Q1: Write on the answer sheet the output of the following programs, when they are executed. There are no compilation errors in the programs.

A.
```
#include<iostream>
using namespace std;
class Point {
private:
    int x, y;
public:
      Point(){Point(1,1);}
    Point(int i, int j);
    Point(const Point &t);
};
Point::Point(int i, int j)  {
    x = i;
    y = j;
    cout  <<  x<<"  "<<y<<"Normal
Constructor called\n";
}
Point::Point(const Point &t) {
   y = t.y;
   cout   <<   y   <<"   "<<"Copy
Constructor called\n";
}
int main()
{
   Point *t1, *t2;
   t1 = new Point(10, 15);
   t2 = new Point(*t1);
   Point t3 = *t1;
   Point t4;
   t4 = t3;
   return 0;
}

Output:
```
```
10 15Normal Constructor called
15 Copy Constructor called
15 Copy Constructor called
1 1Normal Constructor called
```

B.
```
#include<iostream>
using namespace std;
int func(int n){
      if(n==0){
            cout<<"The    value    is
zero"<<endl;
            return 10;}
      if(n>0){
            cout<<"The    value    is
greater than zero"<<endl;
            throw 'e';}
      if(n<0){
            cout<<"The  value  is  less
than zero"<<endl;
            throw 9.9f;}
}
int main(){
     try {
      func(0); func(10); func(-10);
     } catch (int x) {
      cout << "Catching Integer\n";
      func(10);
     } catch (float f) {
      cout << "Catching Float\n";
      func(10);
     } catch (char c) {
      cout << "Catching Character\n";
      func(10);
     }
}

Output:
```
```
The value is zero
The value is greater than zero
Catching Character
The value is greater than zero
terminate  called  after  throwing  an
instance of 'char'
Aborted
```

C.

```
#include <iostream>
class Base {
public:
    static int count;
    Base() {count++;}
    virtual ~Base() {count--;}
    static void printCount() {
    std::cout  <<  "Count:  "  <<
count << std::endl;
    }
};
class Derived : public Base {
public:
    Derived() {count++;}
    ~Derived() {count--;}
};
int Base::count = 0;
int main() {
    Base::printCount();
    {
        Base obj1;
        Derived obj2;
        Base::printCount();
    }
    Base::printCount();
    return 0;
}
```

Output:

```
Count: 0
Count: 3
Count: 0
```

D.

```
#include<iostream>
using namespace std;
class A
{
public:
    A    (){    cout    <<    "\n    A's
constructor"; }
    A (const A &a) { cout << "\n A's
Copy constructor";}
    A& operator = (const A &a)
    {
       if(this == &a) return *this;
       cout  <<  "\n  A's  Assignment
Operator";  return *this;
    }
};
class B
{
    A a;
public:
    B(A &a) { this->a = a; cout << "\n
B's constructor"; }
};
int main()
{
    A a1;
    B b(a1);
    return 0;
}
```

Output:

```
A's constructor
 A's constructor
 A's Assignment Operator
 B's constructor
```

Q2: Considering the output given, complete the following code snippets.

A.
```
#include <iostream>
class Base {
private:
    int data;
public:
    Base(int value) : data(value) {}
    void printData() {
        std::cout << "Data: " << data << std::endl;
    }
  Base operator+(const Base& other) {
        return Base(data + other.data);
    }
};
int main() {
    Base obj1(10);
    Base obj2(20);
    Base result = obj1 + obj2;
    result.printData();
    return 0;}
```

```
Output: 30
```

B.
```
#include<iostream>
string maximum(T a, U b) {
    if ((typeid(a).name()) == (typeid(b).name()))
        return (a > b) ? to_string(a) : to_string(b);
    else
        return "Incompatible types";
}
int main() {
    int intMax = maximum<int>(5, 10);
    std::cout << "Max of 5 and 10: " << intMax <<
std::endl;
    double doubleMax = maximum<float>(3.14, 2.71);
    std::cout << "Max of 3.14 and 2.71: " << doubleMax
<< std::endl;
    char charMax = maximum<char>('a', 'z');
    std::cout << "Max of 'a' and 'z': " << charMax <<
std::endl;
    return 0;
  }
```

```
Output:
Maximum of 5
and 10: 10
Maximum of
3.14 and 2.71:
3.14
Maximum of 'a'
and 'z': z
```

