# INTRODUCTION TO OBJECT ORIENTED PROGRAMMING USING C++

**LAB PRACTITAL FILE (ITE321P)**

Submitted as part of the requirements for awarding the degree of

**BACHELOR OF TECHNOLOGY**

**IN**

**ELECTRONICS AND COMMUNICATIONS ENGINEERING**

**SUBMITTED BY :**

**Ketan Kumar (03116412822)**

**B. Tech. (ECE), 5th Semester**

**SUBMITTED TO :**

Dr. M. Bala Krishna,

Prof. at USICT

Guru Gobind Singh
Indraprastha University

**UNIVERSITY SCHOOL OF INFORMATION, COMMUNICATION AND TECHNOLOGY**

**SECTOR 16C, DWARKA, NEW DELHI 110078**

# INDEX

|  |  |  |  |  |
|---|---|---|---|---|
|  | real number and imaginary number. Implement the following member functions:<br>• Constructors - Default, parametric and copy<br>• Overload stream insertion operator < and stream extraction operator >><br>• Overload comparison operators == and !=<br>• Overload binary arithmetic operators +, =, * and /<br>• Overload not operator ! to return normal for complex class (Normal is the square root of the sum of the squares of its real and imaginary parts and it is unary operator)<br>• Overload new) and delete) operators. |  |  |  |
| 7. | Consider 2-D objects circle, square, rectangle and triangle. Implement areal) - Function overloading and display the output. Use new) and delete() to store data members and delete to free memory.<br>Overload >> and << operators. Use pointer to member functions. | 24 - 27 |  |  |
| 8. | Consider class hierarchy with University, Departments, Examination, Accounts and Library classes. Each class has their relevant data members and member functions. Create an abstract class. Use public, private and protected types of inheritance visibility mode. Use single-level, multi-level, multiple and hybrid types of inheritance for implementation. Use functions to getdata() and display). | 28 - 32 |  |  |
| 9. | Design your own inheritance hierarchy comprising the following geometrical objects:<br>Point, Line, Square, Rectangle, Cube, Circle and Cylinder.<br>Find area, volume (if present) and perimeter for all objects. Implement dynamic polymorphism. Implement virtual functions, pure virtual functions and virtual destructors. Use virtual constructors and give the observations. | 33 - 37 |  |  |
| 10. | Write programs using function templates to implement the following sorting methods:<br>• Bubble sort<br>• Selection sort | 38 - 41 |  |  |
| 11. | Write program which illustrates the use of class template for the following data structures: | 42 - 48 |  |  |

| | | | | |
|---|---|---|---|---|
| | • Stack class<br>• Linked list<br>• Dequeue - Double ended queue<br>Implement all operations of stack and linked list data structures. | | | |
| 12. | Write a program to convert decimal to binary, decimal to octal and decimal to hexadecimal. Use user defined manipulator for implementation. | 49 - 50 | | |
| 13. | Using stream manipulators display the contents of the string "Software" in pyramid format<br>S<br>So<br>Sof<br>Soft<br>Softw<br>Softwa<br>Softwar<br>Software | 51 - 52 | | |
| 14. | Create a class whose name is Physical_Education. This class contains data members: name_of_game, cost, duration, place, finals and member functions: get_data(), put_data(). Overload input stream operator and store the data in ASCIl and binary format. Overload output stream operator and display the contents of the data file. | 53 - 56 | | |
| 15. | Create a text data file. Write a program to count the number of characters, number of words, sentences, paragraphs and control characters. Use 1/0 stream functions. | 57 - 59 | | |
| 16. | Consider that the base class stack is available. It does not take care of situations such as overflow and underflow. Enhance this class to MyStack which raises an exception whenever overflow and underflow occurs. Implement various types of exceptions. | 60 - 63 | | |
| 17. | Write a program to implement generic algorithm for sorting set of 10 numbers. | 64 - 65 | | |
| 18. | Write a program to illustrate the operations provided by the Set Container, iterators and allocators. | 66 - 68 | | |

# PRACTICAL No. 1

## AIM
Create a Student class with data members name, age, and address. Implement member functions getdata() and putdata(). Create an array of objects and use a pointer to access the member functions of the objects.

## THEORY
In this practical, the task is to create a class with three data members (name, age, and address) and implement functions to get data from the user (getdata()) and display it (putdata()). We will use an array of objects and a pointer to access the class members in the program.
The getdata() function will prompt the user for input and store it in the respective data members.
The putdata() function will display the stored data.
We'll use a pointer to the array of objects to demonstrate access to class member functions.

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

class Student {
public:
    string name;
    int age;
    string address;

    void getdata() {
        cout << "Enter name: ";
        getline(cin, name);
        cout << "Enter age: ";
        cin >> age;
        cin.ignore();
        cout << "Enter address: ";
        getline(cin, address);
    }

    void putdata() {
        cout << "Name: " << name << endl;
        cout << "Age: " << age << endl;
        cout << "Address: " << address << endl;
    }
};

int main() {
    const int N = 2;
    Student students[N];
    Student* ptr = students;

    for (int i = 0; i < N; i++) {
        cout << "Enter details for student " << i + 1 << endl;
        ptr[i].getdata();
    }

    for (int i = 0; i < N; i++) {
        cout << "Student " << i + 1 << " details:" << endl;
        ptr[i].putdata();
        cout << endl;
    }

    return 0;
}
```

## OUTPUT

```
Enter details for student 1
Enter name: Ketan
Enter age: 22
Enter address: New Delhi

Enter details for student 2
Enter name: Oggy
Enter age: 22
Enter address: New Delhi

Student 1 details:
Name: Ketan
Age: 22
Address: New Delhi

Student 2 details:
Name: Oggy
Age: 22
Address: New Delhi
```

# PRACTICAL No. 2

## AIM
Write a function called neg() that reverses the sign of its integer parameters. Implement the function as follows:

- Using a pointer parameter
- Using a reference parameter.

## THEORY
*Pointer Parameter*: When using a pointer parameter, we pass the address of the variable to the function. The function can then modify the value stored at that address using dereferencing (*).
*Reference Parameter*: When using a reference parameter, we pass the variable directly, but the function works with the actual variable as if it were a reference. No need for dereferencing here; the function directly modifies the variable.

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

void negPointer(int* num) {
    *num = -(*num);
}

void negReference(int& num) {
    num = -num;
}

int main() {
    int a = 5, b = -10;

    cout << "Original value of a: " << a << endl;
    cout << "Original value of b: " << b << endl;

    negPointer(&a);
    cout << "After negPointer(), a: " << a << endl;

    negReference(b);
    cout << "After negReference(), b: " << b << endl;
    return 0;
}
```

## OUTPUT

```
Original value of a: 5
Original value of b: -10
After negPointer(), a: -5
After negReference(), b: 10
```

# PRACTICAL No. 3

## AIM

Create a Time class and implement the addtime() function using:

- Call by value
- Call by reference
- Call by pointer

## THEORY

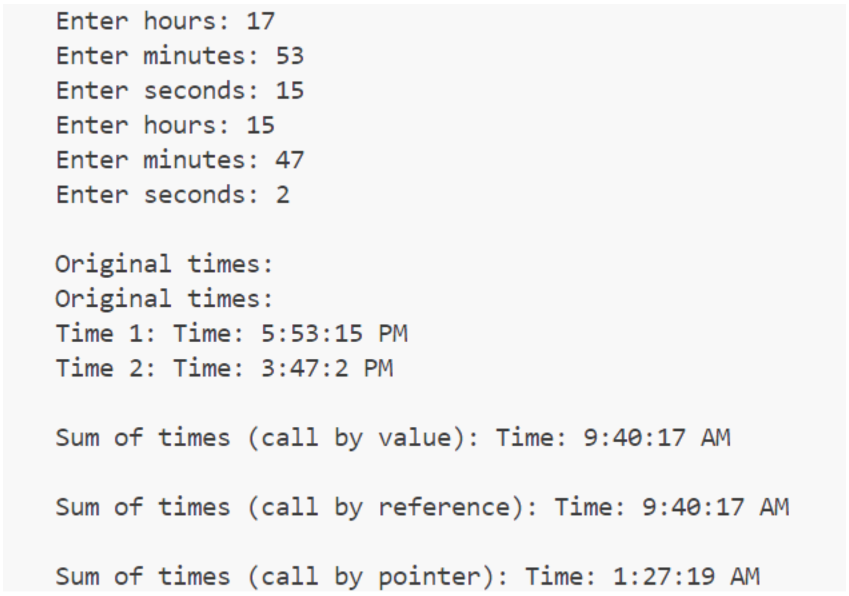
To avoid ambiguity between function signatures, I will rename the functions based on the type of argument passed:

- addtime_value() for call by value.
- addtime_ref() for call by reference.
- addtime_ptr() for call by pointer.

## PROGRAM

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

class Time {
private:
    int hours, minutes, seconds;

public:
    Time(int h = 0, int m = 0, int s = 0) : hours(h), minutes(m), seconds(s)
{}

    void gettime() {
        cout << "Enter hours: ";
        cin >> hours;
        cout << "Enter minutes: ";
        cin >> minutes;
        cout << "Enter seconds: ";
        cin >> seconds;
    }
    void display() {
        string am_pm = (hours >= 12) ? "PM" : "AM";
        int display_hour = hours % 12;
        if (display_hour == 0) display_hour = 12;
        cout << "Time: " << display_hour << ":" << minutes << ":" << seconds
<< " " << am_pm << endl;
    }

    Time addtime_value(Time t) {
        int total_seconds = (hours * 3600 + minutes * 60 + seconds) +
(t.hours * 3600 + t.minutes * 60 + t.seconds);
        return Time((total_seconds / 3600) % 24, (total_seconds % 3600) / 60,
total_seconds % 60);
    }

    void addtime_ref(Time &t) {
        int total_seconds = (hours * 3600 + minutes * 60 + seconds) +
(t.hours * 3600 + t.minutes * 60 + t.seconds);
        hours = (total_seconds / 3600) % 24;
        minutes = (total_seconds % 3600) / 60;
        seconds = total_seconds % 60;
    }

    void addtime_ptr(Time *t) {
        int total_seconds = (hours * 3600 + minutes * 60 + seconds) + (t-
>hours * 3600 + t->minutes * 60 + t->seconds);
        hours = (total_seconds / 3600) % 24;
        minutes = (total_seconds % 3600) / 60;
        seconds = total_seconds % 60;
    }
};

int main() {
    Time t1, t2;
    t1.gettime();
    t2.gettime();

    cout << "\nOriginal times:" << endl;
    cout << "Time 1: ";
    t1.display();
    cout << "Time 2: ";
    t2.display();

