Bangladesh Army University of Engineering & Technology (BAUET)

Qadirabad, Natore 6431, Bangladesh

Department of Information and Communication Engineering (ICE)

Object Oriented Programming Language Lab Manual

Course Code	ICE-2232
Course Title	Object Oriented Programming Language Sessional
Credit Hours	1.5

Prepared by	Verified By
1. Nasirul Mumenin Lecturer, ICE.	Dr. Md. Rubel Basar Assistant Professor Dept. of ICE Head of the Department

Table of Content

General Guideline and Safety Instructions

1. Strictly follow the written and verbal instructions given by the teacher /Lab Instructor. If you do not understand the instructions, the handouts and the procedures, ask the instructor or teacher.
2. Students are required to attend all labs with official dress code and wearing ID card.
3. Mobile phones should be switched off in the lab. Keep bags in the bag shelf.
4. Keep the labs clean at all times, no food and drinks allowed inside the lab.
5. Students should work individually/team in the hardware and software task.
6. Students have to bring the lab manual cum lab report file along with them whenever they come for lab work.
7. Should take only the lab manual, calculator (if needed) and a pen or pencil to the work area.
8. Should utilize 3 hour"s time properly to perform the experiment and to record the readings. Do the calculations and take signature from the instructor.
9. If the experiment is not completed in the stipulated time, the pending work has to be carried out in the leisure hours or extended hours.
10. Intentional misconduct will lead to expulsion from the lab.
11. Do not handle any equipment without reading the safety instructions. Read the handout and procedures in the Lab Manual before starting the experiments.

Course Description

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior. OOP focuses on the objects that developers want to manipulate rather than the logic required to manipulate them. This approach to programming is well-suited for programs that are large, complex and actively updated or maintained. This includes programs for manufacturing and design, as well as mobile applications; for example, OOP can be used for manufacturing system simulation software. This understanding is a critical step towards being able to design new programs or use them appropriately as part of a larger engineering system. Hence the course seeks to develop foundational concepts on various key aspects of oop and it's practical implementation to solve problems.

Course Objective:

1. To gather hands-on experience about object-oriented programming.
2. To develop technical hands on how to solve and, implement various problems regarding programming.
3. To mature the skill of using various modern tools such as codeblockss.
4. To translate the theoretical knowledge of object-oriented programming in practical environment.

Statement of Course Outcomes (CO):

Upon completion of all sessional, the students will be able to:

1. Apply information and knowledge of object-oriented programming language to demonstrate and analyze the various practical problems.

2. Design and develop solution for different components of complex engineering problem related to practical programming problems.
3. Analysis and Interpretation of problems and solutions to provide valid conclusion acknowledging the limitations.

Assessment of Course Outcomes (CO):

CO	PO	Bloom's Taxonomy Level	KP	CP	CA	Delivery methods and activities	Assessment Tools
CO1	PO1	C2, C3, C4	KP1			Lecture, Lab Manual, and Demonstration	Lab Quiz, Lab Viva
CO2	PO3	C4	KP5			Lab Manual, Demonstration	Lab Performance/Lab Test
CO3	PO4	C4				Lab Manual, Demonstration	Lab Report, Lab Test

Assessment Criteria and Marks Distribution

Si. No.	Particulars	Marks
1	Lab Performance	10
2	Lab Reports	20
3	Lab Test	40
4	Quiz Test	20
6	Lab Viva	10
Total		100

Summary of Lab Report

Ex. No.	Experiment Title	Date of Exp.	Date of Subm	Marks	Teacher's Signature	Remarks
01	Familiarization with object-oriented programming language.					
02	Familiarization with Basic C++ programs.					
03	Experimentation with the conditional statements in C++.					
04	Experimentation with the iteration in C++.					
05	Implementation of C++ in mathematical problem solving.					
06	Experimentation with non-numeric data types.					
07	Experimentation with pointer and dynamic memory allocation.					
08	Implementation of class declaration, definition, and access modifiers in C++.					
09	Implementation of constructor and destructor in C++.					
10	Experimentation with the inheritance in C++.					
11	Implementation of operator overloading in C++.					
12	Experimentation with function overloading and overriding in C++.					
13	Implementation of exception handling in C++.					
	Average marks in lab Report (out of 20)					

Format of Lab Report

All lab reports should have to follow the common format as below

- Experiment No
- Experiment Title
- Objectives
- Theory Overview (with formula/equations and/or figure if required)
- Circuit diagram (with adequate labeling and figure caption)
- Results
- Discussion
- Conclusion

Instruction for Lab Report Writing

- Lab report must be handwritten without copying from other works.
- Writing should be neat and clean with proper caption and labeling in figure and table.
- The title page of report should contain all the basic information such as experiment no & title, course code & title, student's information, teacher's information, experiment date, submission date.
- Result should include calculated and/or simulated and/or measured data with proper unit.
- Table and/or graph of result should be neat and clear with axis label and units where applicable.
- The discussion should present your findings from the experiment. Evaluate the outcome objectively, taking a candid and unbiased point of view. Suppose that the outcome is not close to what you expected. Even then, after checking your results, give reasons why you believe that outcome is not consistent with the expected.
- In discussion, state the discrepancies between the experimental results and the model (theory), and discuss the sources of the differences in terms of the errors by offering logical inferences and suggest improvements.
- Conclusion should present, a brief summary of what was done, how it was done, show the results and conclusions of the experiment.
- Report should be submitted timely, late submission will cause reduction of marking.
- All lab reports have to be maintained in a single file which has to bring in every laboratory class.

Mini Lab Project

All students have to do a mini lab project under this sessional course. The project will be in different group with the number of group members assigned by course teacher. The project will be accessed by course teacher in three criteria: 1) project show, 2) project presentation, and 3) project report.

1. Familiarization with Object-oriented Programming Language.

Objective

In this experiment, the basics and foundations of object oriented programming will be explored.

Theory Overview

Object-oriented programming (OOP) is a computer programming model that organizes software design around data, or objects, rather than functions and logic. An object can be defined as a data field that has unique attributes and behavior. OOP focuses on the objects that developers want to manipulate rather than the logic required to manipulate them. This approach to programming is well-suited for programs that are large, complex and actively updated or maintained. This includes programs for manufacturing and design, as well as mobile applications; for example, OOP can be used for manufacturing system simulation software. The organization of an object-oriented program also makes the method beneficial to collaborative development, where projects are divided into groups. Additional benefits of OOP include code reusability, scalability, and efficiency. Some examples of object-oriented incudes java, C++, C#, python, ruby, php etc.

The structure, or building blocks, of object-oriented programming include the following:

- **Classes** are user-defined data types that act as the blueprint for individual objects, attributes and methods.
- **Objects** are instances of a class created with specifically defined data. Objects can correspond to real-world objects or an abstract entity. When class is defined initially, the description is the only object that is defined.
- **Methods** are functions that are defined inside a class that describe the behaviors of an object. Each method contained in class definitions starts with a reference to an instance object. Additionally, the subroutines contained in an object are called instance methods. Programmers use methods for reusability or keeping functionality encapsulated inside one object at a time.
- **Attributes** are defined in the class template and represent the state of an object. Objects will have data stored in the attributes field. Class attributes belong to the class itself.

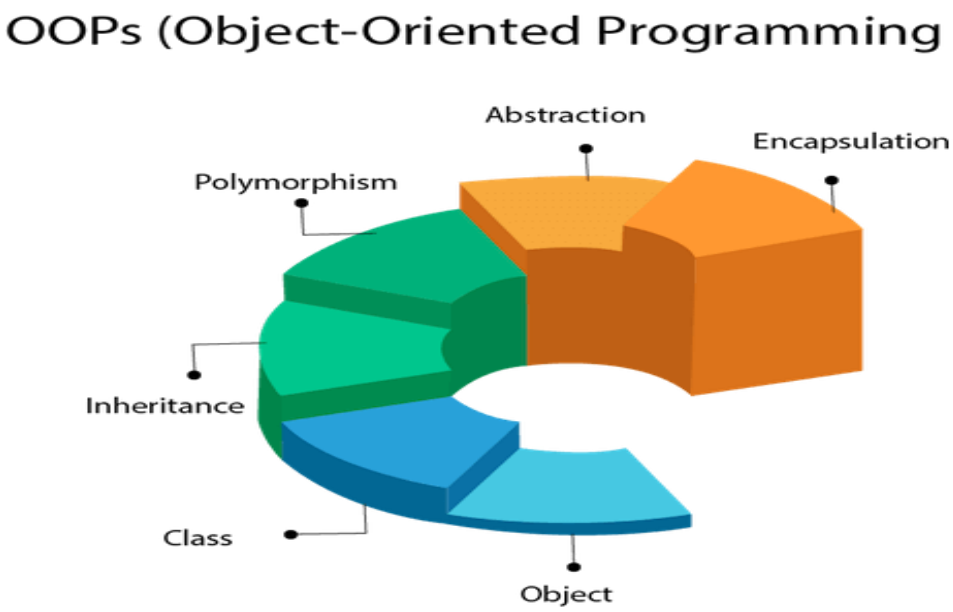
Object-oriented programming is based on the following principles:

Encapsulation. This principle states that all important information is contained inside an object and only select information is exposed. The implementation and state of each object are privately held inside a defined class. Other objects do not have access to this class or the authority to make changes. They are only able to call a list of public functions or methods. This characteristic of data hiding provides greater program security and avoids unintended data corruption.

