MVP from Julio Veganos e Hijos (User Managment)

Generated by Doxygen 1.9.8

Chapter 1

User Database Management System

1.1 Introduction

The User Database Management System is a C++ application that demonstrates fundamental database operations, user authentication, and exception handling. This project is designed to showcase proper software engineering practices including object-oriented design, error handling, and documentation.

1.2 Key Features

- **User Management**: Create, read, update, and delete (CRUD) operations
- **Authentication**: Secure login process with role-based access control
- **Persistence**: Binary file storage with random access capabilities
- **Exception Handling**: Custom exception classes with informative error messages
- **Role-Based Access Control**: Different permissions for Admin and Employee users

1.3 Main Components

The system consists of several key components:

- **User Classes**: Base User class with Admin and Employee derived classes
- **UserDatabase**: Core class managing user storage and retrieval
- **UserFactory**: Factory pattern implementation for creating and deserializing users
- **Exception Classes**: Custom exception hierarchy for robust error handling

1.4 Testing Interface

The application provides a comprehensive testing interface through main.cpp that allows:

- Adding, updating, and removing users
- Testing user authentication
- Changing passwords with proper authorization
- Clearing the database (with Administrator privileges)
- Testing random access to binary database records
- Demonstrating exception handling capabilities

1.5 Basic Usage

```
// Create a UserDatabase instance
UserDatabase db("users");

// Add a new admin user
Admin* admin = new Admin(10000, "A12345678", "password", User::ADMIN);
db.addUser(admin);

// Authenticate a user
User* user = db.login(10000, "A12345678", "password");
if (user) {
    cout « "Login successful!" « endl;
}
```

1.6 Requirements

- C++11 compliant compiler
- Standard C++ libraries
- Doxygen (for generating documentation)

Author

Carlos Nebril

Date

April 25, 2025

Chapter 2

File Index

2.1 File List

Here is a list of all files with brief descriptions:

Chapter 3

File Documentation

3.1 main.cpp File Reference

User Database Testing Program.

```
#include <iostream>
#include <fstream>
#include <string>
#include <set>
#include <limits>
#include "../UserDatabase.h"
#include "../../Users/User.h"
#include "../../Users/Admin.h"
#include "../../Users/Employee.h"
#include "../../Users/UserFactory.h"
#include "../Exceptions/UserDatabaseException.h"
```

Include dependency graph for main.cpp:

Functions

- void displayMenu ()

  Displays the main menu options to the user.
- int getMenuChoice ()

  Gets a validated menu choice from the user.
- void displayUserList (const UserDatabase &db)

  Displays a list of all users in the database.
- void testAddUser (UserDatabase &db)

  Test function for adding new users to the database.
- void testUpdateUser (UserDatabase &db)

  Test function for updating existing users in the database.
- void testRemoveUser (UserDatabase &db)

Test function for removing users from the database.

- void testLogin (UserDatabase &db)

  Test function for user login verification.
- void testPasswordChange (UserDatabase &db)

  Tests the password change functionality with exception re-throwing.
- void testClearFile (UserDatabase &db, const string &filename)

  Test function for clearing the database file.
- void testRandomAccess (const string &filename)

  Test function for random access to user records in binary file.
- void testBadAllocException (UserDatabase &db)

  Test function to demonstrate handling of std::bad_alloc exceptions.
- void displayUserDetails (const User *user)

  Displays detailed information about a user.
- void pauseExecution ()

  Pauses program execution until user presses Enter.
- int main ()

  Main entry point for the UserDatabase test program.

3.1.1 Detailed Description

User Database Testing Program.

This program provides a command-line interface to test various features of the UserDatabase class, including CRUD operations, authentication, and exception handling.

Author

Carlos Nebril

Date

April 25, 2025

Definition in file main.cpp.

3.1.2 Function Documentation

3.1.2.1 displayMenu()

```
void displayMenu ( )
```

Displays the main menu options to the user.

Shows a formatted menu with numbered options for all available operations in the program. Options include user management, authentication testing, and file operations.

Definition at line 153 of file main.cpp.

```
00153                     {
00154     cout « "\n=== Menu Options ===" « endl;
00155     cout « "1. Display all users" « endl;
00156     cout « "2. Add a new user" « endl;
00157     cout « "3. Update an existing user" « endl;
```

```
00158     cout « "4. Remove a user" « endl;
00159     cout « "5. Test login" « endl;
00160     cout « "6. Change a user's password" « endl;
00161     cout « "7. Clear database file" « endl;
00162     cout « "8. Test random access to user records" « endl;
00163     cout « "9. Test bad_alloc exception handling" « endl;
00164     cout « "0. Exit" « endl;
00165 }
```

Referenced by main().

Here is the caller graph for this function:

3.1.2.2 displayUserDetails()

```
void displayUserDetails (
            const User * user )
```

Displays detailed information about a user.

Shows user number, NIF, masked password, and role. Handles null user pointer gracefully by showing an error message.

Parameters

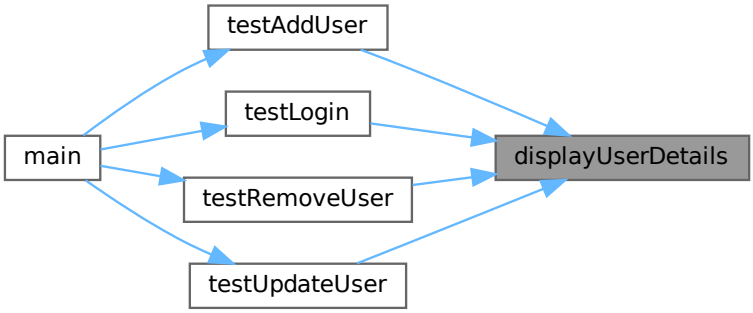
user	Pointer to the User object to display

Definition at line 727 of file main.cpp.

```
00727                                       {
00728     if (!user) {
00729         cout « "No user to display." « endl;
00730         return;
00731     }
00732
00733     cout « "----------------------" « endl;
00734     cout « "User Number: " « user->getUsrNumber() « endl;
00735     cout « "NIF: " « user->getNIF() « endl;
00736     cout « "Password: " « string(strlen(user->getPwd().c_str()), '*') « endl;
00737     cout « "Role: " « (user->getRole() == User::ADMIN ? "ADMIN" : "EMPLOYEE");
00738     cout « "\n----------------------" « endl;
00739 }
```

Referenced by testAddUser(), testLogin(), testRemoveUser(), and testUpdateUser().

Here is the caller graph for this function:



3.1.2.3 displayUserList()

```
void displayUserList (
            const UserDatabase & db )
```

Displays a list of all users in the database.

Retrieves all users from the database and displays their details. If no users are found, displays an appropriate message.