    Time t3 = t1.addtime_value(t2);
    cout << "\nSum of times (call by value): ";
    t3.display();

    t1.addtime_ref(t2);
    cout << "\nSum of times (call by reference): ";
    t1.display();

    t1.addtime_ptr(&t2);
    cout << "\nSum of times (call by pointer): ";
    t1.display();

    return 0;
}
```

## OUTPUT



```
Enter hours: 17
Enter minutes: 53
Enter seconds: 15
Enter hours: 15
Enter minutes: 47
Enter seconds: 2

Original times:
Original times:
Time 1: Time: 5:53:15 PM
Time 2: Time: 3:47:2 PM

Sum of times (call by value): Time: 9:40:17 AM

Sum of times (call by reference): Time: 9:40:17 AM

Sum of times (call by pointer): Time: 1:27:19 AM
```

# PRACTICAL No. 4

## AIM

Consider Student class with data members: name, roll num, year, total_subjects, dept_name, percentage, num_of_students. Define any data members as static. Member functions: void getdata(), void showdata(), friend void display_year_info) and static void update(student s). There is another class whose name is University_record and its data members: year, dept_name, count _deptt. The following are its member functions: void updateinfo(student st), void displayinfo(student st) This class is made as a friend to the Student class defined above.

## THEORY

*Student Class*: The class will have static and non-static data members. The static members are shared across all instances of the class. Member functions include:

- getdata() for collecting student data.
- showdata() for displaying student data.
- A friend function display_year_info() to access and display year-wise information of students.
- A static function update(Student s) to update student data.

*University_record Class*: This class contains data about university records such as the year, department name, and department count. It is made a friend to the Student class to access private members of the Student class.

*Friend Function*: A friend function allows one class to access the private and protected members of another class.

*Static Members*: Static members are shared among all instances of the class. We will define the number of students and any other appropriate static data members.

## PROGRAM

```
#include <iostream>
```

```cpp
#include <string>
#include <map>
#include <limits>

using namespace std;

class Student {
public:
    string name;
    int roll_num;
    int year;
    int total_subjects;
    string dept_name;
    float percentage;
    static int num_of_students;

    void getdata() {
        cout << "Enter name: ";
        getline(cin, name);
        cout << "Enter roll number: ";
        cin >> roll_num;
        cin.ignore();
        cout << "Enter year: ";
        cin >> year;
        cin.ignore();
        cout << "Enter total subjects: ";
        cin >> total_subjects;
        cin.ignore();
        cout << "Enter department name: ";
        getline(cin, dept_name);
        cout << "Enter percentage: ";
        cin >> percentage;
        cin.ignore();
        num_of_students++;
    }

    void showdata() {
        cout << "Name: " << name << endl;
        cout << "Roll Number: " << roll_num << endl;
        cout << "Year: " << year << endl;
        cout << "Total Subjects: " << total_subjects << endl;
        cout << "Department: " << dept_name << endl;
        cout << "Percentage: " << percentage << endl;
    }
    friend void display_year_info(Student s);
    friend class University_record;

    static void update(Student s) {
        num_of_students--;
    }
};

int Student::num_of_students = 0;

class University_record {
public:
    map<pair<int, string>, int> year_dept_count;

    void updateinfo(Student st) {
        pair<int, string> key = make_pair(st.year, st.dept_name);
        year_dept_count[key]++;
    }

    void displayinfo() {
        for (const auto& entry : year_dept_count) {
            cout << "Year: " << entry.first.first << endl;
            cout << "Department: " << entry.first.second << endl;
            cout << "Number of Students: " << entry.second << endl;
            cout << endl;
        }
    }
};

int main() {
    int n;
    cout << "Enter the number of students: ";

    while (!(cin >> n)) {
        cout << "Invalid input. Please enter a valid number for the number of
students: ";
        cin.clear();
        cin.ignore(numeric_limits<streamsize>::max(), '\n');
    }

    cin.ignore();

    Student* students = new Student[n];

    for (int i = 0; i < n; i++) {
        cout << "Enter details for student " << i + 1 << endl;
        students[i].getdata();
    }

    University_record univ_rec;
    for (int i = 0; i < n; i++) {
        univ_rec.updateinfo(students[i]);
    }

    for (int i = 0; i < n; i++) {
        cout << "Student " << i + 1 << " details:" << endl;
        students[i].showdata();
        cout << endl;
    }

    univ_rec.displayinfo();

    delete[] students;

    return 0;
}
```

## OUTPUT

```
Enter the number of students: 2

Enter details for student 1
Enter name: Ketan
Enter roll number: 031
Enter year: 3
Enter total subjects: 3
Enter department name: ECE
Enter percentage: 77

Enter details for student 2
Enter name: Oggy
Enter roll number: 031
Enter year: 3
Enter total subjects: 3
Enter department name: ECE
Enter percentage: 77
Student 1 details:
Name: Ketan
Roll Number: 31
Year: 3
Total Subjects: 3
Department: ECE
Percentage: 77

Student 2 details:
Name: Oggy
Roll Number: 31
Year: 3
Total Subjects: 3
Department: ECE
Percentage: 77

Year: 3
Department: ECE
Number of Students: 2
```

# PRACTICAL No. 5

## AIM
Consider 2-D objects circle, square, rectangle and triangle. Implement areal) - Function overloading and display the output. Use new() and delete() to store data members and delete to free memory.
Overload >> and << operators. Use this pointer.

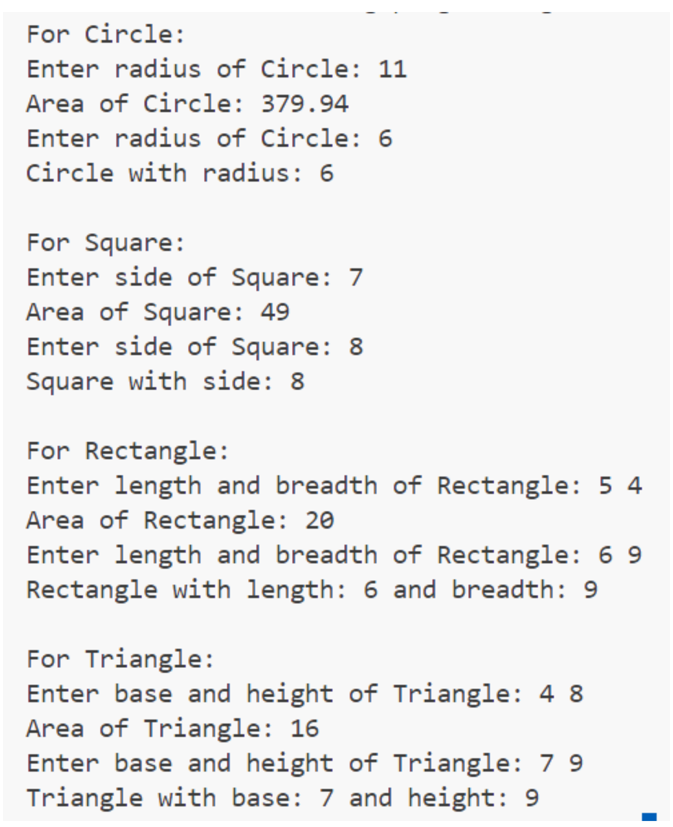
## THEORY
In this program, we demonstrate the use of *Function Overloading*, *Operator Overloading*, and *Dynamic Memory Management in C++*.

- *Function Overloading*: It allows us to define multiple functions with the same name but different parameters. In this case, the areal() function is overloaded to calculate the area for different shapes such as Circle, Square, Rectangle, and Triangle.
- *Operator Overloading*: Operators like >> and << are overloaded to handle input and output for user-defined types (shapes). This allows for easy reading and displaying of shape data.
- *Dynamic Memory Management*: We use new to allocate memory for shape objects dynamically and delete to free the memory when it's no longer needed.
- *The 'this' Pointer*: It refers to the current object instance in a class and is used to access the object's members.

This program allows us to input, process, and display areas of different shapes efficiently.

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

class Shape {
public:
    virtual void areal() = 0;
    virtual void display() = 0;
    virtual ~Shape() {}
};

class Circle : public Shape {
private:
    float radius;
public:
    Circle() {
        radius = 0;
    }
    void areal() override {
        float area = 3.14 * radius * radius;
        cout << "Area of Circle: " << area << endl;
    }
    void display() override {
        cout << "Enter radius of Circle: ";
        cin >> radius;
    }
    friend istream& operator>>(istream& in, Circle& c) {
        cout << "Enter radius of Circle: ";
        in >> c.radius;
        return in;
    }
    friend ostream& operator<<(ostream& out, const Circle& c) {
        out << "Circle with radius: " << c.radius;
        return out;
    }
};

class Square : public Shape {
private:
    float side;
public:
    Square() {
        side = 0;
    }
    void areal() override {
        float area = side * side;
        cout << "Area of Square: " << area << endl;
    }
    void display() override {
        cout << "Enter side of Square: ";
        cin >> side;
    }
    friend istream& operator>>(istream& in, Square& s) {
        cout << "Enter side of Square: ";
        in >> s.side;
        return in;
    }
    friend ostream& operator<<(ostream& out, const Square& s) {
        out << "Square with side: " << s.side;
        return out;
    }
};

class Rectangle : public Shape {
private:
    float length, breadth;
public:
    Rectangle() {
        length = breadth = 0;
    }
    void areal() override {
        float area = length * breadth;
        cout << "Area of Rectangle: " << area << endl;
    }
    void display() override {
        cout << "Enter length and breadth of Rectangle: ";
        cin >> length >> breadth;
    }
    friend istream& operator>>(istream& in, Rectangle& r) {
        cout << "Enter length and breadth of Rectangle: ";
        in >> r.length >> r.breadth;
        return in;
    }
    friend ostream& operator<<(ostream& out, const Rectangle& r) {
        out << "Rectangle with length: " << r.length << " and breadth: " <<
r.breadth;
        return out;
    }
};

class Triangle : public Shape {
private:
    float base, height;
public:
    Triangle() {
        base = height = 0;
    }
    void areal() override {
        float area = 0.5 * base * height;
        cout << "Area of Triangle: " << area << endl;
    }
    void display() override {
        cout << "Enter base and height of Triangle: ";
        cin >> base >> height;
    }
    friend istream& operator>>(istream& in, Triangle& t) {
        cout << "Enter base and height of Triangle: ";
        in >> t.base >> t.height;
        return in;
    }
    friend ostream& operator<<(ostream& out, const Triangle& t) {
        out << "Triangle with base: " << t.base << " and height: " <<
t.height;
        return out;
    }
};
int main() {
    Shape* shapePtr;
    Circle* circlePtr = new Circle();
    Square* squarePtr = new Square();
    Rectangle* rectPtr = new Rectangle();
    Triangle* trianglePtr = new Triangle();
    cout << "For Circle:" << endl;
    cin >> *circlePtr;
    circlePtr->areal();
    circlePtr->display();
    cout << *circlePtr << endl;
    cout << "\nFor Square:" << endl;
    cin >> *squarePtr;
    squarePtr->areal();
    squarePtr->display();
    cout << *squarePtr << endl;
    cout << "\nFor Rectangle:" << endl;
    cin >> *rectPtr;
    rectPtr->areal();
    rectPtr->display();
    cout << *rectPtr << endl;
    cout << "\nFor Triangle:" << endl;
    cin >> *trianglePtr;
    trianglePtr->areal();
    trianglePtr->display();
    cout << *trianglePtr << endl;
    delete circlePtr;
    delete squarePtr;
    delete rectPtr;
    delete trianglePtr;
    return 0;
}
```