C.
```
#include <iostream>
#include <fstream>
#include <string>

class Person {
  protected:
      std::string name;
      int age;
  public:
      Person(const std::string& name, int age) :
  name(name), age(age) {}
     virtual void saveToFile(const std::string&
  filename) const = 0;
```

```
Output:
Data saved to
file with
details:
Ali
21
FAST NUCES
```

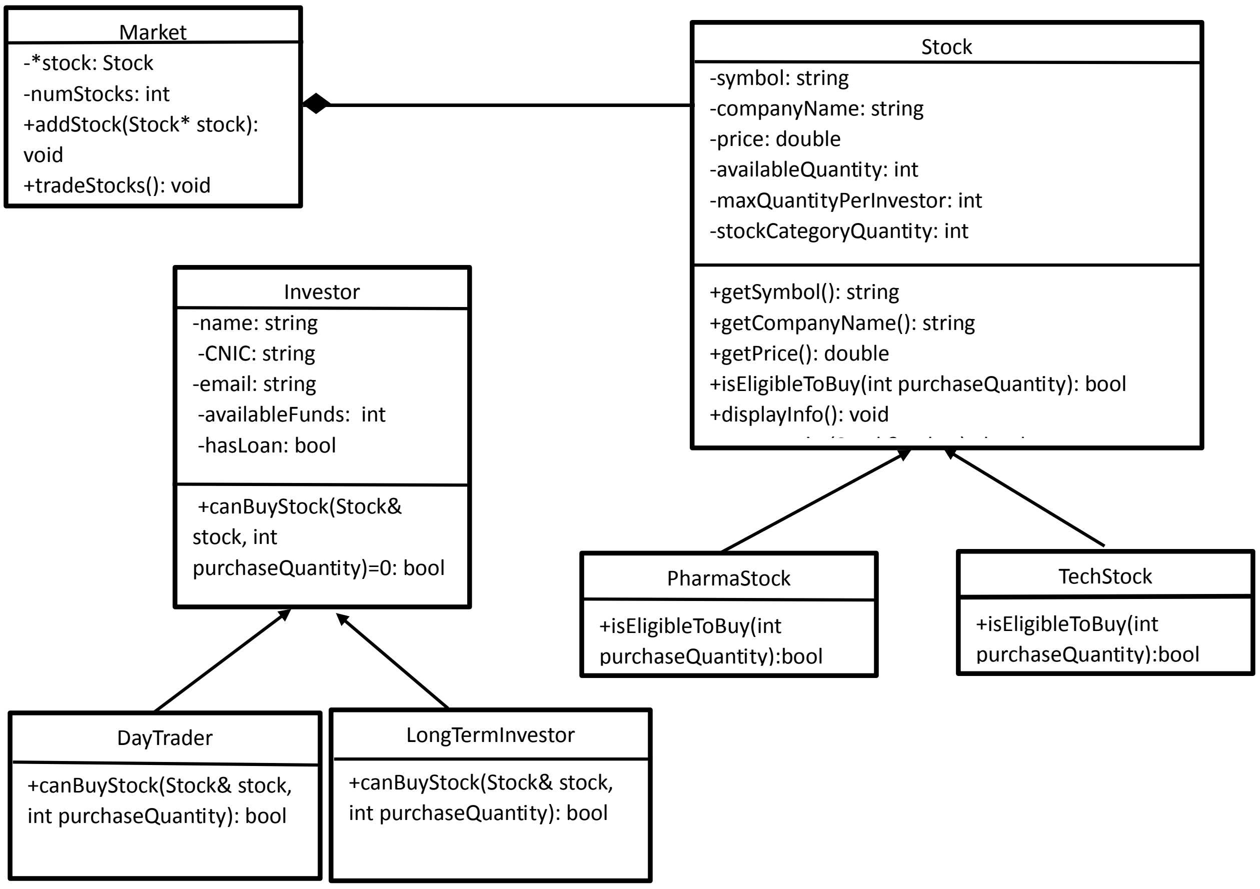
```
  };
class Student : public Person {
  private:
      std::string university;
  public:
  Student(const std::string& name, int age, const std::string& university)
  : Person(name, age), university(university) {}

  void saveToFile(const std::string& filename) const override {
          std::ofstream file(filename);
          if (file.is_open()) {
              file << "Name: " << name << std::endl;
              file << "Age: " << age << std::endl;
              file << "University: " << university << std::endl;
              file.close();
              std::cout << "Data saved to file with details : " << name <<
  “\n” << age  << “\n” << university << std::endl;
          } else {
              std::cerr << "Unable to open file: " << filename << std::endl;
          }
      }
  };
  int main() {
      Student student("Ali", 21, "FAST NUCES");
      student.saveToFile("student.txt");
      return 0;
  }
```