Parameters

db	Reference to the user database to query

Definition at line 234 of file main.cpp.

```
00234                                           {
00235     cout « "\n=== User List ===" « endl;
00236
00237     set<User*> users = db.getAllUsers();
00238     if (users.empty()) {
00239         cout « "No users found in the database." « endl;
00240         return;
00241     }
00242
00243     cout « "Number of users: " « users.size() « endl;
00244     cout « "-----------------------" « endl;
00245
00246     for (const auto& user : users) {
00247         cout « *user « endl;
00248         cout « "-----------------------" « endl;
00249     }
00250 }
```

Referenced by main().

Here is the caller graph for this function:

3.1.2.4 getMenuChoice()

```
int getMenuChoice ( )
```

Gets a validated menu choice from the user.

Ensures the input is a single digit between 0-9, handling various invalid input scenarios with appropriate error messages. Uses string-based input processing to avoid buffer issues and provides robust error handling.

Returns

int The validated menu choice (0-9)

Definition at line 176 of file main.cpp.

```
00176                     {
00177     int choice;
00178     bool validInput = false;
00179     string input;
00180
00181     while (!validInput) {
00182         cout « "Enter your choice (0-9): ";
00183
00184         // Read entire line as string
00185         getline(cin, input);
00186
00187         // Check if input contains only digits
00188         bool onlyDigits = true;
00189         for (char c : input) {
00190             if (!isdigit(c)) {
00191                 onlyDigits = false;
00192                 break;
00193             }
00194         }
00195
00196         // Input should be exactly one digit between 0-9
00197         if (!onlyDigits || input.empty()) {
00198             cout « "Invalid input. Please enter a number between 0 and 9."
00199                  « endl;
00200             continue;
00201         }
00202
00203         // Convert string to integer
00204         try {
00205             choice = stoi(input);
00206
00207             // Check if number is in valid range
00208             if (choice < 0 || choice > 9) {
00209                 cout « "Invalid choice. Please enter a number between 0 and 9."
00210                      « endl;
00211                 continue;
00212             }
00213
00214             // If we get here, the input is valid
00215             validInput = true;
00216         }
00217         catch (const exception&) {
00218             cout « "Invalid input. Please enter a number between 0 and 9."
```

```
00219                   « endl;
00220         }
00221     }
00222
00223     return choice;
00224 }
```

Referenced by main().

Here is the caller graph for this function:

3.1.2.5 main()

```
int main ( )
```

Main entry point for the UserDatabase test program.

Initializes a test database file and provides a menu-driven interface for testing various database operations. The program demonstrates:

- User management (add, update, remove)
- Authentication (login)
- Password management
- Exception handling
- File operations

The database is automatically saved when the program exits.

Returns

int Exit status code (0 for successful execution)

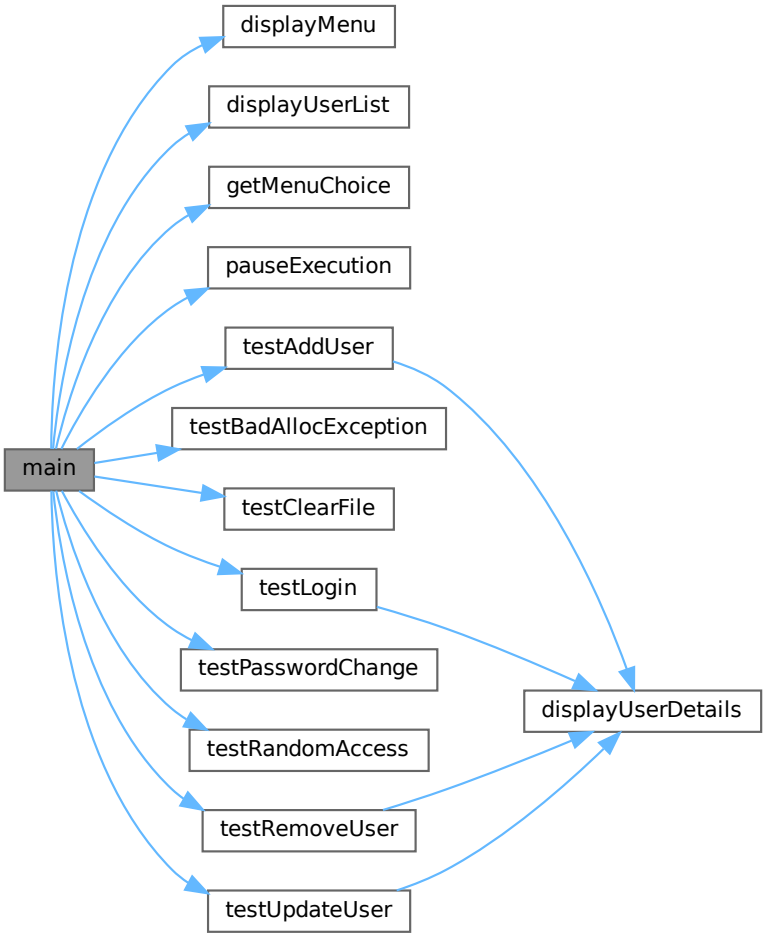
Definition at line 65 of file main.cpp.

```
00065           {
00066     cout « "=== UserDatabase Testing Program ===" « endl;
00067
00068     // Create a test database file
00069     const char* testFile = "test_users";
00070     UserDatabase db(testFile);
00071
00072     cout « "UserDatabase initialized with file: "
00073          « testFile « ".dat" « endl;
00074
00075     int choice = 0;
00076     bool exitProgram = false;
00077
00078     // Main program loop
00079     while (!exitProgram) {
```

```
00080         displayMenu();
00081         choice = getMenuChoice();
00082 
00083         switch (choice) {
00084             case 1:
00085                 displayUserList(db);
00086                 break;
00087 
00088             case 2:
00089                 testAddUser(db);
00090                 break;
00091 
00092             case 3:
00093                 testUpdateUser(db);
00094                 break;
00095 
00096             case 4:
00097                 testRemoveUser(db);
00098                 break;
00099 
00100             case 5:
00101                 testLogin(db);
00102                 break;
00103 
00104             case 6:
00105                 try {
00106                     testPasswordChange(db);
00107                 }
00108                 catch (const exception& e) {
00109                     // This will only be reached if an exception escapes testPasswordChange
00110                     cout << "Unhandled exception in password change operation: " << e.what() << endl;
00111                 }
00112                 break;
00113 
00114             case 7:
00115                 testClearFile(db, string(testFile) + ".dat");
00116                 break;
00117 
00118             case 8:
00119                 testRandomAccess(string(testFile) + ".dat");
00120                 break;
00121 
00122             case 9:
00123                 testBadAllocException(db);  // New test function
00124                 break;
00125 
00126             case 0:
00127                 exitProgram = true;
00128                 cout << "Exiting program. Database will be saved." << endl;
00129                 break;
00130 
00131             default:
00132                 cout << "Invalid choice. Please try again." << endl;
00133                 break;
00134         }
00135 
00136         if (!exitProgram) {
00137             pauseExecution();
00138         }
00139     }
00140 
00141     return 0;
00142 }
```

References displayMenu(), displayUserList(), getMenuChoice(), pauseExecution(), testAddUser(), testBadAllocException(), testClearFile(), testLogin(), testPasswordChange(), testRandomAccess(), testRemoveUser(), and testUpdateUser().

