

Object Oriented Programming (CT-260)

Lab 01

Introduction to Object Oriented Programming with C++

Objectives

The objective of this lab is to familiarize students with the basic structure of C++ programs. By the end of this lab, students will be able to create, compile, execute basic input-output programs written in C++ language.

Tools Required

DevC++ IDE / Visual Studio / Visual Code

Course Coordinator – Dr. Murk Marvi

Course Instructor – Samia Masood Awan

Lab Instructor – Samia Masood Awan

Prepared By Department of Computer Science and Information Technology

NED University of Engineering and Technology

Introduction to C++

C++ is a cross-platform language that can be used to create high-performance applications. C++ was developed by Bjarne Stroustrup, as an extension to the C language. C++ gives programmers a high level of control over system resources and memory. The language was updated 3 major times in 2011, 2014, and 2017 to C++11, C++14, and C++17.

C++ is very similar to the C Language.

- For the input/output stream we use <iostream> library (in C it was <stdio>).
- For taking input and out we cout and cin (in C it was printf and scanf).
- cout uses insertion (<<) operator.
- cin uses extraction (>>) operator.

Basics of C++ Program

/* Comments can also be written starting with a slash followed by a star, and ending with a star followed by a slash. As you can see, comments written in this way can span more than one line. */ /* Programs should ALWAYS include plenty of comments! */ /* This program prints the table of entered number */

- **The #include Directive**
The #include directive causes the contents of another file to be inserted into the program. Preprocessor directives are not C++ statements and do not require semicolons at the end.
- **Using namespace std;**
The names cout and endl belong to the std namespace. They can be referenced via fully qualified name std::cout and std::endl, or simply as cout and endl with a "using namespace std;" statement.
- **return 0;**
The return value of 0 indicates normal termination; while non-zero (typically 1) indicates abnormal termination. C++ compiler will automatically insert a "return 0;" at the end of the the main () function, thus, this statement can be omitted.
- **Output using cout**
 - Cout is an object
 - Corresponds to standard output stream
 - << is called insertion or input operator
- **Input With cin**
 - Cin is an object
 - Corresponds to standard input stream
 - >> is called extraction or get from operator

Example of a Basic C++ Program:

```
#include <iostream>
using namespace std;
int main() {
    char a;
    int num;
    cout << "Enter a character and an integer: ";
    cin >> a >> num;
    cout << "Character: " << a << endl;
    cout << "Number: " << num;
    return 0;
}
```

Output:

```
Enter a character and an integer: X
20
Character: X
Number: 20
```

Data Types in C++

C++ supports the following basic data types:

boolean	1 byte	Stores true or false values
char	1 byte	Stores a single character/letter/number, or ASCII values
int	2 or 4 bytes	Stores whole numbers, without decimals
float	4 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 6-7 decimal digits
double	8 bytes	Stores fractional numbers, containing one or more decimals. Sufficient for storing 15 decimal digits

String Data Type:

Apart from these basic types, C++ also support 'string' data type to store multiple characters. i.e., a series of characters or text. Strings may only be used if an additional header file, the <string> library, is included in the source code.

```
#include<string>
```

In latest version of the C++ compilers <string> library has been included in <iostream> library, and you don't need to include it again.

String values must be enclosed in double quotations.

```
string introduction = "Welcome to OOP Lab";
cout << introduction;
```

Basic C++ String Program

```
#include <iostream>
#include <string>
using namespace std;
int main() {
    string greeting = "Hello";
    cout << greeting;
    return 0;
}
```

Comments in C++

A comment is text that is ignored by the compiler yet is beneficial to programmers. Normally, comments are intended to mark code for future reference. They are treated as white space by the compiler. C++ supports two types of comments

- **Single line comment** – The // (two slashes) characters, followed by any character sequence. This type of comment is terminated by a new line that is not immediately followed by a backslash.
- **Multiple Line Comment** – The characters /* (slash, asterisk), followed by any sequence of characters (including new lines), followed by the characters */.

Pointers

A Pointer is a variable whose content is a memory address.

- To declare a single pointer variable, you need to specify the data type, an asterisk symbol (*) and the name of the pointer variable.

```
dataType *ptrName;
```

- Following is the declaration of a Pointer variable.

```
int *ptr;
```

- DataType: Integer
- Name: ptr
- Pointer variable holds the memory address of the variable which is of same data type (integer in this case).
- To assign the memory address of any variable to the pointer variable we use **Address of Operator (&)**.

```
int intVar = 5;
```

```
ptr = &intVar;
```

- In this statement **ptr** now holds the memory address of an integer variable '**intVar**'.
- To access the value at the memory address (currently stored) in the variable we use Dereferencing Operator (*).
- Do not confuse this with the symbol used for the declaration of a pointer.

```
int intVar2 = *ptr;
```

- In this statement another integer variable '**intVar2**' is now initialized with the value at the memory address which is stored in **ptr** (that is the value of intVar).

Example Code for Pointers:

```
#include <iostream>
using namespace std;
int main( ){
    int *p;
    int x = 37;
    cout << "Line 1: x = " << x << endl; //Line 1
    p = &x; //Line 2
    //Line 3
    cout << "Line 3: *p = " << *p << ", x = " << x << endl;
    *p = 58; //Line 4
    //Line 5
    cout << "Line 5: *p = " << *p << ", x = " << x << endl;
    cout << "Line 6: Address of p = " << &p << endl; //Line 6
    cout << "Line 7: Value of p = " << p << endl; //Line 7
    cout << "Line 8: Value of the memory location " << "pointed to
    by *p = " << *p << endl; //Line 8
    cout << "Line 9: Address of x = " << &x << endl; //Line 9
    cout << "Line 10: Value of x = " << x << endl; //Line 10
    return 0;
}
```

Output:

```

Line 1: x = 37
Line 3: *p = 37, x = 37
Line 5: *p = 58, x = 58
Line 6: Address of p = 0x6ffe08
Line 7: Value of p = 0x6ffe04
Line 8: Value of the memory location pointed to by *p = 58
Line 9: Address of x = 0x6ffe04
Line 10: Value of x = 58

```

Dynamic Memory Allocation

If we need a variable amount of memory that can only be determined during runtime, then we use dynamic memory which can be implemented by using new and delete operators. For normal variables like "int a", "char str[10]", etc, memory is automatically allocated and de-allocated on stack. For dynamically allocated memory like "int *p = new int[10]", it is programmers responsibility to de-allocate memory when no longer needed.

new Operator

Variables created during the program execution are called dynamic variables. To create a dynamic variable, we use new operator.

```

new dataType;           // to allocate a single variable
new dataType [ size];   // to allocate an array of variables.

```

- The new operator allocates the memory of a designated type.
- It returns a pointer to the allocated memory.

Following is the declaration of a dynamic variable.

```

int *p = new int;
char *cArray = new char[5];

```

- Line 01: creates a single variable of integer type whose address is stored in p.
- Line 02: Creates an array of 5 characters whose initial address is stored in cArray.

Example of new Operator:

```

// Pointer initialized with NULL, Then request memory for the variable
int *p = NULL;
p = new int;

```

OR

```

// Combine declaration of pointer and their assignment
int *p = new int;

```

We can also initialize the memory using new operator:

```

pointer-variable = new data-type(value);

```

Example:

```

int *p = new int(25);
float *q = new float(75.25);

```

new operator is also used to allocate a block (an array) of memory of type data-type. Note: size(a variable) specifies the number of elements in an array.

```

pointer-variable = new data-type[size];

```

Example Code:

```
#include<iostream>
using namespace std;
int main(){
    int* intPtr;
    char* charArray;
    int Size;
    intPtr = new int; // allocating memory to single variable
    cout << "Enter an Integer Value: ";
    cin >> *intPtr;
    cout << "Enter the size of the Character Array : ";
    cin >> Size;
    charArray = new char[Size]; // allocating memory to array
    for (int i = 0; i < Size; i++)
        cin >> charArray[i];
    for (int i = 0; i < Size; i++)
        cout << charArray[i];
    return 0;
}
```

Output:

```
Enter an Integer Value: 2
Enter the size of the Character Array : 2
a b
ab
```

delete Operator

Since it is programmer's responsibility to deallocate dynamically allocated memory, programmers are provided delete operator by C++ language.

```
delete ptrVar;           //to deallocate single dynamic variable
delete[] ptrArray;       //to deallocate dynamically created array
```

delete operator is used to free the memory which is dynamically allocated using new operator.

Function

A function is a self-contained program segment that carries out a specific well-defined task. In C++, every program contains one or more functions which can be invoked from other parts of a program, if required.

When passing parameters to a function there are ways to do that.

1. Value Parameters (Pass by Value).
2. Reference Parameters (Pass by Reference).

Value Parameters (Pass by Value)

When passing value parameters in a function, the parameter is copied into the corresponding formal parameter. There is no connection between the actual and formal parameter values, this means that these parameters cannot be used to pass the result back to the calling function.

Example Code:

```
#include <iostream>
using namespace std;
void funcValueParam (int num);
int main (){
    int number = 6; //Line 1
    cout << "Line 2: Before calling the function " <<
    "funcValueParam, number = " << number << endl; //Line 2
    funcValueParam(number); //Line 3
    cout << "Line 4: After calling the function " <<
    "funcValueParam, number = " << number << endl; //Line 4
    return 0;
}
void funcValueParam (int num){
    cout << "Line 5: In the function funcValueParam, " <<
    "before changing, num = " << num << endl; //Line 5
    num = 15; //Line 6
    Value Parameters | 367
    cout << "Line 7: In the function funcValueParam, " <<
    "after changing, num = " << num << endl; //Line 7
}
```

Output:

```
Line 2: Before calling the function funcValueParam, number = 6
Line 5: In the function funcValueParam, before changing, num = 6
Line 7: In the function funcValueParam, after changing, num = 15
Line 4: After calling the function funcValueParam, number = 6
```

Reference Parameters (Pass by Reference)

When a reference parameter is passed in a function, it receives the address (memory location) of the actual parameter. Reference parameters can change the value of the actual parameter.

Reference parameters are useful in following situations.

- 1- When the value of the actual parameter needs to be changed.
- 2- When you want to return more than one value from a function.
- 3- When passing the address would save memory space and time relative to copying a large amount of data.

Example Code:

```
//This program reads a course score and prints the
//associated course grade.
#include <iostream>
using namespace std;
void getScore (int& score);
void printGrade (int score);
int main (){
    int courseScore;
    cout << "Line 1: Based on the course score, \n" << " this
    program computes the " << "course grade." << endl; //Line 1
    getScore(courseScore); //Line 2
    printGrade(courseScore); //Line 3
    return 0; }
void getScore (int& score){
    cout << "Line 4: Enter course score: "; //Line 4
    cin >> score; //Line 5
    cout << endl << "Line 6: Course score is "<< score << endl;
    //Line 6
```

```

}
void printGrade (int cScore){
    cout << "Line 7: Your grade for the course is "; //Line 7
    if (cScore >= 90) //Line 8
        cout << "A." << endl;
    else if (cScore >= 80)
        cout << "B." << endl;
    else if (cScore >= 70)
        cout << "C." << endl;
    else if (cScore >= 60)
        cout << "D." << endl;
    else
        cout << "F." << endl;
}

```

Output:

Line 1: Based on the course score, this program computes the course grade.

Line 4: Enter course score: 85

Line 6: Course score is 85

Line 7: Your grade for the course is B.

Static and Automatic Variables in C++

A variable for which memory is allocated at block entry and de-allocated at block exit is called an automatic variable. - A variable for which memory remains allocated as long as the program executes is called a static variable. Global variables are static variables by default and variables declared in a block are automatic variables. Syntax for declaring a static variable is:

```
static datatype identifier;
```

Example Code for Static Variables:

```

//Program: Static and automatic variables
#include <iostream>
using namespace std;
void test ();
int main (){
    int count;
    for (count = 1; count <= 5; count++)
        test ();
    return 0;
}
void test (){
    static int x = 0;
    int y = 10;
    x = x + 2;
    y = y + 1;
    cout << "Inside test x = " << x << " and y = " << y << endl;
}

```

Output:

Inside test x = 2 and y = 11

Inside test x = 4 and y = 11

Inside test x = 6 and y = 11

Inside test x = 8 and y = 11

Inside test x = 10 and y = 11

Exercise

1. Write a program that take input of your roll number along with the marks obtained in five subjects and display the total marks obtained and the percentage.
2. Write a program to swap three numbers entered by a user using pointers.
3. Write a program to convert temp from Fahrenheit to Celsius unit using equation $C=(F-32)/1.8$
4. Using 2-D arrays, write a program that allows the user to input two, 3x3 matrices. Write a function for adding two matrices. Write another function for multiplying the two matrices.
5. Write a program to find Surface area and volume of a sphere using functions.
6. Write a program to help a bank create its withdrawal system. Your program should allow the user to input their account type. Account types are: savings, current. Following business rules apply when withdrawing from a certain account:
 - **Savings:**
User must provide the savings account number and code 'S' (for savings). When withdrawing from a savings account, users need to pay a set 2% of the money that they withdraw. If the amount of money withdrawn is over 50,000, then a 5% tax will be deducted. The money deducted shall be from the remaining money in the account.
 - **Current:**
User must provide the current account number and code „C" (for current). When withdrawing from a current account, users need to pay a withdrawal fee of 100. If the amount of money withdrawn is over 50,000, then a 5% tax will be deducted. The money deducted shall be from the remaining money in the account.

Assume all users have the 200,000 in their accounts, and cannot withdraw more than 100,000 at a time.

Object Oriented Programming (CT-260)

Lab 02

Introduction to Object Oriented Programming with C++

Objectives

The objective of this lab is to familiarize students with object-oriented programming. By the end of this lab, students will be able to understand the concepts of classes, objects, and access modifiers.

Tools Required

DevC++ IDE / Visual Studio / Visual Code

Course Coordinator –

Course Instructor –

Lab Instructor –

Prepared By Department of Computer Science and Information Technology

NED University of Engineering and Technology

Object Oriented Programming

The fundamental idea behind object-oriented languages is to combine into a single unit both data and the functions that operate on that data. Such a unit is called an object. An object's functions, called member functions in C++, typically provide the only way to access its data.

To read a data item in an object, member function in the object is called, which in turn will access the data and return the value. The data cannot be accessed directly. The data is hidden, so it is safe from accidental alteration. Data encapsulation and data hiding are key terms in object-oriented languages. This simplifies writing, debugging, and maintaining the program.

A C++ program typically consists of a number of objects, which communicate with each other by calling one another's member functions. Member functions are also called methods in some other object-oriented (OO) languages. Also, data items are referred to as attributes or instance variables, features. The major elements of object-oriented languages in general, and C++ in particular are:

- ✓ Objects
- ✓ Classes
- ✓ Inheritance.
- ✓ Reusability
- ✓ Polymorphism and Overloading.

In C++, everything is linked to classes and objects, as well as their properties and functions. An automobile, for example, is an object in real life. The automobile has attributes like weight and color, as well as methods like drive and brake. Attributes and methods are essentially the class's variables and functions. These are commonly known as "class members."

Class

A class is a definition of objects of the same kind. In other words, a class is a blueprint, template, or prototype that defines and describes the static attributes and dynamic behaviors common to all objects of the same kind.

Instance

An instance is a realization of a particular item of a class. In other words, an instance is an instantiation of a class. All the instances of a class have similar properties, as described in the class definition. For example, you can define a class called "Student" and create three instances of the class "Student" for "Paul", "Peter" and "Pauline". The term "object" usually refers to instance. But it is often used quite loosely, which may refer to a class or an instance. A class can be visualized as a three-compartment box:

Name / Identifier
Data Members / Variables
Member Functions / Methods

Example:

Name (Identifier)	Student	Circle	SoccerPlayer	Car
Variables (Static attributes)	name gpa	radius color	name number xLocation yLocation	plateNumber xLocation yLocation speed
Methods (Dynamic behaviors)	getName() setGpa()	getRadius() getArea()	run() jump() kickBall()	move() park() accelerate()

Examples of classes

The following figure shows two instances of the class Student, identified as "paul" and "peter".

Name	<u>paul:Student</u>	<u>peter:Student</u>
Variables	name="Paul Lee" gpa=3.5	name="Peter Tan" gpa=3.9
Methods	getName() setGpa()	getName() setGpa()

Two instances - paul and peter - of the class Student

Class Definition

In C++, we use the keyword class to define a class. There are two sections in the class declaration:

- ✓ private
- ✓ public

```

class Circle                                // classname
{
private:
    double radius;                            // Data members
    string color;
public:
    double getRadius( );                      // Member functions
    double getArea( );
}

```

Circle
radius color
getRadius() getArea()

```

class SoccerPlayer                          // classname
{
private:
    int number;                               // Data members
    string name;
    int x, y;
public:
    void run();                               // Member functions
    void kickBall();
}

```

SoccerPlayer
name number xLocation yLocation
run() jump() kickBall()

Creating Instances of a Class

To create an instance of a class, you have to:

1. Declare an instance identifier (name) of a particular class.
2. Invoke a constructor to construct the instance (i.e., allocate storage for the instance and initialize the variables).

For examples, suppose that we have a class called Circle, we can create instances of Circle as follows:

Construct 3 instances of the class Circle: c1, c2, and c3	
Circle c1(1.2, "red");	// radius, color
Circle c2(3.4);	// radius, default color
Circle c3;	// default radius and color

Dot (.) Operator

To reference a member of an object (data member or member function), you must:

1. First identify the instance you are interested in, and then
2. Use the dot operator (.) to reference the member, in the form of *instanceName.memberName*.

For example, suppose that we have a class called Circle, with two data members (radius and color) and two functions (getRadius() and getArea()). We have created three instances of the class Circle, namely, c1, c2 and c3. To invoke the function getArea(), you must first identify the instance of interest, say c2, then use the dot operator, in the form of c2.getArea(), to invoke the getArea() function of instance c2.

Example:

```
//Declare and construct instances c1 and c2 of the class Circle
Circle c1(1.2, "blue");
Circle c2(3.4, "green");
//Invoke member function via dot operator
cout << c1.getArea() << endl;
cout << c2.getArea() << endl;
//Reference data members via dot operator
c1.radius = 5.5;
c2.radius = 6.6;
```

Calling getArea() without identifying the instance is meaningless, as the radius is unknown (there could be many instances of Circle - each maintaining its own radius).

Putting them together an OOP Example

Class Definition

Circle
-radius:double=1.0 -color:String="red"
+Circle() +Circle(r:double) +Circle(r:double,c:String) +getRadius():double +getColor():String +getArea():double

Instances

c1:Circle	c2:Circle	c3:Circle
-radius=2.0 -color="blue"	-radius=2.0 -color="red"	-radius=1.0 -color="red"
+getRadius() +getColor() +getArea()	+getRadius() +getColor() +getArea()	+getRadius() +getColor() +getArea()

A class called Circle is to be defined as illustrated in the class diagram. It contains two data members: radius (of type double) and color (of type String); and three member functions: getRadius(), getColor(), and getArea(). Three instances of Circles called c1, c2, and c3 shall then be constructed with their respective data members, as shown in the instance diagrams.

Public" vs. "Private" Access Control Modifiers

An access control modifier can be used to control the visibility of a data member or a member function within a class. We begin with the following two access control modifiers:

1. **public:** The member (data or function) is accessible and available to all in the system.
2. **private:** The member (data or function) is accessible and available within this class only.

For example, in the above Circle definition, the data member radius is declared private. As the result, radius is accessible inside the Circle class, but NOT outside the class.

In other words, you cannot use "c1.radius" to refer to c1's radius in main().

