Implement a base class Shape with derived classes Circle, Rectangle, and Triangle. Use virtual functions to calculate the area of each shape.

```
#include <iostream>

using namespace std;


class Shape {

public:

   virtual float area() = 0; // Pure virtual

};


class Circle : public Shape {

   float radius;

public:

   Circle(float r) : radius(r) {}

   float area() override {

      return 3.14 * radius * radius;

   }

};


class Rectangle : public Shape {

   float length, width;

public:

   Rectangle(float l, float w) : length(l), width(w) {}

   float area() override {

      return length * width;

   }
```

```
};

class Triangle : public Shape {
    float base, height;
public:
    Triangle(float b, float h) : base(b), height(h) {}
    float area() override {
        return 0.5 * base * height;
    }
};

int main() {
    Shape* s;
    Circle c(5);
    Rectangle r(4, 6);
    Triangle t(3, 7);

    s = &c; cout << "Circle Area: " << s->area() << endl;
    s = &r; cout << "Rectangle Area: " << s->area() << endl;
    s = &t; cout << "Triangle Area: " << s->area() << endl;

    return 0;
}
```

Create a base class Animal with a virtual function speak(). Implement derived classes Dog, Cat, and Bird, each overriding the speak() function.

```
class Animal {
```

```cpp
public:
    virtual void speak() { cout << "Animal speaks\n"; }
};

class Dog : public Animal {
public:
    void speak() override { cout << "Dog barks\n"; }
};

class Cat : public Animal {
public:
    void speak() override { cout << "Cat meows\n"; }
};

class Bird : public Animal {
public:
    void speak() override { cout << "Bird chirps\n"; }
};

int main() {
    Animal* a;
    Dog d; Cat c; Bird b;

    a = &d; a->speak();
    a = &c; a->speak();
```

```
    a = &b; a->speak();

    return 0;
}
```

Write a program that demonstrates function overriding using a base class Employee and derived classes Manager and Worker.

```
class Employee {
public:
    virtual void work() { cout << "Employee working...\n"; }
};

class Manager : public Employee {
public:
    void work() override { cout << "Manager planning...\n"; }
};

class Worker : public Employee {
public:
    void work() override { cout << "Worker executing...\n"; }
};

int main() {
    Employee* e;
    Manager m;
    Worker w;
```

```
    e = &m; e->work();
    e = &w; e->work();

    return 0;
}
```

Write a program to demonstrate pointer arithmetic by creating an array and accessing its elements using pointers.

```
int main() {
    int arr[] = {10, 20, 30, 40};
    int* ptr = arr;

    for (int i = 0; i < 4; i++) {
        cout << "Value at arr[" << i << "] = " << *(ptr + i) << endl;
    }

    return 0;
}
```

Implement a program that dynamically allocates memory for an integer array and initializes it using pointers.

```
int main() {
    int n;
    cout << "Enter size: ";
    cin >> n;

    int* arr = new int[n];
    for (int i = 0; i < n; i++) {
```

```
        arr[i] = i + 1;
    }

    cout << "Array Elements: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    delete[] arr;
    return 0;
}
```

Create a program that uses a pointer to swap the values of two variables.

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

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 10, y = 20;
    swap(&x, &y);
    cout << "x = " << x << ", y = " << y << endl;
```

```
    return 0;
}
```

Write a program that creates a dynamic object of a class Student and accesses its members using pointers.

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


class Student {
    string name;
    int age;
public:
    void setData(string n, int a) {
        name = n; age = a;
    }
    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};


int main() {
    Student* s = new Student;
    s->setData("Sakshi", 18);
    s->display();
    delete s;
    return 0;
}
```

Implement a program that uses a pointer to an array of objects to store and display details of multiple Book objects.

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

class Book {
    string title;
    float price;
public:
    void set(string t, float p) {
        title = t;
        price = p;
    }
    void display() {
        cout << "Title: " << title << ", Price: " << price << endl;
    }
};

int main() {
    int n = 2;
    Book* books = new Book[n];

    books[0].set("C++ Basics", 299.99);
    books[1].set("OOP Concepts", 399.99);

    for (int i = 0; i < n; i++) {
```

```
        books[i].display();
    }

    delete[] books;
    return 0;
}
```

Create a program that demonstrates the use of a pointer to an object in a class member function.

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

class Demo {
    int value;
public:
    void set(int val) {
        value = val;
    }
    void show(Demo* ptr) {
        cout << "Value from pointer: " << ptr->value << endl;
    }
};

int main() {
    Demo d1;
    d1.set(50);
    d1.show(&d1);
```

```
    return 0;
}
```

Write a class Box with a member function that returns the current object using the this pointer.

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

class Box {
    int length;
public:
    Box(int l) {
        this->length = l;
    }
    Box& getObject() {
        return *this;
    }
    void display() {
        cout << "Length: " << length << endl;
    }
};

int main() {
    Box b1(15);
    Box& b2 = b1.getObject();
    b2.display();
    return 0;
```

```
}
```

Implement a program that uses the this pointer to chain member function calls in a class Person.

```
#include <iostream>

#include <string>

using namespace std;


class Person {

   string name;

   int age;

public:

   Person& setName(string n) {

      name = n;

      return *this;

   }

   Person& setAge(int a) {

      age = a;

      return *this;

   }

   void display() {

      cout << "Name: " << name << ", Age: " << age << endl;

   }

};


int main() {

   Person p;
```

```
    p.setName("Sakshi").setAge(20).display();
    return 0;
}
```

Create a class Counter with a member function that compares two objects using the this pointer.

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

class Counter {
    int count;
public:
    Counter(int c) : count(c) {}
    void compare(Counter& other) {
        if (this->count > other.count)
            cout << "Current object has a higher count." << endl;
        else if (this->count < other.count)
            cout << "Other object has a higher count." << endl;
        else
            cout << "Both objects have equal count." << endl;
    }
};

int main() {
    Counter c1(10), c2(20);
    c1.compare(c2);
    return 0;
```

```
}
```

Write a program that uses pure virtual functions to create an abstract class Vehicle with derived classes Car and Bike.

```
#include <iostream>

using namespace std;


class Vehicle {

public:

    virtual void start() = 0; // Pure virtual function

};


class Car : public Vehicle {

public:

    void start() override {

        cout << "Car started." << endl;

    }

};


class Bike : public Vehicle {

public:

    void start() override {

        cout << "Bike started." << endl;

    }

};


int main() {
```

```cpp
    Vehicle* v;
    Car c;
    Bike b;

    v = &c;
    v->start();

    v = &b;
    v->start();

    return 0;
}
```

Implement a program that demonstrates runtime polymorphism using a virtual function in a base class Shape and derived classes Circle and Square.

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

class Shape {
public:
    virtual void draw() {
        cout << "Drawing Shape." << endl;
    }
};

class Circle : public Shape {
public:
```

```
    void draw() override {
        cout << "Drawing Circle." << endl;
    }
};

class Square : public Shape {
public:
    void draw() override {
        cout << "Drawing Square." << endl;
    }
};

int main() {
    Shape* s;
    Circle c;
    Square sq;

    s = &c;
    s->draw();

    s = &sq;
    s->draw();

    return 0;
}
```

Create a class Account with a pure virtual function calculateInterest(). Implement derived classes SavingsAccount and CurrentAccount.

```
#include <iostream>

using namespace std;


class Account {

public:

    virtual void calculateInterest() = 0; // Pure virtual function

};


class SavingsAccount : public Account {

public:

    void calculateInterest() override {

        cout << "Calculating interest for Savings Account." << endl;

    }

};


class CurrentAccount : public Account {

public:

    void calculateInterest() override {

        cout << "Calculating interest for Current Account." << endl;

    }

};


int main() {

    Account* acc;
```

```
    SavingsAccount sa;
    CurrentAccount ca;

    acc = &sa;
    acc->calculateInterest();

    acc = &ca;
    acc->calculateInterest();

    return 0;
}
```

Write a program that demonstrates polymorphism using a base class Media and derived classes Book and DVD.

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

class Media {
public:
    virtual void display() {
        cout << "Displaying Media." << endl;
    }
};

class Book : public Media {
public:
    void display() override {
```

```
        cout << "Displaying Book." << endl;
    }
};

class DVD : public Media {
public:
    void display() override {
        cout << "Displaying DVD." << endl;
    }
};

int main() {
    Media* m;
    Book b;
    DVD d;

    m = &b;
    m->display();

    m = &d;
    m->display();

    return 0;
}
```

Implement a class hierarchy with a base class Appliance and derived classes WashingMachine, Refrigerator, and Microwave. Use virtual functions to display the functionality of each appliance.

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

class Appliance {
public:
    virtual void functionality() {
        cout << "General Appliance Functionality." << endl;
    }
};

class WashingMachine : public Appliance {
public:
    void functionality() override {
        cout << "Washing clothes." << endl;
    }
};

class Refrigerator : public Appliance {
public:
    void functionality() override {
        cout << "Cooling food items." << endl;
    }
};

class Microwave : public Appliance {
```

```
public:
    void functionality() override {
        cout << "Heating food." << endl;
    }
};

int main() {
    Appliance* a;
    WashingMachine wm;
    Refrigerator rf;
    Microwave mw;

    a = &wm;
    a->functionality();

    a = &rf;
    a->functionality();

    a = &mw;
    a->functionality();

    return 0;
}
```

Create a program that uses polymorphism to calculate the area of different geometric shapes using a base class Shape and derived classes Circle and Rectangle.

```
#include <iostream>
```

```
using namespace std;

class Shape {
public:
    virtual void area() = 0; // Pure virtual function
};

class Circle : public Shape {
    float radius;
public:
    Circle(float r) : radius(r) {}
    void area() override {
        cout << "Area of Circle: " << 3.14 * radius * radius << endl;
    }
};

class Rectangle : public Shape {
    float length, breadth;
public:
    Rectangle(float l, float b) : length(l), breadth(b) {}
    void area() override {
        cout << "Area of Rectangle: " << length * breadth << endl;
    }
};
```

```
int main() {
    Shape* s;
    Circle c(5);
    Rectangle r(4, 6);

    s = &c;
    s->area();

    s = &r;
    s->area();

    return 0;
}
```

Write an abstract class Employee with pure virtual functions calculateSalary() and displayDetails(). Implement derived classes Manager and Engineer.

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

class Employee {
public:
    virtual void calculateSalary() = 0;
    virtual void displayDetails() = 0;
};

class Manager : public Employee {
public:
```

```
    void calculateSalary() override {
        cout << "Calculating salary for Manager." << endl;
    }
    void displayDetails() override {
        cout << "Manager Details." << endl;
    }
};

class Engineer : public Employee {
public:
    void calculateSalary() override {
        cout << "Calculating salary for Engineer." << endl;
    }
    void displayDetails() override {
        cout << "Engineer Details." << endl;
    }
};

int main() {
    Employee* e;
    Manager m;
    Engineer eng;

    e = &m;
    e->calculateSalary();
```

```cpp
    e->displayDetails();

    e = &eng;
    e->calculateSalary();
    e->displayDetails();

    return 0;
}
```

Implement an abstract class Payment with a pure virtual function processPayment(). Create derived classes CreditCardPayment and DebitCardPayment.