Here is the call graph for this function:



3.1.2.6 pauseExecution()

```
void pauseExecution ( )
```

Pauses program execution until user presses Enter.

Provides a clean way to pause the program between operations, allowing the user to view results before continuing. Handles input buffer clearing to ensure consistent behavior regardless of previous input operations.

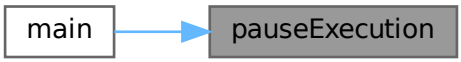
Definition at line 749 of file main.cpp.

```
00749                       {
00750     cout « "\nPress Enter to continue...";
00751
00752     // Clean input buffer and wait for Enter
00753     if (cin.peek() == '\n') {
00754         // If there's a pending newline character, consume it
00755         cin.get();
00756     } else {
00757         // If there's no pending newline or other characters in buffer,
00758         // clear the entire buffer up to the next newline
00759         cin.ignore(numeric_limits<streamsize>::max(), '\n');
```

```
00760     }
00761 }
```

Referenced by main().

Here is the caller graph for this function:



3.1.2.7 testAddUser()

```
void testAddUser (
            UserDatabase & db )
```

Test function for adding new users to the database.

Prompts for user information (number, NIF, password, role), creates the appropriate User object, and attempts to add it to the database. Handles potential exceptions during user creation.

Parameters

db	Reference to the user database

Definition at line 261 of file main.cpp.

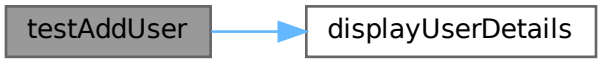
```
00261                                       {
00262     cout « "\n=== Add New User ===" « endl;
00263
00264     u_int32_t usrNumber;
00265     char NIF[User::MAX_NIF];
00266     char pwd[User::MAX_STR];
00267     int roleChoice;
00268
00269     // Get user input for new user
00270     cout « "Enter user number (10000-99999): ";
00271     cin » usrNumber;
00272
00273     cout « "Enter NIF (up to " « User::MAX_NIF - 1 « " characters): ";
00274     cin » NIF;
00275
00276     cout « "Enter password: ";
00277     cin » pwd;
00278
00279     cout « "Select role (0 for ADMIN, 1 for EMPLOYEE): ";
00280     cin » roleChoice;
00281
00282     User* newUser = nullptr;
00283
00284     try {
00285         // Create appropriate user type
00286         if (roleChoice == 0) {
00287             newUser = new Admin(usrNumber, NIF, pwd, User::ADMIN);
00288         } else {
00289             newUser = new Employee(usrNumber, NIF, pwd, User::EMPLOYEE);
00290         }
00291
00292         // Try to add user to database
00293         if (db.addUser(newUser)) {
```

```
00294             cout « "User added successfully!" « endl;
00295             displayUserDetails(newUser);
00296         } else {
00297             cout « "Failed to add user. User with number "
00298                  « usrNumber « " may already exist." « endl;
00299             delete newUser; // Clean up if not added to database
00300         }
00301     } catch (const exception& e) {
00302         cout « "Error creating user: " « e.what() « endl;
00303         if (newUser) delete newUser;
00304     }
00305 }
```

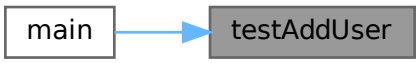
References displayUserDetails().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:



3.1.2.8 testBadAllocException()

```
void testBadAllocException (
            UserDatabase & db )
```

Test function to demonstrate handling of std::bad_alloc exceptions.

This function deliberately attempts to allocate an excessive amount of memory to trigger a std::bad_alloc exception, then shows proper exception handling.

Parameters

db	Reference to the user database (not used but kept for consistency)

Definition at line 686 of file main.cpp.

```
00686                                           {
00687     cout « "\n=== Testing std::bad_alloc Exception Handling ===" « endl;
00688
00689     // Size that will likely trigger a bad_alloc exception on most systems
00690     // Using size_t max value effectively requests all available memory
00691     const size_t HUGE_SIZE = std::numeric_limits<size_t>::max() / 10;
00692
00693     try {
00694         cout « "Attempting to allocate " « HUGE_SIZE
00695              « " bytes of memory..." « endl;
00696
00697         // This will almost certainly fail and throw std::bad_alloc
00698         int* hugeArray = new int[HUGE_SIZE];
00699
00700         // This code should never execute if the allocation fails as expected
00701         cout « "Allocation successful (unexpected!)" « endl;
00702         delete[] hugeArray;  // Clean up if allocation somehow succeeds
00703     }
00704     catch (const std::bad_alloc& e) {
00705         // Specific handler for memory allocation failure
00706         cout « "Memory allocation failed as expected!" « endl;
00707         cout « "Exception details: " « e.what() « endl;
00708         cout « "This demonstrates proper handling of std::bad_alloc" « endl;
00709     }
00710     catch (const std::exception& e) {
00711         // Generic handler for other exceptions
00712         cout « "Unexpected exception: " « e.what() « endl;
00713     }
00714
00715     // Parameter is not used but kept for consistency with other test functions
00716     (void)db;
00717 }
```

Referenced by main().

Here is the caller graph for this function:

3.1.2.9 testClearFile()

```
void testClearFile (
            UserDatabase & db,
            const string & filename )
```

Test function for clearing the database file.

Prompts for confirmation and then removes all users from the database file, except for the default administrator. Handles potential exceptions during the clear operation.

Parameters

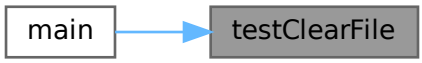
db	Reference to the user database
filename	The name of the database file to clear

Definition at line 570 of file main.cpp.

```
00570                                                       {
00571     cout « "\n=== Clear Database File ===" « endl;
00572
00573     char confirm;
00574     cout « "WARNING: This will delete all users from the file." « endl;
00575     cout « "Are you sure? (y/n): ";
00576     cin » confirm;
00577
00578     if (tolower(confirm) == 'y') {
00579         try {
00580             if (db.clearFile(filename.c_str())) {
00581                 cout « "Database file cleared successfully." « endl;
00582             } else {
00583                 cout « "Failed to clear database file." « endl;
00584             }
00585         } catch (const runtime_error& e) {
00586             cout « "Error: " « e.what() « endl;
00587         }
00588     } else {
00589         cout « "Operation cancelled." « endl;
00590     }
00591 }
```

Referenced by main().