Try inserting the statement `"cout << c1.radius;"` in main() and observe the error message.

Encapsulation in OOP

A class encapsulates the static attributes and the dynamic behaviors into a "3-compartment box". Once a class is defined, you can seal up the "box" and put the "box" on the shelf for others to use and reuse. Anyone can pick up the "box" and use it in their application. This cannot be done in the traditional procedural-oriented language like C, as the static attributes (or variables) are scattered over the entire program and header files. You cannot "cut" out a portion of C program, plug into another program and expect the program to run without extensive changes.

Data member of a class are typically hidden from the outside world, with private access control modifier. Access to the private data members e.g., are provided via public assessor functions, getRadius() and getColor().

This follows the principle of information hiding. That is, objects communicate with each other using well-defined interfaces (public functions). Objects are not allowed to know the implementation details of

Getters and Setters

To allow other to read the value of a private data member says xxx, you shall provide a get function (or getter or accessor function) called getXxx(). A getter need not expose the data in raw format. It can process the data and limit the view of the data others will see. Getters shall not modify the data member.

To allow other classes to modify the value of a private data member says xxx, you shall provide a set function (or setter or mutator function) called setXxx(). A setter could provide data validation (such as range checking), and transform the raw data into the internal representation.

For example, in our Circle class, the data member's radius and color are declared private. That is to say, they are only available within the Circle class and not visible outside the

Circle class - including main (). You cannot access the private data members radius and color from the main () directly – via says c1.radius or c1.color. The Circle class provides two public accessor functions, namely, getRadius () and getColor (). These functions are declared public. The main () can invoke these public accessor functions to retrieve the radius and color of a Circle object, via says c1.getRadius () and c1.getColor ().

There is no way you can change the radius or color of a Circle object, after it is constructed in main (). You cannot issue statements such as c1.radius = 5.0 to change the radius of instance c1, as radius is declared as private in the Circle class and is not visible to other including main (). If the designer of the Circle class permits the change the radius and color after a Circle object is constructed, he has to provide the appropriate setter, e.g.,

```
// Setter for color
void setColor(string c) {
    color = c;
}

// Setter for radius
void setRadius(double r) {
    radius = r;
}
```

Example:

```
#include <iostream>
using namespace std;
class firstprogram {
    private: // we declare a as private to hide it from outside
        int number1;
    public:
        void set(int input1){//set() function to set the value of a
            number1 = input1;
        }
        int get() { // get() function to return the value of a
            return number1;
        }
};

// main function
int main() {
    firstprogram myInstance;
    myInstance.set(10);
    cout << myInstance.get() << endl;
    return 0;
}
```

Exercise

1. Write a program in which a class named student has member variables name, roll_no, semester and section. Create 4 objects of this class to store data of 4 different students, now display data of only those students who belong to section A.
2. You are a programmer for the ABC Bank assigned to develop a class that models the basic workings of a bank account. The class should perform the following tasks:
 - Save the account balance.
 - Save the number of transactions performed on the account.
 - Allow deposits to be made to the account.
 - Allow with draws to be taken from the account.
 - Report the current account balance at any time.
 - Report the current number of transactions at any time.

Menu

1. Display the account balance
 2. Display the number of transactions
 3. Display interest earned for this period
 4. Make a deposit
 5. Make a withdrawal
 6. Exit the program
3. Create a class called Employee that includes three pieces of information as data members—a first name (type char* (DMA)), a last name (type string) and a monthly salary (type int). Your class should have a setter function that initializes the three data members. Provide a getter function for each data member. If the monthly salary is not positive, set it to 0. Write a test program that demonstrates class Employee's capabilities. Create two Employee objects and display each object's yearly salary. Then give each Employee a 10 percent raise and display each Employee's yearly salary again. Identify and add any other related functions to achieve the said goal.
 4. Write C++ code to represent a hitting game by using OOP concept. The details are as follows: This game is being played between two teams (i.e. your team and the enemy team). The total number of players in your team is randomly generated and stored accordingly. The function generates a pair of numbers and matches each pair. If the numbers get matched, the following message is displayed: "Enemy got hit by your team!" Otherwise, the following message is displayed: "You got hit by the enemy team!" The number of hits should be equal to the number of players in your team. The program should tell the final result of your team by counting the hits of both the teams. Consider the following sample output:

```
Total No. Of Players in your team: 3
Pair of numbers:
Number1: 3
Number2: 3
Enemy got hit by your team!
Pair of numbers:
Number1: 1
Number2: 1
Enemy got hit by your team!
Pair of numbers:
Number1: 5
Number2: 1
You got hit by the enemy team!
Game Over! You won
```


Object Oriented Programming (CT-260)

Lab 03

Introduction to Constructor and Destructor

Objectives

The objective of this lab is to familiarize students with constructor and destructor. By the end of this lab, students will be able to understand the concepts of constructor, different types of constructor, and destructor.

Tools Required

DevC++ IDE / Visual Studio / Visual Code

Course Coordinator –

Course Instructor –

Lab Instructor –

Prepared By Department of Computer Science and Information Technology

NED University of Engineering and Technology

Constructor

Constructor is a member function that is automatically called when we create an object. Constructor is used to initialize the data members according to the desired value. A constructor has the same name as that of the class (It is case sensitive) and it does not have a return type, not even void.

```
class Person {
public:
    Person() { // constructor
        // code
    }
};
```

Constructor can also be used to declare run time memory (dynamic memory for the data members). By providing many definitions for constructor, you are actually implementing 'function overloading'; i.e. functions of same name but different parameters (not return type) and function definition.

Types of Constructor

There are 3 types of constructor:

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

Default Constructor

- This type of constructor does not have any arguments inside its parenthesis. It is called when creating objects without passing any arguments.

Example 1: Default Constructor

```
#include <iostream>
using namespace std;
class construct {
public:
    int a, b;
    construct() {
        a = 10;
        b = 20;
    }
};
int main() {
    construct c;
    cout << "a: " << c.a << endl
    << "b: " << c.b;
    return 1;
}
```

Example 2: Default Constructor

```
#include <iostream>
using namespace std;
class Wall { // declare a class
private:
    double length;
public:
    // default constructor to initialize variable
```

```
    Wall() {
        length = 5.5;
        cout << "Creating a wall." << endl;
        cout << "Length = " << length << endl;
    }
};
int main() {
    Wall wall1;
    return 0;
}
```

Parameterized Constructor

- This type of constructor accepts arguments inside its parenthesis.
- Is called when creating objects by passing relevant arguments.
- These arguments help initialize an object when it is created.
- To create a parameterized constructor, simply add parameters to it the way you would to any other function.
- When you define the constructor's body, use the parameters to initialize the object.

Example 1: Parameterized Constructor

```
#include <iostream>
using namespace std;
class Point {
private:
    int x, y;
public:
    Point(int x1, int y1){
        x = x1;
        y = y1;
    }
    int getX(){
        return x;
    }
    int getY(){
        return y;
    }
};
int main(){
    Point p1(10, 15);
    cout << "p1.x = " << p1.getX() << ", p1.y = " << p1.getY();
    return 0;
}
```

Example 2: Parameterized Constructor

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

// declare a class
class Wall {
private:
    double length;
    double height;
```

```

    public:
        // parameterized constructor to initialize variables
        Wall(double len, double hgt) {
            length = len;
            height = hgt;
        }
        double calculateArea() {
            return length * height;
        }
};

int main() {
    // create object and initialize data members
    Wall wall1(10.5, 8.6);
    Wall wall2(8.5, 6.3);
    cout << "Area of Wall 1: " << wall1.calculateArea() << endl;
    cout << "Area of Wall 2: " << wall2.calculateArea();
    return 0;
}

```

Copy Constructor

- A copy constructor is a member function which initializes an object using another object of the same class. The copy constructor in C++ is used to copy data of one object to another.

Example 1: Copy Constructor

```

#include<iostream>
#include<conio.h>
using namespace std;
class Example {
    int a, b;
public:
    Example(int x, int y) {
        a = x;
        b = y;
        cout << "\nIm Constructor";
    }
    Example(const Example& obj) {
        a = obj.a;
        b = obj.b;
        cout << "\nIm Copy Constructor";
    }
    void Display() {
        cout << "\nValues :" << a << "\t" << b;
    }
};

int main() {
    Example Object(10, 20);
    Example Object2(Object);
    Example Object3 = Object;
    Object.Display();
    Object2.Display();
    Object3.Display();
    return 0;
}

```

```
}
```

Example 2: Copy Constructor

```
#include <iostream>
using namespace std;
class Wall { // declare a class
private:
    double *length;
    double *height;
public:
// initialize variables with parameterized constructor
Wall() {
    length=NULL;
    height=NULL;
}
Wall(double len, double hgt) {
    length = new double;
    height = new double;
    if (length!=NULL && height !=NULL){
        *length=len;
        *height=hgt;
    }
    else
        exit(1);
}
// copy constructor - Deep Copy with a Wall object as parameter
// copies data of the obj parameter
Wall(Wall &obj) {
    length=new double;
    height= new double;
    *length = *(obj.length);
    *height = *(obj.height);
}
set_values(double len, double hei){
    *length=len;
    *height=hei;
}
double calculateArea() {
    return (*length) * (*height);
}
~Wall(){
    delete height;
    delete length;
}
};

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() << endl;
    wall2.set_values(2.3,1.5);
    cout<<"after changing values for wall 2"<<endl;
```

```
    cout <<"Area of Wall 1: " << wall1.calculateArea()<< endl;
    cout << "Area of Wall 2: " << wall2.calculateArea();
    return 0;
}
```

Destructor

A destructor is a special member function that works just opposite to constructor, unlike constructors that are used for initializing an object, destructors destroy (or delete) the object. Destructor function is automatically invoked when the objects are destroyed. It cannot be declared static or const. A destructor has the same name as the constructor but is preceded by a tilde (~).

Destructor does not have arguments. It has no return type not even void. An object of a class with a Destructor cannot become a member of the union. A destructor should be declared in the public section of the class. The programmer cannot access the address of destructor.

```
class Person {
public:
    Person(){ // constructor
        // code
    }
    ~Person(){ //destructor
        // code
    }
};
```

In a class definition, if both *constructor* and *destructor* are not provided, the compiler will automatically provide default *constructor* for you. Hence during the instantiation of the object, the data member will be initialized to any value.

Example:

```
#include <iostream>
using namespace std;
class HelloWorld{
public:
    HelloWorld(){
        cout<<"Constructor is called"<<endl;
    }
    ~HelloWorld(){
        cout<<"Destructor is called"<<endl;
    }
    void display(){
        cout<<"Hello World!"<<endl;
    }
};
int main(){
    HelloWorld obj;
    obj.display();
    return 0;
}
```

Exercise

1. Write a C++ program to copy the value of one object to another object using copy constructor. For example you can define a class for complex number and create its object for performing this task. Remember that you must allocate memory dynamically to data members.
2. In a virtual battle arena game called "Epic Clash," players control powerful characters to engage in intense battles against each other. Each character has distinct abilities and attributes, including health, attack power, and defense. Your task is to implement the Character class to encapsulate these attributes, provide getter and setter methods for them, and offer different constructors to create characters with various starting conditions.

Encapsulation: Ensure that the character's attributes (health, attackPower, and defense) are private to the Character class, accessible only through appropriate getter and setter methods, allocated memory dynamically in heap.

Constructors: Implement three constructors: A default constructor that initializes a character with standard starting values for health, attack power, and defense. A parameterized constructor that allows specifying custom values for health, attack power, and defense. A copy constructor that creates a new Character object by copying the data from an existing Character object.

3. Create a class tollbooth. The two data items are a type int to hold the total number of cars and a type double to hold the total amount of money collected. A constructor initializes both these to 0. When a car passes the toll, a member function called payingCar() increments the car total and adds 0.50 to the cash total. Another member function displays the two totals. DESIGN and IMPLEMENT this case. Make assumptions (if required) and include it in the description before designing the solution.
4. Some of the characteristics of a book are the title, author(s), publisher, ISBN, price, and year of publication. Design a class **bookType** that defines the book as an ADT.
 - Each object of the class **bookType** can hold the following information about a book: title, up to four authors, publisher, ISBN, price, and number of copies in stock. To keep track of the number of authors, add another member variable.
 - Include the member functions to perform the various operations on objects of type **bookType**.

For example, the usual operations that can be performed on the title are to show the title, set the title, and check whether a title is the same as the actual title of the book. Similarly, the typical operations that can be performed on the number of copies in stock are to show the number of copies in stock, set the number of copies in stock, update the number of copies in stock, and return the number of copies in stock. Add similar operations for the publisher, ISBN, book price, and authors. Add the appropriate constructors and a destructor (if one is needed).
 - Write the definitions of the member functions of the class **bookType**.
 - Write a program that uses the class **bookType** and tests various operations on the objects of the class **bookType**. Declare an array of 100 components of type **bookType**. Some of the operations that you should perform are to search for a book by its title, search by ISBN, and update the number of copies of a book.

Object Oriented Programming (CT-260)

Lab 04

Introduction to Use of Static and Constant Keywords for Data, Member Functions and Objects of a Class

Objectives

The objective of this lab is to familiarize students with use of constant and static keywords in Object-oriented paradigm. By the end of this lab, students will be able to understand the concepts of static and constant data, member functions and objects.

- **Tools Required**

DevC++ IDE / Visual Studio / Visual Code

Course Coordinator –

Course Instructor –

Lab Instructor –

Prepared By Department of Computer Science and Information Technology

NED University of Engineering and Technology

this POINTER

You can access class members using by default, the compiler provides each member function of a class with an implicit parameter that points to the object through which the member function is called. The implicit parameter is this pointer.

```
#include<iostream>
using namespace std;
class example{
private:
    int x;
public:
    void set(int x){
        (*this).x = x;
    }
    int get( ){
        return x;
    }
    void printAddressAndValue( ){
        cout << "The address is"<<this<<"and the value is"<<(*this).x<<endl;
    }
};
```

```
The address is 0x23fe40 and the value is 5
The address is 0x23fe30 and the value is 6
```

One copy of each member function in a class is stored no matter how many objects exist, and each instance of a class uses the same function code. When you call a member function, it knows which object to use because you use the object's name. The address of the correct object is stored in this pointer and automatically passed to the function.

Within any member function, you can explicitly use this pointer to access the object's data fields. You can use the C++ pointer-to-member operator, which looks like an arrow (->).

```
#include<iostream>
using namespace std;
class Test{
private:
    int x;
    int y;
public:
    Test(int x = 0, int y = 0){
        this->x = x;
        this->y = y;
    }
    void print( ){
        cout<<"x="<<x<<" y="<<y <<endl;
    }
};
int main( ){
    Test obj1(10, 20);
    obj1.print( );
}
```

```
x = 10 y = 20
-----
Process exited after 0.01273 seconds with return value 0
Press any key to continue . . .
```

Member Initialization List

Initializer List is used in initializing the data members of a class. The list of members to be initialized is indicated with constructor as a comma-separated list followed by a colon. Following is an example that uses the initializer list to initialize x and y of Point class.

```
#include<iostream>
using namespace std;
class Point {
private:
    int x;
    int y;
public:
    Point(int i = 0, int j = 0):x(i), y(j) { }
    int getX( ) const {return x;}
    int getY( ) const {return y;}
};

int main( ) {
    Point t1(10, 15);
    cout<<"x = "<<t1.getX( )<<" ";
    cout<<"y = "<<t1.getY( );
    return 0;
}
```

Uses

- For initialization of non-static const data members.
- For initialization of reference members.
- For initialization of member objects which do not have a default constructor.
- For initialization of base class members.
- When the constructor's parameter name is same as the data member.
- For Performance reasons

Constant Keyword

If there is a need to initialize some data members of an object when it is created and cannot be changed afterward, use const keyword with data members.

CONSTANT DATA MEMBERS

Those members of a class are made constant which needs not to be changed after its initialization. The data member needs to be initialized constant when its created.

```
#include <iostream>
using namespace std;
class Students{
private:
    string name;
```

```

    const int rollno;
    float cgpa;
public:
    Students (int rno): rollno(rno){ }
    void set(string sname, float cg){
        name = sname;
        cgpa = cg;
    }
    void print( ){
        cout<<"Name: "<<name<<" , Roll #"<<rollno<<" , CGPA, "<<cgpa<<endl;
    }
};
int main( ){
    Students s(12);
    s.set("Ahmad",3.67);
    s.print( );
}

```

CONSTANT MEMBER FUNCTIONS

- Constant member function is the function that cannot modify the data members.
- To declare a constant member function, write the const keyword after the closing parenthesis of the parameter list. If there is separate declaration and definition, then the const keyword is required in both the declaration and the definition.
- Constant member functions are used, so that accidental changes to objects can be avoided. A constant member function can be applied to a non-const object.
- keyword const can't be used for constructors and destructors because the purpose of a constructor is to initialize data members, so it must change the object. Same goes for destructors.

```

#include<iostream>
using namespace std;
class test{
private:
    int a;
public:
    int nonconstFuction(int a){
        cout<<"Non Constant Function is called"<<endl;
        a=a+10;
        return a;
    }
    int constFuction(int a) const{
        return a;
    }
};
main(){
    test t;
    cout<<"Constant Function is called"<<endl;
    cout<<t.nonconstFuction(10)<<endl;
    cout<<t.constFuction(30);
    return 0;
}

```

Output:

```
Constant Function is called
Non Constant Function is called
20
30
-----
Process exited after 0.6311 seconds with return value 0
Press any key to continue . . .
```

CONSTANT OBJECTS

As with normal variables we can also make class objects constant so that their value can't change during program execution. Constant objects can only call constant member functions. The reason is that only constant member function will make sure that it will not change value of the object. They are also called as read only objects. To declare constant object just write const keyword before object declaration.