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

class Payment {
public:
    virtual void processPayment() = 0;
};

class CreditCardPayment : public Payment {
public:
    void processPayment() override {
        cout << "Processing credit card payment." << endl;
    }
};

class DebitCardPayment : public Payment {
```

```cpp
public:
    void processPayment() override {
        cout << "Processing debit card payment." << endl;
    }
};

int main() {
    Payment* p;
    CreditCardPayment ccp;
    DebitCardPayment dcp;

    p = &ccp;
    p->processPayment();

    p = &dcp;
    p->processPayment();

    return 0;
}
```

Create an abstract class Device with a pure virtual function turnOn(). Implement derived classes Laptop and Smartphone.

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

class Device {
public:
```

```
    virtual void turnOn() = 0; // Pure virtual function
    virtual ~Device() {}
};

class Laptop : public Device {
public:
    void turnOn() override {
        cout << "Laptop is turning on." << endl;
    }
};

class Smartphone : public Device {
public:
    void turnOn() override {
        cout << "Smartphone is turning on." << endl;
    }
};

int main() {
    Device* d1 = new Laptop();
    Device* d2 = new Smartphone();
    d1->turnOn();
    d2->turnOn();
    delete d1;
    delete d2;
```

```cpp
    return 0;
}
```

Write a program that handles division by zero using exception handling.

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

int main() {
    int numerator = 10, denominator = 0;
    try {
        if (denominator == 0)
            throw runtime_error("Division by zero error");
        cout << "Result: " << numerator / denominator << endl;
    } catch (const exception& e) {
        cerr << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

Implement a program that demonstrates the use of multiple catch blocks to handle different types of exceptions.

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

int main() {
    try {
```

```
        throw runtime_error("Runtime error occurred");
    } catch (const logic_error& e) {
        cerr << "Logic error: " << e.what() << endl;
    } catch (const runtime_error& e) {
        cerr << "Runtime error: " << e.what() << endl;
    } catch (...) {
        cerr << "Unknown exception caught" << endl;
    }
    return 0;
}
```

Create a custom exception class InvalidAgeException and use it to handle invalid age input in a program.

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

class InvalidAgeException : public exception {
public:
    const char* what() const noexcept override {
        return "Invalid age entered";
    }
};

int main() {
    int age;
    cout << "Enter age: ";
```

```
    cin >> age;
    try {
        if (age < 0 || age > 150)
            throw InvalidAgeException();
        cout << "Valid age: " << age << endl;
    } catch (const InvalidAgeException& e) {
        cerr << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

Write a program that uses exception handling to manage file input/output errors.

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

int main() {
    ifstream file("nonexistent.txt");
    try {
        if (!file)
            throw runtime_error("File could not be opened");
        // File processing logic here
    } catch (const exception& e) {
        cerr << "Exception: " << e.what() << endl;
    }
```

```
    return 0;
}
```

Implement a program that demonstrates the use of the finally block to release resources in exception handling.

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

int main() {
    try {
        ifstream file("data.txt");
        if (!file)
            throw runtime_error("Failed to open file");
        // File processing logic here
    } catch (const exception& e) {
        cerr << "Exception: " << e.what() << endl;
    }
    // File is automatically closed when it goes out of scope
    return 0;
}
```

Write a function template to perform a linear search on an array of any data type.

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

template <typename T>
int linearSearch(T arr[], int size, T key) {
```

```cpp
    for (int i = 0; i < size; ++i)
        if (arr[i] == key)
            return i;
    return -1;
}

int main() {
    int arr[] = {1, 3, 5, 7, 9};
    int index = linearSearch(arr, 5, 7);
    if (index != -1)
        cout << "Element found at index " << index << endl;
    else
        cout << "Element not found" << endl;
    return 0;
}
```

Implement a class template Stack with member functions to push, pop, and display elements.

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

template <typename T>
class Stack {
    T arr[100];
    int top;
public:
    Stack() : top(-1) {}
```

```
    void push(T val) {
        if (top < 99)
            arr[++top] = val;
        else
            cout << "Stack overflow" << endl;
    }
    void pop() {
        if (top >= 0)
            --top;
        else
            cout << "Stack underflow" << endl;
    }
    void display() {
        for (int i = top; i >= 0; --i)
            cout << arr[i] << " ";
        cout << endl;
    }
};

int main() {
    Stack<int> s;
    s.push(10);
    s.push(20);
    s.display();
    s.pop();
```

```
    s.display();
    return 0;
}
```

Create a function template to find the maximum of two values of any data type.

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

template <typename T>
T maxVal(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << "Max: " << maxVal(10, 20) << endl;
    cout << "Max: " << maxVal(3.5, 2.5) << endl;
    return 0;
}
```

Write a class template LinkedList with member functions to insert, delete, and display nodes.

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

template <typename T>
class LinkedList {
    struct Node {
        T data;
```

```cpp
        Node* next;
    };
    Node* head;
public:
    LinkedList() : head(nullptr) {}
    void insert(T val) {
        Node* newNode = new Node{val, head};
        head = newNode;
    }
    void remove(T val) {
        Node** curr = &head;
        while (*curr) {
            if ((*curr)->data == val) {
                Node* temp = *curr;
                *curr = (*curr)->next;
                delete temp;
                return;
            }
            curr = &((*curr)->next);
        }
    }
    void display() {
        Node* curr = head;
        while (curr) {
            cout << curr->data << " ";
```

```
            curr = curr->next;
        }
        cout << endl;
    }
};

int main() {
    LinkedList<int> list;
    list.insert(10);
    list.insert(20);
    list.display();
    list.remove(10);
    list.display();
    return 0;
}
```

Implement a function template to perform bubble sort on an array of any data type.

```
#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])
                swap(arr[j], arr[j + 1]);
```

```
}


int main() {
    int arr[] = {5, 2, 9, 1, 5};
    bubbleSort(arr, 5);
    for (int i = 0; i < 5; ++i)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Create a class template Queue with member functions to enqueue, dequeue, and display elements.

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


template <typename T>
class Queue {
    T arr[100];
    int front, rear;
public:
    Queue() : front(0), rear(0) {}
    void enqueue(T val) {
        if (rear < 100)
            arr[rear++] = val;
        else
            cout << "Queue overflow" << endl;
```

```
    }
    void dequeue() {
        if (front < rear)
            ++front;
        else
            cout << "Queue underflow" << endl;
    }
    void display() {
        for (int i = front; i < rear; ++i)
            cout << arr[i] << " ";
        cout << endl;
    }
};

int main() {
    Queue<int> q;
    q.enqueue(10);
    q.enqueue(20);
    q.display();
    q.dequeue();
    q.display();
    return 0;
}
```

Write a program that uses polymorphism to create a menu-driven application for managing different types of bank accounts.

```
#include <iostream>
```

```cpp
using namespace std;

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

class Savings : public Account {
public:
    void display() override {
        cout << "Savings Account" << endl;
    }
};

class Current : public Account {
public:
    void display() override {
        cout << "Current Account" << endl;
    }
};

int main() {
    Account* acc;
    int choice;
```

```
    cout << "1. Savings\n2. Current\nEnter choice: ";
    cin >> choice;
    if (choice == 1)
        acc = new Savings();
    else
        acc = new Current();
    acc->display();
    delete acc;
    return 0;
}
```

Implement a program that demonstrates the use of smart pointers for dynamic memory management.

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

class Demo {
public:
    Demo() { cout << "Constructor\n"; }
    ~Demo() { cout << "Destructor\n"; }
    void show() { cout << "Demo function\n"; }
};

int main() {
    unique_ptr<Demo> ptr = make_unique<Demo>();
    ptr->show();
```

```
    return 0;
}
```

Create a program that uses exception handling and templates to implement a safe array class.

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

template <typename T>
class SafeArray {
    T arr[100];
public:
    T& operator[](int index) {
        if (index < 0 || index >= 100)
            throw out_of_range("Index out of bounds");
        return arr[index];
    }
};

int main() {
```

::contentReference[oaicite:0]{index=0}

Write a program that demonstrates the use of virtual inheritance to avoid the diamond problem in multiple inheritance.

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

```
class A {
public:
    void show() {
        cout << "Class A\n";
    }
};

class B : virtual public A {};
class C : virtual public A {};
class D : public B, public C {};

int main() {
    D obj;
    obj.show();  // Only one copy of A
    return 0;
}
```

Implement a class Polynomial with member functions to add and multiply polynomials using operator overloading.

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

class Polynomial {
    vector<int> coeffs;
```

```
public:
    Polynomial(vector<int> c) : coeffs(c) {}

    Polynomial operator+(const Polynomial& p) {
        vector<int> result(max(coeffs.size(), p.coeffs.size()), 0);
        for (size_t i = 0; i < coeffs.size(); ++i)
            result[i] += coeffs[i];
        for (size_t i = 0; i < p.coeffs.size(); ++i)
            result[i] += p.coeffs[i];
        return Polynomial(result);
    }

    void display() {
        for (int i = coeffs.size() - 1; i >= 0; --i)
            cout << coeffs[i] << "x^" << i << (i ? " + " : "\n");
    }
};

int main() {
    Polynomial p1({3, 2, 1}); // 1x^2 + 2x^1 + 3x^0
    Polynomial p2({1, 4});    // 4x^1 + 1x^0
    Polynomial sum = p1 + p2;
    sum.display();
    return 0;
}
```

Create a program that uses function pointers to implement a callback mechanism.

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

void greet() {
    cout << "Hello, this is a callback function!\n";
}

void callFunction(void (*fptr)()) {
    fptr();
}

int main() {
    callFunction(greet);
    return 0;
}
```

Write a program that uses class templates and exception handling to implement a generic and robust data structure.

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

template <typename T>
class SafeArray {
    T* arr;
    int size;
```

```
public:
    SafeArray(int n) : size(n) {
        arr = new T[size];
    }

    T& operator[](int index) {
        if (index < 0 || index >= size)
            throw out_of_range("Index out of range");
        return arr[index];
    }

    ~SafeArray() {
        delete[] arr;
    }
};

int main() {
    try {
        SafeArray<int> arr(5);
        arr[0] = 10;
        cout << arr[0] << endl;
        cout << arr[10] << endl;  // Out of range
    } catch (exception& e) {
        cout << "Error: " << e.what() << endl;
```

```
    }
    return 0;
}
```

Implement a program that demonstrates the use of virtual destructors in a class hierarchy.