D.

```
#include <iostream>
  template<typename T>
  class Pair {
  private:
      T first;
      T second;
  public:
      Pair(T f, T s) : first(f), second(s) {}
      T sum() const {
          return first + second;
      }
      template<typename U>
      friend U average(const Pair<U>& p);
  };
  template<typename U>
  U average(const Pair<U>& p) {
      return (p.first + p.second) / static_cast<U>(2);
  }
  int main() {
      Pair<int> p(5, 7);
      std::cout << "Sum: " << p.sum() << std::endl;
      std::cout<<"Average: "<<average(p)<< std::endl;
  }
```

```
Output:
Sum: 12
Average: 6
```


Market
-*stock: Stock
-numStocks: int
+addStock(Stock* stock): void
+tradeStocks(): void
Stock
-symbol: string
-companyName: string
-price: double
-availableQuantity: int
-maxQuantityPerInvestor: int
-stockCategoryQuantity: int
+getSymbol(): string
+getCompanyName(): string
+getPrice(): double
+isEligibleToBuy(int purchaseQuantity): bool
+displayInfo(): void
Investor
-name: string
-CNIC: string
-email: string
-availableFunds: int
-hasLoan: bool
+canBuyStock(Stock& stock, int purchaseQuantity)=0: bool
PharmaStock
+isEligibleToBuy(int purchaseQuantity):bool
TechStock
+isEligibleToBuy(int purchaseQuantity):bool
DayTrader
+canBuyStock(Stock& stock, int purchaseQuantity): bool
LongTermInvestor
+canBuyStock(Stock& stock, int purchaseQuantity): bool

Q3: [25 min, 10 Marks, CLO 3] Consider the above given class diagram that demonstrates a stock market scenario. You are required to write the skeleton of the classes (i.e., member variables and function signatures) as shown in the above given class diagram. The parameterized constructor in all inherited classes must call parent's constructor to initialize variables. Create getter methods for all the attributes in all the classes. You are also required to define following functions:

- Stock::displayInfo() const: Displays the information about the stock. Prints the symbol, company name, price, available quantity, max quantity per investor, and category quantity of the stock.
- Market::getNumStocks() const: Gets the number of stocks currently in the market. Returns An integer representing the number of stocks in the market.

Q4: Use the class diagram and your answer of Q3 as reference and write programs for the following.

A. Implement functions as described below.

Stock::isEligibleToBuy() const: Following checks must be confirmed to check the buying eligibility of a stock. A stock is eligible for purchase if all conditions are met.

- Purchase quantity should not exceed maximum quantity limit per investor. Also, purchase quantity should not exceed the available quantity of the stock. If the purchaseQuantity is less than or equal to zero, then display an error message stating that the purchase quantity is invalid and return false.

TechStock::isEligibleToBuy(): Along with all parent class conditions, eligibility to buy of TechStock includes following checks.

- The purchase quantity must be a multiple of 10 for TechStock. If not, it displays an error message and returns false. Also, if the purchaseQuantity is greater than 100, it will display an error message stating that the maximum purchase quantity for TechStock is 100 and return false.