```
#include<iostream>
using namespace std;
class test{
public:
    int a;
    test(){
        a=8;
    }
    int nonconstFuction(){
        cout<<"Non Constant Function is called "<<endl;
        //a=a+10;
        return a;
    }
    int constFuction(int a) const{
        this->a=a+10; //error
        return a;
    }
};
int main(){
    const test t;
    cout<<"Constant Function is called"<<endl;
    t.a=10; // error, can't modify const objects
    cout<<t.nonconstFuction();//error, can't call non const objects
    cout<<t.constFuction(10);
    return 0;
}
```

Static Members of a Class

Similar to **static variables** a class can have **static members**. Let us note the following about the static members of a class.

- If a function of a class is static, in the class definition it is declared using the keyword **static** in its heading.
- If a member variable of a class is static, it is declared using the keyword **static**.
- A static member, function, or variable of a class can be accessed using the class name and **the scope resolution operator '::'**.

Defining the static data member

It should be defined outside of the class following this syntax:

- `data_type class_name :: member_name =value;`
- If you are calling a static data member within a member function, member function should be declared as static (i.e. a static member function can access the static data members)

```
#include <iostream>
using namespace std;
class Demo{
private:
    static int x;
public:
    static void fun( ){
        cout<<"value of X: " << x << endl;
    }
};
//defining
int Demo::x=10;
int main( ){
    Demo X;
    X.fun( );
    return 0;
}
```

Accessing static data member without static member function:

```
#include <iostream>
using namespace std;
class Demo{
public:
    static int ABC;
};
//defining
int Demo :: ABC =10;
int main( ){
    cout<<"\nValue of ABC: "<<Demo::ABC;
    return 0;
}
```

Exercise

1. Create a class 'Employee' having two data members '**EmployeeName**'(char*) and 'EmployeeId' (int). Keep both data members private. Create three initialized objects 'Employee1', 'Employee2' and 'Employee3' of type 'Employee' in such a way that the employee name for each employee can be changed when required but the employee Id for each employee must be initialized only once and should remain same always. Use member initializer list, accessors/getters and mutators/setters for appropriate data members. The result must be displayed by calling the accessors.
2. Create a class called **DynamicArray**. The class should contain a pointer to the array, and size of the array as data members. Create the parameterized constructor which take size of the array as input and initializes all the values with 0. Create the member function "push" which takes value as parameter and push it to the end of the array. Create the member function size() which returns the size of the array.
3. Write a program in which a class named **Account** has private member variables named account_no ,account_bal ,security_code. Use a public function to initialize the variables and print all data. Keep track of number of objects using the keyword static.
4. By considering a scenario of your own choice, write a program to demonstrate the concept of constant keyword.
5. "Hotel Mercato" requires a system module that will help the hotel to calculate the rent of the customers. You are required to develop one module of the system according to the following requirements:
 - The hotel wants such a system that should have the feature to change the implementation independently of the interface. This will help when dealing with changing requirements.
 - The hotel charges each customer 1000.85/- per day. This amount is being decided by the hotel committee and cannot be changed fulfilling certain complex formalities.
 - The module then analyses the number of days. If the customer has stayed for more than a week in the hotel, he gets discount on the rent. Otherwise, he is being charged normally.
 - The discounted rent is being calculated after subtracting one day from the total number of days.
 - In the end, the module displays the following details:
 - Customer name
 - Days
 - Rent

Note that, the function used for displaying purpose must not have the ability to modify any data member.

Object Oriented Programming (CT-260)

Lab 05

Introduction to inheritance

Objectives

The objective of this lab is to familiarize students with the concept of inheritance in object oriented programming. By the end of this lab, students will be able to understand and implement various types of inheritances in C++ language.

Tools Required

DevC++ IDE / Visual Studio / Visual Code

Course Coordinator – Dr. Murk Marvi

Course Instructor – Ms. Samia Masood Awan

Lab Instructor – Ms. Samia Masood Awan

Prepared By Department of Computer Science and Information Technology

NED University of Engineering and Technology

INTRODUCTION TO INHERITANCE

Inheritance is one of the key features of Object-oriented programming in C++. It allows us to create a new class (derived class) from an existing class (base class).

Base Class:

A base class is the class from which features are to be inherited into another class.

Derived Class:

A derived class is the one which inherits features from the base class. It can have additional properties and methods that are not present in the parent class that distinguishes it and provides additional functionality.

Is – A Relationship:

Inheritance is represented by an is-a relationship which means that an object of a derived class also can be treated as an object of its base class for example, a Car is a Vehicle, so any attributes and behaviors of a Vehicle are also attributes and behaviors of a Car.

Advantages of Inheritance:

The main advantage of inheritance is code reusability. You can reuse the members of your base class inside the derived class, without having to rewrite the code.

Real World Example:

A real world example of inheritance constitutes the concept that children inherit certain features and traits from their parents. In addition, children also have their unique features and traits that distinguishes them from their parents.

Basic syntax for Inheritance:

```
class derived-class-name : access base-class-name {  
    // body of class  
};
```

MODES OF INHERITANCE BASED ON BASE CLASS ACCESS CONTROL

There are three types of inheritance with respect to base class access control:

- Public
- Private
- Protected

Public Mode:

If we derive a subclass from a public base class. Then the public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class.

Protected Mode:

If we derive a subclass from a Protected base class. Then both public members and protected members of the base class will become protected in the derived class.

Private Mode:

If we derive a subclass from a Private base class. Then both public members and protected members of the base class will become Private in the derived class.

Note: The private members in the base class cannot be directly accessed in the derived class, while protected members can be directly accessed.

Syntax for public Inheritance:

```
Class (name of the derived class) : public (name of the base class)  
Class Car : public Vehicle
```


Base Class Access Control for Public, Private and Protected:

Access Modifier of Base Class Member	Mode of Inheritance		
	Public Inheritance	Private Inheritance	Protected Inheritance
Public	Public in derived class	Private in derived class	Protected in derived class
Private	Hidden in derived class	Hidden in derived class	Hidden in derived class
Protected	Protected in derived class	Hidden in derived class	Protected in derived class

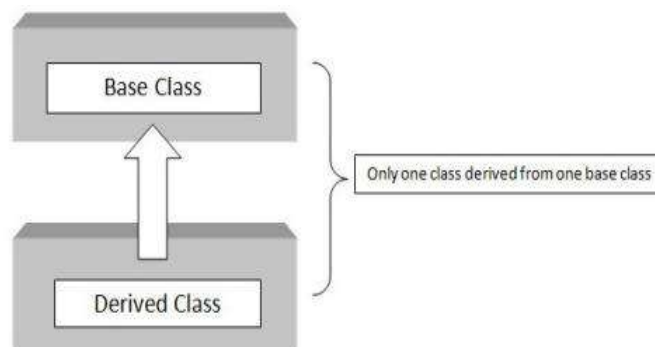
TYPES OF INHERITANCE BASED ON DERIVED CLASSES

Inheritance based on derived classes can be categorized as follows:

- Single Inheritance
- Multiple Inheritance (will be covered after Mid Exam)
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance

Single Inheritance:

In this type of inheritance there is one base class and one derived class. As shown in the figure below, in single inheritance only one class can be derived from the base class. Based on the visibility mode used or access specifier used while deriving, the properties of the base class are derived

**Syntax for single Inheritance:**

```

class A // base class
{
    // body of the class
};
class B : access_specifier A // derived class
{
    // body of the class
};
  
```

Example code for single Inheritance:

```

#include <iostream>
using namespace std;
class base { //single base class
public:
    int x;

    base ( ){
        cout << "Calling base constructor" << endl;
    }
};
  
```

```
    }

    void getdata( ) {
        cout << "Enter the value of x = ";
        cin >> x;
    }
};

class derive : public base { //single derived class
private:
    int y;
public:
    derive(){
        cout << "Calling derive constructor" << endl;
    }

    void readdata( ) {
        cout << "Enter the value of y = ";
        cin >> y;
    }

    void product( ) {
        cout << "Product = " << x * y;
    }
};

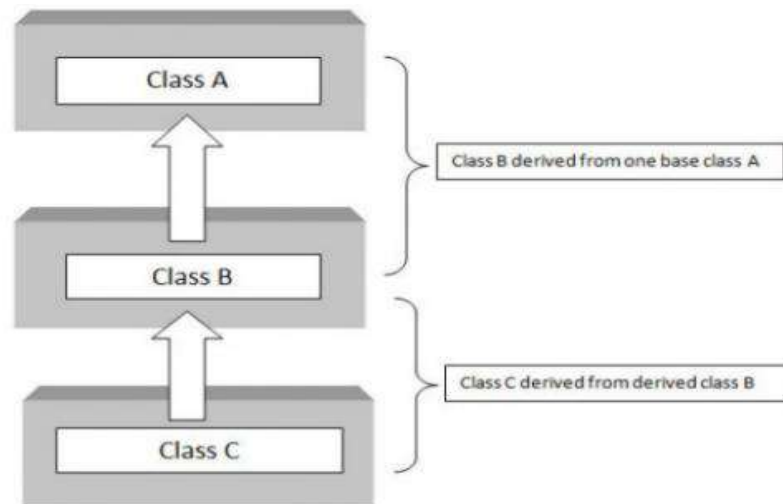
int main( ) {
    derive a; //object of derived class
    a.getdata( );
    a.readdata( );
    a.product( );
    return 0;
} //end of program
```

Sample Run

```
Calling base constructor
Calling derive constructor
Enter the value of x = 3
Enter the value of y = 4
Product = 12
```

Multilevel Inheritance:

- If a class is derived from another derived class then it is called multilevel inheritance, so in multilevel inheritance, a class has more than one parent class.
- As shown in the figure below, class C has class B and class A as parent classes.
- As in other inheritance, based on the visibility mode used or access specifier used. while deriving, the properties of the base class are derived. Access specifier can be private, protected or public



Syntax for multilevel Inheritance:

```

class A // base class
{
    // body of the class
};
class B : access_specifier A // derived class
{
    // body of the class
};
class C : access_specifier B // derived from class B
{
    // body of the class
};
  
```

Example code for multilevel Inheritance:

```

#include<iostream>
using namespace std;
class base { // single base class
public:
    int x;

    base ( ){
        cout << "Calling base constructor" << endl;
    }

    void getdata( ) {
        cout << "Enter value of x= ";
        cin >> x;
    }
};

class derive1 : public base { // derived class from base class
public:
    int y;

    derive1( ){
        cout << "Calling derive constructor" << endl;
    }
}
  
```

```

        void readdata( ) {
            cout << "\nEnter value of y= ";
            cin >> y;
        }
};

class derive2 : public derive1{ // derived from class derive1
private:
    int z;
public:
    derive( ){
        cout << "Calling derive constructor" << endl;
    }

    void indata( ) {
        cout << "\nEnter value of z= ";
        cin >> z;
    }

    void product( ) {
        cout << "\nProduct= " << x * y * z;
    }
};

int main( ) {
    derive2 a; //object of derived class
    a.getdata( );
    a.readdata( );
    a.indata( );
    a.product( );
    return 0;
} //end of program

```

Sample Run

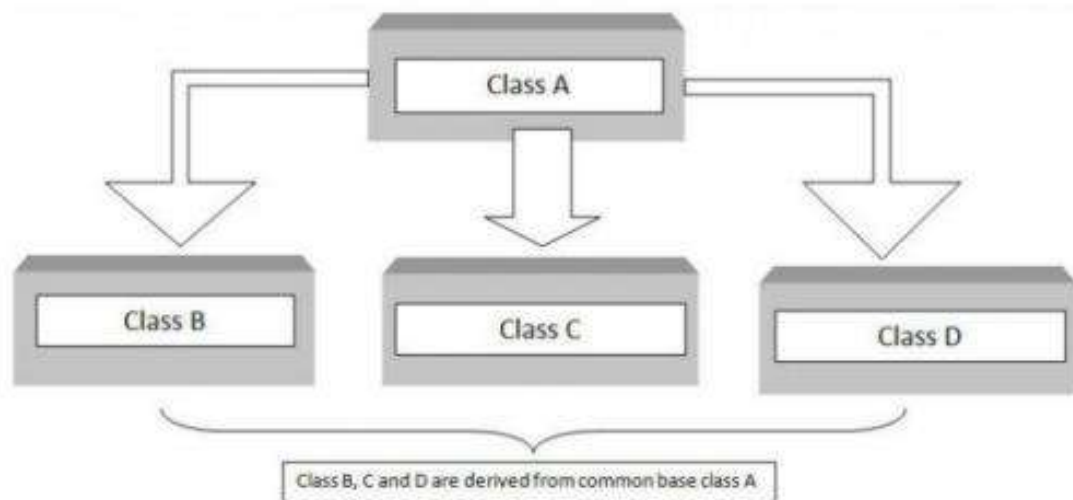
```

Calling base constructor
Calling derive1 constructor
Calling derive2 constructor
Enter value of x= 2
Enter value of y= 3
Enter value of z= 3 Product= 18

```

Hierarchical Inheritance:

When several classes are derived from a common base class it is called as hierarchical inheritance. In C++ hierarchical inheritance, the feature of the base class is inherited onto more than one sub-class. For example, a car is a common class from which Audi, Ferrari, Mustang etc can be derived. As shown in the figure below, in C++ hierarchical inheritance all the derived classes have a common base class. The base class includes all the features that are common to derived classes.



Syntax for hierarchical Inheritance:

```

class A // base class
{
    // body of the class
};
class B : access_specifier A // derived class from A
{
    // body of the class
};
class C : access_specifier A // derived class from A
{
    // body of the class
};
class D : access_specifier A // derived class from A
{
    // body of the class
};
  
```

Example code for hierarchical Inheritance:

```

#include<iostream>
using namespace std;
class A { //single base class
public:
    int x, y;

    A(int val1, int val2): x(val1),y(val2) {
        cout << "A class constructor called with values: " << val1,val2<< endl;
    }

    void getdata( ) {
        cout << "\n Enter value of x and y:\n";
        cin >> x >> y;
    }
};

class B : public A { //B is derived from class base
public:
    int z;

    B(int val1, int val2, int val3): A(val1,val2), z(val3) {
  
```

```
        cout << "B class constructor called with value: " << val3 << endl;
    }

    void product( ) {
        cout << "\n Product= " << x * y * z;
    }
};
class C : public A { //C is also derived from class base
public:
    void sum() {
        cout << "\n Sum= " << x + y;
    }
};
int main( ) {
    B obj1(2,3,4); //object of derived class B
    C obj2; //object of derived class C
    obj1.product( );
    obj2.getdata( );
    obj2.sum( );
    return 0;
} //end of program
```

Sample Run

A class constructor called with values: 2 3

B class constructor called with value: 4

Product= 24

Enter value of x and y: 2 3

Sum= 5

Exercise

1. Create a base class with the following members:

- Private integer privateInt
- Protected integer protectedInt
- Public integer publicInt

Create getters and setters for each of these variables. Derive 3 classes from the base class with the three types of inheritance based on **visibility**(public, protected, private). You can name these classes as **publicChild**, **privateChild** and **protectedChild**. After doing this, try and figure out which member you can access publicly or through getters and setters. Then print out the way that you were able to access them. **For example**, if you did private inheritance, you could not be able to access the members in the child classes, and would need to use their getters or setters.

2. Create a class Teacher with the following protected attributes: Name, Age, and Institute. Derive three classes from it that has the following names: **HumanitiesTeacher**, **ScienceTeacher**, **MathsTeacher**. These classes should have the following members:

- Department (this should have the value "science", "maths" or "humanities")
- Course Name
- Designation (for example, lecturer, professor, etc)

Create proper accessors and mutators for the attributes. Create objects for each of the classes and display the values. You can ask the user to input the values.

3. A defense organization is making a hierarchy of different types of weapons. They have classified the nuclear bomb as follows: **Weapons** → **Hot Weapons** → **Bombs** → **Nuclear Bombs**. Create classes and apply inheritance as necessary for the above hierarchy. Each class should have a method called: "xxxxxDescription", where xxxx would be class name. The method should print out what that weapon does. Eg. Hot weapons uses gunpowder, or explode. Bombs blow up. Nuclear bombs blow up, and use nuclear fission and fusion.
4. A bakery is making a program to calculate the bills for each of their customers. The items in the bakery are categorized as follows:

Item:

- Name
- Quantity

Baked Goods: (derived from item)

- Discount = 10%

Cakes: (derived from baked goods)

- Price = 600

Bread: (derived from baked goods)

- Price = 200

Drinks: (derived from item)

- Discount = 5%
- Price = 100

In your main function, as a bakery personnel, input the items and their quantities and calculate the bill accordingly.

Object Oriented Programming (CT-260)

Lab 06

Introduction to polymorphism

Objectives

The objective of this lab is to familiarize students with the concept of polymorphism in object oriented programming. By the end of this lab, students will be able to understand and implement function and operator overloading in C++ language.

Tools Required

DevC++ IDE / Visual Studio / Visual Code

Course Coordinator – Dr. Murk Marvi

Course Instructor – Ms. Samia Masood Awan

Lab Instructor – Ms. Samia Masood Awan

Prepared By Department of Computer Science and Information Technology

NED University of Engineering and Technology

INTRODUCTION TO POLYMORPHISM

The word polymorphism means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance. C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

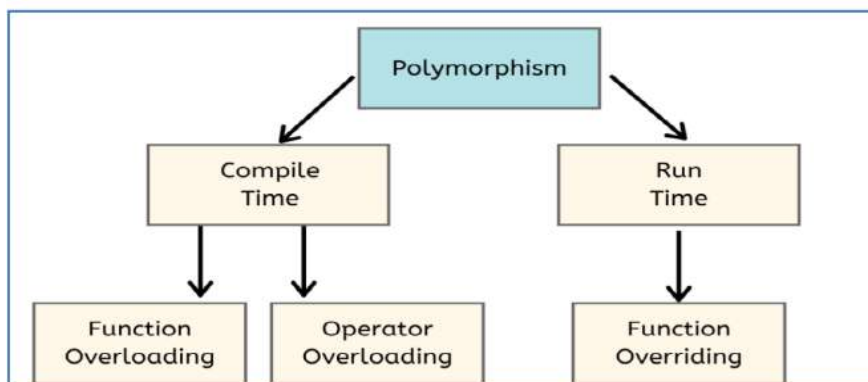
Real World Example:

A real-life example of polymorphism is that a person at the same time can have different characteristics. A man at the same time is a father, a husband, an employee, so the same person possesses different behavior in different situations. This is called as polymorphism. Polymorphism is considered as one of the important features of Object-Oriented Programming.

TYPES OF POLYMORPHISM:

In C++ polymorphism is mainly divided into two types:

- Compile time Polymorphism
- Runtime Polymorphism



Compile time Polymorphism

This type of polymorphism is achieved by function overloading or operator overloading.

Function Overloading:

- When there are multiple functions with same name but different parameters then these functions are said to be overloaded.
- Functions can be overloaded by a change in the number of arguments or/and change in the type of arguments.

Example Code for Function Overloading:

```
// C++ program for function overloading
#include<iostream>
using namespace std;
class overload {
public:
// function with 1 int parameter
void func(int x) {
    cout << "value of x is " << x << endl;
}
// function with same name but 1 double parameter
void func(double x) {
```

```

        cout << "value of x is " << x << endl;
    }
    // function with same name and 2 int parameters
    void func(int x, int y) {
        cout << "value of x and y is " << x << ", " << y << endl;
    }
};
int main() {
    overload obj1;
    // Which function is called will depend on the parameters passed. The first 'func' is called
    obj1.func(7);
    // The second 'func' is called
    obj1.func(9.132);
    // The third 'func' is called
    obj1.func(85,64);
    return 0;
}

```

Sample Run:

value of x is 7

value of x is 9.132

value of x and y is 85, 64

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**.

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.

Example Code for Function Overloading:

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:

```
// 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;
    }
}
```

Run time Polymorphism

This type of polymorphism is achieved by Function Overriding.

Function Overriding

Function overriding is a feature that allows us to have a same function in child class which is already present in the parent class.

- A child class inherits the data members and member functions of parent class, but when you want to override a functionality in the child class then you can use function overriding. It is like creating a new version of an old function, in the child class.
- To override a function you must have the same signature in the child class.

Syntax for Function Overriding:

```
public class Parent{
    access_modifier:
    return_type method_name(){}
};

public class child : public Parent {
    access_modifier:
    return_type method_name(){}
};
```

Example Code for Function Overriding:

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

```

class BaseClass {
public:
    void disp( ){
        cout<<"Function of Parent Class";
    }
};
class DerivedClass: public BaseClass{
public:
    void disp( ) {
        cout<<"Function of Child Class";
    }
};
int main( ) {
    DerivedClass obj = DerivedClass( );
    obj.disp( );
    return 0;
}

```

Sample Run:
Function of Child Class

Note: In function overriding, the function in parent class is called the overridden function and function in child class is called overriding function.

Function Overriding - during runtime

```

Class a{
public:
    virtual void display( ){ cout << "hello"; }
};

Class b:public a{
public:
    void display( ){ cout << "bye";}
};

```

Example code:

```

// Function Overriding
#include<iostream>
using namespace std;
class BaseClass{
public:
    virtual void Display( ){
        cout << "\nThis is Display( ) method of BaseClass";
    }
    void Show( ){
        cout << "\nThis is Show() method of BaseClass";
    }
};

class DerivedClass : public BaseClass {
public:
    // Overriding method - new working of base class display method
    void Display( ){

```

