DOCUMENTATION

Project Name: ***Hospital Management App***

Student's Name: ***Shurid Sadi Mahmud***

Neptun Code: **NN449O**

Introduction:

The Hospital Management App is an object-oriented programming (OOP) project that demonstrates the use of classes, inheritance, dynamic memory management(new and delete), encapsulation, file management, and exception handling to create a system for managing patients and appointments in a hospital. The application provides a user-friendly interface for performing various tasks such as creating patient records, displaying patient information, scheduling appointments, and displaying appointment details.

Object-Oriented Programming (OOP) is a programming paradigm that organizes code around objects, which are instances of classes. In this project, OOP principles have been applied to design and implement various classes that represent different entities and functionalities within the hospital management system.

Encapsulation is a concept in OOP that involves bundling data and related methods into a single unit called a class. In this project, the "Patient" class encapsulates patient information such as name, gender, and address. The class provides member functions to access and modify this information.

Inheritance is another important concept in OOP, allowing classes to inherit properties and behaviors from other classes. In this project, the "Appointment" class inherits from the "Patient" class, extending its functionalities to handle appointments. By inheriting from the "Patient" class, the "Appointment" class can use the existing attributes and methods defined in the parent class, promoting code to be reused.

Polymorphism, a key feature of OOP, enables objects of different classes to be treated interchangeably through a common interface. In this project, polymorphism is demonstrated when displaying patient records and appointment details. Both "Patient" and "Appointment" objects are processed and displayed using a common interface, facilitating code flexibility and modifiability.

File management and exception handling have also been inplemented where they were applicable.

By employing OOP concepts, this Hospital Management App demonstrates the benefits of code organization, and code reuse. The application provides a basic foundation for managing patient information and appointments, enhancing the overall efficiency and effectiveness of hospital operations.

Program Code with Description:

Headers:

`<iostream>:` This header file is used for input and output operations in C++ for reading and writing data to the standard input/output streams, such as cin and cout.

`<fstream>:` This header file is used for file management operations in C++. It provides classes functions for reading from and writing to files. It includes classes like ifstream (input file stream) and ofstream (output file stream) for handling input and output operations with files.

`<string>`: This header file provides the string class, which represents a sequence of characters. It offers various functions and operators for manipulating strings.

Classes:

Patient:

Code Snippet:

```
class Patient {
public:
    Patient() : name(nullptr), gender('\0'), address(nullptr) {}
    Patient(const string& n, char g, const string& a) {
        name = new string(n);
        gender = g;
        address = new string(a);
    }

    ~Patient() {
        delete name;
        delete address;
    }

    const string& getName() const {
        return *name;
    }

    char getGender() const {
        return gender;
    }

    const string& getAddress() const {
        return *address;
    }
    void createPatient() {
    string name, address;
    char gender;

    cout << "Enter patient name: ";
    getline(cin, name);

    cout << "Enter patient gender: ";
    cin >> gender;

    cin.ignore();  // Ignore the newline character left in the input buffer

    cout << "Enter patient address: ";
    getline(cin, address);

    Patient patient(name, gender, address);

    ofstream file;
```

```
    file.open("patients.txt", ios::out | ios::app);

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

    file << patient.getName() << "," << patient.getGender() << "," <<
patient.getAddress() << endl;

    file.close();

    cout << "Patient created and saved!" << endl;
}
    void displayPatients() {
        ifstream file;
        file.open("patients.txt", ios::in);

        if (!file) {
            cout << "No patient records found!" << endl;
            return;
        }

        string line;
        string name, address;
        char gender;

        cout << "Patient Records:" << endl;

        while (getline(file, line)) {
           size_t pos = 0;


            pos = line.find(",");
            name = line.substr(0, pos);
            line.erase(0, pos + 1);

            gender = line[0];
            line.erase(0, 2);

            address = line;

            Patient patient(name, gender, address);

            cout << "Name: " << patient.getName() << endl;
            cout << "Gender(M/F): " << patient.getGender() << endl;
            cout << "Address: " << patient.getAddress() << endl;
            cout << "-------------------------" << endl;
        }

        file.close();
    }

private:
    string* name;
    char gender;
    string* address;
};
```

Explanation:

The class has three **private** member variables:

name: A pointer to a string object representing the patient's name.

gender: A char variable representing the patient's gender.

address: A pointer to a string object representing the patient's address.

The **public** section of the class contains the following member functions:

Default Constructor: Patient():

Initializes the name, gender, and address pointers to nullptr and sets the gender to a null character ('\0').

Parameterized Constructor: Patient(const string& n, char g, const string& a)

Takes arguments for the patient's name (n), gender (g), and address (a).

Dynamically allocates memory for the name and address strings using the new operator.

Copies the values of the arguments into the dynamically allocated strings.

Destructor: ~Patient()

Deallocates the dynamically allocated memory for the name and address strings using the delete operator.

const string& getName() const:

Returns a const reference to the patient's name (name).