PharmaStock::isEligibleToBuy(): Along with all parent class conditions, eligibility to buy of PharmaStock includes following checks.:

- The purchase quantity must be at least 50 for PharmaStock. If not, it displays an error message and returns false. If the purchaseQuantity is not a multiple of 5, it will display an error message indicating that the quantity must be a multiple of 5 for PharmaStock and return false.

B. Implement functions as described below.

Investor::canBuyStock() is pure virtual function.

DayTrader::canBuyStock(): Checks if the investor can buy a given stock based on their financial status. If the investor has availed a loan, it displays an error message and returns false. The function should also calculate the total price of the stock purchase and check if it exceeds the available funds of the day trader. If so, it displays an error message and returns false. Finally, uses isEligibleToBuy() method of the stock and returns its response.

LongTermInvestor::canBuyStock(): Checks if the investor can buy a given stock based on their financial status. If the LongTermInvestor has availed a loan and has availableFunds less than 50000, it displays an error message and returns false. The function should also check that the purchaseQuantity does not exceed the maxQuantityPerInvestor limit. If it does, then it displays an error message and returns false. Finally, uses isEligibleToBuy() method of the stock and returns its response.

C. Overloaded inequality operator to compare stocks based on their symbols¸ i.e. the following statement should work: bool isNotEqual = stockObject1 != stockObject 2.

