

Question : 1

Q1: Write short answers (2-3 lines) for the following questions: [10 minutes, 1*5=5 marks]

a) What is the difference between compile-time error and runtime error? Which types of errors can be dealt with using exceptions?

Answer: Exceptions can be handled during run-time whereas errors cannot be because exceptions occur due to some unexpected conditions during run-time whereas about errors compiler is sure and tells about them during compile-time.

b) Which principle of object-oriented programming suits in the below scenarios?

1. A bank vault with multiple layers of security protecting valuable assets.

Encapsulation

2. A car with different driving modes for different road conditions. Polymorphism

c) Write an advantage of using overloading instead of generics in C++?

Answer: Overloading allows the programmer to write more specific code for each data type.

d) Can a friend function of Base class access the data members of a derived class? No

e) If class C inherits class B and class B inherits class A, then Class C object can be upcasted to object of either class A or B? Yes

Question : 2

Q2: Predict the output of the following code snippets. If there are any errors, provide a proper justification.

Exclude errors related to libraries. [25 minutes, 2*5=10 marks]

A. class Test

```
{ public: Test() { cout <<"Hello from Test() "; }
```

```
} a;
```

```
int main()
```

```
{ cout <<"Main Started "; }
```

```
Hello from Test() Main Started
```

B. class Test {

```
public:
```

```
Test() { cout << "Constructing an object of Test " << endl; }
```

```
~Test() { cout << "Destructing an object of Test " << endl; }
```

```
};
```

```
int main() {
```

```
try {
```

```
Test t1;
```

```
throw 10;
```

```
} catch(int i) { cout << "Caught " << i << endl; } }
```

```
Constructing an object of Test
```

```
Destructing an object of Test
```

```
Caught 10
```

```

C. template <typename T>
void fun(const T&x)
{ static int count = 0;
  cout << "x = " << x << " count = " << count;
  ++count;
  return;
}
int main()
{
  fun<int> (1); cout << endl;
  fun<int>(1); cout << endl;
  fun<double>(1.1); cout << endl;
}
x = 1 count = 0
x = 1 count = 1
x = 1.1 count = 0

```

```

D. int main() {
  try {
    cout<<"Throwing Block"<<endl;
    throw 'x';
  }
  catch (int e) {
    cout <<"Catching Int" << e << endl; }
  catch (...) {
    cout <<"Catching Default" << endl; }
  catch (char e) {
    cout <<"Catching Char" << e << endl; }
  return 0;}
Error because default catch comes in the end

```

```

E. char foo(int i) {
  if (i % 3 != 0)
    return 't';
  throw 'o';
  return 'p';
}
int main() {
  char c = 'c';
  try {
    for (int i = 1; ; i++) { c = foo(i); cout << "c" << endl;}
  }
  catch (char ex) {

```

```
cout << c << endl; cout << ex << endl; } }
```

c
c
t
o

Question 03:

Q3: Complete the C++ code for an Object-Oriented Programming (OOP) scenario. [25 minutes, 5*2=10 marks] A. For the given class, you are required to create a specialized template that manages computations specifically when both arrays are characters with a size of 10. Overload the function so that it returns a string containing all elements of arr1 followed by all elements of arr2.

```
template <class T, int size>
class QuestionTemplate {
    T arr1[size];
    T arr2[size];
public:
    QuestionTemplate() {
        //assume numbers only for now
        for (int i = 0; i < size; i++){
            arr1[i] = i;
            arr2[size - i - 1] = i;
        }
    }
    T* add() {
        T* arr = new T[size];
        for (int i = 0; i < size; i++)
            arr[i] = arr1[i] + arr2[i];
        return arr;
    }
};
```

```
int main() {
    QuestionTemplate <int, 10> qt;
    int* res = qt.add();
    for (int i = 0; i < 10; i++)
        cout << res[i] << endl;
    QuestionTemplate <char, 10> ct;
    cout << ct.add();
}
```

```
//---- start code completion ----
template <>
class QuestionTemplate <char, 10> {
    char arr1[10];
    char arr2[10];
public:
    QuestionTemplate() {
        char c = 'a';
        for (int i = 0; i < 10; i++) {
            arr1[i] = c + i;
            arr2[9 - i] = c + i;
        }
    }
    string add() {
        string str = "";
        for (int i = 0; i < 10; i++)
            str += arr1[i];
        for (int i = 0; i < 10; i++)
            str += arr2[i];

        return str;
    }
};
//---- finish code completion ----
```

