DEPARTMENT OF COMPUTER ENGINEERING

LABORATORY MANUAL

OBJECT ORRIENTED PROGRAMMING LABORATORY

(BTCOL306)

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY DHULE (M.S.)

DEPARTMENT OF COMPUTER ENGINEERING

Vision of Institute

To be a socially sensitive engineering institute of excellence adding value to the nation.

Mission of Institute

1. To provide resources of excellence with a focus on nurturing and developing the society.
2. To strive to be an institute of global recognition.

Vision of Department

We envision a globally recognized and innovative computer engineer who meets socio-economical and industrial needs.

Mission of Department

1. To empower students with comprehensive knowledge of Computer Engineering to be successful professionals.
2. To ensure quality education that prepares students for careers in industry and higher education.
3. To develop leadership skills in students while instilling strong ethical values and encouraging lifelong learning.

Program Educational Objectives (PEOs) of Department

PEO I: Graduates will have a successful professional career while maintaining strong ethical values.

PEO II: Graduates must demonstrate knowledge and technological skills to resolve real life problems.

PEO III: Graduates will progress as professionals, researchers, or entrepreneurs who continues to learn and adopt emerging technology.

COMPUTER ENGINEERING DEPARTMENT

Program Outcomes

Engineering Graduates will be able to:

1. Engineering knowledge: Apply the knowledge of mathematics, science, engineering fundamentals, and an engineering specialization to the solution of complex engineering problems.

2. Problem analysis: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.

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 the public health and safety, and the cultural, societal, and environmental considerations.

4. Conduct investigations of complex problems: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

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. 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. Environment and sustainability: Understand the impact of the professional engineering solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.

8. Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.

9. Individual and team work: Function effectively as an individual, and as a member or leader in diverse teams, and in multidisciplinary settings.

10. Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.

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. Life-long learning: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

COMPUTER ENGINEERING DEPARTMENT

Program Specific Outcomes (PSO) addressed by the Course:

A graduate of the Computer Engineering Program will demonstrate-

PSO1: Professional Skills-To gain the ability to comprehend, analyze, design and implement computer programs in the fields of computer algorithms, web development, data science, computer network and security, software design, system software cloud computing and allied fields.

PSO2: Problem-Solving Skills- Capability to provide computer based solutions to a variety of problems by applying standard practices, problem solving strategies and methodologies.

PSO3: Professional Career - The ability to create an innovative career path by utilizing modern computer tools and technologies.

Dr. Babasaheb Ambedkar Technological University

Syllabus & Scheme

Course Code	Course Title	Weekly Teaching hrs	Evaluation Scheme			Credit
			MSE	CA	ESE	
BTCOL306	Object Oriented Programming Lab	**4**	**-**	**60**	**40**	**2**

BTCOL306 Object Oriented Programming Laboratory

The main aim of object-oriented programming is to implement real-world entities, for example, object, classes, abstraction, inheritance, polymorphism, etc. Object Oriented Programming is to Understand the features of C++, the relative merits of C++ as an object oriented programming language and to understand how to produce object-oriented software using C++ It also helps in understanding the advanced features of C++ specifically stream I/O, templates and operator overloading

Shri Vile Parle Kelavani Mandal's

Institute of Technology, Dhule.

www.svkm-iot.ac.in

Survey No. 499, Plot No.2, Behind Gurudwara, Mumbai-Agra Road, Dhule(M.S.)

DEPARMENT OF COMPUTER ENGG.

LABORATORY : BTCOL306 Object Oriented Programming Laboratory | **SEMESTER : 4th Sem**

Master List of Practical

SR. NO.	PRACTICAL DESCRIPTION	Page No
1	Programs on Operators and Inline Function.	
2	Programs on dealing with Arrays and Sorting of Array	
3	Programs on Classes and Objects	
4	Programs on Inheritance and Polymorphism.	
5	Programs on Operator Overloading .	
6	Programs on Friend Function.	
7	Programs on file handling and stream manipulation.	
8	Programs on Dynamic Memory Management	
9	Programs on Exception Handling.	
10	Programs on generic programming using templates.	
11	Open Ended Assignment: Minor project.	

Subject In-charge: Dr.Kavita Patil Prof.Rinku M.Sharma	**APPROVED BY :** Dr. Makarand Shahade HOD, Department of Computer Engineering

Rubrics for Assessment

Assignment/Experiment :

Date of Performance: **Date of submission :**

Marks Split Up to	Maximum Marks	Marks Obtained
Performance/ conduction	3	
Report Writing	3	
Attendance	2	
Viva/Oral	2	
Total Marks	10	
Signature of Subject Teacher		

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : OOP Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 01

Title : Programs on Operators and Inline Function.

Aim :	Write a program using inline function to perform multiplication and division of two numbers.
Objective:	Programs on Operators and Inline Function.
Theory:	**Operators in C++** An operator is a symbol that operates on a value to perform specific mathematical or logical computations. They form the foundation of any programming language. In C++, we have built-in operators to provide the required functionality.An operator operates the operands. For example, `int c = a + b;` Here, '+' is the addition operator. 'a' and 'b' are the operands that are being 'added'. **Operators in C++ can be classified into 6 types:** 1. Arithmetic Operators 2. Relational Operators 3. Logical Operators 4. Bitwise Operators 5. Assignment Operators 6. Ternary or Conditional Operators **Arithmetic Operators can be classified into 2 Types:** **Unary Operators:** These operators operate or work with a single operand. For example: Increment(++) and Decrement(–) Operators.

Name	Symbol	Description	Example
Increment Operator	++	Increases the integer value of the variable by one	int a = 5; a++; // returns 6

Decrement Operator	—	Decreases the integer value of the variable by one	int a = 5; a–; // returns 4

Binary Operators: These operators operate or work with two operands. For example: Addition(+), Subtraction(-), etc.

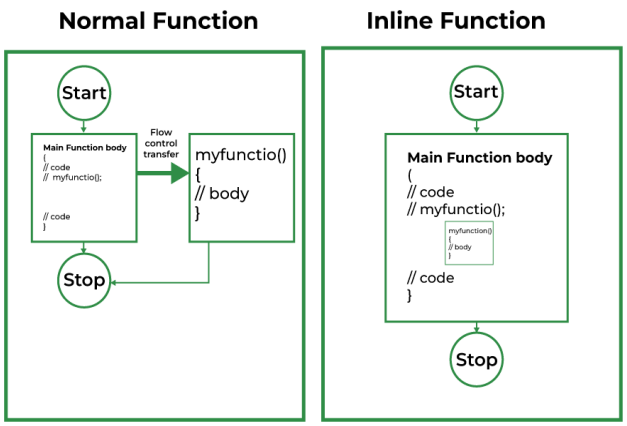
Name	Symbol	Description	Example
Addition	+	Adds two operands	int a = 3, b = 6; int c = a+b; // c = 9
Subtraction	–	Subtracts second operand from the first	int a = 9, b = 6; int c = a-b; // c = 3
Multiplication	*	Multiplies two operands	int a = 3, b = 6; int c = a*b; // c = 18
Division	/	Divides first operand by the second operand	int a = 12, b = 6; int c = a/b; // c = 2
Modulo Operation	%	Returns the remainder an integer division	int a = 8, b = 6; int c = a%b; // c = 2

Inline Functions in C++

C++ provides inline functions to reduce the function call overhead. An inline function is a function that is expanded in line when it is called. When the inline function is called whole code of the inline function gets inserted or substituted at the point of the inline function call. This substitution is performed by the C++ compiler at compile time. An inline function may increase efficiency if it is small.

Syntax:

```
inline return-type function-name(parameters)
{
    // function code
}
```



Remember, inlining is only a request to the compiler, not a command. The compiler can ignore the request for inlining.

The compiler may not perform inlining in such circumstances as:

1. If a function contains a loop. (*for, while and do-while*)
2. If a function contains static variables.
3. If a function is recursive.
4. If a function return type is other than void, and the return statement doesn't exist in a function body.
5. If a function contains a switch or goto statement.

Why Inline Functions are Used?

When the program executes the function call instruction the CPU stores the memory address of the instruction following the function call, copies the arguments of the function on the stack, and finally transfers control to the specified function. The CPU then executes the function code, stores the function return value in a predefined memory location/register, and returns control to the calling function. This can become overhead if the execution time of the function is less than the switching time from the caller function to called function (callee).

For functions that are large and/or perform complex tasks, the overhead of the function call is usually insignificant compared to the amount of time the function takes to run. However, for small, commonly-used functions, the time needed to make the function call is often a lot more than the time needed to actually execute the function's code. This overhead occurs for small functions because the execution time of a small function is less than the switching time.

Inline functions Advantages:

1. Function call overhead doesn't occur.
2. It also saves the overhead of push/pop variables on the stack when a function is called.
3. It also saves the overhead of a return call from a function.
4. When you inline a function, you may enable the compiler to perform context-specific optimization on the body of the function. Such optimizations are not

	possible for normal function calls. Other optimizations can be obtained by considering the flows of the calling context and the called context. 5. An inline function may be useful (if it is small) for embedded systems because inline can yield less code than the function called preamble and return. **Inline function Disadvantages:** 1. The added variables from the inlined function consume additional registers, After the in-lining function if the variable number which is going to use the register increases then they may create overhead on register variable resource utilization. This means that when the inline function body is substituted at the point of the function call, the total number of variables used by the function also gets inserted. So the number of registers going to be used for the variables will also get increased. So if after function inlining variable numbers increase drastically then it would surely cause overhead on register utilization. 2. If you use too many inline functions then the size of the binary executable file will be large, because of the duplication of the same code. 3. Too much inlining can also reduce your instruction cache hit rate, thus reducing the speed of instruction fetch from that of cache memory to that of primary memory. 4. The inline function may increase compile time overhead if someone changes the code inside the inline function then all the calling location has to be recompiled because the compiler would be required to replace all the code once again to reflect the changes, otherwise it will continue with old functionality. 5. Inline functions may not be useful for many embedded systems. Because in embedded systems code size is more important than speed. 6. Inline functions might cause thrashing because inlining might increase the size of the binary executable file. Thrashing in memory causes the performance of the computer to degrade. The following program demonstrates the use of the inline function.