A stock is not equal to another stock if either the name of the stock company or the stock symbol is different from the other stock.

```
#include <iostream>
#include <string>
using namespace std;
#define MAX_STOCKS 100
class Stock {
protected:
    string symbol;
    string companyName;
    double price;
    int availableQuantity;
    int maxQuantityPerInvestor;
    int stockCategoryQuantity; // Number of stocks available for each category

public:
    Stock(const string& stockSymbol, const string& company, double stockPrice, int quantityLimit, int
categoryQuantity)
        : symbol(stockSymbol), companyName(company), price(stockPrice), availableQuantity(quantityLimit),
        maxQuantityPerInvestor(quantityLimit / 10), stockCategoryQuantity(categoryQuantity) {}

    string getSymbol() const {
        return symbol;
    }

    string getCompanyName() const {
        return companyName;
    }

    double getPrice() const {
        return price;
    }

    bool isEligibleToBuy(int purchaseQuantity) const {

    if (purchaseQuantity <= 0) {
        cout << "Cannot buy stock. Invalid purchase quantity." << endl;
        return false;

    }
   if (purchaseQuantity > maxQuantityPerInvestor) {
        cout << "Cannot buy stock. Maximum quantity limit per investor exceeded." << endl;
        return false;
    }

    if (purchaseQuantity > availableQuantity) {
        cout << "Cannot buy stock. Insufficient stock quantity." << endl;
        return false;
    }

    return true;
}



    void displayInfo() const {
        cout << "Symbol: " << symbol << endl;
        cout << "Company Name: " << companyName << endl;
        cout << "Price: " << price << endl;
        cout << "Available Quantity: " << availableQuantity << endl;
        cout << "Max Quantity Per Investor: " << maxQuantityPerInvestor << endl;
        cout << "Category Quantity: " << stockCategoryQuantity << endl;
    }

    // Overload != operator
    bool operator!=(const Stock& other) const {
        return symbol != other.symbol;
    }
};

class TechStock : public Stock {
public:
    TechStock(const string& stockSymbol, const string& company, double stockPrice, int quantityLimit)
        : Stock(stockSymbol, company, stockPrice, quantityLimit, 0) {}

    bool isEligibleToBuy(int purchaseQuantity) {
        if (!Stock::isEligibleToBuy(purchaseQuantity)) {
            return false;
        }

        // Additional condition specific to TechStock
        if (purchaseQuantity % 10 != 0) {
            cout << "Cannot buy stock. Quantity must be a multiple of 10 for TechStock." << endl;
            return false;
        }

         if (purchaseQuantity > 100) {
        cout << "Cannot buy stock. Maximum purchase quantity for TechStock is 100." << endl;
        return false;
    }

        return true;
    }
};

class PharmaStock : public Stock {
public:
    PharmaStock(const string& stockSymbol, const string& company, double stockPrice, int quantityLimit)
        : Stock(stockSymbol, company, stockPrice, quantityLimit, 0) {}

    bool isEligibleToBuy(int purchaseQuantity) {
        if (!Stock::isEligibleToBuy(purchaseQuantity)) {
            return false;
        }

        // Additional condition specific to PharmaStock
        if (purchaseQuantity < 50) {
            cout << "Cannot buy stock. Minimum purchase quantity for PharmaStock is 50." << endl;
            return false;
        }
        // Additional condition specific to PharmaStock
    if (purchaseQuantity % 5 != 0) {
        cout << "Cannot buy stock. Quantity must be a multiple of 5 for PharmaStock." << endl;
        return false;
    }

        return true;
    }
};

class Investor {
protected:
    string name;
    string CNIC;
    string email;
    int availableFunds;
    bool hasLoan;

public:
    Investor(const string& investorName, const string& cnic, const string& investorEmail, int funds)
        : name(investorName), CNIC(cnic), email(investorEmail), availableFunds(funds), hasLoan(false) {}

    virtual bool canBuyStock(const Stock& stock, int purchaseQuantity) const = 0;

    // Other member functions
};

class DayTrader : public Investor {
public:
    DayTrader(const string& traderName, const string& cnic, const string& traderEmail, int funds)
        : Investor(traderName, cnic, traderEmail, funds) {}

    bool canBuyStock(const Stock& stock, int purchaseQuantity) const override {
        if (hasLoan) {
            cout << "Cannot buy stock. Loan availed." << endl;
            return false;
        }

        double totalPrice = stock.getPrice() * purchaseQuantity;
        if (totalPrice > availableFunds) {
            cout << "Cannot buy stock. Insufficient funds." << endl;
            return false;
        }

        return stock.isEligibleToBuy(purchaseQuantity);
    }
};

class LongTermInvestor : public Investor {
public:
    LongTermInvestor(const string& investorName, const string& cnic, const string& investorEmail, int funds)
        : Investor(investorName, cnic, investorEmail, funds) {}

    bool canBuyStock(const Stock& stock, int purchaseQuantity) const override {
        if (hasLoan) {
            cout << "Cannot buy stock. Loan availed." << endl;
            return false;
        }

        double totalPrice = stock.getPrice() * purchaseQuantity;
        if (totalPrice > availableFunds) {
            cout << "Cannot buy stock. Insufficient funds." << endl;
            return false;
        }

        return stock.isEligibleToBuy(purchaseQuantity);
    }
};

template <class T>
class Market {
public:
    T* stocks[MAX_STOCKS];
    int numStocks;

public:
    Market() : numStocks(0) {}

    void addStock(T* stock) {
        if (numStocks < MAX_STOCKS) {
            if (stock == NULL) {
                cout << "Cannot add stock. Invalid stock object." << endl;
                return;
            }

            stocks[numStocks] = stock;
            numStocks++;
        } else {
            cout << "Cannot add stock. Maximum number of stocks reached." << endl;
        }
    }

    void tradeStocks() {
        // Simulate trading of stocks
        for (int i = 0; i < numStocks; i++) {
            T* stock = stocks[i];
            // Perform trading operations
        }
    }

    int getNumStocks() const {
        return numStocks;
    }
};

int main() {
    // Create stocks and investors
    TechStock* techStock = new TechStock("AAPL", "Apple Inc.", 150.50, 1000);
    PharmaStock* pharmaStock = new PharmaStock("PFE", "Pfizer Inc.", 35.75, 500);
    techStock->displayInfo();
    pharmaStock->displayInfo();
    DayTrader dayTrader("John Doe", "1234567890", "john.doe@example.com", 10000);
    LongTermInvestor longTermInvestor("Jane Smith", "0987654321", "jane.smith@example.com", 50000);

    // Create market
    Market<Stock> market;

    // Add stocks to the market
    market.addStock(techStock);
    market.addStock(pharmaStock);

    // Trade stocks in the market
    cout << "Trading stocks..." << endl;
    for (int i = 0; i < market.getNumStocks(); i++) {
        Stock* stock = market.stocks[i];

        // Check if investors can buy the stock
        if (dayTrader.canBuyStock(*stock, 20)) {
            cout << "DayTrader bought stock: " << stock->getSymbol() << endl;
        }

        if (longTermInvestor.canBuyStock(*stock, 200)) {
            cout << "LongTermInvestor bought stock: " << stock->getSymbol() << endl;
        }
    }

    // Cleanup
    delete techStock;
    delete pharmaStock;

    return 0;
}
```