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

class Base {
public:
    virtual ~Base() {
        cout << "Base destructor\n";
    }
};

class Derived : public Base {
public:
    ~Derived() {
        cout << "Derived destructor\n";
    }
};

int main() {
    Base* ptr = new Derived();
    delete ptr;
    return 0;
```

}

Create a program that uses a function template to perform generic matrix operations (addition, multiplication).

```
#include <iostream>

using namespace std;


template <typename T>

void addMatrices(T a[2][2], T b[2][2], T res[2][2]) {

    for (int i = 0; i < 2; ++i)

        for (int j = 0; j < 2; ++j)

            res[i][j] = a[i][j] + b[i][j];

}


int main() {

    int A[2][2] = {{1, 2}, {3, 4}}, B[2][2] = {{5, 6}, {7, 8}}, C[2][2];

    addMatrices(A, B, C);

    for (auto& row : C) {

        for (int val : row) cout << val << " ";

        cout << endl;

    }

    return 0;

}
```

Write a program that uses polymorphism to create a plugin system for a software application.

```
#include <iostream>

using namespace std;
```

```
class Plugin {
public:
    virtual void execute() = 0;
};

class PluginA : public Plugin {
public:
    void execute() override {
        cout << "PluginA executed.\n";
    }
};

int main() {
    Plugin* p = new PluginA();
    p->execute();
    delete p;
    return 0;
}
```

Implement a program that uses class templates to create a generic binary tree data structure.

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

template <typename T>
class Node {
public:
```

```
    T data;
    Node* left;
    Node* right;

    Node(T d) : data(d), left(NULL), right(NULL) {}
};

template <typename T>
class BinaryTree {
    Node<T>* root;

    void inorder(Node<T>* node) {
        if (node) {
            inorder(node->left);
            cout << node->data << " ";
            inorder(node->right);
        }
    }

public:
    BinaryTree() : root(NULL) {}

    void insert(T data) {
        root = new Node<T>(data);  // Simplified
    }
```

```
    void display() {
        inorder(root);
    }
};

int main() {
    BinaryTree<int> tree;
    tree.insert(10);
    tree.display();
    return 0;
}
```

Create a program that demonstrates the use of polymorphism to implement a dynamic dispatch mechanism.

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

class UIComponent {
public:
    virtual void draw() = 0;
};

class Button : public UIComponent {
public:
    void draw() override {
        cout << "Drawing Button\n";
```

```
    }
};

int main() {
    UIComponent* comp = new Button();
    comp->draw();
    delete comp;
    return 0;
}
```

Write a program that uses smart pointers and templates to implement a memory-efficient and type-safe container.

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

template <typename T>
class Container {
    unique_ptr<T[]> data;
    int size;

public:
    Container(int s) : size(s), data(new T[s]) {}

    T& operator[](int i) { return data[i]; }

    void display() {
```

```
        for (int i = 0; i < size; ++i)
            cout << data[i] << " ";
        cout << endl;
    }
};

int main() {
    Container<int> c(3);
    c[0] = 1; c[1] = 2; c[2] = 3;
    c.display();
    return 0;
}
```

Implement a program that uses virtual functions and inheritance to create a simulation of an ecosystem with different types of animals.

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

class Animal {
public:
    virtual void sound() = 0;
};

class Lion : public Animal {
public:
    void sound() override {
        cout << "Roar\n";
```

```
    }
};

class Bird : public Animal {
public:
    void sound() override {
        cout << "Chirp\n";
    }
};

int main() {
    Animal* a1 = new Lion();
    Animal* a2 = new Bird();
    a1->sound();
    a2->sound();
    delete a1;
    delete a2;
    return 0;
}
```

Create a program that uses exception handling and function templates to implement a robust mathematical library.

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

template <typename T>
```

```
T divide(T a, T b) {
    if (b == 0)
        throw runtime_error("Division by zero");
    return a / b;
}

int main() {
    try {
        cout << divide(10, 2) << endl;
        cout << divide(10, 0) << endl;
    } catch (exception& e) {
        cout << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

Write a program that uses polymorphism to create a flexible and extensible GUI framework.

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

// Base GUI component
class GUIComponent {
public:
    virtual void render() = 0;
    virtual ~GUIComponent() {}
```

```
};

class Button : public GUIComponent {
public:
    void render() override {
        cout << "Rendering Button\n";
    }
};

class TextBox : public GUIComponent {
public:
    void render() override {
        cout << "Rendering TextBox\n";
    }
};

// Application managing components
class GUIApplication {
    vector<GUIComponent*> components;

public:
    void addComponent(GUIComponent* comp) {
        components.push_back(comp);
    }
```

```
    void renderAll() {
        for (auto comp : components)
            comp->render();
    }

    ~GUIApplication() {
        for (auto comp : components)
            delete comp;
    }
};

int main() {
    GUIApplication app;
    app.addComponent(new Button());
    app.addComponent(new TextBox());
    app.renderAll();
    return 0;
}
```

Implement a program that demonstrates the use of virtual functions and templates to create a generic and reusable algorithm library.

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

class SortStrategy {
public:
```

```
    virtual void sort(vector<int>& data) = 0;
    virtual ~SortStrategy() {}
};

class BubbleSort : public SortStrategy {
public:
    void sort(vector<int>& data) override {
        int n = data.size();
        for (int i = 0; i < n - 1; i++)
            for (int j = 0; j < n - i - 1; j++)
                if (data[j] > data[j + 1])
                    swap(data[j], data[j + 1]);
    }
};

template <typename T>
void print(const vector<T>& data) {
    for (auto& val : data) cout << val << " ";
    cout << endl;
}

int main() {
    vector<int> nums = {5, 3, 1, 4, 2};
    SortStrategy* strategy = new BubbleSort();
    strategy->sort(nums);
```

```
    print(nums);
    delete strategy;
    return 0;
}
```

Create a program that uses polymorphism, templates, and exception handling to implement a comprehensive and type-safe collection framework.

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

template <typename T>
class Collection {
    vector<T> data;

public:
    void add(const T& item) {
        data.push_back(item);
    }

    T get(int index) {
        if (index < 0 || index >= data.size())
            throw out_of_range("Index out of range");
        return data[index];
    }
```

```
    virtual void display() {
        for (auto& item : data)
            cout << item << " ";
        cout << endl;
    }

    virtual ~Collection() {}
};

template <typename T>
class NumberCollection : public Collection<T> {
public:
    void display() override {
        cout << "Numbers: ";
        Collection<T>::display();
    }
};

int main() {
    try {
        NumberCollection<int> coll;
        coll.add(10);
        coll.add(20);
        coll.display();
        cout << "Item at 1: " << coll.get(1) << endl;
```

```cpp
        cout << "Item at 5: " << coll.get(5) << endl;  // will throw
    } catch (exception& e) {
        cout << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

Implement a base class Shape with derived classes Circle, Rectangle, and Triangle. Use virtual functions to calculate the area of each shape.

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

class Shape {
public:
    virtual float area() = 0; // Pure virtual
};

class Circle : public Shape {
    float radius;
public:
    Circle(float r) : radius(r) {}
    float area() override {
        return 3.14 * radius * radius;
    }
};
```

```
class Rectangle : public Shape {
    float length, width;
public:
    Rectangle(float l, float w) : length(l), width(w) {}
    float area() override {
        return length * width;
    }
};

class Triangle : public Shape {
    float base, height;
public:
    Triangle(float b, float h) : base(b), height(h) {}
    float area() override {
        return 0.5 * base * height;
    }
};

int main() {
    Shape* s;
    Circle c(5);
    Rectangle r(4, 6);
    Triangle t(3, 7);
```

```cpp
    s = &c; cout << "Circle Area: " << s->area() << endl;
    s = &r; cout << "Rectangle Area: " << s->area() << endl;
    s = &t; cout << "Triangle Area: " << s->area() << endl;

    return 0;
}
```

Create a base class Animal with a virtual function speak(). Implement derived classes Dog, Cat, and Bird, each overriding the speak() function.

```cpp
class Animal {
public:
    virtual void speak() { cout << "Animal speaks\n"; }
};

class Dog : public Animal {
public:
    void speak() override { cout << "Dog barks\n"; }
};

class Cat : public Animal {
public:
    void speak() override { cout << "Cat meows\n"; }
};

class Bird : public Animal {
public:
    void speak() override { cout << "Bird chirps\n"; }
```

```
};

int main() {
    Animal* a;
    Dog d; Cat c; Bird b;

    a = &d; a->speak();
    a = &c; a->speak();
    a = &b; a->speak();

    return 0;
}
```

Write a program that demonstrates function overriding using a base class Employee and derived classes Manager and Worker.

```
class Employee {
public:
    virtual void work() { cout << "Employee working...\n"; }
};

class Manager : public Employee {
public:
    void work() override { cout << "Manager planning...\n"; }
};

class Worker : public Employee {
public:
```

```
    void work() override { cout << "Worker executing...\n"; }
};

int main() {
    Employee* e;
    Manager m;
    Worker w;

    e = &m; e->work();
    e = &w; e->work();

    return 0;
}
```

Write a program to demonstrate pointer arithmetic by creating an array and accessing its elements using pointers.

```
int main() {
    int arr[] = {10, 20, 30, 40};
    int* ptr = arr;

    for (int i = 0; i < 4; i++) {
        cout << "Value at arr[" << i << "] = " << *(ptr + i) << endl;
    }

    return 0;
}
```

Implement a program that dynamically allocates memory for an integer array and initializes it using pointers.

```
int main() {
    int n;
    cout << "Enter size: ";
    cin >> n;

    int* arr = new int[n];
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

    cout << "Array Elements: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    delete[] arr;
    return 0;
}
```

Create a program that uses a pointer to swap the values of two variables.

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

void swap(int* a, int* b) {
    int temp = *a;
```

```
    *a = *b;

    *b = temp;

}


int main() {

    int x = 10, y = 20;

    swap(&x, &y);

    cout << "x = " << x << ", y = " << y << endl;

    return 0;

}
```

Write a program that creates a dynamic object of a class Student and accesses its members using pointers.