## OUTPUT



```
For Circle:
Enter radius of Circle: 11
Area of Circle: 379.94
Enter radius of Circle: 6
Circle with radius: 6

For Square:
Enter side of Square: 7
Area of Square: 49
Enter side of Square: 8
Square with side: 8

For Rectangle:
Enter length and breadth of Rectangle: 5 4
Area of Rectangle: 20
Enter length and breadth of Rectangle: 6 9
Rectangle with length: 6 and breadth: 9

For Triangle:
Enter base and height of Triangle: 4 8
Area of Triangle: 16
Enter base and height of Triangle: 7 9
Triangle with base: 7 and height: 9
```

# PRACTICAL No. 6

## AIM

Create a class whose name is complex. This class contains data members: _num and i_num for storing real number and imaginary number. Implement the following member functions:

- Constructors - Default, parametric and copy
- Overload stream insertion operator < and stream extraction operator >>
- Overload comparison operators == and !=
- Overload binary arithmetic operators +, =, * and /
- Overload not operator ! to return normal for complex class (Normal is the square root of the sum of the squares of its real and imaginary parts and it is unary operator)
- Overload new) and delete) operators.

## THEORY

The task involves creating a Complex class with real and imaginary parts, where we will perform various operations such as:

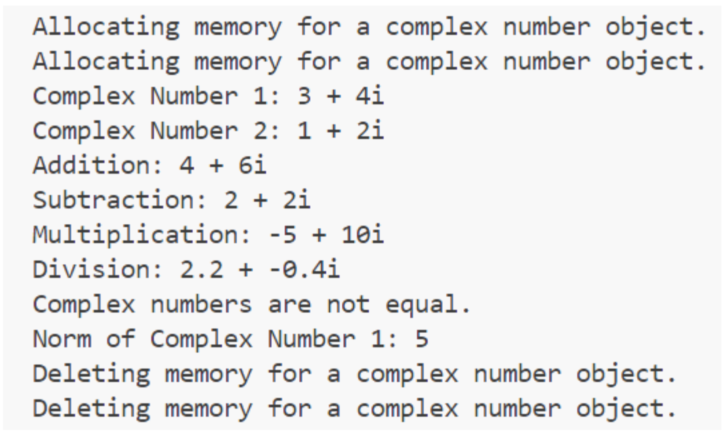
- *Constructors*: We will define three types of constructors:
  - *Default Constructor*: Initializes the complex number to zero.
  - *Parametric Constructor*: Allows setting custom real and imaginary values.
  - *Copy Constructor*: Creates a copy of an existing complex number.
- *Operator Overloading*:
  - *Stream Insertion (<<) and Stream Extraction (>>) Operators*: These will allow reading and printing complex numbers to/from streams (like cin and cout).
  - *Comparison Operators (==, !=)*: These will check if two complex numbers are equal or not based on their real and imaginary parts.
  - *Binary Arithmetic Operators (+, -, *, /)*: These will allow addition, subtraction, multiplication, and division of complex numbers.

    - *Unary ! Operator*: This will compute the "norm" of the complex number, which is the square root of the sum of squares of its real and imaginary parts.
    - *New and Delete Operators*: These are overloaded to manage dynamic memory allocation for complex objects.
- *Dynamic Memory Management*: The program will use new to allocate memory dynamically for complex number objects and delete to free that memory.

## PROGRAM

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

class Complex {
private:
    double _num;
    double i_num;

public:
    Complex() : _num(0), i_num(0) {}

    Complex(double real, double imag) : _num(real), i_num(imag) {}

    Complex(const Complex &c) : _num(c._num), i_num(c.i_num) {}

    friend istream& operator>>(istream &in, Complex &c) {
        cout << "Enter real part: ";
        in >> c._num;
        cout << "Enter imaginary part: ";
        in >> c.i_num;
        return in;
    }

    friend ostream& operator<<(ostream &out, const Complex &c) {
        out << c._num << " + " << c.i_num << "i";
        return out;
    }

    bool operator==(const Complex &c) {
        return (_num == c._num) && (i_num == c.i_num);
    }
    bool operator!=(const Complex &c) {
        return !(*this == c);
    }

    Complex operator+(const Complex &c) {
        return Complex(_num + c._num, i_num + c.i_num);
    }

    Complex operator-(const Complex &c) {
        return Complex(_num - c._num, i_num - c.i_num);
    }

    Complex operator*(const Complex &c) {
        double real = (_num * c._num) - (i_num * c.i_num);
        double imag = (_num * c.i_num) + (i_num * c._num);
        return Complex(real, imag);
    }

    Complex operator/(const Complex &c) {
        double denom = (c._num * c._num) + (c.i_num * c.i_num);
        double real = ((_num * c._num) + (i_num * c.i_num)) / denom;
        double imag = ((i_num * c._num) - (_num * c.i_num)) / denom;
        return Complex(real, imag);
    }

    double operator!() {
        return sqrt((_num * _num) + (i_num * i_num));
    }

    void* operator new(size_t size) {
        cout << "Allocating memory for a complex number object." << endl;
        return ::operator new(size);
    }

    void operator delete(void* pointer) {
        cout << "Deleting memory for a complex number object." << endl;
        ::operator delete(pointer);
    }
};

int main() {
    Complex *c1 = new Complex(3.0, 4.0);
    Complex *c2 = new Complex(1.0, 2.0);

    cout << "Complex Number 1: " << *c1 << endl;
    cout << "Complex Number 2: " << *c2 << endl;

    Complex c3 = *c1 + *c2;
    cout << "Addition: " << c3 << endl;

    Complex c4 = *c1 - *c2;
    cout << "Subtraction: " << c4 << endl;

    Complex c5 = *c1 * *c2;
    cout << "Multiplication: " << c5 << endl;

    Complex c6 = *c1 / *c2;
    cout << "Division: " << c6 << endl;

    if (*c1 == *c2)
        cout << "Complex numbers are equal." << endl;
    else
        cout << "Complex numbers are not equal." << endl;

    cout << "Norm of Complex Number 1: " << !*c1 << endl;

    delete c1;
    delete c2;

    return 0;
}
```

## OUTPUT



```
Allocating memory for a complex number object.
Allocating memory for a complex number object.
Complex Number 1: 3 + 4i
Complex Number 2: 1 + 2i
Addition: 4 + 6i
Subtraction: 2 + 2i
Multiplication: -5 + 10i
Division: 2.2 + -0.4i
Complex numbers are not equal.
Norm of Complex Number 1: 5
Deleting memory for a complex number object.
Deleting memory for a complex number object.
```

# PRACTICAL No. 7

## AIM

Consider 2-D objects circle, square, rectangle and triangle. Implement areal) - Function overloading and display the output. Use new) and delete() to store data members and delete to free memory.

Overload >> and << operators. Use pointer to member functions.

## THEORY

In this task, we will create a class hierarchy for different 2-D shapes: Circle, Square, Rectangle, and Triangle. The objective is to demonstrate:

- *Function Overloading*: This will be used to calculate the area for each shape, allowing us to calculate the area in multiple ways (e.g., depending on the shape).
- *Operator Overloading*:
  - Overloading the stream insertion (<<) and stream extraction (>>) operators to input and output the data for shapes.
  - Using pointer to member functions to call member functions dynamically.
- *Memory Management*: Using new and delete to allocate and free memory for storing the objects' data members.

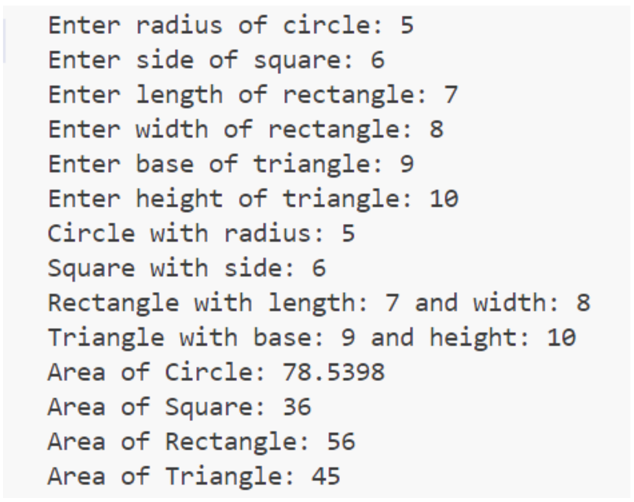
## PROGRAM

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

class Shape {
public:
    virtual void calculateArea() = 0;
    virtual void display() = 0;
    virtual ~Shape() {}
};
class Circle : public Shape {
private:
    double radius;
```

```cpp
public:
    Circle(double r = 0) : radius(r) {}
    void calculateArea() override {
        double area = M_PI * radius * radius;
        cout << "Area of Circle: " << area << endl;
    }
    void display() override {
        cout << "Circle with radius: " << radius << endl;
    }
    friend istream& operator>>(istream &in, Circle &c) {
        cout << "Enter radius of circle: ";
        in >> c.radius;
        return in;
    }
    friend ostream& operator<<(ostream &out, const Circle &c) {
        out << "Circle with radius: " << c.radius;
        return out;
    }
};
class Square : public Shape {
private:
    double side;
public:
    Square(double s = 0) : side(s) {}
    void calculateArea() override {
        double area = side * side;
        cout << "Area of Square: " << area << endl;
    }
    void display() override {
        cout << "Square with side: " << side << endl;
    }
    friend istream& operator>>(istream &in, Square &s) {
        cout << "Enter side of square: ";
        in >> s.side;
        return in;
    }
    friend ostream& operator<<(ostream &out, const Square &s) {
        out << "Square with side: " << s.side;
        return out;
    }
};
class Rectangle : public Shape {
private:
    double length, width;
public:
    Rectangle(double l = 0, double w = 0) : length(l), width(w) {}
    void calculateArea() override {
        double area = length * width;
        cout << "Area of Rectangle: " << area << endl;
    }
    void display() override {
        cout << "Rectangle with length: " << length << " and width: " <<
width << endl;
    }
    friend istream& operator>>(istream &in, Rectangle &r) {
        cout << "Enter length of rectangle: ";
        in >> r.length;
        cout << "Enter width of rectangle: ";
        in >> r.width;
        return in;
    }
    friend ostream& operator<<(ostream &out, const Rectangle &r) {
        out << "Rectangle with length: " << r.length << " and width: " <<
r.width;
        return out;
    }
};
class Triangle : public Shape {
private:
    double base, height;
public:
    Triangle(double b = 0, double h = 0) : base(b), height(h) {}
    void calculateArea() override {
        double area = 0.5 * base * height;
        cout << "Area of Triangle: " << area << endl;
    }
    void display() override {
        cout << "Triangle with base: " << base << " and height: " << height
<< endl;
    }
    friend istream& operator>>(istream &in, Triangle &t) {
        cout << "Enter base of triangle: ";
        in >> t.base;
        cout << "Enter height of triangle: ";
        in >> t.height;
        return in;
    }
    friend ostream& operator<<(ostream &out, const Triangle &t) {
        out << "Triangle with base: " << t.base << " and height: " <<
t.height;
```

```
        return out;
    }
};
int main() {
    Shape *circle = new Circle();
    Shape *square = new Square();
    Shape *rectangle = new Rectangle();
    Shape *triangle = new Triangle();
    cin >> *dynamic_cast<Circle*>(circle);
    cin >> *dynamic_cast<Square*>(square);
    cin >> *dynamic_cast<Rectangle*>(rectangle);
    cin >> *dynamic_cast<Triangle*>(triangle);
    cout << *dynamic_cast<Circle*>(circle) << endl;
    cout << *dynamic_cast<Square*>(square) << endl;
    cout << *dynamic_cast<Rectangle*>(rectangle) << endl;
    cout << *dynamic_cast<Triangle*>(triangle) << endl;
    void (Shape::*ptr)() = &Shape::calculateArea;
    (circle->*ptr)();
    (square->*ptr)();
    (rectangle->*ptr)();
    (triangle->*ptr)();
    delete circle;
    delete square;
    delete rectangle;
    delete triangle;
    return 0;
}
```

