The ICFAI University Raipur

PRACTICAL FILE

For the partial fulfilment of BCA 2nd year

Lab work

OBJECT ORIENTED PROGRAMMING

NAME OF THE STUDENT: ______________________

ROLL NO: ________________BRANCH:____________

YEAR: ____________ SEMESTER: _________________

Faculty of Information Technology

INDEX

Sr. No.	Title	Page No.

Practical -1

Aim: Write a C++ program to demonstrate conditional statements.

Theory:

if statement in C/C++
if statement is the most simple decision-making statement. It is used to decide whether a certain statement or block of statements will be executed or not i.e if a certain condition is true then a block of statement is executed otherwise not.

```
if(condition)

{

   // Statements to execute if

   // condition is true

}
```

Here, the **condition** after evaluation will be either true or false. C if statement accepts boolean values – if the value is true then it will execute the block of statements below it otherwise not. If we do not provide the curly braces '{' and '}' after if(condition) then by default if statement will consider the first immediately below statement to be inside its block.

Program:

```
// 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.
```

Practical -2

Aim: Write a C++ program to demonstrate looping statements.

Theory:

This is a repetition control structure that helps us iterate over a section of C++ code for a fixed number of times. A for loop runs provided the test expression is true. The loop terminates execution immediately the test expression becomes false. This means before the execution of the loop body in each iteration, the condition has to be evaluated. If the evaluation returns a true, the loop body is executed. If the evaluation returns a false, execution of the loop body is terminated.

Here is the syntax for the for loop:

```
for ( initialization;condition;increment ) {
  statement(s);
}
```

Here is an explanation of the above parameters:

- **Initialization:** This part is executed first and only once. Here, you declare and initialize loop control variables. The loop control variables can be more than one, and their values will change after every iteration. However, their values must be evaluated before an iteration runs.
- **Condition:** This part is executed next. For the loop body to be executed, this condition must be true. If the condition is false, execution will jump to statements immediately after the loop body. If the condition is false on the first evaluation, the loop body will never be executed.
- **Increment:** Once the loop body has been executed, control jumps to the increment. You can leave out this part and use a semicolon instead.
- Again, the condition is evaluated. If it’s true, the loop body is executed, and this continues. The loop terminates immediately the condition becomes false.

Program:

```
// The factorial Program in C++ using loop.
#include <iostream>
using namespace std;
int main()
{
  int i,fact=1,number;
  cout<<"Enter any Number: ";
 cin>>number;
  for(i=1;i<=number;i++){
      fact=fact*i;
  }
  cout<<"Factorial of " <<number<<" is: "<<fact<<endl;
  return 0;
}
```

Output:

```
Enter any Number: 5
 Factorial of 5 is: 120
```

Practical -3

Aim: Write a C++ program to demonstrate Class and Object.

Theory:

Class

A class is a blueprint for the object.

We can think of a class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows, etc. Based on these descriptions we build the house. House is the object.

Create a Class

A class is defined in C++ using keyword class followed by the name of the class.

The body of the class is defined inside the curly brackets and terminated by a semicolon at the end.

```
class className {

   // some data

   // some functions

};
```

Objects

When a class is defined, only the specification for the object is defined; no memory or storage is allocated.

To use the data and access functions defined in the class, we need to create objects.

Syntax to Define Object in C++

```
className objectVariableName;
```

Program:

```
// Program to illustrate the working of
// objects and class in C++ Programming
#include <iostream>
using namespace std;
// create a class
class Room {
   public:
    double length;
    double breadth;
    double height;

    double calculateArea() {
        return length * breadth;
    }

    double calculateVolume() {
        return length * breadth * height;
    }
};

int main() {

    // create object of Room class
    Room room1;

    // assign values to data members
    room1.length = 42.5;
    room1.breadth = 30.8;
    room1.height = 19.2;

    // calculate and display the area and volume of the room
    cout << "Area of Room =  " << room1.calculateArea() << endl;
    cout << "Volume of Room =  " << room1.calculateVolume() << endl;

    return 0;
}
```

Output:

```
Area of Room =  1309
Volume of Room =  25132.8
```

Practical -4

Aim: Write a C++ program to demonstrate constructor.

Theory:

Constructor in C++ is a special method that is invoked automatically at the time of object creation. It is used to initialize the data members of new objects generally. The constructor in C++ has the same name as the class or structure. Constructor is invoked at the time of object creation. It constructs the values i.e. provides data for the object which is why it is known as constructors.
Constructor does not have a return value, hence they do not have a return type.

The prototype of Constructors is as follows:

<class-name> (list-of-parameters);

The syntax for defining the constructor within the class:

<class-name> (list-of-parameters) { // constructor definition }

The syntax for defining the constructor outside the class:

<class-name>: :<class-name> (list-of-parameters){ // constructor definition}

A constructor is different from normal functions in following ways:

- Constructor has same name as the class itself
- Default Constructors don't have input argument however, Copy and Parameterized Constructors have input arguments
- Constructors don't have return type
- A constructor is automatically called when an object is created.
- It must be placed in public section of class.
- If we do not specify a constructor, C++ compiler generates a default constructor for object (expects no parameters and has an empty body).