```
#include <iostream>

using namespace std;


class Student {

    string name;

    int age;

public:

    void setData(string n, int a) {

        name = n; age = a;

    }

    void display() {

        cout << "Name: " << name << ", Age: " << age << endl;

    }

};
```

```
int main() {
    Student* s = new Student;
    s->setData("Sakshi", 18);
    s->display();
    delete s;
    return 0;
}
```

Implement a program that uses a pointer to an array of objects to store and display details of multiple Book objects.

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

class Book {
    string title;
    float price;
public:
    void set(string t, float p) {
        title = t;
        price = p;
    }
    void display() {
        cout << "Title: " << title << ", Price: " << price << endl;
    }
};
```

```
int main() {
    int n = 2;
    Book* books = new Book[n];

    books[0].set("C++ Basics", 299.99);
    books[1].set("OOP Concepts", 399.99);

    for (int i = 0; i < n; i++) {
        books[i].display();
    }

    delete[] books;
    return 0;
}
```

Create a program that demonstrates the use of a pointer to an object in a class member function.

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

class Demo {
    int value;
public:
    void set(int val) {
        value = val;
    }
    void show(Demo* ptr) {
```

```
        cout << "Value from pointer: " << ptr->value << endl;
    }
};

int main() {
    Demo d1;
    d1.set(50);
    d1.show(&d1);
    return 0;
}
```

Write a class Box with a member function that returns the current object using the this pointer.

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

class Box {
    int length;
public:
    Box(int l) {
        this->length = l;
    }
    Box& getObject() {
        return *this;
    }
    void display() {
        cout << "Length: " << length << endl;
```

```
    }
};

int main() {
    Box b1(15);
    Box& b2 = b1.getObject();
    b2.display();
    return 0;
}
```

Implement a program that uses the this pointer to chain member function calls in a class Person.

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

class Person {
    string name;
    int age;
public:
    Person& setName(string n) {
        name = n;
        return *this;
    }
    Person& setAge(int a) {
        age = a;
        return *this;
```

```
    }
    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Person p;
    p.setName("Sakshi").setAge(20).display();
    return 0;
}
```

Create a class Counter with a member function that compares two objects using the this pointer.

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

class Counter {
    int count;
public:
    Counter(int c) : count(c) {}
    void compare(Counter& other) {
        if (this->count > other.count)
            cout << "Current object has a higher count." << endl;
        else if (this->count < other.count)
            cout << "Other object has a higher count." << endl;
        else
```

```
        cout << "Both objects have equal count." << endl;
    }
};

int main() {
    Counter c1(10), c2(20);
    c1.compare(c2);
    return 0;
}
```

Write a program that uses pure virtual functions to create an abstract class Vehicle with derived classes Car and Bike.

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

class Vehicle {
public:
    virtual void start() = 0; // Pure virtual function
};

class Car : public Vehicle {
public:
    void start() override {
        cout << "Car started." << endl;
    }
};
```

```
class Bike : public Vehicle {
public:
    void start() override {
        cout << "Bike started." << endl;
    }
};

int main() {
    Vehicle* v;
    Car c;
    Bike b;

    v = &c;
    v->start();

    v = &b;
    v->start();

    return 0;
}
```

Implement a program that demonstrates runtime polymorphism using a virtual function in a base class Shape and derived classes Circle and Square.

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

class Shape {
```

```
public:
    virtual void draw() {
        cout << "Drawing Shape." << endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing Circle." << endl;
    }
};

class Square : public Shape {
public:
    void draw() override {
        cout << "Drawing Square." << endl;
    }
};

int main() {
    Shape* s;
    Circle c;
    Square sq;
```

```
    s = &c;
    s->draw();

    s = &sq;
    s->draw();

    return 0;
}
```

Create a class Account with a pure virtual function calculateInterest(). Implement derived classes SavingsAccount and CurrentAccount.

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

class Account {
public:
    virtual void calculateInterest() = 0; // Pure virtual function
};

class SavingsAccount : public Account {
public:
    void calculateInterest() override {
        cout << "Calculating interest for Savings Account." << endl;
    }
};

class CurrentAccount : public Account {
```

```
public:
    void calculateInterest() override {
        cout << "Calculating interest for Current Account." << endl;
    }
};

int main() {
    Account* acc;
    SavingsAccount sa;
    CurrentAccount ca;

    acc = &sa;
    acc->calculateInterest();

    acc = &ca;
    acc->calculateInterest();

    return 0;
}
```

Write a program that demonstrates polymorphism using a base class Media and derived classes Book and DVD.

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

class Media {
public:
```

```
    virtual void display() {
        cout << "Displaying Media." << endl;
    }
};

class Book : public Media {
public:
    void display() override {
        cout << "Displaying Book." << endl;
    }
};

class DVD : public Media {
public:
    void display() override {
        cout << "Displaying DVD." << endl;
    }
};

int main() {
    Media* m;
    Book b;
    DVD d;

    m = &b;
```

```cpp
    m->display();

    m = &d;
    m->display();

    return 0;
}
```

Implement a class hierarchy with a base class Appliance and derived classes WashingMachine, Refrigerator, and Microwave. Use virtual functions to display the functionality of each appliance.

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

class Appliance {
public:
    virtual void functionality() {
        cout << "General Appliance Functionality." << endl;
    }
};

class WashingMachine : public Appliance {
public:
    void functionality() override {
        cout << "Washing clothes." << endl;
    }
};
```

```
class Refrigerator : public Appliance {
public:
    void functionality() override {
        cout << "Cooling food items." << endl;
    }
};

class Microwave : public Appliance {
public:
    void functionality() override {
        cout << "Heating food." << endl;
    }
};

int main() {
    Appliance* a;
    WashingMachine wm;
    Refrigerator rf;
    Microwave mw;

    a = &wm;
    a->functionality();

    a = &rf;
    a->functionality();
```

```
    a = &mw;
    a->functionality();

    return 0;
}
```

Create a program that uses polymorphism to calculate the area of different geometric shapes using a base class Shape and derived classes Circle and Rectangle.

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

class Shape {
public:
    virtual void area() = 0; // Pure virtual function
};

class Circle : public Shape {
    float radius;
public:
    Circle(float r) : radius(r) {}
    void area() override {
        cout << "Area of Circle: " << 3.14 * radius * radius << endl;
    }
};

class Rectangle : public Shape {
```

```
    float length, breadth;
public:
    Rectangle(float l, float b) : length(l), breadth(b) {}
    void area() override {
        cout << "Area of Rectangle: " << length * breadth << endl;
    }
};

int main() {
    Shape* s;
    Circle c(5);
    Rectangle r(4, 6);

    s = &c;
    s->area();

    s = &r;
    s->area();

    return 0;
}
```

Write an abstract class Employee with pure virtual functions calculateSalary() and displayDetails(). Implement derived classes Manager and Engineer.

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

```
class Employee {
public:
    virtual void calculateSalary() = 0;
    virtual void displayDetails() = 0;
};

class Manager : public Employee {
public:
    void calculateSalary() override {
        cout << "Calculating salary for Manager." << endl;
    }
    void displayDetails() override {
        cout << "Manager Details." << endl;
    }
};

class Engineer : public Employee {
public:
    void calculateSalary() override {
        cout << "Calculating salary for Engineer." << endl;
    }
    void displayDetails() override {
        cout << "Engineer Details." << endl;
    }
};
```

```cpp
int main() {
    Employee* e;
    Manager m;
    Engineer eng;

    e = &m;
    e->calculateSalary();
    e->displayDetails();

    e = &eng;
    e->calculateSalary();
    e->displayDetails();

    return 0;
}
```

Implement an abstract class Payment with a pure virtual function processPayment(). Create derived classes CreditCardPayment and DebitCardPayment.

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

class Payment {
public:
    virtual void processPayment() = 0;
};
```

```
class CreditCardPayment : public Payment {
public:
    void processPayment() override {
        cout << "Processing credit card payment." << endl;
    }
};

class DebitCardPayment : public Payment {
public:
    void processPayment() override {
        cout << "Processing debit card payment." << endl;
    }
};

int main() {
    Payment* p;
    CreditCardPayment ccp;
    DebitCardPayment dcp;

    p = &ccp;
    p->processPayment();

    p = &dcp;
    p->processPayment();
```

```
    return 0;
}
```

Create an abstract class Device with a pure virtual function turnOn(). Implement derived classes Laptop and Smartphone.

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


class Device {
public:
    virtual void turnOn() = 0; // Pure virtual function
    virtual ~Device() {}
};


class Laptop : public Device {
public:
    void turnOn() override {
        cout << "Laptop is turning on." << endl;
    }
};


class Smartphone : public Device {
public:
    void turnOn() override {
        cout << "Smartphone is turning on." << endl;
    }
};
```

```
int main() {
    Device* d1 = new Laptop();
    Device* d2 = new Smartphone();
    d1->turnOn();
    d2->turnOn();
    delete d1;
    delete d2;
    return 0;
}
```

Write a program that handles division by zero using exception handling.

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

int main() {
    int numerator = 10, denominator = 0;
    try {
        if (denominator == 0)
            throw runtime_error("Division by zero error");
        cout << "Result: " << numerator / denominator << endl;
    } catch (const exception& e) {
        cerr << "Exception: " << e.what() << endl;
    }
    return 0;
```

```
}
```

Implement a program that demonstrates the use of multiple catch blocks to handle different types of exceptions.

```
#include <iostream>

#include <stdexcept>

using namespace std;


int main() {

    try {

        throw runtime_error("Runtime error occurred");

    } catch (const logic_error& e) {

        cerr << "Logic error: " << e.what() << endl;

    } catch (const runtime_error& e) {

        cerr << "Runtime error: " << e.what() << endl;

    } catch (...) {

        cerr << "Unknown exception caught" << endl;

    }

    return 0;

}
```

Create a custom exception class InvalidAgeException and use it to handle invalid age input in a program.

```
#include <iostream>

#include <stdexcept>

using namespace std;


class InvalidAgeException : public exception {
```

```
public:
    const char* what() const noexcept override {
        return "Invalid age entered";
    }
};

int main() {
    int age;
    cout << "Enter age: ";
    cin >> age;
    try {
        if (age < 0 || age > 150)
            throw InvalidAgeException();
        cout << "Valid age: " << age << endl;
    } catch (const InvalidAgeException& e) {
        cerr << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

Write a program that uses exception handling to manage file input/output errors.

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

```
int main() {
    ifstream file("nonexistent.txt");
    try {
        if (!file)
            throw runtime_error("File could not be opened");
        // File processing logic here
    } catch (const exception& e) {
        cerr << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

Implement a program that demonstrates the use of the finally block to release resources in exception handling.

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


int main() {
    try {
        ifstream file("data.txt");
        if (!file)
            throw runtime_error("Failed to open file");
        // File processing logic here
    } catch (const exception& e) {
        cerr << "Exception: " << e.what() << endl;
    }
```

```
    // File is automatically closed when it goes out of scope
    return 0;
}
```

Write a function template to perform a linear search on an array of any data type.