Abstraction. Objects only reveal internal mechanisms that are relevant for the use of other objects, hiding any unnecessary implementation code. The derived class can have its functionality extended. This concept can help developers more easily make additional changes or additions over time.

Inheritance. Classes can reuse code from other classes. Relationships and subclasses between objects can be assigned, enabling developers to reuse common logic while still maintaining a unique hierarchy. This property of OOP forces a more thorough data analysis, reduces development time and ensures a higher level of accuracy.

Polymorphism. Objects are designed to share behaviors and they can take on more than one form. The program will determine which meaning or usage is necessary for each execution of that object from a parent class, reducing the need to duplicate code. A child class is then created, which extends the functionality of the parent class. Polymorphism allows different types of objects to pass through the same interface.



Benefits of OOP include:

- **Modularity.** Encapsulation enables objects to be self-contained, making troubleshooting and collaborative development easier.
- **Reusability.** Code can be reused through inheritance, meaning a team does not have to write the same code multiple times.
- **Productivity.** Programmers can construct new programs quicker through the use of multiple libraries and reusable code.
- **Easily upgradable and scalable.** Programmers can implement system functionalities independently.

- **Interface descriptions.** Descriptions of external systems are simple, due to message passing techniques that are used for objects communication.
- **Security.** Using encapsulation and abstraction, complex code is hidden, software maintenance is easier and internet protocols are protected.
- **Flexibility.** Polymorphism enables a single function to adapt to the class it is placed in. Different objects can also pass through the same interface.

Equipment/Software

1. Laptop/ Desktop
2. Codeblocks

Procedure

1.Start your laptop/computer.
2.Open codeblocks IDE.
3.Read and try to understand the problems given.
4.Practice the given codes.
5.Write the codes for given problem.
6.Compile and run the code.
7.Check if it is providing the correct outputs. If the output is not as desired, find the errors and correct it. Continue this process until you get the desired output.

Computer Simulation/Codes

1. **Print HelloWorld in C++.**

```
// Your First C++ Program

#include <iostream>

int main() {

   std::cout << "Hello World!";

    return 0;

}
```

Questions

1. What happens if we do not use “using namespace std”? Why?

2. What are differences between object-oriented programming and structured programming?

Experiment No. 2: Familiarization with Basic C++ programs.

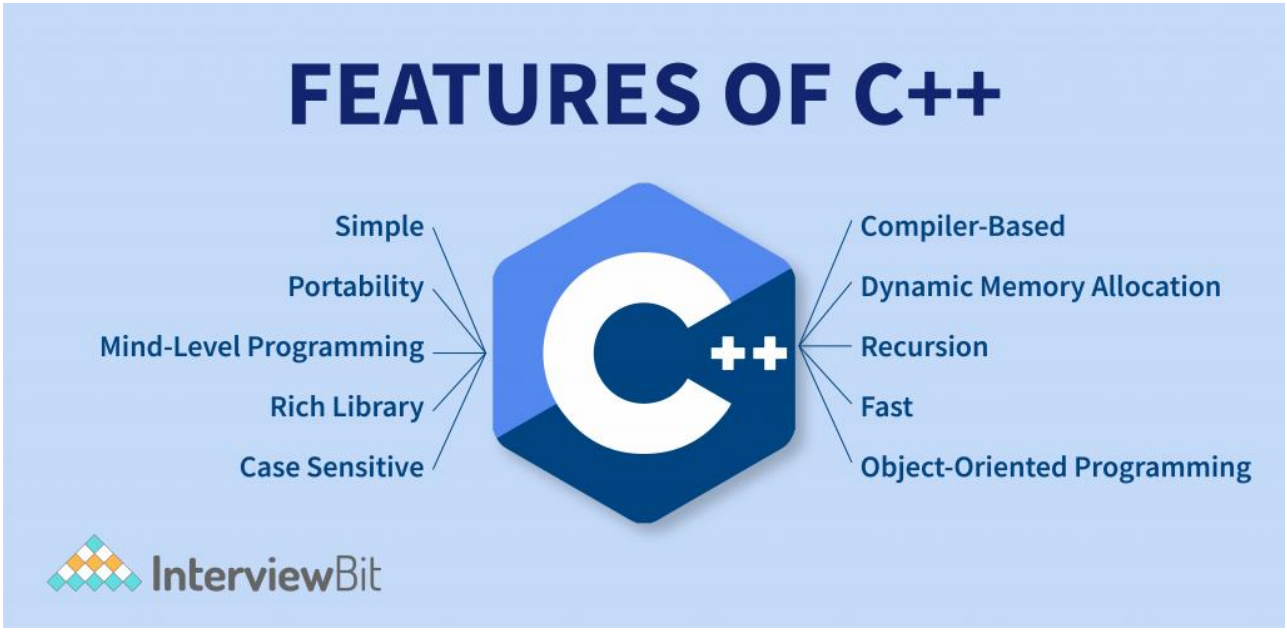
Objective

In this experiment, some of the basic programs will be explored. It will help to develop some fundamental intuitions.

Theory Overview

C++ is a middle-level programming language developed by Bjarne Stroustrup starting in 1979 at Bell Labs. **C++** runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX. This **C++** tutorial adopts a simple and practical approach to describe the concepts of **C++** for beginners to advanded software engineers. **C++** is a MUST for students and working professionals to become a great Software Engineer. I will list down some of the key advantages of learning C++:

- C++ is very close to hardware, so you get a chance to work at a low level which gives you lot of control in terms of memory management, better performance and finally a robust software development.
- **C++ programming** gives you a clear understanding about Object Oriented Programming. You will understand low level implementation of polymorphism when you will implement virtual tables and virtual table pointers, or dynamic type identification.
- C++ is one of the every green programming languages and loved by millions of software developers. If you are a great C++ programmer then you will never sit without work and more importantly you will get highly paid for your work.
- C++ is the most widely used programming languages in application and system programming. So you can choose your area of interest of software development.
- C++ really teaches you the difference between compiler, linker and loader, different data types, storage classes, variable types their scopes etc.



As mentioned before, C++ is one of the most widely used programming languages. It has it's presence in almost every area of software development. I'm going to list few of them here:

- **Application Software Development** - C++ programming has been used in developing almost all the major Operating Systems like Windows, Mac OSX and Linux. Apart from the operating systems, the core part of many browsers like Mozilla Firefox and Chrome have been written using C++. C++ also has been used in developing the most popular database system called MySQL.
- **Programming Languages Development** - C++ has been used extensively in developing new programming languages like C#, Java, JavaScript, Perl, UNIX's C Shell, PHP and Python, and Verilog etc.
- **Computation Programming** - C++ is the best friends of scientists because of fast speed and computational efficiencies.
- **Games Development** - C++ is extremely fast which allows programmers to do procedural programming for CPU intensive functions and provides greater control over hardware, because of which it has been widely used in development of gaming engines.
- **Embedded System** - C++ is being heavily used in developing Medical and Engineering Applications like softwares for MRI machines, high-end CAD/CAM systems etc.

This list goes on, there are various areas where software developers are happily using C++ to provide great softwares. I highly recommend you learn C++ and contribute great softwares to the community.

Equipment

(1) Codeblocks IDE

(2) Laptop/ Desktop

Procedure

1. Start your laptop/computer.
2. Open codeblocks IDE.
3. Read and try to understand the problems given.
4. Practice the given codes.
5. Write the codes for given problem.
6. Compile and run the code.
7. Check if it is providing the correct outputs. If the output is not as desired, find the errors and correct it. Continue this process until you get the desired output.

Computer Simulation

1. **Print Number Entered by User**

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

int main() {
   int number;

   cout << "Enter an integer: ";
```

```
    cin >> number;
    cout << "You entered " << number;
    return 0;
}
```

Output:

```
  Enter an integer: 23
  You entered 23
```

2. Program to Add Two Integers

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

int main() {
  int first_number, second_number, sum;
  cout << "Enter two integers: ";
  cin >> first_number >> second_number;

  // sum of two numbers in stored in variable sumOfTwoNumbers
  sum = first_number + second_number;
  // prints sum
  cout << first_number << " + " <<  second_number << " = " << sum;

  return 0;
}
```

Output:

```
  Enter two integers: 4
  5
  4 + 5 = 9
```

3. Compute quotient and remainder of two number.

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

int main()
{
    int divisor, dividend, quotient, remainder;
    cout << "Enter dividend: ";
    cin >> dividend;
    cout << "Enter divisor: ";
    cin >> divisor;
    quotient = dividend / divisor;
    remainder = dividend % divisor;
    cout << "Quotient = " << quotient << endl;
    cout << "Remainder = " << remainder;
```

```
    return 0;
}
```

Output:

```
Enter dividend: 13
Enter divisor: 4
Quotient = 3
Remainder = 1
```

4. **Find Size of a Variable**

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

int main()
{
    cout << "Size of char: " << sizeof(char) << " byte" << endl;
    cout << "Size of int: " << sizeof(int) << " bytes" << endl;
    cout << "Size of float: " << sizeof(float) << " bytes" << endl;
    cout << "Size of double: " << sizeof(double) << " bytes" << endl;

    return 0;
}
```

Output:

```
Size of char: 1 byte
Size of int: 4 bytes
Size of float: 4 bytes
Size of double: 8 bytes
```

Questions

1. What happens if we save the file in .c format?

2. What are the differences between C and C++?

Experiment No. 3: Experimentation with the conditional statements in C++.

Objective

In this exercise, the conditional statements will be explored and implemented to understand how they function and where to implement them.

Theory Overview

Conditional statements, also known as selection statements, are used to make decisions based on a given condition. If the condition evaluates to True, a set of statements is executed, otherwise another set of statements is executed.