```

        cout << "\nThis is Display( ) method of DerivedClass";
    }
};

// Driver code
int main(){
    DerivedClass dr;
    BaseClass &bs = dr;
    bs.Display( );
    dr.Show( );
}

```

Sample run

```

This is Display() method of DerivedClass
This is Show() method of BaseClass

```

Exercise

1. Create a class named Shape. All our shapes will be inherited from this class. It will contain the following data members and functions: numberOfSides, area, parametrized constructor, accessors and mutators for the data members. Create classes called Rectangle, Circle and Triangle, which are all inherited from the class Shape. Create a class called Square which is inherited from Rectangle. The derived classes will have the following members:

Rectangle: length, width, parameterized constructor, generateArea() – should place the result in area.

Circle: radius, parameterized constructor, generateArea() – should place the result in area

Triangle: height, base, parameterized constructor, generateArea() – should place the result in area (Area = height*base/2)

Square:

- It should have a parameterized constructor that takes one side as input. The constructor should call the constructor for the Rectangle class with that value as parameters.
- checkSides(); - checks if both sides are equal. Sides are inherited from Rectangle.
- generateArea() – should place the result in area

You must make use of parameterized constructors to initialize the values.

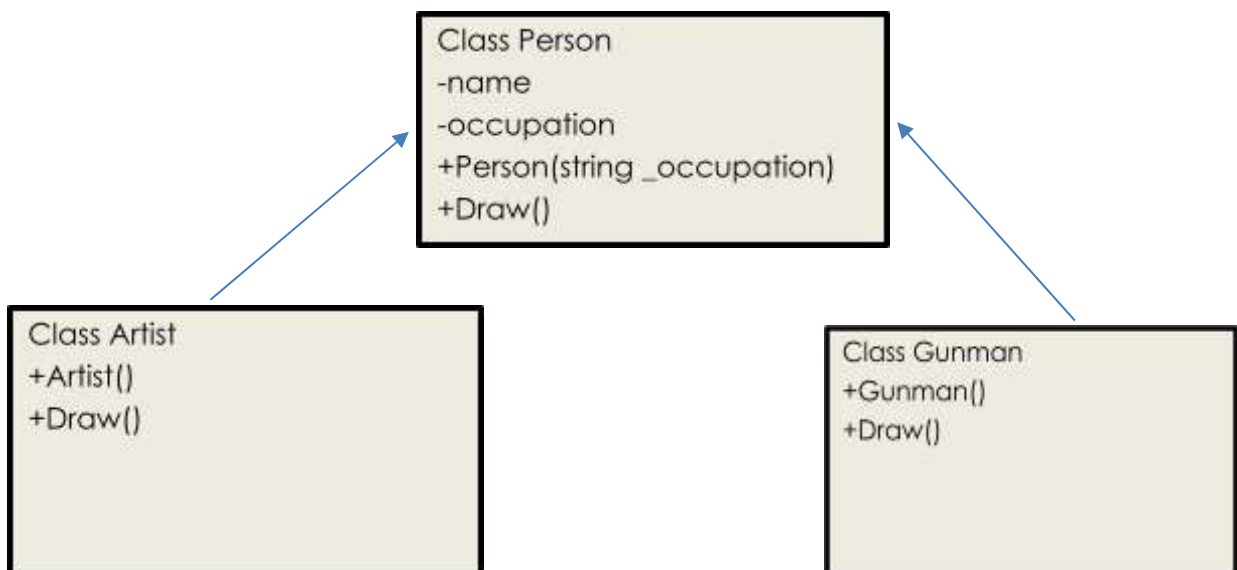
1. Create a class called **Calculator** that has three private member variables Num1, Num2, Num3. In this class, you have to overload the functions for addition and multiplication such that they take two and three inputs respectively. You also have to make methods for subtraction and division. **For example:** add(1,2) and add(1,2,3) similarly for multiply, it would be multiply(1,2) and multiply(1,2,3). You may ask the user for input at the time of object creation. Afterwards just demonstrate how the functions are being called.
2. Create a class called **Vector** which represents a two-dimensional vector with **x** and **y** components. The class should have the following member functions:
 - A constructor that initializes the x and y components of the vector.
 - An overloaded operator + that adds two Vector objects and returns a new Vector object.
 - An overloaded operator - that subtracts two Vector objects and returns a new Vector object.

- An overloaded operator `*` that multiplies a Vector object by a scalar value and returns a new Vector object.
- An overloaded operator `/` that divides a Vector object by a scalar value and returns a new Vector object.
- A member function `magnitude` that returns the magnitude of the vector.

In the main function, create two Vector objects and demonstrate the use of all the overloaded operators.

4. Create the classes following the diagram given below. Keep the following things in mind:

- When an object of Artist is created, the value "artist" will be set to occupation.
- When an object of Gunman is created, the value "gunman" will be set to occupation.
- `Person::Draw()` will print out "A person can draw in many ways"
- `Artist::Draw()` will print out "An artist can draw with a paint brush"
- `Gunman::Draw()` will print out "A gunman draws a gun to shoot"
- Write a test code by creating an array of pointers of type Person. Dynamically create objects of each class and store address in the array. After that call Draw function for each object.



Object Oriented Programming (CT-260)

Lab 07

Friend Functions, Friend Classes, Dynamic Array Class - Case Study

Objectives

The objective of this lab is to familiarize students with the concept of dynamic arrays, friend functions and friend classes.

Tools Required

DevC++ IDE

Course Coordinator – Dr. Murk Marvi

Course Instructor – Ms. Samia Masood Awan

Lab Instructor – Ms. Samia Masood Awan

Prepared By Department of Computer Science and Information Technology

NED University of Engineering and Technology

Dynamic Array

The Array class is defined with private data members size and ptr where size represents the size of the array, and ptr is a pointer to **dynamically** allocated memory to store the array elements.

Sample Code:

```
#include<iostream>
#include <cstdlib>
using namespace std;
class Array{
    int size;
    int *ptr;
public:
    Array(int s=0):size(s), ptr(NULL){
        if(size>0)
            ptr=new int[size];
        if(ptr==NULL)
            exit(1);
    }
    Array(Array& ob){
        if(ob.size>0){
            size=ob.size;
            ptr=new int[size];
            if(ptr==NULL)
                exit(1);
            for(int i=0;i<size;i++){
                ptr[i]=ob[i];
            }
        }
        else
            exit(1);
    }
    int& operator[](int in){
        if(in>=size){
            cout<<"index "<<in<<" is out of bound"<<endl;
            exit(1);
        }
        return ptr[in];
    }
    Array operator+(Array& ob){
        if (size==ob.size){
            Array temp(size);
            for(int i=0; i<size;i++)
                temp[i]=ptr[i]+ob[i];
            return temp;
        }
        else{
            cout<<"The size of array does not match!"<<endl;
            exit(1);
        }
    }
}
```



```

        int operator*(Array& ob){
            if (size==ob.size){
                int temp=0;
                for(int i=0; i<size;i++)
                    temp+=ptr[i]*ob[i];
                return temp;
            }
            else{
                cout<<"The size of array does not match!"<<endl;
                exit(1);
            }
        }

    ~Array(){
        delete[] ptr;
    }
};

int main(){
    Array a1(5), a2(5);
    for(int i=0;i<5;i++){
        a1[i]=i*2;
        a2[i]=i+3;
    }
    for (int i=0;i<5;i++)
        cout<< a1[i]<<endl;
    for (int i=0;i<5;i++)
        cout<< a2[i]<<endl;

    Array a3(a1);
    Array a4= a1+a2;
    for(int i=0;i<5;i++)
        cout<<a4[i]<<"\t";
    cout<<endl;
    cout<<a1*a2<<endl;
}

```

The constructor `Array(int s=0)` is a default constructor that initializes an Array object. It takes an optional parameter `s` to specify the size of the array. If `s` is greater than zero, it allocates memory for the array elements using `new`. If `ptr` is NULL after allocation, it means memory allocation failed, so it exits the program with an error code.

The copy constructor `Array(Array& ob)` is used to create a new Array object by copying the values from another Array object `ob`. It checks if `ob.size` is greater than zero and allocates memory for the new object. Then, it copies the elements from `ob.ptr` to the new object's `ptr` using a loop.

The overloaded indexing operator `int& operator[](int in)` allows accessing the elements of the array using the subscript operator `[]`. It takes an index `in` and returns a reference to the

element at that index. Before returning the reference, it checks if the index is out of bounds (greater than or equal to size) and exits the program if it is.

Friend Functions

The concepts of encapsulation and data hiding dictate that nonmember functions should not be able to access an object's private or protected data. The policy is, if you're not a member, you can't get in. However, there are situations where such rigid discrimination leads to considerable inconvenience.

Friend as Bridges

Imagine that you want a function to operate on objects of two different classes. Perhaps the function will take objects of the two classes as arguments, and operate on their private data. In this situation there's nothing like a friend function. Here's a simple example, FRIEND that shows how friend functions can act as a bridge between two classes:

Sample Code 1:

```
#include<iostream>
using namespace std;
class beta;
class alpha{
private:
    int data;
public:
    alpha(): data(3) { }
    friend int frifunc(alpha, beta);
};

class beta{
private:
    int data;
public:
    beta(): data(7) { }
    friend int frifunc(alpha, beta);
};

int frifunc(alpha a, beta b){
    return(a.data+b.data);
}

int main(){
    alpha aa;
    beta bb;
    cout << frifunc(aa, bb) << endl;
    return 0;
}
```

In this program, the two classes are **alpha** and **beta**. The constructors in these classes initialize their single data items to fixed values (3 in **alpha** and 7 in **beta**).

We want the function **frifunc()** to have access to both of these private data members, so we make it a Friend function. It's declared with the friend keyword in both classes:

```
friend int frifunc(alpha, beta);
```

This declaration can be placed anywhere in the class; it doesn't matter whether it goes in the public or the private section.

An object of each class is passed as an argument to the function `frifunc()`, and it accesses the private data member of both classes through these arguments. The function doesn't do much: It adds the data items and returns the sum. The `main()` program calls this function and prints the result.

A minor point: Remember that a class can't be referred to until it has been declared. Class `beta` is referred to in the declaration of the function `frifunc()` in class `alpha`, so `beta` must be declared before `alpha`. Hence the declaration

```
class beta;
```

at the beginning of the program.

Sample Code 2:

```
#include<iostream>
using namespace std;
class Array{
    int size;
    int *ptr;
    public:
    Array(int s=0):size(s), ptr(NULL){
        if(size>0)
            ptr=new int[size];
        if(ptr==NULL)
            exit(1);
    }

    int& operator[](int in){
        if(in>=size){
            cout<<"index "<<in<<" is out of bound"<<endl;
            exit(1);
        }
        return ptr[in];
    }
    friend int dotproduct(Array, Array);
};

int dotproduct(Array a1, Array a2){
    if(a1.size==a2.size){
        int temp=0;
        for(int i=0;i<a1.size;i++)
            temp+=a1[i]*a2[i];
        return temp;
    }
    else{
        cout<<"Size mismatch error"<<endl;
```

```

        exit(1);
    }
}

int main() {
    Array a1(5), a2(5);
    for(int i=0;i<5;i++){
        a1[i]=i*2;
        a2[i]=i+3;
    }
    for (int i=0;i<5;i++)
        cout<< a1[i]<<endl;
    for (int i=0;i<5;i++)
        cout<< a2[i]<<endl;
    cout<<dotproduct(a1,a2)<<endl;
    return 0;
}

```

Friend Classes

The member functions of a class can all be made friends at the same time when you make the entire class a friend. The program FRICLASS shows how this looks.

Sample Code

```

#include<iostream>
using namespace std;

class alpha{
private:
    int data1;
public:
    alpha( ): data1(99) { }
    friend class beta;
};

class beta{
public:
    void func1(alpha a) { cout << "\ndata1=" << a.data1; }
    void func2(alpha a) { cout << "\ndata1=" << a.data1; }
};

int main( ){
    alpha a;
    beta b;
    b.func1(a);
    b.func2(a);
    cout << endl;
    return 0;    }

```

In class alpha the entire class **beta** is proclaimed a friend. Now all the member functions of beta can access the private data of alpha (in this program, the single data item data1).

Note that in the friend declaration we specify that beta is a class using the class keyword:

friend class beta;

We could have also declared beta to be a class before the alpha class specifier, as in previous examples

class beta;

and then, within alpha, referred to beta without the class keyword:

friend beta;

Exercise

1. Create a class for 2D dynamic Array, of type integer, by using a double pointer, that is, a pointer to the array of pointers. The class must have the following methods.
 - Default, parameterized, and copy constructor
 - Overloaded assignment operator
 - Overloaded +, -, and * operator
 - Overloaded indexing operator
 - Destructor.

Write a test program for verifying the working of the designed class.

2. By using the 2D Array class that you created in Problem 1, solve the following problem. Given a 2D Array (matrix) of integers of size m x n with the following two properties:
 - Each row is sorted in non-decreasing order
 - The first integer of each row is greater than the last integer of the previous row and an integer target, return true if the target is in the matrix otherwise false otherwise.

Hint: Apply binary search algorithm.

Example 1:

1	3	5	7
10	11	16	20
23	30	34	60

Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

Output: true

Example 2:

1	3	5	7
10	11	16	20
23	30	34	60

Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13

Output: false

3. A company has decided to update the salaries of the employees for which updation is required in the payroll system so that employees can be paid according to the revised budget. How can you implement the concept of friend class here? Consider there are 2 classes. One is "Employee" having private data members (name, id, designation, salary etc.) and the other is "Payroll". The function for updating salaries can be made inside the "Payroll" class that can access the private member "salary" of the "Employee" class and allow the required updation. Implement this scenario.
4. Repeat Q3 and implement this scenario using the friend function.

Object Oriented Programming (CT-260)

Lab 08

Multiple Inheritance, Diamond Problem, Virtual Inheritance and Virtual Functions

Objectives

The objective of this lab is to familiarize students with the multiple inheritance and the problem related to it. Implementation of Virtual functions associated with inheritance.

Tools Required

DevC++ IDE

Course Coordinator –

Course Instructor –

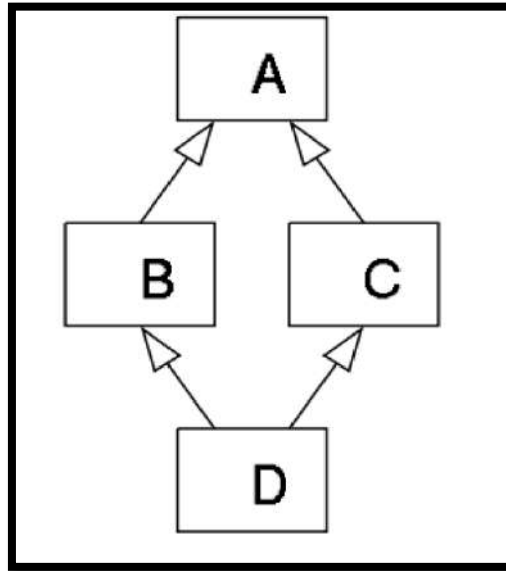
Lab Instructor –

Prepared By Department of Computer Science and Information Technology

NED University of Engineering and Technology

Diamond Problem in Hybrid Inheritance:

In case of hybrid inheritance, a Diamond problem may arise. The “dreaded diamond” refers to a class structure in which a particular class appears more than once in a class’s inheritance hierarchy.



Example of Diamond Problem:

```
#include <iostream>
using namespace std;
class A{
public:
    int a;
};
class B : public A{
public:
    int b;
};
class C : public A{
public:
    int c;
};
class D : public B, public C{
public:
    int d;
};
int main(){
    D obj;
    obj.a = 200; // will cause an error
}
```

How to Solve the Diamond Problem?

Answer: Virtual Base Classes

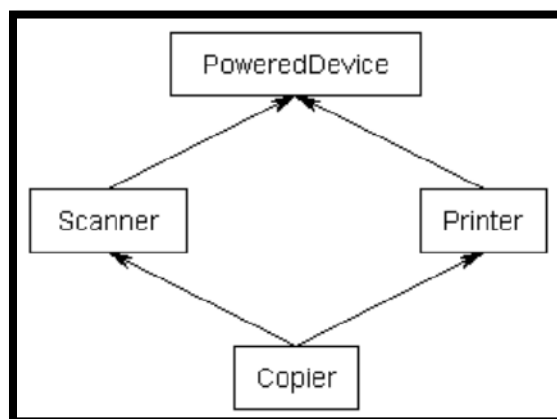
To **share** a base class, simply insert the “virtual” keyword in the inheritance list of the derived class. This creates what is called a **virtual base class**, which means there is only

one base object. The base object is shared between all objects in the inheritance tree and it is only constructed once.

Solving the Diamond Problem:

```
#include <iostream>
using namespace std;
class A{
public:
    int a;
};
class B : virtual public A{ // adding the virtual keyword
public:
    int b;
};
class C : virtual public A{ // adding the virtual keyword
public:
    int c;
};
class D : public B, public C{
public:
    int d;
};
int main(){
    D obj;
    obj.a = 200; // will no longer cause an error
}
```

Diamond Problem with Real Classes and Objects:



```
#include <iostream>
using namespace std;
class PoweredDevice{
public:
    PoweredDevice(int power){
        cout << "PoweredDevice: " << power << '\n';
    }
};
class Scanner : public PoweredDevice{
public:
    Scanner(int scanner, int power) : PoweredDevice(power){
        cout << "Scanner: " << scanner << '\n';
    }
}
```

```

    }
};
class Printer : public PoweredDevice{
public:
    Printer(int printer, int power) : PoweredDevice(power){
        cout << "Printer: " << printer << '\n';
    }
};
class Copier : public Scanner, public Printer{
public:
    Copier(int scanner, int printer, int power) : Scanner(scanner, power),
        Printer(printer, power) {}
};
int main(){
    Copier copier(1, 2, 3);
    return 0;
}

```

If you were to create a Copier class object, by default you would end up with two copies of the PoweredDevice class -- one from Printer, and one from Scanner. This has the following structure:

```

PoweredDevice: 3
Scanner: 1
PoweredDevice: 3
Printer: 2

-----
Process exited after 0.2705 seconds with return value 0
Press any key to continue . . .

```

By using Virtual Base Classes:

```

#include <iostream>
using namespace std;
class PoweredDevice
{
public:
    PoweredDevice(int power){
        cout << "PoweredDevice: " << power << '\n';
    }
};
class Scanner : virtual public PoweredDevice{
public:
    Scanner(int scanner, int power) : PoweredDevice(power){
        cout << "Scanner: " << scanner << '\n';
    }
};
class Printer : virtual public PoweredDevice{
public:
    Printer(int printer, int power) : PoweredDevice(power){
        cout << "Printer: " << printer << '\n';
    }
};
class Copier : public Scanner, public Printer{

```

