
    private:
        int rollNumber;
        int marks;
    public:
        void input(){
            cout << "Enter Roll No.: "; cin >> rollNumber;
            cout << "Enter Marks: "; cin >> marks;
        }
        void output(){
            cout << "Roll No.: " << rollNumber << endl;
            cout << "Marks: " << marks << endl;
        }
};

int main(){
    Student s1;
    // Input Data
    cout << "##### INPUT #####" << endl;
    s1.Person::input();
    s1.input();

    // Output Data
    cout << "##### OUTPUT #####" << endl;
    s1.Person::output();
    s1.output();

    return 0;
}
```

Q3). Write a base class Computer that contains data members of wordsize(in bits), memorysize (in megabytes), storagesize (in megabytes) and speed (in megahertz). Derive a Laptop class that is a kind of computer but also specifies the object's length, width, height, and weight. Member functions for both classes should include a default constructor, a constructor to inialize all components and a function to display data members.

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

class Computer{
    protected:
        int wordSize;
        int memorySize;
        int storageSize;
        int speed;
    public:
        // Constructors
        Computer(): wordSize(64), memorySize(2048), storageSize(10240), speed(24000) {
}
        Computer(int wS, int mS, int sS, int sp):
            wordSize(wS), memorySize(mS), storageSize(sS), speed(sp) {}

        void display(){
            cout << "Word Size: " << wordSize << " bits" << endl;
            cout << "Memory Size: " << memorySize << " MB" << endl;
            cout << "Storage Size: " << storageSize << " MB" << endl;
            cout << "Speed: " << speed << " Mhz" << endl;

        }
};
class Laptop: public Computer{
    private:
        int length, width, height;
        float weight;
    public:
        Laptop(): length(0), width(0), height(0), weight(0.0){};
        Laptop(int l, int wi, int h, int we ): length(l), width(wi), height(h), weight
(we) {}

        void display(){
            cout << "Length: " << length << " inches" << endl;
            cout << "Width: " << width << " inches" << endl;
            cout << "Height: " << height << " inches" << endl;
            cout << "Weight: " << weight << " kilogram" << endl;
            Computer::display();
        }
};
int main(){
    Laptop l1(54,78,30, 0.544);
    l1.display();
    return 0;
}
```

Home Activity

Q1). Write a program having a base class Student with data members rollno, name and Class define a member functions getdata() to input values and another function putdata() to display all values. A class Test is derived from class Student with data members T1marks, T2marks, T3marks, Sessional1, Sessional2, Assignment and Final. Also make a function getmarks() to enter marks for all variables except Final and also make a function putmarks() to display result. Make a function Finalresult() to calculate value for final variable using other marks. Then display the student result along with student data.

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

class Student{
    public:
        void getData();
        void putData();
    protected:
        string name;
        int rollNo;
};
class Test: public Student {
    public:
        void getMarks();
        void putMarks();
        void finalResult();
    protected:
    int t1Marks, t2Marks, t3Marks,
        sessional1, sessional2,
        assignment, finalMarks;
};
int main(){
    Test t1;

    t1.getData();
    t1.getMarks();
    t1.finalResult();

    return 0;
}
// ####################################################
//                STUDENT CLASS MATHODS
// ####################################################
void Student::getData(){
    cout << "######### INPUT STUDNET INFORMATION #########" << endl;
    cout << "Enter Name: "; cin >> name;
    cout << "Enter Roll No.: "; cin >> rollNo;
}
void Student::putData(){
    cout << "######### OUTPUT STUDNET INFORMATION #########" << endl;
    cout <<"Name: " << name << endl;
    cout << "Roll No.: " << rollNo << endl;
}
// ###################################################
//                  TEST CLASS MATHODS
// ###################################################
void Test::getMarks(){
    cout << "######### GETTING MARKS #########" << endl;
    cout << "Enter test 1 marks: "; cin >> t1Marks;
    cout << "Enter test 2 marks: "; cin >> t2Marks;
    cout << "Enter sessional 1 marks: "; cin >> sessional1;
    cout << "Enter sessional 2 marks: "; cin >> sessional2;
    cout << "Enter assignment marks: "; cin >> assignment;
}
void Test::putMarks(){
    cout << "######### PRINTING MARKS #########" << endl;
    cout << "Test 1: " << t1Marks << endl;
    cout << "Test 2: " << t2Marks << endl;
    cout << "Sessional 1: " << sessional1 << endl;
    cout << "Sessional 2: " << sessional2 << endl;
    cout << "Assignment: " << assignment << endl;
}
void Test::finalResult(){
    // Calculating final marks
    finalMarks = t1Marks + t2Marks + sessional1 + sessional2 + assignment;

    Student::putData();
    Test::putMarks();
    cout << "Final Marks: " << finalMarks << endl;
}
```

Q2). Write a program that declares two classes. The parent class is called Simple that has two data members num1 and num2 to store two numbers. It also has four member functions.

- The add() function adds two numbers and displays the result.
- The sub() function subtracts two numbers and displays the result.
- The mul() function multiplies two numbers and displays the result.
- The div() function divides two numbers and displays the result.

The child class is called Complex that overrides all four functions. Each function in the child class checks the value of data members. It calls the corresponding member function in the parent class if the values are greater than 0. Otherwise it displays error message.

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

class Simple {
    protected:
        int num1,num2;

        void add();
        void sub();
        void mul();
        void div();
};
class Complex: public Simple {
    public:
        Complex(int ,int);
        void add();
        void sub();
        void mul();
        void div();
        void error();
};
int main(){
    Complex c1(2,3);
    c1.add();
    c1.div();
    return 0;
}

// Simple
void Simple::add(){
    cout << "Addition: " << num1+num2 << endl;
}
void Simple::sub(){
    cout << "Subtration: " << num1-num2 << endl;
}
void Simple::mul(){
    cout << "Multiplication: " << num1*num2 << endl;
}
void Simple::div(){
    cout << "Division: " << num1/num2 << endl;
}
// Complex
Complex(int n1, int n2){
    num1 = n1;
    num2 = n2;
}
void Complex::add(){
    if(num1>0 && num2>0)
        Simple::add();
    else
        error();
}
void Complex::sub(){
    if(num1>0 && num2>0)
        Simple::sub();
    else
        error();
}
void Complex::mul(){
    if(num1>0 && num2>0)
        Simple::mul();
    else
        error();
}
void Complex::div(){
    if(num1>0 && num2>0)
        Simple::div();
    else
        error();
}
void Complex::error(){
    cout << "ERROR: Values should be greator then 0." << endl;
}
```

LAB NO. 11

PUBLIC, PROTECTED AND PRIVATE INHERITANCE IN C++

Lab Objectives

Following are the lab objectives:

1. Public, private and protected inheritance in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

Public, Protected and Private Inheritance in C++ Programming

In C++ inheritance, we can derive a child class from the base class in different access modes. For example,

```
class Base {
.... ... ....
};
class Derived : public Base {
.... ... ....
};
```

Notice the keyword public in the code

```
class Derived : public Base
```

This means that we have created a derived class from the base class in public mode. Alternatively, we can also derive classes in protected or private modes.

These 3 keywords (public, protected, and private) are known as access specifiers in C++ inheritance.

Public, protected and private inheritance in C++

Public, protected, and private inheritance have the following features:

- **Public inheritance** makes public members of the base class public in the derived class, and the protected members of the base class remain protected in the derived class.
- **Protected inheritance** makes the public and protected members of the base class protected in the derived class.
- **Private inheritance** makes the public and protected members of the base class private in the derived class.

Note: private members of the base class are inaccessible to the derived class.

```
class Base {
    public:
        int x;
    protected:
        int y;
    private:
        int z;
};
class PublicDerived: public Base {
     // x is public
     // y is protected
     // z is not accessible from PublicDerived
};
class ProtectedDerived: protected Base {
     // x is protected
     // y is protected
     // z is not accessible from ProtectedDerived
};
class PrivateDerived: private Base {
     // x is private
     // y is private
     // z is not accessible from PrivateDerived
}
```

Example 1: C++ public Inheritance

```
// C++ program to demonstrate the working of public inheritance
#include <iostream>
using namespace std; class
Base {
    private:
     int pvt = 1;
    protected:
     int prot = 2;
    public:
     int pub = 3;
     // function to access private member
     int getPVT() {
       return pvt;
     }
};
  class PublicDerived : public Base {
    public:
     // function to access protected member from Base
     int getProt() {
       return prot;
     }
};
  int main() {
     PublicDerived object1;
     cout << "Private = " << object1.getPVT() << endl;
     cout << "Protected = " << object1.getProt() << endl;
     cout << "Public = " << object1.pub << endl;
     return 0;
}
```

```
Output Private
= 1
Protected = 2
Public = 3
```

Here, we have derived PublicDerived from Base in public mode.

As a result, in PublicDerived:

prot is inherited as protected.

pub and getPVT() are inherited as public. pvt

is inaccessible since it is private in Base.

Since private and protected members are not accessible from main(), we need to create public functions getPVT() and getProt() to access them:

```
// Error: member "Base::pvt" is inaccessible
```

```
cout << "Private = " << object1.pvt;

// Error: member "Base::prot" is inaccessible

cout << "Protected = " << object1.prot;
```

Notice that the getPVT() function has been defined inside Base. But the getProt() function has been defined inside PublicDerived.

This is because pvt, which is private in Base, is inaccessible to PublicDerived.

However, prot is accessible to PublicDerived due to public inheritance. So, getProt() can access the protected variable from within PublicDerived.

Accessibility in public Inheritance

Accessibility	**private members**	**protected members**	**public members**
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes

Example 2: C++ protected Inheritance

```
// C++ program to demonstrate the working of protected inheritance

#include <iostream>

using namespace std; class

Base {

    private:

     int pvt = 1;

    protected:

     int prot = 2;

    public:

     int pub = 3;

     // function to access private member

     int getPVT() {

        return pvt;

     }

};

class ProtectedDerived : protected Base {
    public:
     // function to access protected member from Base
     int getProt() {
        return prot;
     }
     // function to access public member from Base
     int getPub() {
        return pub;
     }
};
  int main() {
     ProtectedDerived object1;
     cout << "Private cannot be accessed." << endl;
     cout << "Protected = " << object1.getProt() << endl;
     cout << "Public = " << object1.getPub() << endl;
     return 0;
}
```

Output

```
Private cannot be accessed.
Protected = 2
Public = 3
```

Here, we have derived ProtectedDerived from Base in protected mode.

As a result, in ProtectedDerived:

prot, pub and getPVT() are inherited as protected.

pvt is inaccessible since it is private in Base.

As we know, protected members cannot be directly accessed from outside the class. As a result, we cannot use getPVT() from ProtectedDerived.

That is also why we need to create the getPub() function in ProtectedDerived in order to access the pub variable.

```
// Error: member "Base::getPVT()" is inaccessible
```

```
cout << "Private = " << object1.getPVT();
  // Error: member "Base::pub" is
  inaccessible cout << "Public = " <<
  object1.pub;
```

Accessibility in protected Inheritance

Accessibility	private members	protected members	public members
Base Class	Yes	Yes	Yes
Derived Class	No	Yes	Yes (inherited as protected variables)

Example 3: C++ private Inheritance

```
// C++ program to demonstrate the working of private inheritance
#include <iostream>
using namespace std;
class Base {
    private:
     int pvt = 1;
    protected:
     int prot = 2;
    public:
     int pub = 3;
     // function to access private member
     int getPVT() {
         return pvt;
     }
};
  class PrivateDerived : private Base {
    public:
```

```
    // function to access protected member from Base
    int getProt() {
        return prot;
    }
    // function to access private member
    int getPub() {
        return pub;
    }
};
  int main() {
    PrivateDerived object1;
    cout << "Private cannot be accessed." << endl;
    cout << "Protected = " << object1.getProt() << endl;
    cout << "Public = " << object1.getPub() << endl;
    return 0;
}
```

Output

```
Private cannot be accessed.
Protected = 2
Public = 3
```

Here, we have derived PrivateDerived from Base in private mode.

As a result, in PrivateDerived:

prot, pub and getPVT() are inherited as private.

pvt is inaccessible since it is private in Base.

As we know, private members cannot be directly accessed from outside the class. As a result, we cannot use getPVT() from PrivateDerived.

That is also why we need to create the getPub() function in PrivateDerived in order to access the pub variable.

// Error: member "Base::getPVT()" is

inaccessible cout << "Private = " <<

object1.getPVT();

// Error: member "Base::pub" is

inaccessible cout << "Public = "

<< object1.pub; **Accessibility**

in private Inheritance

Accessibility	**private members**	**protected members**	**public members**
Base Class	Yes	Yes	Yes
Derived Class	No	Yes (inherited as private variables)	Yes (inherited as private variables)

Lab Tasks

Q1). Imagine a publishing company that markets both book and audio-cassette versions of its works. Create a class publication that stores the title and price of a publication. From this class derive two **public** classes:

1. book, which adds a page count and
2. tape, which adds a playing time in minutes.
3. each of these three classes should have getdata() function to get its data from the user at the keyboard and a putdata() function to display its data.

Write a main() program to test the book and tape class by creating instances of them, asking the user to fill in their data with getdata() and then displaying the data with putdata().

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

class Publication{
    private:
        string title;
        double price;
    public:
        void getData(){
            cout << "Enter Title: "; cin >> title;
            cout << "Enter Price: "; cin >> price;
        }
        void putData(){
            cout << "Title: " << title << endl;
            cout << "Price: " << price << endl;
        }
```

```
};
class Book: private Publication{
    private:
        int pageCount;
    public:
        void getData(){
            Publication::getData();
            cout << "Enter page count: "; cin >> pageCount;
        }
        void putData(){
            Publication::putData();
            cout << "Page Count: " << pageCount << endl;
        }

};
class Tape: private Publication{
    private:
        int time;
    public:
        void getData(){
            Publication::getData();
            cout << "Enter playing time: "; cin >> time;
        }
        void putData(){
            Publication::putData();
            cout << "Playing Time: " << time << endl;
        }
};


int main(){
    Book b1;
    Tape t1;
    // Get Data
    cout << "####### BOOK DATA #######" << endl;
    b1.getData();

    cout << "####### TAPE DATA #######" << endl;

    t1.getData();

    // Put Data
    cout << "####### PRINT BOOK DATA #######" << endl;
    b1.putData();

    cout << "####### PRINT TAPE DATA #######" << endl;
    t1.putData();

    return 0;
```

```
}
```

Q2). Write a class Person that has attributes of id, name and address. It has a constructor to initialize, a member function to input and a member function to display data members. Create another **protected** class Student that inherits Person class. It has additional attributes of rollnumber and marks. It also has member function to input and display its data members.

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

class Person{
    private:
        int id;
        string name;
        string address;
    public:
        Person(): id(0), name(""), address("") { }

        void input(){
            cout << "Enter ID: "; cin >> id;
            cout << "Enter Name: "; cin >> name;
            cout << "Enter Address: "; cin >> address;
        }
        void output(){
            cout << "ID: " << id << endl;
            cout << "Name: " << name << endl;
            cout << "Address: " << address << endl;
        }
};
class Student: protected Person{
    private:
        int rollNumber;
        int marks;
    public:
        void input(){
            Person::input();
            cout << "Enter Roll No.: "; cin >> rollNumber;
            cout << "Enter Marks: "; cin >> marks;
        }
        void output(){
            Person::output();
            cout << "Roll No.: " << rollNumber << endl;
            cout << "Marks: " << marks << endl;
        }
};

int main(){
    Student s1;
    // Input Data
    cout << "##### INPUT #####" << endl;
    s1.input();

    // Output Data
    cout << "##### OUTPUT #####" << endl;;
    s1.output();

    return 0;
}
```

Q3). Write a base class Computer that contains data members of wordsize(in bits), memorysize (in megabytes), storagesize (in megabytes) and speed (in megahertz). Derive a **private** Laptop class that is a kind of computer but also specifies the object's length, width, height, and weight. Member functions for both classes should include a default constructor, a constructor to inialize all components and a function to display data members.

