Proiect in C++:

Să discutăm despre un proiect simplu de sistem de gestionare a bibliotecii în C++.

Acest proiect va include funcționalități pentru:

1. **Adăugarea cărților**
2. **Ștergerea cărților**
3. **Afișarea cărților**
4. **Gestionarea utilizatorilor**
5. **Gestionarea împrumuturilor**

Vom folosi programarea orientată pe obiect (OOP) pentru a organiza codul într-un mod clar și modular.

În acest exemplu, vom crea câteva clase: Carte, Utilizator, Biblioteca, și Imprumut.

1. Definirea clasei Carte:

```
#include <iostream>
#include <string>
class Carte {
public:
   std::string titlu;
   std::string autor;
   int anPublicatie;
   bool esteDisponibila;

   Carte(std::string t, std::string a, int an)
      : titlu(t), autor(a), anPublicatie(an), esteDisponibila(true) {}

   void afiseazaCarte() const {
      std::cout << "Titlu: " << titlu << ", Autor: " << autor
            << ", Anul Publicatiei: " << anPublicatie
            << ", Disponibilitate: " << (esteDisponibila ? "Disponibila" :
"Imprumutata") << std::endl;
   }
};
```

2. Definirea clasei Utilizator:

```
#include <vector>
class Utilizator {public:
    std::string nume;
    std::vector<Carte*> cartiImprumutate;

    Utilizator(std::string n) : nume(n) {}

    void imprumutaCarte(Carte* carte) {
        if (carte->esteDisponibila) {
            cartiImprumutate.push_back(carte);
            carte->esteDisponibila = false;
            std::cout << "Cartea a fost imprumutata cu succes!" <<
std::endl;
        } else {
            std::cout << "Cartea nu este disponibila!" << std::endl;
        }
    }

    void returneazaCarte(Carte* carte) {
        for (auto it = cartiImprumutate.begin(); it !=
cartiImprumutate.end(); ++it) {
            if (*it == carte) {
                cartiImprumutate.erase(it);
                carte->esteDisponibila = true;
                std::cout << "Cartea a fost returnata cu succes!" <<
std::endl;
                return;
            }
        }
        std::cout << "Utilizatorul nu a imprumutat aceasta carte!" <<
std::endl;
    }

    void afiseazaCartiImprumutate() const {
        std::cout << "Carti imprumutate de " << nume << ":" << std::endl;
        for (const auto& carte : cartiImprumutate) {
            carte->afiseazaCarte();
        }
    }
```

```
};
```

3. Definirea clasei Biblioteca:

```
#include <vector>
class Biblioteca {
public:
    std::vector<Carte> carti;
    std::vector<Utilizator> utilizatori;

    void adaugaCarte(const Carte& carte) {
        carti.push_back(carte);
        std::cout << "Cartea a fost adaugata cu succes!" << std::endl;
    }

    void stergeCarte(const std::string& titlu) {
        for (auto it = carti.begin(); it != carti.end(); ++it) {
            if (it->titlu == titlu) {
                carti.erase(it);
                std::cout << "Cartea a fost stearsa cu succes!" << std::endl;
                return;
            }
        }
        std::cout << "Cartea nu a fost gasita!" << std::endl;
    }

    void afiseazaCarti() const {
        std::cout << "Cartile din biblioteca:" << std::endl;
        for (const auto& carte : carti) {
            carte.afiseazaCarte();
        }
    }

    void adaugaUtilizator(const Utilizator& utilizator) {
        utilizatori.push_back(utilizator);
        std::cout << "Utilizatorul a fost adaugat cu succes!" << std::endl;
    }

    Utilizator* gasesteUtilizator(const std::string& nume) {
        for (auto& utilizator : utilizatori) {
            if (utilizator.nume == nume) {
```

```
            return &utilizator;
        }
    }
    return nullptr;
}

Carte* gasesteCarte(const std::string& titlu) {
    for (auto& carte : carti) {
        if (carte.titlu == titlu) {
            return &carte;
        }
    }
    return nullptr;
}
};
```

4. Implementarea în funcția main:

```
int main() {
    Biblioteca biblioteca;

    // Adaugă cărți în bibliotecă
    biblioteca.adaugaCarte(Carte("1984", "George Orwell", 1949));
    biblioteca.adaugaCarte(Carte("To Kill a Mockingbird", "Harper Lee",
1960));

    // Adaugă utilizatori
    biblioteca.adaugaUtilizator(Utilizator("Alice"));
    biblioteca.adaugaUtilizator(Utilizator("Bob"));

    // Afișează cărțile din bibliotecă
    biblioteca.afiseazaCarti();

    // Imprumută o carte
    Utilizator* alice = biblioteca.gasesteUtilizator("Alice");
    Carte* carte = biblioteca.gasesteCarte("1984");

    if (alice && carte) {
        alice->imprumutaCarte(carte);
    }
```

```
    // Afișează cărțile imprumutate de Alice
    if (alice) {
        alice->afiseazaCartiImprumutate();
    }

    // Returnează o carte
    if (alice && carte) {
        alice->returneazaCarte(carte);
    }

    // Afișează cărțile din bibliotecă după returnare
    biblioteca.afiseazaCarti();

    return 0;
}
```