This allows accessing the name without making a copy and the const operator ensures the name cannot be modified through this reference.

char getGender() const:

Returns the patient's gender (gender) as a char value.

const string& getAddress() const:

Returns a const reference to the patient's address (address).

Similar to getName(), this allows accessing the address without making a copy and ensures the address cannot be modified through this reference.

Appointment:

Code snippet:

```
class Appointment : public Patient {
public:
    Appointment() {}
    Appointment(const string& n, char g, const string& a) : Patient(n, g, a) {}

    void saveAppointment() {
        ofstream file;
        file.open("appointments.txt", ios::out | ios::app);

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

        file << getName() << "," << getGender() << "," << getAddress() << endl;

        file.close();
    }
};

void displayHospitalInfo() {
    // Code for displaying hospital information
    cout << "Hospital Information" << endl;
    cout << "--------------------" << endl;
    cout << "Budapest University of Technology and Economics Health Center" << endl;
    cout << "Location:Budapest, Hungary" << endl;
    cout << "Contact: +3620596" << endl << "E-mail address:
bmehealthcenter@gmail.com" << endl;
    cout << "Website: www.bmehealthcenter.com" << endl;
    cout << endl;
}
```

Explanation:

The Appointment class inherits publicly from the Patient class. This means that an Appointment inherits all the public, private and protected members of the Patient class.

The class provides two constructors:

Default Constructor: Appointment()

It is an empty constructor that doesn't perform any additional initialization beyond what is provided by the default constructor of the base class (Patient).

Parameterized Constructor: Appointment(const string& n, char g, const string& a) : Patient(n, g, a)

It takes the patient's name (n), gender (g), and address (a) as arguments and initializes the Patient base class using the provided arguments.

The saveAppointment() function saves the appointment information to a file named "appointments.txt":

It creates an ofstream object named file and opens the file in append mode using the open() function.

If there is an error opening the file, it displays an error message and returns.

It then writes the appointment details (name, gender, and time) to the file using the file stream.

Finally, it closes the file using the close() function.

In summary, the Appointment class extends the functionality of the Patient class by adding the ability to save appointment information to a file. It inherits the properties and member functions of the Patient class and adds its own function saveAppointment() to handle file management.

Functions:

createPatient():

Code snippet:

```
void createPatient() {
 string name, address;
 char gender;

 cout << "Enter patient name: ";
 getline(cin, name);

 cout << "Enter patient gender: ";
 cin >> gender;

 cin.ignore();  // Ignore the newline character left in the input buffer

 cout << "Enter patient address: ";
 getline(cin, address);

 Patient patient(name, gender, address);

 ofstream file;
 file.open("patients.txt", ios::out | ios::app);

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

 file << patient.getName() << "," << patient.getGender() << "," <<
patient.getAddress() << endl;

 file.close();

 cout << "Patient created and saved!" << endl;
}
```

Explanation:

This function is responsible for creating a new patient and saving their information to a file named "patients.txt".

- ➔ It first prompts the user to enter the patient's name, gender, and address.
- ➔ The input for name and address is obtained using getline(cin, name) and getline(cin, address) respectively, the gender is read using cin >> gender.
- ➔ After reading the input, cin.ignore() is used to clear the newline character from the input buffer.
- ➔ A Patient object named patient is then created using the provided information.

- ➔ An output file stream file is opened in append mode using file.open("patients.txt", ios::out | ios::app).
- ➔ If there is an error opening the file, an error message is displayed, and the function returns.
- ➔ The patient's information is written to the file using file << patient.getName() << "," << patient.getGender() << "," << patient.getAddress() << endl.
- ➔ Finally, the file is closed, and a success message is displayed.

displayPatients():

Code snippet:

```
void displayPatients() {
    ifstream file;
    file.open("patients.txt", ios::in);

    if (!file) {
        cout << "No patient records found!" << endl;
        return;
    }

    string line;
    string name, address;
    char gender;

    cout << "Patient Records:" << endl;

    while (getline(file, line)) {
       size_t pos = 0;


        pos = line.find(",");
        name = line.substr(0, pos);
        line.erase(0, pos + 1);

        gender = line[0];
        line.erase(0, 2);

        address = line;

        Patient patient(name, gender, address);

        cout << "Name: " << patient.getName() << endl;
        cout << "Gender(M/F): " << patient.getGender() << endl;
        cout << "Address: " << patient.getAddress() << endl;
        cout << "------------------------" << endl;
    }

    file.close();
}
```

Explanation:

This function is responsible for reading patient records from the "patients.txt" file and displaying them on the console.