Note: In all lab tasks data members and member functions must declare with three specifier **public**, **protected** and **private.**

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

class Computer{
    protected:
        int wordSize;
        int memorySize;
        int storageSize;
        int speed;
    public:
        // Constructors
        Computer(): wordSize(64), memorySize(2048), storageSize(10240), speed(24000
) {}
        Computer(int wS, int mS, int sS, int sp):
            wordSize(wS), memorySize(mS), storageSize(sS), speed(sp) {}

        void display(){
            cout << "Word Size: " << wordSize << " bits" << endl;
            cout << "Memory Size: " << memorySize << " MB" << endl;
            cout << "Storage Size: " << storageSize << " MB" << endl;
            cout << "Speed: " << speed << " Mhz" << endl;

        }
};
class Laptop: private Computer{
    private:
        int length, width, height;
        float weight;
    public:
        Laptop(): length(0), width(0), height(0), weight(0.0){};
        Laptop(int l, int wi, int h, int we ): length(l), width(wi), height(h), wei
ght(we) {}

        void display(){
            cout << "Length: " << length << " inches" << endl;
            cout << "Width: " << width << " inches" << endl;
            cout << "Height: " << height << " inches" << endl;
            cout << "Weight: " << weight << " kilogram" << endl;
            Computer::display();
        }
};
int main(){
    Laptop l1(54,78,30, 0.544);
    l1.display();
    return 0;
}
```

Home Activity

Q1). Write a program having a base class Student with data members rollno, name and Class define a member functions getdata() to input values and another function putdata() to display all values. A class Test is derived from class Student with data members T1marks, T2marks, T3marks, Sessional1, Sessional2, Assignment and Final. Also make a function getmarks() to enter marks for all variables except Final and also make a function putmarks() to display result. Make a function Finalresult() to calculate value for final variable using other marks. Then display the student result along with student data.

Q2). Write a program that declares two classes. The parent class is called Simple that has two data members num1 and num2 to store two numbers. It also has four member functions.

- The add() function adds two numbers and displays the result.
- The sub() function subtracts two numbers and displays the result.
- The mul() function multiplies two numbers and displays the result.
- The div() function divides two numbers and displays the result.

The child class is called Complex that overrides all four functions. Each function in the child class checks the value of data members. It calls the corresponding member function in the parent class if the values are greater than 0. Otherwise it displays error message.

Note: In all home tasks data members and member functions must declare with three specifier **public**, **protected** and **private**.

Lab No. 12

Header files in C++

Lab Objectives

Following are the lab objectives:

1. Header files in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

Header files in C++

C++ offers its users a variety of functions, one of which is included in header files. In C++, all the header files may or may not end with the ".h" extension but in C, all the header files must necessarily end with the ".h" extension.
A header file contains:

- Function definitions
- Data type definitions

It offers the above features by importing them into the program with the help of a preprocessor directive "#include". These preprocessor directives are used for instructing compiler that these files need to be processed before compilation.
In C++ program has the header file which stands for input and output stream used to take input with the help of "cin" and "cout" respectively.
There are of 2 types of header file:

- **Pre-existing header files:** Files which are already available in C/C++ compiler we just need to import them.
- **User-defined header files:** These files are defined by the user and can be imported using "#include".

Syntax:

```
#include <filename.h>

or

#include "filename.h"
```

We can include header files in our program by using one of the above two syntax whether it is pre-defined or user-defined header file. The "#include" preprocessor is responsible for directing the compiler that the header file needs to be processed before compilation and includes all the necessary data type and function definitions.
Note: We can't include the same header file twice in any program.

Create your own Header File:

Instead of writing a large and complex code, we can create your own header files and include them in our program to use it whenever we want. It enhances code functionality and readability. Below are the steps to create our own header file:

Example 1

1. Write your own C/C++ code and save that file with ".h" extension. Below is the illustration of header file:

```
// Function to find the sum of two
// numbers passed
int sumOfTwoNumbers(int a, int b)
{
    return (a + b);
}
```

2. Include your header file with “#include” in your C/C++ program as shown below:

```
// C++ program to find the sum of two
// numbers using function declared in
// header file
#include "iostream"
// Including header file
#include "sum.h" using
namespace std;
// Driver Code
int main()
{
    // Given two numbers
    int a = 13, b = 22;
    // Function declared in header
    // file to find the sum
    cout << "Sum is: "
        << sumOfTwoNumbers(a, b)
        << endl;
}
```

Output

```
Sum is: 35
```

Example 2

For our Rectangle class, the header file looks like this:

```
// Rectangle.p

class Rectangle
{

private:
    int height;

    int width;

public:
    Rectangle();

    Rectangle(int w, int h);

    void setWidth(int);

    int getWidth();

    void setHeight(int);

    int getHeight();

    int area();
};
```

The implementation file looks like this:

```
// Rectangle.cpp

#include "Rectangle.hpp"

// default constructor

Rectangle::Rectangle()

{
    width = 0;

    height = 0;

}
// constructor that takes arguments for width and height Rectangle::Rectangle(int

w, int h)

{
    width = w;
    height = h;
}

void Rectangle::setWidth(int w)
{
    width = w;

}
int Rectangle::getWidth()
{

    return width;
}
void Rectangle::setHeight(int h)

{
    height = h;
}

int Rectangle::getHeight()
{
    return height;

}

int Rectangle::area()

{
    return width * height;
}
```

Using your class in main and other classes
In any file where you want to use your class, just include its header file.If you use your class in the main C++ file, you would include your header in the .cpp file for main.

```
// main.cpp

#include "Rectangle.h" int

main()

{

   Rectangle rec;

   Rectangle rec(5,6);
          cout << rec.area(); int n1

          = 10, n2=10;

          rec.setHeight(n1);

          rec.setWidth(n2); cout

          << rec.area();

}
```

Lab Tasks

Q1). Write a C++ program to create student class (in header file as .h) that contains attributes of the student name, roll no, and total-marks. Write two functions to get and display these attributes.

// student.h

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

class Student{
    private:
        string name;
        int rollno,marks;
    public:
        void set(string n,int r, int m){
            name=n;
            rollno=r;
            marks=m;
        }
        void display(){
            cout << "Name: " << name << endl;
            cout << "Roll No.: " << rollno << endl;
            cout << "Marks: " << marks << endl;
        }
};
// main.cpp
#include "student.h"

using namespace std;

int main(){
    Student s1;
    s1.set("Zafeer", 22,5);
    s1.display();
}
```

Q2). Write a C++ program to create a class name as arithmetic_operations (in header file as .h) with two attributes (number 1 and number 2). Write four functions addition, subtraction, multiplication and division. Program menu should be user friendly.

```
// header.h
class ArithmeticOperations{
    public:
        int num1, num2;
        int addition(){
            return num1+num2;
        }
        int subtraction(){
            return num1-num2;
        }
        int multiplication(){
            return num1*num2;
        }
        int division(){
            return num1/num2;
        }
};

// main.cpp
#include <iostream>
#include "header.h"

using namespace std;

int main(){
    ArithmeticOperations a1;
    a1.num1 =20;
    a1.num2=5;

    cout << "Addition: " << a1.addition() << endl;
    cout << "Subtraction: " << a1.subtraction() << endl;
    cout << "Multiplication: " << a1.multiplication() << endl;
    cout << "Division: " << a1.division() << endl;

    return 0;
}
```

Home Activity

Q3). Create a class named “Student” (in header file as .h) with a string variable “name” and an integer variable “roll_no”. Assign the value of roll_no as “101” and that of name as "Ali" by creating an object of the class Student.

// header.h

```
#include <iostream>
class Student{
    public:
        std::string name;
        int rollNo;
        Student(std::string n, int r): name(n), rollNo(r) { }
};
```

// main.cpp

```
#include "header.h"
```

```
using namespace std;

int main(){
    Student s1("Ali", 101);
    cout << "Name: " << s1.name << endl;
    cout << "Roll Number: " << s1.rollNo << endl;
    return 0;
}
```

Q4). Write a C++ program to print the area of a rectangle by creating a class named 'Area' (in header file as .h) having two functions. First function named as “Set_Dim” takes the length and breadth of the rectangle as parameters and the second function named as 'Get_Area' returns the area of the rectangle. Length and breadth of the rectangle are entered through user.

// header.h

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

class Area{
    private:
        double length;
        double breadth;
    public:
        void setDim(){
            cout << "Enter Length: "; cin >> length;
            cout << "Enter Breadth: "; cin >> breadth;
        }
        double getArea(){
            return length*breadth;
        }
};
```

// main.cpp

```
#include "004.header.h"

int main(){
    Area a1;
    a1.setDim();

    cout << "Area: " << a1.getArea() << endl;
    return 0;
}
```

LAB NO. 13

POINTER AND DYNAMIC MEMORY IN C++

Lab Objectives

Following are the lab objectives:

1. Pointers and Dynamic memory in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

C++ Pointers

Pointers are used to store addresses rather than values.

Here is how we can declare pointers.

int *pointVar;

Here, we have declared a pointer pointVar of the int type. We

can also declare pointers in the following way.

int* pointVar; // preferred syntax

Let's take another example of declaring pointers. int*

pointVar, p;

Here, we have declared a pointer pointVar and a normal variable p. Note:

The * operator is used after the data type to declare pointers. **Assigning**

Addresses to Pointers

Here is how we can assign addresses to pointers:

int* pointVar, var;

var = 5; // assign address of var to pointVar pointer

pointVar = &var;

Here, 5 is assigned to the variable var. And, the address of var is assigned to the pointVar pointer with the code pointVar = &var.

Get the Value from the Address Using Pointers

To get the value pointed by a pointer, we use the * operator. For example: int*

pointVar, var;

var = 5;

// assign address of var to pointVar

pointVar = &var;

// access value pointed by pointVar

cout << *pointVar << endl; // Output: 5

In the above code, the address of var is assigned to pointVar. We have used the *pointVar to get the value stored in that address.

When * is used with pointers, it's called the dereference operator. It operates on a pointer and gives the value pointed by the address stored in the pointer. That is, *pointVar = var.

Note: In C++, pointVar and *pointVar is completely different. We cannot do something like *pointVar = &var;

Example 1: Working of C++ Pointers

```
#include        <iostream>

using  namespace  std;  int

main() {

        int var = 5;

        // declare pointer variable

        int* pointVar;

        // store address of var

        pointVar = &var;

        // print value of var

        cout << "var = " << var << endl;

        // print address of var

        cout << "Address of var (&var) = " << &var << endl

           << endl;

        // print pointer pointVar

        cout << "pointVar = " << pointVar << endl;

        // print the content of the address pointVar points to

        cout << "Content of the address pointed to by pointVar (*pointVar) = " << *pointVar << endl;

        return 0;

}
```

Output

```
var = 5

Address of var (&var) = 0x61ff08

pointVar = 0x61ff08

Content of the address pointed to by pointVar (*pointVar) = 5
```

Example 2: Changing Value Pointed by Pointers

```
#include <iostream>
```

```
using namespace std;
int main() {
    int var = 5;
    int* pointVar;

    // store address of var
    pointVar = &var;

    // print var
    cout << "var = " << var << endl;

    // print *pointVar
    cout << "*pointVar = " << *pointVar << endl
         << endl;

    cout << "Changing value of var to 7:" << endl;

    // change value of var to 7
    var = 7;

    // print var
    cout << "var = " << var << endl;

    // print *pointVar
    cout << "*pointVar = " << *pointVar << endl
         << endl;

    cout << "Changing value of *pointVar to 16:" << endl;

    // change value of var to 16
    *pointVar = 16;

    // print var
    cout << "var = " << var << endl;

    // print *pointVar
    cout << "*pointVar = " << *pointVar << endl;
    return 0;
}
```

Output

```
var = 5
*pointVar = 5

Changing value of var to 7:
var = 7
*pointVar = 7

Changing value of *pointVar to 16:
var = 16
*pointVar = 16
```

Pointer to Class in C++

A class pointer is a pointer variable that stores address of an object of a class. As shown in the above diagram we have a class Rectangle with 2 data members and 1 member function. We have also created an object of that class named var1. Now we create a pointer variable *ptr of type Rectangle and assign the address of the object var1 to this pointer variable. As shown in the diagram the address of the object var1 is stored in the pointer variable ptr. Let us see an example of the same:

Example

```
 #include  <iostream>

 using namespace std;

 class Rectangle

 {

private:

            int length;

            int breadth;

public:

            Rectangle(int l, int b)

                    length=l;

                    breadth=b;

            }

            int getArea()

            {

                    return 2*length*breadth;
            }
 };
 int main()
 {
   // creating an object of Rectangle
   Rectangle var1(5,2); // parameterized constrcutor
   /* creating a pointer of Rectangle type &
   assigning address of var1 to this pointer */
   Rectangle* ptr = &var1;
   /* calculating area of rectangle by using pointer
   ptr to call the objects getArea() function
   */
   int area = ptr->getArea();
  cout<<"Area of rectangle is: "<<area;
  return 0;
 }
```

Pointer to Class Array in C++

As we just saw pointer to class, a pointer can also point to an an array of class. That is it can be used to access the elements of an array of type class.