Q5: [45 min, 10+10+10 Marks, CLO 5] Consider a chatbot system designed to provide responses to users' queries. The system consists of four chatbot variants tailored for medical, technology, legal, and general queries. Your task is to implement an object-oriented program that fulfills the following requirements:

A. Design a User class to store user data, including attributes such as username, country, interest, and age. User have a method Ask() which takes a string as a query and generates a specific response.
   - The medical chatbot should respond if the query's prefix (first word) is "doc". The legal chatbot should respond if the query begins with "attorney". The technology chatbot should respond only if the query starts with "guru".
   - If a query begins with "special", check the user's interest, and forward the query to the relevant chatbot variant based on their interest.
   - If a query does not match the relevant prefix, it must throw a custom exception object of type "Bot_Exception" with an error message.

B. Implement a Chatbot class as the base class for all chatbot variants. Each chatbot variant (MedicalChatbot, TechnologyChatbot, LegalChatbot, GeneralChatbot) should be inherited from the Chatbot class. Make sure that each chatbot class should keep track of the number of instances created throughout the program.
   - Each chatbot variant should have a method **string generate_response(string query, User u)** to generate responses based on user queries. It should store the name of the most recent user that interacted with it and maintain a total user count, tracking the number of users who have ever interacted with it.
   - Provide functionality to access the total user count for each chatbot variant at any given time.

C. As specified in part A, when a chatbot variant throws a "Bot_Exception", an error message should notify the user that the query is invalid. Capture the username and query of the user causing the exception and write it to the "error_log.txt" file. The "error_log.txt" file should contain a list of usernames + queries for users who caused exceptions to be thrown. Also, you need to write a function “Analysis()” that will open the file “error_log.txt” in read mode and will perform the following analysis:

   - It will print the username who caused the maximum number of exceptions.
   - It will print the total count of words from each query stored.