Here is the caller graph for this function:



3.1.2.10 testLogin()

```
void testLogin (
            UserDatabase & db )
```

Test function for user login verification.

Prompts for login credentials (user number, NIF, password) and attempts to authenticate the user. Displays success or failure message and shows user details if login is successful.

Parameters

db	Reference to the user database

Definition at line 447 of file main.cpp.

```
00447                                   {
00448     cout « "\n=== User Login ===" « endl;
00449
00450     u_int32_t usrNumber;
00451     char NIF[User::MAX_NIF];
00452     char pwd[User::MAX_STR];
00453
00454     cout « "Enter user number: ";
00455     cin » usrNumber;
00456
00457     cout « "Enter NIF: ";
00458     cin » NIF;
00459
```

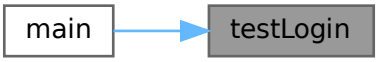
```
00460     cout « "Enter password: ";
00461     cin » pwd;
00462
00463     // Attempt login
00464     string passwordStr(pwd);
00465     User* loggedInUser = db.login(usrNumber, NIF, passwordStr);
00466
00467     if (loggedInUser) {
00468         cout « "Login successful!" « endl;
00469         cout « "Welcome, "
00470              « (loggedInUser->getRole() == User::ADMIN
00471                  ? "Administrator" : "Employee")
00472              « " #" « loggedInUser->getUsrNumber() « endl;
00473         displayUserDetails(loggedInUser);
00474     } else {
00475         cout « "Login failed. Invalid credentials." « endl;
00476     }
00477 }
```

References displayUserDetails().

Referenced by main().

Here is the call graph for this function:

Here is the caller graph for this function:



3.1.2.11 testPasswordChange()

```
void testPasswordChange (
            UserDatabase & db )
```

Tests the password change functionality with exception re-throwing.

This function demonstrates exception re-throwing by:

1. Catching authentication exceptions
2. Logging information about the error
3. Re-throwing the exception to be handled at a higher level

Parameters

db	Reference to the user database

Definition at line 489 of file main.cpp.

```
00489                                           {
00490     cout « "\n=== Change User Password ===" « endl;
00491
00492     // First, login to get an active user
00493     u_int32_t activeUsrNumber;
00494     char activeNIF[User::MAX_NIF];
00495     char activePwd[User::MAX_STR];
00496
00497     try {
00498         cout « "First, please login:" « endl;
00499         cout « "Enter your user number: ";
00500         cin » activeUsrNumber;
00501
00502         cout « "Enter your NIF: ";
00503         cin » activeNIF;
00504
00505         cout « "Enter your password: ";
00506         cin » activePwd;
00507
00508         // Attempt login
00509         string passwordStr(activePwd);
00510         User* activeUser = nullptr;
00511
00512         try {
00513             activeUser = db.login(activeUsrNumber, activeNIF, passwordStr);
00514
00515             if (!activeUser) {
00516                 throw runtime_error("Login failed: Invalid credentials");
00517             }
00518
00519             // Get the user whose password will be changed
00520             u_int32_t targetUsrNumber;
00521             cout « "Enter the user number whose password you want to change: ";
00522             cin » targetUsrNumber;
00523
00524             User* targetUser = db.findUserByNumber(targetUsrNumber);
00525             if (!targetUser) {
00526                 throw runtime_error("User not found");
00527             }
00528
00529             // Check if active user has permission
00530             if (activeUser->getRole() != User::ADMIN &&
00531                 activeUser->getUsrNumber() != targetUser->getUsrNumber()) {
00532                 throw runtime_error("Permission denied");
00533             }
00534
00535             // Attempt password change
00536             if (db.changeUserPass(activeUser, targetUser)) {
00537                 cout « "Password changed successfully." « endl;
00538             } else {
00539                 throw runtime_error("Password change operation failed");
00540             }
00541         }
00542         catch (const runtime_error& e) {
00543             // Log the error details
00544             cout « "Authentication error: " « e.what() « endl;
00545             cout « "User attempted: " « activeUsrNumber « endl;
00546
00547             // Example of re-throwing the same exception
00548             // This will be caught by the outer try-catch block
00549             cout « "Re-throwing exception..." « endl;
00550             throw;  // Re-throw the current exception
00551         }
00552     }
00553     catch (const exception& e) {
00554         // Handle the re-thrown exception
00555         cout « "Operation aborted: " « e.what() « endl;
00556         cout « "Please try again with valid credentials." « endl;
00557     }
00558 }
```

Referenced by main().

Here is the caller graph for this function:

3.1.2.12 testRandomAccess()

```
void testRandomAccess (
            const string & filename )
```

Test function for random access to user records in binary file.

Opens the database file in binary mode and allows direct access to user records by index. Shows file statistics including the number of records stored. Demonstrates low-level binary file operations.

Parameters

filename	The name of the database file to access

Definition at line 602 of file main.cpp.

```
00602                                                {
00603     cout « "\n=== Testing Random Binary Access ===" « endl;
00604
00605     // Open the binary file for reading
00606     ifstream file(filename, ios::in | ios::binary);
00607     if (!file.is_open()) {
00608         cout « "Could not open file " « filename
00609              « " for random access test." « endl;
00610         return;
00611     }
00612
00613     // Calculate file size
00614     file.seekg(0, ios::end);
00615     streampos fileSize = file.tellg();
00616     int numRecords = fileSize / sizeof(UserRecord);
00617
00618     cout « "File contains " « numRecords « " user records" « endl;
00619
00620     // Check if there are any records
00621     if (numRecords <= 0) {
00622         cout « "No records available for random access." « endl;
00623         file.close();
00624         return;
00625     }
00626
00627     // Display the first record (Admin user)
00628     file.seekg(0, ios::beg);
00629     User* firstUser = UserFactory::readUserFromFile(file);
00630
00631     if (firstUser) {
00632         cout « "First record (default admin):" « endl;
00633         cout « *firstUser « endl;
00634         delete firstUser;
00635     } else {
00636         cout « "Failed to read first record." « endl;
00637     }
00638
00639     // If more records exist, access random records
00640     if (numRecords > 1) {
00641         // Let the user choose a record to access
```

```
00642         int recordIndex;
00643         while (true) {
00644             cout « "Enter a record index to access (0-" « numRecords - 1
00645                  « ", or -1 to exit): ";
00646             cin » recordIndex;
00647 
00648             if (recordIndex == -1) {
00649                 break;
00650             }
00651 
00652             if (recordIndex < 0 || recordIndex >= numRecords) {
00653                 cout « "Invalid record index." « endl;
00654                 continue;
00655             }
00656 
00657             // Seek to the selected record
00658             file.seekg(recordIndex * sizeof(UserRecord), ios::beg);
00659 
00660             // Read the user from this position
00661             User* user = UserFactory::readUserFromFile(file);
00662 
00663             if (user) {
00664                 cout « "User at position " « recordIndex « ":" « endl;
00665                 cout « *user « endl;
00666                 delete user;
00667             } else {
00668                 cout « "Failed to read user at position "
00669                      « recordIndex « endl;
00670             }
00671         }
00672     }
00673 
00674     file.close();
00675 }
```

Referenced by main().