B. Complete the given code below in such a way that when we run this code, a similar kind of output is stored in the file.

outfile.txt :

ID = 1, Name = 0001

ID = 2, Name = 0002

```
class Test {
    int ID;
    string name;
    static int genID;
public:
    //---- start code completion ----
    Test() {
        ID = ++genID;
        name = "000" + to_string(genID);
    }
    void operator + (string filename) {
        ofstream fout(filename, ios::app); //ios::app is required, otherwise one line will exist in the
file
        fout << "ID = " << this->ID << ", Name = " << this->name <<endl;

    }
    //---- finish code completion ----

};

int Test::genID = 0;

int main() {
    Test t1, t2;

    t1 + "outfile.txt";
    t2 + "outfile.txt";
}
```

Question 04:

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

```
using namespace std;
```

```
class TelemetryData {
```

```
private:
```

```
    float fuelConsumption; // liters per hour
    float speed; // kilometers per hour
    bool engineStatus; // true if running, false otherwise
```

```
public:
```

```
    TelemetryData(float fuelConsumption = 0.0, float speed = 0.0, bool engineStatus = false)
        : fuelConsumption(fuelConsumption), speed(speed), engineStatus(engineStatus) {}
```

```
    void displayTelemetry() const {
```

```
        cout << "Telemetry Data: \n";
        cout << "Fuel Consumption: " << fuelConsumption << " liters/hour\n";
        cout << "Speed: " << speed << " km/h\n";
        cout << "Engine Status: " << (engineStatus ? "Running" : "Stopped") << "\n";
    }
```

```
};
```

```
class Vehicle {
```

```
protected:
```

```
    char vehicleID[10];
    char model[50];
    char fuelType[10];
    char currentLocation[50];
    TelemetryData telemetry;
```

```
public:
```

```
    Vehicle(const char* vehicleID, const char* model, const char* fuelType, const char* currentLocation,
const TelemetryData& telemetry)
```

```
        : telemetry(telemetry) {
        strncpy(this->vehicleID, vehicleID, 10);
        strncpy(this->model, model, 50);
        strncpy(this->fuelType, fuelType, 10);
        strncpy(this->currentLocation, currentLocation, 50);
    }
```

```
    virtual void displayDetails() const {
```

```

        cout << "Vehicle ID: " << vehicleID << "\n";
        cout << "Model: " << model << "\n";
        cout << "Fuel Type: " << fuelType << "\n";
        cout << "Current Location: " << currentLocation << "\n";
        telemetry.displayTelemetry();
    }

    void updateLocation(const char* newLocation) {
        strncpy(currentLocation, newLocation, 50);
    }

    // Friend class declaration
    friend class VehicleManager;
};

class VehicleManager {
public:
    void trackVehicleLocation(const Vehicle& vehicle) const {
        cout << "Tracking Vehicle ID: " << vehicle.vehicleID << ", Current Location: " <<
        vehicle.currentLocation << "\n";
    }

    void writeToFile(Vehicle* vehiclesArray[], int size) const {
        ofstream outFile("vehicle_info.txt");
        if (!outFile) {
            cout << "Error opening file for writing.\n";
            return;
        }

        for (int i = 0; i < size; ++i) {
            outFile << vehiclesArray[i]->vehicleID << " "
                << vehiclesArray[i]->model << " "
                << vehiclesArray[i]->fuelType << " "
                << vehiclesArray[i]->currentLocation << "\n";
        }

        outFile.close();
    }

    void readFromFile() const {
        ifstream inFile("vehicle_info.txt");
        if (!inFile) {
            cout << "Error opening file for reading.\n";
            return;
        }
    }
};

```

```

    }

    char vehicleID[10];
    char model[50];
    char fuelType[10];
    char location[50];

    while (inFile >> vehicleID >> model >> fuelType >> location) {
        if (strcmp(location, "Karachi") == 0) {
            cout << "Vehicle ID: " << vehicleID << ", Model: " << model << ", Fuel Type: " <<
fuelType << ", Location: " << location << "\n";
        }
    }

    inFile.close();
}
};

class MeterReadingVehicle : public Vehicle {
private:
    int readingCapacity;

public:
    MeterReadingVehicle(const char* vehicleID, const char* model, const char* fuelType, const char*
currentLocation, const TelemetryData& telemetry, int readingCapacity)
        : Vehicle(vehicleID, model, fuelType, currentLocation, telemetry),
readingCapacity(readingCapacity) {}

    void displayDetails() const override {
        Vehicle::displayDetails();
        cout << "Reading Capacity: " << readingCapacity << "\n";
    }
};

class PipelineInspectionVehicle : public Vehicle {
private:
    int inspectionRange;

public:
    PipelineInspectionVehicle(const char* vehicleID, const char* model, const char* fuelType, const
char* currentLocation, const TelemetryData& telemetry, int inspectionRange)
        : Vehicle(vehicleID, model, fuelType, currentLocation, telemetry),
inspectionRange(inspectionRange) {}