Note: Ensure that your program demonstrates proper usage of object-oriented principles such as inheritance, encapsulation, exception handling, generics, and file handling. Implement appropriate methods and attributes in each class to fulfill the requirements outlined above.

```
class Bot_Exception {
public:
Bot_Exception(const string& errorMessage) : message(errorMessage) {}
string getMessage() const {
return message; }
private:
string message;
};

// Part A
class User {
public:
User(const string& uname, const string& cntry, const string& intst, int usrAge)
: username(uname), country(cntry), interest(intst), age(usrAge) {}
 string getUsername() const {
return username; }
void setUsername(const  string& uname) {
username = uname; }
 string getCountry() const {
```

```
return country;
}
void setCountry(const  string& cntry) {
country = cntry; }
 string getInterest() const {
return interest; }
void setInterest(const  string& intst) {
interest = intst; }
int getAge() const {
return age; }
void setAge(int usrAge) {
age = usrAge; }
// Part A
Chatbot * c;
string Ask(const  string& query) const {
string prefix = getPrefix(query);
string response;
if (prefix == "doc") { c = new MedicalChatbot;
response = c->generate_response(query, this); }
else if (prefix == "attorney") { c = new LegalChatbot;
response = c->generate_response(query, this);  }
else if (prefix == "guru") { c = new TechnologyChatbot;
response = c->generate_response(query, this); }
else if (prefix == "special") { if (interest == "medical") { c = new MedicalChatbot;
response = c->generate_response(query, this);}
else if (interest == "legal") c = new LegalChatbot;
response = c->generate_response(query, this);  }
else if (interest == "technology") { c = new TechnologyChatbot;
response = c->generate_response(query, this); }
else { c = new GeneralChatbot;
response = c->generate_response(query, this); } }
else { throw Bot_Exception("Invalid query: " + query); }
return response;
}
private:
 string username;
 string country;
 string interest;
int age;
 string getPrefix(const  string& query) const {
 istringstream iss(query);
 string prefix;
iss >> prefix;
return prefix;
}
};

//Part B
class Chatbot {
public:
static int totalUserCount;
static string lastUser;
IncreaseUsers() { totaUserCount++; }
virtual  string generate_response(const  string& query, User u) = 0;
}
 string getLastUser() const {
return lastUser;
}
static int getTotalUserCount() {
return totalUserCount;
```

```
}
};
int Chatbot::totalUserCount = 0;
string Chatbot::lastUser = “ ”;

class MedicalChatbot : public Chatbot {
public:
string generate_response(const  string& query, User u) {
IncreaseUsers();
lastUser = u.getUsername();
string response; // create response
try {
// response generation logic
} catch (const Bot_Exception& e) {
logError(u.getUsername(), query);
response = "Error: " + e.getMessage();
}
return response;
}
private:
// Part C
void logError(const  string& username, const  string& query) {
 ofstream errorLog("error_log.txt",  ios::app);
if (errorLog.is_open()) {
errorLog << "Username: " << username << ", Query: " << query <<  endl;
errorLog.close();
}
}
};

class LegalChatbot : public Chatbot {
public:
string generate_response(const  string& query, User u) {
IncreaseUsers();
lastUser = u.getUsername();

string response; // create response
try {
// response generation logic
} catch (const Bot_Exception& e) {
logError(u.getUsername(), query);
response = "Error: " + e.getMessage();
}
return response;
}
// Part C
private:
void logError(const  string& username, const  string& query) {
 ofstream errorLog("error_log.txt",  ios::app);
if (errorLog.is_open()) {
errorLog << "Username: " << username << ", Query: " << query <<  endl;
errorLog.close();
}
}
};

class TechnologyChatbot : public Chatbot {
public:
string generate_response(const  string& query, User u) {
IncreaseUsers();
```

```
lastUser = u.getUsername();
string response; // create response
try {
// response generation logic
} catch (const Bot_Exception& e) {
logError(u.getUsername(), query);
response = "Error: " + e.getMessage();
}
return response;
}
```

// Part C

```
private:
void logError(const  string& username, const  string& query) {
 ofstream errorLog("error_log.txt",  ios::app);
if (errorLog.is_open()) {
errorLog << "Username: " << username << ", Query: " << query <<  endl;
errorLog.close();
}
}
};

class GeneralChatbot : public Chatbot {
public:
string generate_response(const  string& query, User u) {
IncreaseUsers();
lastUser = u.getUsername();
string response; // create response
try {
// response generation logic
} catch (const Bot_Exception& e) {
logError(u.getUsername(), query);
response = "Error: " + e.getMessage();
}
return response;
}
```

// Part C

```
private:
void logError(const  string& username, const  string& query) {
 ofstream errorLog("error_log.txt",  ios::app);
if (errorLog.is_open()) {
errorLog << "Username: " << username << ", Query: " << query <<  endl;
errorLog.close();
}
}
};

void Analysis() {
 ifstream errorLog("error_log.txt");
if (!errorLog.is_open()) {
 cout << "Error: Failed to open error_log.txt" <<  endl;
return;
}
 map< string, int> exceptionCount;
 map< string, int> wordCount;
 string line;
while ( getline(errorLog, line)) {
 istringstream iss(line);
 string username, query;
iss >> username >> query;
// Count exceptions per user
if (exceptionCount.find(username) != exceptionCount.end()) {
exceptionCount[username]++;
} else {
exceptionCount[username] = 1;
}
// Count words in query
 istringstream wordStream(query);
 string word;
while (wordStream >> word) {
wordCount[username]++;
}
}
errorLog.close();
// Print username with maximum exceptions
int maxExceptions = 0;
 string maxUser;
for (const auto& entry : exceptionCount) {
if (entry.second > maxExceptions) {
maxExceptions = entry.second;
maxUser = entry.first;
} }
 cout << "Username with maximum exceptions: " << maxUser <<  endl;
// Print total count of words from each query
for (const auto& entry : wordCount) {
 cout << "Username: " << entry.first << ", Total word count: " << entry.second <<  endl;
} }
```