The if Statement: The if statement selects and executes the statement(s) based on a given condition. If the condition evaluates to True then a given set of statement(s) is executed. However, if the condition evaluates to False, then the given set of statements is skipped and the program control passes to the statement following the if statement. The syntax of the if statement is

```
if (condition) {
  statement 1;
  statement 2;
  statement 3;
}
```

The if-else Statement: The if – else statement causes one of the two possible statement(s) to execute, depending upon the outcome of the condition.

The syntax of the if-else statements is

```
if (condition) // if part {
  statement1;
  statement2;
}
else // else part
  statement3;
```

Here, the if-else statement comprises two parts, namely, if and else. If the condition is True the if part is executed. However, if the condition is False, the else part is executed.

To understand the concept of the if -else statement, consider this example.

Example : A code **segment to determine the greater of two numbers**

```
if(x>y)
  cout<<"x is greater";
else
cout<<"y is greater";
```

Nested if-else Statement: A nested if-else statement contains one or more if-else statements. The if else can be nested in three different ways, which are discussed here.

• **The** if – else **statement is nested within the** if **part.**

The syntax is

```
  if (condition1) {
    statement1;
  if(condition2)
    statement2;
  else
    statement3;
  }
else
    statement4;
```

• **The** if-else **statement is nested within the** else **part.**

The syntax is

```
if(condition1)
  statement1;
else {
  statement4;
if (condition2)
  statement2;
else
  statement3;
}
  statement5;
```

• **The** if-else **statement is nested within both the if and the** else **parts.**

The syntax is

```
if(condition1) {
  statement1;
if (condition2)
  statement2;
else
  statement3;
}
else {
  statement4;
if (condition3)
  statement5;
else
  statement6;
}
statement7;
```

To understand the concept of nested if-else, consider this example.

Example : A code segment to determine the largest of three numbers

```
if (a>b) {
  if (a>c)
    cout<<"a is largest";
 }
else //nested if-else statement within else {
 if (b>c)
   cout<<"b is largest";
 else
   cout<<"c is largest";
}
```

The if-else-if ladder, also known as the if-else-if *staircase*, has an if-else statement within the outermost else statement. The inner else statement can further have other if-else statements.

The syntax of the if -e1se- if ladder is

To understand the concept of the if-else-if ladder, consider this example.

Example: A program to determine whether a character is in lower-case or upper case

```
#include<iostream>
using namespace std;
int main() {
  char ch;
  cout<<"Enter an alphabet:";
  cin>>ch;
  if( (ch>='A') && (ch<='Z'))
    cout<<"The alphabet is in upper case";
  else
  if ( (ch>=' a') && (ch<=' z' ) )
    cout<<"The alphabet is in lower case";
  else
  cout<<"It is not an alphabet";
  return 0;
}
```

The output of this program is

Enter an alphabet: $

It is not an alphabet

Conditional Operator as an Alternative: The conditional operator '? :' selects one of the two values or expressions based on a given condition. Due to this decision-making nature of the conditional operator, it is sometimes used as an alternative to if-else statements. Note that the conditional operator selects one of the two values or expressions and not the statements as in the case of an if-else statement. In addition, it cannot select more than one value at a time, whereas if-else statement can select and execute more than one statement at a time. For example, consider this· statement.

```
1max = (x>y ? x : y)
```

This statement assigns maximum of x and y to max

The switch Statement: The switch statement selects a set of statements from the available sets of statements. The switch statement tests the value of an expression in a sequence and compares it with the list of integers or character constants. When a match is found, all the statements associated with that constant are executed.

The syntax of the switch statement

```
switch(expression) {
case <constant1>: statement1;
 [break;]
case <constant2>: statement2;
[break;]
case <constant3>: statement3;
```

```
[default: statement4;]
[break;]
}
Statement5;
```

The C++ keywords case and default provide the list of alternatives. Note that it is not necessary for every case label to specify a unique set of statements. The same set of statements can be shared by multiple case labels.

The keyword default specifies the set of statements to be executed in case no match is found. Note that there can be multiple case labels but there can be only one default label. The break statements in the switch block are optional. However, it is used in the switch block to prevent a fall through. Fall through is a situation that causes the execution of the remaining cases even after a match has been found. In order to prevent this, break statements are used at the end of statements specified by each case and default. This causes the control to immediately break out of the switch block and execute the next statement.

To understand the concept of switch statement, consider this code segment.

Example : A code segment to demonstrate the use of switch statement

```
cin>>x;
int x;
switch(x) {
case l: cout<<"Option1 is selected";
break;
case 2: cout<<"Option2 is selected";
break;
case 3: cout<<"Option3 is selected";
break;
case 4: cout<<"Option4 is selected;
break;
default: cout<<"Invalid option!";
}
```

In this example, depending upon the input, an appropriate message is displayed. That is, if 2 are entered, then the message Option 2 is selected is displayed. In case, 5 is entered, then the message Invalid option! is displayed.

Similar to if and if-else statements, switch statements can also be nested within one another. A nested switch statement contains one or more switch statements within its case label or default label (if any).

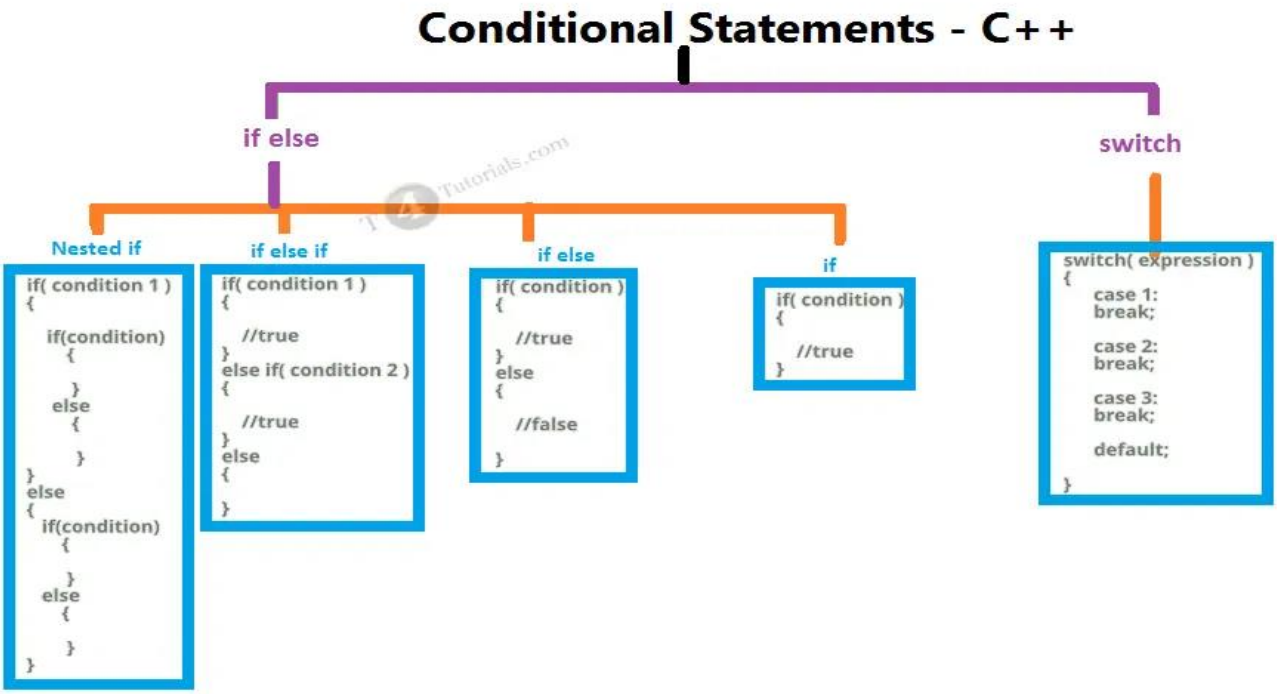
Equipment

1. Codeblocks IDE
2. Laptop/Desktop

Procedure

1. Start your laptop/computer.
2. Open codeblocks IDE.

3. Read and try to understand the problems given.
4. Practice the given codes.
5. Write the codes for given problem.
6. Compile and run the code.
7. Check if it is providing the correct outputs. If the output is not as desired, find the errors and correct it. Continue this process until you get the desired output.



Computer Simulation

1. Linear Search

```
#include <iostream>
using namespace std;
void findElement(int arr[], int size, int key)
{
	// loop to traverse array and search for key
	for (int i = 0; i < size; i++) {
		if (arr[i] == key) {
			cout << "Element found at position: " << (i + 1);
			break;
		}
	}
}
```

```
// Driver program to test above function
int main()
{
	int arr[] = { 1, 2, 3, 4, 5, 6 };
	int n = 6; // no of elements
	int key = 3; // key to be searched
	// Calling function to find the key
	findElement(arr, n, key);
	return 0;
}
```

Sample Input: 1

Sample Output: Element found at position: 3

2. C++ program to print numbers from 1 to 10 using goto statement.

```
#include <iostream>
using namespace std;
// function to print numbers from 1 to 10
void printNumbers()
{
	int n = 1;
label:
	cout << n << " ";
	n++;
	if (n <= 10)
		goto label;
}
// Driver program to test above function
int main()
{
	printNumbers();
	return 0;
}
```

Questions

1. Why the conditionals statements are required?

2. Can the purpose of a specific statements be replaced with another one?

Experiment No. 4: Experimentation with the iteration in C++.