Code:

CODE 1:=

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

int main()
{
    int a = 8, b = 3;

    // Addition operator
    cout << "a + b = " << (a + b) << endl;

    // Subtraction operator
    cout << "a - b = " << (a - b) << endl;

    // Multiplication operator
    cout << "a * b = " << (a * b) << endl;

    // Division operator
    cout << "a / b = " << (a / b) << endl;

    // Modulo operator
    cout << "a % b = " << (a % b) << endl;

    return 0;
}
```

Output

```
a + b = 11
a - b = 5
a * b = 24
a / b = 2
a % b = 2
```

CODE 2:=

```
// CPP Program to demonstrate the Relational Operators
#include <iostream>
using namespace std;

int main()
{
    int a = 6, b = 4;

    // Equal to operator
    cout << "a == b is " << (a == b) << endl;

    // Greater than operator
    cout << "a > b is " << (a > b) << endl;

    // Greater than or Equal to operator
    cout << "a >= b is " << (a >= b) << endl;
```

```
    //  Lesser than operator
    cout << "a < b is " << (a < b) << endl;

    // Lesser than or Equal to operator
    cout << "a <= b is " << (a <= b) << endl;

    // true
    cout << "a != b is " << (a != b) << endl;

    return 0;
}
```

Output
a == b is 0
a > b is 1
a >= b is 1
a < b is 0
a <= b is 0
a != b is 1

CODE 3:

```
include <iostream>
using namespace std;
inline int cube(int s) { return s * s * s; }
int main()
{
    cout << "The cube of 3 is: " << cube(3) << "\n";
    return 0;
}
```

Output
The cube of 3 is: 27

```
// C++ Program to demonstrate inline functions and classes
#include <iostream>

using namespace std;

class operation {
    int a, b, add, sub, mul;
    float div;

public:
    void get();
    void sum();
    void difference();
    void product();
    void division();
```

```
};
inline void operation ::get()
{
   cout << "Enter first value:";
   cin >> a;
   cout << "Enter second value:";
   cin >> b;
}
 inline void operation ::sum()
{
   add = a + b;
   cout << "Addition of two numbers: " << a + b << "\n";
}
 inline void operation ::difference()
{
   sub = a - b;
   cout << "Difference of two numbers: " << a - b << "\n";
}
 inline void operation ::product()
{
   mul = a * b;
   cout << "Product of two numbers: " << a * b << "\n";
}
 inline void operation ::division()
{
   div = a / b;
   cout << "Division of two numbers: " << a / b << "\n";
}
 int main()
{
   cout << "Program using inline function\n";
   operation s;
   s.get();
   s.sum();
   s.difference();
   s.product();
   s.division();
   return 0;
}
```

Output:

```
Enter first value: 45
Enter second value: 15
Addition of two numbers: 60
Difference of two numbers: 30
Product of two numbers: 675
Division of two numbers: 3
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

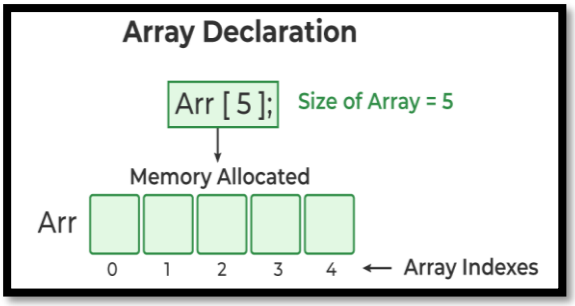
DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Operating System Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 02

Title : Programs on dealing with Arrays and Sorting of Array

Aim:	Write a Program to for matrix addition and sorting of array
Objective:	To implement matrix addition and sorting of array.
Theory :	**Arrays-in-C++** An Array is a collection of data of the same data type, stored at a contiguous memory location. Indexing of an array starts from 0. It means the first element is stored at the 0th index, the second at 1st, and so on. Elements of an array can be accessed using their indices. Once an array is declared its size remains constant throughout the program. An array can have multiple dimensions. The size of the array in bytes can be determined by the sizeof operator using which we can also find the number of elements in the array. We can find the size of the type of elements stored in an array by subtracting adjacent addresses.  Given two N x M matrices. Find a N x M matrix as the sum of given matrices each value at the sum of values of corresponding elements of the given two matrices. In this article, we will learn about the addition of two matrices. **Program for Addition of Two Matrices** $\begin{bmatrix} 0 & 4 \\ 1 & 3 \end{bmatrix} + \begin{bmatrix} 3 & 2 \\ 5 & 1 \end{bmatrix} = \begin{bmatrix} 0+3 & 4+2 \\ 1+5 & 3+1 \end{bmatrix} = \begin{bmatrix} 3 & 6 \\ 6 & 4 \end{bmatrix}$

Sorting an Array in C++

To sort an array in C++, we can simply use the std::sort() method provided by the STL, which takes two parameters, the first one points to the array where the sorting needs to begin, and the second parameter is the length up to which we want to sort the array.
The default sorting order is ascending order but we can also provide the custom comparator to sort in different order as the third parameter.
Syntax of std::sort()
Use the below syntax to sort an array in C++.

sort(arr, arr+n)

Here,

- **arr** is the name of the array which also acts as a pointer to the first element of the array.
- **n** is the size of the array.

Code:

```
include <iostream>
using namespace std;

int main()
{
   int r, c, a[100][100], b[100][100], sum[100][100], i, j;

   cout << "Enter number of rows (between 1 and 100): ";
   cin >> r;

   cout << "Enter number of columns (between 1 and 100): ";
   cin >> c;

   cout << endl << "Enter elements of 1st matrix: " << endl;

   // Storing elements of first matrix entered by user.
   for(i = 0; i < r; ++i)
     for(j = 0; j < c; ++j)
     {
        cout << "Enter element a" << i + 1 << j + 1 << " : ";
        cin >> a[i][j];
     }

   // Storing elements of second matrix entered by user.
   cout << endl << "Enter elements of 2nd matrix: " << endl;
   for(i = 0; i < r; ++i)
     for(j = 0; j < c; ++j)
     {
        cout << "Enter element b" << i + 1 << j + 1 << " : ";
        cin >> b[i][j];
     }

   // Adding Two matrices
```

```
    for(i = 0; i < r; ++i)
        for(j = 0; j < c; ++j)
            sum[i][j] = a[i][j] + b[i][j];

    // Displaying the resultant sum matrix.
    cout << endl << "Sum of two matrix is: " << endl;
    for(i = 0; i < r; ++i)
        for(j = 0; j < c; ++j)
        {
            cout << sum[i][j] << "  ";
            if(j == c - 1)
                cout << endl;
        }

    return 0;
}
```

Output

```
Enter number of rows (between 1 and 100): 2
Enter number of columns (between 1 and 100): 2

Enter elements of 1st matrix:
Enter element a11: -4
Enter element a12: 5
Enter element a21: 6
Enter element a22: 8

Enter elements of 2nd matrix:
Enter element b11: 3
Enter element b12: -9
Enter element b21: 7
Enter element b22: 2

Sum of two matrix is:
-1   -4
13   10
```

Sorting of array

```
#include <bits/stdc++.h>
using namespace std;
int main()
{
    int n;
    cout<<"Enter the size of array: "; cin>>n;
    int a[n];
    cout<<"\nEnter the elements: ";
    for(int i=0; i<n; i++) cin>>a[i];
     for(int i=0; i<n; i++)
    {
```

```
        for(int j=i+1; j<n; j++) { if(a[i]>a[j])
            {
                int temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    cout<<"\nArray after swapping: "
    for(int i=0; i<n; i++) cout<<a[i]<<" ";
    return 0;
}
```

Output:

Enter the size of array: 5

Enter the elements: 1 3 2 5 4

Array after swapping: 1 2 3 4 5

USING SORT FUNCTION

```
/ C++ Program to how to sort an array
#include <algorithm>
#include <iostream>
using namespace std;
int main()
{
        int arr[] = { 5, 4, 1, 2, 3 };
        int n = sizeof(arr) / sizeof(arr[0]);
       cout << "Array Before Sorting: ";
    for (int i = 0; i < n; i++)
     {
        cout << arr[i] << " ";
    }
    cout << endl;
    sort(arr, arr + n);
       cout << "Array After Sorting: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Output

Array Before Sorting: 5 4 1 2 3

Array After Sorting: 1 2 3 4 5

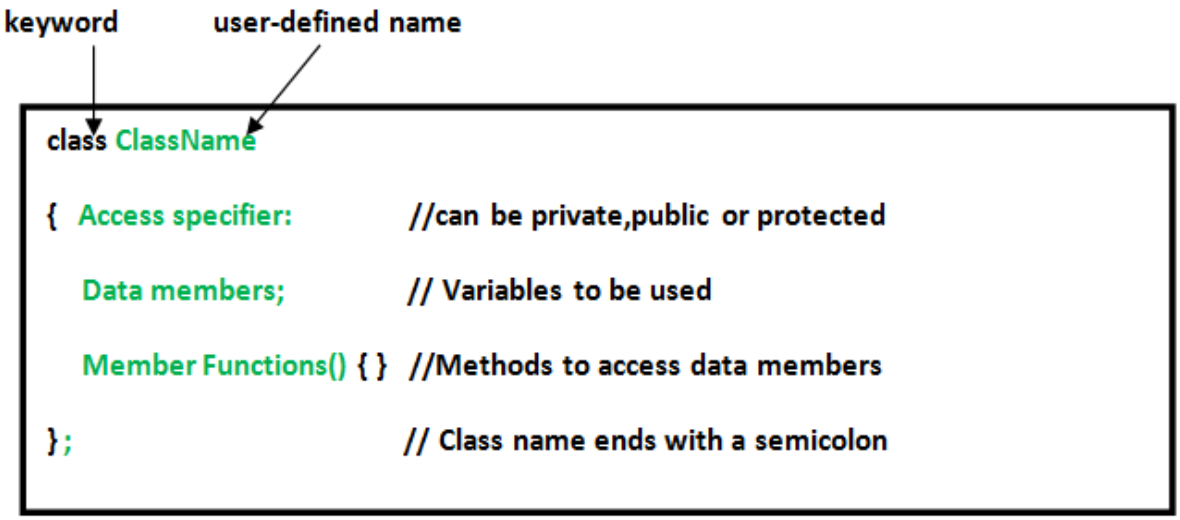
Subject : Operating System Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 03

Title : Programs on Classes and Objects

Aim:	Programs on Classes and Objects
Objective:	To learn how to create objects and classes.
Theory :	What is a Class in C++? **A class is a user-defined data type, which holds its own data members and member functions, which can be accessed and used by creating an instance of that class.** A C++ class is like a blueprint for an object. **For Example:** Consider the Class of **Cars**. There may be many cars with different names and brands but all of them will share some common properties like all of them will have *4 wheels, Speed Limit, Mileage range,* etc. So here, the Car is the class, and wheels, speed limits, and mileage are their properties. • A Class is a user-defined data type that has data members and member functions. • Data members are the data variables and member functions are the functions used to manipulate these variables together, these data members and member functions define the properties and behaviour of the objects in a Class. • In the above example of class *Car*, the data member will be *speed limit*, *mileage,* etc, and member functions can be *applying brakes*, *increasing speed,* etc. But we cannot use the class as it is. We first have to create an object of the class to use its features. An **Object** is an instance of a Class. When a class is defined, no memory is allocated but when it is instantiated (i.e. an object is created) memory is allocated. Defining Class in C++ A class is defined in C++ using the keyword c**lass** followed by the name of the class. The following is the syntax: **class ClassName** **{** *access_specifier:* *// Body of the class* **};** Here, the access specifier defines the level of access to the class's data members. **Example** class ThisClass { public:

```
    int var;     // data member
    void print() {          // member method
        cout << "Hello";
     }
};
```



What is an Object in C++?

When a class is defined, only the specification for the object is defined; no memory or storage is allocated. To use the data and access functions defined in the class, you need to create objects.

Syntax to Create an Object

We can create an object of the given class in the same way we declare the variables of any other inbuilt data type.

ClassName ObjectName;

Example

MyClass obj;

In the above statement, the object of MyClass with name obj is created.

Access Modifiers

In C++ classes, we can control the access to the members of the class using Access Specifiers. Also known as access modifier, they are the keywords that are specified in the class and all the members of the class under that access specifier will have particular access level.

In C++, there are 3 access specifiers that are as follows:

1. Public: Members declared as public can be accessed from outside the class.
2. Private: Members declared as private can only be accessed within the class itself.
3. Protected: Members declared as protected can be accessed within the class and by derived classes.

std::string class in C++

C++ has in its definition a way to represent a **sequence of characters as an object of the class**. This class is called std:: string. The string class stores the characters as a sequence of bytes with the functionality of allowing **access to the single-byte character**.

Operations on Strings

1) Input Functions

Function	Definition
getline()	This function is used to store a stream of characters as entered by the user in the object memory.
push_back()	This function is used to input a character at the end of the string.
pop_back()	This function is used to delete the last character from the string.

2) Capacity Functions

Function	Definition
capacity()	This function returns the capacity allocated to the string, which can be equal to or more than the size of the string. Additional space is allocated so that when the new characters are added to the string, the operations can be done efficiently.
resize()	This function changes the size of the string, the size can be increased or decreased.
length()	This function finds the length of the string.
shrink_to_fit()	This function decreases the capacity of the string and makes it equal to the minimum capacity of the string. This operation is useful to save additional memory if we are sure that no further addition of characters has to be made.

3) Iterator Functions

Function	Definition
begin()	This function returns an iterator to the beginning of the string.

end()	This function returns an iterator to the next to the end of the string.
rbegin()	This function returns a reverse iterator pointing at the end of the string.
rend()	This function returns a reverse iterator pointing to the previous of beginning of the string.
cbegin()	This function returns a constant iterator pointing to the beginning of the string, it cannot be used to modify the contents it points-to.
cend()	This function returns a constant iterator pointing to the next of end of the string, it cannot be used to modify the contents it points-to.
crbegin()	This function returns a constant reverse iterator pointing to the end of the string, it cannot be used to modify the contents it points-to.
crend()	This function returns a constant reverse iterator pointing to the previous of beginning of the string, it cannot be used to modify the contents it points-to.

4) Manipulating Functions:

Function	Definition
copy("char array", len, pos)	This function copies the substring in the target character array mentioned in its arguments. It takes 3 arguments, target char array, length to be copied, and starting position in the string to start copying.
swap()	This function swaps one string with another

Code:

```
 / C++ program to illustrate how create a simple class and  object
#include <iostream>
#include <string>
using namespace std;
class Person {
public:
   string name;
   int age;
```

```
  void introduce()
   {
      cout << "Hi, my name is " << name << " and I am "
          << age << " years old." << endl;
   }
};

int main()
{
   Person person1;
   person1.name = "Alice";
   person1.age = 30;
   person1.introduce();
   return 0;
}
```

Output

Hi, my name is Alice and I am 30 years old.

// C++ Program to demonstrate the working of begin(), end(), rbegin(), rend(), cbegin(), cend(), crbegin(), crend()

```
#include <iostream>
#include <string> // for string class
using namespace std;

int main()
{
    string str = "geeksforgeeks";
    std::string::iterator it;
   std::string::reverse_iterator it1;
   cout<<"Str:"<<str<<"\n";
   cout << "The string using forward iterators is : ";
   for (it = str.begin(); it != str.end(); it++){
      if(it == str.begin()) *it='G';
      cout << *it;
   }
   cout << endl;
   str = "geeksforgeeks";
   cout << "The reverse string using reverse iterators is " ": ";
   for (it1 = str.rbegin(); it1 != str.rend(); it1++){
      if(it1 == str.rbegin()) *it1='S';
      cout << *it1;
   }
   cout << endl;
     str = "geeksforgeeks";
  cout<<"The string using constant forward iterator is :";
  for(auto it2 = str.cbegin(); it2!=str.cend(); it2++){
    cout<<*it2;
```

```
  }
  cout<<"\n";

  str = "geeksforgeeks";
  cout<<"The reverse string using constant reverse iterator is :";
  for(auto it3 = str.crbegin(); it3!=str.crend(); it3++){
      cout<<*it3;
  }
  cout<<"\n";

    return 0;
}
```

Output
Str:geeksforgeeks
The string using forward iterators is : Geeksforgeeks
The reverse string using reverse iterators is : Skeegrofskeeg
The string using constant forward iterator is :geeksforgeeks
The reverse string using constant reverse iterator is :skeegrofskeeg

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Operating System Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 04

Title : Programs on Inheritance and Polymorphism.

Aim:	Write a program on Inheritance and Polymorphism using classes and objects
Objective:	Learn how to implement Inheritance and Polymorphism using classes and objects
Theory :	Inheritance in C++ 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 in C++. In this article, we will learn about inheritance in C++, its modes and types along with the information about how it affects different properties of the class.

Syntax of Inheritance in C++

```
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**: Specifies the access mode which can be either of private, public or protected. If neither is specified, private is taken as default.
- **base-class-name**: name of the base class.

```
class ABC : private XYZ {...}          // private derivation
class ABC : public XYZ {...}           // public derivation
class ABC : protected XYZ {...}      // protected derivation
class ABC: XYZ {...}                   // private derivation by default
```

Modes of Inheritance in C++

Mode of inheritance controls the access level of the inherited members of the base class in the derived class. In C++, there are 3 modes of inheritance:

- **Public Mode**
- **Protected Mode**
- **Private Mode**

Public Inheritance 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.

Example:

```
class ABC : public XYZ {...}           // public derivation
```

Protected Inheritance 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.

Example:

```
class ABC : protected XYZ {...}      // protected derivation
```

Private Inheritance 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. They can only be accessed by the member functions of the derived class.

Private mode is the default mode that is applied when we don't specify any mode.

Example:

```
class ABC : private XYZ {...}          // private derivation
class ABC: XYZ {...}                   // private derivation by default
```

Types Of Inheritance in C++

The inheritance can be classified on the basis of the relationship between the derived class and the base class. In C++, we have 5 types of inheritances:

1. **Single inheritance**

2. **Multilevel inheritance**
3. **Multiple inheritance**
4. **Hierarchical inheritance**
5. **Hybrid inheritance**

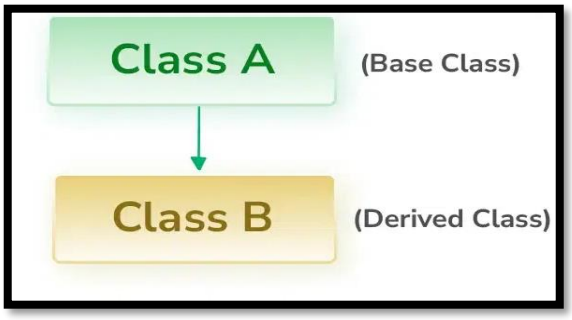
1. Single Inheritance

In single inheritance, a class is allowed to inherit from only one class. i.e. one base class is inherited by one derived class only.

