3.1: Techniques Used

- Qt framework to create and implement Graphical User Interfaces (GUI)
- Use of SQLite for local database management
- Use of try-catch statements for error detection and handling
- The use of an implementation of the Supermemo algorithm for spaced repetition memorization

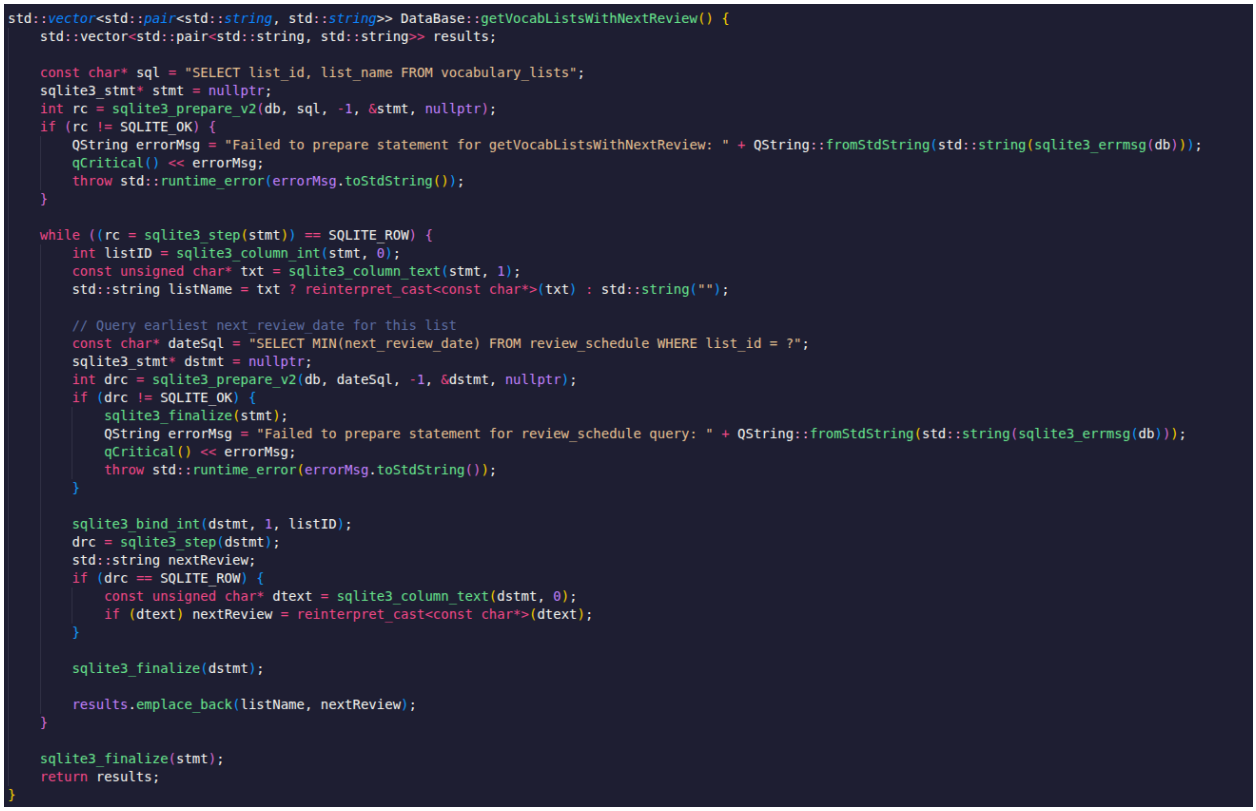
SQL Queries

1. Adding a word into a vocabulary list

```
int DataBase::addWordAndSetup(int listID, const std::string& word, const std::string& partOfSpeech, const std::string& definition, const std::string& language) {
    try {
        beginTransaction();

        int wordID = addOrGetWord(word, partOfSpeech, definition, language);
        if (wordID < 0) {
            rollbackTransaction();
            QString errorMsg = "Failed to obtain or create word id in addWordAndSetup";
            qCritical() << errorMsg;
            throw std::runtime_error(errorMsg.toStdString());
        }

        bool inserted = addWordToList(listID, wordID);

        // Initialize review schedule regardless (INSERT OR IGNORE inside)
        initReviewSchedule(wordID, listID);

        // Only increment progress when a new membership row was inserted
        if (inserted) {
            incrementListProgress(listID, 1);
        }

        commitTransaction();
        return wordID;
    } catch (...) {
        try { rollbackTransaction(); } catch (...) {}
        throw; // rethrow original exception
    }
}
```

This SQL method adds a word that was input by the user. First it checks whether the word has already been added through the method addOrGetWord, which is an integer that acts as a boolean when the word does not exist, returning a 1. But when the word does exist, it just returns the ID of that word. It also initializes the study schedule for that word to be able to keep track of when the user should study it based on the super memo algorithm.