Explicații suplimentare:

- **Clasa Carte:** Reprezintă o carte cu titlu, autor, an de publicare și disponibilitate.
- **Clasa Utilizator:** Reprezintă un utilizator al bibliotecii care poate împrumuta și returna cărți. Lista cartiImprumutate păstrează cărțile împrumutate de utilizator.
- **Clasa Biblioteca:** Conține liste de cărți și utilizatori. Metodele permit adăugarea, ștergerea și afișarea cărților, precum și gestionarea utilizatorilor.
- **Funcția main:** Exemplifică cum se pot folosi aceste clase pentru a crea o aplicație simplă de gestionare a bibliotecii.

Acest proiect de bază poate fi extins cu funcționalități suplimentare, cum ar fi căutarea cărților după autor, gestionarea rezervărilor, istoricul împrumuturilor etc.

Să extindem proiectul de gestionare a bibliotecii pentru a include următoarele funcționalități suplimentare:

1. **Căutarea cărților după autor**
2. **Gestionarea rezervărilor**
3. **Istoricul împrumuturilor**

1. Căutarea cărților după autor:

Adăugăm o metodă în clasa Biblioteca pentru a căuta cărțile după autor.

Actualizare clasa Biblioteca:

```
#include <vector>
#include <algorithm>
class Biblioteca {public:
    // ... alte metode ...

    void cautaCartiDupaAutor(const std::string& autor) const {
        std::cout << "Carti scrise de " << autor << ":" << std::endl;
        for (const auto& carte : carti) {
            if (carte.autor == autor) {
                carte.afiseazaCarte();
            }
        }
    }
};
```

Utilizare în funcția main:

```
int main() {
    Biblioteca biblioteca;
    // ... alte inițializări ...

    // Căutarea cărților după autor
    biblioteca.cautaCartiDupaAutor("George Orwell");

    return 0;
}
```

2. Gestionarea rezervărilor:

Adăugăm funcționalitatea de rezervare a cărților. Vom modifica clasa Carte pentru a include o stare de rezervare și vom adăuga funcționalitate în Utilizator și Biblioteca pentru a gestiona rezervările.

Actualizare clasa Carte:

```
class Carte {
public:
```

```
    // ... alte membri ...

    bool esteRezervata;

    Carte(std::string t, std::string a, int an)
        : titlu(t), autor(a), anPublicatie(an), esteDisponibila(true),
esteRezervata(false) {}

    void afiseazaCarte() const {
        std::cout << "Titlu: " << titlu << ", Autor: " << autor
                  << ", Anul Publicatiei: " << anPublicatie
                  << ", Disponibilitate: " << (esteDisponibila ? "Disponibila" :
"Imprumutata")
                  << ", Rezervare: " << (esteRezervata ? "Rezervata" : "Ne-
rezervata") << std::endl;
    }
};
```

Actualizare clasa Utilizator:

```
class Utilizator {
public:
    // ... alte membri ...

    void rezervaCarte(Carte* carte) {
        if (carte->esteDisponibila && !carte->esteRezervata) {
            carte->esteRezervata = true;
            std::cout << "Cartea a fost rezervata cu succes!" << std::endl;
        } else if (carte->esteRezervata) {
            std::cout << "Cartea este deja rezervata!" << std::endl;
        } else {
            std::cout << "Cartea nu este disponibila pentru rezervare!" <<
std::endl;
        }
    }

    void anuleazaRezervare(Carte* carte) {
        if (carte->esteRezervata) {
            carte->esteRezervata = false;
            std::cout << "Rezervarea pentru carte a fost anulata cu
succes!" << std::endl;
```

```
        } else {
            std::cout << "Cartea nu era rezervata!" << std::endl;
        }
    }
};
```

Actualizare clasa Biblioteca:

```
class Biblioteca {
public:
    // ... alte metode ...

    void rezervaCarte(const std::string& titlu, Utilizator* utilizator) {
        Carte* carte = gasesteCarte(titlu);
        if (carte && utilizator) {
            utilizator->rezervaCarte(carte);
        } else {
            std::cout << "Cartea sau utilizatorul nu a fost gasit!" <<
std::endl;
        }
    }

    void anuleazaRezervare(const std::string& titlu, Utilizator*
utilizator) {
        Carte* carte = gasesteCarte(titlu);
        if (carte && utilizator) {
            utilizator->anuleazaRezervare(carte);
        } else {
            std::cout << "Cartea sau utilizatorul nu a fost gasit!" <<
std::endl;
        }
    }
};
```