Syntax

```
class subclass_name : access_mode base_class
{
  //  body of subclass
};
```

Example:




```
Class A
{
... .. ...
};
class B: public A
{
... .. ...
};
```

2. Multiple Inheritance

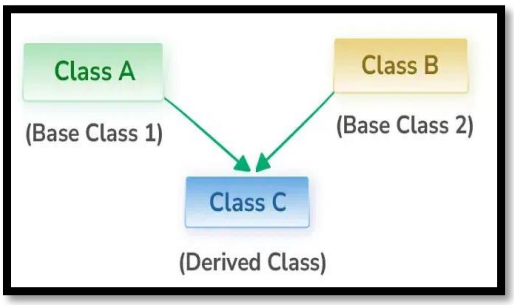
Multiple Inheritance is a feature of C++ where a class can inherit from more than one class. i.e one **subclass** is inherited from more than one **base class**.

Syntax

```
class subclass_name : access_mode base_class1, access_mode base_class2, ....
{
  // body of subclass
};
```

Here, the number of base classes will be separated by a comma (‘, ‘) and the access mode for every base class must be specified and can be different.

Example:



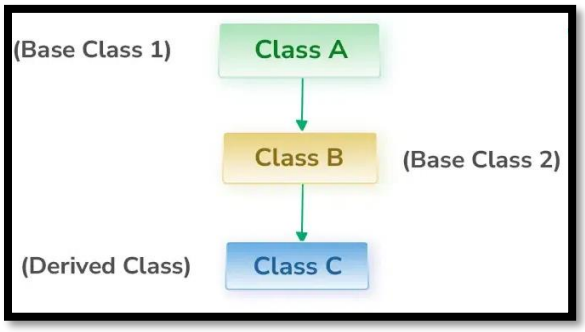

```
class B
{
... .. ...
};
class C
{
... .. ...
};
class A: public B, public C
{
... ... ...
};
```

3. Multilevel Inheritance

In this type of inheritance, a derived class is created from another derived class and that derived class can be derived from a base class or any other derived class. There can be any number of levels.

Syntax

```
class derived_class1: access_specifier base_class
{
... .. ...
}
class derived_class2: access_specifier derived_class1
{
... .. ...
}
.....
```




```
class C
{
... .. ...
};
```

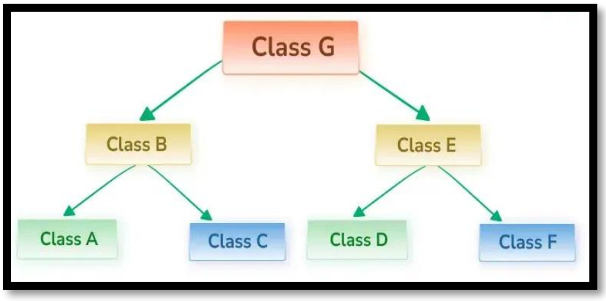
```
class B : public C
{
... .. ...
};
class A: public B
{
... ... ...
};
```

4. Hierarchical Inheritance

In this type of inheritance, more than one subclass is inherited from a single base class. i.e. more than one derived class is created from a single base class.

Syntax

```
class derived_class1: access_specifier base_class
{
... .. ...
}
class derived_class2: access_specifier base_class
{
... .. ...
}
```

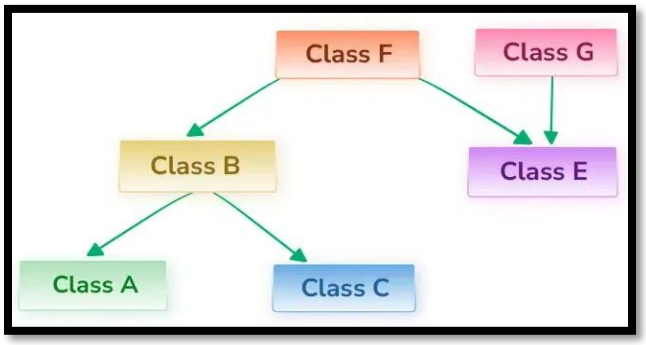



```
class A
{
   // body of the class A.
}
class B : public A
{
   // body of class B.
}
class C : public A
{
   // body of class C.
}
class D : public A
{
   // body of class D.
}
```

5. Hybrid Inheritance

Hybrid Inheritance is implemented by combining more than one type of inheritance. For example: Combining Hierarchical inheritance and Multiple Inheritance will create hybrid inheritance in C++

There is no particular syntax of hybrid inheritance. We can just combine two of the above inheritance types.



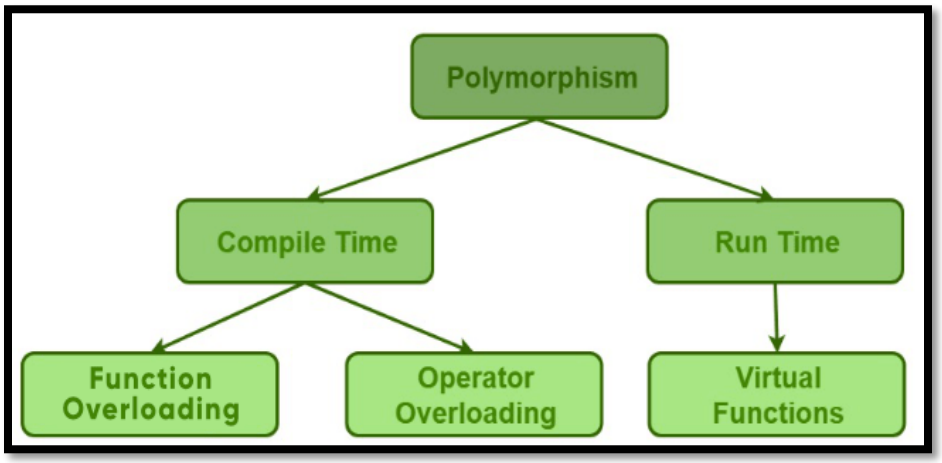

```
class F
{
... .. ...
}
class G
{
... .. ...
}
class B : public F
{
... .. ...
}
class E : public F, public G
{
... .. ...
}
class A : public B {
... .. ...
}
class C : public B {
... .. ...
}
```

C++ Polymorphism

Polymorphism as the ability of a message to be displayed in more than one form. A real-life example of polymorphism is a person who at the same time can have different characteristics. A man at the same time is a father, a husband, and an employee. So the same person exhibits different behavior in different situations. This is called polymorphism. Polymorphism is considered one of the important features of Object-Oriented Programming.

Types of Polymorphism

- **Compile-time Polymorphism**
- **Runtime Polymorphism**



1. Compile-Time Polymorphism

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

A. Function Overloading

When there are multiple functions with the same name but different parameters, then the functions are said to be **overloaded,** hence this is known as Function Overloading. Functions can be overloaded by **changing the number of arguments** or/and **changing the type of arguments**. In simple terms, it is a feature of object-oriented programming providing many functions that have the same name but distinct parameters when numerous tasks are listed under one function name. There are certain Rules of Function Overloading that should be followed while overloading a function.

B. Operator Overloading

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 make use of the addition operator (+) for string class to concatenate two strings. We know that the task of this operator is to add two operands. So a single operator '+', when placed between integer operands, adds them and when placed between string operands, concatenates them.

2. Runtime Polymorphism

This type of polymorphism is achieved by **Function Overriding**. Late binding and dynamic polymorphism are other names for runtime polymorphism. The function call is resolved at runtime in runtime polymorphism. In contrast, with compile time polymorphism, the compiler determines which function call to bind to the object after deducing it at runtime.

A. Function Overriding

Function Overriding occurs when a derived class has a definition for one of the member functions of the base class. That base function is said to be overridden.

B.Virtual Function

A virtual function is a member function that is declared in the base class using the keyword virtual and is re-defined (Overridden) in the derived class.