```
 #include  <iostream>
 using namespace std;
 class Rectangle
 {
private:
            int length;
            float breadth;
public:
            void setData(int l, int b)
            {
                    length=l;
```

```
                        breadth=b;
                }
                int getArea()
                {
                        return 2*length*breadth;
                }
        };

        int main()
        {
          // creating an object array of Rectangle
          Rectangle var[2];
           // setting values of array elements
          var[0].setData(5,2);
          var[1].setData(3,2);
          /* creating a pointer of Rectangle type &
          assigning address of var to this pointer */
          Rectangle* ptr;
          ptr = v
          /* calculating area of rectangles by using pointer
          ptr to call the objects getArea() function
          */
          for(int i=0;i<2;i++)
          {
       cout<<"Area of Rectangle"<<(i+1)<<" : "<<(ptr+i)->getArea()<<endl;
          }
         return 0;
        }
```

Output

```
Area of Rectangle1 : 20
```

Area of Rectangle2 : 12

Dynamic Variables

Variables that are created during program execution are called dynamic variables. With the help of pointers, C++ creates dynamic variables. C++ provides two operators, new and delete, to create and destroy dynamic variables, respectively. When a program requires a new variable, the operator new is used. When a program no longer needs a dynamic variable, the operator delete is used.

C++ new Operator

The new operator allocates memory to a variable. For example,

```
// declare an int pointer
int* pointVar;

// dynamically allocate memory
// using the new keyword
pointVar = new int;

// assign value to allocated memory
*pointVar = 45;
```

Here, we have dynamically allocated memory for an int variable using the new operator. Notice that we have used the pointer pointVar to allocate the memory dynamically. This is because the new operator returns the address of the memory location. In the case of an array, the new operator returns the address of the first element of the array. From the example above, we can see that the syntax for using the new operator is

```
pointerVariable = new dataType;
```

Delete Operator

Once we no longer need to use a variable that we have declared dynamically, we can deallocate the memory occupied by the variable. For this, the delete operator is used. It returns the memory to the operating system. This is known as **memory deallocation**. The syntax for this operator is

```
delete pointerVariable;
```

Consider the code:

```
// declare an int pointer
int* pointVar;
```

```
// dynamically allocate memory
// for an int variable
pointVar = new int;

// assign value to the variable memory
*pointVar = 45;

// print the value stored in memory
cout << *pointVar; // Output: 45

// deallocate the memory
delete pointVar;
```

Here, we have dynamically allocated memory for an int variable using the pointer pointVar. After printing the contents of pointVar, we deallocated the memory using delete.

Note: If the program uses a large amount of unwanted memory using new, the system may crash because there will be no memory available for the operating system. In this case, the delete operator can help the system from crash.

Example 1: C++ Dynamic Memory Allocation

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

int main() {
    // declare an int pointer
    int* pointInt;

    // declare a float pointer
    float* pointFloat;

    // dynamically allocate memory
    pointInt = new int;
    pointFloat = new float;

    // assigning value to the memory
    *pointInt = 45;
    *pointFloat = 45.45f;

    cout << *pointInt << endl;
    cout << *pointFloat << endl;
    // deallocate the memory
    delete pointInt;
    delete pointFloat;

    return 0;
}
```

Output

```
45
45.45
```

In this program, we dynamically allocated memory to two variables of int and float types. After assigning values to them and printing them, we finally deallocate the memories using the code

```
delete pointInt;
delete pointFloat;
```

Note: Dynamic memory allocation can make memory management more efficient.

Especially for arrays, where a lot of the times we don't know the size of the array until the run time.

Example 2: C++ new and delete Operator for Arrays

```
// C++ Program to store GPA of n number of students and display it
// where n is the number of students entered by the user

#include <iostream>
using namespace std;

int main(){
    int num;
    cout << "Enter total number of students: ";
    cin >> num;
    float* ptr;

    // memory allocation of num number of floats
    ptr = new float[num];

    cout << "Enter GPA of students." << endl;
    for (int i = 0; i < num; ++i) {
        cout << "Student" << i + 1 << ": ";
        cin >> *(ptr + i);
    }

    cout << "\nDisplaying GPA of students." << endl;
    for (int i = 0; i < num; ++i) {
        cout << "Student" << i + 1 << " :" << *(ptr + i) << endl;
    }

    // ptr memory is released
    delete[] ptr;

    return 0;
}
```

Output

```
Enter total number of students: 4
Enter GPA of students.
Student1: 3.6
Student2: 3.1
Student3: 3.9
Student4: 2.9

Displaying GPA of students.
Student1 :3.6
Student2 :3.1
Student3 :3.9
Student4 :2.9
```

In this program, we have asked the user to enter the number of students and store it in the num variable.

Then, we have allocated the memory dynamically for the float array using new. We enter data into the array (and later print them) using pointer notation.

After we no longer need the array, we deallocate the array memory using the code delete[] ptr;.

Notice the use of [] after delete. We use the square brackets [] in order to denote that the memory deallocation is that of an array.

Example 3: C++ new and delete Operator for Objects

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

class Student {
    int age;

   public:

    // constructor initializes age to 12
    Student() : age(12) {}

    void getAge() {
        cout << "Age = " << age << endl;
    }
};

int main() {

    // dynamically declare Student object
    Student* ptr = new Student();

    // call getAge() function
    ptr->getAge();

    // ptr memory is released
    delete ptr;

    return 0;
}
```

Output

```
Age =  12
```

In this program, we have created a Student class that has a private variable age.

We have initialized age to 12 in the default constructor Student() and print its value with the function getAge().

In main(), we have created a Student object using the new operator and use the pointer ptr to point to its address.

The moment the object is created, the Student() constructor initializes age to 12.

We then call the getAge() function using the code:

ptr->getAge();

Notice the arrow operator ->. This operator is used to access class members using pointers.

Lab Tasks

Q1. Write a function which will take pointer and display the number on screen. Take number from user and print it on screen using that function.

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

void display(int *ptr){
    cout << "You Entered: " << *ptr << endl;
}
int main(){
    int inpNum;
    int *ptr;

    ptr = &inpNum;

    cout << "Enter a number: "; cin >> inpNum;

    display(ptr);
    return 0;
}
```

Q2. Write a program to print the roll number and average marks of 8 students in three subjects (each out of 100). The marks are entered by the user and the roll numbers are automatically assigned. (using pointer type array)

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

int main(){
    // Creating Arrays with dynamic memory allocation
    int *subMarks = new int[3];
    int *rollNums = new int[8];
    float *avgMarks = new float[8];

    // Geting Data of 8 students
    for(int studentNum=1;studentNum<=8; studentNum++){
        // This variable is like a temporary
        // vaiable which will store average value.
        float average=0.0f;
        cout << "Student: " << studentNum << endl;

        // Get Marks from users.
        for(int marks=1; marks<=3; marks++){
            cout << "Marks " << marks << ": "; cin >> *(subMarks + marks - 1);
        }

        // Calculate Average
        for(int i=0; i<3; i++){
            average += *(subMarks + i);
        }
        *(avgMarks + studentNum-1)=average/3;

        // Roll Number: Student Number + CONSTANT VALUE + Average marks of that student
        *(rollNums+studentNum-1) = studentNum+678643+*(avgMarks + studentNum-1);
    }

    // Display Average
    for(int i=0; i<8; i++){
        cout <<"###################"<<endl;
        cout << "Student: " << i+1 << endl;
        cout <<"###################"<<endl;

        cout << "Average: " << *(avgMarks+i) << endl;
        cout << "Roll Number: " << *(rollNums+i) << endl;
    }

    // Deallocating memory
    delete[] subMarks;
    delete[] rollNums;
    delete[] avgMarks;
    return 0;
}
```

Q3). Write a C++ program to create student class that contains attributes of the student name, roll no, and total-marks. Write two functions to get and display these attributes. Create a pointer object of student class and invoke the get and display function using this pointer object.

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

class Student{
    private:
        string name;
        int rollNum;
        int totalMarks;
    public:
        void input(){
            cout << "Name: "; cin >> name;
            cout << "Roll Number: "; cin >> rollNum;
            cout << "Total Marks: "; cin >> totalMarks;
        }
        void display(){
            cout << "Name: " << name << endl;
            cout << "Roll Number: " << rollNum << endl;
            cout << "Total Marks: " << totalMarks << endl;

        }
};
int main(){
    Student* stPtr = new Student();
    stPtr->input();
    stPtr->display();

    delete stPtr;
    return 0;
}
```

Lab No. 14

UML CLASS DIAGRAM, COMPOSITION, AGGREGATION, ASSOCIATION IN C++

Lab Objectives

Following are the lab objectives:

1. UML class diagram, composition, aggregation and association relationship in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

What is Class in UML Diagram?

A Class in UML diagram is a blueprint used to create an object or set of objects. The Class defines what an object can do. It is a template to create various objects and implement their behavior in the system. A Class in UML is represented by a rectangle that includes rows with class names, attributes, and operations.

What is Class Diagram?

A Class Diagram in Software engineering is a static structure that gives an overview of a software system by displaying classes, attributes, operations, and their relationships between each other. This Diagram includes the class name, attributes, and operation in separate designated compartments. Class Diagram helps construct the code for the software application development.

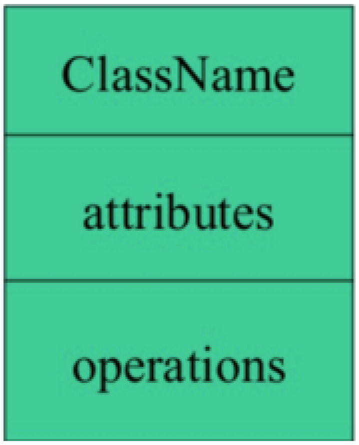
Class Diagram defines the types of objects in the system and the different types of relationships that exist among them. It gives a high-level view of an application. This modeling method can run with almost all Object-Oriented Methods. A class can refer to another class. A class can have its objects or may inherit from other classes.

Essential elements of A UML class diagram

1. Class Name
2. Attributes
3. Operations

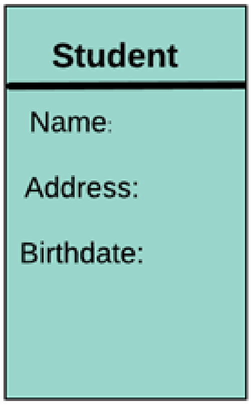
Class Name

The name of the class is only needed in the graphical representation of the class. It appears in the topmost compartment. A class is the blueprint of an object which can share the same relationships, attributes, operations, & semantics. The class is rendered as a rectangle, including its name, attributes, and operations in sperate compartments.

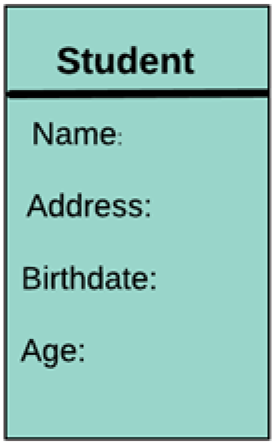


Attributes:

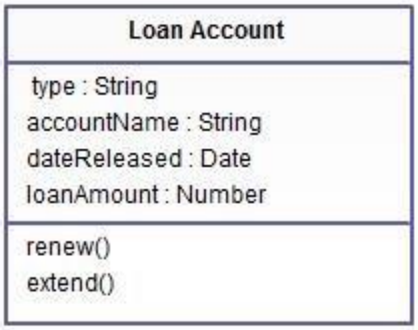
An attribute is named property of a class which describes the object being modeled. In the class diagram, this component is placed just below the name-compartment.



A derived attribute is computed from other attributes. For example, an age of the student can be easily computed from his/her birth date.



Operations/ Methods



Relationships

There are mainly three kinds of relationships in UML:

1. Associations
2. Generalizations

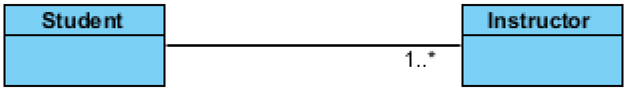
Association

If two classes in a model need to communicate with each other, there must be a link between them, and that can be represented by an association (connector).

Association can be represented by a line between these classes with an arrow indicating the navigation direction. In case an arrow is on both sides, the association is known as a bidirectional association.

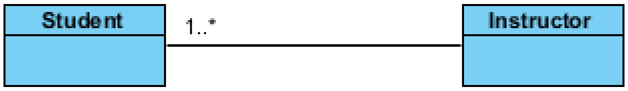
We can indicate the multiplicity of an association by adding multiplicity adornments to the line denoting the association. The example indicates that a Student has one or more Instructors:

A single student can associate with multiple teachers:



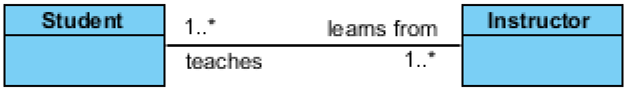
Association multiplicity example 1

The example indicates that every Instructor has one or more Students:



Association multiplicity example 2

We can also indicate the behavior of an object in an association (i.e., the role of an object) using role names.



Association multiplicity example 3

Association vs Aggregation vs Composition

The question "What is the difference between association, aggregation, and composition" has been frequently asked lately.

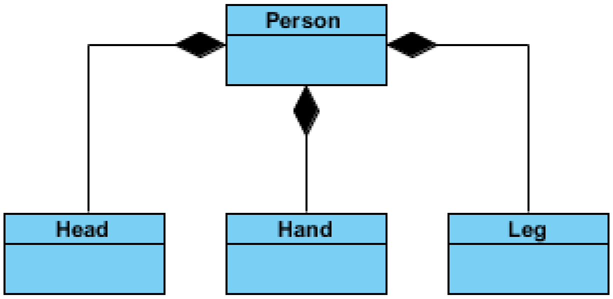
Aggregation and Composition are subsets of association meaning they are specific cases of association. In both aggregation and composition object of one class "owns" object of another class. But there is a subtle difference:

Aggregation implies a relationship where the child can exist independently of the parent. Example: Class (parent) and Student (child). Delete the Class and the Students still exist.

Composition implies a relationship where the child cannot exist independent of the parent. Example: House (parent) and Room (child). Rooms don't exist separate to a House.

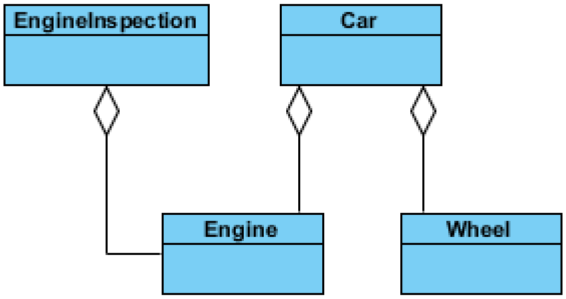
Composition Example:

We should be more specific and use the composition link in cases where in addition to the part-of relationship between Class A and Class B - there's a strong lifecycle dependency between the two, meaning that when Class A is deleted then Class B is also deleted as a result



Aggregation Example:

It's important to note that the aggregation link doesn't state in any way that Class A owns Class B nor that there's a parent-child relationship (when parent deleted all its child's are being deleted as a result) between the two. Actually, quite the opposite! The aggregation link is usually used to stress the point that Class A instance is not the exclusive container of Class B instance, as in fact the same Class B instance has another container/s.