## OUTPUT



```
Enter radius of circle: 5
Enter side of square: 6
Enter length of rectangle: 7
Enter width of rectangle: 8
Enter base of triangle: 9
Enter height of triangle: 10
Circle with radius: 5
Square with side: 6
Rectangle with length: 7 and width: 8
Triangle with base: 9 and height: 10
Area of Circle: 78.5398
Area of Square: 36
Area of Rectangle: 56
Area of Triangle: 45
```

# PRACTICAL No. 8

## AIM

Consider class hierarchy with University, Departments, Examination, Accounts and Library classes. Each class has their relevant data members and member functions. Create an abstract class. Use public, private and protected types of inheritance visibility mode. Use single-level, multi-level, multiple and hybrid types of inheritance for implementation. Use functions to getdata() and display).

## THEORY

*Abstract Class*: An abstract class contains at least one pure virtual function. It is used as a base class for other classes that will implement these functions.

*Inheritance Types*:

- *Single-level inheritance*: A class inherits from another class.
- *Multi-level inheritance*: A class inherits from another class, which in turn inherits from another class.
- *Multiple inheritance*: A class inherits from more than one class.
- *Hybrid inheritance*: A combination of single-level, multi-level, and multiple inheritance.

*Encapsulation with Access Specifiers*:

- *Public inheritance*: Members of the base class are accessible as public members in the derived class.
- *Protected inheritance*: Members of the base class are accessible as protected members in the derived class.
- *Private inheritance*: Members of the base class are accessible only in the derived class and not externally.

*Functions*:

- *getdata()*: This function is used to take input for the data members of the class.
- *display()*: This function is used to display the data members of the class.

## PROGRAM

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

class University {
public:
    virtual void getdata() = 0;
    virtual void display() = 0;
    virtual ~University() {}
};

class Department : public University {
protected:
    string departmentName;
    int numFaculty;

public:
    void getdata() override {
        cout << "Enter Department Name: ";
        cin >> departmentName;
        cout << "Enter number of Faculty: ";
        cin >> numFaculty;
    }

    void display() override {
        cout << "Department Name: " << departmentName << endl;
        cout << "Number of Faculty: " << numFaculty << endl;
    }
};

class Examination : public University {
protected:
    string examSchedule;
    float examResults;

public:
    void getdata() override {
        cout << "Enter Exam Schedule: ";
        cin >> examSchedule;
        cout << "Enter Exam Results (percentage): ";
        cin >> examResults;
    }
    void display() override {
        cout << "Exam Schedule: " << examSchedule << endl;
        cout << "Exam Results: " << examResults << "%" << endl;
    }
};

class Accounts : public University {
protected:
    float budget;
    float feeCollection;

public:
    void getdata() override {
        cout << "Enter University Budget: ";
        cin >> budget;
        cout << "Enter Fee Collection: ";
        cin >> feeCollection;
    }

    void display() override {
        cout << "University Budget: " << budget << endl;
        cout << "Fee Collection: " << feeCollection << endl;
    }
};

class Library : public University {
protected:
    int numBooks;
    int booksIssued;

public:
    void getdata() override {
        cout << "Enter number of Books in Library: ";
        cin >> numBooks;
        cout << "Enter number of Books Issued: ";
        cin >> booksIssued;
    }

    void display() override {
        cout << "Total Books in Library: " << numBooks << endl;
        cout << "Books Issued: " << booksIssued << endl;
    }
};
```

```
class UniversityWithLibrary : public Accounts, public Library {
public:
    void getdata() override {
        Accounts::getdata();
        Library::getdata();
    }

    void display() override {
        Accounts::display();
        Library::display();
    }
};

int main() {
    Department dept;
    Examination exam;
    Accounts acc;
    Library lib;

    cout << "Enter Department Data: \n";
    dept.getdata();
    cout << "\nEnter Examination Data: \n";
    exam.getdata();
    cout << "\nEnter Accounts Data: \n";
    acc.getdata();
    cout << "\nEnter Library Data: \n";
    lib.getdata();

    cout << "\nDepartment Data: \n";
    dept.display();
    cout << "\nExamination Data: \n";
    exam.display();
    cout << "\nAccounts Data: \n";
    acc.display();
    cout << "\nLibrary Data: \n";
    lib.display();

    UniversityWithLibrary uniLib;
    cout << "\nEnter Data for University with Accounts and Library: \n";
    uniLib.getdata();
    cout << "\nDisplaying Data for University with Accounts and Library: \n";
    uniLib.display();

    return 0;
}
```

## OUTPUT

```
Enter Department Data:
Enter Department Name: ECE
Enter number of Faculty: 6

Enter Examination Data:
Enter Exam Schedule: 06/12/24
Enter Exam Results (percentage): 78

Enter Accounts Data:
Enter University Budget: 2000000000
Enter Fee Collection: 1000000

Enter Library Data:
Enter number of Books in Library: 1300
Enter number of Books Issued: 150