```
public:
    Copier(int scanner, int printer, int power) : PoweredDevice(power),
        Scanner(scanner, power), Printer(printer, power){ }
};
int main(){
    Copier copier(1, 2, 3);
    return 0;
}
```

Now, when you create a Copier class object, you will get only one copy of PoweredDevice per Copier that will be shared by both Scanner and Printer. However, this leads to one more problem: if Scanner and Printer share a PoweredDevice base class, who is responsible for creating it?

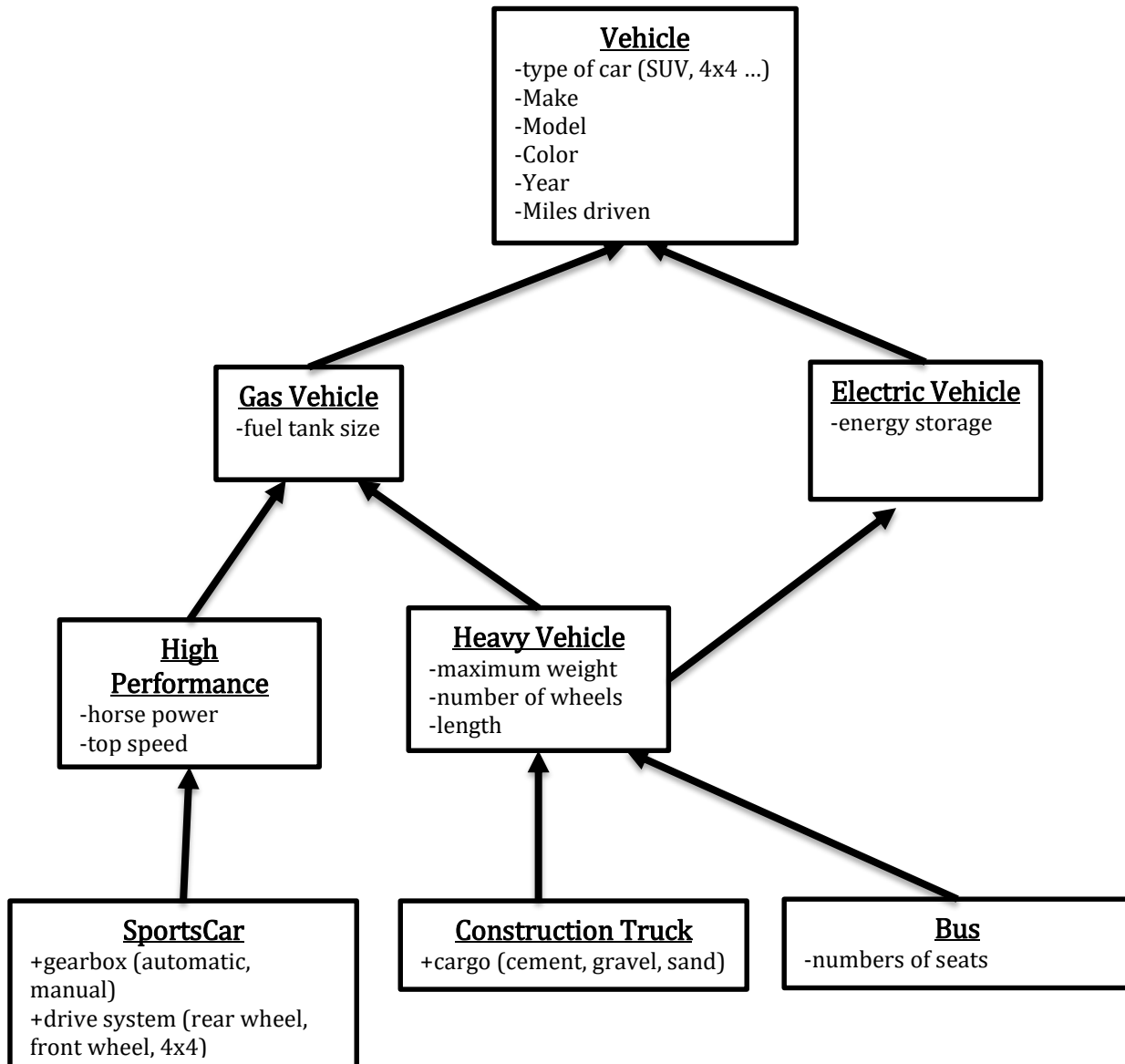
The answer, as it turns out, is Copier. The Copier constructor is responsible for creating PoweredDevice. Consequently, this is one time when Copier is allowed to call a non-immediate-parent constructor directly.

```
PoweredDevice: 3
Scanner: 1
Printer: 2

-----
Process exited after 0.3112 seconds with return value 0
Press any key to continue . . .
```

Exercise

1. Implement the given UML class diagram by following the guidelines given below.
 - All the values are required to be set through parameterized constructor
 - Provide necessary accessor methods where required.
 - Create an object of the class bus by initializing it through a parameterized constructor in the main function and display all data members by calling the display function of class bus.



2. Suppose you are designing a game engine for a new video game. The game engine needs to support different types of characters, such as warriors, mages, and archers. Each character type has specific attributes and abilities. The game also includes non-playable characters (NPCs) that have their own unique behaviors. Additionally, there are special characters called "Mighty" that possess both the abilities of a warrior and a mage. To design the character classes and utilize multiple inheritance in this scenario, you can create a class hierarchy with appropriate base classes and derived classes. Here's a possible implementation:
 - Create a base class called Character that contains common attributes and behaviors for all characters, such as name, level, and health.
 - Create derived classes for specific character types: a. Warrior class, derived from Character, which includes attributes and abilities specific to warriors, such as strength, melee

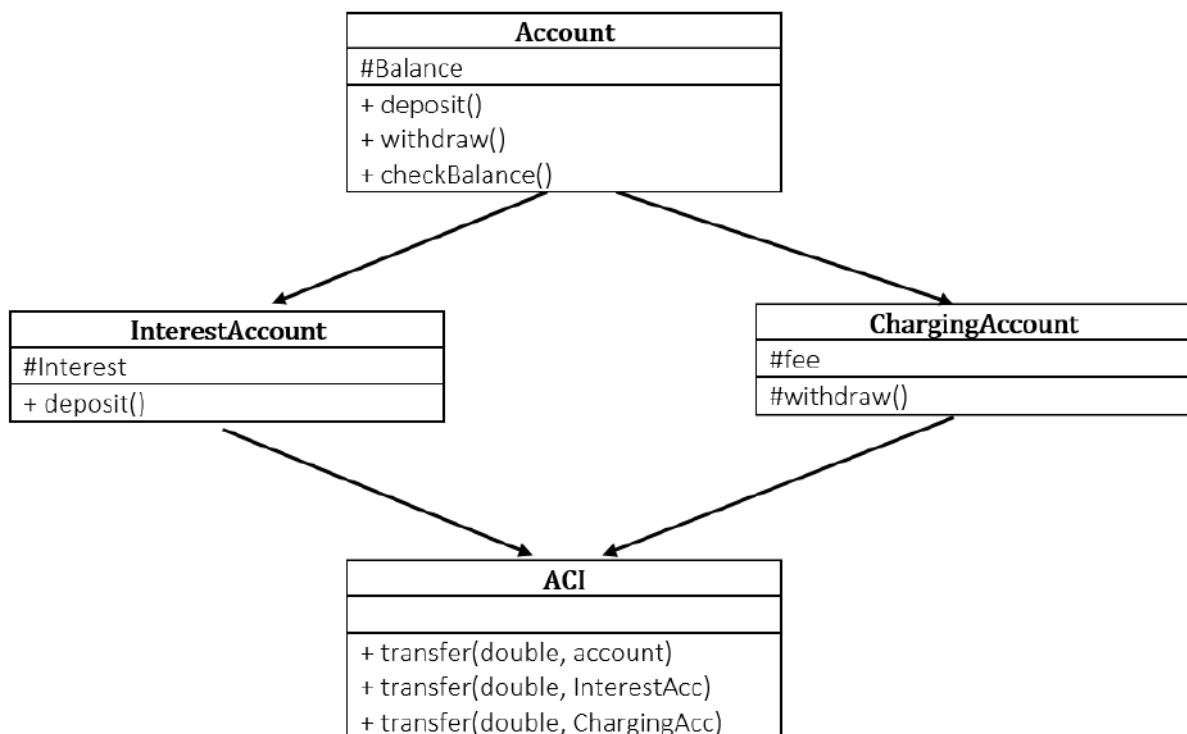
weapons proficiency, and a "slash" ability. b. Mage class, derived from Character, which includes attributes and abilities specific to mages, such as intelligence, spell casting proficiency, and a "fireball" ability. c. Archer class, derived from Character, which includes attributes and abilities specific to archers, such as dexterity, ranged weapons proficiency, and a "rapid shot" ability.

- Create a class called NPC (Non-Playable Character), derived from Character, which includes additional behaviors specific to non-playable characters, such as predefined movement patterns or scripted dialogues.
- Create a class called Mighty, derived from both Warrior and Mage, to represent special Mighty characters. This class inherits attributes and abilities from both Warrior and Mage, allowing the Mighty to possess the combined traits of a warrior and a mage. For example, a Mighty might have high strength and melee prowess like a warrior, as well as the ability to cast powerful spells like a mage.

By using multiple inheritance, you can design a class hierarchy that allows for the sharing of common attributes and behaviors while also enabling specific character types to inherit and extend upon those features. The Mighty class demonstrates the ability to combine attributes and abilities from multiple base classes, showcasing the flexibility of multiple inheritance in this scenario.

3. Implement the given UML class diagram by following the guidelines given below.

- The interest Account class adds interest for every deposit, assuming a default of 30%.
- The charging account class charges a default fee of Rs. 25 for every withdrawal.
- Transfer method of ACI class tasks two parameters: amount to be transferred and object of class in which we have to transfer that amount.
- Make parameterized constructor and default constructor to take user input for all data members.
- Make a driver program to test all functionalities.



4. You are developing a software application for a library that manages various types of media, including books, magazines, and DVDs. Each type of media has specific attributes and functionalities, such as the author for books, issue number for magazines, and director for DVDs. Additionally, there are certain operations that are common to all types of media, such as borrowing, returning, and displaying information. Design a class hierarchy using multiple inheritance to handle the different types of media and their functionalities.
- Create a base class called Media to represent the common attributes and functionalities shared by all types of media. This class will include operations such as borrowing, returning, and displaying information.
 - Create individual classes for each type of media, such as Book, Magazine, and DVD. Each class will inherit from both the Media class and their respective specific attribute classes.
 - Create individual classes for each type of media, such as Book, Magazine, and DVD. Each class will inherit from both the Media class and their respective specific attribute classes.
 - Create individual classes for each type of media, such as Book, Magazine, and DVD. Each class will inherit from both the Media class and their respective specific attribute classes.

With this class hierarchy, the library application can handle different types of media such as books, magazines, and DVDs. The common functionalities like borrowing, returning, and displaying information will be inherited from the Media class, while the specific attributes and functionalities for each media type will be defined in their respective classes.

Object Oriented Programming (CT-260)

Lab 09

Abstract Base Class and Pure Virtual Function

Objectives

The objective of this lab is to familiarize students with the abstract base class and pure virtual function.

Tools Required

DevC++ IDE

Course Coordinator –

Course Instructor –

Lab Instructor –

Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

Theory:

A base class which does not have the implementation of one or more of its virtual member functions is said to be an abstract base class. It serves as a framework in a hierarchy and the derived classes will be more specific and detailed.

Further, a virtual member function which does not have its implementation/definition is called a “pure” virtual member function. An abstract base class is said to be incomplete – missing some of the definition of its virtual member functions– and cannot be used to instantiate objects. However, it still can be used to instantiate pointer that will be used to send messages to different objects in the inheritance hierarchy to affect polymorphism.

Pure virtual member function declaration:

```
virtual return_type functionName( argument_list ) = 0;
```

	Abstract base class	Concrete base class
Data member	Yes	Yes
Virtual function	Yes	Yes
Pure virtual function	Yes	No
Object instantiation	No	Yes
Pointer instantiation	Yes	Yes

Abstract Classes:

An abstract class is a class that has at least one pure virtual function (i.e., a function that has no function body). The classes inheriting the abstract class must provide a definition for the pure virtual function; otherwise, the subclass would become an abstract class itself.

Abstract classes are essential for providing abstraction to the code to make it reusable and extendable. For Example, a Vehicle parent class with Truck and Motorbike inheriting from it is an abstraction that easily allows more vehicles to be added. However, even though all vehicles have wheels, not all vehicles have the same number of wheels – this is where a pure virtual function is needed.

Virtual Functions:

A pure virtual function (or abstract function) in C++ is a virtual function for which we don't have implementation (we do not have function body), we only declare it. A pure virtual function is declared by assigning 0 in declaration.

See the following example.

```
// An abstract class
class Test
{
    // Data members of class
public:
    // Pure Virtual Function
    virtual void show() = 0;

    /* Other members */
};
```


Note

- The = 0 syntax doesn't mean we are assigning 0 to the function. It's just the way we define pure virtual function.
- A pure virtual function is implemented by classes which are derived from Abstract class.
- A class is abstract if it has at least one pure virtual function. In example given above, Test is abstract class as it has virtual function show().

Example

A pure virtual function is implemented by classes which are derived from an Abstract class. Following is a simple example to demonstrate the same.

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

class Base
{
    int x;
public:
    virtual void fun() = 0;
    int getX() { return x; }
};

// This class inherits from Base and implements fun()
class Derived: public Base
{
    int y;
public:
    void fun() { cout << "fun() called"; }
};

int main(void)
{
    Derived d;
    d.fun();
    return 0;
}
```

Output

```
fun() called
```

Object of abstract class cannot be created.

```
// pure virtual functions make a class abstract
#include <iostream>
using namespace std;
class Test{
    int x;
public:
    virtual void show() = 0;
    int getX(){
        return x;
    }
};
```

```

    }
};

int main(void){
    Test t;
    return 0;
}

```

Output

Compiler Error: cannot declare variable 't' to be of abstract type 'Test' because the following virtual functions are pure within 'Test': note: virtual void Test::show()

An abstract class can have constructors. Consider this code that compiles and runs fine.

```

#include <iostream>
using namespace std;
// An abstract class with constructor
class Base{
protected:
    int x;
    virtual void fun( ) = 0;
public:
    Base(int i) { x = i; }
};

class Derived : public Base{
    int y;
public:
    Derived(int i, int j) : Base(i) { y = j; }
    void fun( ) { cout << "x = " << x << ", y = " << y; }
};

int main(void){
    Derived d(4, 5);
    d.fun( );
    return 0;
}

```

Output

x = 4, y = 5

Example: Abstract class and Virtual Function in calculating Area of Square and Circle:

```

#include <iostream>
using namespace std;

class Shape{
protected:
    float dimension;

```

```
public:
    void getDimension( ){
        cin >> dimension;
    }
private:
    // 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 length to calculate the area of a square: 4
Area of square: 16

Enter radius to calculate the area of a 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.

Exercise

1. Define an abstract class ***ArrayMultiplier*** with a pure virtual function ***calculate()***. Then, create two derived classes: ***ArrayMultiplier1D*** for 1D array multiplication and ***ArrayMultiplier2D*** for 2D array multiplication. In the ***calculate()*** function of each derived class, perform the multiplication operation based on the specific array dimension.
2. Write a program to calculate final bill after discount. "ImtiazStore" gives 7 percent discount on total_bill while "BinHashimStore" gives 5 percent discount on total_bill. You have to initialize value of total_bill through a constructor and then calculate final bill after discount for both stores using the concept of abstract class and virtual functions.
3. You are tasked with developing a car rental system for a company. The system should allow customers to rent cars based on availability and manage the inventory of cars. To implement this system, you decide to use object-oriented programming and apply the concept of abstraction.
 - Abstract Base Class: Vehicle - You create an abstract base class called Vehicle, which represents a generic vehicle in the rental system. The Vehicle class contains common attributes like carId, brand, and model. It also defines pure virtual functions such as ***isAvailable()*** and ***rent()***, which represent the availability of the vehicle and the process of renting it. Since these functions are pure virtual, they have no implementation in the base class.
 - Derived Class: Car - You derive a Car class from the Vehicle class, representing a car in the rental system. The Car class provides concrete implementations for the pure virtual functions ***isAvailable()*** and ***rent()***. In the ***isAvailable()*** function, you check the availability of the car by querying its specific availability attribute. In the ***rent()*** function, you update the availability status of the car after it has been rented.
 - Rental System Class: RentalSystem - You create a RentalSystem class responsible for managing the rental operations. This class interacts with the Vehicle objects through the base class interface. It has functions like ***rentVehicle()*** and ***returnVehicle()***, which take a Vehicle object as a parameter and call the corresponding ***isAvailable()*** and ***rent()*** functions. The RentalSystem class can handle rental operations for any type of Vehicle, as long as it inherits from the Vehicle base class.
 - Customer Class: Customer - You create a Customer class to represent the customers of the rental system. The Customer class has methods like ***rentVehicle()*** and ***returnVehicle()***, which take a Vehicle object as a parameter. These methods interact with the RentalSystem class to perform the rental operations without knowing the specific type of the vehicle. This allows for flexibility and extensibility if new types of vehicles are added to the system in the future.

By utilizing pure virtual functions, you create an abstraction that defines the common interface and behavior for all vehicles in the system. The derived classes provide concrete implementations specific to their type of vehicle. The RentalSystem class and the Customer class can interact with vehicles through the abstract base class interface, allowing for polymorphism and decoupling from the

specific implementations. This abstraction simplifies the codebase, promotes modularity, and allows for easy addition of new vehicle types without modifying existing classes. Write a test program by defining an array of pointers of the Base class and dynamic memory for allocating memory to objects created.

4. Suppose you have to send an encrypted message over the network. There are two techniques that you can use to encrypt your message. Suppose your message is "Hello". It will be encrypted as follows according to those 2 techniques.

Technique 1: This technique simply converts alphabets into their respective ASCII codes.	H=72 E=69 L=76 L=76 O=79	Now the string will become, 7269767679
Technique 2: This technique simply converts alphabets into their respective ASCII code and then adding 2.	H=72+2 E=69+2 L=76+2 L=76+2 O=79+2	Now the string will become, 7471787881

Make use of an abstract class ***EncryptionTechnique*** and pure virtual function ***encrypt*** and implement this scenario. Make derived classes ***EncryptionTechnique1*** and ***EncryptionTechnique2***. Make a test program that takes an input string and encrypts it using both encryption techniques calling their respective encrypt functions.

5. Design the corresponding decryption techniques for Q4.

Object Oriented Programming (CT-260)

Lab 10

Introduction to Template Function & Template Class

Objectives

The objective of this lab is to familiarize students with template functions and template classes. By the end of this lab, students will be able to understand the concepts of template and generic functions along with template classes using examples.

Tools Required

DevC++ IDE / Visual Studio / Visual Code

Course Coordinator –

Course Instructor –

Lab Instructor –

Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

Templates in C++

Templates are the foundation of generic programming, which involves writing code in a way that is independent of any particular type. Templates are powerful features of C++ which allows you to write generic programs.

A template is a *blueprint* or *formula* for creating a generic class or a function. In simple terms, you can create a single function or a class to work with different data types using templates.

Templates are often used in larger codebase for the purpose of code reusability and flexibility of the programs.

Difference between function overloading and templates

Both function overloading and templates are examples of polymorphism feature of OOP. Function overloading is used when multiple functions do similar operations, templates are used when multiple functions do identical operations. You can use overloading when you want to apply different operations depending on the type. Templates provide an advantage when you want to perform the same action on types that can be different.

How templates work?

Templates are expanded at compile time. This is like macros except that the compiler does type checking before template expansion. The idea is simple, source code contains only function/class, but compiled code may contain multiple copies of the same function/class.

The concept of templates can be used in two different ways:

- Function Templates
- Class Templates

Function Templates

A function template works in a similar to a normal function, with one key difference.

A single function template can work with different data types at once but a single normal function can only work with one set of data types.

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

Defining a Function Template

A function template starts with the keyword `template` followed by template parameter/s inside `< >` which is followed by function declaration.