Objective

In this exercise, the iteration or loop statements will be explored and implemented. Through practical work, the functions of these statements would be understood.

Theory Overview

Iteration statements cause statements (or compound statements) to be executed zero or more times, subject to some loop-termination criteria. When these statements are compound statements, they are executed in order, except when either the break statement or the continue statement is encountered. C++ provides four iteration statements — while, do, for, and range-based for. Each of these iterates until its termination expression evaluates to zero (false), or until loop termination is forced with a break statement. The following table summarizes these statements and their actions; each is discussed in detail in the sections that follow.

Iteration Statements

Statement	Evaluated At	Initialization	Increment
`while`	Top of loop	No	No
`do`	Bottom of loop	No	No
`for`	Top of loop	Yes	Yes
range-based for	Top of loop	Yes	Yes

An iteration (or looping) is a sequence of one or more statements that are repeatedly executed until a condition is satisfied. These statements are also called as control flow statements. It is used to reduce the length of code, to reduce time, to execute program and takes less memory space. C++ supports three types of iteration statements.

i. for statement

ii. while statement

iii. do-while statement

All looping statements repeat a set statement as long as a specified condition is remains true. The specified condition is referred as a loop control. For all three loop statements, a true condition is any nonzero value, and a zero value shows a false condition.

Parts of a loop

Every loop has four elements that are used for different purposes. These elements are

• Initialization expression

• Test expression

• Update expression

• The body of the loop

Initialization expression(s): The control variable(s) must be initialized before the control enters into loop. The initialization of the control variable takes place under the initialization expressions. The initialization expression is executed only once in the beginning of the loop.

Test Expression: The test expression is an expression or condition whose value decides whether the loop-body will be execute or not. If the expression evaluates to true (i.e., 1), the body of the loop executed, otherwise the loop is terminated. In an entry-controlled loop, the test-expression is evaluated before the entering into a loop whereas in an exit-controlled loop, the test-expression is evaluated before exit from the loop.

Update expression: It is used to change the value of the loop variable. This statement is executed at the end of the loop after the body of the loop is executed.

The body of the loop: A statement or set of statements forms a body of the loop that are executed repetitively. In an entry-controlled loop, first the test-expression is evaluated and if it is nonzero, the body of the loop is executed otherwise the loop is terminated. In an exit-controlled loop, the body of the loop is executed first then the test-expression is evaluated. If the test-expression is true the body of the loop is repeated otherwise loop is terminated.

Equipment

(1) Codeblocks IDE

(2) Laptop/ Desktop

Procedure

1. Start your laptop/computer.
2. Open codeblocks IDE.
3. Read and try to understand the problems given.
4. Practice the given codes.
5. Write the codes for given problem.
6. Compile and run the code.
7. Check if it is providing the correct outputs. If the output is not as desired, find the errors and correct it. Continue this process until you get the desired output.

Computer Simulation

1. A program to display a countdown using for loop

```
#include<iostream>
using namespace std;
int main() {
  int n;
  for(n=l; n<=10; n++)
    cout<<n<<" "; // body of the loop
    cout<<"\n This is an example of for loop!! !";
    //next statement in sequence
return 0;
}
```

2. C++ program to sum the numbers from 1 to 10

```
#include <iostream>

using namespace std;

int main ()

{

int i, sum=0, n;

cout<<"\n Enter The value of n";

cin>>n;

i =1;

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

     {

     sum += i; }

cout<<"\n The sum of 1 to " <<n<<"is "<<sum;

return 0;}
```

Output

Enter the value of n 5

The sum of 1 to 5 is 15

Questions

1. Why the iteration statements are required?

2. Can the purpose of a specific statements be replaced with another one?

Experiment No. 5: Implementation of C++ in mathematical problem solving.

Objective

In this exercise, some basic and intermediate mathematical problems will be solved using C++ to create a clear understanding and also to have experience on practical problem solving.

Theory Overview

Object Oriented programming languages, such as C++ can be used to solve various types of mathematical problems we encounter in our daily life. C++ has an enriched library to support in solving mathematical problems. In order to solve a problem, the problem must be analyzed first. There are many ways to solve a problem.

Equipment/Software

1. Laptop/ Desktop
2. Codeblocks

Procedure

1. Start your laptop/computer.
2. Open codeblocks IDE.
3. Read and try to understand the problems given.
4. Practice the given codes.
5. Write the codes for given problem.
6. Compile and run the code.
7. Check if it is providing the correct outputs. If the output is not as desired, find the errors and correct it. Continue this process until you get the desired output.

Computer Simulation/Codes

1. Program to add, subtract, multiply and divide two number.

```
#include<iostream.h>
#include<conio.h>
void main() {
 clrscr();
int a, b, res;
cout<<"Enter two number :";
cin>>a>>b; res=a+b;
 cout<<"\nAddition = "<<res;
res=a-b;
 cout<<"\nSubtraction = "<<res; res=a*b;
cout<<"\nMultiplication = "<<res;
res=a/b; cout<<"\nDivision = "<<res;
getch(); }
```

2. Program to check if n is a multiple of 3

```
#include <bits/stdc++.h>
using namespace std;
/* Function to check if n is a multiple of 3*/
int isMultipleOf3(int n)
{
	int odd_count = 0;
	int even_count = 0;
	/* Make no positive if +n is multiple of 3
	then is -n. We are doing this to avoid
	stack overflow in recursion*/
	if (n < 0)
		n = -n;
	if (n == 0)
		return 1;
	if (n == 1)
		return 0;
	while (n) {
		/* If odd bit is set then
		increment odd counter */
		if (n & 1)
			odd_count++;
		/* If even bit is set then
		increment even counter */
		if (n & 2)
			even_count++;
		n = n >> 2;
	}
	return isMultipleOf3(abs(odd_count - even_count));
}
int main()
{
	int num = 24;
```

```
        if (isMultipleOf3(num))
                printf("%d is multiple of 3", num);
        else
                printf("%d is not a multiple of 3", num);
        return 0;
}
```

3. Program to calculate Fibonacci Series

```
#include<bits/stdc++.h>
using namespace std;
int fib(int n)
{
        int a = 0, b = 1, c, i;
        if( n == 0)
                return a;
        for(i = 2; i <= n; i++)
        {
        c = a + b;
        a = b;
        b = c;
        }
        return b;
}
int main()
{       int n = 9;
        cout << fib(n);
        return 0;}
```

Questions

1. Can a problem be solved using various techniques?
2. Do you think it is necessary to solve various types of problems to learn coding?

Experiment No 6: Experimentation with character and string data types.

Objective

In this exercise, the concept of character and string data types will be explored to understand it's implementation and functions.

Theory Overview

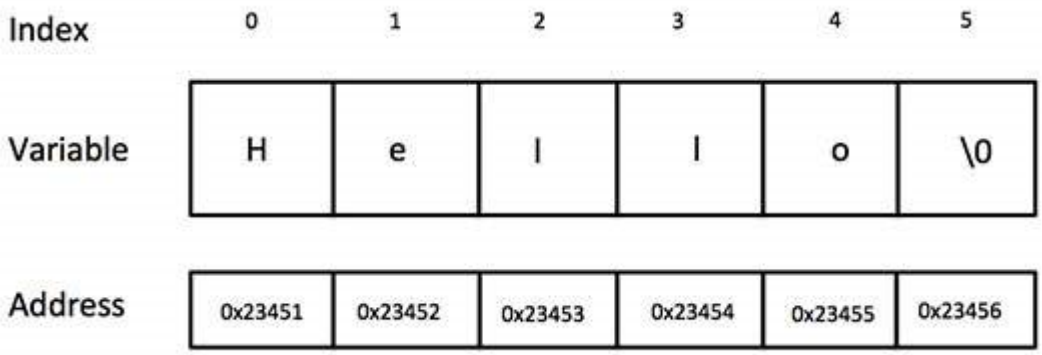
C++ provides following two types of string representations − The C-style character string, the string class type introduced with Standard C++. The C-style character string originated within the C language and continues to be supported within C++. This string is actually a one-dimensional array of characters which is terminated by a **null** character '\0'. Thus a null-terminated string contains the characters that comprise the string followed by a **null**.

The following declaration and initialization create a string consisting of the word "Hello". To hold the null character at the end of the array, the size of the character array containing the string is one more than the number of characters in the word "Hello."

char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};

If you follow the rule of array initialization, then you can write the above statement as follows –

char greeting[] = "Hello";
Following is the memory presentation of above defined string in C/C++ −



Actually, you do not place the null character at the end of a string constant. The C++ compiler automatically places the '\0' at the end of the string when it initializes the array. Let us try to print above-mentioned string –

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

int main () {

   char greeting[6] = {'H', 'e', 'l', 'l', 'o', '\0'};
```

```
  cout << "Greeting message: ";
  cout << greeting << endl;

  return 0;
}
```

Equipment/Software

1. Laptop/ Desktop
2. Codeblocks

Procedure

1.Start your laptop/computer.
2. Open codeblocks IDE.
3. Read and try to understand the problems given.
4. Practice the given codes.
5.Write the codes for given problem.
6. Compile and run the code.
7. Check if it is providing the correct outputs. If the output is not as desired, find the errors and correct it. Continue this process until you get the desired output.

Computer Simulation/Codes

1. C++ program to display a string entered by user.