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

template <typename T>
int linearSearch(T arr[], int size, T key) {
    for (int i = 0; i < size; ++i)
        if (arr[i] == key)
            return i;
    return -1;
}

int main() {
    int arr[] = {1, 3, 5, 7, 9};
    int index = linearSearch(arr, 5, 7);
    if (index != -1)
        cout << "Element found at index " << index << endl;
    else
        cout << "Element not found" << endl;
    return 0;
}
```

Implement a class template Stack with member functions to push, pop, and display elements.

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

template <typename T>
class Stack {
    T arr[100];
    int top;
public:
    Stack() : top(-1) {}
    void push(T val) {
        if (top < 99)
            arr[++top] = val;
        else
            cout << "Stack overflow" << endl;
    }
    void pop() {
        if (top >= 0)
            --top;
        else
            cout << "Stack underflow" << endl;
    }
    void display() {
        for (int i = top; i >= 0; --i)
            cout << arr[i] << " ";
        cout << endl;
```

```
    }
};

int main() {
    Stack<int> s;
    s.push(10);
    s.push(20);
    s.display();
    s.pop();
    s.display();
    return 0;
}
```

Create a function template to find the maximum of two values of any data type.

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

template <typename T>
T maxVal(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << "Max: " << maxVal(10, 20) << endl;
    cout << "Max: " << maxVal(3.5, 2.5) << endl;
    return 0;
```

```
}
```

Write a class template LinkedList with member functions to insert, delete, and display nodes.

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

template <typename T>
class LinkedList {
    struct Node {
        T data;
        Node* next;
    };
    Node* head;
public:
    LinkedList() : head(nullptr) {}
    void insert(T val) {
        Node* newNode = new Node{val, head};
        head = newNode;
    }
    void remove(T val) {
        Node** curr = &head;
        while (*curr) {
            if ((*curr)->data == val) {
                Node* temp = *curr;
                *curr = (*curr)->next;
                delete temp;
```

```
                return;
            }
            curr = &((*curr)->next);
        }
    }
    void display() {
        Node* curr = head;
        while (curr) {
            cout << curr->data << " ";
            curr = curr->next;
        }
        cout << endl;
    }
};

int main() {
    LinkedList<int> list;
    list.insert(10);
    list.insert(20);
    list.display();
    list.remove(10);
    list.display();
    return 0;
}
```

Implement a function template to perform bubble sort on an array of any data type.

```
#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])
                swap(arr[j], arr[j + 1]);
}

int main() {
    int arr[] = {5, 2, 9, 1, 5};
    bubbleSort(arr, 5);
    for (int i = 0; i < 5; ++i)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
}
```

Create a class template Queue with member functions to enqueue, dequeue, and display elements.

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

template <typename T>
class Queue {
```

```
    T arr[100];
    int front, rear;
public:
    Queue() : front(0), rear(0) {}
    void enqueue(T val) {
        if (rear < 100)
            arr[rear++] = val;
        else
            cout << "Queue overflow" << endl;
    }
    void dequeue() {
        if (front < rear)
            ++front;
        else
            cout << "Queue underflow" << endl;
    }
    void display() {
        for (int i = front; i < rear; ++i)
            cout << arr[i] << " ";
        cout << endl;
    }
};

int main() {
    Queue<int> q;
```

```
    q.enqueue(10);

    q.enqueue(20);

    q.display();

    q.dequeue();

    q.display();

    return 0;

}
```

Write a program that uses polymorphism to create a menu-driven application for managing different types of bank accounts.

```
#include <iostream>

using namespace std;


class Account {

public:

    virtual void display() = 0;

    virtual ~Account() {}

};


class Savings : public Account {

public:

    void display() override {

        cout << "Savings Account" << endl;

    }

};


class Current : public Account {
```

```
public:
    void display() override {
        cout << "Current Account" << endl;
    }
};

int main() {
    Account* acc;
    int choice;
    cout << "1. Savings\n2. Current\nEnter choice: ";
    cin >> choice;
    if (choice == 1)
        acc = new Savings();
    else
        acc = new Current();
    acc->display();
    delete acc;
    return 0;
}
```

Implement a program that demonstrates the use of smart pointers for dynamic memory management.

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

class Demo {
```

```cpp
public:
    Demo() { cout << "Constructor\n"; }
    ~Demo() { cout << "Destructor\n"; }
    void show() { cout << "Demo function\n"; }
};

int main() {
    unique_ptr<Demo> ptr = make_unique<Demo>();
    ptr->show();
    return 0;
}
```

Create a program that uses exception handling and templates to implement a safe array class.

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

template <typename T>
class SafeArray {
    T arr[100];
public:
    T& operator[](int index) {
        if (index < 0 || index >= 100)
            throw out_of_range("Index out of bounds");
        return arr[index];
    }
```

```
};

int main() {

::contentReference[oaicite:0]{index=0}
```

Write a program that demonstrates the use of virtual inheritance to avoid the diamond problem in multiple inheritance.

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

class A {
public:
    void show() {
        cout << "Class A\n";
    }
};

class B : virtual public A {};
class C : virtual public A {};
class D : public B, public C {};

int main() {
    D obj;
    obj.show();  // Only one copy of A
    return 0;
```

```
}
```

Implement a class Polynomial with member functions to add and multiply polynomials using operator overloading.

```
#include <iostream>

#include <vector>

using namespace std;


class Polynomial {

  vector<int> coeffs;


public:

  Polynomial(vector<int> c) : coeffs(c) {}


  Polynomial operator+(const Polynomial& p) {

    vector<int> result(max(coeffs.size(), p.coeffs.size()), 0);

    for (size_t i = 0; i < coeffs.size(); ++i)

      result[i] += coeffs[i];

    for (size_t i = 0; i < p.coeffs.size(); ++i)

      result[i] += p.coeffs[i];

    return Polynomial(result);

  }


  void display() {

    for (int i = coeffs.size() - 1; i >= 0; --i)

      cout << coeffs[i] << "x^" << i << (i ? " + " : "\n");

  }
```

```
};

int main() {
    Polynomial p1({3, 2, 1}); // 1x^2 + 2x^1 + 3x^0
    Polynomial p2({1, 4});    // 4x^1 + 1x^0
    Polynomial sum = p1 + p2;
    sum.display();
    return 0;
}
```

Create a program that uses function pointers to implement a callback mechanism.

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

void greet() {
    cout << "Hello, this is a callback function!\n";
}

void callFunction(void (*fptr)()) {
    fptr();
}

int main() {
    callFunction(greet);
    return 0;
}
```

Write a program that uses class templates and exception handling to implement a generic and robust data structure.

```
#include <iostream>

#include <stdexcept>

using namespace std;


template <typename T>

class SafeArray {

   T* arr;

   int size;


public:

   SafeArray(int n) : size(n) {

      arr = new T[size];

   }


   T& operator[](int index) {

      if (index < 0 || index >= size)

         throw out_of_range("Index out of range");

      return arr[index];

   }


   ~SafeArray() {

      delete[] arr;

   }

};
```

```cpp
int main() {
    try {
        SafeArray<int> arr(5);
        arr[0] = 10;
        cout << arr[0] << endl;
        cout << arr[10] << endl;  // Out of range
    } catch (exception& e) {
        cout << "Error: " << e.what() << endl;
    }
    return 0;
}
```

Implement a program that demonstrates the use of virtual destructors in a class hierarchy.

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

class Base {
public:
    virtual ~Base() {
        cout << "Base destructor\n";
    }
};

class Derived : public Base {
public:
```

```
    ~Derived() {
        cout << "Derived destructor\n";
    }
};

int main() {
    Base* ptr = new Derived();
    delete ptr;
    return 0;
}
```

Create a program that uses a function template to perform generic matrix operations (addition, multiplication).

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

template <typename T>
void addMatrices(T a[2][2], T b[2][2], T res[2][2]) {
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
            res[i][j] = a[i][j] + b[i][j];
}

int main() {
    int A[2][2] = {{1, 2}, {3, 4}}, B[2][2] = {{5, 6}, {7, 8}}, C[2][2];
    addMatrices(A, B, C);
    for (auto& row : C) {
```

```
        for (int val : row) cout << val << " ";
        cout << endl;
    }
    return 0;
}
```

Write a program that uses polymorphism to create a plugin system for a software application.

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

class Plugin {
public:
    virtual void execute() = 0;
};

class PluginA : public Plugin {
public:
    void execute() override {
        cout << "PluginA executed.\n";
    }
};

int main() {
    Plugin* p = new PluginA();
    p->execute();
    delete p;
```

```
    return 0;
}
```

Implement a program that uses class templates to create a generic binary tree data structure.

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

template <typename T>
class Node {
public:
    T data;
    Node* left;
    Node* right;

    Node(T d) : data(d), left(NULL), right(NULL) {}
};

template <typename T>
class BinaryTree {
    Node<T>* root;

    void inorder(Node<T>* node) {
        if (node) {
            inorder(node->left);
            cout << node->data << " ";
            inorder(node->right);
```

```
        }
    }

public:
    BinaryTree() : root(NULL) {}

    void insert(T data) {
        root = new Node<T>(data);  // Simplified
    }

    void display() {
        inorder(root);
    }
};

int main() {
    BinaryTree<int> tree;
    tree.insert(10);
    tree.display();
    return 0;
}
```

Create a program that demonstrates the use of polymorphism to implement a dynamic dispatch mechanism.

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

```cpp
class UIComponent {
public:
    virtual void draw() = 0;
};

class Button : public UIComponent {
public:
    void draw() override {
        cout << "Drawing Button\n";
    }
};

int main() {
    UIComponent* comp = new Button();
    comp->draw();
    delete comp;
    return 0;
}
```

Write a program that uses smart pointers and templates to implement a memory-efficient and type-safe container.

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

template <typename T>
class Container {
```

```
    unique_ptr<T[]> data;
    int size;

public:
    Container(int s) : size(s), data(new T[s]) {}

    T& operator[](int i) { return data[i]; }

    void display() {
        for (int i = 0; i < size; ++i)
            cout << data[i] << " ";
        cout << endl;
    }
};

int main() {
    Container<int> c(3);
    c[0] = 1; c[1] = 2; c[2] = 3;
    c.display();
    return 0;
}
```

Implement a program that uses virtual functions and inheritance to create a simulation of an ecosystem with different types of animals.