Here is the caller graph for this function:

3.1.2.13 testRemoveUser()

```
void testRemoveUser (
             UserDatabase & db )
```

Test function for removing users from the database.

Demonstrates using and catching custom UserDatabaseException Uses confirmation prompt before removal

Parameters

db	Reference to the user database

Definition at line 395 of file main.cpp.

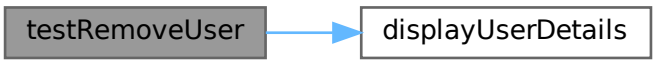
```
00395                                       {
00396     cout « "\n=== Remove User ===" « endl;
00397 
```

```
00398     u_int32_t usrNumber;
00399     cout « "Enter user number to remove: ";
00400     cin » usrNumber;
00401
00402     // Try to find user
00403     User* userToRemove = db.findUserByNumber(usrNumber);
00404     if (!userToRemove) {
00405         cout « "User with number " « usrNumber « " not found." « endl;
00406         return;
00407     }
00408
00409     // Display user info
00410     cout « "User to remove:" « endl;
00411     displayUserDetails(userToRemove);
00412
00413     // Confirm removal
00414     char confirm;
00415     cout « "Are you sure you want to remove this user? (y/n): ";
00416     cin » confirm;
00417
00418     if (tolower(confirm) == 'y') {
00419         try {
00420             if (db.removeUser(userToRemove)) {
00421                 cout « "User removed successfully!" « endl;
00422             } else {
00423                 cout « "Failed to remove user." « endl;
00424             }
00425         } catch (const UserDatabaseException& e) {
00426             cout « "Database Exception: " « e.what() « endl;
00427             cout « "Error type: " « e.getErrorString() « endl;
00428             cout « "Error code: " « e.getErrorCode() « endl;
00429         } catch (const std::exception& e) {
00430             // Fallback for other types of exceptions
00431             cout « "Error: " « e.what() « endl;
00432         }
00433     } else {
00434         cout « "User removal cancelled." « endl;
00435     }
00436 }
```

References displayUserDetails().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:

3.1.2.14 testUpdateUser()

```
void testUpdateUser (
            UserDatabase & db )
```

Test function for updating existing users in the database.

Prompts for a user number, finds the user, and allows updating NIF, password, and/or role. Empty input keeps current values. Creates a new user object with updated information and replaces the existing user in the database.

Parameters

db	Reference to the user database

Definition at line 317 of file main.cpp.

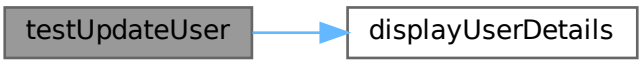
```
00317                                       {
00318     cout « "\n=== Update User ===" « endl;
00319
00320     u_int32_t usrNumber;
00321     cout « "Enter user number to update: ";
00322     cin » usrNumber;
00323
00324     // Try to find user
00325     User* existingUser = db.findUserByNumber(usrNumber);
00326     if (!existingUser) {
00327         cout « "User with number " « usrNumber « " not found." « endl;
00328         return;
00329     }
00330
00331     // Display current user info
00332     cout « "Current user information:" « endl;
00333     displayUserDetails(existingUser);
00334
00335     // Get updated info
00336     char NIF[User::MAX_NIF];
00337     char pwd[User::MAX_STR];
00338     int roleChoice;
00339
00340     cout « "Enter new NIF (or leave blank to keep current): ";
00341     cin.ignore(); // Clear input buffer
00342     cin.getline(NIF, User::MAX_NIF);
00343
00344     cout « "Enter new password (or leave blank to keep current): ";
00345     cin.getline(pwd, User::MAX_STR);
00346
00347     cout « "Select new role (0 for ADMIN, 1 for EMPLOYEE, "
00348          « "or any other number to keep current): ";
00349     cin » roleChoice;
00350
00351     // Create updated user object
00352     User* updatedUser = nullptr;
00353     try {
00354         // Use existing values if fields were left blank
00355         const char* newNIF = (NIF[0] == '\0') ?
00356              existingUser->getNIF() : NIF;
00357         const char* newPwd = (pwd[0] == '\0') ?
00358              existingUser->getPwd().c_str() : pwd;
00359         User::Role newRole = (roleChoice == 0) ? User::ADMIN :
00360                              (roleChoice == 1) ? User::EMPLOYEE :
00361                              existingUser->getRole();
00362
00363         // Create appropriate user type
00364         if (newRole == User::ADMIN) {
00365             updatedUser = new Admin(usrNumber, newNIF, newPwd, User::ADMIN);
00366         } else {
00367             updatedUser = new Employee(usrNumber,
00368                                        newNIF,
00369                                        newPwd,
00370                                        User::EMPLOYEE);
00371         }
00372
00373         // Update the user in the database
00374         if (db.updateUser(updatedUser)) {
00375             cout « "User updated successfully!" « endl;
00376             displayUserDetails(updatedUser);
00377         } else {
00378             cout « "Failed to update user." « endl;
```

```
00379             delete updatedUser;
00380         }
00381     } catch (const exception& e) {
00382         cout « "Error updating user: " « e.what() « endl;
00383         if (updatedUser) delete updatedUser;
00384     }
00385 }
```

References displayUserDetails().

Referenced by main().