```
template <class T>
T someFunction(T arg)

{
    ... ..
}
```

In the above code, **T** is a template argument also called *place holder* that accepts different data types (int, float), and `class` is a keyword.

You can also use keyword *typename* instead of `class` in the above example.

When, an argument of a data type is passed to `someFunction()`, compiler generates a new version of `someFunction()` for the given data type.

Calling a Function Template

We can call the template function `someFunction()` a couple of ways. Firstly, we can call it by explicitly specifying the type like

```
int myint = 5;
someFunction <int>(myint);
//Explicit type parametrizing
double mydouble = 99.9;
someFunction <double>(mydouble);
```

However with template function the compiler can perform type deduction to determine the parametrizing types when we don't provide them, hence we can also call `someFunction()` like

```
int myint = 5;
someFunction <>(myint);
//Implicit type parametrizing
double mydouble = 99.9;
someFunction (mydouble);
```

The first call with empty angle brackets tells the compiler that we are calling a template function and the second call leaves it up to the compiler to infer. The problem is with the second call you cannot have any other functions by the same name as function templates cannot be overloaded.

```
// overloaded functions
#include<iostream>
using namespace std;

int Max (int a, int b)
{
    return a < b ? b : a;
}

double Max (double a, double b)
{
    return a < b ? b : a;
}

int main()
{
    cout << sum (10, 20) << endl;
    cout << sum (1.0, 1.5) << endl;
    return 0;
}
```


Example 1: Template Functions

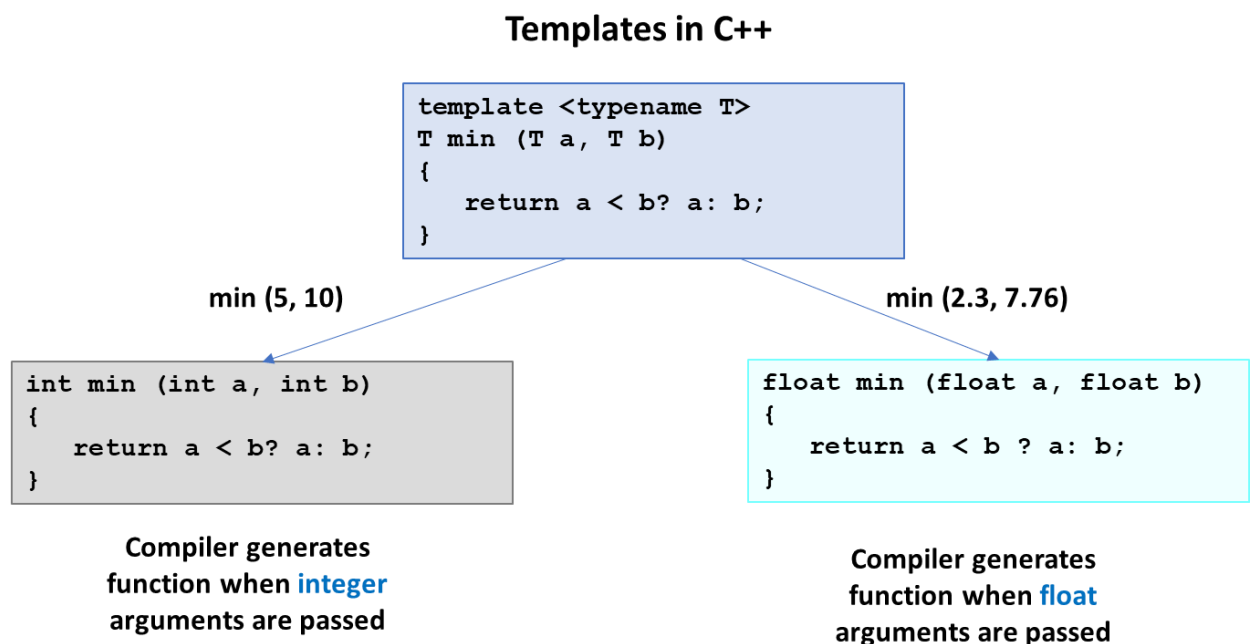
This code can be reduced by templates.

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

template <typename T1>
T1 Max (T1 a, T1 b){
    return a < b? b : a;
}

int main()
{
    cout << "Max integer are:" << Max(22, 2) << endl;
    cout << "Max float are:" << Max (3.9, 22.8) << endl;
    return 0;
}
```

The figure below explains the execution of the above-mentioned program and the concept of how templates work.



Function template with more than one type of parameters

All you need to do is add the extra type to the template prefix, so it looks like this:

```
// 2 type parameters
template<class T1, class T2>
void someFunc(T1 var1, T2 var2 )
{ // some code in here...
}
```

Example 2: Template Functions with multiple type parameters

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

template <typename T1, typename T2>
T Max (T1 a, T2 b){
```

```

        return a < b? b : a;
    }

    int main()
    {
        cout << "Max integer are:" << Max(22, 2.77) << endl;
        cout << "Max float are:" << Max (3.9, 22) << endl;
        return 0;
    }

```

Template Functions with Unused type parameters

If you declare a template parameter, then you ***absolutely must*** use it inside of your function definition otherwise the compiler will complain. So, in the example above, you would have to use both T1 and T2, or you will get a compiler error.

Class Templates

Like function templates, you can also create class templates for generic class operations. Sometimes, you need a class implementation that is same for all classes, only the data types used are different.

Declaring a Class Template

```

template <class T>
class className
{
    ... ..
    public:
        T var;
        T someOperation(T arg);
        ... ..
};

```

In the above declaration, T is the template argument which is a placeholder for the data type used. Inside the class body, a member variable var and a member function someOperation() are both of type T.

Creation of a Class Template Object

To create a class template object, you need to define the data type inside a < > when creation.

```
className<dataType> classObject;
```

For example:

```
className<int> classObject;
```

```
className<float> classObject;
```

```
className<string> classObject;
```

Example 3: Class Template

```

// Class Templates
#include <iostream>
using namespace std;

template <class T>
class mypair {
    T a, b;
    public:

```

```

        mypair (T first, T second)
            {a=first; b=second;}
        T getmax ();
};

template <class T>
T mypair<T>::getmax () {
    T retval;
    retval = a < b? b : a;
    return retval;
}

int main()
{
    mypair <int> myobject (100,75);
    cout << myobject.getmax();
    return 0;
}

```

Multiple Argument Class Templates

Like normal parameters, we can pass more than one data type as arguments to templates. The following example demonstrates the same.

Example 4: Multiple Argument Template

```

#include<iostream>
#include<conio.h>
using namespace std;

template <class t1, t2>
class sample {
    t1 a; t2 b;
public:
    void getdata(){
        cout<< "Enter value for a and b" << endl;
        cin >> a >> b;
    }
    void display(){
        cout << "Value of a:" << a << endl;
        cout << "Value of b:" << b << endl;
    }
};

int main()
{
    sample <int, int> s1;
    sample <float, float> s2;
    cout << "Two integer data" << endl;
    s1. getdata();
    s1. display();
    cout << "two float data" << endl;
    s2. getdata();
    s2. display();
    return 0;
}

```

Specializing templates

Normally when we write a template class or function, we want to use it with many different types, however sometimes we want to code a function or class to make use of a particular type more efficiently. This is when we use a template specialization.

To declare a template specialization we still use the template keyword and angle brackets `<>` but leave out the parameters as :

```
template <>
```

Hence, we can create a function called `printFunction` which prints out its type and value as shown in code below.

```
template<typename T>
void printFunction(T arg) {
    cout<<"printFunction arg is type" << typeid(arg).name() <<"
    with value " << arg << endl;
}
```

Then we can specialize this for integer values as shown in the below mentioned code

```
template<>
void printFunction(int intarg) {
    cout << " printFunction specialization with int arg only
    called with type " <<typeid(intarg).name()<< "with value
    " << intarg << endl;
}
```

And we can do the same thing with classes. The syntax used in the class template specialization is mentioned below:

```
template <> class mycontainer <char> { ... };
```

First of all, notice that we precede the class template name with an empty template `<>` parameter list. This is to explicitly declare it as a template specialization.

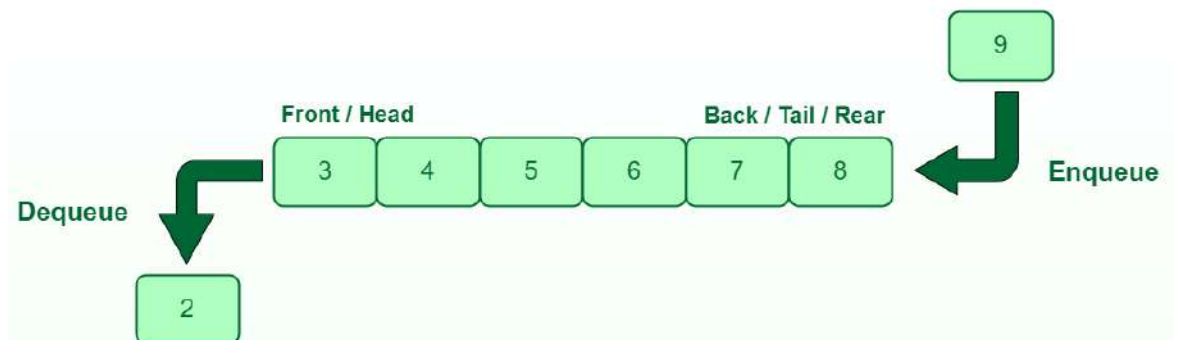
But more important than this prefix, is the `<char>` specialization parameter after the class template name. This specialization parameter itself identifies the type for which we are going to declare a template class specialization (`char`). Notice the differences between the generic class template and the specialization:

```
template <class T> class mycontainer { ... };
template <> class mvcontainer <char> { ... };
```

The first line is the generic template, and the second one is the specialization. When we declare specializations for a template class, we must also define all its members, even those exactly equal to the generic template class, because there is no "inheritance" of members from the generic template to the specialization.

Exercise

1. Create a C++ Program to add, subtract, multiply and divide two numbers using class template. Two numbers can be of the same datatype or combination of different data types.
2. Create a C++ Program to swap the data using template function, instead of calling a function by passing a value, use call by reference, Two numbers can be of same datatype or combination of different data types.
3. Create a C++ class called mycontainer that can store one element of any type and it has just one member function called increase, which increases its value. But we find that when it stores an element of type char it would be more convenient to have a completely different implementation with a function member uppercase, declare a class template specialization for that type.
4. Create an abstract class template for 1D dynamic array. Inherit a dynamic Queue template class from the abstract class which must have following methods: isFull(), isEmpty(), size(), Front(), Rear(), enqueue(), dequeue(), resize().



5. A print shop receives print job requests from various clients throughout the day. The print shop wants to efficiently manage the print jobs and ensure they are printed in the correct order.
 - Create an object of class queue that you have just created to represent the printer job queue.
 - Whenever a print job request arrives, enqueue (add) the job to the back of the queue.
 - The print shop has a printer that can handle one job at a time. If the printer is idle and there are jobs in the queue, dequeue (remove) the job from the front of the queue and start printing it.
 - Once a print job is completed, check if there are more jobs in the queue. If there are, dequeue the next job and start printing it. If the queue is empty, the printer remains idle.
 - Repeat steps 3 and 4 until all print jobs have been completed.

This scenario can be solved using a queue because it follows the First-In-First-Out (FIFO) principle. The print jobs are handled in the same order they arrived at the print shop. The queue ensures that jobs are printed in the correct sequence, without skipping or rearranging the order in which they were received.

Object Oriented Programming (CT-260)

Lab 11

Introduction to Exception Handling, C++ Standard Template Library (STL) and Vectors.

Objectives

The objective of this lab is to familiarize students C++ Introduction to Exception Handling, C++ Standard Template Library (STL) and Vectors. By the end of this lab, students will be able to understand the concepts of Exception Handling along with use of STL, vector, set, and map classes using examples.

Tools Required

DevC++ IDE / Visual Studio / Visual Code

Course Coordinator –

Course Instructor –

Lab Instructor –

Prepared By Department of Computer Science and Information Technology

NED University of Engineering and Technology

Exception Handling in C++

The error handling mechanism of C++ is generally referred to as exception handling. C++ provides a mechanism of handling runtime errors in a program. Generally, exceptions are classified into synchronous and asynchronous exceptions.

Synchronous Exceptions: The exceptions which occur during the program execution (runtime) due to some fault in the input data or technique that is not suitable to handle the current class of data, within the program are known as synchronous exceptions. For example: errors such as out of range, division by zero, overflow, underflow and so on belong to the class of synchronous exceptions.

Asynchronous Exceptions: The exceptions caused by events or faults unrelated (external) to the program and beyond the control of the program are called asynchronous exceptions. For example: errors such as keyboard interrupts, hardware malfunctions, disk failure and so on belong to the class of asynchronous exceptions.

The exception handling mechanism of C++ is designed to handle only synchronous exceptions within a program. This is done by throwing an exception. The exception handling mechanism uses three blocks: **try**, **throw** and **catch**. The try-block must be followed immediately by a *handler*, which is a catch block. If an exception is thrown in the try block, the program control is transferred to the appropriate exception handler. The program should attempt to catch any exception that is thrown by any function. Failure to do so may result in abnormal program termination.

Why Exception Handling?

The following are the main advantages of exception handling over traditional error handling:

1) **Separation of Error Handling code from Normal Code:** In traditional error handling codes, there are always if-else conditions to handle errors. These conditions and the code to handle errors get mixed up with the normal flow. This makes the code less readable and maintainable. With try/catch blocks, the code for error handling becomes separate from the normal flow.

2) **Functions/Methods can handle only the exceptions they choose:** A function can throw many exceptions but may choose to handle some of them. The other exceptions, which are thrown but not caught, can be handled by the caller. If the caller chooses not to catch them, then the exceptions are handled by the caller of the caller. In C++, a function can specify the exceptions that it throws using the throw keyword. The caller of this function must handle the exception in some way (either by specifying it again or catching it).

3) **Grouping of Error Types:** In C++, both basic types and objects can be thrown as exceptions. We can create a hierarchy of exception objects, group exceptions in namespaces or classes and categorize them according to their types.

Exception Handling Constructs:

1. **throw:** The keyword throw is used to raise an exception when an error is generated in the computation.
2. **catch:** The exception handler is indicated by the catch keyword. It must be used immediately after the statements marked by the try keyword.
3. **try:** The try keyword defines the boundary within which an exception can occur.

Thus, the error handling code must perform the following tasks:

1. Detect the problem causing exception. (Will hit the exception).
2. Inform that an error has occurred. (Throw the exception).
3. Receive the error information (Catch the exception).
4. Take corrective actions. (Handle the exception).

The basic syntax for exception handling in C++ is given below:

```
try {  
  
    // code that may raise an exception  
    throw argument;  
}  
  
catch (exception) {  
    // code to handle exception  
}
```

The following is a simple example to show exception handling in C++.

```
#include <iostream>  
using namespace std;  
  
int main() {  
  
    double numerator, denominator, answer;  
  
    cout << "Enter numerator: ";  
    cin >> numerator;  
  
    cout << "Enter denominator: ";  
    cin >> denominator;  
  
    try {  
  
        // throw an exception if denominator is 0  
        if (denominator == 0)  
            throw denominator;  
  
        // not executed if denominator is 0  
        answer = numerator / denominator;  
        cout<<numerator<<"/"<<denominator<<"="<<divide<<endl;  
    }  
  
    catch (int num_exception) {  
        cout<<"Error: Cannot divide by"<<num_exception<<endl;  
    }  
  
    return 0;  
}
```

The above program divides two numbers and displays the result. But an exception occurs if the denominator is 0. To handle the exception, we have put the code `answer = numerator / denominator;` inside the try block. Now, when an exception occurs, the rest of the code inside the try block is skipped.

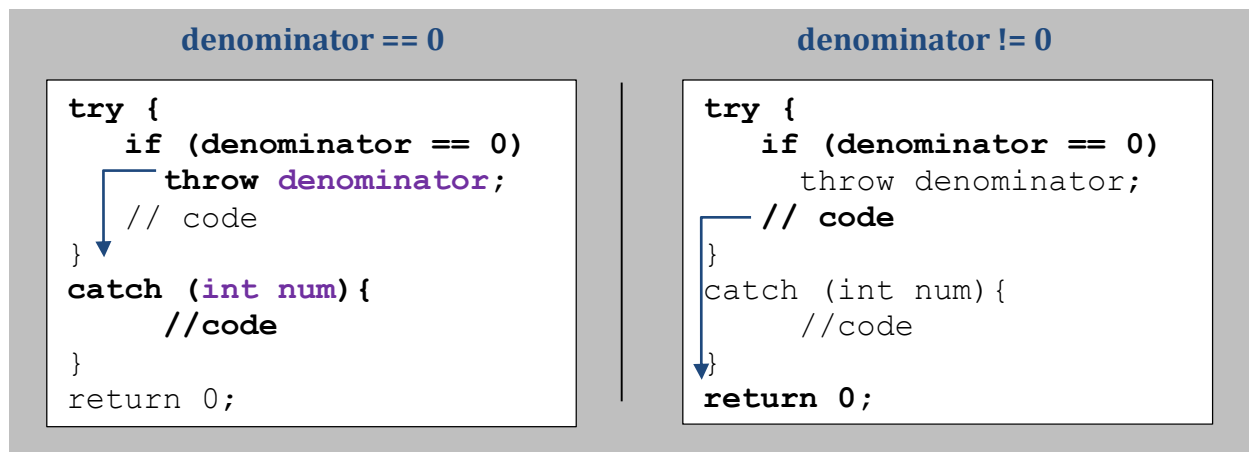
Output 1:

```
Enter numerator: 72
Enter denominator: 0
Error: Cannot divide by 0
```

Output 2:

```
Enter numerator: 72
Enter denominator: 3
72 / 3 = 24
```

The catch block catches the thrown exception and executes the statements inside it. If none of the statements in the try block generates an exception, the catch block is skipped.



Catching All Types of Exceptions

In exception handling, it is important that we know the types of exceptions that can occur due to the code in our try statement so that we can use the appropriate catch parameters. Otherwise, the try...catch statements might not work properly.

If we do not know the types of exceptions that can occur in our try block, there is a special catch block called the 'catch all' block, written as `catch(...)`, that can be used to catch all types of exceptions. For example, in the following program, an `int` is thrown as an exception, but there is no catch block for `int`, so the `catch(...)` block will be executed.

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

int main()
{
    try {
        throw 10;
    }
    catch (char excp) {
        cout << "Caught " << excp;
    }
    catch (...) {
        cout << "Default Exception\n";
    }
}
```

```

    }
    return 0;
}

```

Since there is no exception for an int in the code, the default catch will be executed.

Output

Default Exception

Normally, if you need to perform identical operations on two or more types of data, you use function overloading to create two functions with the required function declaration.