```
#include <iostream>
using namespace std;
int main()
{
   char str[100];
   cout << "Enter a string: ";
   cin >> str;
   cout << "You entered: " << str << endl;
   cout << "\nEnter another string: ";
   cin >> str;
   cout << "You entered: "<<str<<endl;

   return 0;
}
```

2. Demonstrate the use of library

```
// C++ Program to demonstrate the working of

// getline(), push_back() and pop_back()

#include <iostream>

#include <string> // for string class

using namespace std;
```

```
int main()
{
	// Declaring string
	string str;
	// Taking string input using getline()
	getline(cin, str);
	// Displaying string
	cout << "The initial string is : ";
	cout << str << endl;
	// Inserting a character
	str.push_back('s');
	// Displaying string
	cout << "The string after push_back operation is : ";
	cout << str << endl;
	// Deleting a character
	str.pop_back();
	// Displaying string
	cout << "The string after pop_back operation is : ";
	cout << str << endl;
	return 0;
}
```

Output:

```
The initial string is : geeksforgeek
The string after push_back operation is : geeksforgeeks
The string after pop_back operation is : geeksforgeek
```

Questions

1. Is array of character and string is same?

2. What are the advantages and disadvantages of string?

Experiment No 7: Experimentation with pointer and dynamic memory allocation.

Objective

In this exercise, the concept of pointer and dynamic memory allocation will be explored.

Theory Overview

A **pointer** is a variable whose value is the address of another variable. Like any variable or constant, you must declare a pointer before you can work with it. The general form of a pointer variable declaration is −

type *var-name;

Here, **type** is the pointer's base type; it must be a valid C++ type and **var-name** is the name of the pointer variable. The asterisk you used to declare a pointer is the same asterisk that you use for multiplication. However, in this statement the asterisk is being used to designate a variable as a pointer. Following are the valid pointer declaration −

```
int    *ip;    // pointer to an integer
double *dp;    // pointer to a double
float  *fp;    // pointer to a float
char   *ch     // pointer to character
```

The actual data type of the value of all pointers, whether integer, float, character, or otherwise, is the same, a long hexadecimal number that represents a memory address. The only difference between pointers of different data types is the data type of the variable or constant that the pointer points to. There are few important operations, which we will do with the pointers very frequently. **(a)** We define a pointer variable. **(b)** Assign the address of a variable to a pointer. **(c)** Finally access the value at the address available in the pointer variable. This is done by using unary operator * that returns the value of the variable located at the address specified by its operand.A reference variable is an alias, that is, another name for an already existing variable. Once a reference is initialized with a variable, either the variable name or the reference name may be used to refer to the variable. References are often confused with pointers but three major differences between references and pointers are −

- You cannot have NULL references. You must always be able to assume that a reference is connected to a legitimate piece of storage.
- Once a reference is initialized to an object, it cannot be changed to refer to another object. Pointers can be pointed to another object at any time.
- A reference must be initialized when it is created. Pointers can be initialized at any time.

Equipment/Software

1. Laptop/ Desktop
2. Codeblocks

Procedure

1. Start your laptop/computer.
2. Open codeblocks IDE.

3. Read and try to understand the problems given.
4. Practice the given codes.
5. Write the codes for given problem.
6. Compile and run the code.
7. Check if it is providing the correct outputs. If the output is not as desired, find the errors and correct it. Continue this process until you get the desired output.

Computer Simulation/Codes

1.

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

int main () {
   int  var = 20;   // actual variable declaration.
   int  *ip;        // pointer variable
   ip = &var;       // store address of var in pointer variable

   cout << "Value of var variable: ";
   cout << var << endl;
   // print the address stored in ip pointer variable
   cout << "Address stored in ip variable: ";
   cout << ip << endl;
   // access the value at the address available in pointer
   cout << "Value of *ip variable: ";
   cout << *ip << endl;
   return 0;}
```

2.

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

int main () {
   // declare simple variables
   int    i;
   double d;
 // declare reference variables
   int&    r = i;
   double& s = d;
   i = 5;
   cout << "Value of i : " << i << endl;
   cout << "Value of i reference : " << r  << endl;
   d = 11.7;
   cout << "Value of d : " << d << endl;
   cout << "Value of d reference : " << s  << endl;
   return 0; }
```

Output:
Value of i : 5
Value of i reference : 5

Value of d : 11.7
Value of d reference : 11.7

Questions

1. What are the necessity of pointer?

2. What are the ways to extract values from pointer?

Experiment No. 8: Implementation of class declaration, definition, and access modifiers in C++.

Objective

In this exercise, the concept of class declaration, definition and access modifiers will be explored. Different types of access modifiers will be implemented to understand their basic differences.

Theory Overview

In C++ programming, a Class is a fundamental block of a program that has its own set of methods and variables. You can access these methods and variables by creating an object or the instance of the class. For example, a class of movies may have different movies with different properties, like genres, ratings, length, etc. You can access these properties by creating an object of class movies. A class is a user-defined data type representing a group of similar objects, which holds member functions and variables together. In other words, a class is a collection of objects of the same kind. For example, Facebook, Instagram, Twitter, and Snapchat all come under social media class.

Now, have a look at its declaration and definition.

You can define classes using the keyword ‘class’ followed by the name of the class. Here, inside the class, there are access-modifiers, data variables, and member functions. Now, understand them in detail. Access modifiers: These are the specifiers which provide or grant access for the members. They are of three types:

Private: Only members of the same class have access to private members.

Public: You can access the public members from within, as well as from outside of the class.
Protected: You can access the protected members from the same class members and members of the derived class. It is also accessible from outside the class but with the help of the friend function.

Member-function: Member functions are standard functions declared inside a class. You can invoke it with the help of the dot operator and object, which you will see later. Note: It is necessary to put a semicolon at the end of the class.

```
class Class-name
{
  access-specifier:
    data_variable;

   member-function
   {
      body
    }
};
```

An object is a recognizable entity having a state and behavior, and these objects hold variables of a class in accordance with the access modifiers. It is also known as an instance of a class. You can call a member function with the help object and use the dot operator. During the declaration of the class, no memory is assigned, but when you create an object, then the memory is allocated.

For better understanding, take a look at the syntax below.

```
Class_Name Object_Name;
```

Equipment/Software

1. Laptop/ Desktop
2. Codeblocks

Procedure

1. Start your laptop/computer.
2. Open codeblocks IDE.
3. Read and try to understand the problems given.
4. Practice the given codes.
5. Write the codes for given problem.
6. Compile and run the code.
7. Check if it is providing the correct outputs. If the output is not as desired, find the errors and correct it. Continue this process until you get the desired output.

Computer Simulation/Codes

This example has created a class 'Franchise', and inside that class, there are two functions, i.e., KFC() and BurgerKing() with access specifier as public.

Inside the main function, there is an object fran of class Franchise. You will call both the functions KFC() and BurgerKing() with the help of object fran using the dot operator. By calling both the functions, the message inside these functions is displayed.

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

class Franchise
{
public:
    void KFC()
    {
        cout << "We have the best chicken" << endl;
    }
    void BurgerKing()
    {
        cout << "We have the best burgers" << endl;
    }
};

int main()
{
    Franchise fran;
    fran.KFC();
    fran.BurgerKing();
    return 0;
}
```

Public Class Declaration

1.
```
class S
{
public:
   S();         // public constructor
   S(const S&);  // public copy constructor
   virtual ~S(); // public virtual destructor
private:
   int* ptr; // private data member
};
```
2.

Protected Class Declaration

```
class Base
{
protected:
   int d;
};

class Derived : public Base
{
public:
   using Base::d; // make Base's protected member d a public member of Derived
   using Base::Base; // inherit all bases' constructors (C++11)
};
```

Private Class Declaration

```
// C++ program to demonstrate private
```

```
// access modifier

#include<iostream>
using namespace std;

class Circle
{
	// private data member
	private:
		double radius;

	// public member function
	public:
		double compute_area()
		{ // member function can access private
			// data member radius
			return 3.14*radius*radius;
		}

};

// main function
int main()
{
	// creating object of the class
	Circle obj;

	// trying to access private data member
	// directly outside the class
	obj.radius = 1.5;

	cout << "Area is:" << obj.compute_area();
	return 0;
}
```

Questions

1. Why the access modifiers are necessary?

2. What are the differences between access modifiers.

Experiment No. 9: Implementation of constructor and destructor in C++.

Objective

The purpose of this experiment is to explore and implement constructor and destructor of a class to understand their importance and application.

Theory

A constructor is a member function of a class that has the same name as the class name. It helps to initialize the object of a class. It can either accept the arguments or not. It is used to allocate the memory to an object of the class. It is called whenever an instance of the class is created. It can be defined manually with arguments or without arguments. There can be many constructors in a class. It can be overloaded but it can not be inherited or virtual. There is a concept of copy constructor which is used to initialize an object from another object.

Syntax:

```
ClassName()
 {
   //Constructor's Body
 }
```

Like a constructor, Destructor is also a member function of a class that has the same name as the class name preceded by a tilde(~) operator. It helps to deallocate the memory of an object. It is called while the object of the class is freed or deleted. In a class, there is always a single destructor without any parameters so it can't be overloaded. It is always called in the reverse order of the constructor. if a class is inherited by another class and both the classes have a destructor then the destructor of the child class is called first, followed by the destructor of the parent or base class.
Syntax:

```
~ClassName()
 {
    //Destructor's Body
 }
```

Note: If we do not specify any access modifiers for the members inside the class then by default the access modifier for the members will be Private.

Equipment/Software

1. Laptop/ Desktop
2. Codeblocks

Procedure

1.Start your laptop/computer.
2.Open codeblocks IDE.
3.Read and try to understand the problems given.
4.Practice the given codes.
5.Write the codes for given problem.
6.Compile and run the code.
7.Check if it is providing the correct outputs. If the output is not as desired, find the errors and correct it. Continue this process until you get the desired output.