```



```

    void displayDetails() const override {
        Vehicle::displayDetails();
        cout << "Inspection Range: " << inspectionRange << " km\n";
    }
};

class MaintenanceVehicle : public Vehicle {
private:
    int equipmentCapacity;

public:
    MaintenanceVehicle(const char* vehicleID, const char* model, const char* fuelType, const char*
currentLocation, const TelemetryData& telemetry, int equipmentCapacity)
        : Vehicle(vehicleID, model, fuelType, currentLocation, telemetry),
equipmentCapacity(equipmentCapacity) {}

    void displayDetails() const override {
        Vehicle::displayDetails();
        cout << "Equipment Capacity: " << equipmentCapacity << " units\n";
    }
};

int main() {
    TelemetryData telemetry(8.5, 60.0, true);
    MeterReadingVehicle meterVehicle("V1234", "Toyota Corolla 2022", "Petrol", "Karachi", telemetry,
100);
    PipelineInspectionVehicle pipelineVehicle("V5678", "Honda Civic 2023", "Diesel", "Lahore",
telemetry, 200);
    MaintenanceVehicle maintenanceVehicle("V9101", "Ford F-150", "Diesel", "Islamabad", telemetry,
300);

    meterVehicle.displayDetails();
    cout << "-----\n";
    pipelineVehicle.displayDetails();
    cout << "-----\n";
    maintenanceVehicle.displayDetails();

    Vehicle* vehiclesArray[] = { &meterVehicle, &pipelineVehicle, &maintenanceVehicle };
    VehicleManager manager;
    manager.writeToFile(vehiclesArray, 3);
    cout << "-----\n";
    cout << "Vehicles in Karachi:\n";
    manager.readFromFile();
}

```

```
    return 0;  
}
```

Question : 5

Note:

1. It is a demo solution. Other solutions can also be correct.
2. For the solution below, I have created both functions as pure virtual in JewelryItem; only one can be considered correct.

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

```
using namespace std;
```

```
// Abstract class JewelryItem
```

```
class JewelryItem {
```

```
protected:
```

```
    string itemCode;  
    string itemName;  
    double weightInGrams;  
    int purity;
```

```
public:
```

```
    JewelryItem(string code, string name, double weight, int pur) : itemCode(code),  
    itemName(name), weightInGrams(weight), purity(pur) {}  
    virtual void displayDetails() const = 0;  
    virtual double calculatePrice() const = 0;  
};
```

```
// GoldJewelry class derived from JewelryItem
```

```
class GoldJewelry : public JewelryItem {
```

```
private:
```

```
    int goldKarat;
```

```
public:
```

```
    GoldJewelry(string code, string name, double weight, int pur) : JewelryItem(code, name, weight,  
    pur) {}  
    void setGoldKarat(int karat) {  
        goldKarat = karat;  
    }  
    void displayDetails() const override {
```

```

        cout << "Item Code: " << itemCode << endl;
        cout << "Item Name: " << itemName << endl;
        cout << "Weight: " << weightInGrams << " grams" << endl;
        cout << "Purity: " << purity << "%" << endl;
        cout << "Gold Karat: " << goldKarat << endl;
    }
    double calculatePrice() const override {
        return weightInGrams * (purity / 100.0) * goldKarat * 50; // Example pricing formula
    }
};

// DiamondJewelry class derived from JewelryItem
class DiamondJewelry : public JewelryItem {
private:
    int numDiamonds;
    double diamondCarat;

public:
    DiamondJewelry(string code, string name, double weight, int pur) : JewelryItem(code, name,
weight, pur) {}
    void addDiamonds(int num, double carat) {
        numDiamonds = num;
        diamondCarat = carat;
    }
    void displayDetails() const override {
        cout << "Item Code: " << itemCode << endl;
        cout << "Item Name: " << itemName << endl;
        cout << "Weight: " << weightInGrams << " grams" << endl;
        cout << "Purity: " << purity << "%" << endl;
        cout << "Number of Diamonds: " << numDiamonds << endl;
        cout << "Diamond Carat: " << diamondCarat << endl;
    }
    double calculatePrice() const override {
        return weightInGrams * (purity / 100.0) * diamondCarat * 1000; // Example pricing formula
    }
};

// Customer class
class Customer {
private:
    string customerID;
    string name;
    JewelryItem* purchasedGold;
    JewelryItem* purchasedDiamond;

public:
    Customer(string id, string nm) : customerID(id), name(nm), purchasedGold(NULL),
purchasedDiamond(NULL) {}