Program:

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

// declare a class
class Wall {
  private:
    double length;
    double height;
  public:
    // initialize variables with parameterized constructor
    Wall(double len, double hgt) {
```

```
    length = len;
    height = hgt;
  }

  // copy constructor with a Wall object as parameter
  // copies data of the obj parameter
  Wall(Wall &obj) {
    length = obj.length;
    height = obj.height;
  }

  double calculateArea() {
    return length * height;
  }
};

int main() {
  // create an object of Wall class
  Wall wall1(10.5, 8.6);

  // copy contents of wall1 to wall2
  Wall wall2 = wall1;

  // print areas of wall1 and wall2
  cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
  cout << "Area of Wall 2: " << wall2.calculateArea();

  return 0;
}
```

Output:

```
Area of Wall 1: 90.3
Area of Wall 2: 90.3
```

Practical -5

Aim: Write a C++ program to demonstrate Friend function.

Theory:

Friend function

If a function is defined as a friend function in C++, then the protected and private data of a class can be accessed using the function.

By using the keyword friend compiler knows the given function is a friend function.

For accessing the data, the declaration of a friend function should be done inside the body of a class starting with the keyword friend.

Declaration of friend function in C++

```
class class_name
{
   friend data_type function_name(argument/s);            // syntax of friend function.
};
```

In the above declaration, the friend function is preceded by the keyword friend. The function can be defined anywhere in the program like a normal C++ function. The function definition does not use either the keyword **friend or scope resolution operator**.

Characteristics of a Friend function:

- The function is not in the scope of the class to which it has been declared as a friend.
- It cannot be called using the object as it is not in the scope of that class.
- It can be invoked like a normal function without using the object.
- It cannot access the member names directly and has to use an object name and dot membership operator with the member name.
- It can be declared either in the private or the public part.

Program:

```
#include <iostream>
using namespace std;
class B;          // forward declarartion.
class A
{
    int x;
    public:
    void setdata(int i)
```

```
        {
            x=i;
        }
        friend void min(A,B);          // friend function.
    };
    class B
    {
        int y;
        public:
        void setdata(int i)
        {
            y=i;
        }
        friend void min(A,B);                    // friend function
    };
    void min(A a,B b)
    {
        if(a.x<=b.y)
        std::cout << a.x << std::endl;
        else
        std::cout << b.y << std::endl;
    }
      int main()
    {
      A a;
      B b;
      a.setdata(10);
      b.setdata(20);
      min(a,b);
       return 0;
     }
```

Output:

```
10
```

Practical -6

Aim: Write a C++ program to demonstrate function overloading.

Theory:

Function Overloading

Function Overloading is defined as the process of having two or more function with the same name, but different in parameters is known as function overloading in C++. In function overloading, the function is redefined by using either different types of arguments or a different number of arguments. It is only through these differences compiler can differentiate between the functions.

The **advantage** of Function overloading is that it increases the readability of the program because you don't need to use different names for the same action.

Program:

```
// program of function overloading when number of arguments vary.
#include <iostream>
using namespace std;
class Cal {
    public:
static int add(int a,int b){
        return a + b;
    }
static int add(int a, int b, int c)
    {
        return a + b + c;
    }
};
int main(void) {
    Cal C;                              //    class object declaration.
    cout<<C.add(10, 20)<<endl;
    cout<<C.add(12, 20, 23);
   return 0;
}
```

Output:

```
30
55
```

Practical -7

Aim: Write a C++ program to demonstrate Operator overloading.

Theory:

Operators Overloading

Operator overloading is a compile-time polymorphism in which the operator is overloaded to provide the special meaning to the user-defined data type. Operator overloading is used to overload or redefines most of the operators available in C++. It is used to perform the operation on the user-defined data type. For example, C++ provides the ability to add the variables of the user-defined data type that is applied to the built-in data types.

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

Operator that cannot be overloaded are as follows:

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

Syntax of Operator Overloading

```
return_type class_name  : : operator op(argument_list)
{
    // body of the function.
}
```

Where the **return type** is the type of value returned by the function.

class_name is the name of the class.

operator op is an operator function where op is the operator being overloaded, and the operator is the keyword.

Rules for Operator Overloading

- Existing operators can only be overloaded, but the new operators cannot be overloaded.

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

Program:

```
#include <iostream>
using namespace std;
class A
{

    int x;
      public:
      A(){}
    A(int i)
    {
       x=i;
    }
    void operator+(A);
    void display();
};

void A :: operator+(A a)
{

    int m = x+a.x;
    cout<<"The result of the addition of two objects is : "<<m;


}
```

```
int main()
{
    A a1(5);
    A a2(4);
    a1+a2;
    return 0;
}
```

Output:

```
The result of the addition of two objects is : 9
```

Practical -8

Aim: Write a C++ program to demonstrate Single and Multiple Inheritance.

Theory:

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.

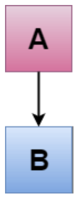
In C++, the class which inherits the members of another class is called derived class and the class whose members are inherited is called base class. The derived class is the specialized class for the base class.

Advantage of C++ Inheritance

Code reusability: Now you can reuse the members of your parent class. So, there is no need to define the member again. So less code is required in the class.

Single Inheritance

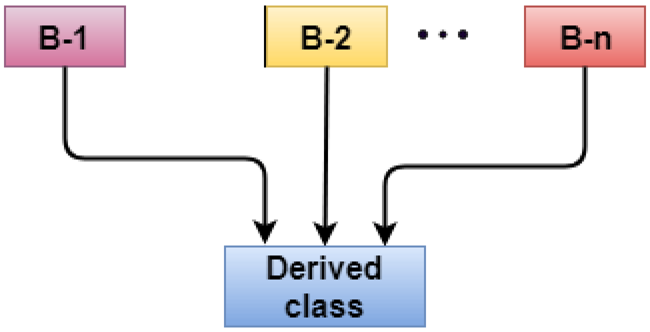
Single inheritance is defined as the inheritance in which a derived class is inherited from the only one base class.



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

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;
}
```

Program:

```
#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
```

```
#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:

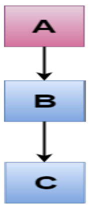
```
The value of a is : 10
The value of b is : 20
Addition of a and b is : 30
```

Practical -9

Aim: Write a C++ program to demonstrate Multilevel and Hierarchical Inheritance.

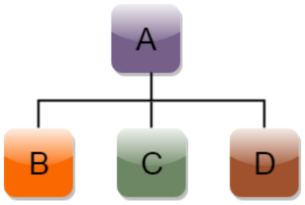
Multilevel Inheritance

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



Hierarchical Inheritance

Hierarchical inheritance is defined as the process of deriving more than one class from a base class.



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.
}
```

Program:

```
#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...
```

```
#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;
  std::cout << "Enter the length and breadth of a rectangle: " << std::endl;
  cin>>length>>breadth;
  r.get_data(length,breadth);
  int m = r.rect_area();
  std::cout << "Area of the rectangle is : " <<m<< std::endl;
  std::cout << "Enter the base and height of the triangle: " << std::endl;
  cin>>base>>height;
  t.get_data(base,height);
  float n = t.triangle_area();
  std::cout <<"Area of the triangle is : "  << n<<std::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
```

Practical -10

Aim: Write a C++ program to demonstrate Exception Handling.

Theory:

Exception Handling

Exception Handling in C++ is a process to handle runtime errors. We perform exception handling so the normal flow of the application can be maintained even after runtime errors.

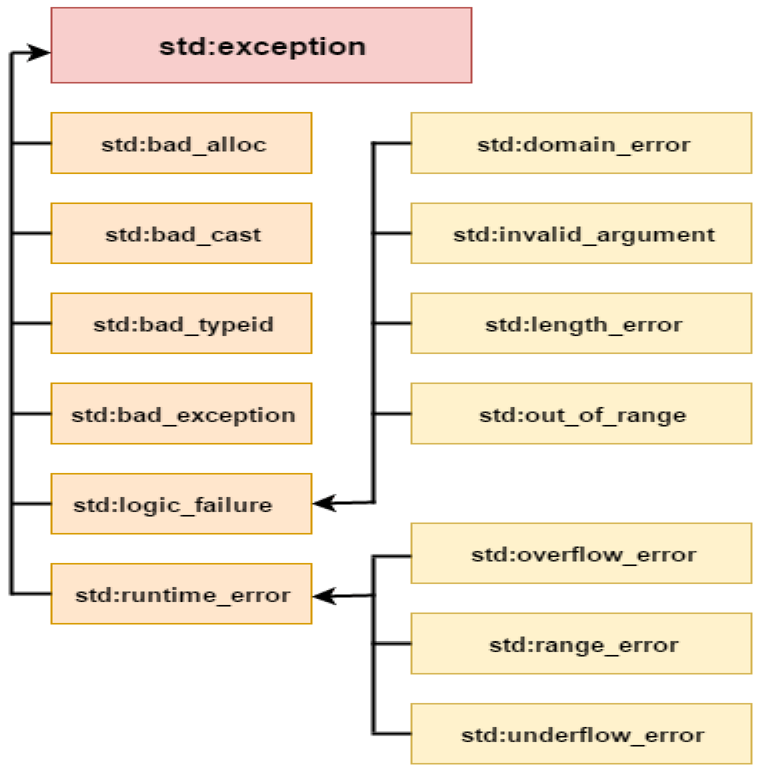
In C++, exception is an event or object which is thrown at runtime. All exceptions are derived from std::exception class. It is a runtime error which can be handled. If we don't handle the exception, it prints exception message and terminates the program.

Advantage

It maintains the normal flow of the application. In such case, rest of the code is executed even after exception.

C++ Exception Classes

In C++ standard exceptions are defined in <exception> class that we can use inside our programs. The arrangement of parent-child class hierarchy is shown below:



Program:

```
#include <iostream>
using namespace std;
float division(int x, int y) {
  return (x/y);
}
int main () {
  int i = 50;
  int j = 0;
  float k = 0;
    k = division(i, j);
    cout << k << endl;
  return 0;
}
```

Output:

```
Floating point exception (core dumped)
```