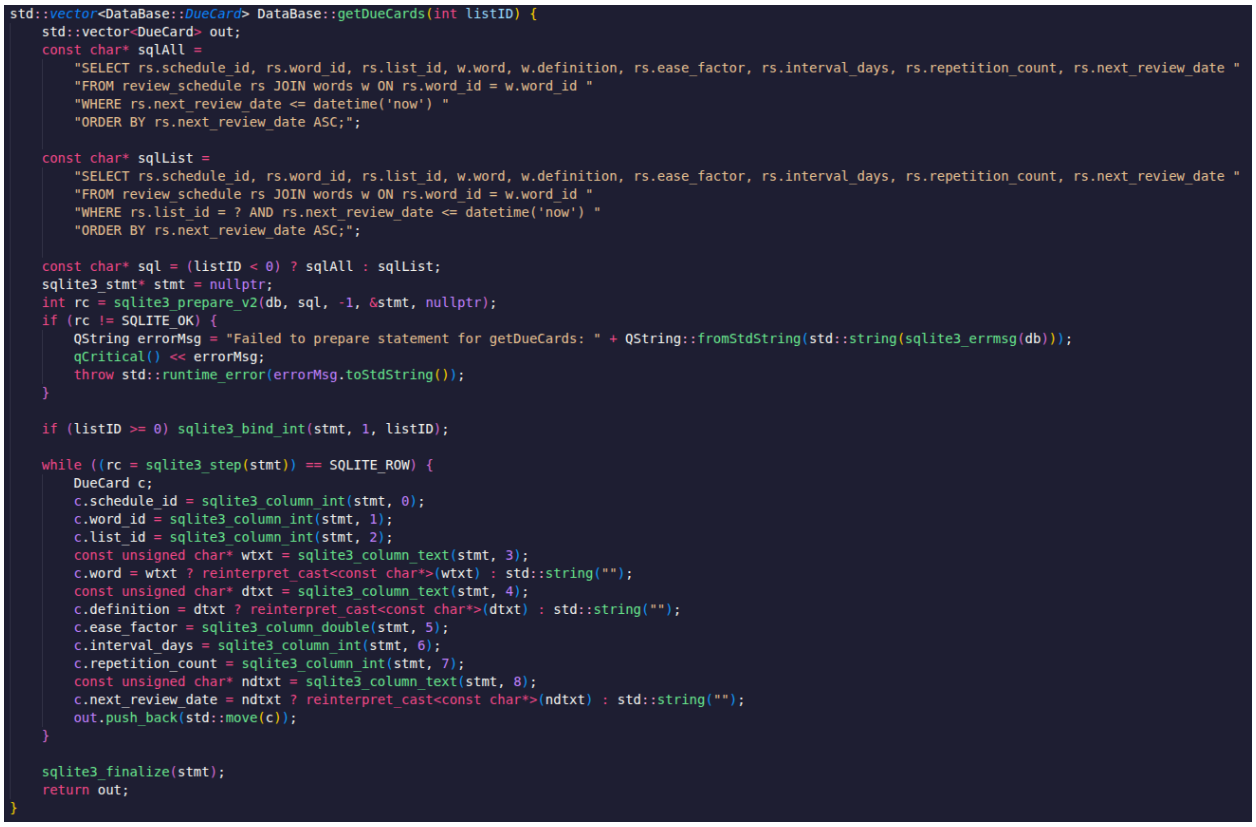
2. Getting the words due for the next study in every single list to display for the user



```
std::vector<std::pair<std::string, std::string>> DataBase::getVocabListsWithNextReview() {
    std::vector<std::pair<std::string, std::string>> results;

    const char* sql = "SELECT list_id, list_name FROM vocabulary_lists";
    sqlite3_stmt* stmt = nullptr;
    int rc = sqlite3_prepare_v2(db, sql, -1, &stmt, nullptr);
    if (rc != SQLITE_OK) {
        QString errorMsg = "Failed to prepare statement for getVocabListsWithNextReview: " + QString::fromStdString(std::string(sqlite3_errmsg(db)));
        qCritical() << errorMsg;
        throw std::runtime_error(errorMsg.toStdString());
    }

    while ((rc = sqlite3_step(stmt)) == SQLITE_ROW) {
        int listID = sqlite3_column_int(stmt, 0);
        const unsigned char* txt = sqlite3_column_text(stmt, 1);
        std::string listName = txt ? reinterpret_cast<const char*>(txt) : std::string("");

        // Query earliest next_review_date for this list
        const char* dateSql = "SELECT MIN(next_review_date) FROM review_schedule WHERE list_id = ?";
        sqlite3_stmt* dstmt = nullptr;
        int drc = sqlite3_prepare_v2(db, dateSql, -1, &dstmt, nullptr);
        if (drc != SQLITE_OK) {
            sqlite3_finalize(stmt);
            QString errorMsg = "Failed to prepare statement for review_schedule query: " + QString::fromStdString(std::string(sqlite3_errmsg(db)));
            qCritical() << errorMsg;
            throw std::runtime_error(errorMsg.toStdString());
        }

        sqlite3_bind_int(dstmt, 1, listID);
        drc = sqlite3_step(dstmt);
        std::string nextReview;
        if (drc == SQLITE_ROW) {
            const unsigned char* dtext = sqlite3_column_text(dstmt, 0);
            if (dtext) nextReview = reinterpret_cast<const char*>(dtext);
        }

        sqlite3_finalize(dstmt);

        results.emplace_back(listName, nextReview);
    }

    sqlite3_finalize(stmt);
    return results;
}
```