C++ Multiple catch Statements

In C++, we can use multiple catch statements for different kinds of exceptions that can result from a single block of code.

```

try {
    // code
}
catch (exception1) {
    // code
}
catch (exception2) {
    // code
}
catch (...) {
    // code
}

```

The following program divides two numbers and stores the result in an array element. There are two possible exceptions that can occur in this program:

- If the index is out of bounds (index is greater than the size of the array)
- If a number is divided by 0

These exceptions are caught in multiple catch statements.

```

#include <iostream>
using namespace std;

int main() {

    double numerator, denominator, arr[4] = {0.0, 0.0, 0.0, 0.0};
    int index;

    cout << "Enter array index: ";
    cin >> index;

    try {
        // throw exception if array out of bounds
        if (index >= 4)
            throw "Error: Array out of bounds!";

        // not executed if array is out of bounds
        cout << "Enter numerator: ";
        cin >> numerator;
    }
}

```

```

        cout << "Enter denominator: ";
        cin >> denominator;

        // throw exception if denominator is 0
        if (denominator == 0)
            throw denominator;

        // not executed if denominator is 0
        arr[index] = numerator / denominator;
        cout << arr[index] << endl;
    }

    // catch "Array out of bounds" exception
    catch (const char* msg) {
        cout << msg << endl;
    }

    // catch "Divide by 0" exception
    catch (int num) {
        cout << "Error: Cannot divide by " << num << endl;
    }

    // catch any other exception
    catch (...) {
        cout << "Unexpected exception!" << endl;
    }

    return 0;
}

```

Considering the following two cases, the outputs are explained in the text below.

Output 1:

```

Enter numerator: 72
Enter denominator: 0
Error: Cannot divide by 0

```

Output 2:

```

Enter numerator: 72
Enter denominator: 3
72 / 3 = 24

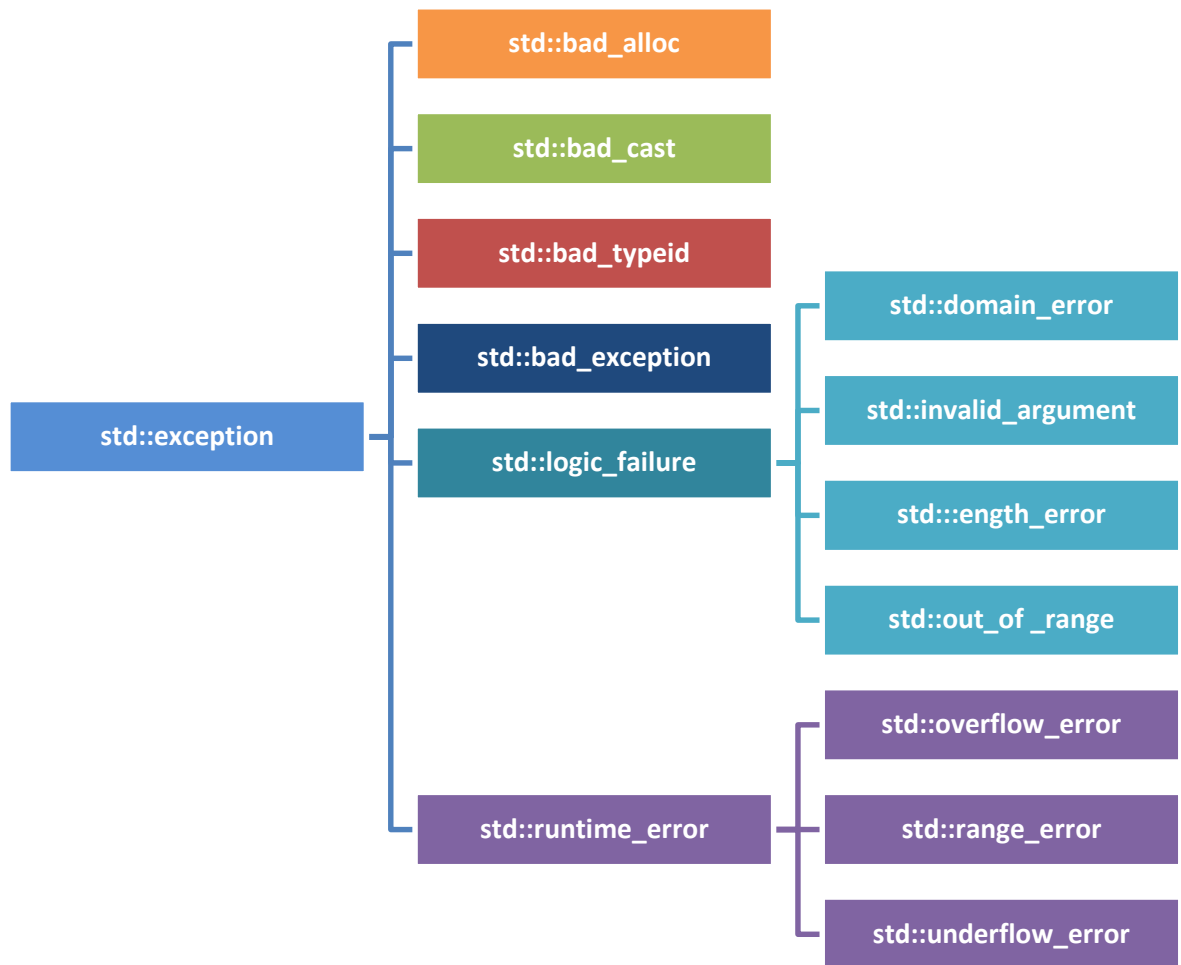
```

As the array `arr` only has 4 elements, the index cannot be greater than 3. In Output 1, index is 5. So we throw a string literal "Error: Array out of bounds!". This exception is caught by the first catch block. Notice the catch parameter `const char* msg`. This indicates that the catch statement takes a string literal as an argument.

In Output 2, the denominator is 0. So we throw the `int` denominator. This exception is caught by the second catch block. If any other exception occurs, it is caught by the default catch block.

C++ Standard Exception

In C++, the `<stdexcept>` header defines a set of standard exception types that are commonly used to handle errors and exceptional situations. Some of the standard exceptions in C++ are listed below:



These are some of the standard exceptions provided by C++. They help categorize and handle different types of errors and provide meaningful information about exceptional situations. They can be used by including the `<stdexcept>` header as shown in the example below.

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

void divide(int numerator, int denominator) {
try{
    if (denominator == 0) {
        throw runtime_error("Division by zero error");
    }
    cout<<"The result is : "<<float(numerator)/denominator;
}

catch (const exception& ex) {
    cout<<"Exception caught: "<<ex.what()<<endl;
}

}

int main() {
    divide(10, 0);
    divide(10, 20);
    return 0;
}
```

Define New Exceptions

You can define your own exceptions standalone or by inheriting and overriding exception class functionality. Following is the example, which shows how you can define your own exception class.

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

class MyException {
public:
    const char * what()
    {
        return "Attempted to divide by zero!\n";
    }
};

int main() {
    try{
        int x, y;
        cout << "Enter the two numbers : \n";
        cin >> x >> y;
        if (y == 0) {
            MyException z;
            throw z;
        }
        else {
            cout << "x / y = " << x/y << endl;
        }
    }
    catch(MyException& e) {
        cout << e.what();
    }
}
```

This code will produce the following output.

Output:

```
Attempted to divide by 0!
```

Here, what() is a public method provided by MyException class. This returns the cause of an exception.

The C++ Standard Template Library (STL)

The Standard Template Library (STL) is a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators. It is a generalized library and so, its components are parameterized. Working knowledge of template classes is a prerequisite for working with STL.

The C++ Standard Template Library (STL) is a collection of algorithms, data structures, and other components that can be used to simplify the development of C++ programs. The STL provides a range of containers, such as vectors, lists, and maps, as well as algorithms for searching, sorting and manipulating data. One of the key benefits of the STL is that it provides a way to write generic, reusable code that can be applied to different data types. This means that you can write an algorithm once, and then use it with different types of

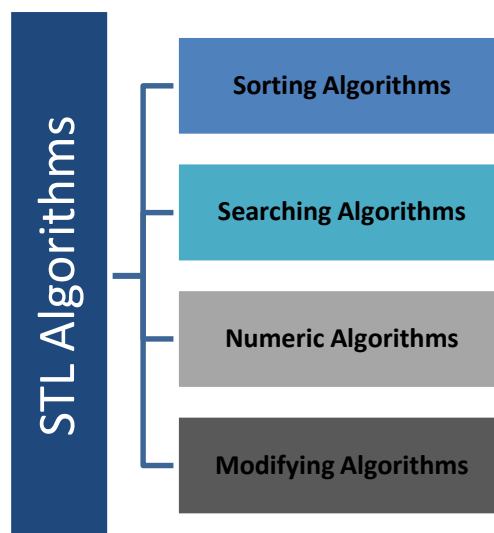
data without having to write separate code for each type. The STL also provides a way to write efficient code. Many of the algorithms and data structures in the STL are implemented using optimized algorithms, which can result in faster execution times compared to custom code. C++ STL has 3 major components:

1. Algorithms
2. Containers
3. Iterators

In addition to these, STL also provides several other features, including function objects, smart pointers, and exception handling mechanisms.

C++ STL Algorithms

In C++, the Standard Template Library (STL) provides a rich set of algorithms that operate on containers or sequences of elements. These algorithms are generic and can be used with various container types. They offer a wide range of functionality, including sorting, searching, transforming, and modifying elements. Some of the most commonly used STL algorithms in C++ are shown in the following diagram.



These are just a few examples of the many algorithms available in the STL. These algorithms are designed to work efficiently with different container types, providing a powerful toolset for manipulating and processing data. To use these algorithms, include the `<algorithm>` header and make use of the appropriate algorithm by providing the necessary arguments and range iterators.

C++ STL Containers

The C++ Standard Library (STL) provides a rich set of container classes that offer different data structures for storing and manipulating collections of objects. Here are some commonly used STL containers in C++.

a. Sequence Containers:

In C++, the Standard Template Library (STL) provides several sequential containers that store and manage elements in a sequential manner. Some examples include vector (dynamic array class), linkedlist, stack, queues, etc. These sequential containers differ in their underlying data structures, performance characteristics, and supported operations. Depending on your requirements, you can choose the appropriate sequential container that suits your needs for element access, insertion, deletion, or traversal.

b. Associative containers:

In C++, the Standard Template Library (STL) provides several associative containers that store and manage elements in an associative manner, allowing efficient lookup and

retrieval based on a key. The associative containers in C++ are set, multiset, map, and multimap. These associative containers offer efficient lookup and retrieval operations based on keys. They provide different performance characteristics and trade-offs based on the underlying data structures used, such as binary search trees for ordered containers or hash tables for unordered containers. You can choose the appropriate associative container based on your requirements for fast access, unique or duplicate keys, or sorted order.

c. Unordered Associative Containers in C++

In C++, the Standard Template Library (STL) provides unordered associative containers that store and manage elements using hash-based data structures for efficient lookup and retrieval. These containers are implemented as hash tables and provide fast average-case performance for insertion, deletion, and search operations. The examples include unordered multiset, unordered multimap, etc. These unordered associative containers provide constant-time average-case complexity for insertion, deletion, and search operations, making them suitable for scenarios where fast lookup based on a key is required. The actual performance may vary depending on the quality of the hash function and the number of elements in the container.

C++ STL ITERATOR

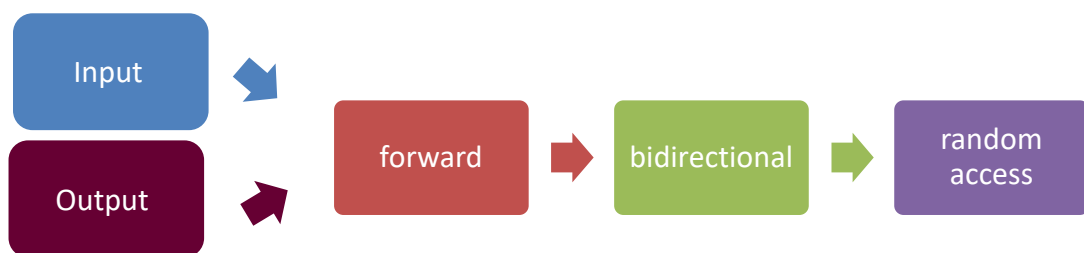
In C++, iterators are used to traverse and access elements in various data structures, including containers and sequences. They provide a uniform interface to work with different types of data structures, allowing you to iterate over elements, access values, and perform operations. Iterators are pointer-like entities used to access the individual elements in a container. Iterators are moved sequentially from one element to another element. This process is known as iterating through a container. Iterator contains mainly two functions:

begin(): The member function begin() returns an iterator to the first element of the vector.

end(): The member function end() returns an iterator to the past-the-last element of a container.

Iterator Categories

Iterators are mainly divided into five categories:



The C++ Standard Template Library (STL) provides different types of iterators with varying levels of functionality and capabilities. STL algorithms and containers often use iterators as arguments to perform operations on the underlying elements. For example, algorithms like `std::sort` and `std::find` accept iterators to specify the range of elements to operate on. Containers like `std::vector` and `std::list` provide member functions to obtain iterators for iterating over their elements.

Vectors

In C++, vectors are like dynamic arrays which are used to store elements of similar data types. However, unlike arrays, the size of a vector can grow dynamically. That is, we can change the size of the vector during the execution of a program as per our requirements.

The Standard Template Library (STL) provides a container called `std::vector` that represents a dynamic array. It is one of the most commonly used containers in C++ due to

its flexibility, efficiency, and ease of use. To use vectors, we need to include the `<vector>` header file in our program.

C++ Vector Declaration

Once we include the header file, here's how we can declare a vector in C++:

```
vector<T> vector name;
```

The type parameter `<T>` specifies the type of the vector. It can be any primitive data type such as `int`, `char`, `float`, etc.

```
vector<int> num;
```

Notice that we have not specified the size of the vector named `num` during the declaration. This is because the size of a vector can grow dynamically so it is not necessary to define it.

C++ Vector Initialization

```
vector<int> vector1 = {1, 2, 3, 4, 5};
```

or

```
vector<int> vector2 {1, 2, 3, 4, 5};
```

or

```
vector<int> vector3(5, 12); //initialized with size 5 and value 12.
```

Add Elements to a Vector

To add a single element into a vector, we use the `push_back()` function. It inserts an element into the end of the vector. For example,

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

int main() {
    vector<int> num {1, 2, 3, 4, 5}; //this may raise error

    cout << "Initial Vector: ";

    for (const int& i : num) { //this may raise error
        cout << i << " ";
    }

    // add the integers 6 and 7 to the vector
    num.push_back(6);
    num.push_back(7);

    cout << "\nUpdated Vector: ";

    for (const int& i : num) { //this may raise error
        cout << i << " ";
    }

    return 0;
}
```

Here, we have initialized an int vector num with the elements {1, 2, 3, 4, 5} and the push_back() function adds elements 6 and 7 to the vector.

Output:

```
Initial Vector: 1  2  3  4  5
Updated Vector: 1  2  3  4  5  6  7
```

Access Elements of a Vector

In C++, we use the index number to access the vector elements. Here, we use the at() function to access the element from the specified index. For example,

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

int main() {
    vector<int> num {1, 2, 3, 4, 5}; // this may raise error

    cout << "Element at Index 0: " << num.at(0) << endl;
    cout << "Element at Index 2: " << num.at(2) << endl;
    cout << "Element at Index 4: " << num.at(4);

    return 0;
}
```

Output:

```
Element at Index 0: 1
Element at Index 2: 3
```

Access Elements of a Vector

We can change an element of the vector using the same at() function. For example,

```

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

int main() {
    vector<int> num {1, 2, 3, 4, 5}; //this may raise error

    cout << "Initial Vector: ";

    for (const int& i : num) { //this may raise error
        cout << i << " ";
    }

    // change elements at indexes 1 and 4
    num.at(1) = 9;
    num.at(4) = 7;

    cout << "\nUpdated Vector: ";

    for (const int& i : num) { // this may raise error
        cout << i << " ";
    }

    return 0;
}

```

Output:

```

Initial Vector: 1  2  3  4  5
Updated Vector: 1  9  3  4  7

```

Delete Elements of a Vector

To delete a single element from a vector, we use the `pop_back()` function. For example,

```

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

int main() {
    vector<int> prime_numbers{2, 3, 5, 7}; //this may raise error

    // initial vector
    cout << "Initial Vector: ";
    for (int i : prime_numbers) { //this may raise error
        cout << i << " ";
    }

    // remove the last element
    prime_numbers.pop_back();

    // final vector
    cout << "\nUpdated Vector: ";
    for (int i : prime_numbers) { //this may raise error
        cout << i << " ";
    }

    return 0;
}

```

Output:

```

Initial Vector: 2 3 5 7
Updated Vector: 2 3 5

```

C++ Vector Functions

In C++, the vector header file provides various functions that can be used to perform different operations on a vector.

Function	Description
size()	returns the number of elements present in the vector
clear()	removes all the elements of the vector
front()	returns the first element of the vector
back()	returns the last element of the vector
empty()	returns 1 (true) if the vector is empty
capacity()	check the overall size of a vector

We can declare an iterator for each container in the C++ Standard Template Library. For example,

```
vector<int>::iterator it;
```

We often use iterator member functions like `begin()`, `end()`, etc. to return iterators that point to container elements. For example,

```

vector<int> numbers = {3, 2, 5, 1, 4};
vector<int>::iterator itr1 = numbers.begin();
vector<int>::iterator itr2 = numbers.end();

```

```

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

int main() {
    // initialize vector of int type
    vector<int> numbers {1, 2, 3, 4, 5}; //this may raise error

    // initialize vector iterator to point to the first element
    vector<int>::iterator itr = numbers.begin();
    cout << "First Element: " << *itr << " " << endl;

    // change iterator to point to the last element
    itr = numbers.end() - 1;
    cout << "Last Element: " << *itr;

    return 0;
}

```

In the above code, the `begin()` and `end()` member functions of the vector container return iterators pointing to the first and one-past-the-last elements, respectively. Here, we have used `numbers.end() - 1` instead of `numbers.end()`. This is because the `end()` function points to the theoretical element that comes after the final element of the container. So, we need to subtract 1 from `numbers.end()` in order to point to the final element. Similarly, using the code `numbers.end() - 2` points to the second-last element, and so on.

The iterators are then used in a for loop to iterate over the elements, and the values are printed to the console. Iterators provide a flexible and generic way to work with elements in containers and sequences, enabling algorithms and operations to be performed uniformly across different data structures. They allow you to access and manipulate data efficiently and provide a powerful toolset for working with collections of elements.

Maps

Maps are associative containers that store elements in a mapped fashion. Each element has a key value and a mapped value. No two mapped values can have the same key values.

```

#include <iostream>
#include <map>
using namespace std;
int main( ){
    // Create a map of strings to integers
    map<string, int> m;

    // Insert some values into the map
    m["one"] = 1;
    m["two"] = 2;
    m["three"] = 3;

    // Get an iterator pointing to the first element in the map
    map<string, int>::iterator it = m.begin();

    // Iterate through the map and print the elements
    while (it != m.end( )){
        cout << "Key: " << it->first << ", Value: " << it->second
<< endl;
        ++it;
    }
    return 0;
}

```

```
}
```

Sets

Sets are a type of associative container in which each element has to be unique because the value of the element identifies it. The values are stored in a specific sorted order i.e. either ascending or descending. The set class is the part of C++ Standard Template Library (STL) and it is defined inside the <set> header file.