Department Data:
Department Name: ECE
Number of Faculty: 6

Examination Data:
Exam Schedule: 06/12/24
Exam Results: 78%

Accounts Data:
University Budget: 2e+009
Fee Collection: 1e+006

Library Data:
Total Books in Library: 1300
Books Issued: 150
```

# PRACTICAL No. 9

## AIM

Design your own inheritance hierarchy comprising the following geometrical objects: Point, Line, Square, Rectangle, Cube, Circle and Cylinder.

Find area, volume (if present) and perimeter for all objects. Implement dynamic polymorphism.

Implement virtual functions, pure virtual functions and virtual destructors. Use virtual constructors and give the observations.

## THEORY

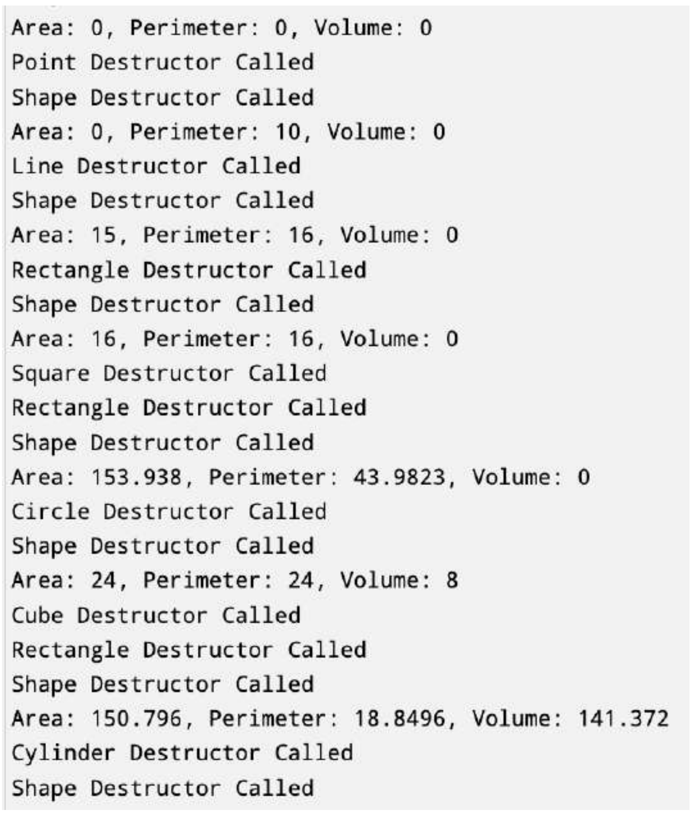
*Inheritance Hierarchy:*

- *Base Class*: A generic class Shape is defined with pure virtual functions for area(), perimeter(), and volume().
- *Derived Classes*:
  - *Point*: Has no area, volume, or perimeter.
  - *Line*: Has a length (perimeter equivalent) but no area or volume.
  - *Square, Rectangle, Circle*: Have area and perimeter.
  - *Cube, Cylinder*: Have area, perimeter, and volume.
- *Dynamic Polymorphism*: Achieved by overriding base class functions in derived classes and accessing them via pointers or references to the base class.
- *Virtual Functions and Pure Virtual Functions*: area(), perimeter(), and volume() are declared as virtual or pure virtual in the base class to ensure derived classes implement their own versions.
- *Virtual Destructors*: Ensures proper cleanup of derived class objects when deleted through a base class pointer.
- *Virtual Constructors*: While C++ does not natively support virtual constructors, we simulate the behavior using a clone() function in the base class that is overridden by derived classes.

## PROGRAM

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

class Shape {
public:
    virtual double area() const = 0;
    virtual double perimeter() const = 0;
    virtual double volume() const = 0;

    virtual ~Shape() { cout << "Shape Destructor Called" << endl; }

    virtual Shape* clone() const = 0;
};

class Point : public Shape {
protected:
    double x, y;

public:
    Point(double x = 0, double y = 0) : x(x), y(y) {}

    double area() const override { return 0; }
    double perimeter() const override { return 0; }
    double volume() const override { return 0; }

    void display() const { cout << "Point(" << x << ", " << y << ")" << endl;
}

    Shape* clone() const override { return new Point(*this); }

    ~Point() override { cout << "Point Destructor Called" << endl; }
};

class Line : public Shape {
protected:
    double length;

public:
    Line(double length = 0) : length(length) {}

    double area() const override { return 0; }
    double perimeter() const override { return length; }
    double volume() const override { return 0; }
    Shape* clone() const override { return new Line(*this); }

    ~Line() override { cout << "Line Destructor Called" << endl; }
};

class Rectangle : public Shape {
protected:
    double length, width;

public:
    Rectangle(double length = 0, double width = 0) : length(length), 
width(width) {}

    double area() const override { return length * width; }
    double perimeter() const override { return 2 * (length + width); }
    double volume() const override { return 0; }

    Shape* clone() const override { return new Rectangle(*this); }

    ~Rectangle() override { cout << "Rectangle Destructor Called" << endl; }
};

class Square : public Rectangle {
public:
    Square(double side = 0) : Rectangle(side, side) {}

    Shape* clone() const override { return new Square(*this); }

    ~Square() override { cout << "Square Destructor Called" << endl; }
};

class Circle : public Shape {
protected:
    double radius;

public:
    Circle(double radius = 0) : radius(radius) {}

    double area() const override { return M_PI * radius * radius; }
    double perimeter() const override { return 2 * M_PI * radius; }
    double volume() const override { return 0; }

    Shape* clone() const override { return new Circle(*this); }
    ~Circle() override { cout << "Circle Destructor Called" << endl; }
};

class Cube : public Rectangle {
protected:
    double height;

public:
    Cube(double length = 0) : Rectangle(length, length), height(length) {}

    double area() const override { return 6 * length * length; }
    double perimeter() const override { return 12 * length; }
    double volume() const override { return length * length * length; }

    Shape* clone() const override { return new Cube(*this); }

    ~Cube() override { cout << "Cube Destructor Called" << endl; }
};

class Cylinder : public Shape {
protected:
    double radius, height;

public:
    Cylinder(double radius = 0, double height = 0) : radius(radius),
height(height) {}

    double area() const override { return 2 * M_PI * radius * (radius +
height); }
    double perimeter() const override { return 2 * M_PI * radius; }
    double volume() const override { return M_PI * radius * radius * height;
}

    Shape* clone() const override { return new Cylinder(*this); }

    ~Cylinder() override { cout << "Cylinder Destructor Called" << endl; }
};

int main() {
    Shape* shapes[] = {
        new Point(3, 4),
        new Line(10),
        new Rectangle(5, 3),
        new Square(4),
        new Circle(7),
        new Cube(2),
        new Cylinder(3, 5)
    };

    for (Shape* shape : shapes) {
        cout << "Area: " << shape->area() << ", Perimeter: " << shape-
>perimeter()
             << ", Volume: " << shape->volume() << endl;
        delete shape;
    }

    return 0;
}
```

## OUTPUT



```
Area: 0, Perimeter: 0, Volume: 0
Point Destructor Called
Shape Destructor Called
Area: 0, Perimeter: 10, Volume: 0
Line Destructor Called
Shape Destructor Called
Area: 15, Perimeter: 16, Volume: 0
Rectangle Destructor Called
Shape Destructor Called
Area: 16, Perimeter: 16, Volume: 0
Square Destructor Called
Rectangle Destructor Called
Shape Destructor Called
Area: 153.938, Perimeter: 43.9823, Volume: 0
Circle Destructor Called
Shape Destructor Called
Area: 24, Perimeter: 24, Volume: 8
Cube Destructor Called
Rectangle Destructor Called
Shape Destructor Called
Area: 150.796, Perimeter: 18.8496, Volume: 141.372
Cylinder Destructor Called
Shape Destructor Called
```

# PRACTICAL No. 10

## AIM

Write programs using function templates to implement the following sorting methods:

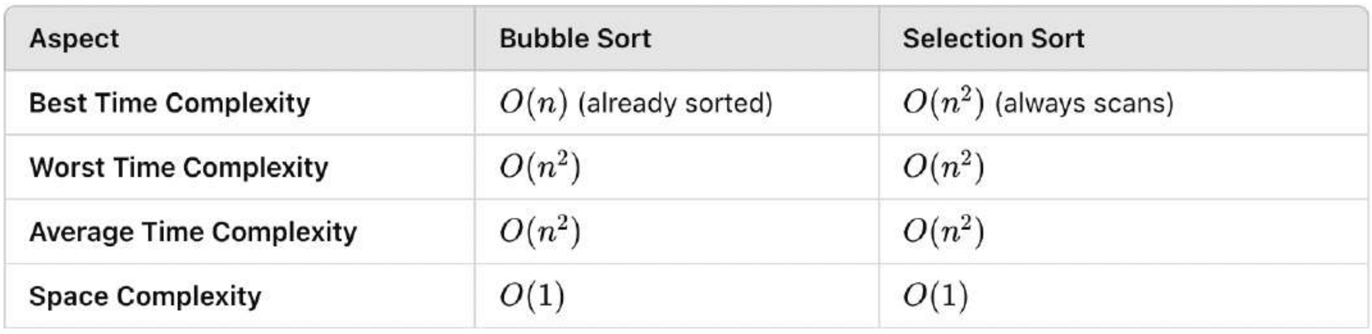
- Bubble sort
- Selection sort

## THEORY

*Bubble Sort:*

- Compares adjacent elements and swaps them if they are in the wrong order.
- Repeats the process for every element.

*Selection Sort:* Finds the smallest element in the unsorted part and swaps it with the first element of the unsorted part.



| Aspect | Bubble Sort | Selection Sort |
|---|---|---|
| Best Time Complexity | $O(n)$ (already sorted) | $O(n^2)$ (always scans) |
| Worst Time Complexity | $O(n^2)$ | $O(n^2)$ |
| Average Time Complexity | $O(n^2)$ | $O(n^2)$ |
| Space Complexity | $O(1)$ | $O(1)$ |

## PROGRAM

**1. BUBBLE SORT**

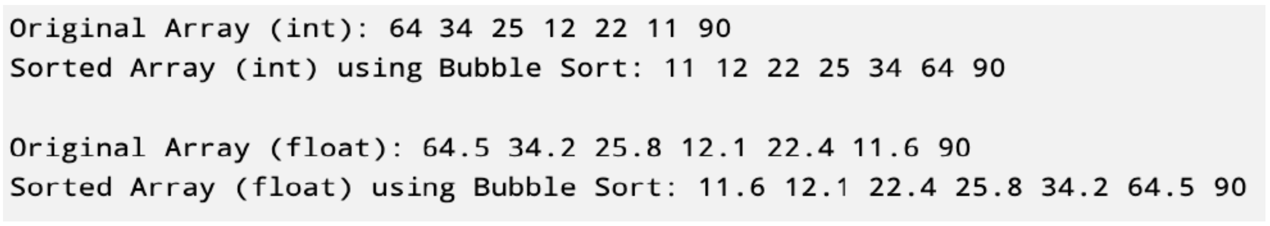
```
#include <iostream>
using namespace std;

template <typename T>
void bubbleSort(T arr[], int size) {
    for (int i = 0; i < size - 1; ++i) {
        for (int j = 0; j < size - i - 1; ++j) {
            if (arr[j] > arr[j + 1]) {
                T temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

template <typename T>
void displayArray(const T arr[], int size) {
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arrInt[] = {64, 34, 25, 12, 22, 11, 90};
    int sizeInt = sizeof(arrInt) / sizeof(arrInt[0]);

    cout << "Original Array (int): ";
    displayArray(arrInt, sizeInt);

    bubbleSort(arrInt, sizeInt);
    cout << "Sorted Array (int) using Bubble Sort: ";
    displayArray(arrInt, sizeInt);

    float arrFloat[] = {64.5, 34.2, 25.8, 12.1, 22.4, 11.6, 90.0};
    int sizeFloat = sizeof(arrFloat) / sizeof(arrFloat[0]);

    cout << "\nOriginal Array (float): ";
    displayArray(arrFloat, sizeFloat);

    bubbleSort(arrFloat, sizeFloat);
    cout << "Sorted Array (float) using Bubble Sort: ";
    displayArray(arrFloat, sizeFloat);

    return 0;
}
```