Some Key Points About Virtual Functions:

- Virtual functions are Dynamic in nature.
- They are defined by inserting the keyword "**virtual**" inside a base class and are always declared with a base class and overridden in a child class
- A virtual function is called during Runtime

Code:

```
// C++ program to demonstrate how to inherit a class
#include <iostream>
```

```
using namespace std;

// Base class that is to be inherited
class Parent {
public:
    // base class members
    int id_p;
    void printID_p()
    {
        cout << "Base ID: " << id_p << endl;
    }
};

// Sub class or derived publicly inheriting from Base
// Class(Parent)
class Child : public Parent {
public:
    // derived class members
    int id_c;
    void printID_c()
    {
        cout << "Child ID: " << id_c << endl;
    }
};

// main function
int main()
{
    // creating a child class object
    Child obj1;

    // An object of class child has all data members
    // and member functions of class parent
    // so we try accessing the parents method and data from
    // the child class object.
    obj1.id_p = 7;
    obj1.printID_p();

    // finally accessing the child class methods and data
    // too
    obj1.id_c = 91;
    obj1.printID_c();
```

```
    return 0;
}
```

Output

```
Base ID: 7
Child ID: 91
```

```
// C++ program to illustrate how to access the inherited
// members of the base class in derived class
#include <iostream>
using namespace std;

// Base class
class Base {
public:
    // data member
    int publicVar;

    // member method
    void display()
    {
        cout << "Value of publicVar: " << publicVar;
    }
};

// Derived class
class Derived : public Base {
public:
    // Function to display inherited member
    void displayMember()
    {
        // accessing public base class member method
        display();
    }

    // Function to modify inherited member
    void modifyMember(int pub)
    {
        // Directly modifying public member
        publicVar = pub;
    }
```

```
};

int main()
{
   // Create an object of Derived class
   Derived obj;

   // Display the initial values of inherited member
   obj.modifyMember(10);

   // Display the modified values of inherited member
   obj.displayMember();

   return 0;
}
```

Output

Value of publicVar: 10

```
/ C++ program to illustrate the implementation of Hybrid Inheritance
#include <iostream>
using namespace std;

// base class
class Vehicle {
public:
   Vehicle() { cout << "This is a Vehicle\n"; }
};

// base class
class Fare {
public:
   Fare() { cout << "Fare of Vehicle\n"; }
};

// first sub class
class Car : public Vehicle {
  public:
  Car() { cout << "This Vehical is a Car\n"; }
};

// second sub class
```

```
class Bus : public Vehicle, public Fare {
  public:
  Bus() { cout << "This Vehicle is a Bus with Fare\n"; }
};

// main function
int main()
{
   // Creating object of sub class will
   // invoke the constructor of base class.
   Bus obj2;
   return 0;
}
```

Output

```
This is a Vehicle
Fare of Vehicle
This Vehicle is a Bus with Fare
```

```
// C++ program to demonstrate compile time function
// overriding

#include <iostream>
using namespace std;

class Parent {
public:
   void GeeksforGeeks_Print()
   {
      cout << "Base Function" << endl;
   }
};

class Child : public Parent {
public:
   void GeeksforGeeks_Print()
   {
      cout << "Derived Function" << endl;
   }
};

int main()
```

```
{
    Child Child_Derived;
    Child_Derived.GeeksforGeeks_Print();
    return 0;
}
```

Output

Derived Function

```
// Base class
class Animal {
  public:
    void animalSound() {
      cout << "The animal makes a sound \n";
    }
};

// Derived class
class Pig : public Animal {
  public:
    void animalSound() {
      cout << "The pig says: wee wee \n";
    }
};
// Derived class
class Dog : public Animal {
  public:
    void animalSound() {
      cout << "The dog says: bow wow \n";
    }
};
int main() {
  Animal myAnimal;
  Pig myPig;
  Dog myDog;

  myAnimal.animalSound();
  myPig.animalSound();
  myDog.animalSound();
  return 0;

}
```

Subject : Operating System Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 05

Title : Programs on Operator Overloading .

Aim:	Write a programs on Operator Overloading
Objective:	
Theory :	Operator Overloading in C++ C++ has the ability to provide the operators with a special meaning for a data type, this ability is known as operator overloading. Operator overloading is a compile-time polymorphism. 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 integers, etc. Example: int a; float b,sum; sum = a + b; Syntax for C++ Operator Overloading The syntax for overloading an operator is similar to that of function with the addition of the operator keyword followed by the operator symbol. returnType operator symbol (arguments) { ... . } Here, returnType - the return type of the function operator - a special keyword symbol - the operator we want to overload (+, <, -, ++, etc.) arguments - the arguments passed to the function 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.

Can We Overload All Operators?
Almost all operators can be overloaded except a few. Following is the list of operators that cannot be overloaded.

sizeof
typeid
Scope resolution (::)
Class member access operators (.(dot), .* (pointer to member operator))
Ternary or conditional (?:)

Operators that can be Overloaded in C++
We can overload

- Unary operators
- Binary operators
- Special operators ([ ], (), etc)

But, among them, there are some operators that cannot be overloaded. They are

- Scope resolution operator
- Member selection operator
- Member selection through *

Pointer to a member variable

- Conditional operator
- Sizeof operator sizeof()

Operators that can be overloaded	Examples
Binary Arithmetic	+, -, *, /, %
Unary Arithmetic	+, -, ++, —
Assignment	=, +=,*=, /=,-=, %=
Bitwise	& , | , << , >> , ~ , ^
De-referencing	(->)
Dynamic memory allocation, De-allocation	New, delete
Subscript	[ ]
Function call	()
Logical	&, | |, !

	Relational >, < , = =, <=, >=

Important Points about Operator Overloading

1) For operator overloading to work, at least one of the operands must be a user-defined class object.

2) Assignment Operator: Compiler automatically creates a default assignment operator with every class. The default assignment operator does assign all members of the right side to the left side and works fine in most cases (this behavior is the same as the copy constructor). See this for more details.

3) Conversion Operator: We can also write conversion operators that can be used to convert one type to another type. Overloaded conversion operators must be a member method. Other operators can either be the member method or the global method.

4) Any constructor that can be called with a single argument works as a conversion constructor, which means it can also be used for implicit conversion to the class being constructed.

Code:

```
/ C++ Program to Demonstrate  Operator Overloading
#include <iostream>
using namespace std;

class Complex {
private:
   int real, imag;

public:
   Complex(int r = 0, int i = 0)
   {
      real = r;
      imag = i;
   }

   // This is automatically called when '+' is used with
   // between two Complex objects
   Complex operator+(Complex const& obj)
   {
      Complex res;
      res.real = real + obj.real;
      res.imag = imag + obj.imag;
      return res;
   }
   void print() { cout << real << " + i" << imag << '\n'; }
};

int main()
{
   Complex c1(10, 5), c2(2, 4);
   Complex c3 = c1 + c2;
```

```
    c3.print();
}
```

Output

```
12 + i9
```

```
// C++ program to demonstrate operator overloading
// using dot operator
#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
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Operating System Lab **Subject Code :** BTCOL306

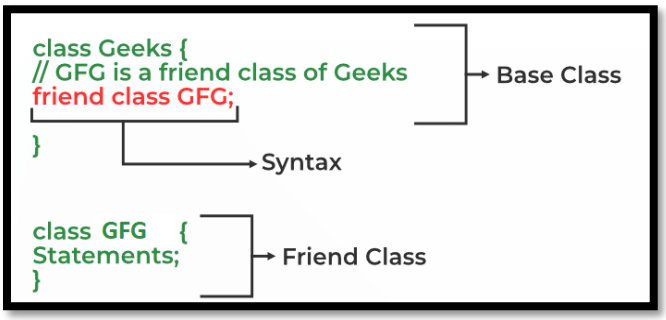
Class: S. Y. Comp. Engg. **Expt. No. :** 06

Title : Programs on Friend Function.

Aim:	Write a programs on Friend Function.
Objective:	
Theory :	Friend Class and Function in C++

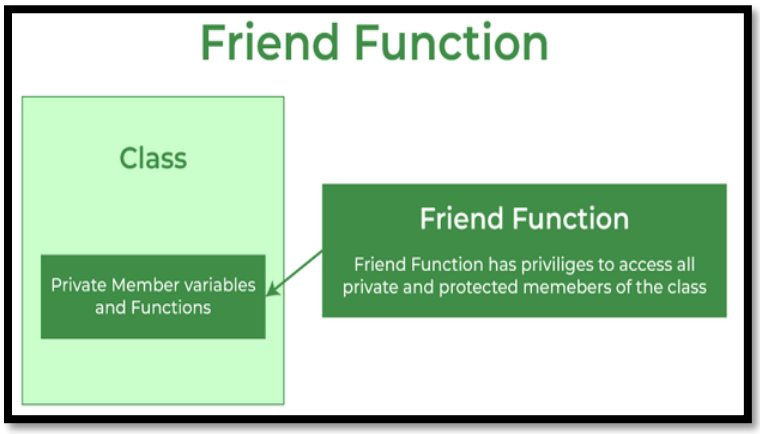
A friend class can access private and protected members of other classes in which it is declared as a friend. It is sometimes useful to allow a particular class to access private and protected members of other classes. For example, a LinkedList class may be allowed to access private members of Node.
We can declare a friend class in C++ by using the friend keyword.
Syntax:

```
friend class class_name;    // declared in the base class
```



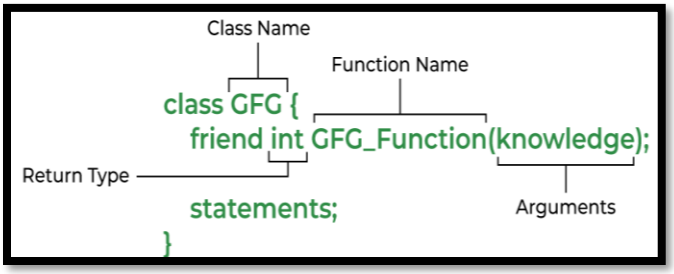
Friend Function
Like a friend class, a friend function can be granted special access to private and protected members of a class in C++. They are not the member functions of the class but can access and manipulate the private and protected members of that class for they are declared as friends.
A friend function can be:

1. A global function
2. A member function of another class



Syntax:

```
friend return_type function_name (arguments);    // for a global function
        or
friend return_type class_name::function_name (arguments);    // for a member function of another class
```



Features of Friend Functions

- A friend function is a special function in C++ that in spite of not being a member function of a class has the privilege to access the private and protected data of a class.
- A friend function is a non-member function or ordinary function of a class, which is declared as a friend using the keyword “friend” inside the class. By declaring a function as a friend, all the access permissions are given to the function.
- The keyword “friend” is placed only in the function declaration of the friend function and not in the function definition or call.
- A friend function is called like an ordinary function. It cannot be called using the object name and dot operator. However, it may accept the object as an argument whose value it wants to access.
- A friend function can be declared in any section of the class i.e. public or private or protected.

Advantages of Friend Functions

- A friend function is able to access members without the need of inheriting the class.
- The friend function acts as a bridge between two classes by accessing their private data.
- It can be used to increase the versatility of overloaded operators.
- It can be declared either in the public or private or protected part of the class.

Disadvantages of Friend Functions

	• Friend functions have access to private members of a class from outside the class which violates the law of data hiding. • Friend functions cannot do any run-time polymorphism in their members. • **Important Points About Friend Functions and Classes** 1. Friends should be used only for limited purposes. Too many functions or external classes are declared as friends of a class with protected or private data access lessens the value of encapsulation of separate classes in object-oriented programming. 2. Friendship is not mutual. If class A is a friend of B, then B doesn't become a friend of A automatically. 3. Friendship is not inherited. 4. The concept of friends is not in Java.
Code:	**// C++ Program to demonstrate the functioning of a friend class** #include <iostream> using namespace std; class GFG { private: int private_variable; protected: int protected_variable; public: GFG() { private_variable = 10; protected_variable = 99; } // friend class declaration friend class F; }; // Here, class F is declared as a // friend inside class GFG. Therefore, // F is a friend of class GFG. Class F // can access the private members of // class GFG. class F { public: void display(GFG& t) { cout << "The value of Private Variable = " << t.private_variable << endl; cout << "The value of Protected Variable = " << t.protected_variable; }

```
};

// Driver code
int main()
{
    GFG g;
    F fri;
    fri.display(g);
    return 0;
}
```

Output

```
The value of Private Variable = 10

The value of Protected Variable = 99
```

// C++ program to create a global function as a friend function of some class

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

class base {
private:
    int private_variable;

protected:
    int protected_variable;

public:
    base()
    {
        private_variable = 10;
        protected_variable = 99;
    }

      // friend function declaration
    friend void friendFunction(base& obj);
};


// friend function definition
void friendFunction(base& obj)
{
    cout << "Private Variable: " << obj.private_variable
         << endl;
    cout << "Protected Variable: " << obj.protected_variable;
}

// driver code
int main()
{
```

```
    base object1;
    friendFunction(object1);

    return 0;
}
```

Output

```
Private Variable: 10

Protected Variable: 99
```

/ C++ program to create a member function of another class as a friend function

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

class base; // forward definition needed
// another class in which function is declared
class anotherClass {
public:
    void memberFunction(base& obj);
};

// base class for which friend is declared
class base {
private:
    int private_variable;

protected:
    int protected_variable;

public:
    base()
    {
        private_variable = 10;
        protected_variable = 99;
    }

    // friend function declaration
    friend void anotherClass::memberFunction(base&);
};

// friend function definition
void anotherClass::memberFunction(base& obj)
{
    cout << "Private Variable: " << obj.private_variable
         << endl;
    cout << "Protected Variable: " << obj.protected_variable;
}

// driver code
```

```
int main()
{
   base object1;
   anotherClass object2;
   object2.memberFunction(object1);

   return 0;
}
```

Output

```
Private Variable: 10

Protected Variable: 99
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Operating System Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 07

Title : Programs on file handling and stream manipulation.

Aim:	Write programs on file handling and stream manipulation.
Objective:	
Theory :	**File Handling through C++ Classes** File handling is used to store data permanently in a computer. Using file handling we can store our data in secondary memory (Hard disk). How to achieve the File Handling For achieving file handling we need to follow the following steps:- STEP 1-Naming 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”. **Classes for File stream operations :-** 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. 1. ios:- • ios stands for input output stream. • This class is the base class for other classes in this class hierarchy. • This class contains the necessary facilities that are used by all the other derived classes for input and output operations.

2. istream:-

- istream stands for input stream.
- This class is derived from the class 'ios'.
- This class handle input stream.
- The extraction operator(>>) is overloaded in this class to handle input streams from files to the program execution.
- This class declares input functions such as get(), getline() and read().

3. ostream:-

- ostream stands for output stream.
- This class is derived from the class 'ios'.
- This class handle output stream.
- The insertion operator(<<) is overloaded in this class to handle output streams to files from the program execution.
- This class declares output functions such as put() and write().

4. streambuf:-

- This class contains a pointer which points to the buffer which is used to manage the input and output streams.

5. fstreambase:-

- This class provides operations common to the file streams. Serves as a base for fstream, ifstream and ofstream class.
- This class contains open() and close() function.

6. ifstream:-

- This class provides input operations.
- It contains open() function with default input mode.
- Inherits the functions get(), getline(), read(), seekg() and tellg() functions from the istream.

7. ofstream:-

- This class provides output operations.
- It contains open() function with default output mode.
- Inherits the functions put(), write(), seekp() and tellp() functions from the ostream.

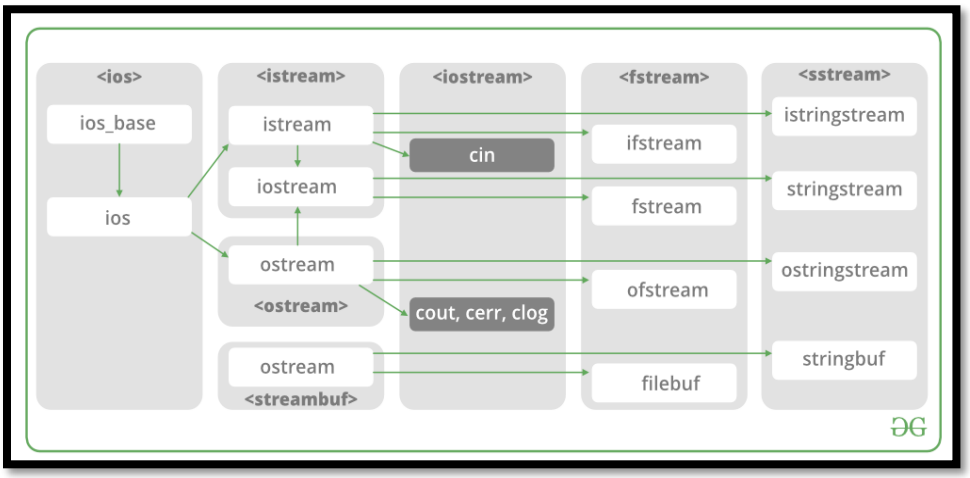
8. fstream:-

- This class provides support for simultaneous input and output operations.
- Inherits all the functions from istream and ostream classes through iostream.

9. filebuf:-

- Its purpose is to set the file buffers to read and write.
- We can also use file buffer member function to determine the length of the file.

In C++, files are mainly dealt by using three classes fstream, ifstream, ofstream available in fstream headerfile.
ofstream: Stream class to write on files
ifstream: Stream class to read from files
fstream: Stream class to both read and write from/to files.



Modes :

Member Constant	Stands For	Access
in *	input	File open for reading: the internal stream buffer supports input operations.
out	output	File open for writing: the internal stream buffer supports output operations.
binary	binary	Operations are performed in binary mode rather than text.
ate	at end	The output position starts at the end of the file.
app	append	All output operations happen at the end of the file, appending to its existing contents.
trunc	truncate	Any contents that existed in the file before it is open are discarded.

Default Open Modes :

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

	stream manipulation Manipulators' are operators used to format the output in C++. Stream Manipulators are functions particularly developed to be used in combination with insertion and extraction of stream object operators. endl - This manipulator invokes a newline character. The output stream is also flushed. showpoint / noshowpoint - This manipulator determines whether or not the decimal point to be included in the floating-point representation. setprecision - This manipulator changes floating-point precision. setw - This manipulator will change the width of the subsequent input/output field. setfill - If a value does not fill a field in its entirety, the character stated in this manipulator argument is used to fill the fields.
Code:	*** File Handling with C++ using ifstream & ofstream class object*/** **/* To write the Content in File*/** **/* Then to read the content of file*/** #include <iostream> /* fstream header file for ifstream, ofstream, fstream classes */ #include <fstream> using namespace std; // Driver Code int main() { // Creation of ofstream class object ofstream fout; string line; // by default ios::out mode, automatically deletes // the content of file. To append the content, open in ios:app // fout.open("sample.txt", ios::app) fout.open("sample.txt"); // Execute a loop If file successfully opened while (fout) { // Read a Line from standard input getline(cin, line);

```
        // Press -1 to exit
        if (line == "-1")
            break;

        // Write line in file
        fout << line << endl;
    }

    // Close the File
    fout.close();

    // Creation of ifstream class object to read the file
    ifstream fin;

    // by default open mode = ios::in mode
    fin.open("sample.txt");

    // Execute a loop until EOF (End of File)
    while (getline(fin, line)) {

        // Print line (read from file) in Console
        cout << line << endl;
    }

    // Close the file
    fin.close();

    return 0;
}
// Q: write a single file handling program in c++ to reading
// and writing data on a file.

// Include necessary header files
#include <fstream>
#include <iostream>
#include <string>

// Use the standard namespace
using namespace std;

int main()
{
    // Create an output file stream object
    ofstream fout;
    // Open a file named "NewFile.txt" for writing
    fout.open("NewFile.txt");

    // Check if the file opened successfully
    if (!fout) {
```

```
        // Print an error message if the file couldn't be
        // opened
        cerr << "Error opening file!" << endl;
        // Return 1 to indicate failure
        return 1;
    }

    // Declare a string variable to hold each line of text
    string line;
    // Initialize a counter to limit input to 5 lines
    int i = 0;

    // Prompt the user to enter 5 lines of text
    cout << "Enter 5 lines of text:" << endl;
    // Loop to read 5 lines of input from the user
    while (i < 5) {
        // Read a line of text from standard input
        getline(cin, line);
        // Write the line of text to the file
        fout << line << endl;
        // Increment the counter
        i += 1;
    }

    // Close the file after writing
    fout.close();

    // Print a success message
    cout << "Text successfully written to NewFile.txt"
         << endl;

    // Return 0 to indicate successful execution
    return 0;
}
```

Output

```
Enter 5 lines of text:
Hi, Welcome to GeeksforGeeks
Learn to code
C++
Java
Python
Text successfully written to NewFile.txt
```

Shri Vile Parle Kelavani Mandal's

INSTITUTE OF TECHNOLOGY

DHULE (M.S.)