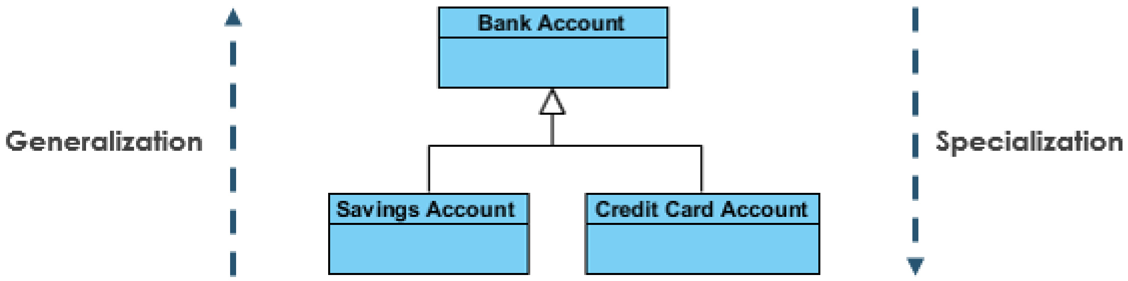


Generalization vs Specialization

Generalization is a mechanism for combining similar classes of objects into a single, more general class. Generalization identifies commonalities among a set of entities. The commonality may be of attributes, behavior, or both. In other words, a superclass has the most general attributes, operations, and relationships that may be shared with subclasses. A subclass may have more specialized attributes and operations.

Specialization is the reverse process of Generalization means creating new sub-classes from an existing class.

For Example, a Bank Account is of two types - Savings Account and Credit Card Account. Savings Account and Credit Card Account inherit the common/ generalized properties like Account Number, Account Balance, etc. from a Bank Account and also have their specialized properties like unsettled payment etc.



Generalization vs Inheritance

Generalization is the term that we use to denote abstraction of common properties into a base class in UML. The UML diagram's Generalization association is also known as Inheritance. When we implement Generalization in a programming language, it is often called Inheritance instead. Generalization and inheritance are the same. The terminology just differs depending on the context where it is being used.

Program example: how to use C++ Composition in programming:

```
#include  <iostream>

using namespace std;

class X

{

   private:

   int d;

   public:

   void set_value(int k)

   {

      d=k;

   }

   void show_sum(int n)

   {

      cout<<"sum of "<<d<<" and "<<n<<" = "<<d+n<<endl;

   }

};

class Y

{
  public:
  X a;
  void print_result()
  {
    a.show_sum(5);

  }

};
int main()
{
 Y b;
 b.a.set_value(20);
 b.a.show_sum(100);
 b.print_result();

}
```

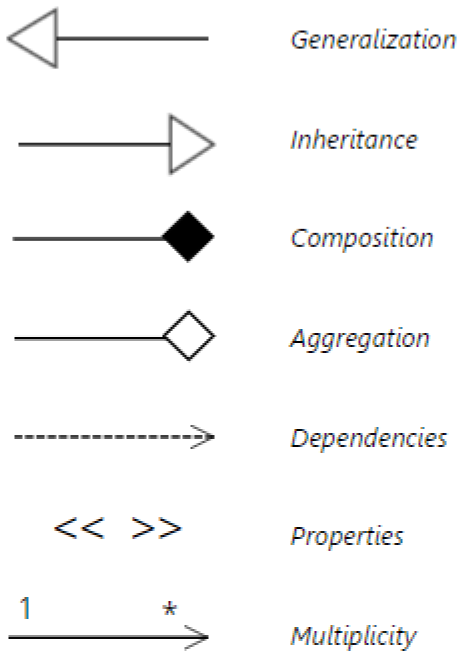
Programming Explanation:

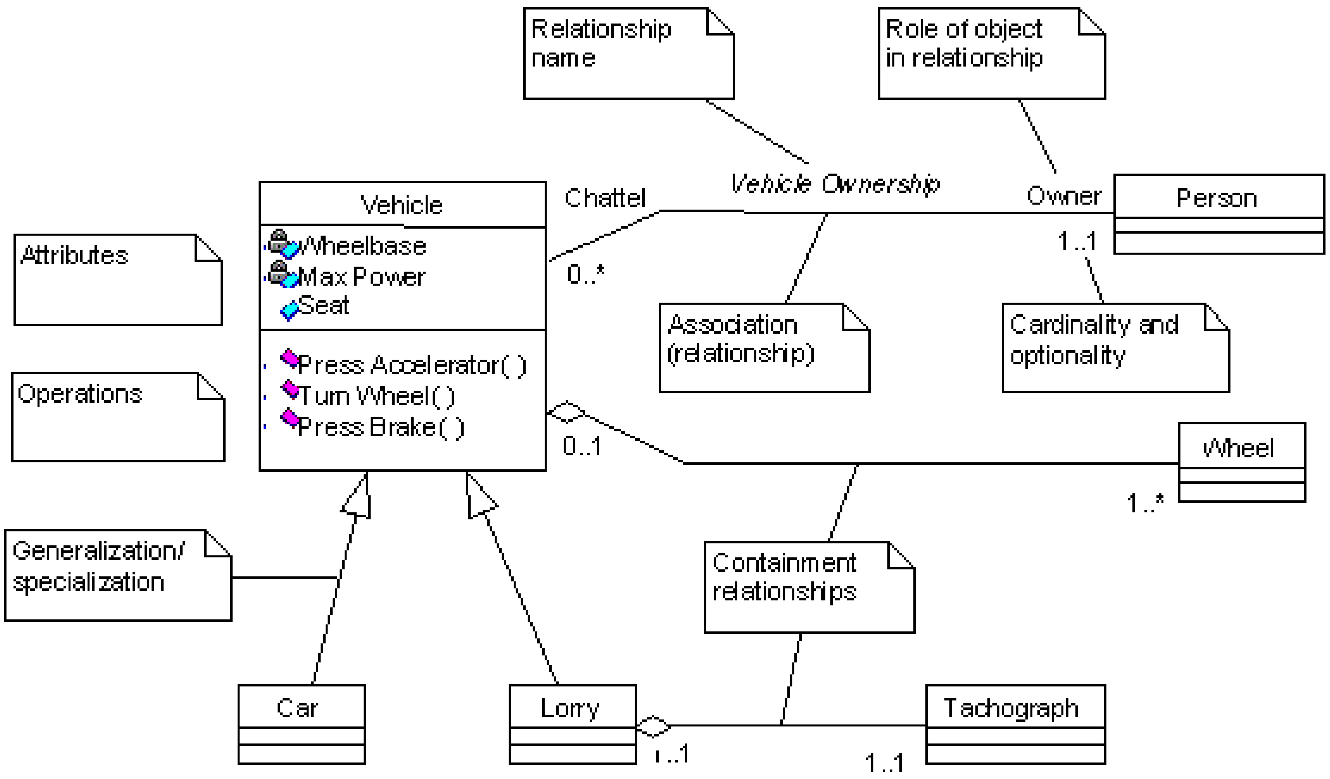
In this program, class X has one data member ‘d’ and two member functions ‘set_value()’ and ‘show_sum()’. The set_value() function is used to assign value to ‘d’. the show_sum() function uses an integer type parameter. It adds the value of parameter with the value of ‘d’ and displays the result o the screen.

Another class Y is defined after the class x. the class Y has an object of class x that is the C++ Composition relationship between classes x and y. this class has its own member function print_result().

In the main() function, an object ‘b’ of class y is created. The member function set_value() of object ‘a’ that is the sub-object of object ‘b’ is called by using tw dot operators. One dot operator is used to access the member of the object ‘b’ that is object ‘a’, and second is used to access the member function set_value() of sub-object ‘a’ and ‘d’ is assigned a value 20.

In the same way, the show_sum() member function is called by using two dot operators. The value 100 is also passed as a parameter. The member function print_result of object ‘b’ of class Y is also called for execution. In the body of this function, the show_sum() function of object ‘a’ of class X is called for execution by passing value 5.


Generalization
Inheritance
Composition
Aggregation
Dependencies
<< >>
Properties
1
*
Multiplicity

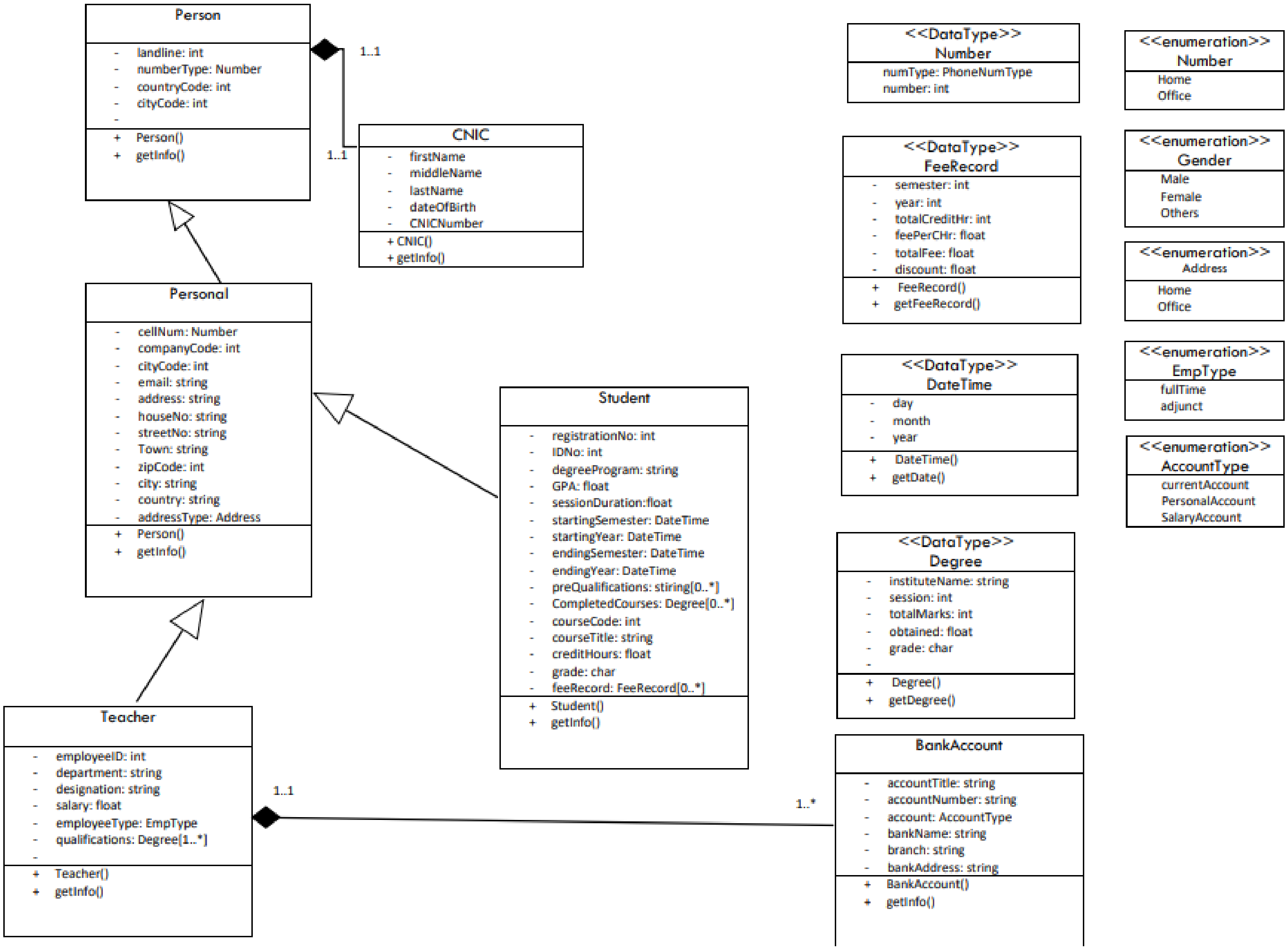

Relationship name
Role of object in relationship
Vehicle
Wheelbase
Max Power
Seat
Press Accelerator( )
Turn Wheel( )
Press Brake( )
Chattel
Vehicle Ownership
Owner
Person
0..*
1..1
Attributes
Operations
Association (relationship)
Cardinality and optionality
0..1
Wheel
1..*
Generalization/ specialization
Containment relationships
Car
Lorry
Tachograph
1..1
1..1

Lab Tasks

Q1. Write a C++ program to show the concept of Aggregation (with appropriate constructors and destructors).