This SQL method in my database wrapper class gets all of the lists and the next review dates for those lists to eventually display for the user in the GUI in an ordered table. The query first selects all of the vocabulary lists from the corresponding SQL table and then loops through the corresponding entry for the shortest review time within the review_schedule table. The method then returns a std::vector of std::pairs of the list name and the next review date for each. The use of std::vector allows for easy management of several elements and the use of std::pair allows for multiple items to be stored together without having multi-dimensional vectors. Additionally, there is error handling throughout the method to ensure that there are not failures involving the database.

3. Getting all the cards a user needs to study



```
std::vector<DataBase::DueCard> DataBase::getDueCards(int listID) {
    std::vector<DueCard> out;
    const char* sqlAll =
        "SELECT rs.schedule_id, rs.word_id, rs.list_id, w.word, w.definition, rs.ease_factor, rs.interval_days, rs.repetition_count, rs.next_review_date "
        "FROM review_schedule rs JOIN words w ON rs.word_id = w.word_id "
        "WHERE rs.next_review_date <= datetime('now') "
        "ORDER BY rs.next_review_date ASC;";

    const char* sqlList =
        "SELECT rs.schedule_id, rs.word_id, rs.list_id, w.word, w.definition, rs.ease_factor, rs.interval_days, rs.repetition_count, rs.next_review_date "
        "FROM review_schedule rs JOIN words w ON rs.word_id = w.word_id "
        "WHERE rs.list_id = ? AND rs.next_review_date <= datetime('now') "
        "ORDER BY rs.next_review_date ASC;";

    const char* sql = (listID < 0) ? sqlAll : sqlList;
    sqlite3_stmt* stmt = nullptr;
    int rc = sqlite3_prepare_v2(db, sql, -1, &stmt, nullptr);
    if (rc != SQLITE_OK) {
        QString errorMsg = "Failed to prepare statement for getDueCards: " + QString::fromStdString(std::string(sqlite3_errmsg(db)));
        qCritical() << errorMsg;
        throw std::runtime_error(errorMsg.toStdString());
    }

    if (listID >= 0) sqlite3_bind_int(stmt, 1, listID);

    while ((rc = sqlite3_step(stmt)) == SQLITE_ROW) {
        DueCard c;
        c.schedule_id = sqlite3_column_int(stmt, 0);
        c.word_id = sqlite3_column_int(stmt, 1);
        c.list_id = sqlite3_column_int(stmt, 2);
        const unsigned char* wtxt = sqlite3_column_text(stmt, 3);
        c.word = wtxt ? reinterpret_cast<const char*>(wtxt) : std::string("");
        const unsigned char* dtxt = sqlite3_column_text(stmt, 4);
        c.definition = dtxt ? reinterpret_cast<const char*>(dtxt) : std::string("");
        c.ease_factor = sqlite3_column_double(stmt, 5);
        c.interval_days = sqlite3_column_int(stmt, 6);
        c.repetition_count = sqlite3_column_int(stmt, 7);
        const unsigned char* ndtxt = sqlite3_column_text(stmt, 8);
        c.next_review_date = ndtxt ? reinterpret_cast<const char*>(ndtxt) : std::string("");
        out.push_back(std::move(c));
    }

    sqlite3_finalize(stmt);
    return out;
}
```