- ➔ It first opens an input file stream file using file.open("patients.txt", ios::in).
- ➔ If there is an error opening the file, a message is displayed, and the function returns.
- ➔ Inside a while loop, the function reads each line from the file using getline(file, line).
- ➔ Then it retrieves the patient's name, gender, and address from the line by using line.find(",") to find the comma-separated values and line.substr() to extract the substrings.
- ➔ A Patient object named patient is created using the retrieved information.
- ➔ The patient's information is then displayed on the console using patient.getName(), patient.getGender(), and patient.getAddress().
- ➔ After displaying each patient's information, a separator line is printed for visual separation.
- ➔ After that the file is closed.

int main():

Code snippet:

```
int main() {
    int choice;
    Patient patient;

    do {
        cout << "                                ||  HOSPITAL MANAGENENT APP  ||" <<
endl;
        cout << "                                                  ---------" << endl << endl
<< endl << endl;
        cout << "1. Create patient" << endl << endl;
        cout << "2. Display patients" << endl << endl;
        cout << "3. Create appointment" << endl << endl;
        cout << "4. Display appointments" << endl << endl;
        cout << "5. Contact" << endl << endl;
        cout << "0. Exit" << endl << endl;
```

```
        cout << "Enter your choice: ";
        cin >> choice;

        cin.ignore();  // Ignore the newline character left in the input buffer

        switch (choice) {
        case 1:
            system("cls");
            patient.createPatient();
            break;
        case 2:
            system("cls");
            patient.displayPatients();
            break;
        case 3: {
            system("cls");
            string name, time;
            char gender;

            cout << "Enter patient name: ";
            getline(cin, name);

            bool validGenderInput = false;

            while (!validGenderInput) {
                cout << "Enter patient gender(M/F): ";
                cin >> gender;

                cin.ignore();  // Ignore the newline character left in the input
buffer

                try {
                    if (gender != 'M' && gender != 'F') {
                        throw invalid_argument("Invalid gender input. Please enter
'M' or 'F'.");
                    }

                    validGenderInput = true;
                }
                catch (const exception& e) {
                    cout << "Error: " << e.what() << endl;
                }
            }

            cout << "Enter Time(HH:MM): ";
            getline(cin, time);

            Appointment* appointment = new Appointment(name, gender, time);
            appointment->saveAppointment();
            delete appointment;

            cout << "Appointment created and saved!" << endl;
            break;
        }
        case 4: {
            system("cls");
            ifstream file;
            file.open("appointments.txt", ios::in);
```

```
            if (!file) {
                cout << "No appointments found!" << endl;
                break;
            }

            string line;
            Appointment appointment;

            cout << "Appointments:" << endl;

            while (getline(file, line)) {
                size_t pos = 0;

                pos = line.find(",");
                appointment.Patient::Patient(line.substr(0, pos), line[pos + 1],
line.substr(pos + 3));
                line.erase(0, pos + 1);

                cout << "Name: " << appointment.getName() << endl;
                cout << "Gender(M/F): " << appointment.getGender() << endl;
                cout << "Time(HH:MM) " << appointment.getAddress() << endl;
                cout << "-----------------------" << endl;
            }

            file.close();
            break;
        }
        case 5:
            system("cls");
            displayHospitalInfo();
            break;
        case 0:
            cout << "Exiting..." << endl;
            break;
        default:
            cout << "Invalid choice!" << endl;
        }

        cout << endl;
    } while (choice != 0);

    return 0;
}
```

Explanation:

This part of the code represents the main function. This provides an interactive menu for the hospital management app, allowing the user to create patients, display patient records, create appointments, display appointment records, view hospital information, and exit the program.

- ➔ The int main() function begins by declaring an integer variable choice to store the user's menu selection.
- ➔ The code enters a do-while loop, which ensures that the menu is displayed and the corresponding actions are performed until the user chooses to exit (selects 0).
- ➔ Within the loop, the menu options are displayed using cout statements. The user is prompted to enter their choice.
- ➔ The user's input is stored in the choice variable using cin >> choice.
- ➔ cin.ignore() is called to ignore the newline character left in the input buffer after reading the choice.
- ➔ A switch statement is used to perform different actions based on the user's choice.
- ➔ Case 1: Calls the createPatient() function, which asks the user to enter patient details and saves the patient information to a file.
- ➔ Case 2: Calls the displayPatients() function, which reads patient records from a file and displays them on the screen.
- ➔ Case 3: It asks the user to enter the patient's name, gender, and appointment time. It validates the gender input and creates an Appointment object with the provided information. The saveAppointment() function is called to save the appointment details to a file.
- ➔ Case 4: Reads appointment records from a file and displays them on the screen by creating Appointment objects and retrieving the appointment details using the inherited getName(), getGender(), and getAddress() functions.
- ➔ Case 5: Calls the displayHospitalInfo() function, which displays information about the hospital.
- ➔ Case 0: Exits the program by breaking out of the do-while loop.
- ➔ Default case: Displays an "Invalid choice!" message if the user enters an invalid option.
- ➔ Finally, the return 0; statement indicates that the program has executed successfully.

Summary:

The Hospital Management App is a console application which is designed to assist in managing patient records and appointments in a hospital setting. It provides a menu interface for various functionalities, allowing users to create patients, display patient records, create appointments, display appointment records, and access contact information for the hospital.

*****