```

```

    void purchaseGold(GoldJewelry* gold) {
        purchasedGold = gold;
    }
    void purchaseDiamond(DiamondJewelry* diamond) {
        purchasedDiamond = diamond;
    }
    double calculateTotalPurchasePrice() const {
        double total = 0.0;
        if (purchasedGold != NULL) {
            total += purchasedGold->calculatePrice();
        }
        if (purchasedDiamond != NULL) {
            total += purchasedDiamond->calculatePrice();
        }
        return total;
    }
};

// StoreInventory class
class StoreInventory {
private:
    static const int MAX_ITEMS = 10; // Maximum number of items in the inventory
    JewelryItem* inventory[MAX_ITEMS];
    int itemCount;

public:
    StoreInventory() : itemCount(0) {}

    void addItemToInventory(JewelryItem* item) {
        if (itemCount < MAX_ITEMS) {
            inventory[itemCount++] = item;
        } else {
            cout << "Inventory is full. Cannot add more items." << endl;
        }
    }

    void displayInventory() const {
        if (itemCount == 0) {
            cout << "Inventory is empty." << endl;
        } else {
            cout << "Store Inventory:" << endl;
            for (int i = 0; i < itemCount; ++i) {
                cout << "Item " << i + 1 << ": " << endl;
                inventory[i]->displayDetails();
                cout << "Price: $" << inventory[i]->calculatePrice() << endl;
                cout << endl;
            }
        }
    }
};

```

```

    }
};

// Operator overloading for comparison of JewelryItem objects based on their price
bool operator<(const JewelryItem& item1, const JewelryItem& item2) {
    return item1.calculatePrice() < item2.calculatePrice();
}

bool operator>(const JewelryItem& item1, const JewelryItem& item2) {
    return item1.calculatePrice() > item2.calculatePrice();
}

bool operator==(const JewelryItem& item1, const JewelryItem& item2) {
    return item1.calculatePrice() == item2.calculatePrice();
}

int main() {
    // Example usage
    GoldJewelry goldItem1("G1001", "Gold Necklace", 20.5, 95);
    goldItem1.setGoldKarat(22);
    DiamondJewelry diamondItem1("D2001", "Diamond Ring", 5.2, 99);
    diamondItem1.addDiamonds(1    0, 0.5);

    GoldJewelry goldItem2("G1002", "Gold Bracelet", 15.0, 90);
    goldItem2.setGoldKarat(24);
    DiamondJewelry diamondItem2("D2002", "Diamond Earrings", 8.0, 98);
    diamondItem2.addDiamonds(6, 0.3);

    Customer customer("C001", "John Doe");
    customer.purchaseGold(&goldItem1);
    customer.purchaseDiamond(&diamondItem1);

    StoreInventory inventory;
    inventory.addItemToInventory(&goldItem1);
    inventory.addItemToInventory(&diamondItem1);
    inventory.addItemToInventory(&goldItem2);
    inventory.addItemToInventory(&diamondItem2);

    cout << "Customer Total Purchase Price: $" << customer.calculateTotalPurchasePrice() << endl;
    inventory.displayInventory();

    return 0;
}

```

Question : 6

Note:

1. It is a demo solution. Other solutions can also be correct.

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

using namespace std;

// Custom exception classes
class DuplicateItemException : public exception {
public:
    const char* find() const {
        return "Duplicate Item Exception";
    }
};

class ItemNotFoundException : public exception {
public:
    const char* find() const {
        return "Item Not Found Exception";
    }
};

class OutOfBoundException : public exception {
public:
    const char* find() const {
        return "Out of bound";
    }
};

// Base class for products
class Product {
public:
    virtual void print() const = 0;
    virtual bool equals(const Product& other) const = 0;
};

// Specific product types
class Book : public Product {
public:
    string title;
    string author;