**2. SELECTION SORT**

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

template <typename T>
void selectionSort(T arr[], int size) {
    for (int i = 0; i < size - 1; ++i) {
        int minIndex = i;
        for (int j = i + 1; j < size; ++j) {
            if (arr[j] < arr[minIndex]) {
                minIndex = j;
            }
        }
        T temp = arr[minIndex];
        arr[minIndex] = arr[i];
        arr[i] = temp;
    }
}

template <typename T>
void displayArray(const T arr[], int size) {
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int arrInt[] = {29, 10, 14, 37, 13};
    int sizeInt = sizeof(arrInt) / sizeof(arrInt[0]);

    cout << "Original Array (int): ";
    displayArray(arrInt, sizeInt);

    selectionSort(arrInt, sizeInt);
    cout << "Sorted Array (int) using Selection Sort: ";
    displayArray(arrInt, sizeInt);

    double arrDouble[] = {29.5, 10.3, 14.7, 37.1, 13.2};
    int sizeDouble = sizeof(arrDouble) / sizeof(arrDouble[0]);

    cout << "\nOriginal Array (double): ";
    displayArray(arrDouble, sizeDouble);

    selectionSort(arrDouble, sizeDouble);
    cout << "Sorted Array (double) using Selection Sort: ";
    displayArray(arrDouble, sizeDouble);

    return 0;
}
```

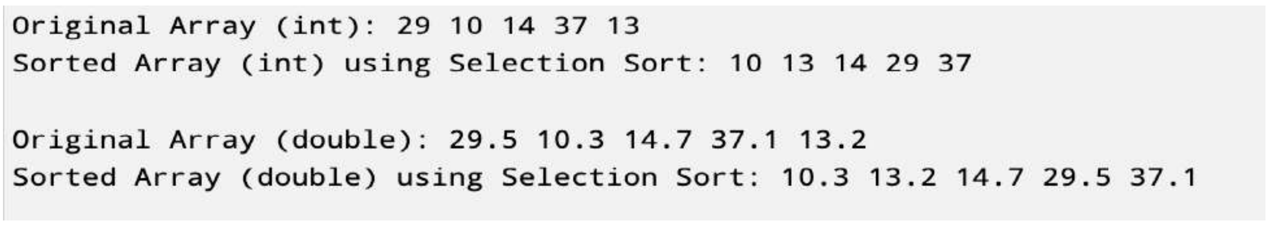
## OUTPUT

### 1. BUBBLE SORT



```
Original Array (int): 64 34 25 12 22 11 90
Sorted Array (int) using Bubble Sort: 11 12 22 25 34 64 90

Original Array (float): 64.5 34.2 25.8 12.1 22.4 11.6 90
Sorted Array (float) using Bubble Sort: 11.6 12.1 22.4 25.8 34.2 64.5 90
```

### 2. SELECTION SORT



```
Original Array (int): 29 10 14 37 13
Sorted Array (int) using Selection Sort: 10 13 14 29 37

Original Array (double): 29.5 10.3 14.7 37.1 13.2
Sorted Array (double) using Selection Sort: 10.3 13.2 14.7 29.5 37.1
```

# PRACTICAL No. 11

## AIM

Write program which illustrates the use of class template for the following data structures:

- Stack class
- Linked list
- Dequeue - Double ended queue

Implement all operations of stack and linked list data structures.

## THEORY

*Class Templates* allow the creation of generic data structures that can work with any data type without rewriting the code for each type. This is particularly useful for implementing data structures like stacks, linked lists, and deques.

*Stack Operations*:

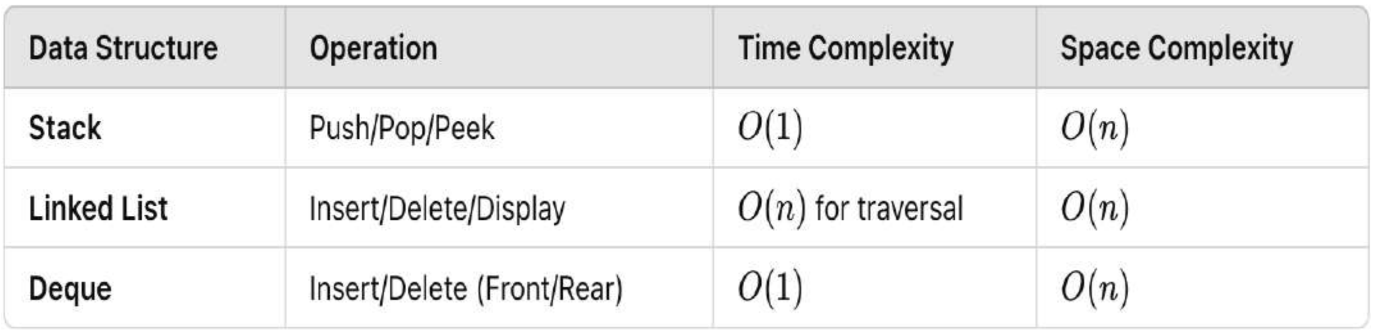
- Push (Add an element to the top)
- Pop (Remove the top element)
- Peek (Retrieve the top element without removing it)
- Check if empty

*Linked List Operations*:

- Insert at the front
- Insert at the end
- Delete a node
- Traverse the list

*Deque Operations*:

- Add element to the front
- Add element to the rear
- Remove element from the front
- Remove element from the rear.

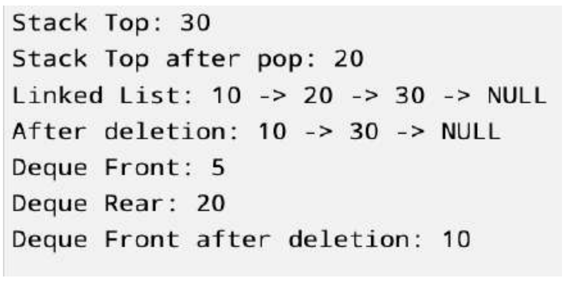


| Data Structure | Operation | Time Complexity | Space Complexity |
|---|---|---|---|
| **Stack** | Push/Pop/Peek | $O(1)$ | $O(n)$ |
| **Linked List** | Insert/Delete/Display | $O(n)$ for traversal | $O(n)$ |
| **Deque** | Insert/Delete (Front/Rear) | $O(1)$ | $O(n)$ |

## PROGRAM

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

template <typename T>
class Stack {
    T* arr;
    int top;
    int capacity;

public:
    Stack(int size) {
        capacity = size;
        arr = new T[size];
        top = -1;
    }

    ~Stack() { delete[] arr; }

    void push(T data) {
        if (top == capacity - 1) {
            cout << "Stack Overflow!" << endl;
            return;
        }
        arr[++top] = data;
    }

    void pop() {
        if (top == -1) {
            cout << "Stack Underflow!" << endl;
            return;
        }
        top--;
    }
    T peek() {
        if (top == -1) {
            cout << "Stack is Empty!" << endl;
            return T();
        }
        return arr[top];
    }

    bool isEmpty() {
        return top == -1;
    }
};

template <typename T>
class LinkedList {
    struct Node {
        T data;
        Node* next;
    };

    Node* head;

public:
    LinkedList() : head(nullptr) {}

    ~LinkedList() {
        Node* temp;
        while (head) {
            temp = head;
            head = head->next;
            delete temp;
        }
    }

    void insertFront(T data) {
        Node* newNode = new Node{data, head};
        head = newNode;
    }

    void insertEnd(T data) {
        Node* newNode = new Node{data, nullptr};
        if (!head) {
            head = newNode;
            return;
        }
        Node* temp = head;
        while (temp->next) {
            temp = temp->next;
        }
        temp->next = newNode;
    }

    void deleteNode(T data) {
        if (!head) return;

        if (head->data == data) {
            Node* temp = head;
            head = head->next;
            delete temp;
            return;
        }

        Node* temp = head;
        while (temp->next && temp->next->data != data) {
            temp = temp->next;
        }

        if (temp->next) {
            Node* toDelete = temp->next;
            temp->next = temp->next->next;
            delete toDelete;
        }
    }

    void display() {
        Node* temp = head;
        while (temp) {
            cout << temp->data << " -> ";
            temp = temp->next;
        }
        cout << "NULL" << endl;
    }
};

template <typename T>
class Deque {
    T* arr;
    int front, rear, capacity, count;

public:
    Deque(int size) {
        capacity = size;
        arr = new T[size];
        front = 0;
        rear = -1;
        count = 0;
    }

    ~Deque() { delete[] arr; }

    void insertFront(T data) {
        if (isFull()) {
            cout << "Deque Overflow!" << endl;
            return;
        }
        front = (front - 1 + capacity) % capacity;
        arr[front] = data;
        count++;
    }

    void insertRear(T data) {
        if (isFull()) {
            cout << "Deque Overflow!" << endl;
            return;
        }
        rear = (rear + 1) % capacity;
        arr[rear] = data;
        count++;
    }

    void deleteFront() {
        if (isEmpty()) {
            cout << "Deque Underflow!" << endl;
            return;
        }
        front = (front + 1) % capacity;
        count--;
    }

    void deleteRear() {
        if (isEmpty()) {
            cout << "Deque Underflow!" << endl;
            return;
        }
        rear = (rear - 1 + capacity) % capacity;
        count--;
    }

    bool isEmpty() {
        return count == 0;
    }

    bool isFull() {
        return count == capacity;
    }

    T getFront() {
        if (isEmpty()) {
            cout << "Deque is Empty!" << endl;
            return T();
        }
        return arr[front];
    }

    T getRear() {
        if (isEmpty()) {
            cout << "Deque is Empty!" << endl;
            return T();
        }
        return arr[rear];
    }
};

int main() {
    Stack<int> stack(5);
    stack.push(10);
    stack.push(20);
    stack.push(30);
    cout << "Stack Top: " << stack.peek() << endl;
    stack.pop();
    cout << "Stack Top after pop: " << stack.peek() << endl;

    LinkedList<int> list;
    list.insertFront(10);
    list.insertEnd(20);
    list.insertEnd(30);
    cout << "Linked List: ";
    list.display();
    list.deleteNode(20);
    cout << "After deletion: ";
    list.display();

    Deque<int> deque(5);
    deque.insertRear(10);
    deque.insertRear(20);
    deque.insertFront(5);
    cout << "Deque Front: " << deque.getFront() << endl;
    cout << "Deque Rear: " << deque.getRear() << endl;
    deque.deleteFront();
    cout << "Deque Front after deletion: " << deque.getFront() << endl;

    return 0;
}
```

## OUTPUT



```
Stack Top: 30
Stack Top after pop: 20
Linked List: 10 -> 20 -> 30 -> NULL
After deletion: 10 -> 30 -> NULL
Deque Front: 5
Deque Rear: 20
Deque Front after deletion: 10
```

# PRACTICAL No. 12

## AIM

Write a program to convert decimal to binary, decimal to octal and decimal to hexadecimal. Use user defined manipulator for implementation.

## THEORY

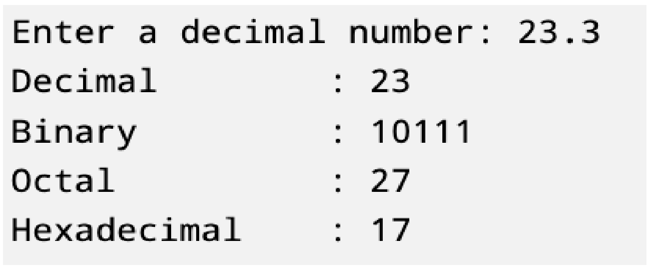
*Manipulators in C++* are used to modify the formatting of input and output streams. User-defined manipulators are custom functions that can be used like standard manipulators (e.g., std::setw, std::endl). For this program:

- We will use a user-defined manipulator to format the output.
- Conversion of decimal to other bases (binary, octal, hexadecimal) involves repeated division by the target base while recording remainders.