Computer Simulation/Codes

```
#include <iostream>
using namespace std;
class Z
{
public:
        // constructor
        Z()
        {
                cout<<"Constructor called"<<endl; }
        // destructor
        ~Z()
        {
                cout<<"Destructor called"<<endl;
        }
};
int main()
{
        Z z1; // Constructor Called
        int a = 1;
        if(a==1)
        {
                Z z2; // Constructor Called
        } // Destructor Called for z2
} // Destructor called for z1
```

2. Parameterized Constructor

```
#include <iostream>

using namespace std;
class Line {
  public:
    void setLength( double len );
    double getLength( void );
    Line(double len);  // This is the constructor

  private:
    double length;
};
```

```
// Member functions definitions including constructor
Line::Line( double len) {
   cout << "Object is being created, length = " << len << endl;
   length = len;
}
void Line::setLength( double len ) {
   length = len;
}
double Line::getLength( void ) {
   return length;
}

// Main function for the program
int main() {
   Line line(10.0);

   // get initially set length.
   cout << "Length of line : " << line.getLength() <<endl;

   // set line length again
   line.setLength(6.0);
   cout << "Length of line : " << line.getLength() <<endl;

   return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Object is being created, length = 10
Length of line : 10
Length of line : 6
```

3, Destructor

```
#include <iostream>

using namespace std;
class Line {
   public:
      void setLength( double len );
      double getLength( void );
      Line();   // This is the constructor declaration
      ~Line();  // This is the destructor: declaration

   private:
      double length;
};

// Member functions definitions including constructor
Line::Line(void) {
   cout << "Object is being created" << endl;
}
Line::~Line(void) {
   cout << "Object is being deleted" << endl;
}
```

```
void Line::setLength( double len ) {
  length = len;
}
double Line::getLength( void ) {
  return length;
}

// Main function for the program
int main() {
  Line line;

  // set line length
  line.setLength(6.0);
  cout << "Length of line : " << line.getLength() <<endl;

  return 0;
}
```

When the above code is compiled and executed, it produces the following result −

```
Object is being created
Length of line : 6
Object is being deleted
```

Questions

1. What are the differences between constructor and destructor?

2. Can a destructor be parameterized like a constructor?

Experiment No. 10: Experimentation with the inheritance in C++.

Objectives

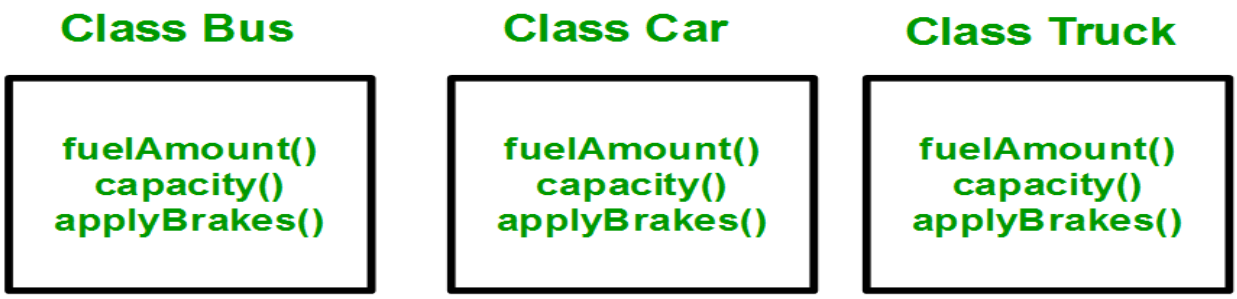
1. To explore the concepts of inheritance.
2. To implement and understand the basics of inheritance.

Theory

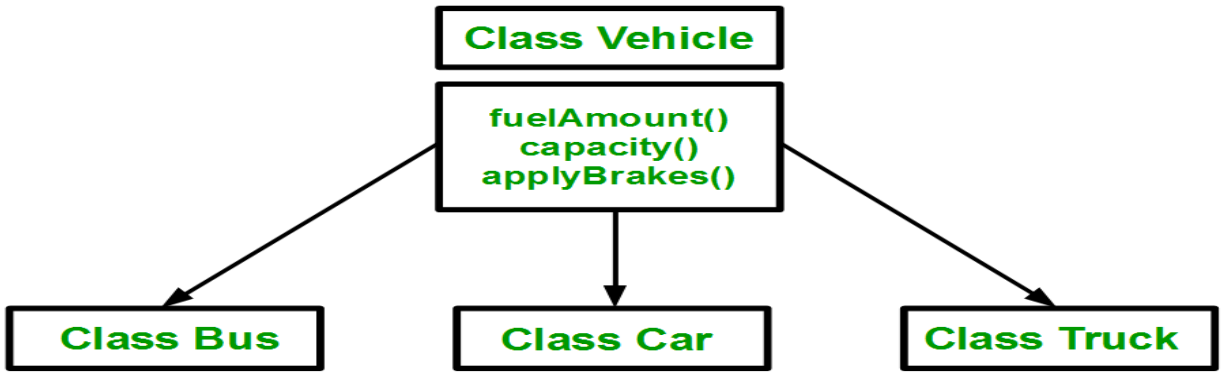
The capability of a class to derive properties and characteristics from another class is called **Inheritance**. Inheritance is one of the most important features of Object-Oriented Programming. Inheritance is a feature or a process in which, new classes are created from the existing classes. The new class created is called "derived class" or "child class" and the existing class is known as the "base class" or "parent class". The derived class now is said to be inherited from the base class. When we say derived class inherits the base class, it means, the derived class inherits all the properties of the base class, without changing the properties of base class and may add new features to its own. These new features in the derived class will not affect the base class. The derived class is the specialized class for the base class.

- **Sub Class:** The class that inherits properties from another class is called Subclass or Derived Class.
- **Super Class:** The class whose properties are inherited by a subclass is called Base Class or Superclass.

Consider a group of vehicles. You need to create classes for Bus, Car, and Truck. The methods fuelAmount(), capacity(), applyBrakes() will be the same for all three classes. If we create these classes avoiding inheritance then we have to write all of these functions in each of the three classes as shown below figure:



You can clearly see that the above process results in duplication of the same code 3 times. This increases the chances of error and data redundancy. To avoid this type of situation, inheritance is used. If we create a class Vehicle and write these three functions in it and inherit the rest of the classes from the vehicle class, then we can simply avoid the duplication of data and increase re-usability. Using inheritance, we have to write the functions only one time instead of three times as we have inherited the rest of the three classes from the base class (Vehicle). Look at the below diagram in which the three classes are inherited from vehicle class:



Derived Classes: A Derived class is defined as the class derived from the base class.
Syntax:

```
class  <derived_class_name> : <access-specifier> <base_class_name>
{
      //body
}
```

Where
class — keyword to create a new class
derived_class_name — name of the new class, which will inherit the base class
access-specifier — either of private, public or protected. If neither is specified, PRIVATE is taken as default
base-class-name — name of the base class

Note: A derived class doesn't inherit ***access*** to private data members. However, it does inherit a full parent object, which contains any private members which that class declares.

Example:

```
1. class ABC : private XYZ          //private derivation
        {           }
2. class ABC : public XYZ           //public derivation
        {           }
3. class ABC : protected XYZ        //protected derivation
        {           }
4. class ABC: XYZ                   //private derivation by default
{         }
```

Equipment/Software

1. Laptop/ Desktop
2. Codeblocks

Procedure

1. Start your laptop/computer.
2. Open codeblocks IDE.
3. Read and try to understand the problems given.
4. Practice the given codes.
5. Write the codes for given problem.
6. Compile and run the code.
7. Check if it is providing the correct outputs. If the output is not as desired, find the errors and correct it. Continue this process until you get the desired output.

Computer Simulation/Codes

```
// Example: define member function without argument within the class

#include<iostream>
using namespace std;

class Person
{
        int id;
        char name[100];

        public:
                void set_p()
                {
                        cout<<"Enter the Id:";
                        cin>>id;
                        fflush(stdin);
                        cout<<"Enter the Name:";
                        cin.get(name,100);
                }
```

```
		void display_p()
		{
			cout<<endl<<id<<"\t"<<name;
		}
};

class Student: private Person
{
	char course[50];
	int fee;

	public:
	void set_s()
		{
			set_p();
			cout<<"Enter the Course Name:";
			fflush(stdin);
			cin.getline(course,50);
			cout<<"Enter the Course Fee:";
			cin>>fee;
		}

		void display_s()
		{
			display_p();
			cout<<"t"<<course<<"\t"<<fee;
		}
};

main()
{
	Student s;
	s.set_s();
	s.display_s();
	return 0;
}

// Example: define member function without argument outside the class

#include<iostream>
using namespace std;

class Person
{
	int id;
	char name[100];

	public:
		void set_p();
		void display_p();
};

void Person::set_p()
{
	cout<<"Enter the Id:";
```

```
	cin>>id;
	fflush(stdin);
	cout<<"Enter the Name:";
	cin.get(name,100);
}

void Person::display_p()
{
	cout<<endl<<id<<"\t"<<name;
}

class Student: private Person
{
	char course[50];
	int fee;

	public:
		void set_s();
		void display_s();
};

void Student::set_s()
{
	set_p();
	cout<<"Enter the Course Name:";
	fflush(stdin);
	cin.getline(course,50);
	cout<<"Enter the Course Fee:";
	cin>>fee;
}

void Student::display_s()
{
	display_p();
	cout<<"t"<<course<<"\t"<<fee;
}

main()
{
	Student s;
	s.set_s();
	s.display_s();
	return 0;
}
```

Questions

1. What is the importance of inheritance?
2. what are the differences between various types of inheritance?

Experiment No. 11: Implementation of operator overloading in C++.