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

class Person{
    protected:
        string name;
        int age;
};

class Teacher: public Person{
    private:
        string subj;
    public:
        Teacher(string n, int a, string s): Person(n,a,s) { }
};


int main(){

    return 0;
}
```

Q2. See the detail of following objects and model different classes where necessary. Make UML diagram for all classes. Identify and implement all kinds of relationships, i.e. Inheritance, Composition, Aggregation, etc. Try to call base class's constructor(s) in child class, if required.



LAB NO. 15

FILE HANDLING IN C++

Lab Objectives

Following are the lab objectives:

1. File handling in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

File Handling in C++

Files are used to store data in a storage device permanently. File handling provides a mechanism to store the output of a program in a file and to perform various operations on it.

A stream is an abstraction that represents a device on which operations of input and output are performed. A stream can be represented as a source or destination of characters of indefinite length depending on its usage.

In C++ we have a set of file handling methods. These include ifstream, ofstream, and fstream. These classes are derived from fstrembase and from the corresponding iostream class. These classes, designed to manage the disk files, are declared in fstream and therefore we must include fstream and therefore we must include this file in any program that uses files.

In C++, files are mainly dealt by using three classes fstream, ifstream, ofstream.

- ofstream: This Stream class signifies the output file stream and is applied to create files for writing information to files
- ifstream: This Stream class signifies the input file stream and is applied for reading information from files
- fstream: This Stream class can be used for both read and write from/to files.

All the above three classes are derived from fstreambase and from the corresponding iostream class and they are designed specifically to manage disk files. C++ provides us with the following operations in File Handling:

- Creating a file: open()
- Reading data: read()
- Writing new data: write()
- Closing a file: close()

Opening a File

Generally, the first operation performed on an object of one of these classes is to associate it to a real file. This procedure is known to open a file.

We can open a file using any one of the following methods:

1. First is bypassing the file name in constructor at the time of object creation.
2. Second is using the open() function.

Syntax

void open(const char *filename, ios::openmode mode);

Here, the first argument specifies the name and location of the file to be opened and the second argument of the open() member function defines the mode in which the file should be opened.

Sr.No	Mode Flag & Description
1	**ios::app** Append mode. All output to that file to be appended to the end.
2	**ios::ate** Open a file for output and move the read/write control to the end of the file.
3	**ios::in** Open a file for reading.

4	**ios::out** Open a file for writing.
5	**ios::trunc** If the file already exists, its contents will be truncated before opening the file.

Syntax

```
fstream new_file;
new_file.open(“newfile.txt”, ios::out);
```

In the above example, new_file is an object of type fstream, as we know fstream is a class so we need to create an object of this class to use its member functions. So we create new_file object and call open() function. Here we use out mode that allows us to open the file to write in it.

Default Open Modes :

- ifstream ios::in
- ofstream ios::out
- fstream ios::in | ios::out

We can combine the different modes using or symbol OR (|).

Syntax

```
ofstream new_file;

new_file.open(“new_file.txt”, ios::out | ios::app );
```

Here, input mode and append mode are combined which represents the file is opened for writing and appending the outputs at the end.

Example 1:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
        fstream my_file;
        my_file.open("my_file", ios::out);
        if (!my_file) {
                cout << "File not created!";
        }
        else {
                cout << "File created successfully!";
                my_file.close();
        }
        return 0;
}
```

Example 2:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
        fstream my_file;
        my_file.open("my_file.txt", ios::out);
        if (!my_file) {
                cout << "File not created!";
        }
        else {
                cout << "File created successfully!";
                my_file << "Lahore Garrison University";
                my_file.close();
        }
        return 0;
}
```

Example 3:

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
        fstream my_file;
        my_file.open("my_file.txt", ios::in);
        if (!my_file) {
                cout << "No such file";
        }
        else {
                char ch;

                while (1) {
                        my_file >> ch;
                        if (my_file.eof())
                                break;

                        cout << ch;
                }

        }
        my_file.close();
        return 0;
}
```

Lab Tasks

Q1. Write a program C++ to read the file with name “Question1.txt” and display the following text as output on console (output format should be same as given below).

A.o.A Students.

My name is Waqar Ali!
I am 28 year old…

Code:

```
#include <iostream>
#include <fstream>

using namespace std;

int main(){
    ifstream myFile("resources/Question1.txt");
    string str;

    if(!myFile){
        cout << "Not Sucessfull!" << endl;
    }
    else{
        while(getline(myFile,str)){
            cout << str << endl << endl;
        }
    }
    myFile.close();
    return 0;
}
```

Q.2 Write a C++ program that inputs up to 10 integer values from data file named “Question2.text” and displays them on the screen. If there are not 10 numbers in file, the message “The file is finished” should be displayed after last number.

```
#include <iostream>
#include <fstream>

using namespace std;

int main(){
    int num, counter=1;

    ifstream myFile("resources/Question2.txt");

    if(!myFile){
        cout << "Unable to open." << endl;
    }
    else{
        while(myFile >> num){
            cout << counter << ". " << num << endl;
            counter++;
        }
        cout << "The file is finished." << endl;

    }
    myFile.close();
    return 0;
}
```

Home Activity

Q3. Write a program that counts the number of blank and number of words in a text file named Question3.txt.

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(){
    ifstream myFile("resources/Question3.txt");
    char character;
    int words=1, blanks=0;

    while(myFile){
        myFile.get(character);
        if(character==' '){
            words++;
            blanks++;
        }
    }

    cout << "Words: " << words << endl;
    cout << "Blanks: " << blanks << endl;

    myFile.close();
    return 0;
}
```

Q4. Write a program that copies the contents of one file to another file as a string.

```
#include <iostream>
#include <fstream>
#include <string>

using namespace std;

int main(){
    ifstream inFile("resources/Question4.txt");
    string strLine, strFullText;

    while(getline(inFile, strLine)){
        strFullText += strLine;
    }
    inFile.close();

    ofstream outFile("resources/Question4Copy.txt");
    outFile << strFullText;

    outFile.close();
    return 0;
}
```

Q5. Write a C++ Program to Count Words Lines and Total Size using File Handling.

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

int main(){
    ifstream myFile("resources/Question5.txt");

    int words=1, lines=1;
    int size=2;                // It is 2 because I have noticed that the
                               // file size is only 2 more bytes then the
                               // actual characters in that file.
    char character, prevChar;

    while(myFile.get(character)){
        // Counting all characters as 1 byte.
        size++;

        // It is word, if there is a space as space seperate words
        // or if the previous character is new line and next character
        // is not new line.
        if(character==' ' || prevChar=='\n' && character!='\n')
            words++;
        else if(character=='\n')
            lines++;

        prevChar = character;
    }

    myFile.close();

    cout << "Size: " << size << " bytes" << endl;
    cout << "Words: " << words << endl;
    cout << "Lines: " << lines << endl;
    return 0;
}
```

LAB NO. 16

FRIEND FUNCTION AND FRIEND CLASSES

C++

Lab Objectives

Following are the lab objectives:

1. Friend function and friend classes in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

C++ friend Function and friend Classes

Data hiding is a fundamental concept of object-oriented programming. It restricts the access of private members from outside of the class.

Similarly, protected members can only be accessed by derived classes and are inaccessible from outside. For example,

```
  class MyClass {

     private:

         int member1;
}
  int main() {

     MyClass obj;

     // Error! Cannot access private members from here.

     obj.member1 = 5;

}
```

However, there is a feature in C++ called friend functions that break this rule and allow us to access member functions from outside the class.

Friend Function in C++

A friend function can access the private and protected data of a class. We declare a friend function using the friend keyword inside the body of the class.

```
class className {

     ... .. ...

     friend returnType functionName(arguments);
     ... .. ...
}
```

Example 1: Working of friend Function

```
// C++ program to demonstrate the working of friend function

#include <iostream>

using namespace std;

class Distance {

     private:

         int meter;
```

```
        // friend function
        friend int addFive(Distance);
    public:
        Distance() : meter(0) {}
};
// friend function definition
int addFive(Distance d) {
    //accessing private members from the friend function
    d.meter += 5;
    return d.meter;
}
  int main() {
    Distance D;
    cout << "Distance: " << addFive(D);
    return 0;
}
```

Output Distance:

```
5
```

Here, addFive() is a friend function that can access both private and public data members. Though this example gives us an idea about the concept of a friend function, it doesn't show any meaningful use.

A more meaningful use would be operating on objects of two different classes. That's when the friend function can be very helpful.

Example 2: Add Members of Two Different Classes

```
// Add members of two different classes using friend functions
#include <iostream>
using namespace std;
// forward declaration
class ClassB;
```

```
  class ClassA {

      public:

          // constructor to initialize numA to 12

          ClassA() : numA(12) {}

      private:
          int numA;
           // friend function declaration

           friend int add(ClassA, ClassB);

};
  class ClassB {

      public:

          // constructor to initialize numB to 1

          ClassB() : numB(1) {}

      private:
          int numB;
          // friend function declaration

          friend int add(ClassA, ClassB);

};

// access members of both classes
  int add(ClassA objectA, ClassB objectB) {

      return (objectA.numA + objectB.numB);

}
  int main() {

      ClassA objectA;

      ClassB objectB;

      cout << "Sum: " << add(objectA, objectB);

      return 0;

}
```

Output

```
Sum: 13
```

In this program, ClassA and ClassB have declared add() as a friend function. Thus, this function can access private data of both classes.
One thing to notice here is the friend function inside ClassA is using the ClassB. However, we haven't defined ClassB at this point.

```
// inside classA

friend int add(ClassA, ClassB);
```

For this to work, we need a forward declaration of ClassB in our program.

```
// forward declaration

class ClassB;
```

Friend Class in C++

We can also use a friend Class in C++ using the friend keyword. For example,

```
class ClassB;

class ClassA {
   // ClassB is a friend class of ClassA

   friend class ClassB;

   ... .. ...
}


class ClassB {
   ... .. ...

}
```

When a class is declared a friend class, all the member functions of the friend class become friend functions.
Since ClassB is a friend class, we can access all members of ClassA from inside ClassB.

However, we cannot access members of ClassB from inside ClassA. It is because friend relation in C++ is only granted, not taken.

Example 3: C++ friend Class

```
// C++ program to demonstrate the working of friend class

#include <iostream>
```

```
using namespace std;
// forward declaration

class ClassB;

  class ClassA {

     private:

        int numA;
        // friend class declaration

        friend class ClassB;

     public:
        // constructor to initialize numA to 12

        ClassA() : numA(12) {}

};
  class ClassB {

     private:

        int numB;

     public:

        // constructor to initialize numB to 1

        ClassB() : numB(1) {}

     // member function to add numA

     // from ClassA and numB from ClassB

     int add() {

        ClassA objectA;

        return objectA.numA + numB;

     }

};

  int main() {

     ClassB objectB;

     cout << "Sum: " << objectB.add();

     return 0;

}
```

Output

Sum: 13

Here, ClassB is a friend class of ClassA. So, ClassB has access to the members of classA.

In ClassB, we have created a function add() that returns the sum of numA and numB.

Since ClassB is a friend class, we can create objects of ClassA inside of ClassB.

Lab Tasks

Q1. Write a // C++ program to demonstrate the working of friend function (with appropriate constructors, destructors, member functions and friend functions).

```
#include <iostream>
using namespace std;
class Complex{
  private:
    int numA;
    int numB;
  public:
    Complex() {
      numA = 0;
      numB = 0;
    }
    // friend function
    friend Complex addCompplex(Complex, Complex);
    void set(int n1, int n2){
      numA = n1;
      numB = n2;
    }
    void display(){
      cout << numA << " + " << numB << "i" << endl;
    }
};

Complex addCompplex(Complex c1, Complex c2){
  Complex result;
  result.set((c1.numA + c2.numA),(c1.numB + c2.numB));

  return result;
}

int main(){
  Complex c1,c2;

  c1.set(2,3);
  c1.display();

  c2.set(4,6);
  c2.display();
  cout << "---------" << endl;

  addCompplex(c1,c2).display();

  return 0;
}
```

Q2. Write a class Person that has attributes of id, name and address (declare as private). It has a constructor to initialize, a member function to input and a member function to display data members. Declare class Student as friend of person class. It has additional attributes of rollnumber and marks. It also has member function to input and display its data members (demonstrate the working of friend class student).

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

class Student;

class Person{
  private:
    int id;
    string name;
    string address;

    friend class Student;

    Person(): id(0), name(""), address("") { }
    void input();
    void output();
};
class Student{
  private:
    int rollNumber;
    int marks;
    Person p;
  public:
    Student(): rollNumber(0), marks(0) { }

    void input();
    void output();
};

int main(){
  Student s;
  s.input();
  s.output();
  return 0;
}
```

```
// Person Class
void Person::input(){
  cout << "ID: "; cin >> id;
  cout << "Name: "; cin >> name;
  cout << "Address: "; cin >> address;
}
void Person::output(){
  cout << "ID: " << id << endl;
  cout << "Name: " << name << endl;
  cout << "Address: " << address << endl;

}
// Student Class
void Student::input(){
  p.input();
  cout << "Roll Number: "; cin >> rollNumber;
  cout << "Marks: "; cin >> marks;
}
void Student::output(){
  p.output();
  cout << "Roll Number: " << rollNumber << endl;
  cout << "Marks: " << marks << endl;
}
```

LAB NO. 17

OPERATING OVERLOADING

Lab Objectives

Following are the lab objectives:

1. Operator Overloading in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

C++ Operator Overloading

In C++, we can change the way operators work for user-defined types like objects and structures. This is known as operator overloading. For example,

Suppose we have created three objects c1, c2 and result from a class named Complex that represents complex numbers.
Since operator overloading allows us to change how operators work, we can redefine how the + operator works and use it to add the complex numbers of c1 and c2 by writing the following code:

```
result = c1 + c2;
```

instead of something like

```
result = c1.addNumbers(c2);
```

Note: We cannot use operator overloading for fundamental data types like int, float, char and so on.

1. Syntax for C++ Operator Overloading

To overload an operator, we use a special operator function. We define the function inside the class or structure whose objects/variables we want the overloaded operator to work with.

```
class className {

    ... .. ...
    public
      returnType operator symbol (arguments) {

          ... .. ...
      }
    ... .. ...

};
```

Here,

- returnType is the return type of the function.
- operator is a keyword.
- symbol is the operator we want to overload. Like: +, <, -, ++, etc.
- arguments is the arguments passed to the function.

Operator Overloading in Unary Operators

Unary operators operate on only one operand. The increment operator ++ and decrement operator -- are examples of unary operators.

Example1: ++ Operator (Unary Operator) Overloading

```
// Overload ++ when used as prefix

#include <iostream>

using namespace std; class

Count {

    private:
     int value;

    public:

     // Constructor to initialize count to 5

     Count() : value(5) {}

     // Overload ++ when used as prefix

     void operator ++ () {

        ++value;
     }

     void display() {
        cout << "Count: " << value << endl;
     }

};
  int main() {

     Count count1;

     // Call the "void operator ++ ()" function
     ++count1;

     count1.display();

     return 0;

}
```

Output

```
Count: 6
```

Here, when we use ++count1;, the void operator ++ () is called. This increases the value attribute for the object count1 by 1.

Note: When we overload operators, we can use it to work in any way we like. For example, we could have used ++ to increase value by 100.
However, this makes our code confusing and difficult to understand. It's our job as a programmer to use operator overloading properly and in a consistent and intuitive way.

The above example works only when ++ is used as a prefix. To make ++ work as a postfix we use this syntax.

```
void operator ++ (int) {
      // code

}
```

Notice the int inside the parentheses. It's the syntax used for using unary operators as postfix; it's not a function parameter.

Example 2: ++ Operator (Unary Operator) Overloading

```
// Overload ++ when used as prefix and postfix

#include <iostream>

using namespace std; class

Count {

     private:
      int value;

     public:

      // Constructor to initialize count to 5

      Count() : value(5) {}

      // Overload ++ when used as prefix

      void operator ++ () {

          ++value;
      }
      // Overload ++ when used as postfix

      void operator ++ (int) {

          value++;
      }


      void display() {
          cout << "Count: " << value << endl;
      }

};
  int main() {

      Count count1;

      // Call the "void operator ++ (int)" function

      count1++;

      count1.display();

      // Call the "void operator ++ ()" function
      ++count1;

      count1.display();

      return 0;

}
```

Output

```
Count: 6

Count: 7
```

The Example 2 works when ++ is used as both prefix and postfix. However, it doesn't work if we try to do something like this:

```
Count count1, result;
// Error
result = ++count1;
```

This is because the return type of our operator function is void. We can solve this problem by making Count as the return type of the operator function.