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

```
class Animal {
public:
    virtual void sound() = 0;
};

class Lion : public Animal {
public:
    void sound() override {
        cout << "Roar\n";
    }
};

class Bird : public Animal {
public:
    void sound() override {
        cout << "Chirp\n";
    }
};

int main() {
    Animal* a1 = new Lion();
    Animal* a2 = new Bird();
    a1->sound();
    a2->sound();
    delete a1;
```

```
    delete a2;
    return 0;
}
```

Create a program that uses exception handling and function templates to implement a robust mathematical library.

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


template <typename T>
T divide(T a, T b) {
    if (b == 0)
        throw runtime_error("Division by zero");
    return a / b;
}


int main() {
    try {
        cout << divide(10, 2) << endl;
        cout << divide(10, 0) << endl;
    } catch (exception& e) {
        cout << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

Write a program that uses polymorphism to create a flexible and extensible GUI framework.

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

// Base GUI component
class GUIComponent {
public:
    virtual void render() = 0;
    virtual ~GUIComponent() {}
};

class Button : public GUIComponent {
public:
    void render() override {
        cout << "Rendering Button\n";
    }
};

class TextBox : public GUIComponent {
public:
    void render() override {
        cout << "Rendering TextBox\n";
    }
};
```

```
;

// Application managing components
class GUIApplication {
    vector<GUIComponent*> components;

public:
    void addComponent(GUIComponent* comp) {
        components.push_back(comp);
    }

    void renderAll() {
        for (auto comp : components)
            comp->render();
    }

    ~GUIApplication() {
        for (auto comp : components)
            delete comp;
    }
};

int main() {
    GUIApplication app;
    app.addComponent(new Button());
    app.addComponent(new TextBox());
    app.renderAll();
```

```
    return 0;
}
```

Implement a program that demonstrates the use of virtual functions and templates to create a generic and reusable algorithm library.

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

class SortStrategy {
public:
    virtual void sort(vector<int>& data) = 0;
    virtual ~SortStrategy() {}
};

class BubbleSort : public SortStrategy {
public:
    void sort(vector<int>& data) override {
        int n = data.size();
        for (int i = 0; i < n - 1; i++)
            for (int j = 0; j < n - i - 1; j++)
                if (data[j] > data[j + 1])
                    swap(data[j], data[j + 1]);
    }
};

template <typename T>
```

```cpp
void print(const vector<T>& data) {
    for (auto& val : data) cout << val << " ";
    cout << endl;
}

int main() {
    vector<int> nums = {5, 3, 1, 4, 2};
    SortStrategy* strategy = new BubbleSort();
    strategy->sort(nums);
    print(nums);
    delete strategy;
    return 0;
}
```

Create a program that uses polymorphism, templates, and exception handling to implement a comprehensive and type-safe collection framework.

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

template <typename T>
class Collection {
    vector<T> data;

public:
    void add(const T& item) {
```

```
        data.push_back(item);
    }

    T get(int index) {
        if (index < 0 || index >= data.size())
            throw out_of_range("Index out of range");
        return data[index];
    }

    virtual void display() {
        for (auto& item : data)
            cout << item << " ";
        cout << endl;
    }

    virtual ~Collection() {}
};

template <typename T>
class NumberCollection : public Collection<T> {
public:
    void display() override {
        cout << "Numbers: ";
        Collection<T>::display();
    }
```

```
};

int main() {
    try {
        NumberCollection<int> coll;
        coll.add(10);
        coll.add(20);
        coll.display();
        cout << "Item at 1: " << coll.get(1) << endl;
        cout << "Item at 5: " << coll.get(5) << endl;  // will throw
    } catch (exception& e) {
        cout << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

Implement a base class Shape with derived classes Circle, Rectangle, and Triangle. Use virtual functions to calculate the area of each shape.

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

class Shape {
public:
    virtual float area() = 0; // Pure virtual
};
```

```
class Circle : public Shape {
    float radius;
public:
    Circle(float r) : radius(r) {}
    float area() override {
        return 3.14 * radius * radius;
    }
};

class Rectangle : public Shape {
    float length, width;
public:
    Rectangle(float l, float w) : length(l), width(w) {}
    float area() override {
        return length * width;
    }
};

class Triangle : public Shape {
    float base, height;
public:
    Triangle(float b, float h) : base(b), height(h) {}
    float area() override {
        return 0.5 * base * height;
```

```cpp
    }
};

int main() {
    Shape* s;
    Circle c(5);
    Rectangle r(4, 6);
    Triangle t(3, 7);

    s = &c; cout << "Circle Area: " << s->area() << endl;
    s = &r; cout << "Rectangle Area: " << s->area() << endl;
    s = &t; cout << "Triangle Area: " << s->area() << endl;

    return 0;
}
```

Create a base class Animal with a virtual function speak(). Implement derived classes Dog, Cat, and Bird, each overriding the speak() function.

```cpp
class Animal {
public:
    virtual void speak() { cout << "Animal speaks\n"; }
};

class Dog : public Animal {
public:
    void speak() override { cout << "Dog barks\n"; }
};
```

```
class Cat : public Animal {
public:
    void speak() override { cout << "Cat meows\n"; }
};

class Bird : public Animal {
public:
    void speak() override { cout << "Bird chirps\n"; }
};

int main() {
    Animal* a;
    Dog d; Cat c; Bird b;

    a = &d; a->speak();
    a = &c; a->speak();
    a = &b; a->speak();

    return 0;
}
```

Write a program that demonstrates function overriding using a base class Employee and derived classes Manager and Worker.

```
class Employee {
public:
    virtual void work() { cout << "Employee working...\n"; }
```

```
};

class Manager : public Employee {
public:
    void work() override { cout << "Manager planning...\n"; }
};

class Worker : public Employee {
public:
    void work() override { cout << "Worker executing...\n"; }
};

int main() {
    Employee* e;
    Manager m;
    Worker w;

    e = &m; e->work();
    e = &w; e->work();

    return 0;
}
```

Write a program to demonstrate pointer arithmetic by creating an array and accessing its elements using pointers.

```
int main() {
    int arr[] = {10, 20, 30, 40};
```

```cpp
    int* ptr = arr;

    for (int i = 0; i < 4; i++) {
        cout << "Value at arr[" << i << "] = " << *(ptr + i) << endl;
    }

    return 0;
}
```

Implement a program that dynamically allocates memory for an integer array and initializes it using pointers.

```cpp
int main() {
    int n;
    cout << "Enter size: ";
    cin >> n;

    int* arr = new int[n];
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }

    cout << "Array Elements: ";
    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    delete[] arr;
```

```
    return 0;
}
```

Create a program that uses a pointer to swap the values of two variables.

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

void swap(int* a, int* b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 10, y = 20;
    swap(&x, &y);
    cout << "x = " << x << ", y = " << y << endl;
    return 0;
}
```

Write a program that creates a dynamic object of a class Student and accesses its members using pointers.

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

class Student {
    string name;
    int age;
```

```
public:
    void setData(string n, int a) {
        name = n; age = a;
    }
    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Student* s = new Student;
    s->setData("Sakshi", 18);
    s->display();
    delete s;
    return 0;
}
```

Implement a program that uses a pointer to an array of objects to store and display details of multiple Book objects.

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

class Book {
    string title;
    float price;
public:
    void set(string t, float p) {
```

```
        title = t;
        price = p;
    }
    void display() {
        cout << "Title: " << title << ", Price: " << price << endl;
    }
};

int main() {
    int n = 2;
    Book* books = new Book[n];

    books[0].set("C++ Basics", 299.99);
    books[1].set("OOP Concepts", 399.99);

    for (int i = 0; i < n; i++) {
        books[i].display();
    }

    delete[] books;
    return 0;
}
```

Create a program that demonstrates the use of a pointer to an object in a class member function.

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

```
class Demo {
    int value;
public:
    void set(int val) {
        value = val;
    }
    void show(Demo* ptr) {
        cout << "Value from pointer: " << ptr->value << endl;
    }
};

int main() {
    Demo d1;
    d1.set(50);
    d1.show(&d1);
    return 0;
}
```

Write a class Box with a member function that returns the current object using the this pointer.

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

class Box {
    int length;
public:
```

```
    Box(int l) {
        this->length = l;
    }
    Box& getObject() {
        return *this;
    }
    void display() {
        cout << "Length: " << length << endl;
    }
};

int main() {
    Box b1(15);
    Box& b2 = b1.getObject();
    b2.display();
    return 0;
}
```

Implement a program that uses the this pointer to chain member function calls in a class Person.

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

class Person {
    string name;
    int age;
```

```
public:
    Person& setName(string n) {
        name = n;
        return *this;
    }
    Person& setAge(int a) {
        age = a;
        return *this;
    }
    void display() {
        cout << "Name: " << name << ", Age: " << age << endl;
    }
};

int main() {
    Person p;
    p.setName("Sakshi").setAge(20).display();
    return 0;
}
```

Create a class Counter with a member function that compares two objects using the this pointer.

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

class Counter {
    int count;
```

```
public:
    Counter(int c) : count(c) {}
    void compare(Counter& other) {
        if (this->count > other.count)
            cout << "Current object has a higher count." << endl;
        else if (this->count < other.count)
            cout << "Other object has a higher count." << endl;
        else
            cout << "Both objects have equal count." << endl;
    }
};

int main() {
    Counter c1(10), c2(20);
    c1.compare(c2);
    return 0;
}
```

Write a program that uses pure virtual functions to create an abstract class Vehicle with derived classes Car and Bike.

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

class Vehicle {
public:
    virtual void start() = 0; // Pure virtual function
};
```

```cpp
class Car : public Vehicle {
public:
    void start() override {
        cout << "Car started." << endl;
    }
};

class Bike : public Vehicle {
public:
    void start() override {
        cout << "Bike started." << endl;
    }
};

int main() {
    Vehicle* v;
    Car c;
    Bike b;

    v = &c;
    v->start();

    v = &b;
    v->start();
```

```
    return 0;
}
```

Implement a program that demonstrates runtime polymorphism using a virtual function in a base class Shape and derived classes Circle and Square.

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

class Shape {
public:
    virtual void draw() {
        cout << "Drawing Shape." << endl;
    }
};

class Circle : public Shape {
public:
    void draw() override {
        cout << "Drawing Circle." << endl;
    }
};

class Square : public Shape {
public:
    void draw() override {
        cout << "Drawing Square." << endl;
```

```
    }
};

int main() {
    Shape* s;
    Circle c;
    Square sq;

    s = &c;
    s->draw();

    s = &sq;
    s->draw();

    return 0;
}
```

Create a class Account with a pure virtual function calculateInterest(). Implement derived classes SavingsAccount and CurrentAccount.

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

class Account {
public:
    virtual void calculateInterest() = 0; // Pure virtual function
};
```

```
class SavingsAccount : public Account {
public:
    void calculateInterest() override {
        cout << "Calculating interest for Savings Account." << endl;
    }
};

class CurrentAccount : public Account {
public:
    void calculateInterest() override {
        cout << "Calculating interest for Current Account." << endl;
    }
};

int main() {
    Account* acc;
    SavingsAccount sa;
    CurrentAccount ca;

    acc = &sa;
    acc->calculateInterest();

    acc = &ca;
    acc->calculateInterest();
```

```
    return 0;
}
```

Write a program that demonstrates polymorphism using a base class Media and derived classes Book and DVD.