Objective

The objective of this exercise is to examine the operation of operator overloading in C++.

Theory Overview

In C++, we can make operators work for user-defined classes. This means C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. For example, we can overload an operator '+' in a class like String so that we can concatenate two strings by just using +. Other example classes where arithmetic operators may be overloaded are Complex Numbers, Fractional Numbers, Big Integer, etc.

Operator overloading is a compile-time polymorphism. It is an idea of giving special meaning to an existing operator in C++ without changing its original meaning.
Example:

```
 int a;
float b,sum;
sum=a+b;
```

Here, variables "a" and "b" are of types "int" and "float", which are built-in data types. Hence the addition operator '+' can easily add the contents of "a" and "b". This is because the addition operator "+" is predefined to add variables of built-in data type only.

Now, consider another example

```
class A
{

};

int main()
{
    A   a1,a2,a3;

    a3= a1 + a2;

    return 0;
}
```

In this example, we have 3 variables "a1", "a2" and "a3" of type "class A". Here we are trying to add two objects "a1" and "a2", which are of user-defined type i.e. of type "class A" using the "+" operator. This is not allowed, because the addition operator "+" is predefined to operate only on built-in data types. But here, "class A" is a user-defined type, so the compiler generates an error. This is where the concept of "Operator overloading" comes in.
Now, if the user wants to make the operator "+" to add two class objects, the user has to redefine the meaning of the "+" operator such that it adds two class objects. This is done by using the concept "Operator overloading". So the main idea behind "Operator overloading" is to use c++

operators with class variables or class objects. Redefining the meaning of operators really does not change their original meaning; instead, they have been given additional meaning along with their existing ones.

What is the difference between operator functions and normal functions?
Operator functions are the same as normal functions. The only differences are, that the name of an operator function is always the operator keyword followed by the symbol of the operator and operator functions are called when the corresponding operator is used.
Following is an example of a global operator function.

Output

```
12 + i9
```

Operators that can be overloaded

1. Binary Arithmetic -> +, -, *, /, %
2. Unary Arithmetic -> +, -, ++, —
3. Assignment -> =, +=,*=, /=,-=, %=
4. Bit- wise -> & , | , << , >> , ~ , ^
5. De-referencing -> (->)
6. Dynamic memory allocation and De-allocation -> New, delete
7. Subscript -> [ ]
8. Function call -> ()
9. Logical -> &, | |, !
10. Relational -> >, < , = =, <=, >=

Equipment/Software

1. Laptop/ Desktop
2. Codeblocks

Procedure

1. Start your laptop/computer.
2. Open codeblocks IDE.
3. Read and try to understand the problems given.
4. Practice the given codes.
5. Write the codes for given problem.
6. Compile and run the code.
7. Check if it is providing the correct outputs. If the output is not as desired, find the errors and correct it. Continue this process until you get the desired output.

Computer Simulation/Codes

1. First Example

```
#include <iostream>
```

```
using namespace std;
class ComplexNumber{
private:
int real;
int imaginary;
public:
ComplexNumber(int real, int imaginary){
        this->real = real;
        this->imaginary = imaginary;
}
void print(){
        cout<<real<<" + i"<<imaginary;
}
ComplexNumber operator+ (ComplexNumber c2){
        ComplexNumber c3(0,0);
        c3.real = this->real+c2.real;
        c3.imaginary = this->imaginary + c2.imaginary;
        return c3;
}
};
int main() {
        ComplexNumber c1(3,5);
        ComplexNumber c2(2,4);
        ComplexNumber c3 = c1 + c2;
        c3.print();
        return 0;
}
```

Output

```
5 + i9
```

2. Second Example

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

```
{
private:
        int num, den;
public:
        Fraction(int n, int d) { num = n; den = d; }

        // Conversion operator: return float value of fraction
        operator float() const {
                return float(num) / float(den);
        }
};

int main() {
        Fraction f(2, 5);
        float val = f;
        cout << val << '\n';
        return 0;
}
```

Questions

Experiment No. 12: Experimentation with function overloading and overriding in C++.

Objective

The objective of this exercise is to observe and understanding of function overloading and function overriding through implementation.

Theory Overview

In C++, two or more functions can have the same name if the number or the type of parameters are different, this is known as function overloading whereas function overriding is the redefinition of base class function in its derived class with the same signature. Functions are a way to bind together code that is meant to be called repeatedly. After all, you wouldn't want to copy paste that code of yours again and again, right? What if you needed to modify that code? You would need to change the code at every instance mindlessly. That is one of the reasons functions are one of the essential parts of procedural programming.

But, there are instances where you would need the alias (the name) of the function to be reused depending upon the context. This is what we call polymorphism. Polymorphism literally means multiple forms (poly-morphisms) so in our instance we mean that the function alias will have multiple forms ! One good example of polymorphism is:

We make sound via talking but so do other animals. So, if we were to design an interface for Animals, makeSound() would be a common interface for all.

Function Overloading VS Function Overriding:

1. **Inheritance:** Overriding of functions occurs when one class is inherited from another class. Overloading can occur without inheritance.
2. **Function Signature:** Overloaded functions must differ in function signature ie either number of parameters or type of parameters should differ. In overriding, function signatures must be same.
3. **Scope of functions:** Overridden functions are in different scopes; whereas overloaded functions are in same scope.
4. **Behavior of functions:** Overriding is needed when derived class function has to do some added or different job than the base class function. Overloading is used to have same name functions which behave differently depending upon parameters passed to them.
5. **Execution:** Function overloading takes place at compile time so known as compile time polymorphism and Function Overriding occurs at run time so known as run time polymorphism.

Equipment/Software

1. Laptop/ Desktop
2. Codeblocks

Procedure

1. Start your laptop/computer.
2. Open codeblocks IDE.
3. Read and try to understand the problems given.
4. Practice the given codes.
5. Write the codes for given problem.
6. Compile and run the code.
7. Check if it is providing the correct outputs. If the output is not as desired, find the errors and correct it. Continue this process until you get the desired output.

Computer Simulation/Codes

1. Function Overloading

```
// CPP program to illustrate

// Function Overloading

#include <iostream>

using namespace std;

// overloaded functions
```

```
void test(int);
void test(float);
void test(int, float);
int main()
{
	int a = 5;
	float b = 5.5;

	// Overloaded functions
	// with different type and
	// number of parameters
	test(a);
	test(b);
	test(a, b);
	return 0;
}
// Method 1
void test(int var)
{
	cout << "Integer number: " << var << endl;
}
// Method 2
void test(float var)
{
	cout << "Float number: "<< var << endl;
}
// Method 3
void test(int var1, float var2)
{
	cout << "Integer number: " << var1;
	cout << " and float number:" << var2;
}
```

Output:

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

2. Function Overriding

```
// CPP program to illustrate
// 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's display method
	void Display()
	{
		cout << "\nThis is Display() method"
			" of DerivedClass";
	}
};
// Driver code
```

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

Output:

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

Questions

1. Why function overloading and function overriding is necessary?

2. Is function overriding related with inheritance?

Experiment No. 13: Implementation of exception handling in C++.

Objective

The objective of this exercise is to examine exception handling in C++ through practical implementation.

Theory Overview

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

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

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

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

```
try {
   // protected code
} catch( ExceptionName e1 ) {
   // catch block
} catch( ExceptionName e2 ) {
   // catch block
} catch( ExceptionName eN ) {
   // catch block
}
```

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

Exceptions can be thrown anywhere within a code block using **throw** statement. The operand of the throw statement determines a type for the exception and can be any expression and the type of the result of the expression determines the type of exception thrown.

Following is an example of throwing an exception when dividing by zero condition occurs −

```
double division(int a, int b) {
   if( b == 0 ) {
      throw "Division by zero condition!";
   }
   return (a/b);
}
```

The **catch** block following the **try** block catches any exception. You can specify what type of exception you want to catch and this is determined by the exception declaration that appears in parentheses following the keyword catch.

```
try {
   // protected code
} catch( ExceptionName e ) {
  // code to handle ExceptionName exception
}
```

Above code will catch an exception of **ExceptionName** type. If you want to specify that a catch block should handle any type of exception that is thrown in a try block, you must put an ellipsis, ..., between the parentheses enclosing the exception declaration as follows −

```
try {
   // protected code
} catch(...) {
  // code to handle any exception
}
```

Equipment/Software

1. Laptop/ Desktop
2. Codeblocks

Procedure

1. Start your laptop/computer.
2. Open codeblocks IDE.
3. Read and try to understand the problems given.
4. Practice the given codes.
5. Write the codes for given problem.
6. Compile and run the code.
7. Check if it is providing the correct outputs. If the output is not as desired, find the errors and correct it. Continue this process until you get the desired output.