Syntax:

```
set <data_type> set_name;
```

Example

```
#include <iostream>
#include <set>
using namespace std;
int main(){
    set<char> a;
    set<char>::iterator p;
    a.insert('G');
    a.insert('F');
    a.insert('G');

    p=a.begin();
    while(p!=a.end()) {
        cout << *p << ' ';
        p++;
    }
    cout << '\n';
    return 0;
}
```

Exercise

1. Create a C++ Program to implement login. It should accept user name and password and throw a custom exception if the password has less than 6 characters or does not contain a digit.
2. Create a class for dynamic stack using vectors. Implement the push(), pop() and peek() functions. The object of stack class will be used to take any sentence as input and it should have a method reverse() which will reverse each word in the string.
3. You are given N integers. Store N integers in a vector and write a function Sort for sorting the N integers and print the sorted order. Now use the sorting algorithms provided in STL for sorting the vector. Measure the time taken by the two methods for sorting the vector and print the results. [Hint: You can use built-in function time() or anyother built-in method for measuring the processing time of sorting operation.]
4. You are a teacher and want to keep track of your students' grades. You decide to use the map container in C++ to store the student names as keys and their corresponding grades as values. Design and implement a program in C++ to fulfill the following requirements:
 - Input: Prompt the user to enter the names and grades of multiple students. Allow the user to continue entering data until they choose to stop.
 - Storage: Store the student names as keys and their grades as values in a map container.
 - Retrieval: Implement a function to retrieve the grade associated with a given student's name. If the student is not found in the map, display an appropriate message.
 - Update: Implement a function to update the grade of a specific student. Prompt the user to enter the new grade and update the corresponding value in the map.
 - Deletion: Implement a function to remove a student and their grade from the map based on a

given student's name.

- Display: Display all the student names and their corresponding grades stored in the map.

Testing: Write a program to test the functionality of your map implementation. Add multiple students with their grades, retrieve grades for specific students, update grades, delete students, and display the final list of students and their grades. As you implement the program, consider using appropriate data types, input validation, error handling, and clear user prompts. Remember to thoroughly test your implementation to ensure correctness and accuracy in storing and retrieving the student grades.

5. You are hosting a party and want to keep track of the unique guests attending. To achieve this, you decide to use the set container in C++ to store the names of the guests.

Design and implement a program in C++ to fulfill the following requirements:

- Input: Prompt the user to enter the names of multiple guests attending the party. Allow the user to continue entering names until they choose to stop.
- Storage: Store the guest names in a set container, which will automatically enforce uniqueness by discarding duplicate names.
- Display: Display all the unique guest names stored in the set in alphabetical order.
- Count: Display the total number of unique guests attending the party.

Testing: Write a program to test the functionality of your set implementation. Add multiple guest names, including duplicates, and ensure that only unique names are stored in the set. Display the final list of unique guest names and the total count of guests attending the party. As you implement the program, consider using appropriate data types, input validation, error handling, and clear user prompts.

Object Oriented Programming (CT-260)

Lab 12

Introduction to Filing

Objectives

The objective of this lab is to familiarize students with filing operations in C++. By the end of this lab, students will be able to understand the concepts of opening a file, writing data to a file, and reading data from a file along with closing a file .

Tools Required

DevC++ IDE / Visual Studio / Visual Code

Course Coordinator –

Course Instructor –

Lab Instructor –

Prepared By Department of Computer Science and Information Technology
NED University of Engineering and Technology

File Handling in C++

File handling in C++ is a mechanism to store the output of a program in a file and help perform various operations on it. Files help store these data permanently on a storage device.

For achieving file handling, we need to follow the following steps:

STEP 1-Creating a file

STEP 2-Opening a file

STEP 3-Writing data into the file

STEP 4-Reading data from the file

STEP 5-Closing a file.

Streams in C++

We give input to the executing program and the execution program gives back the output. The sequence of bytes given as input to the executing program and the sequence of bytes that comes as output from the executing program are called stream. In other words, streams are nothing but the flow of data in a sequence.

The input and output operation between the executing program and the devices like keyboard and monitor are known as “console I/O operation”. The input and output operation between the executing program and files are known as “disk I/O operation”.

In **input operations**, the bytes flow from a device (e.g., a keyboard, a disk drive, a network connection, etc.) to main memory.

In **output operations**, bytes flow from main memory to a device (e.g., a display screen, a printer, a disk drive, a network connection, etc.).

So far, we have been using the `iostream` standard library, which provides ***cin*** and ***cout*** methods for reading from standard input and writing to standard output respectively. C++ provides you with a library that comes with methods for file handling.

```
#include <iostream> // contains cin and cout
```

Stream Input/Output Classes and Objects

The I/O system of C++ contains a set of classes which define the file handling methods. These include `ifstream`, `ofstream` and `fstream` classes. These classes are derived from `fstream` and from the corresponding `iostream` class. These classes, designed to manage the disk files, are declared in `fstream` and therefore we must include this file in any program that uses files.

fstream library:

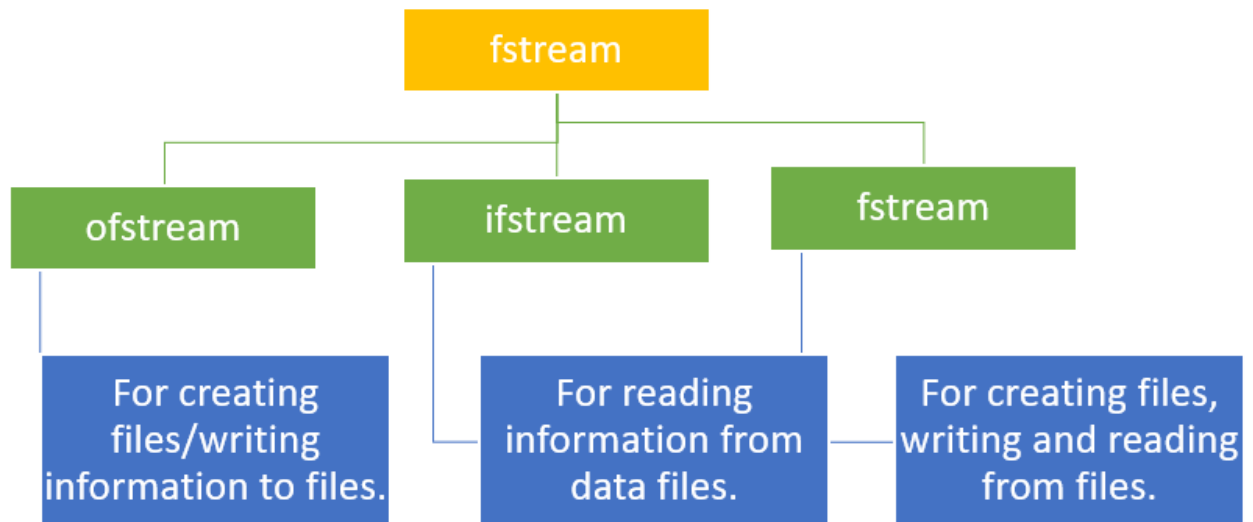
Three classes known as `ofstream`, `ifstream`, and `fstream` are utilised by the C++ `fstream` library to manage files.

1. **ofstream:** This class assists in creating and adding data to the file that is created using the program's output. The output stream is another name for it.
2. **ifstream:** This class, sometimes referred to as the input stream, is used to read data from files.
3. **fstream:** `ofstream` and `ifstream` are combined into one class in this case. It offers the ability to write, read, and create files.

You must include the `fstream` as a header file, similar to how we define `iostream` in the header, in order to use the following classes.

```
#include<fstream>
```

Stream-I/O template hierarchy



C++ File Operations

For managing files, C++ has four main operations.

- ✓ **open()**: Create a file using the open() method.
- ✓ **read()**: The data from the file is read using the read() function.
- ✓ **write()**: Write new data to a file using the write() function.
- ✓ **close()**: The file is closed using the close() method.

1. Opening a File

We must first open a file in order to read or write data to it. We can open file by

1. passing file name in constructor at the time of object creation
2. using the open method

File creation and opening using constructor method

We can open file using a constructor like this:

```

ifstream (const char* filename, ios_base::openmode mode= ios_base::in);
ifstream fin(filename, openmode) by default openmode = ios::in
ifstream fin("filename");
  
```

Example: File creation

```

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

int main(){
    ofstream myfile ("file1");
    if (!myfile){
        cout << "File not created!";
    }
    else {
        cout << "File created successfully!";
        myfile.close();
    }
}
  
```

Output:

```

C:\Users\samia\File.exe
File created successfully!
-----
Process exited after 0.03292 seconds with return value 0
Press any key to continue . . .

```

File open using open() method

The three objects, that is, `fstream`, `ofstream`, and `ifstream`, have the `open()` function defined in them. The function takes this syntax:

```

// Calling of default constructor
ifstream fin;
fin.open(filename, openmode)
fin.open("filename");

```

The filename denotes the name of the file whereas, mode determines different modes to open the file.

Mode	Description
ios::in	File opened in reading mode
ios::out	File opened in writing mode
ios::app	File opened in append mode
ios::ate	File opened in reading mode but read and write performed at the end of the file
ios::binary	File opened in binary mode
ios::trunc	File opened in truncate mode
ios::nocreate	The file opens only if it exists
ios::noreplace	The file opens only if it doesn't exist

These Mode Flags help you open file in a mode of your choice as per the need of the program. If you want to set more than one open mode, just use the OR operator | like:

```
ios::ate | ios::binary
```

Example: Creating a file using open ()

```

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

int main(){
    fstream my_file;
    my_file.open("file2", ios::out);
    if (!my_file){
        cout << "File not created!";
    }
    else {
        cout << "File created successfully!";
        my_file.close();
    }
    return 0;
}

```

Opening a file associates a file stream variable declared in the program with a physical file at the source, such as a disk. In the case of an input file:

- ✓ the file must exist before the open statement executes.
- ✓ If the file does not exist, the open statement fails and the input stream enters the fail state

An output file does not have to exist before it is opened, if the output file does not exist, the computer prepares an empty file for output. If the designated output file already exists, by default, the old contents are erased when the file is opened.

File validation before access

Before accessing a file in C++, it's a good practice to validate its existence and ensure that you have the necessary permissions to access it. We can use the predefined Boolean functions like `.good()` and `.open()` to check if a file is available and the connection is open.

Example: Validating file existence and access

```
#include <iostream>
#include <fstream>
Using namespace std;
bool fileExists(const string& filePath) {
    ifstream file(filePath);
    return file.good();
}

bool canAccessFile(const string& filePath) {
    ifstream file(filePath);
    return file.is_open();
}

int main() {
    string filePath = "path/to/file.txt";

    if (fileExists(filePath)) {
        if (canAccessFile(filePath)) {
            // File exists and can be accessed
            // Proceed with accessing the file
            ifstream file(filePath);
            // Perform necessary operations on the file
            // ...
        } else {
            cout << "Cannot access the file." << endl;
        }
    } else {
        cout << "File does not exist." << endl;
    }

    return 0;
}
```

In the above code, the ***fileExists()*** function checks if the file exists by attempting to open it using an `std::ifstream` and checking if it is in a good state (`good()` function returns true). The ***canAccessFile()*** function checks if the file can be accessed by attempting to open it using an `std::ifstream` and checking if it is successfully opened (`is_open()` function returns true).

The existence of the file is checked using `fileExists`. If the file exists, it proceeds to check if it can be accessed using `canAccessFile`. By validating the file before accessing it, you can handle scenarios where the file doesn't exist or you don't have the required permissions to access it, preventing potential runtime errors. It is a good practice to use exception handling while opening a file. We can replace the if and else blocks by the try and catch blocks for the program to work efficiently.

2. Writing Files

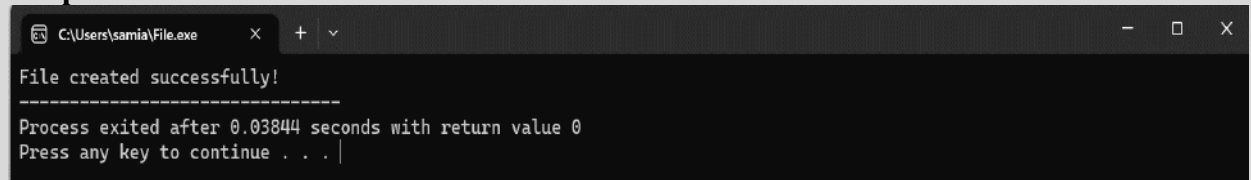
When writing data into a file, the insertion operator (`<<`) and text wrapped in double-quotes are used with a `fstream` or `ofstream` object.

Example: Writing data to a file

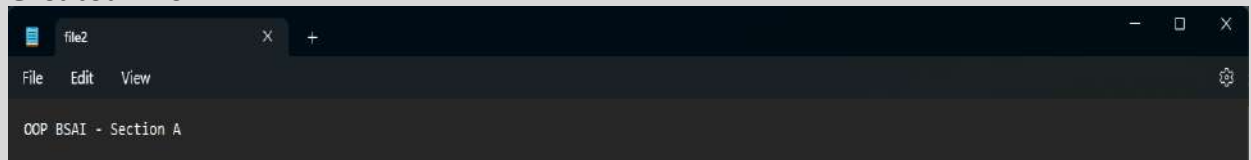
```
#include <iostream>
#include <fstream>
using namespace std;

int main() {
    fstream my_file;
    my_file.open("file2", ios::out);
    if (!my_file) {
        cout << "File not created!";
    }
    else {
        cout << "File created successfully!";
        my_file << "OOP BSAI - Section A";
        my_file.close();
    }
    return 0;
}
```

Output:



Created File:



3. Reading Files

You can read information from files into your C++ program. This is possible using stream extraction operator (`>>`). You use the operator in the same way you use it to read user input from the keyboard. However, instead of using the `cin` object, you use the `ifstream`/ `fstream` object.

Example: Reading data from a file.

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

```
using namespace std;

int main(){
    fstream my_file;
    my_file.open("file2.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;
}
```

Here, after opening the file, we have taken a char variable and in a while loop starting from the beginning of the file till the end of file which is donated by an eof character. The *eof()* will generate true when eof character is reached which will terminate the loop otherwise the character on the file will be transferred to the output buffer.

Using Member Function getline

As its name states, read a whole line, or at least till a delimiter that can be specified

```
istream& getline(istream& is, string& str, char delim);
```

Parameters:

Mode	Description
is	It is an object of istream class and tells the function about the stream from where to read the input from.
str	It is a string object, the input is stored in this object after being read from the stream.
delim	It is the delimitation character which tells the function to stop reading further input after reaching this character.

```
istream& getline(istream& is, string& str);
```

The second declaration is almost the same as that of the first one. The only difference is, the latter has a delimitation character which is by default a new line(\n) character.

Example: Read a Line

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

int main(){
    fstream my_file;
    my_file.open("file2", ios::in);
    string line;
    if (!my_file){
        cout << "No such file";
    }
}
```

```

    }
    else {
        string line;
        while(!my_file.eof()){
            getline (my_file, line);
            cout << line;
        }
        my_file.close();
    }
    return 0;
}

```

Example: Read a character

```

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

int main(){
    ifstream openFile ("file2.txt");
    char ch;
    if (!openFile){
        cout << "No such file";
    }
    else {
        while(!openFile.eof()){
            openFile.get(ch);
            cout << ch;
        }
    }
    openFile.close();
    return 0;
}

```

Writing and Reading data in a file:

User provided data that we normally take input using cin can be written easily to a file and can also be read very easily. The example below writes data in a file and reads the data from file to display it on screen.

Example: Write user's Name and Age in a file

```

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

int main(){
    char data[100];

    //open a file in write mode
    ofstream outfile;
    outfile.open("array.dat");

    cout<< "Writing to the file" << endl;
    cout<< "Enter your name:";
    cin.getline(data, 100);

    //write input name into the file.

```

```

    outfile << data << endl;

    cout << "Enter your age:";
    cin >> data;
    cin.ignore();

    //again write input age into the file.
    outfile << data <<endl;

    //close the opened file
    outfile.close();

    //open file in read mode
    ifstream infile;
    infile.open ("array.dat");

    cout<<endl << "Reading data fom file" << endl;
    infile >> data;

    //write the name at screen
    cout << "Name:" << data << endl;

    //again read the age from the file and display it
    infile >> data;
    cout << "Age: " << data << endl;

    //close the opened file
    infile.close();

    return 0;
}

```

Examples make use of additional functions from cin object, like `getline()` function to read the line from outside and `ignore()` function to ignore the extra characters left by previous read statement

Output:

```

Writing to the file
Enter your name:Ali
Enter your age:20

Reading data from file
Name:Ali
Age: 20

```

Write() function

The `write()` function is used to write object or record (sequence of bytes) to the file. A record may be an array, structure or class. Syntax of `write()` function is:

```

fstream fout;
fout.write((char *) &obj, sizeof(obj));

```

The `write()` function takes two arguments, ***&obj*** which refers to the initial byte of an object

stored in the memory and ***sizeof(obj)*** that provides the size of object represents the total

number of bytes to be written from initial byte.

Example: Write () function

```
#include <iostream>
#include <fstream>
using namespace std;
class student{
    int roll;
    char name[25];
    float marks;
public:
    void getdata(){
        cout<< "Enter roll no:";
        cin >> roll;
        cout << "Enter name:" ;
        cin >> name;
        cout << "Enter marks:";
        cin >> marks;
    }

    void addRecord(){
        fstream f;
        student s;
        f.open("student.dat",ios::app | ios::binary );
        s.getdata();
        f.write((char*)&s, sizeof(s));
        f.close();
    }
};

int main(){
    student s;
    char c = 'n';
    do{
        s.addRecord();
        cout<<"Do you want to add another record.Press Y to continue: ";
        cin >> c;
    } while (c == 'y' || c == 'Y');

    cout << endl << "Data written successfully";
}
```

Output:

```
Enter roll no:45
Enter name:Ali
Enter marks:67
Do you want to add another record. Press Y to continue: y

Enter roll no:98
Enter name:Jehadad
Enter marks:78
Do you want to add another record. Press Y to continue: n

Data written successfully
```

Read() function

The read() function is used to read object (sequence of bytes) to the file. Syntax of read() function is:

```
fstream fin;  
fin.read((char *) &obj, sizeof(obj));
```

The read() function takes two arguments, **&obj** which refers to the initial byte of an object stored in the memory and **sizeof(obj)** that provides the size of object represents the total number of bytes to be written from initial byte. The read() function returns NULL if no data read.

Example: Read () function

```
#include <iostream>  
#include <fstream>  
using namespace std;  
  
class student{  
    int roll;  
    char name[25];  
    float marks;  
public:  
  
    void displayStudent(){  
        cout<<"Roll no: "<<roll <<endl;  
        cout <<"Name: "<< name << endl;  
        cout <<"Marks: "<<marks<<endl;  
    }  
  
    void Readdata()  
    {  
        fstream f;  
        student s;  
        f.open("student.dat",ios::in | ios::binary);  
        if(f.read((char*)&s,sizeof(s))){  
            cout<<endl<<endl;  
            s.displayStudent();  
        }  
        else{  
            cout<<"Error in reading data from file...\n";  
        }  
    }  
};  
  
int main(){  
    student s;  
    s.Readdata();  
}
```

Output:

```
Enter roll no:45  
Enter name:Ali  
Enter marks:67
```

Exercise

1. Write a program to implement I/O operations on characters. I/O operations includes inputting a string, calculating length of the string, Storing the String in a file and fetch the stored characters from it.
2. Write a program to copy the contents of one file to another.
3. Take a class Person having two attributes name and age. Include a parametrized constructor to give values to all data members. In main function Create an instance of the person class and name it person1. Create a binary file person.bin and write person1 object into it. Read the person1 object from the file.
4. Take a class Participant having three attributes (ID, name and score) and following member functions.
 - ✓ Input () function takes data of the object and stores it in a file name participant.dat
 - ✓ Output () function takes id from user and show respective data of that id.
 - ✓ Max () gives the highest score of the Participant in the file.
5. Write a function in C++ to count and display the number of lines not starting with alphabet 'A' present in a text file "STORY.TXT".
Example: If the file "STORY.TXT" contains the following lines,

```
The rose is red.  
A girl is playing there.  
There is a playground.  
An airplane is in the sky.  
Numbers are not allowed in the password.
```

The function should display the output as 3.