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

class Media {
public:
    virtual void display() {
        cout << "Displaying Media." << endl;
    }
};

class Book : public Media {
public:
    void display() override {
        cout << "Displaying Book." << endl;
    }
};

class DVD : public Media {
public:
    void display() override {
        cout << "Displaying DVD." << endl;
    }
```

```
};

int main() {
    Media* m;
    Book b;
    DVD d;

    m = &b;
    m->display();

    m = &d;
    m->display();

    return 0;
}
```

Implement a class hierarchy with a base class Appliance and derived classes WashingMachine, Refrigerator, and Microwave. Use virtual functions to display the functionality of each appliance.

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

class Appliance {
public:
    virtual void functionality() {
        cout << "General Appliance Functionality." << endl;
    }
};
```

```cpp
class WashingMachine : public Appliance {
public:
    void functionality() override {
        cout << "Washing clothes." << endl;
    }
};

class Refrigerator : public Appliance {
public:
    void functionality() override {
        cout << "Cooling food items." << endl;
    }
};

class Microwave : public Appliance {
public:
    void functionality() override {
        cout << "Heating food." << endl;
    }
};

int main() {
    Appliance* a;
    WashingMachine wm;
```

```
    Refrigerator rf;
    Microwave mw;

    a = &wm;
    a->functionality();

    a = &rf;
    a->functionality();

    a = &mw;
    a->functionality();

    return 0;
}
```

Create a program that uses polymorphism to calculate the area of different geometric shapes using a base class Shape and derived classes Circle and Rectangle.

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

class Shape {
public:
    virtual void area() = 0; // Pure virtual function
};

class Circle : public Shape {
    float radius;
```

```
public:
    Circle(float r) : radius(r) {}
    void area() override {
        cout << "Area of Circle: " << 3.14 * radius * radius << endl;
    }
};

class Rectangle : public Shape {
    float length, breadth;
public:
    Rectangle(float l, float b) : length(l), breadth(b) {}
    void area() override {
        cout << "Area of Rectangle: " << length * breadth << endl;
    }
};

int main() {
    Shape* s;
    Circle c(5);
    Rectangle r(4, 6);

    s = &c;
    s->area();

    s = &r;
```

```
    s->area();

    return 0;
}
```

Write an abstract class Employee with pure virtual functions calculateSalary() and displayDetails(). Implement derived classes Manager and Engineer.

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

class Employee {
public:
    virtual void calculateSalary() = 0;
    virtual void displayDetails() = 0;
};

class Manager : public Employee {
public:
    void calculateSalary() override {
        cout << "Calculating salary for Manager." << endl;
    }
    void displayDetails() override {
        cout << "Manager Details." << endl;
    }
};

class Engineer : public Employee {
```

```
public:
    void calculateSalary() override {
        cout << "Calculating salary for Engineer." << endl;
    }
    void displayDetails() override {
        cout << "Engineer Details." << endl;
    }
};

int main() {
    Employee* e;
    Manager m;
    Engineer eng;

    e = &m;
    e->calculateSalary();
    e->displayDetails();

    e = &eng;
    e->calculateSalary();
    e->displayDetails();

    return 0;
}
```

Implement an abstract class Payment with a pure virtual function processPayment(). Create derived classes CreditCardPayment and DebitCardPayment.

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

class Payment {
public:
    virtual void processPayment() = 0;
};

class CreditCardPayment : public Payment {
public:
    void processPayment() override {
        cout << "Processing credit card payment." << endl;
    }
};

class DebitCardPayment : public Payment {
public:
    void processPayment() override {
        cout << "Processing debit card payment." << endl;
    }
};

int main() {
    Payment* p;
    CreditCardPayment ccp;
```

```
    DebitCardPayment dcp;

    p = &ccp;
    p->processPayment();

    p = &dcp;
    p->processPayment();

    return 0;
}
```

Create an abstract class Device with a pure virtual function turnOn(). Implement derived classes Laptop and Smartphone.

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

class Device {
public:
    virtual void turnOn() = 0; // Pure virtual function
    virtual ~Device() {}
};

class Laptop : public Device {
public:
    void turnOn() override {
        cout << "Laptop is turning on." << endl;
    }
```

```
};

class Smartphone : public Device {
public:
    void turnOn() override {
        cout << "Smartphone is turning on." << endl;
    }
};

int main() {
    Device* d1 = new Laptop();
    Device* d2 = new Smartphone();
    d1->turnOn();
    d2->turnOn();
    delete d1;
    delete d2;
    return 0;
}
```

Write a program that handles division by zero using exception handling.

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

int main() {
    int numerator = 10, denominator = 0;
```

```
    try {
        if (denominator == 0)
            throw runtime_error("Division by zero error");
        cout << "Result: " << numerator / denominator << endl;
    } catch (const exception& e) {
        cerr << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

Implement a program that demonstrates the use of multiple catch blocks to handle different types of exceptions.

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

int main() {
    try {
        throw runtime_error("Runtime error occurred");
    } catch (const logic_error& e) {
        cerr << "Logic error: " << e.what() << endl;
    } catch (const runtime_error& e) {
        cerr << "Runtime error: " << e.what() << endl;
    } catch (...) {
        cerr << "Unknown exception caught" << endl;
    }
    return 0;
```

```
}
```

Create a custom exception class InvalidAgeException and use it to handle invalid age input in a program.

```
#include <iostream>

#include <stdexcept>

using namespace std;


class InvalidAgeException : public exception {

public:

    const char* what() const noexcept override {

        return "Invalid age entered";

    }

};


int main() {

    int age;

    cout << "Enter age: ";

    cin >> age;

    try {

        if (age < 0 || age > 150)

            throw InvalidAgeException();

        cout << "Valid age: " << age << endl;

    } catch (const InvalidAgeException& e) {

        cerr << "Exception: " << e.what() << endl;

    }

    return 0;
```

```
}
```

Write a program that uses exception handling to manage file input/output errors.

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

int main() {
    ifstream file("nonexistent.txt");
    try {
        if (!file)
            throw runtime_error("File could not be opened");
        // File processing logic here
    } catch (const exception& e) {
        cerr << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

Implement a program that demonstrates the use of the finally block to release resources in exception handling.

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

int main() {
    try {
```

```
        ifstream file("data.txt");
        if (!file)
            throw runtime_error("Failed to open file");
        // File processing logic here
    } catch (const exception& e) {
        cerr << "Exception: " << e.what() << endl;
    }
    // File is automatically closed when it goes out of scope
    return 0;
}
```

Write a function template to perform a linear search on an array of any data type.

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

template <typename T>
int linearSearch(T arr[], int size, T key) {
    for (int i = 0; i < size; ++i)
        if (arr[i] == key)
            return i;
    return -1;
}

int main() {
    int arr[] = {1, 3, 5, 7, 9};
    int index = linearSearch(arr, 5, 7);
```

```
    if (index != -1)
        cout << "Element found at index " << index << endl;
    else
        cout << "Element not found" << endl;
    return 0;
}
```

Implement a class template Stack with member functions to push, pop, and display elements.

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

template <typename T>
class Stack {
    T arr[100];
    int top;
public:
    Stack() : top(-1) {}
    void push(T val) {
        if (top < 99)
            arr[++top] = val;
        else
            cout << "Stack overflow" << endl;
    }
    void pop() {
        if (top >= 0)
            --top;
```

```
        else
            cout << "Stack underflow" << endl;
    }
    void display() {
        for (int i = top; i >= 0; --i)
            cout << arr[i] << " ";
        cout << endl;
    }
};

int main() {
    Stack<int> s;
    s.push(10);
    s.push(20);
    s.display();
    s.pop();
    s.display();
    return 0;
}
```

Create a function template to find the maximum of two values of any data type.

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

template <typename T>
T maxVal(T a, T b) {
```

```
    return (a > b) ? a : b;
}

int main() {
    cout << "Max: " << maxVal(10, 20) << endl;
    cout << "Max: " << maxVal(3.5, 2.5) << endl;
    return 0;
}
```

Write a class template LinkedList with member functions to insert, delete, and display nodes.

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

template <typename T>
class LinkedList {
    struct Node {
        T data;
        Node* next;
    };
    Node* head;
public:
    LinkedList() : head(nullptr) {}
    void insert(T val) {
        Node* newNode = new Node{val, head};
        head = newNode;
    }
```

```
    void remove(T val) {
        Node** curr = &head;
        while (*curr) {
            if ((*curr)->data == val) {
                Node* temp = *curr;
                *curr = (*curr)->next;
                delete temp;
                return;
            }
            curr = &((*curr)->next);
        }
    }
    void display() {
        Node* curr = head;
        while (curr) {
            cout << curr->data << " ";
            curr = curr->next;
        }
        cout << endl;
    }
};

int main() {
    LinkedList<int> list;
    list.insert(10);
```

```
    list.insert(20);
    list.display();
    list.remove(10);
    list.display();
    return 0;
}
```

Implement a function template to perform bubble sort on an array of any data type.

```
#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])
                swap(arr[j], arr[j + 1]);
}

int main() {
    int arr[] = {5, 2, 9, 1, 5};
    bubbleSort(arr, 5);
    for (int i = 0; i < 5; ++i)
        cout << arr[i] << " ";
    cout << endl;
    return 0;
```

}

Create a class template Queue with member functions to enqueue, dequeue, and display elements.

```
#include <iostream>

using namespace std;


template <typename T>

class Queue {

    T arr[100];

    int front, rear;

public:

    Queue() : front(0), rear(0) {}

    void enqueue(T val) {

        if (rear < 100)

            arr[rear++] = val;

        else

            cout << "Queue overflow" << endl;

    }

    void dequeue() {

        if (front < rear)

            ++front;

        else

            cout << "Queue underflow" << endl;

    }

    void display() {

        for (int i = front; i < rear; ++i)
```

```
            cout << arr[i] << " ";
        cout << endl;
    }
};

int main() {
    Queue<int> q;
    q.enqueue(10);
    q.enqueue(20);
    q.display();
    q.dequeue();
    q.display();
    return 0;
}
```

Write a program that uses polymorphism to create a menu-driven application for managing different types of bank accounts.