DEPARMENT OF COMPUTER ENGINEERING

Subject : Operating System Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 08

Title :. Programs on Dynamic Memory Management

Aim:	
Objective:	
Theory :	**C++ Program Memory Management** When we run a C++ program on our machine, it requires some space to store its instructions (statements), local variables, global variables, and various other functions in C++. This space required to run a C++ Program is known as memory in computers. There are two types of memory in our system, Static Memory and Dynamic Memory. • Static Memory: It is a constant space allocated by the operating system during the compile time of a C++ program and it internally uses stack data structure to manage the static memory allocation. We can't reallocate the space consumed by the program until its execution is over. • Dynamic Memory: It is the memory that can be allocated or de-allocated by the operating system during the run-time of a C++ program. It is more efficient than static memory because we can de-allocate and reuse our memory during the run-time of our program. The memory used by a C++ program can be divided further into four parts: 1. Run-time Stack (Static Memory) 2. Static Data Memory (for Global and Static Variables) 3. Instructions / Statements (Static Memory) 4. Heap (Dynamic Memory) **Types of Memory Allocation** 1. Static Memory Allocation in C++ (Compile-time Memory Allocation) • When memory is allocated at compile-time, it is referred to as Static Memory Allocation. • A fixed space is allocated for the local variables, function calls, and local statements, that can not be changed during the execution of the program.

- We can not allocate or de-allocate a memory block once the execution of the program starts.
- We can't re-use the static memory while the program is running. As a result, it is less effective.

2. Dynamic Memory Allocation in C++ (Run-time Memory Allocation)

- When memory is allocated or de-allocated during run-time, it is referred to as Dynamic Memory Allocation in C++.
- A variable space is allocated that can be changed during the execution of the program.
- We use dynamic/heap memory to allocate and de-allocate a block of memory during the execution of the program using new and delete operators.
- We can re-use our heap memory during the run-time of our program. As a result, it is highly effective.

How is Memory Allocated/De-Allocated in C++?

In C Language, before allocating or de-allocating memory dynamically (at run-time), we have to include <stdlib.h> header file to use the library functions like malloc(), calloc(), realloc() and free().

In C++ Language, **new** and **delete** operators are pre-defined in the C++ Standard Library and don't require to include any library file for run-time allocation and de-allocation of memory blocks. Although we can use malloc(), calloc(), and other functions in C++ as well by adding the <stdlib.h> header file because of the backward compatibility of C++ with C Language. This article explains the **new** and **delete** operators of the C++ language.
Visit Dynamic Memory Allocation in C - Scaler Topics to know more about DMA functions in C Language.

new operator
The new operator denotes a request for memory allocation on the Free Store. If sufficient memory is available, a new operator initializes the memory and returns the address of the newly allocated and initialized memory to the pointer variable.
Syntax to use new operator

```
pointer-variable = new data-type;
```

Here, the pointer variable is the pointer of type data-type. Data type could be any built-in data type including array or any user-defined data type including structure and class. Example:

```
// 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;
```

delete operator

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

Syntax:

```
// Release memory pointed by pointer-variable
delete pointer-variable;
```

Here, the pointer variable is the pointer that points to the data object created by new.

Examples:

```
delete p;
delete q;
```

To free the dynamically allocated array pointed by pointer variable, use the following form of delete:

```
// Release block of memory
// pointed by pointer-variable
delete[] pointer-variable;

Example:
// It will free the entire array
// pointed by p.
  delete[] p;
```

Code:

```
// C++ program to demonstrate how to create dynamic variable using new
#include <iostream>
#include <memory>
using namespace std;

int main()
{
   // pointer to store the address returned by the new
   int* ptr;
   // allocating memory for integer
   ptr = new int;

   // assigning value using dereference operator
   *ptr = 10;

   // printing value and address
   cout << "Address: " << ptr << endl;
   cout << "Value: " << *ptr;

   return 0;
}
```

Output

```
Address: 0x162bc20

Value: 10
```

```
// C++ program to illustrate how to initialize a dynamic variable with allocation
#include <iostream>
```

```
#include <memory>

using namespace std;

// Custom data type with constructor to take initial value
struct cust {
    int p;
    cust(int q)
        : p(q)
    {
    }
    cust() = default;
};

int main()
{
    // creating inbuit data types with initial value
    int* p = new int(25);
    float* q = new float(75.25);

    // Works fine, doesn't require constructor
    cust* var1 = new cust;

    // OR

    // Works fine, doesn't require constructor
    var1 = new cust();

    // Notice error if you comment this line
    cust* var = new cust(25);

    cout << *p << " " << *q << " " << var->p;
    return 0;
}
```

Output
25 75.25 25

// C++ program to illustrate dynamic allocation and deallocation of memory using new and delete

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

int main()
{
    // Pointer initialization to null
    int* p = NULL;

    // Request memory for the variable
```

```
    // using new operator
    p = new (nothrow) int;
    if (!p)
       cout << "allocation of memory failed\n";
    else {
       // Store value at allocated address
       *p = 29;
       cout << "Value of p: " << *p << endl;
    }

    // Request block of memory
    // using new operator
    float* r = new float(75.25);

    cout << "Value of r: " << *r << endl;

    // Request block of memory of size n
    int n = 5;
    int* q = new (nothrow) int[n];

    if (!q)
       cout << "allocation of memory failed\n";
    else {
       for (int i = 0; i < n; i++)
          q[i] = i + 1;

       cout << "Value store in block of memory: ";
       for (int i = 0; i < n; i++)
          cout << q[i] << " ";
    }

    // freed the allocated memory
    delete p;
    delete r;

    // freed the block of allocated memory
    delete[] q;

    return 0;
}
```

Output

```
Value of p: 29
Value of r: 75.25
Value store in block of memory: 1 2 3 4 5
```

Subject : Operating System Lab **Subject Code :** BTCOL306

Class: S. Y. Comp. Engg. **Expt. No. :** 09

Title : Programs on Exception Handling.

Aim:	
Objective:	
Theory :	Exception Handling in C++

What is a C++ Exception?
An exception is an unexpected problem that arises during the execution of a program our program terminates suddenly with some errors/issues. Exception occurs during the running of the program (runtime).

Types of C++ Exception
There are two types of exceptions in C++

1. Synchronous: Exceptions that happen when something goes wrong because of a mistake in the input data or when the program is not equipped to handle the current type of data it's working with, such as dividing a number by zero.
2. Asynchronous: Exceptions that are beyond the program's control, such as disc failure, keyboard interrupts, etc.

C++ try and catch
C++ provides an inbuilt feature for Exception Handling. It can be done using the following specialized keywords: try, catch, and throw with each having a different purpose.

Syntax of try-catch in C++

```
try {
    // Code that might throw an exception
    throw SomeExceptionType("Error message");
 }
catch( ExceptionName e1 )  {
    // catch block catches the exception that is thrown from try block
 }
```

1. try in C++
The try keyword represents a block of code that may throw an exception placed inside the try block. It's followed by one or more catch blocks. If an exception occurs, try block throws that exception.
2. catch in C++

The catch statement represents a block of code that is executed when a particular exception is thrown from the try block. The code to handle the exception is written inside the catch block.
3. throw in C++
An exception in C++ can be thrown using the throw keyword. When a program encounters a throw statement, then it immediately terminates the current function and starts finding a matching catch block to handle the thrown exception.

Why do we need Exception Handling in C++?
The following are the main advantages of exception handling over traditional error handling:

1. Separation of Error Handling Code from Normal Code: There are always if-else conditions to handle errors in traditional error handling codes. 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.

Properties of Exception Handling in C++
Property 1
There is a special catch block called the 'catch-all' block, written as catch(…), that can be used to catch all types of exceptions.

Property 2
Implicit type conversion doesn't happen for primitive types.

Property 3
If an exception is thrown and not caught anywhere, the program terminates abnormally.

Property 4
Unlike Java, in C++, all exceptions are unchecked, i.e., the compiler doesn't check whether an exception is caught or not (See this for details). So, it is not necessary to specify all uncaught exceptions in a function declaration. However,exception-handling it's a recommended practice to do so.

Property 5
In C++, try/catch blocks can be nested. Also, an exception can be re-thrown using "throw; ".

Property 6

When an exception is thrown, all objects created inside the enclosing try block are destroyed before the control is transferred to the catch block.

Limitations of Exception Handling in C++

The exception handling in C++ also have few limitations:

- Exceptions may break the structure or flow of the code as multiple invisible exit points are created in the code which makes the code hard to read and debug.
- If exception handling is not done properly can lead to resource leaks as well.
- It's hard to learn how to write Exception code that is safe.
- There is no C++ standard on how to use exception handling, hence many variations in exception-handling practices exist.

Code:

```
// C++ program to demonstate the use of try,catch and throw in exception handling.
 #include <iostream>
#include <stdexcept>
using namespace std;
 int main()
{

   // try block
   try {
      int numerator = 10;
      int denominator = 0;
      int res;