Here is the call graph for this function:



Here is the caller graph for this function:

3.2 main.cpp

Go to the documentation of this file.

```
00001 #include <iostream>
00002 #include <fstream>
00003 #include <string>
00004 #include <set>
00005 #include <limits>
00006 #include "../UserDatabase.h"
00007 #include "../../Users/User.h"
00008 #include "../../Users/Admin.h"
00009 #include "../../Users/Employee.h"
00010 #include "../../Users/UserFactory.h"
00011 #include "../Exceptions/UserDatabaseException.h"
00012
00013 using namespace std;
00014
00015 class UserDatabase;
00016 class User;
00017 class Admin;
00018 class Employee;
00019 class UserFactory;
00020 class UserDatabaseException;
00021
00022 // Function declarations
00023 void displayMenu();
00024 int getMenuChoice();
00025 void displayUserList(const UserDatabase& db);
00026 void testAddUser(UserDatabase& db);
```

```
00027 void testUpdateUser(UserDatabase& db);
00028 void testRemoveUser(UserDatabase& db);
00029 void testLogin(UserDatabase& db);
00030 void testPasswordChange(UserDatabase& db);
00031 void testClearFile(UserDatabase& db, const string& filename);
00032 void testRandomAccess(const string& filename);
00033 void testBadAllocException(UserDatabase& db);
00034 void displayUserDetails(const User* user);
00035 void pauseExecution();
00036
00037
00065 int main() {
00066     cout << "=== UserDatabase Testing Program ===" << endl;
00067
00068     // Create a test database file
00069     const char* testFile = "test_users";
00070     UserDatabase db(testFile);
00071
00072     cout << "UserDatabase initialized with file: "
00073          << testFile << ".dat" << endl;
00074
00075     int choice = 0;
00076     bool exitProgram = false;
00077
00078     // Main program loop
00079     while (!exitProgram) {
00080         displayMenu();
00081         choice = getMenuChoice();
00082
00083         switch (choice) {
00084             case 1:
00085                 displayUserList(db);
00086                 break;
00087
00088             case 2:
00089                 testAddUser(db);
00090                 break;
00091
00092             case 3:
00093                 testUpdateUser(db);
00094                 break;
00095
00096             case 4:
00097                 testRemoveUser(db);
00098                 break;
00099
00100             case 5:
00101                 testLogin(db);
00102                 break;
00103
00104             case 6:
00105                 try {
00106                     testPasswordChange(db);
00107                 }
00108                 catch (const exception& e) {
00109                     // This will only be reached if an exception escapes testPasswordChange
00110                     cout << "Unhandled exception in password change operation: " << e.what() << endl;
00111                 }
00112                 break;
00113
00114             case 7:
00115                 testClearFile(db, string(testFile) + ".dat");
00116                 break;
00117
00118             case 8:
00119                 testRandomAccess(string(testFile) + ".dat");
00120                 break;
00121
00122             case 9:
00123                 testBadAllocException(db);  // New test function
00124                 break;
00125
00126             case 0:
00127                 exitProgram = true;
00128                 cout << "Exiting program. Database will be saved." << endl;
00129                 break;
00130
00131             default:
00132                 cout << "Invalid choice. Please try again." << endl;
00133                 break;
00134         }
00135
00136         if (!exitProgram) {
00137             pauseExecution();
00138         }
00139     }
00140
```

```
00141     return 0;
00142 }
00143
00144
00145
00153 void displayMenu() {
00154     cout « "\n=== Menu Options ===" « endl;
00155     cout « "1. Display all users" « endl;
00156     cout « "2. Add a new user" « endl;
00157     cout « "3. Update an existing user" « endl;
00158     cout « "4. Remove a user" « endl;
00159     cout « "5. Test login" « endl;
00160     cout « "6. Change a user's password" « endl;
00161     cout « "7. Clear database file" « endl;
00162     cout « "8. Test random access to user records" « endl;
00163     cout « "9. Test bad_alloc exception handling" « endl;
00164     cout « "0. Exit" « endl;
00165 }
00166
00176 int getMenuChoice() {
00177     int choice;
00178     bool validInput = false;
00179     string input;
00180
00181     while (!validInput) {
00182         cout « "Enter your choice (0-9): ";
00183
00184         // Read entire line as string
00185         getline(cin, input);
00186
00187         // Check if input contains only digits
00188         bool onlyDigits = true;
00189         for (char c : input) {
00190             if (!isdigit(c)) {
00191                 onlyDigits = false;
00192                 break;
00193             }
00194         }
00195
00196         // Input should be exactly one digit between 0-9
00197         if (!onlyDigits || input.empty()) {
00198             cout « "Invalid input. Please enter a number between 0 and 9."
00199                  « endl;
00200             continue;
00201         }
00202
00203         // Convert string to integer
00204         try {
00205             choice = stoi(input);
00206
00207             // Check if number is in valid range
00208             if (choice < 0 || choice > 9) {
00209                 cout « "Invalid choice. Please enter a number between 0 and 9."
00210                      « endl;
00211                 continue;
00212             }
00213
00214             // If we get here, the input is valid
00215             validInput = true;
00216         }
00217         catch (const exception&) {
00218             cout « "Invalid input. Please enter a number between 0 and 9."
00219                  « endl;
00220         }
00221     }
00222
00223     return choice;
00224 }
00225
00234 void displayUserList(const UserDatabase& db) {
00235     cout « "\n=== User List ===" « endl;
00236
00237     set<User*> users = db.getAllUsers();
00238     if (users.empty()) {
00239         cout « "No users found in the database." « endl;
00240         return;
00241     }
00242
00243     cout « "Number of users: " « users.size() « endl;
00244     cout « "------------------------" « endl;
00245
00246     for (const auto& user : users) {
00247         cout « *user « endl;
00248         cout « "------------------------" « endl;
00249     }
00250 }
00251
```

```
00261 void testAddUser(UserDatabase& db) {
00262     cout « "\n=== Add New User ===" « endl;
00263 
00264     u_int32_t usrNumber;
00265     char NIF[User::MAX_NIF];
00266     char pwd[User::MAX_STR];
00267     int roleChoice;
00268 
00269     // Get user input for new user
00270     cout « "Enter user number (10000-99999): ";
00271     cin » usrNumber;
00272 
00273     cout « "Enter NIF (up to " « User::MAX_NIF - 1 « " characters): ";
00274     cin » NIF;
00275 
00276     cout « "Enter password: ";
00277     cin » pwd;
00278 
00279     cout « "Select role (0 for ADMIN, 1 for EMPLOYEE): ";
00280     cin » roleChoice;
00281 
00282     User* newUser = nullptr;
00283 
00284     try {
00285         // Create appropriate user type
00286         if (roleChoice == 0) {
00287             newUser = new Admin(usrNumber, NIF, pwd, User::ADMIN);
00288         } else {
00289             newUser = new Employee(usrNumber, NIF, pwd, User::EMPLOYEE);