```
// return Count when ++ used as prefix


Count operator ++ () {
    // code
}


// return Count when ++ used as postfix
```

```
Count operator ++ (int) {
    // code
```

Example 3: Return Value from Operator Function (++ Operator)

```
#include   <iostream>
using  namespace  std;
class Count {
    private:
     int value;
    public:
     // Constructor to initialize count to 5
     Count() : value(5) {}
     // Overload ++ when used as prefix
     Count operator ++ () {
        Count temp;
        // Here, value is the value attribute of the calling object
        temp.value = ++value;
        return temp;
     }
     // Overload ++ when used as postfix
     Count operator ++ (int) {
        Count temp;
        // Here, value is the value attribute of the calling object
        temp.value = value++;
        return temp;
     }
     void display() {
        cout << "Count: " << value << endl;
     }
};
```

```
int main() {
      Count count1, result;
      // Call the "Count operator ++ ()" function

      result = ++count1;

      result.display();
      // Call the "Count operator ++ (int)" function

      result = count1++;

      result.display();

      return 0;

}
```

Output

```
Count: 6

Count: 6
```

Here, we have used the following code for prefix operator overloading:

```
// Overload ++ when used as prefix

Count operator ++ () {

      Count temp;
      // Here, value is the value attribute of the calling object

      temp.value = ++value;

      return temp;
}
```

The code for the postfix operator overloading is also similar. Notice that we have created an object temp and returned its value to the operator function.
Also, notice the code

```
temp.value = ++value;
```

The variable value belongs to the count1 object in main() because count1 is calling the function, while temp.value belongs to the temp object.

2. Operator Overloading in Binary Operators

Binary operators work on two operands. For example,

```
result = num + 9;
```

Here, + is a binary operator that works on the operands num and 9.

When we overload the binary operator for user-defined types by using the code: obj3

= obj1 + obj2;

The operator function is called using the obj1 object and obj2 is passed as an argument to the function.

Example 4: C++ Binary Operator Overloading

```
// C++ program to overload the binary operator +

// This program adds two complex numbers

#include <iostream>

using namespace std; class

Complex {

    private:
     float real;

    float imag;

    public:

     // Constructor to initialize real and imag to 0

     Complex() : real(0), imag(0) {}

     void input() {
        cout << "Enter real and imaginary parts respectively: ";

        cin >> real;

        cin >> imag;
     }

     // Overload the + operator

     Complex operator + (const Complex& obj) {

        Complex temp;

        temp.real = real + obj.real;

        temp.imag = imag + obj.imag;

        return temp;

     }
     void output() {
        if (imag < 0)
          cout << "Output Complex number: " << real << imag << "i";
        else
          cout << "Output Complex number: " << real << "+" << imag << "i";
     }
};

int main() {
     Complex complex1, complex2, result;

     cout << "Enter first complex number:\n";

     complex1.input();

     cout << "Enter second complex number:\n";

     complex2.input();

    // complex1 calls the operator function
    // complex2 is passed as an argument to the function

     result = complex1 + complex2;

     result.output();

     return 0;

}
```

Output

```
Enter first complex number:

Enter real and imaginary parts respectively: 9 5 Enter

second complex number:

Enter real and imaginary parts respectively: 7 6

Output Complex number: 16+11i
```

In this program, the operator function is:

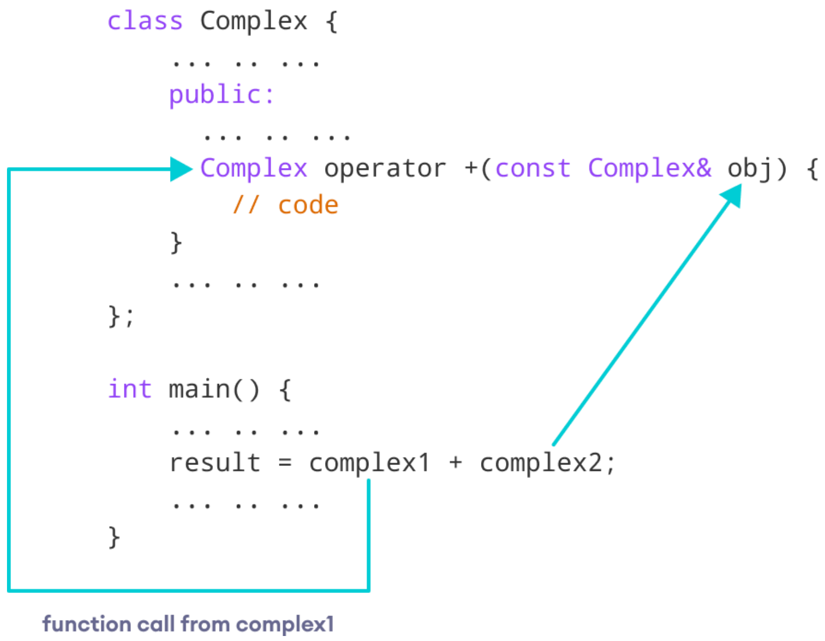
```
Complex operator + (const Complex& obj) {

    // code

}
```

Instead of this, we also could have written this function like:

```
Complex operator + (Complex obj) {
    // code
}
```

However,

- using & makes our code efficient by referencing the complex2 object instead of making a duplicate object inside the operator function.
- using const is considered a good practice because it prevents the operator function from modifying complex2.



```
class Complex {
    ... .. ...
    public:
      ... .. ...
      Complex operator +(const Complex& obj) {
        // code
    }
    ... .. ...
};

int main() {
    ... .. ...
    result = complex1 + complex2;
    ... .. ...
}
```

Overloading binary operators in C++

3. Things to Remember in C++ Operator Overloading

1. Two operators = and & are already overloaded by default in C++. For example, to copy objects of the same class, we can directly use the = operator. We do not need to create an operator function.
2. Operator overloading cannot change the precedence and associativity of operators. However, if we want to change the order of evaluation, parentheses should be used.
3. There are 4 operators that cannot be overloaded in C++. They are:
   a. :: (scope resolution)
   b. . (member selection)

c. .* (member selection through pointer to function)

d. ?: (ternary operator)

Lab Tasks

Q1. Write a class Time that has three data member hour, minutes and seconds. The class has

- The following member functions
- A constructor to initialize the time.
- Show function to show the time.
- Overload ++ operator to increase the time by 1 minute.
- Overload --- operator to decrease the time by 1 minutes.

```
#include <iostream>
#include <cstdlib>
#include <windows.h>

using namespace std;

class Time{
  private:
    int hours;
    int minutes;
    int seconds;

  public:
     Time(){
       hours = 0;
       minutes = 0;
       seconds = 0;
      }
     Time(int hr, int min, int sec){
       hours = hr;
       minutes = min;
       seconds = sec;
      }

     void operator++(){
         minutes++;
     }

     void operator --(){
         minutes--;
     }
     void put(int hr, int min, int sec){
       hours = hr;
       minutes = min;
       seconds = sec;
     }
     void show(){
       system("cls");
       cout << hours << ":" << minutes << ":" << seconds << endl;
     }
};
int main(){

  Time t1;
  t1.put(3,15,21);
  for(int i=0; i<10;++i, ++t1){
    Sleep(90);
    t1.show();
  }
  Sleep(600);
  for(int i=0; i<2;++i, --t1){
    Sleep(90);
    t1.show();
  }
  return 0;
}
```

Q2. Write a class array that contains an array of integer as data member. The class contains the following data member functions:

- A constructor or that initializes the array elements to -1.
- Input function to input the values in the array.
- Show function to display the values of the array.
- Overload == operator to compare the values of two objects. The overloaded function return 1 if values of both objects are same and return 0 otherwise.

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

class Array{
  public:
    int array[7];
    const int arrSize = 7;

    Array(){
      for(int index=0; index<arrSize; index++){
        array[index] = -1;
      }
    }
    void input(){
      for(int index=0; index<arrSize; index++){
        cout << index+1 << ": ";
        cin >> array[index];
      }
    }
    void print(){
      cout << "[";
      for(int index=0; index<arrSize; index++){
```

```
    cout << array[index] << ",";
   }
   cout << "\b]";
  }
  bool operator ==(Array &a){
   for(int i=0; i<arrSize; i++){
    if(array[i]!=a.array[i]){
     return 0;
    }
   }
   return 1;
  }
};
int main(){

 Array a1,a2;
 a1.input();
 a2.input();

 a1.print();
 cout << " is " << ((a1==a2)?"":"not") << " equal to ";
 a2.print();
 return 0;
}
```

LAB NO. 18

FUNCTION OVERRIDING, DIAMOND PROBLEM AND VIRTUAL CLASSES C++

Lab Objectives

Following are the lab objectives:

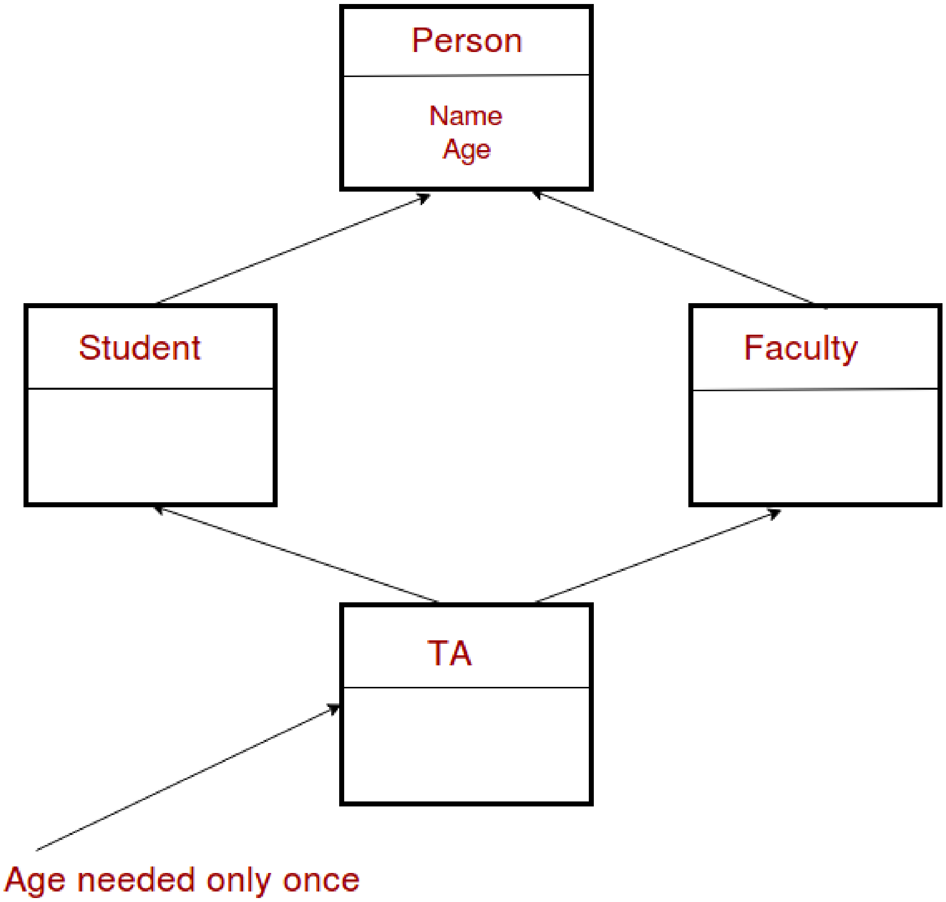
1. Friend function and friend classes in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

The diamond problem

The diamond problem occurs when two superclasses of a class have a common base class. For example, in the following diagram, the TA class gets two copies of all attributes of Person class, this causes ambiguities.



For example, consider the following program.

```
         #include<iostream>
         using namespace std;
         class Person {
         // Data members of person
         public:
Person(int x) { cout << "Person::Person(int ) called" << endl; }
         };

         class Faculty : public Person {
         // data members of Faculty
         public:
Faculty(int x):Person(x) { cout<<"Faculty::Faculty(int )
called"<< endl;
}
```

```
    };

    class Student : public Person {
    // data members of Student
    public:
        Student(int x):Person(x) {
            cout<<"Student::Student(int ) called"<< endl;
}
    };

    class TA : public Faculty, public Student {
    public:
        TA(int x):Student(x), Faculty(x) {
            cout<<"TA::TA(int ) called"<< endl;
}
    };
    int main() {
TA ta1(30);
    }
```

Output

```
Person::Person(int ) called

Faculty::Faculty(int ) called

Person::Person(int ) called

Student::Student(int ) called

TA::TA(int ) called
```

In the above program, constructor of ‘Person’ is called two times. Destructor of ‘Person’ will also be called two times when object ‘ta1’ is destructed. So object ‘ta1’ has two copies of all members of ‘Person’, this causes ambiguities. The solution to this problem is ‘virtual’ keyword. We make the classes ‘Faculty’ and ‘Student’ as virtual base classes to avoid two copies of ‘Person’ in ‘TA’ class. For example, consider the following program.

```
#include<iostream>
using namespace std;
class Person {
public:
    Person(int x) { cout << "Person::Person(int ) called" << endl; }
    Person()         { cout << "Person::Person() called" << endl; }
};

class Faculty : virtual public Person {
public:
    Faculty(int x):Person(x) {
    cout<<"Faculty::Faculty(int ) called"<< endl;
```

```
        }
    };

    class Student : virtual public Person {
    public:
Student(int x):Person(x) { cout<<"Student::Student(int )
    called"<< endl;
        }
    };

    class TA : public Faculty, public Student {
    public:
TA(int x):Student(x), Faculty(x) { cout<<"TA::TA(int )
    called"<< endl;
        }
    };

    int main() {
        TA ta1(30);
    }
```

Output

```
Person::Person() called

Faculty::Faculty(int ) called

Student::Student(int ) called

TA::TA(int ) called
```

In the above program, constructor of ‘Person’ is called once. One important thing to note in the above output is, the default constructor of ‘Person’ is called. When we use ‘virtual’ keyword, the default constructor of grandparent class is called by default even if the parent classes explicitly call parameterized constructor.

How to call the parameterized constructor of the ‘Person’ class? The constructor has to be called in ‘TA’ class. For example, see the following program.

```
#include<iostream>
using namespace std;
class Person {
public:
    Person(int x) { cout << "Person::Person(int ) called" << endl; }
    Person()      { cout << "Person::Person() called" << endl; }
};

class Faculty : virtual public Person {
public:
        Faculty(int x):Person(x) {
        cout<<"Faculty::Faculty(int ) called"<< endl;
        }
    };

    class Student : virtual public Person {
    public:
Student(int x):Person(x) { cout<<"Student::Student(int )
   called"<< endl;
        }
    };

    class TA : public Faculty, public Student {
    public:
TA(int x):Student(x), Faculty(x), Person(x) { cout<<"TA::TA(int
   ) called"<< endl;
        }
    };

    int main() {
        TA ta1(30);
    }
```

Output

```
Person::Person(int ) called
Faculty::Faculty(int ) called
Student::Student(int ) called
TA::TA(int ) called
```

In general, it is not allowed to call the grandparent's constructor directly, it has to be called through parent class. It is allowed only when 'virtual' keyword is used.