    Book(const string& bookTitle, const string& bookAuthor) :
        title(bookTitle), author(bookAuthor) {}

    void print() const override {
        cout << "Book: " << title << " by " << author << endl;
    }
};
```

```

        bool equals(const Product& other) const override {
            const Book* otherBook = dynamic_cast<const Book*>(&other);
            return otherBook && title == otherBook->title && author == otherBook->author;
        }
};

class Electronic : public Product {
public:
    string name;
    double price;

    Electronic(const string& itemName, double itemPrice) :
        name(itemName), price(itemPrice) {}

    void print() const override {
        cout << "Electronic: " << name << " costs $" << price << endl;
    }

    bool equals(const Product& other) const override {
        const Electronic* otherElectronic = dynamic_cast<const Electronic*>(&other);
        return otherElectronic && name == otherElectronic->name && price == otherElectronic->price;
    }
};

class Clothing : public Product {
public:
    string type;
    string size;

    Clothing(const string& itemType, const string& itemSize) :
        type(itemType), size(itemSize) {}

    void print() const override {
        cout << "Clothing: " << type << " size " << size << endl;
    }

    bool equals(const Product& other) const override {
        const Clothing* otherClothing = dynamic_cast<const Clothing*>(&other);
        return otherClothing && type == otherClothing->type && size == otherClothing->size;
    }
};

// Define the UniqueCart template class
template<typename T, int MaxItems>
class UniqueCart {
private:
    T* items[MaxItems];

```

```

    int itemCount;

public:
    UniqueCart() : itemCount(0) {
        for (int i = 0; i < MaxItems; ++i) {
            items[i] = NULL;
        }
    }

    ~UniqueCart() {
        for (int i = 0; i < itemCount; ++i) {
            delete items[i];
        }
    }

    // Method to add items to the cart
    void add(T* item) {
        if (itemCount >= MaxItems) {
            throw OutOfBoundException();
        }
        for (int i = 0; i < itemCount; ++i) {
            if (items[i] && items[i]->equals(*item)) {
                throw DuplicatItemException();
            }
        }
        items[itemCount++] = item;
        cout << "Item added to cart." << endl;
    }

    // Method to remove items from the cart
    void remove(const T& item) {
        bool found = false;
        for (int i = 0; i < itemCount; ++i) {
            if (items[i] && items[i]->equals(item)) {
                found = true;
                cout << "Item removed from cart." << endl;
                delete items[i];
                items[i] = NULL;
                --itemCount;
            }
            // Shift items to fill the gap
            for (int j = i; j < itemCount; ++j) {
                items[j] = items[j + 1];
            }
            items[itemCount] = NULL;
            break;
        }
        if (!found) {

```



```

        throw ItemNotFoundException();
    }
}

// Method to check if a specific item exists in the cart
bool contains(const T& item) const {
    for (int i = 0; i < itemCount; ++i) {
        if (items[i] && items[i]->equals(item)) {
            return true;
        }
    }
    return false;
}

// Method to print all items in the cart
void print() const {
    for (int i = 0; i < itemCount; ++i) {
        items[i]->print();
    }
}
};

int main() {
    // Create a cart that can hold up to 10 items of various types
    UniqueCart<Product, 10> cart;

    // Add items to the cart
    try {
        cart.add(new Book("The Great Gatsby", "F. Scott Fitzgerald"));
        cart.add(new Electronic("Laptop", 1200.0));
        cart.add(new Clothing("T-shirt", "Medium"));
        // Adding a duplicate item
        // cart.add(new Book("The Great Gatsby", "F. Scott Fitzgerald"));
    } catch (const OutOfBoundException& ex) {
        cout << "Error: " << ex.find() << endl;
    } catch (const DuplicateItemException& ex) {
        cout << "Error: " << ex.find() << endl;
    }

    // Print all items in the cart
    cart.print();

    // Create items to remove/check
    Book book("The Great Gatsby", "F. Scott Fitzgerald");
    Electronic laptop("Laptop", 1200.0);

    // Remove an item from the cart
    try {

```

```
    cart.remove(book);  
    // Trying to remove a non-existing item  
    // cart.remove(laptop);  
    } catch (const ItemNotFoundException& ex) {  
        cout << "Error: " << ex.find() << endl;  
    }  
  
    // Check if items are in the cart  
    cout << "Cart contains The Great Gatsby: " << cart.contains(book) << endl;  
    cout << "Cart contains Laptop: " << cart.contains(laptop) << endl;  
  
    return 0;  
}
```