      // check if denominator is 0 then throw runtime
      // error.
      if (denominator == 0) {
         throw runtime_error(
            "Division by zero not allowed!");
      }

      // calculate result if no exception occurs
      res = numerator / denominator;
      //[printing result after division
      cout << "Result after division: " << res << endl;
   }
   // catch block to catch the thrown exception
   catch (const exception& e) {
      // print the exception
      cout << "Exception " << e.what() << endl;
   }

   return 0;
}
```

Output

Exception Division by zero not allowed!

/ C++ program to demonstate the use of try,catch and throw in exception handling.

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

int main()
{
   int x = -1;

   // Some code
   cout << "Before try \n";

   // try block
   try {
      cout << "Inside try \n";
      if (x < 0) {
         // throwing an exception
         throw x;
         cout << "After throw (Never executed) \n";
      }
   }

   // catch block
   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)

Shri Vile Parle Kelavani Mandal's
INSTITUTE OF TECHNOLOGY
DHULE (M.S.)
DEPARMENT OF COMPUTER ENGINEERING

Subject : Operating System Lab Subject Code : BTCOL306

Class: S. Y. Comp. Engg. Expt. No. : 10

Title : Programs on generic programming using templates..

Aim:	
Objective:	
Theory :	**Generics in C++** Generics is the idea to allow type (Integer, String, … etc and user-defined types) to be a parameter to methods, classes and interfaces. For example, classes like an array, map, etc, which can be used using generics very efficiently. We can use them for any type. The method of Generic Programming is implemented to increase the efficiency of the code. Generic Programming enables the programmer to write a general algorithm which will work with all data types. It eliminates the need to create different algorithms if the data type is an integer, string or a character. **The advantages of Generic Programming are** 1. Code Reusability 2. Avoid Function Overloading 3. Once written it can be used for multiple times and cases. Generics can be implemented in C++ using Templates. Template is a simple and yet very powerful tool in C++. The simple idea is to pass data type as a parameter so that we don't need to write the same code for different data types. For example, a software company may need sort() for different data types. Rather than writing and maintaining the multiple codes, we can write one sort() and pass data type as a parameter. Guidelines for Using Template Functions Generic Data Types Generic types are classes or functions that are parameterized over a type. This is done with templates in C++. The template parameters are used to create a generic data type. This concept is similar to function parameters, we pass some template arguments, and the function receives it as a type value. The **syntax** to specify the **template parameters** is, `template <class identifier> function_declaration;` `// or`

```
template <typename identifier> function_declaration;
```

Declaration and Definition Must Be in the Same File

The templates are not normal functions. They only get compiled when some instantiation with particular template arguments. The compiler generates the exact functionality with the provided arguments and template. Because templates are compiled when required, it is impossible to separate the definition and declaration of the template functions into two files.

Overloading Function Templates

The template functions can be overloaded. The compiler will search for the exact overloaded function definition if any particular function call is encountered inside the program. If found, then the overloaded function will execute. Otherwise, the matched template function will execute. Further, if no template function is associated with the function call, the compiler will throw an error.

Generic Functions using Template:

We write a generic function that can be used for different data types. Examples of function templates are sort(), max(), min(), printArray()

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

// One function works for all data types.
// This would work even for user defined types
// if operator '>' is overloaded
template <typename T>

T myMax(T x, T y)
{
   return (x > y) ? x : y;
}

int main()
{

   // Call myMax for int
   cout << myMax<int>(3, 7) << endl;

   // call myMax for double
   cout << myMax<double>(3.0, 7.0) << endl;

   // call myMax for char
   cout << myMax<char>('g', 'e') << endl;
```

```
    return 0;
}
```

Output:

```
7

7

g
```

Working with multi-type Generics:

We can pass more than one data types as arguments to templates. The following example demonstrates the same.

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

template <class T, class U>
class A {
    T x;
    U y;

public:
    A()
    {
        cout << "Constructor Called" << endl;
    }
};

int main()
{
    A<char, char> a;
    A<int, double> b;
    return 0;
}
```

Output:

```
Constructor Called

Constructor Called
```

Class Template

Like function templates, we can also use templates with the class to make it compatible with more than one data type. In C++ programming, you might have used the vector to create a dynamic array, and you can notice that it works fine with every data type you pass inside the <>, ex- vector<int>. This is just because of the class template.

Syntax:

```
template <class T>
class className {
 // Class Definition.
 // We can use T as a type inside this class.
};
```

Class Template and Friend Functions

A non-member function that is defined outside the class and can access the private and protected members of the class is called a friend function. These are used to create a link between classes and functions. We can define our friend function as a template or a normal function in the class template.

Class Templates and Static Variables

The classes in C++ can contain two types of variables, static and non-static(instance). Each object of the class consists of non-static variables. But the static variable remains the same for each object means it is shared among all the created objects. As we are discussing the templates in C++, a point worth noticing is that the static variable in template classes remains shared among all objects of the **same type**.

Class Template and Inheritance

Concepts of inheritance work in a similar way for class templates also. There could be a few major scenarios in Class Template Inheritance which are listed below.

1. Base Class is not a Template class, but a Derived class is a Template class. We can derive from the non-template class and add template members to the derived class.

```
class Base {

};

template < class T >
class Derived: public Base {
   //Use T inside the Derived class
};
```

2. Base Class is a Template class, but Derived class is not a Template class. We can derive from a template class. If we don't want our derived class to be generic, we can use the Base class by providing the template parameter type.

```
template<class T>
class Base {

};

//Inheriting from the Base<int>
class Derived : public Base<int>{
```

```
};
```

3. Base Class is a Template class, and the Derived class is also a Template class. If we want our derived class to be generic, then it should be a template that can pass the template parameter to the base class.

```
template<class T>
class Base {

};

template<class T>
class Derived : public Base<T>{
   //Pass the T to Base class
};
```

Here Base Class and Derived Class are using the same Type.

4. Base Class is a Template Class, and derived class is a Template class with different Types.

We can also use more types in the derived class by including them in the template parameter of the derived class template.

```
template<class T>
class Base {

};

template<class U, class T>
class Derived : public Base<T>{
  //Use U in Derived class and pass T to Base class.
  //We can also use U and T in the Derived class.
};
```

Templates vs. Macros

The templates and macros are somewhat similar; the table below demonstrates the few differences between them.

Templates	Macros
Templates are the special function or classes with generic types.	Macros are the segment of code that is replaced by the macro value. They are defined by #define directive.
Templates get instantiated by the compiler only if a function call exists.	The preprocessor resolves macros during the first phase of compilation, i.e., preprocessing.

Easy to debug, because at the end template is just a function	Little bit complex to debug because the preprocessor handles everything. Also, while working with macros, you can notice that the error for macros is being shown up at the line where macro is defined.
Template is an advanced form of substitution.	Macro is a primitive form of substitution.
Templates are nothing but function calls, hence less efficient as compared to macros.	The macros are efficient because of inline compilation
Templates go through type-checking during compilation.	There is no type checking in macros.

Advantages and Disadvantages of Templates

Advantages

1. Code Compaction, By using templates, we can define generalized functionality and eliminate the repetition of several algorithms for different data types.

2. Reusability of Code, We can define a template that can be used with multiple data types.

3. In-Demand Compilation, One of the amazing advantages of using the template is that when the compiler encounters any function call with associated data types, only the new function is instantiated from the template.

Disadvantages

1. Data Hiding issue, Since the declaration and definition of the template cannot be separated due to in-demand compilation, it is difficult to hide functionality written inside the template.

2. Lack of Portability, Not every compiler supports the templates; hence in rare cases, there might be some portability issues.

Code:

```
// C++ Program to implement template Array class
#include <iostream>
using namespace std;

template <typename T> class Array {
private:
    T* ptr;
    int size;
```

```
public:
   Array(T arr[], int s);
   void print();
};

template <typename T> Array<T>::Array(T arr[], int s)
{
   ptr = new T[s];
   size = s;
   for (int i = 0; i < size; i++)
      ptr[i] = arr[i];
}

template <typename T> void Array<T>::print()
{
   for (int i = 0; i < size; i++)
      cout << " " << *(ptr + i);
   cout << endl;
}

int main()
{
   int arr[5] = { 1, 2, 3, 4, 5 };
   Array<int> a(arr, 5);
   a.print();
   return 0;
}
```

Output
1 2 3 4 5

// C++ Program to implement Use of template

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

template <class T, class U> class A {
   T x;
   U y;

public:
   A() { cout << "Constructor Called" << endl; }
};

int main()
{
   A<char, char> a;
   A<int, double> b;
   return 0;
}
```

Output
Constructor Called
Constructor Called

```
// C++ Program to implement Class Template Arguments Deduction

#include <iostream>
#include <string>
#include <typeinfo>
using namespace std;

// Defining class template
template <typename T> class student {
private:
    string student_name;
    T total_marks;

public:
    // Parameterized constructor
    student(string n, T m)
        : student_name(n)
        , total_marks(m)
    {
    }

    void getinfo()
    {
        // printing the details of the student
        cout << "STUDENT NAME: " << student_name << endl;
        cout << "TOTAL MARKS: " << total_marks << endl;
        cout << "Type ID: " << typeid(total_marks).name()
             << endl;
    }
};

int main()
{

    student s1("Vipul", 100); // Deduces student<int>
    student s2("Yash", 98.5); // Deduces student<double>

    s1.getinfo();
    s2.getinfo();

    return 0;
}
```

Output

	STUDENT NAME: Vipul TOTAL MARKS: 100 Type ID: i STUDENT NAME: Yash TOTAL MARKS: 98.5 Type ID: d