## PROGRAM

```
#include <iostream>
#include <iomanip>
#include <string>
#include <algorithm>
using namespace std;

ostream& formatOutput(ostream& os) {
    os << setw(15) << left;
    return os;
}

string decimalToBinary(int decimal) {
    string binary = "";
    while (decimal > 0) {
        binary += (decimal % 2) ? '1' : '0';
        decimal /= 2;
    }
    reverse(binary.begin(), binary.end());
    return binary.empty() ? "0" : binary;
}

string decimalToOctal(int decimal) {
    string octal = "";
    while (decimal > 0) {
        octal += to_string(decimal % 8);
        decimal /= 8;
    }
    reverse(octal.begin(), octal.end());
    return octal.empty() ? "0" : octal;
}

string decimalToHexadecimal(int decimal) {
    string hexadecimal = "";
    const string hexDigits = "0123456789ABCDEF";
    while (decimal > 0) {
        hexadecimal += hexDigits[decimal % 16];
        decimal /= 16;
    }
    reverse(hexadecimal.begin(), hexadecimal.end());
    return hexadecimal.empty() ? "0" : hexadecimal;
}

int main() {
    int decimal;

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

    cout << formatOutput << "Decimal" << ": " << decimal << endl;
    cout << formatOutput << "Binary" << ": " << decimalToBinary(decimal) <<
endl;
    cout << formatOutput << "Octal" << ": " << decimalToOctal(decimal) <<
endl;
    cout << formatOutput << "Hexadecimal" << ": " <<
decimalToHexadecimal(decimal) << endl;

    return 0;
}
```

## OUTPUT



```
Enter a decimal number: 23.3
Decimal         : 23
Binary          : 10111
Octal           : 27
Hexadecimal     : 17
```

# PRACTICAL No. 13

## AIM

Using stream manipulators display the contents of the string "Software" in pyramid format.

## THEORY

The program demonstrates the use of *string manipulation and stream formatting* to display a given string, "Software," in a pyramid format. Each line of the pyramid progressively increases in length by adding one character from the string until the entire string is displayed.

Key concepts include:

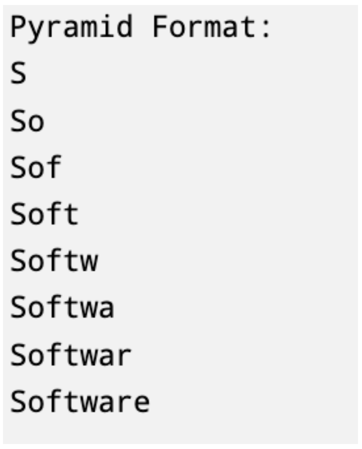
- *String Manipulation*: Characters from the string are accessed iteratively to build each row of the pyramid.
- *Loops*: A loop controls the number of characters printed per line, ensuring the pyramid structure.
- *Function Decomposition*: A helper function (displayPyramidRow) modularizes the task of printing characters, promoting code clarity and reusability.
- *Output Stream*: The std::cout stream is used to display the pyramid in a structured format.

This approach combines fundamental programming constructs like loops, string access, and functions to create a visually appealing output.

## PROGRAM

```
#include <iostream>
#include <iomanip>
#include <string>

using namespace std;

void displayPyramidRow(const string &word, int length) {
    for (int j = 0; j < length; j++) {
        cout << word[j];
    }
    cout << endl;
}

int main() {
    string word = "Software";
    int wordLength = word.length();

    cout << "Pyramid Format:" << endl;

    for (int i = 1; i <= wordLength; i++) {
        displayPyramidRow(word, i);
    }

    return 0;
}
```

## OUTPUT



```
Pyramid Format:
S
So
Sof
Soft
Softw
Softwa
Softwar
Software
```

# PRACTICAL No. 14

## AIM
Create a class whose name is Physical_Education. This class contains data members: name_of_game, cost, duration, place, finals and member functions: get_data(), put_data(). Overload input stream operator and store the data in ASCIl and binary format. Overload output stream operator and display the contents of the data file.

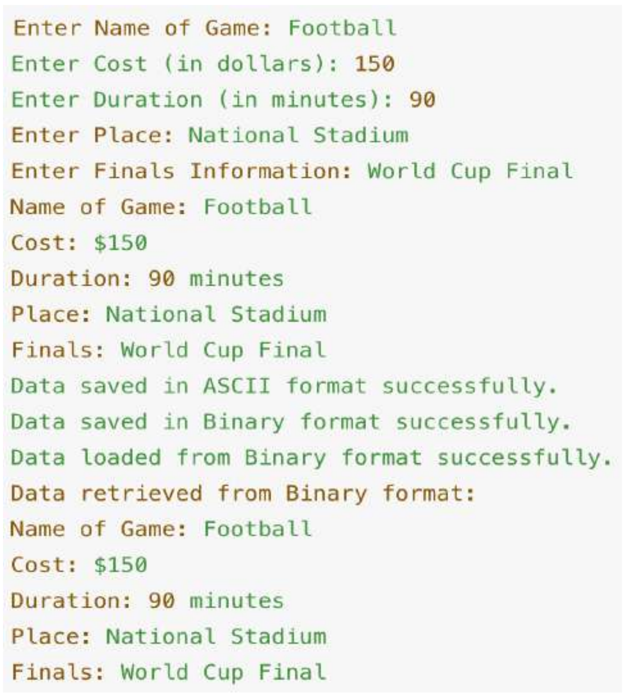
## THEORY
This program uses operator overloading, file handling, and stream manipulation. The Physical_Education class contains data members and functions to perform the following:

- *Input and Output*: Overloading >> and << operators for easy interaction with user-defined data.
- *File Handling*: Writing data into a file in both ASCII (human-readable) and binary (raw format) formats using fstream.
- *Serialization*: Using binary format to save object data ensures space efficiency and precise data recovery.
- *Data Organization*: Separation of concerns with methods for data input/output, and file storage.

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

using namespace std;

class Physical_Education {
private:
    string name_of_game;
    double cost;
    int duration;
    string place;
    string finals;
```

```
public:
    Physical_Education() : name_of_game(""), cost(0.0), duration(0),
place(""), finals("") {}

    friend istream& operator>>(istream& in, Physical_Education& obj) {
        cout << "Enter Name of Game: ";
        getline(in, obj.name_of_game);
        cout << "Enter Cost (in dollars): ";
        in >> obj.cost;
        cout << "Enter Duration (in minutes): ";
        in >> obj.duration;
        in.ignore();
        cout << "Enter Place: ";
        getline(in, obj.place);
        cout << "Enter Finals Information: ";
        getline(in, obj.finals);
        return in;
    }

    friend ostream& operator<<(ostream& out, const Physical_Education& obj) {
        out << "Name of Game: " << obj.name_of_game << endl;
        out << "Cost: $" << obj.cost << endl;
        out << "Duration: " << obj.duration << " minutes" << endl;
        out << "Place: " << obj.place << endl;
        out << "Finals: " << obj.finals << endl;
        return out;
    }

    void writeToASCII(const string& filename) const {
        ofstream outFile(filename, ios::out);
        if (outFile.is_open()) {
            outFile << *this;
            outFile.close();
            cout << "Data saved in ASCII format successfully." << endl;
        } else {
            cerr << "Error opening file for ASCII format!" << endl;
        }
    }

    void writeToBinary(const string& filename) const {
        ofstream outFile(filename, ios::binary | ios::out);
        if (outFile.is_open()) {
            outFile.write((char*)this, sizeof(Physical_Education));
            outFile.close();
            cout << "Data saved in Binary format successfully." << endl;
        } else {
            cerr << "Error opening file for Binary format!" << endl;
        }
    }

    void readFromBinary(const string& filename) {
        ifstream inFile(filename, ios::binary | ios::in);
        if (inFile.is_open()) {
            inFile.read((char*)this, sizeof(Physical_Education));
            inFile.close();
            cout << "Data loaded from Binary format successfully." << endl;
        } else {
            cerr << "Error opening file for reading Binary format!" << endl;
        }
    }
};

int main() {
    Physical_Education game;
    cin >> game;
    cout << game;

    string asciiFile = "game_data.txt";
    string binaryFile = "game_data.dat";

    game.writeToASCII(asciiFile);

    game.writeToBinary(binaryFile);

    Physical_Education gameFromBinary;
    gameFromBinary.readFromBinary(binaryFile);

    cout << "Data retrieved from Binary format:" << endl;
    cout << gameFromBinary;

    return 0;
}
```

## OUTPUT



```
Enter Name of Game: Football
Enter Cost (in dollars): 150
Enter Duration (in minutes): 90
Enter Place: National Stadium
Enter Finals Information: World Cup Final
Name of Game: Football
Cost: $150
Duration: 90 minutes
Place: National Stadium
Finals: World Cup Final
Data saved in ASCII format successfully.
Data saved in Binary format successfully.
Data loaded from Binary format successfully.
Data retrieved from Binary format:
Name of Game: Football
Cost: $150
Duration: 90 minutes
Place: National Stadium
Finals: World Cup Final
```

# PRACTICAL No. 15

## AIM

Create a text data file. Write a program to count the number of characters, number of words, sentences, paragraphs and control characters. Use 1/0 stream functions.

## THEORY

In C++, file handling is done using input/output stream classes like ifstream (for reading from files) and ofstream (for writing to files). To process files efficiently, functions like open(), close(), and get() are used. In this experiment, we focus on reading from a file using the get() function and counting different elements in the text.

The program reads a file character by character and counts:

- *Characters*: Every character, including spaces, punctuation, and newlines.
- *Words*: Words are sequences of characters separated by spaces, tabs, or newlines.
- *Sentences*: Sentences are counted by detecting punctuation marks like period (.), question mark (?), and exclamation mark (!).
- *Paragraphs*: Paragraphs are counted by detecting two or more consecutive newline characters.
- *Control Characters*: These are non-printable characters like newline (\n), tab (\t), etc. These can be detected using the iscntrl() function.

By processing the file character by character, the program efficiently counts these elements, providing an overview of the structure of the text file

## PROGRAM

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

void countFileContents(const string& fileName) {
    ifstream file(fileName);
```

```
if (!file) {
    cout << "Error opening the file!" << endl;
    return;
}

int charCount = 0;
int wordCount = 0;
int sentenceCount = 0;
int paragraphCount = 0;
int controlCharCount = 0;

string word;
bool inParagraph = false;

char ch;
while (file.get(ch)) {
    charCount++;

    if (iscntrl(ch)) {
        controlCharCount++;
    }

    if (isspace(ch) || ch == '\n' || ch == '\r' || ch == '\t') {
        if (!word.empty()) {
            wordCount++;
            word.clear();
        }
    } else {
        word += ch;
    }

    if (ch == '.' || ch == '?' || ch == '!') {
        sentenceCount++;
    }

    if (ch == '\n') {
        if (inParagraph) {
            paragraphCount++;
            inParagraph = false;
        } else {
            inParagraph = true;
        }
    }
}
```

```
    if (!word.empty()) {
        wordCount++;
    }

    if (inParagraph) {
        paragraphCount++;
    }

    cout << "Total Characters: " << charCount << endl;
    cout << "Total Words: " << wordCount << endl;
    cout << "Total Sentences: " << sentenceCount << endl;
    cout << "Total Paragraphs: " << paragraphCount << endl;
    cout << "Total Control Characters: " << controlCharCount << endl;

    file.close();
}

int main() {
    string fileName = "sample.txt";
    countFileContents(fileName);
    return 0;
}
```