C++ Function Overriding

Suppose, the same function is defined in both the derived class and the based class. Now if we call this function using the object of the derived class, the function of the derived class is executed. This is known as function overriding in C++. The function in derived class overrides the function in base class.

Example 1: C++ 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;
}
```

Access Overridden Function in C++

To access the overridden function of the base class, we use the scope resolution operator ::.
We can also access the overridden function by using a pointer of the base class to point to an object of the derived class and then calling the function from that pointer.

Example 2: C++ Access Overridden Function to the Base Class

```
// C++ program to access overridden function
// in main() using the scope resolution operator ::

#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;
}
```

Example 3: C++ Call Overridden Function Using Pointer

```
#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;

    // pointer of Base type that points to
    derived1 Base* ptr = &derived1;

    // call function of Base class using
    ptr ptr->print();

    return 0;
}
```

Output

```
Base Function
```

Lab Tasks

Q1. Write a C++ program to demonstrate the working of override function (with appropriate constructors and destructors).

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

class A{
  public:
    A(){
      cout << "A class constructor..." << endl;
    }
    void display(){
      cout << "A class display..." << endl;
    }
};
class B: public A{
  public:
    B(){
      cout << "B class constructor..." << endl;
    }
    void display(){
      cout << "B class display..." << endl;
    }
};
int main() {
   B b;
   // B class display
   b.display();
   // A class display
   b.A::display();
   return 0;
}
```

Q.2 Write a class Person that has attributes of *id*, *name* and *address*. It has a constructor to initialize, a member function to input and a member function to display data members. Create another class Student that inherits Person class. It has additional attributes of *rollnumber* and *marks*. Override the member function of parent class in child class (input and display).

```
#include <iostream>

using namespace std;

class Person{
  protected:
    int id;
    string name;
    string address;
  public:
    // Constructors
    Person(): id(0), name(""), address("") { }
    Person(int i, string n, string a): id(i), name(n), address(a) { }

    // Input
    void input(){
      cout << "==============================" << endl;
      cout << "             Input            " << endl;
      cout << "==============================" << endl;
      cout << "Enter ID: "; cin >> id;
      cin.clear();
      cin.ignore(123,'\n');
      cout << "Enter Name: "; cin >> name;
      cin.clear();
      cin.ignore(123,'\n');
      cout << "Enter Address: "; getline(cin, address);
      cin.clear();
      cin.ignore(123,'\n');
    }
    // Display
    void display(){
      cout << "==============================" << endl;
      cout << "            Display           " << endl;
      cout << "==============================" << endl;
      cout << "ID: " << id << endl;
      cout << "Name: " << name << endl;
      cout << "Address: " << address << endl;
    }
};
class Student: virtual public Person{
  private:
    int rollNumber;
    float marks;
  public:
    // Constructors
    Student(): rollNumber(0), marks(0) { }
    Student(int i, string n, string a, int r, float m):
        Person(i,n,a), rollNumber(r), marks(m) { }

    // Input
    void input(){
      Person::input();
      cout << "Enter Roll Number: "; cin >> rollNumber;
      cin.clear();
      cin.ignore(123,'\n');
      cout << "Enter Marks: "; cin >> marks;
      cin.clear();
      cin.ignore(123,'\n');
    }
    // Display
    void display(){
      Person::display();
      cout << "Roll Number: " << rollNumber << endl;
      cout << "Marks: " << marks << endl;
    }
};
int main(){
  // Demonstration of function overriding
  Student s1;

  // functions of Student class will be called.
  s1.input();
  s1.display();
  return 0;
```

```
}
```

Q3. Predict the output of following programs.

```
// Program 1
#include<iostream>

using namespace std;
class A
{
  int x;
  public:
    void setX(int i) {x = i;}
    void print() { cout << x; }
};
class B: public A
{
  public:
  B() { setX(10); }
};
class C: public A
{
  public:
  C() { setX(20); }
};
class D: public B, public C { };

int main()
{
  D d;
  d.B::print();
  return 0;
}
```

```
// Program 2
#include<iostream>
using namespace std;
class A
{
  int x;
  public:
  A(int i) { x = i; }
  void print() { cout << x; }
};
class B:  public A
{
  public:
  B():A(20) { }
};
```

```
class C: public A
{
  public:
  C():A(30) { }
};
class D: virtual public B, virtual public C { };
int main()
{
  B *b;
  D d;
  b=&d;
  b->print();
return 0;
}
```

LAB NO. 19

PURE VIRTUAL AND VIRTUAL FUNCTION

Lab Objectives

Following are the lab objectives:

1. Pure Virtual and Virtual Functions in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

C++ virtual function

- A C++ virtual function is a member function in the base class that you redefine in a derived class. It is declared using the virtual keyword.
- It is used to tell the compiler to perform dynamic linkage or late binding on the function.
- There is a necessity to use the single pointer to refer to all the objects of the different classes. So, we create the pointer to the base class that refers to all the derived objects. But, when base class pointer contains the address of the derived class object, always executes the base class function. This issue can only be resolved by using the 'virtual' function.
- A 'virtual' is a keyword preceding the normal declaration of a function.
- When the function is made virtual, C++ determines which function is to be invoked at the runtime based on the type of the object pointed by the base class pointer.

Late binding or Dynamic linkage

In late binding function call is resolved during runtime. Therefore compiler determines the type of object at runtime, and then binds the function call.

Rules of Virtual Function

- Virtual functions must be members of some class.
- Virtual functions cannot be static members.
- They are accessed through object pointers.
- They can be a friend of another class.
- A virtual function must be defined in the base class, even though it is not used.
- The prototypes of a virtual function of the base class and all the derived classes must be identical. If the two functions with the same name but different prototypes, C++ will consider them as the overloaded functions.
- We cannot have a virtual constructor, but we can have a virtual destructor
- Consider the situation when we don't use the virtual keyword.

Example 1 #include

```
<iostream> using

namespace std; class A

{

    int x=5;

     public:

     void display()
```

```
        {
           std::cout << "Value of x is : " << x<<std::endl;

        }
};

class B: public A
{

        int y = 10;

        public:

        void display()
        {

           std::cout << "Value of y is : " <<y<< std::endl;
        }

};

int main()
{

        A *a;

        B b;

        a = &b;
       a->display();

        return 0;

}
```

Output:

```
Value of x is: 5
```

In the above example, *a 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.

C++ virtual function Example

Let's see the simple example of C++ virtual function used to invoke the derived class in a program.

Example 2

```
#include

<iostream> Class A

{
     public:
     virtual void display()
     {

      cout << "Base class is invoked"<<endl;

     }
};
class B:public A
{

     public:
     void display()
     {

      cout << "Derived Class is invoked"<<endl;
     }

};

int main()
{

     A* a;    //pointer of base class

     B b;     //object of derived class

     a = &b;

     a->display(); //Late Binding occurs
}
```

Output:

```
Derived Class is invoked
```

Example 3 (Accessing Private Method of Derived Class)

```
#include      <iostream>

using   namespace   std;

class A

{

      public:
      virtual void display()

      {
       cout << "Base class is invoked"<<endl;
      }

};
class B:public A
{

      private:
      void display()

      {
       cout << "Derived Class is invoked"<<endl;
      }
};

int main()
{
      A a;

      B b;
      A *a1; //pointer of base class
```

```
     //object of derived class

   a1 = &a;

   a1->display();
   a1=&b;//Late Binding occurs

   a1->display();

}
```

Lab Tasks

Q1. Write a class Person that has attributes of id, name and address. It has a constructor to initialize, a member function to input and a member function to display data members. Create another class Student that inherits Person class. It has additional attributes of rollnumber and marks. It also has member function to input and display its data members. (using virtual function)

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

class Person{
  protected:
    int id;
    string name;
    string address;
  public:
    // Constructors
    Person(): id(0), name(""), address("") { }
    Person(int i, string n, string a): id(i), name(n), address(a) { }

    // Input
    virtual void input(){
      cout << "==============================" << endl;
      cout << "             Input            " << endl;
      cout << "==============================" << endl;
      cout << "Enter ID: "; cin >> id;
      cin.clear();
      cin.ignore(123,'\n');
      cout << "Enter Name: "; cin >> name;
      cin.clear();
      cin.ignore(123,'\n');
      cout << "Enter Address: "; getline(cin, address);
      cin.clear();
      cin.ignore(123,'\n');
    }
    // Display
    virtual void display(){
      cout << "==============================" << endl;
      cout << "            Display           " << endl;
      cout << "==============================" << endl;
      cout << "ID: " << id << endl;
      cout << "Name: " << name << endl;
      cout << "Address: " << address << endl;
    }
};
class Student: virtual public Person{
  private:
    int rollNumber;
    float marks;
  public:
    // Constructors
    Student(): rollNumber(0), marks(0) { }
    Student(int i, string n, string a, int r, float m):
        Person(i,n,a), rollNumber(r), marks(m) { }

    // Input
    void input(){
      Person::input();
      cout << "Enter Roll Number: "; cin >> rollNumber;
      cin.clear();
      cin.ignore(123,'\n');
      cout << "Enter Marks: "; cin >> marks;
      cin.clear();
      cin.ignore(123,'\n');
    }
    // Display
    void display(){
      Person::display();
      cout << "Roll Number: " << rollNumber << endl;
      cout << "Marks: " << marks << endl;
    }
};
int main(){
  Person *p1 = new Student;

  p1->input();
  p1->display();

  return 0;
}
```

Q2. Define a class Shape having an attribute Area and a pure virtual function Calculate_Area. Also include following in this class.

- A constructor that initializes Area to zero.
- A method Display_Area() that display value of member variable “Area”.
- A virtual method Print_Area() that display value of member variable “Area”.

Now derive two classes from Shape; Circle having attribute radius, Square having attribute Length and Rectangle having attributes Length and Breadth. Include following in each class.

- A constructor that takes values of member variables as argument.
- A method Display_Area() that overrides Display_Area() method of Shape class.
- A method Print_Area() that overrides Print_Area() method of Shape class.
- A method Calculate_Area() that calculates the area as follows:

  Area of Circle= PI* Radius^2

  Area of Square=Length^2

  Area of Rectangle=Length*Breadth

Make a driver program to test above classes.

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

#define PI 3.14

class Shape{
  protected:
    float area;
  public:
    Shape(): area(0.0) { }

    virtual void calcutateArea() { }
    void displayArea(){
      cout << "Area: " << area << endl;
    }
    virtual void printArea(){
      cout << "Area: " << area << endl;
    }
};
/********************
        Circle
*********************/
class Circle: public Shape{
  private:
    float radius;
  public:
    Circle(): Shape(), radius(0.0) { }
    Circle(float r): Shape(), radius(r) { }

    void calcutateArea(){
      area = PI * radius*radius;
    }
    void displayArea(){
      cout << "Circle Area: " << area << endl;
    }
    void printArea(){
      cout << "Circle Area: " << area << endl;
    }
};
/********************
        Square
********************/
class Square: public Shape{
  private:
    float length;
  public:
    Square(): Shape(), length(0.0) { }
    Square(float l): Shape(), length(l) { }

    void calcutateArea(){
      area = length*length;
    }
    void displayArea(){
      cout << "Square Area: " << area << endl;
    }
    void printArea(){
      cout << "Square Area: " << area << endl;
    }
};
/********************
      Rectangle
********************/
class Rectangle: public Shape{
  private:
    float length;
    float breadth;
  public:
    Rectangle(): Shape(), length(0.0), breadth(0.0) { }
    Rectangle(float l, float b): Shape(), length(l), breadth(b) { }

    void calcutateArea(){
      area = length * breadth;
    }
    void displayArea(){
      cout << "Rectangle Area: " << area << endl;
    }
    void printArea(){
      cout << "Rectangle Area: " << area << endl;
    }
};
int main(){
  // Initializing pointer of shape with a child class type object.
  Shape *s1 = new Circle(4.3);
  Shape *s2 = new Square(54);
  Shape *s3 = new Rectangle(23,13);

  // Calculating Area
  s1->calcutateArea();
  s2->calcutateArea();
  s3->calcutateArea();

  // Printing Area
  s1->printArea();
  s2->printArea();
  s3->printArea();

  return 0;
}
```

LAB NO. 20

ABSTRACT CLASS AND PURE VIRTUAL FUNCTION

Lab Objectives

Following are the lab objectives:

1. Abstract Class and Pure Virtual and Virtual Functions in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

C++ Pure Virtual Functions

Pure virtual functions are used

- if a function doesn't have any use in the base class
- but the function must be implemented by all its derived classes

Let's take an example,

Suppose, we have derived Triangle, Square and Circle classes from the Shape class, and we want to calculate the area of all these shapes.

In this case, we can create a pure virtual function named calculateArea() in the Shape. Since it's a pure virtual function, all derived classes Triangle, Square and Circle must include the calculateArea() function with implementation.

A pure virtual function doesn't have the function body and it must end with = 0. For example,

```
class Shape {
    public:
      // creating a pure virtual function
      virtual void calculateArea() = 0;
};
```

Note: The = 0 syntax doesn't mean we are assigning 0 to the function. It's just the way we define pure virtual functions.

Abstract Class

A class that contains a pure virtual function is known as an abstract class. In the above example, the class Shape is an abstract class.

We cannot create objects of an abstract class. However, we can derive classes from them, and use their data members and member functions (except pure virtual functions).

Some important facts:

1. A class is abstract if it has at least one pure virtual function.
2. We can have pointers and references of abstract class type.
3. If we do not override the pure virtual function in derived class, then derived class also becomes abstract class.
4. Abstract classes cannot be instantiated.

Example: C++ Abstract Class and Pure Virtual Function

```
// C++ program to calculate the area of a square and a circle
#include <iostream>
using namespace std;

// Abstract class
class Shape {
      protected:
       float dimension;
      public:
       void getDimension() {
          cin >> dimension;
       }
       // pure virtual Function
       virtual float calculateArea() = 0;
};
// Derived class
    class Square : public Shape {
      public:
       float calculateArea() {
          return dimension * dimension;
       }
};

// Derived class
    class Circle : public Shape {
      public:
       float calculateArea() {
          return 3.14 * dimension * dimension;
       }
};

    int main() {
       Square square;
       Circle circle;

       cout << "Enter the length of the square: ";
       square.getDimension();
       cout << "Area of square: " << square.calculateArea() << endl;

       cout << "\nEnter radius of the circle: ";
       circle.getDimension();
       cout << "Area of circle: " << circle.calculateArea() << endl;

       return 0;
}
```

Output

```
Enter the length of the square: 4 Area
of square: 16
Enter radius of the circle: 5
Area of circle: 78.5
```

In this program, virtual float calculateArea() = 0; inside the Shape class is a pure virtual function.