Computer Simulation/Codes

1. Example Code

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

struct MyException : public exception {
   const char * what () const throw () {
      return "C++ Exception";
   }
};

int main() {
   try {
      throw MyException();
   } catch(MyException& e) {
      std::cout << "MyException caught" << std::endl;
      std::cout << e.what() << std::endl;
   } catch(std::exception& e) {
      //Other errors
   }
}
```

This would produce the following result −

```
MyException caught
C++ Exception
```

2. Example Code

```
#include <iostream>

using namespace std;

int main()

{

int x = -1;
```

```
// Some code
cout << "Before try \n";
try {
        cout << "Inside try \n";
        if (x < 0)
        {
                throw x;
                cout << "After throw (Never executed) \n";
        }
}
catch (int x ) {
        cout << "Exception Caught \n";
}
cout << "After catch (Will be executed) \n";
return 0;
}
```

Output:

```
Before try
Inside try
Exception Caught
After catch (Will be executed)
```

Questions

1. What is the importance of exception handling?

2. Is it possible to handle every type of exception?

ANNEXURE-I

Assessment Rubric

1. Laboratory Report

Category	Outstanding (Up to 100%)	Accomplished (Up to 75%)	Developing (Up to 50%)	Beginner (Up to 25%)
Write up format	Aim, Apparatus, material requirement, theoretical basis, procedure of experiment, sketch of the experimental setup etc. is demarcated and presented in clearly labeled and neatly organized sections.	The write up follows the specified format but a couple of the specified parameters are missing.	The report follows the specified format but a few of the formats are missing and the experimental sketch is not included in the report	The write up does not follow the specified format and the presentation is shabby.
Observations and Calculations	The experimental observations and calculations are recorded in neatly prepared table with correct units and significant figures. One sample calculation is explained by substitution of values	The experimental observations and calculations are recorded in neatly prepared table with correct units and significant figures but sample calculation is not shown	The experimental observations and calculations are recorded neatly but correct units and significant figures are not used. Sample calculation is also not shown	The experimental observations and results are recorded carelessly. Correct units significant figures are not followed and sample calculations not shown
Results and Graphs	Results obtained are correct within reasonable limits. Graphs are drawn neatly with labeling of the axes. Relevant calculations are performed from the graphs. Equations are obtained by regression analysis or curve fitting if relevant	Results obtained are correct within reasonable limits. Graphs are drawn neatly with labeling of the axes. Relevant calculations from the graphs are incomplete and equations are not obtained by regression analysis or curve fitting	Results obtained are correct within reasonable limits. Graphs are not drawn neatly and or labeling is not proper. No calculations are done from the graphs and equations are not obtained by regression analysis or curve fitting	Results obtained are not correct within reasonable limits. Graphs are not drawn neatly and or labeling is not proper. No calculations are done from the graphs and equations are not obtained by regression
Discussion of results	All relevant points of the result are discussed and justified in light of theoretical expectations. Reasons for divergent results are identified and corrective measures discussed.	Results are discussed but no theoretical reference is mentioned. Divergent results are identified but no satisfactory reasoning is given for the same.	Discussion of results is incomplete and divergent results are not identified.	Neither relevant points of the results are discussed nor divergent results identified

2. Individual Presentation

Category	Outstanding (Up to 100%)	Accomplished (Up to 75%)	Developing (Up to 50%)	Beginner (Up to 25%)
Content	A concise summary of the topic; Convincing justification for choice of topic; Comprehensive and complete coverage of information	A good summary of the topic; Acceptable justification for choice of topic; Most important information covered; Little irrelevant information	Informative but much of the information irrelevant; Confused justification for choice of topic; Coverage of some of the major points	A brief look at the topic; Little justification for choice of topic; Majority of information irrelevant and significant points left out
Organization	Clear purpose and subject; Pertinent examples, facts, and/or statistics; Supports conclusions/ideas with evidence	Somewhat clear purpose and subject; Some examples, facts, and/or statistics that support the subject; Some data or evidence that supports conclusions	Attempts to define purpose and subject; Weak examples, facts, and/or statistics not adequately supporting the subject; Very thin data or evidence to support conclusion	Subject and purpose not clearly defined; Weak or no support of subject; Insufficient support for ideas or conclusions
Visual Aids	Information is clear and concise with proper key information in points or phrases; Visually appealing/ engaging	Too much information in complete sentences on slides along with proper key information in phrases; Significant visual appeal	Too much information in complete sentences on many slides; Some proper key information; Minimal effort made to make slides appealing	Too much information in complete sentences on slides; No or few proper key information; Repetition of the same information on multiple slides; No visual appeal
Delivery Style	Regular eye contact; Appropriate speaking volume & body language; Proper pace and diction; Fluent avoidance of repetitions, hesitations, gap fillers	Steady eye contact; Adequate volume and energy; Generally good pace and diction; Few or no distracting Gestures; Few repetitions, hesitations, gap fillers	More volume or energy needed at times; Pace too slow or fast; Some distracting gestures or posture; Some repetitions, hesitations, gap fillers	Low volume and energy; Pace too slow or fast; Poor diction; Lots of distracting gestures or posture; Frequent repetitions, hesitations, gap fillers
Question -answer Session	Demonstrates knowledge by answering all types of questions with explanations and elaboration in professional manner	Is at ease with expected answers to all questions without elaboration in somewhat professional manner	Is uncomfortable with information and can answer only rudimentary questions	Does not have grasp of information and cannot answer questions about subject

ANNEXURE-II

Program Outcomes

Program Outcomes (POs) represent the knowledge, skills and attitudes the students should have at the end of a four year engineering program. CSE program of BAUET has 12 Program Outcomes. They are briefly described in the following table.

Sl. No	PO	Category	Description
1	**PO 1**	**Engineering Knowledge**	Apply the **knowledge** of **mathematics**, **science**, **engineering fundamentals**, and an **engineering specialization** to the solution of complex engineering problems.
2	**PO 2**	**Problem Analysis**	**Identify**, formulate, research literature, and analyze complex engineering problems reaching substantiated conclusions **using first principles** of mathematics, natural sciences, and engineering sciences.
3	**PO 3**	**Design/Development of Solutions**	**Design solutions** for complex engineering problems and **design system** components or processes that meet the specified needs with appropriate consideration for public health and safety as well as cultural, social and environmental concerns.
4	**PO 4**	**Investigation**	**Conduct investigations** of complex problems, considering design of experiments, analysis and interpretation of data and synthesis of information to provide valid conclusions.
5	**PO 5**	**Modern Tool Usage**	**Create, select, and apply** appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
6	**PO 6**	**The Engineer and Society**	**Apply reasoning** informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
7	**PO 7**	**Environment and Sustainability**	Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of need for sustainable development.
8	**PO 8**	**Ethics**	Apply ethical principles and commit to professional Ethics and responsibilities and norms of the engineering practice.
9	**PO 9**	**Individual and Team Work**	Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.
10	**PO 10**	**Communication**	Communicate effectively on complex engineering activities with the engineering community and with society at large. Some of them are, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
11	**PO 11**	**Project Management and Finance**	Demonstrate knowledge and understanding of the engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
12	**PO 12**	**Life Long Learning**	Recognize the need for, and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

ANNEXURE-III

Knowledge Profile, Complex Engineering Problem and Complex Engineering Activities

Knowledge Profile (WK/K)- CHARACTERISTIC

WK1	**Natural Sciences**	A systematic, theory-based understanding of the natural sciences applicable to the discipline
WK2	**Mathematics**	Conceptually-based mathematics, numerical analysis, statistics and formal aspects of computer and information science to support analysis and modelling applicable to the discipline
WK3	**Engineering fundamentals**	A systematic, theory-based formulation of engineering fundamentals required in the engineering discipline
WK4	**Specialist knowledge**	Engineering specialist knowledge that provides theoretical frameworks and bodies of knowledge for the accepted practice areas in the engineering discipline; much is at the forefront of the discipline.
WK5	**Engineering design**	Knowledge that supports engineering design in a practice area
WK6	**Engineering practice**	Knowledge of engineering practice (technology) in the practice areas in the engineering discipline
WK7	**Comprehension**	Comprehension of the role of engineering in society and identified issues in engineering practice in the discipline: ethics and the professional responsibility of an engineer to public safety; the impacts of engineering activity: economic, social, cultural, environmental and sustainability
WK8	**Research literature**	Engagement with selected knowledge in the research literature of the discipline

Complex Engineering Problem (WP/P)

WP	**Preamble**	**COMPLEX PROBLEMS have characteristic of WP1 and some or all of WP2 to WP7**
WP1	**Depth of Knowledge**	**In-depth engineering knowledge** at the level of one or more of WK3, WK4, WK5, WK6 or WK8 which allows a fundamental based, first principles analytical approach
WP2	**Conflicting requirement**	Wide-ranging or **conflicting** technical, engineering and other issues
WP3	**Depth of analysis**	**no obvious solution** and require abstract thinking, originality in analysis to formulate suitable models
WP4	**Familiarity of issues**	**infrequently** encountered issues
WP5	**Extent of applicable codes**	outside problems encompassed by standards and codes of practice for professional engineering
WP6	**Extent of stakeholder**	**diverse groups** of stakeholders with widely varying needs
WP7	**Interdependence**	high level problems including **many component parts** or sub-problems

Complex Engineering Activities (EA)

Activities	Preamble	**Complex activities** means (engineering) activities or projects that have **some or all** of the following characteristics listed below
EA1	**Range of resources**	**Diverse resources** (people, money, equipment, materials, information and technologies).
EA2	**Level of interaction**	Require resolution of significant problems arising from interactions between **wide ranging** or **conflicting** technical, engineering or other issues.
EA3	**Innovation**	Involve creative use of engineering principles and **research-based** knowledge in **novel** ways.
EA4	**Consequences to society and the environment**	Have **significant consequences** in a **range of contexts**, characterised by **difficulty** of prediction and mitigation.
EA5	**Familiarity**	Can extend **beyond previous** experiences by applying **principles-based** approaches.

The End