## OUTPUT

Sample '.txt' file:

```
This is the first sentence. It is followed by a second sentence.

This is a new paragraph.
It has multiple lines, and it ends here.
```

```
Total Characters: 171
Total Words: 31
Total Sentences: 4
Total Paragraphs: 2
Total Control Characters: 9
```

# PRACTICAL No. 16

## AIM

Consider that the base class stack is available. It does not take care of situations such as overflow and underflow. Enhance this class to MyStack which raises an exception whenever overflow and underflow occurs. Implement various types of exceptions.

## THEORY

In C++, exception handling allows programs to handle runtime errors gracefully. The try, catch, and throw keywords are used to manage exceptions. Custom exceptions can be created by throwing objects or by defining exception classes.

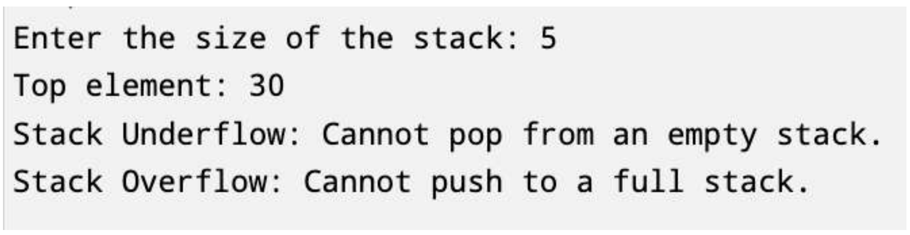
For this experiment:

- The *base class* (Stack) implements basic stack operations (push, pop).
- The *derived class* (MyStack) extends the base class and:
  - Raises an *overflow exception* when trying to push onto a full stack.
  - Raises an *underflow exception* when trying to pop from an empty stack.
- *Custom exception classes* are implemented to define specific error messages for overflow and underflow.

## PROGRAM

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

class Stack {
protected:
    int* arr;
    int top;
    int capacity;

public:
    Stack(int size) : capacity(size), top(-1) {
        arr = new int[capacity];
    }
    virtual ~Stack() {
        delete[] arr;
    }

    virtual void push(int value) {
        arr[++top] = value;
    }

    virtual int pop() {
        return arr[top--];
    }

    bool isEmpty() const {
        return top == -1;
    }

    bool isFull() const {
        return top == capacity - 1;
    }

    int peek() const {
        return arr[top];
    }
};

class OverflowException : public exception {
public:
    const char* what() const noexcept override {
        return "Stack Overflow: Cannot push to a full stack.";
    }
};

class UnderflowException : public exception {
public:
    const char* what() const noexcept override {
        return "Stack Underflow: Cannot pop from an empty stack.";
    }
};

class MyStack : public Stack {
public:
    MyStack(int size) : Stack(size) {}

    void push(int value) override {
        if (isFull()) {
            throw OverflowException();
        }
        Stack::push(value);
    }

    int pop() override {
        if (isEmpty()) {
            throw UnderflowException();
        }
        return Stack::pop();
    }
};

int main() {
    int size;
    cout << "Enter the size of the stack: ";
    cin >> size;

    MyStack stack(size);

    try {
        stack.push(10);
        stack.push(20);
        stack.push(30);

        cout << "Top element: " << stack.peek() << endl;

        stack.pop();
        stack.pop();
        stack.pop();

        stack.pop();
    }
    catch (const exception& e) {
        cerr << e.what() << endl;
    }

    try {
        for (int i = 0; i <= size; i++) {
            stack.push(i * 10);
        }
    }
    catch (const exception& e) {
        cerr << e.what() << endl;
    }

    return 0;
}
```

## OUTPUT



```
Enter the size of the stack: 5
Top element: 30
Stack Underflow: Cannot pop from an empty stack.
Stack Overflow: Cannot push to a full stack.
```

# PRACTICAL No. 17

## AIM

To create a generic algorithm for sorting a set of 10 numbers using function templates. The program will sort numbers of different data types (e.g., integers, floating-point numbers) in ascending order.

## THEORY

A generic algorithm is implemented using function templates in C++, which allows functions to operate with generic types. A template enables the same function to work with different data types, avoiding redundancy and enhancing code reusability.

The sorting algorithm used is Bubble Sort:

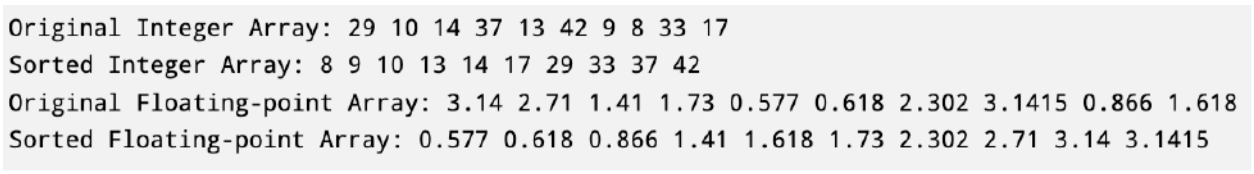
- Compares adjacent elements and swaps them if they are in the wrong order.
- Repeated until the list is sorted.

The function template provides a type-independent implementation of the sorting logic

## PROGRAM

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

template <typename T>
void sortArray(T arr[], int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                T temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
}

template <typename T>
void displayArray(const T arr[], int size) {
    for (int i = 0; i < size; i++) {
        cout << arr[i] << " ";
    }
    cout << endl;
}

int main() {
    int intArr[] = {29, 10, 14, 37, 13, 42, 9, 8, 33, 17};
    int intSize = sizeof(intArr) / sizeof(intArr[0]);

    double floatArr[] = {3.14, 2.71, 1.41, 1.73, 0.577, 0.618, 2.302, 3.1415,
0.866, 1.618};
    int floatSize = sizeof(floatArr) / sizeof(floatArr[0]);

    cout << "Original Integer Array: ";
    displayArray(intArr, intSize);

    sortArray(intArr, intSize);
    cout << "Sorted Integer Array: ";
    displayArray(intArr, intSize);

    cout << "Original Floating-point Array: ";
    displayArray(floatArr, floatSize);

    sortArray(floatArr, floatSize);
    cout << "Sorted Floating-point Array: ";
    displayArray(floatArr, floatSize);

    return 0;
}
```

## OUTPUT



```
Original Integer Array: 29 10 14 37 13 42 9 8 33 17
Sorted Integer Array: 8 9 10 13 14 17 29 33 37 42
Original Floating-point Array: 3.14 2.71 1.41 1.73 0.577 0.618 2.302 3.1415 0.866 1.618
Sorted Floating-point Array: 0.577 0.618 0.866 1.41 1.618 1.73 2.302 2.71 3.14 3.1415
```

# PRACTICAL No. 18

## AIM

Write a program to illustrate the operations provided by the Set Container, iterators and allocators.

## THEORY

The std::set is a container in the C++ Standard Template Library (STL) that stores unique elements in a sorted order. Key properties of set include:

- *Automatic Sorting*: Elements are sorted in ascending order by default.
- *Uniqueness*: No duplicate elements are allowed.
- *Efficient Lookup*: Insertion, deletion, and search operations have logarithmic time complexity.

*Iterators* allow traversal through the elements of the set. Iterators can be forward, reverse, or constant.

*Allocators* are used by STL containers to manage dynamic memory efficiently. The get_allocator() function in std::set retrieves the allocator used.

## PROGRAM

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

int main() {
    set<int> mySet;
    mySet.insert(10);
    mySet.insert(20);
    mySet.insert(30);
    mySet.insert(40);
    mySet.insert(50);
    cout << "Set elements after insertion: ";
    for (auto it = mySet.begin(); it != mySet.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;
    mySet.insert(30);
    cout << "Set elements after attempting to insert duplicate (30): ";
    for (auto it = mySet.begin(); it != mySet.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;
    mySet.erase(20);
    cout << "Set elements after deleting 20: ";
    for (auto it = mySet.begin(); it != mySet.end(); ++it) {
        cout << *it << " ";
    }
    cout << endl;
    cout << "Set elements in reverse order: ";
    for (auto rit = mySet.rbegin(); rit != mySet.rend(); ++rit) {
        cout << *rit << " ";
    }
    cout << endl;
    int value = 30;
    if (mySet.find(value) != mySet.end()) {
        cout << value << " is present in the set." << endl;
    } else {
        cout << value << " is not present in the set." << endl;
    }
    set<int>::allocator_type alloc = mySet.get_allocator();
    int* p = alloc.allocate(5);
    for (int i = 0; i < 5; ++i) {
        alloc.construct(&p[i], i + 1);
    }
    cout << "Memory allocated using set allocator: ";
    for (int i = 0; i < 5; ++i) {
        cout << p[i] << " ";
        alloc.destroy(&p[i]);
    }
    cout << endl;
    alloc.deallocate(p, 5);
    return 0;
}
```

## OUTPUT

```
Set elements after insertion: 10 20 30 40 50
Set elements after attempting to insert duplicate (30): 10 20 30 40 50
Set elements after deleting 20: 10 30 40 50
Set elements in reverse order: 50 40 30 10
30 is present in the set.
Memory allocated using set allocator: 1 2 3 4 5
```