That's why we must provide the implementation of calculateArea() in both of our derived classes, or else we will get an error.

Lab Tasks

Q1. Write a program with Student as abstract class and create derive classes Engineering, Medicine and Science from base class Student. Create the objects of the derived classes and process them and access them using array of pointer of type base class Student.

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

class Student{
  protected:
    string name;
    int rollNum;
  public:
    Student(): name(" "), rollNum(0) {}
    Student(string n, int r): name(n), rollNum(r) {}

    virtual void display() = 0;
    virtual void input() = 0;
};
class Engineering: public Student{
  protected:
    string degree;
  public:
    Engineering(): Student(), degree(" ") {}
    Engineering(string n, int r, string d): Student(n, r), degree(d) {}

    void display(){
      cout << "---------------------------" << endl;
      cout << "        Engineering        " << endl;
      cout << "---------------------------" << endl;
      cout << "Name: " << name << endl;
      cout << "Roll Number: " << rollNum << endl;
      cout << "Degree: " << degree << endl;
    }
    void input(){
      cout << "---------------------------" << endl;
      cout << "        Engineering        " << endl;
      cout << "---------------------------" << endl;
      cout << "Name: "; cin >> name;
      cout << "Roll Number: "; cin >> rollNum;
      cout << "Degree: "; cin >> degree;
    }
};
class Medicine: public Student{
  protected:
    string degree;
  public:
    Medicine(): Student(), degree(" ") {}
    Medicine(string n, int r, string d): Student(n, r), degree(d) {}
    void display(){
      cout << "---------------------------" << endl;
      cout << "         Medicine          " << endl;
      cout << "---------------------------" << endl;
      cout << "Name: " << name << endl;
      cout << "Roll Number: " << rollNum << endl;
      cout << "Degree: " << degree << endl;
    }
    void input(){
      cout << "---------------------------" << endl;
      cout << "         Medicine          " << endl;
      cout << "---------------------------" << endl;
      cout << "Name: "; cin >> name;
      cout << "Roll Number: "; cin >> rollNum;
      cout << "Degree: "; cin >> degree;
    }
};
class Science: public Student{
  protected:
    string degree;
  public:
    Science(): Student(), degree(" ") {}
    Science(string n, int r, string d): Student(n, r), degree(d) {}

    void display(){
      cout << "---------------------------" << endl;
      cout << "         Science           " << endl;
      cout << "---------------------------" << endl;
      cout << "Name: " << name << endl;
      cout << "Roll Number: " << rollNum << endl;
      cout << "Degree: " << degree << endl;
    }
    void input(){
      cout << "---------------------------" << endl;
      cout << "         Science            " << endl;
      cout << "---------------------------" << endl;
      cout << "Name: "; cin >> name;
      cout << "Roll Number: "; cin >> rollNum;
      cout << "Degree: "; cin >> degree;
    }
};
int main(){
  Engineering e1;
  Medicine m1;
  Science s1;

  Student* stdArr[3] = {&e1, &m1, &s1};

  for(int i=0; i<3; i++)
```

```
    stdArr[i]->input();

  for(int i=0; i<3; i++)
    stdArr[i]->display();
  return 0;
}
```

Q2. Define a class Shape having an attribute Area and a pure virtual function Calculate_Area. Also include following in this class.

- A constructor that initializes Area to zero.
- A method Display_Area() that display value of member variable “Area”.
- A virtual method Print_Area() that display value of member variable “Area”.

Now derive two classes from Shape; Circle having attribute radius, Square having attribute Length and Rectangle having attributes Length and Breadth. Include following in each class.

- A constructor that takes values of member variables as argument.
- A method Display_Area() that overrides Display_Area() method of Shape class.
- A method Print_Area() that overrides Print_Area() method of Shape class.
- A method Calculate_Area() that calculates the area as follows:

  Area of Circle= PI* Radius^2

  Area of Square=Length^2

  Area of Rectangle=Length*Breadth

Make a driver program to test above classes.

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

#define PI 3.14

class Shape{
  protected:
    float area;
  public:
    Shape(): area(0.0) { }

    virtual void calcutateArea() = 0;
    void displayArea(){
      cout << "Area: " << area << endl;
    }
    virtual void printArea(){
```

```
   cout << "Area: " << area << endl;
  }
};
/*********************
    Circle
*********************/
class Circle: public Shape{
 private:
  float radius;
 public:
  Circle(): Shape(), radius(0.0) { }
  Circle(float r): Shape(), radius(r) { }

  void calcutateArea(){
   area = PI * radius*radius;
  }
  void displayArea(){
   cout << "Circle Area: " << area << endl;
  }
  void printArea(){
   cout << "Circle Area: " << area << endl;
  }
};
/*********************
    Square
*********************/
class Square: public Shape{
 private:
  float length;
 public:
  Square(): Shape(), length(0.0) { }
  Square(float l): Shape(), length(l) { }

  void calcutateArea(){
   area = length*length;
  }
  void displayArea(){
   cout << "Square Area: " << area << endl;
  }
  void printArea(){
   cout << "Square Area: " << area << endl;
  }
};
/*********************
   Rectangle
*********************/
class Rectangle: public Shape{
 private:
  float length;
```

```
    float breadth;
  public:
    Rectangle(): Shape(), length(0.0), breadth(0.0) { }
    Rectangle(float l, float b): Shape(), length(l), breadth(b) { }

    void calcutateArea(){
      area = length * breadth;
    }
    void displayArea(){
      cout << "Rectangle Area: " << area << endl;
    }
    void printArea(){
      cout << "Rectangle Area: " << area << endl;
    }
};



int main(){
  // Initializing pointer of shape with a child class type object.
  Shape *s1 = new Circle(4.3);
  Shape *s2 = new Square(54);
  Shape *s3 = new Rectangle(23,13);

  // Calculating Area
  s1->calcutateArea();
  s2->calcutateArea();
  s3->calcutateArea();

  // Printing Area
  s1->printArea();
  s2->printArea();
  s3->printArea();

  return 0;
}
```

LAB NO. 21

EXCEPTION HANDLING IN C++

Lab Objectives

Following are the lab objectives:

1. Exception handling in C++

Instructions

- This is individual Lab work/task.
- Complete this lab work within lab timing.
- Discussion with peers is not allowed.
- Copy paste from Internet will give you **negative marks**.
- Lab work is divided into small tasks, complete all tasks sequentially.

Exception Handling in C++

Errors can be broadly categorized into two types. We will discuss them one by one.

1. Compile Time Errors
2. Run Time Errors

Compile Time Errors – Errors caught during compiled time is called Compile time errors. Compile time errors include library reference, syntax error or incorrect class import.
Run Time Errors - They are also known as exceptions. An exception caught during run time creates serious issues.

Errors hinder normal execution of program. Exception handling is the process of handling errors and exceptions in such a way that they do not hinder normal execution of the system. For example, User divides a number by zero, this will compile successfully but an exception or run time error will occur due to which our applications will be crashed. In order to avoid this we'll introduce exception handling technics in our code.

In C++, Error handling is done using three keywords:

- **try**
- **catch**
- **throw**

Syntax:

```
try

{
    //code
    throw parameter;

}

catch(exceptionname ex)
{

    //code to handle exception
}
```

Try block

The code which can throw any exception is kept inside(or enclosed in) atry block. Then, when the code will lead to any error, that error/exception will get caught inside the catch block.

Catch block

Catch block is intended to catch the error and handle the exception condition. We can have multiple catch blocks to handle different types of exception and perform different actions when the exceptions occur. For example, we can display descriptive messages to explain why any particular exception occurred.

Throw statement

It is used to throw exceptions to exception handler i.e. it is used to communicate information about error. A throw expression accepts one parameter and that parameter is passed to handler.
throw statement is used when we explicitly want an exception to occur, then we can use throw statement to throw or generate that exception.

Understanding Need of Exception Handling

Let's take a simple example to understand the usage of try, catch and throw.

Below program compiles successfully but the program fails at runtime, leading to an exception.

```
#include <iostream>#include<conio.h>

using namespace std; int

main()

{
    int a=10,b=0,c;

    c=a/b;

    return 0;

}
```

The above program will not run, and will show runtime error on screen, because we are trying to divide a number with 0, which is not possible.
How to handle this situation? We can handle such situations using exception handling and can inform the user that you cannot divide a number by zero, by displaying a message.

Using try, catch and throw Statement

Now we will update the above program and include exception handling in it.

```
#include <iostream>

using namespace std; int

main()

{
          int a, b;

          cout<<"Enter two integer values: ";

          cin>>a>>b;
```

```
    try
    {

        if(b == 0)
        {
            throw b;

        }
        else
        {

            cout<<(a/b);
        }
    }


    catch(int)
    {
        cout<<"Second value cannot be zero";

    }
    return 0;
}
```

Output

```
Division by zero not possible
```

In the code above, we are checking the divisor, if it is zero, we are throwing an exception message, then the catch block catches that exception and prints the message.

Doing so, the user will never know that our program failed at runtime, he/she will only see the message "Division by zero not possible".
This is gracefully handling the exception condition which is why exception handling is used.

Using Multiple catch blocks

Below program contains multiple catch blocks to handle different types of exception in different way.

```
#include <iostream>

#include<conio.h> using

namespace std;
```

```
int main()
{
    int x[3] = {-1,2};
    for(int i=0; i<2; i++)
    {
      int ex = x[i];
      try
      {
        if (ex > 0)
          // throwing numeric value as exception
          throw ex;
        else
          // throwing a character as exception
          throw 'ex';
      }
      catch (int ex) // to catch numeric exceptions
      {
        cout << "Integer exception\n";
      }
      catch (char ex) // to catch character/string exceptions
      {
        cout << "Character exception\n";
      }
    }
}
```

Output

```
Integer exception
Character exception
```

The above program is self-explanatory, if the value of integer in the array x is less than 0, we are throwing a numeric value as exception and if the value is greater than 0, then we are throwing a character value as exception. And we have two different catch blocks to catch those exceptions.

Generalized catch block in C++

Below program contains a generalized catch block to catch any uncaught errors/exceptions.
catch(...) block takes care of all type of exceptions.

```
#include <iostream>

#include<conio.h> using

namespace std;


int main()
{
    int x[3] = {-1,2};

    for(int i=0; i<2; i++)
    {
      int ex=x[i];

      try

      {
        if (ex > 0)

          throw ex;

        else
          throw 'ex';

      }

      // generalised catch block

      catch (...)

      {
        cout << "Special exception\n";
      }

    }
return 0;
```

}

Output

Special

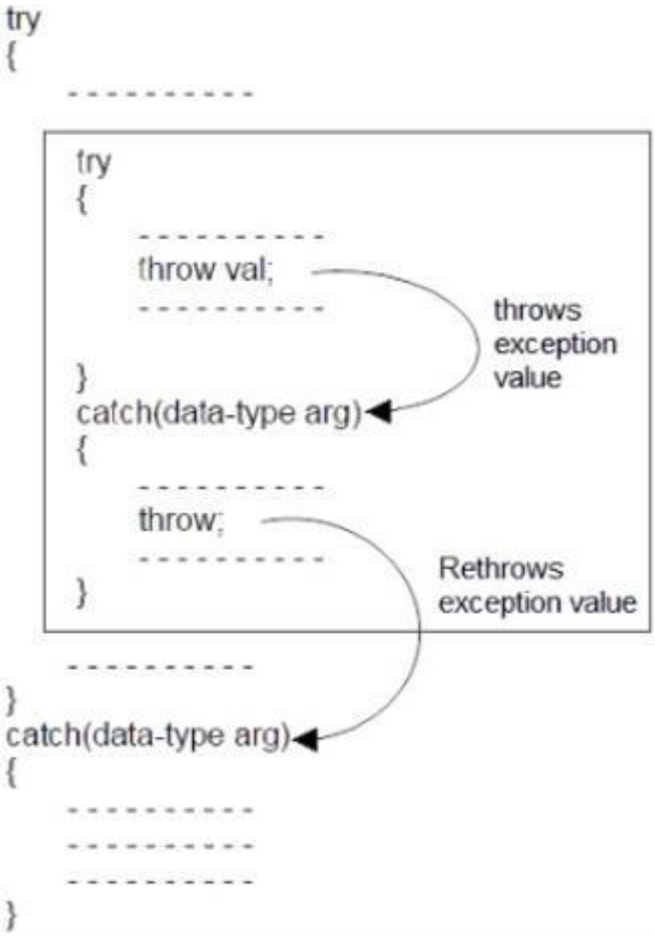
exception

Special

exception

In the case above, both the exceptions are being catched by a single catch block. We can even have separate catch blocks to handle integer and character exception along with th generalised catch block.

Syntax of Re-throwing Exception



```
try
{
    ----------
    try
    {
        ----------
        throw val;
        ----------
    }
    catch(data-type arg)
    {
        ----------
        throw;
        ----------
    }
    ----------
}
catch(data-type arg)
{
    ----------
    ----------
    ----------
}
```

Lab Tasks

Q1. Practice the above mentioned examples

Q2. Create a class which only works for absolute numbers, if it encounters any negative occurrence, then it throw an exception to its handler and display errors.

```
#include <iostream>

class Absolute{
  private:
    int number;
  public:
    Absolute(int n){
      try{
        if(n > 0)
          number = n;
        else
          throw n;
      }
      catch(int){
        std::cout << "You can't use a negative number in this class." <<
std::endl;
      }
    }
};
int main(){
  Absolute i(11);
}
```

Q.3 Write a program that add two integers. If the user enters a negative number, throw and handle an appropriate exception and prompt the user enter the positive numbers.

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

class Addition{
  private:
    int result, numb1, numb2;
  public:
    Addition(): numb1(0), numb2(0), result(0) {}
    int add(){
      bool error = 1;
      while(error){
        try{
          cout << "Enter Number 1: "; cin >> numb1;
          cout << "Enter Number 2: "; cin >> numb2;

          if(numb1 >= 0 && numb2 >= 0){
            result = numb1 + numb2;
            error=0;
          }
          else
```

```
            throw 1;
        }
        catch(int){
          cout << "You can't use a negative number." << endl;
          error = 1;
        }
      }
      return result;
    }

};
int main(){
  Addition a1;
  cout << a1.add();
}
```