This method gets all of the vocabulary words that the user needs to study. The method starts by getting all of the information about a word through the review_schedule table. This is achieved through the JOIN statement, which joins the review_schedule and words tables based on word_id. The two sql statements are prepared in case there isn't a list, so the user can get all of the due words instead for future uses. Then, all of the information of the word is passed to each member variable of a struct that represents the word, before being added to a vector to eventually return from the method. The struct is an excellent way of storing multiple things, almost like a class without the extra overhead. And a vector is just a more memory safe and efficient array.

GUI elements/displaying items

1. Connecting buttons across windows

```
// Connect panel signals
connect(deckListPanel, &DeckListPanel::deckDoubleClicked, this, &MainWindow::onDeckDoubleClicked);
connect(modeSelectorPanel, &ModeSelectorPanel::startStudyClicked, this, &MainWindow::onStartStudy);
connect(modeSelectorPanel, &ModeSelectorPanel::viewAllClicked, this, &MainWindow::onViewAll);
connect(modeSelectorPanel, &ModeSelectorPanel::deleteListClicked, this, &MainWindow::onDeleteList);
connect(studyPanel, &StudyPanel::studyCompleted, this, &MainWindow::onStudyCompleted);
```

Qt has it's own unique way of triggering events across different windows. Through the use of signals and slots, the user is able to use buttons to open new windows, show different panels within the main window of the application, and so on.

2. Offering the user different options for studying cards

```
    // If no due cards, offer random practice mode
    if (cards.empty()) {
        QMessageBox::StandardButton reply;
        reply = QMessageBox::question(this, "No Due Cards",
                                      "No cards are due for review right now.\n\n"
                                      "Would you like to practice random words from this deck?",
                                      QMessageBox::Yes | QMessageBox::No);

        if (reply == QMessageBox::Yes) {
            // Load random practice cards
            auto allWords = db.getWordsInList(listID);

            if (allWords.empty()) {
                QMessageBox::information(this, "No Words", "This deck has no words to practice.");
                return;
            }

            // Create DueCard objects from all words (shuffle them)
            std::vector<DataBase::DueCard> allCards;
            for (const auto &wordTuple : allWords) {
                DataBase::DueCard card;
                card.word_id = std::get<0>(wordTuple);
                card.word = std::get<1>(wordTuple);
                card.definition = std::get<2>(wordTuple);
                card.list_id = listID;
                card.ease_factor = 2.5;
                card.interval_days = 0;
                card.repetition_count = 0;
                card.next_review_date = "";
                card.schedule_id = -1;
                allCards.push_back(card);
            }

            // Shuffle the cards
            std::random_device rd;
            std::mt19937 g(rd());
            std::shuffle(allCards.begin(), allCards.end(), g);

            // Take up to 20 cards for practice session
            size_t practiceSize = std::min(size_t(20), allCards.size());
            cards.clear();
            for (size_t i = 0; i < practiceSize; ++i) {
                cards.push_back(allCards[i]);
            }

            studyPanel->setRandomPracticeMode(true);
        } else {
            return;
        }
    } else {
        studyPanel->setRandomPracticeMode(false);
    }
```

This method, onStartStudy, first checks if there are any cards for the user to study, before offering the opportunity to study all of the cards in a list, in a randomized order. If the user chooses to do so, the program gets all of the cards from a list, adds them to a vector of structs, and shuffles them using the mt19937 random number generator. After shuffling, it takes 20 cards at a time for the user. If the user has cards due, then the program simply allows the user to begin studying. The use of the mt19937 random number generator is best here because it can handle incredibly large amounts of data and still produce randomly reliable results.