00290         }
00291 
00292         // Try to add user to database
00293         if (db.addUser(newUser)) {
00294             cout « "User added successfully!" « endl;
00295             displayUserDetails(newUser);
00296         } else {
00297             cout « "Failed to add user. User with number "
00298                  « usrNumber « " may already exist." « endl;
00299             delete newUser; // Clean up if not added to database
00300         }
00301     } catch (const exception& e) {
00302         cout « "Error creating user: " « e.what() « endl;
00303         if (newUser) delete newUser;
00304     }
00305 }
00306 
00317 void testUpdateUser(UserDatabase& db) {
00318     cout « "\n=== Update User ===" « endl;
00319 
00320     u_int32_t usrNumber;
00321     cout « "Enter user number to update: ";
00322     cin » usrNumber;
00323 
00324     // Try to find user
00325     User* existingUser = db.findUserByNumber(usrNumber);
00326     if (!existingUser) {
00327         cout « "User with number " « usrNumber « " not found." « endl;
00328         return;
00329     }
00330 
00331     // Display current user info
00332     cout « "Current user information:" « endl;
00333     displayUserDetails(existingUser);
00334 
00335     // Get updated info
00336     char NIF[User::MAX_NIF];
00337     char pwd[User::MAX_STR];
00338     int roleChoice;
00339 
00340     cout « "Enter new NIF (or leave blank to keep current): ";
00341     cin.ignore(); // Clear input buffer
00342     cin.getline(NIF, User::MAX_NIF);
00343 
00344     cout « "Enter new password (or leave blank to keep current): ";
00345     cin.getline(pwd, User::MAX_STR);
00346 
00347     cout « "Select new role (0 for ADMIN, 1 for EMPLOYEE, "
00348          « "or any other number to keep current): ";
00349     cin » roleChoice;
00350 
00351     // Create updated user object
00352     User* updatedUser = nullptr;
00353     try {
00354         // Use existing values if fields were left blank
00355         const char* newNIF = (NIF[0] == '\0') ?
00356              existingUser->getNIF() : NIF;
00357         const char* newPwd = (pwd[0] == '\0') ?
```

```
00358             existingUser->getPwd().c_str() : pwd;
00359         User::Role newRole = (roleChoice == 0) ? User::ADMIN :
00360                              (roleChoice == 1) ? User::EMPLOYEE :
00361                              existingUser->getRole();
00362
00363         // Create appropriate user type
00364         if (newRole == User::ADMIN) {
00365             updatedUser = new Admin(usrNumber, newNIF, newPwd, User::ADMIN);
00366         } else {
00367             updatedUser = new Employee(usrNumber,
00368                                        newNIF,
00369                                        newPwd,
00370                                        User::EMPLOYEE);
00371         }
00372
00373         // Update the user in the database
00374         if (db.updateUser(updatedUser)) {
00375             cout « "User updated successfully!" « endl;
00376             displayUserDetails(updatedUser);
00377         } else {
00378             cout « "Failed to update user." « endl;
00379             delete updatedUser;
00380         }
00381     } catch (const exception& e) {
00382         cout « "Error updating user: " « e.what() « endl;
00383         if (updatedUser) delete updatedUser;
00384     }
00385 }
00386
00395 void testRemoveUser(UserDatabase& db) {
00396     cout « "\n=== Remove User ===" « endl;
00397
00398     u_int32_t usrNumber;
00399     cout « "Enter user number to remove: ";
00400     cin » usrNumber;
00401
00402     // Try to find user
00403     User* userToRemove = db.findUserByNumber(usrNumber);
00404     if (!userToRemove) {
00405         cout « "User with number " « usrNumber « " not found." « endl;
00406         return;
00407     }
00408
00409     // Display user info
00410     cout « "User to remove:" « endl;
00411     displayUserDetails(userToRemove);
00412
00413     // Confirm removal
00414     char confirm;
00415     cout « "Are you sure you want to remove this user? (y/n): ";
00416     cin » confirm;
00417
00418     if (tolower(confirm) == 'y') {
00419         try {
00420             if (db.removeUser(userToRemove)) {
00421                 cout « "User removed successfully!" « endl;
00422             } else {
00423                 cout « "Failed to remove user." « endl;
00424             }
00425         } catch (const UserDatabaseException& e) {
00426             cout « "Database Exception: " « e.what() « endl;
00427             cout « "Error type: " « e.getErrorString() « endl;
00428             cout « "Error code: " « e.getErrorCode() « endl;
00429         } catch (const std::exception& e) {
00430             // Fallback for other types of exceptions
00431             cout « "Error: " « e.what() « endl;
00432         }
00433     } else {
00434         cout « "User removal cancelled." « endl;
00435     }
00436 }
00437
00447 void testLogin(UserDatabase& db) {
00448     cout « "\n=== User Login ===" « endl;
00449
00450     u_int32_t usrNumber;
00451     char NIF[User::MAX_NIF];
00452     char pwd[User::MAX_STR];
00453
00454     cout « "Enter user number: ";
00455     cin » usrNumber;
00456
00457     cout « "Enter NIF: ";
00458     cin » NIF;
00459
00460     cout « "Enter password: ";
00461     cin » pwd;
```

```
00462
00463     // Attempt login
00464     string passwordStr(pwd);
00465     User* loggedInUser = db.login(usrNumber, NIF, passwordStr);
00466
00467     if (loggedInUser) {
00468         cout « "Login successful!" « endl;
00469         cout « "Welcome, "
00470              « (loggedInUser->getRole() == User::ADMIN
00471                 ? "Administrator" : "Employee")
00472              « " #" « loggedInUser->getUsrNumber() « endl;
00473         displayUserDetails(loggedInUser);
00474     } else {
00475         cout « "Login failed. Invalid credentials." « endl;
00476     }
00477 }
00478
00489 void testPasswordChange(UserDatabase& db) {
00490     cout « "\n=== Change User Password ===" « endl;
00491
00492     // First, login to get an active user
00493     u_int32_t activeUsrNumber;
00494     char activeNIF[User::MAX_NIF];
00495     char activePwd[User::MAX_STR];
00496
00497     try {
00498         cout « "First, please login:" « endl;
00499         cout « "Enter your user number: ";
00500         cin » activeUsrNumber;
00501
00502         cout « "Enter your NIF: ";
00503         cin » activeNIF;
00504
00505         cout « "Enter your password: ";
00506         cin » activePwd;
00507
00508         // Attempt login
00509         string passwordStr(activePwd);
00510         User* activeUser = nullptr;
00511
00512         try {
00513             activeUser = db.login(activeUsrNumber, activeNIF, passwordStr);
00514
00515             if (!activeUser) {
00516                 throw runtime_error("Login failed: Invalid credentials");
00517             }
00518
00519             // Get the user whose password will be changed
00520             u_int32_t targetUsrNumber;
00521             cout « "Enter the user number whose password you want to change: ";
00522             cin » targetUsrNumber;
00523
00524             User* targetUser = db.findUserByNumber(targetUsrNumber);
00525             if (!targetUser) {
00526                 throw runtime_error("User not found");
00527             }
00528
00529             // Check if active user has permission