Utilizare în funcția main:

```
int main() {
    Biblioteca biblioteca;
    // ... alte inițializări ...

    Utilizator* alice = biblioteca.gasesteUtilizator("Alice");
```

```
    // Rezervare și anulare rezervare
    biblioteca.rezervaCarte("1984", alice);
    biblioteca.anuleazaRezervare("1984", alice);

    return 0;
}
```

3. Istoricul împrumuturilor:

Pentru a menține un istoric al împrumuturilor, adăugăm o structură de date pentru a stoca istoricul și funcționalitatea pentru a vizualiza acest istoric.

Actualizare clasa Utilizator:

```
#include <string>
#include <map>
class Utilizator {public:
    // ... alte membri ...

    std::map<std::string, std::string> istoricImprumuturi; // <titluCarte, dataImprumut>

    void imprumutaCarte(Carte* carte) {
        if (carte->esteDisponibila) {
            cartiImprumutate.push_back(carte);
            carte->esteDisponibila = false;
            istoricImprumuturi[carte->titlu] = "DataImprumut"; // Înlocuiți cu data curentă
            std::cout << "Cartea a fost imprumutata cu succes!" << std::endl;
        } else {
            std::cout << "Cartea nu este disponibila!" << std::endl;
        }
    }

    void afiseazaIstoricImprumuturi() const {
        std::cout << "Istoric imprumuturi pentru " << nume << ":" << std::endl;
        for (const auto& [titlu, data] : istoricImprumuturi) {
```

```
            std::cout << "Titlu: " << titlu << ", Data: " << data << std::endl;
        }
    }
};
```

Utilizare în funcția main:

```
int main() {
    Biblioteca biblioteca;
    // ... alte inițializări ...

    // Afișarea istoricului de împrumuturi
    if (alice) {
        alice->afiseazaIstoricImprumuturi();
    }

    return 0;
}
```

Aceste extensii adaugă funcționalități importante:

1. **Căutarea cărților după autor** permite utilizatorilor să găsească cărțile pe baza autorului.
2. **Gestionarea rezervărilor** permite utilizatorilor să rezerve și să anuleze rezervarea cărților, îmbunătățind gestionarea disponibilității cărților.
3. **Istoricul împrumuturilor** oferă o metodă de a urmări ce cărți au fost împrumutate de către utilizatori și când.

Acest proiect poate fi extins și mai departe cu funcționalități suplimentare, cum ar fi gestionarea amenzilor pentru întârziere, interfețe grafice, sau salvarea și încărcarea datelor într-un fișier sau bază de date.

Să explorăm cum putem extinde proiectul de gestionare a bibliotecii pentru a include:

1. **Gestionarea amenzilor pentru întârziere**
2. **Interfețe grafice**
3. **Salvarea și încărcarea datelor într-un fișier sau bază de date**

1. Gestionarea amenzilor pentru întârziere:

Pentru a gestiona amenzile pentru întârziere, trebuie să adăugăm câteva funcționalități suplimentare:

- **Calcularea amenzilor pe baza întârzierei cărților.**
- **Întârzierea trebuie să fie înregistrată la returnarea cărții.**

Actualizare clasa Utilizator:

Adăugăm o metodă pentru a calcula amenzile și o variabilă pentru a ține evidența amenzilor.

```
#include <chrono>
#include <ctime>
class Utilizator {public:
   // ... alte membri ...

   double calculareAmenzi(int zileIntarziere) const {
      const double taxaPeZi = 1.0; // Taxa pe zi de întârziere
      return zileIntarziere * taxaPeZi;
   }

   void returneazaCarte(Carte* carte, int zileIntarziere = 0) {
      for (auto it = cartiImprumutate.begin(); it !=
cartiImprumutate.end(); ++it) {
         if (*it == carte) {
            cartiImprumutate.erase(it);
            carte->esteDisponibila = true;

            if (zileIntarziere > 0) {
               double amenda = calculareAmenzi(zileIntarziere);
               std::cout << "A întârziat " << zileIntarziere << " zile.
Amenda este: " << amenda << " lei." << std::endl;
            }
            return;
         }
      }
      std::cout << "Utilizatorul nu a imprumutat aceasta carte!" <<
std::endl;
   }
};
```

2. Interfețe grafice:

Pentru a adăuga o interfață grafică, putem folosi o bibliotecă de GUI (Graphic User Interface) precum Qt sau SFML. Voi exemplifica utilizarea Qt, care este o bibliotecă populară pentru dezvoltarea de aplicații cu interfață grafică.