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

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

class Savings : public Account {
```

```
public:
    void display() override {
        cout << "Savings Account" << endl;
    }
};

class Current : public Account {
public:
    void display() override {
        cout << "Current Account" << endl;
    }
};

int main() {
    Account* acc;
    int choice;
    cout << "1. Savings\n2. Current\nEnter choice: ";
    cin >> choice;
    if (choice == 1)
        acc = new Savings();
    else
        acc = new Current();
    acc->display();
    delete acc;
    return 0;
```

```
}
```

Implement a program that demonstrates the use of smart pointers for dynamic memory management.

```
#include <iostream>

#include <memory>

using namespace std;


class Demo {

public:

    Demo() { cout << "Constructor\n"; }

    ~Demo() { cout << "Destructor\n"; }

    void show() { cout << "Demo function\n"; }

};


int main() {

    unique_ptr<Demo> ptr = make_unique<Demo>();

    ptr->show();

    return 0;

}
```

Create a program that uses exception handling and templates to implement a safe array class.

```
#include <iostream>

#include <stdexcept>

using namespace std;


template <typename T>

class SafeArray {
```

```
    T arr[100];
public:
    T& operator[](int index) {
        if (index < 0 || index >= 100)
            throw out_of_range("Index out of bounds");
        return arr[index];
    }
};

int main() {

::contentReference[oaicite:0]{index=0}
```

Write a program that demonstrates the use of virtual inheritance to avoid the diamond problem in multiple inheritance.

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

class A {
public:
    void show() {
        cout << "Class A\n";
    }
};

class B : virtual public A {};
```

```
class C : virtual public A {};
class D : public B, public C {};

int main() {
    D obj;
    obj.show();  // Only one copy of A
    return 0;
}
```

Implement a class Polynomial with member functions to add and multiply polynomials using operator overloading.

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

class Polynomial {
    vector<int> coeffs;

public:
    Polynomial(vector<int> c) : coeffs(c) {}

    Polynomial operator+(const Polynomial& p) {
        vector<int> result(max(coeffs.size(), p.coeffs.size()), 0);
        for (size_t i = 0; i < coeffs.size(); ++i)
            result[i] += coeffs[i];
        for (size_t i = 0; i < p.coeffs.size(); ++i)
            result[i] += p.coeffs[i];
```

```
        return Polynomial(result);
    }

    void display() {
        for (int i = coeffs.size() - 1; i >= 0; --i)
            cout << coeffs[i] << "x^" << i << (i ? " + " : "\n");
    }
};

int main() {
    Polynomial p1({3, 2, 1}); // 1x^2 + 2x^1 + 3x^0
    Polynomial p2({1, 4});    // 4x^1 + 1x^0
    Polynomial sum = p1 + p2;
    sum.display();
    return 0;
}
```

Create a program that uses function pointers to implement a callback mechanism.

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

void greet() {
    cout << "Hello, this is a callback function!\n";
}

void callFunction(void (*fptr)()) {
```

```
    fptr();
}

int main() {
    callFunction(greet);
    return 0;
}
```

Write a program that uses class templates and exception handling to implement a generic and robust data structure.

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

template <typename T>
class SafeArray {
    T* arr;
    int size;

public:
    SafeArray(int n) : size(n) {
        arr = new T[size];
    }

    T& operator[](int index) {
        if (index < 0 || index >= size)
            throw out_of_range("Index out of range");
```

```
        return arr[index];
    }

    ~SafeArray() {
        delete[] arr;
    }
};

int main() {
    try {
        SafeArray<int> arr(5);
        arr[0] = 10;
        cout << arr[0] << endl;
        cout << arr[10] << endl;  // Out of range
    } catch (exception& e) {
        cout << "Error: " << e.what() << endl;
    }
    return 0;
}
```

Implement a program that demonstrates the use of virtual destructors in a class hierarchy.

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

class Base {
public:
```

```
    virtual ~Base() {
        cout << "Base destructor\n";
    }
};

class Derived : public Base {
public:
    ~Derived() {
        cout << "Derived destructor\n";
    }
};

int main() {
    Base* ptr = new Derived();
    delete ptr;
    return 0;
}
```

Create a program that uses a function template to perform generic matrix operations (addition, multiplication).

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

template <typename T>
void addMatrices(T a[2][2], T b[2][2], T res[2][2]) {
    for (int i = 0; i < 2; ++i)
        for (int j = 0; j < 2; ++j)
```

```cpp
            res[i][j] = a[i][j] + b[i][j];
}

int main() {
    int A[2][2] = {{1, 2}, {3, 4}}, B[2][2] = {{5, 6}, {7, 8}}, C[2][2];
    addMatrices(A, B, C);
    for (auto& row : C) {
        for (int val : row) cout << val << " ";
        cout << endl;
    }
    return 0;
}
```

Write a program that uses polymorphism to create a plugin system for a software application.

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

class Plugin {
public:
    virtual void execute() = 0;
};

class PluginA : public Plugin {
public:
    void execute() override {
        cout << "PluginA executed.\n";
```

```
    }
};

int main() {
    Plugin* p = new PluginA();
    p->execute();
    delete p;
    return 0;
}
```

Implement a program that uses class templates to create a generic binary tree data structure.

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

template <typename T>
class Node {
public:
    T data;
    Node* left;
    Node* right;

    Node(T d) : data(d), left(NULL), right(NULL) {}
};

template <typename T>
class BinaryTree {
```

```
    Node<T>* root;

    void inorder(Node<T>* node) {
        if (node) {
            inorder(node->left);
            cout << node->data << " ";
            inorder(node->right);
        }
    }

public:
    BinaryTree() : root(NULL) {}

    void insert(T data) {
        root = new Node<T>(data);  // Simplified
    }

    void display() {
        inorder(root);
    }
};

int main() {
    BinaryTree<int> tree;
    tree.insert(10);
```

```
    tree.display();
    return 0;
}
```

Create a program that demonstrates the use of polymorphism to implement a dynamic dispatch mechanism.

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

class UIComponent {
public:
    virtual void draw() = 0;
};

class Button : public UIComponent {
public:
    void draw() override {
        cout << "Drawing Button\n";
    }
};

int main() {
    UIComponent* comp = new Button();
    comp->draw();
    delete comp;
    return 0;
}
```

Write a program that uses smart pointers and templates to implement a memory-efficient and type-safe container.

```
#include <iostream>

#include <memory>

using namespace std;


template <typename T>

class Container {

    unique_ptr<T[]> data;

    int size;


public:

    Container(int s) : size(s), data(new T[s]) {}


    T& operator[](int i) { return data[i]; }


    void display() {

        for (int i = 0; i < size; ++i)

            cout << data[i] << " ";

        cout << endl;

    }

};


int main() {

    Container<int> c(3);

    c[0] = 1; c[1] = 2; c[2] = 3;
```

```
    c.display();
    return 0;
}
```

Implement a program that uses virtual functions and inheritance to create a simulation of an ecosystem with different types of animals.

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

class Animal {
public:
    virtual void sound() = 0;
};

class Lion : public Animal {
public:
    void sound() override {
        cout << "Roar\n";
    }
};

class Bird : public Animal {
public:
    void sound() override {
        cout << "Chirp\n";
    }
};
```

```
int main() {
    Animal* a1 = new Lion();
    Animal* a2 = new Bird();
    a1->sound();
    a2->sound();
    delete a1;
    delete a2;
    return 0;
}
```

Create a program that uses exception handling and function templates to implement a robust mathematical library.

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

template <typename T>
T divide(T a, T b) {
    if (b == 0)
        throw runtime_error("Division by zero");
    return a / b;
}

int main() {
    try {
        cout << divide(10, 2) << endl;
```

```
        cout << divide(10, 0) << endl;
    } catch (exception& e) {
        cout << "Exception: " << e.what() << endl;
    }
    return 0;
}
```

Write a program that uses polymorphism to create a flexible and extensible GUI framework.

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

// Base GUI component
class GUIComponent {
public:
    virtual void render() = 0;
    virtual ~GUIComponent() {}
};

class Button : public GUIComponent {
public:
    void render() override {
        cout << "Rendering Button\n";
    }
};
```

```
class TextBox : public GUIComponent {
public:
    void render() override {
        cout << "Rendering TextBox\n";
    }
};

// Application managing components
class GUIApplication {
    vector<GUIComponent*> components;

public:
    void addComponent(GUIComponent* comp) {
        components.push_back(comp);
    }

    void renderAll() {
        for (auto comp : components)
            comp->render();
    }

    ~GUIApplication() {
        for (auto comp : components)
            delete comp;
    }
```

```
};

int main() {
    GUIApplication app;
    app.addComponent(new Button());
    app.addComponent(new TextBox());
    app.renderAll();
    return 0;
}
```

Implement a program that demonstrates the use of virtual functions and templates to create a generic and reusable algorithm library.

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

class SortStrategy {
public:
    virtual void sort(vector<int>& data) = 0;
    virtual ~SortStrategy() {}
};

class BubbleSort : public SortStrategy {
public:
    void sort(vector<int>& data) override {
        int n = data.size();
        for (int i = 0; i < n - 1; i++)
```

```
        for (int j = 0; j < n - i - 1; j++)
            if (data[j] > data[j + 1])
                swap(data[j], data[j + 1]);
    }
};

template <typename T>
void print(const vector<T>& data) {
    for (auto& val : data) cout << val << " ";
    cout << endl;
}

int main() {
    vector<int> nums = {5, 3, 1, 4, 2};
    SortStrategy* strategy = new BubbleSort();
    strategy->sort(nums);
    print(nums);
    delete strategy;
    return 0;
}
```

Create a program that uses polymorphism, templates, and exception handling to implement a comprehensive and type-safe collection framework.

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

```
template <typename T>
class Collection {
    vector<T> data;

public:
    void add(const T& item) {
        data.push_back(item);
    }

    T get(int index) {
        if (index < 0 || index >= data.size())
            throw out_of_range("Index out of range");
        return data[index];
    }

    virtual void display() {
        for (auto& item : data)
            cout << item << " ";
        cout << endl;
    }

    virtual ~Collection() {}
};
```

```
template <typename T>
class NumberCollection : public Collection<T> {
public:
    void display() override {
        cout << "Numbers: ";
        Collection<T>::display();
    }
};

int main() {
    try {
        NumberCollection<int> coll;
        coll.add(10);
        coll.add(20);
        coll.display();
        cout << "Item at 1: " << coll.get(1) << endl;
        cout << "Item at 5: " << coll.get(5) << endl;  // will throw
    } catch (exception& e) {
        cout << "Exception: " << e.what() << endl;
    }
    return 0;
}
```