00530             if (activeUser->getRole() != User::ADMIN &&
00531                 activeUser->getUsrNumber() != targetUser->getUsrNumber()) {
00532                 throw runtime_error("Permission denied");
00533             }
00534
00535             // Attempt password change
00536             if (db.changeUserPass(activeUser, targetUser)) {
00537                 cout « "Password changed successfully." « endl;
00538             } else {
00539                 throw runtime_error("Password change operation failed");
00540             }
00541         }
00542         catch (const runtime_error& e) {
00543             // Log the error details
00544             cout « "Authentication error: " « e.what() « endl;
00545             cout « "User attempted: " « activeUsrNumber « endl;
00546
00547             // Example of re-throwing the same exception
00548             // This will be caught by the outer try-catch block
00549             cout « "Re-throwing exception..." « endl;
00550             throw;  // Re-throw the current exception
00551         }
00552     }
00553     catch (const exception& e) {
00554         // Handle the re-thrown exception
00555         cout « "Operation aborted: " « e.what() « endl;
00556         cout « "Please try again with valid credentials." « endl;
00557     }
00558 }
```

```
00559
00570 void testClearFile(UserDatabase& db, const string& filename) {
00571     cout « "\n=== Clear Database File ===" « endl;
00572
00573     char confirm;
00574     cout « "WARNING: This will delete all users from the file." « endl;
00575     cout « "Are you sure? (y/n): ";
00576     cin » confirm;
00577
00578     if (tolower(confirm) == 'y') {
00579         try {
00580             if (db.clearFile(filename.c_str())) {
00581                 cout « "Database file cleared successfully." « endl;
00582             } else {
00583                 cout « "Failed to clear database file." « endl;
00584             }
00585         } catch (const runtime_error& e) {
00586             cout « "Error: " « e.what() « endl;
00587         }
00588     } else {
00589         cout « "Operation cancelled." « endl;
00590     }
00591 }
00592
00602 void testRandomAccess(const string& filename) {
00603     cout « "\n=== Testing Random Binary Access ===" « endl;
00604
00605     // Open the binary file for reading
00606     ifstream file(filename, ios::in | ios::binary);
00607     if (!file.is_open()) {
00608         cout « "Could not open file " « filename
00609              « " for random access test." « endl;
00610         return;
00611     }
00612
00613     // Calculate file size
00614     file.seekg(0, ios::end);
00615     streampos fileSize = file.tellg();
00616     int numRecords = fileSize / sizeof(UserRecord);
00617
00618     cout « "File contains " « numRecords « " user records" « endl;
00619
00620     // Check if there are any records
00621     if (numRecords <= 0) {
00622         cout « "No records available for random access." « endl;
00623         file.close();
00624         return;
00625     }
00626
00627     // Display the first record (Admin user)
00628     file.seekg(0, ios::beg);
00629     User* firstUser = UserFactory::readUserFromFile(file);
00630
00631     if (firstUser) {
00632         cout « "First record (default admin):" « endl;
00633         cout « *firstUser « endl;
00634         delete firstUser;
00635     } else {
00636         cout « "Failed to read first record." « endl;
00637     }
00638
00639     // If more records exist, access random records
00640     if (numRecords > 1) {
00641         // Let the user choose a record to access
00642         int recordIndex;
00643         while (true) {
00644             cout « "Enter a record index to access (0-" « numRecords - 1
00645                  « ", or -1 to exit): ";
00646             cin » recordIndex;
00647
00648             if (recordIndex == -1) {
00649                 break;
00650             }
00651
00652             if (recordIndex < 0 || recordIndex >= numRecords) {
00653                 cout « "Invalid record index." « endl;
00654                 continue;
00655             }
00656
00657             // Seek to the selected record
00658             file.seekg(recordIndex * sizeof(UserRecord), ios::beg);
00659
00660             // Read the user from this position
00661             User* user = UserFactory::readUserFromFile(file);
00662
00663             if (user) {
00664                 cout « "User at position " « recordIndex « ":" « endl;
```

```
00665                 cout « *user « endl;
00666                 delete user;
00667             } else {
00668                 cout « "Failed to read user at position "
00669                      « recordIndex « endl;
00670             }
00671         }
00672     }
00673
00674     file.close();
00675 }
00676
00677
00686 void testBadAllocException(UserDatabase& db) {
00687     cout « "\n=== Testing std::bad_alloc Exception Handling ===" « endl;
00688
00689     // Size that will likely trigger a bad_alloc exception on most systems
00690     // Using size_t max value effectively requests all available memory
00691     const size_t HUGE_SIZE = std::numeric_limits<size_t>::max() / 10;
00692
00693     try {
00694         cout « "Attempting to allocate " « HUGE_SIZE
00695              « " bytes of memory..." « endl;
00696
00697         // This will almost certainly fail and throw std::bad_alloc
00698         int* hugeArray = new int[HUGE_SIZE];
00699
00700         // This code should never execute if the allocation fails as expected
00701         cout « "Allocation successful (unexpected!)" « endl;
00702         delete[] hugeArray;  // Clean up if allocation somehow succeeds
00703     }
00704     catch (const std::bad_alloc& e) {
00705         // Specific handler for memory allocation failure
00706         cout « "Memory allocation failed as expected!" « endl;
00707         cout « "Exception details: " « e.what() « endl;
00708         cout « "This demonstrates proper handling of std::bad_alloc" « endl;
00709     }
00710     catch (const std::exception& e) {
00711         // Generic handler for other exceptions
00712         cout « "Unexpected exception: " « e.what() « endl;
00713     }
00714
00715     // Parameter is not used but kept for consistency with other test functions
00716     (void)db;
00717 }
00718
00727 void displayUserDetails(const User* user) {
00728     if (!user) {
00729         cout « "No user to display." « endl;
00730         return;
00731     }
00732
00733     cout « "----------------------" « endl;
00734     cout « "User Number: " « user->getUsrNumber() « endl;
00735     cout « "NIF: " « user->getNIF() « endl;
00736     cout « "Password: " « string(strlen(user->getPwd().c_str()), '*') « endl;
00737     cout « "Role: " « (user->getRole() == User::ADMIN ? "ADMIN" : "EMPLOYEE");
00738     cout « "\n----------------------" « endl;
00739 }
00740
00749 void pauseExecution() {
00750     cout « "\nPress Enter to continue...";
00751
00752     // Clean input buffer and wait for Enter
00753     if (cin.peek() == '\n') {
00754         // If there's a pending newline character, consume it
00755         cin.get();
00756     } else {
00757         // If there's no pending newline or other characters in buffer,
00758         // clear the entire buffer up to the next newline
00759         cin.ignore(numeric_limits<streamsize>::max(), '\n');
00760     }
00761 }
```

3.3 mainpage.dox File Reference

Index