Exemplu simplu de interfață grafică cu Qt:

1.Instalare Qt: Asigură-te că ai Qt instalat. Poți descărca Qt de pe site-ul oficial Qt.

2.Configurare proiect Qt:

Creăm un proiect simplu de aplicație Qt în Qt Creator și adăugăm următorul cod pentru a crea o fereastră simplă.

Codul pentru fereastra principală (main.cpp):

```
#include <QApplication>
#include <QPushButton>
#include <QWidget>
#include <QVBoxLayout>
class MainWindow : public QWidget {public:
    MainWindow() {
        QVBoxLayout* layout = new QVBoxLayout;

        QPushButton* button = new QPushButton("Click me!");
        layout->addWidget(button);

        setLayout(layout);
        setWindowTitle("Biblioteca GUI");
    }
};
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    MainWindow window;
    window.show();

    return app.exec();
}
```

CMakeLists.txt pentru Qt:

```
cmake_minimum_required(VERSION 3.5)

project(BibliotecaGUI)

set(CMAKE_CXX_STANDARD 11)

find_package(Qt5Widgets REQUIRED)

add_executable(BibliotecaGUI main.cpp)
target_link_libraries(BibliotecaGUI Qt5::Widgets)
```

Construirea aplicației:

Folosește Qt Creator sau comenzi de terminal pentru a construi și rula aplicația.

Salvarea și încărcarea datelor într-un fișier sau bază de date:

Salvarea și încărcarea datelor într-un fișier:

Pentru a salva și încărca datele într-un fișier, putem folosi fișiere text sau binare.

Iată un exemplu simplu folosind fișiere text:

Salvarea cărților într-un fișier (salvare.cpp):

```
#include <fstream>
#include <vector>
void salveazaCarti(const std::vector<Carte>& carti) {
    std::ofstream fout("carti.txt");

    for (const auto& carte : carti) {
        fout << carte.titlu << "," << carte.autor << "," <<
carte.anPublicatie << "," << carte.esteDisponibila << "\n";
    }

    fout.close();
}
```

Încărcarea cărților dintr-un fișier (incarcare.cpp):

```
#include <fstream>
#include <sstream>
void incarcaCarti(std::vector<Carte>& carti) {
    std::ifstream fin("carti.txt");
    std::string linie;

    while (std::getline(fin, linie)) {
        std::stringstream ss(linie);
        std::string titlu, autor;
        int an;
        bool disponibil;

        std::getline(ss, titlu, ',');
        std::getline(ss, autor, ',');
        ss >> an;
        ss.ignore(); // Ignorăm caracterul ',' pentru bool
        ss >> disponibil;

        carti.emplace_back(titlu, autor, an);
        carti.back().esteDisponibila = disponibil;
    }

    fin.close();
}
```

Salvarea și încărcarea utilizatorilor ar funcționa similar, folosind metode adecvate pentru formatele de date și structurile de date.

Utilizarea bazelor de date:

Pentru a utiliza o bază de date, puteți folosi SQLite, MySQL sau PostgreSQL.

Iată un exemplu simplu folosind SQLite:

Instalare SQLite: Asigură-te că ai SQLite instalat și configurat pentru a-l folosi cu C++.

Exemplu simplu de utilizare SQLite:

Cod de conectare și interogare SQLite (sqlite_example.cpp):

```
#include <iostream>
#include <sqlite3.h>
int main() {
   sqlite3* db;
   char* errMsg = 0;

   int rc = sqlite3_open("biblioteca.db", &db);
   if (rc) {
      std::cerr << "Nu am putut deschide baza de date: " <<
sqlite3_errmsg(db) << std::endl;
      return rc;
   }

   const char* sql = "CREATE TABLE IF NOT EXISTS carti (id
INTEGER PRIMARY KEY, titlu TEXT, autor TEXT, anPublicatie
INTEGER, disponibil BOOLEAN);";
   rc = sqlite3_exec(db, sql, 0, 0, &errMsg);
   if (rc != SQLITE_OK) {
      std::cerr << "Eroare SQL: " << errMsg << std::endl;
      sqlite3_free(errMsg);
   }

   sqlite3_close(db);
   return 0;
}
```

Acest exemplu creează o tabelă pentru cărți.

Funcționalitățile pentru inserarea, actualizarea și interogarea bazei de date pot fi adăugate în mod similar.

Concluzie:

Aceste extensii fac ca sistemul de gestionare a bibliotecii să fie mai robust și să ofere funcționalități mai avansate.

Implementarea unui sistem de amenzi, crearea de interfețe grafice și integrarea unei baze de date sunt pași importanți pentru a construi aplicații software mai complexe și mai utile.

Aceste adăugiri vor face aplicația mai completă și mai profesionistă.