实验四：软件分析 实验报告

项目内容：开发一款基于 Qt 的个人记账应用 (LedgerApp)，帮助用户记录日常收支，提供交易记录、账户管理、分类统计及预算设定等功能。

学号：215220030
姓名：朴敏佑
E-mail:215220030@smail.nju.edu.cn
日期：2025年12月8日

1. 实验目的

- 掌握使用分析工具检测项目缺陷的方法
- 评估工具在正确报告和误报方面的表现
- 加深对程序分析技术的理解，提高代码质量和安全性

2. 实验工具和方法

2.1 工具选择

- **静态分析工具**：Cppcheck
- **大语言模型**：ChatGPT (基于 GPT-4/Gemini 模型)
- **形式化验证工具**：ESBMC (Efficient SMT-Based Context-Bounded Model Checker)

2.2 配置与运行方式

- **静态分析工具 (Cppcheck)**：
  - Cppcheck 是一款开源的 C/C++ 静态分析工具。由于 Qt 项目包含大量的框架特定代码（如信号槽机制、UI 生成代码），直接分析可能会产生大量噪音。
  - **配置**：在本次实验中，我使用了 Cppcheck 的默认检查配置，并开启了所有警告类别（`--enable=all`），以尽可能多地发现潜在问题。针对 Qt 项目，我主要关注其对标准 C++ 逻辑的检测能力。
- **大语言模型 (ChatGPT)**：
  - 使用了如下清晰且详细的 Prompt 模板引导模型进行分析：

> "请分析以下 C++ 项目代码（特别是 flaw_demo.cpp 和 database.cpp），找出其中潜在的代码缺陷，特别是关于内存管理、指针安全和数据库操作方面的问题。请列出缺陷类型、位置和原因。"

- **形式化验证工具 (ESBMC)**：
  - ESBMC 是一种基于 SMT 的模型检测器。
  - **配置**：由于对整个 Qt GUI 应用程序进行形式化验证会面临巨大的状态空间爆炸问题，且 ESBMC 对 C++ STL 和 Qt 库的支持有限，因此本次实验主要针对核心逻辑模块和植入缺陷的模块（`flaw_demo.cpp`）进行验证。
  - **运行命令**：使用了 `--memory-leak-check` 和 `--pointer-check` 选项来专门检测内存安全问题。

3. 实验过程

3.1 植入代码缺陷

由于项目主体代码（`LedgerApp`）在开发过程中已经经过了较为严格的测试，且使用了 Qt 的智能指针和容器，内存管理较为安全。为了满足实验要求，评估工具对典型缺陷的检测能力，我在项目中特意创建了一个模块 `project/flaw_demo.cpp`，并在其中植入了三个典型的内存管理缺陷。

植入的代码缺陷类型（共三类）：

1. **CWE-401 (Memory Leak)**：内存泄漏。在 `memory_leak_demo` 函数中，分配了内存但未释放。
2. **CWE-415 (Double Free)**：双重释放。在 `double_free_demo` 函数中，对同一个指针调用了两次 `free`。
3. **CWE-476 (NULL Pointer Dereference)**：空指针解引用。在 `null_pointer_demo` 函数中，逻辑上可能导致对空指针的访问。

3.2 静态分析工具（Cppcheck）的检测结果

- **真实存在缺陷**：
  - Cppcheck 成功检测到了 `flaw_demo.cpp` 中植入的所有三个缺陷：内存泄漏、双重释放和空指针解引用。这证明了它在处理标准 C 风格指针错误时的有效性。
- **误报缺陷（没有对程序造成影响的规范警告）**：
  - **未使用的函数**：报告 `MainWindow::setupUiElements` 为未使用函数。这是因为该函数是在 `MainWindow` 的构造函数中调用的，或者通过 Qt 的元对象系统调用，静态分析工具未能正确推断其调用图。
  - **死代码 (Dead Code)**：报告 `flaw_demo.cpp` 中的 `if (1 > 2)` 为死代码。虽然技术上正确，但这通常是调试或测试逻辑的一部分，不一定是缺陷。

3.3 大语言模型（ChatGPT）的检测结果

- **真实存在缺陷**：
  - ChatGPT 准确识别了植入的全部三个缺陷，并给出了详细的解释和修复建议。
  - 它还指出了代码中一些潜在的规范问题，如建议使用 `nullptr` 代替 `NULL` 。
- **误报缺陷（过度防御性建议）**：
  - **缓冲区溢出风险**：针对 `strcpy` 发出了警告 (CWE-120)。虽然在示例代码中目标缓冲区大小足够，但 LLM 倾向于标记所有不安全的字符串函数，这在某种程度上是误报，但也属于良好的安全建议。
  - **输入验证**：在 `database.cpp` 中，建议对 `bindValue` 进行额外的输入验证。实际上 Qt 的 `QSqlQuery` 已经处理了 SQL 注入问题，这种建议属于过度防御。

3.4 形式化验证工具（ESBMC）的检测结果

- **真实存在缺陷**：
  - ESBMC 在对 `flaw_demo.cpp` 进行验证时，精确地报告了属性违规（Property Violation），包括内存泄漏和非法指针解引用。它提供了反例路径（Counterexample），精确指出了导致错误的执行序列。
- **误报缺陷**：
  - 在受控的验证范围内（ `flaw_demo.cpp` ），ESBMC 没有产生误报。
  - **局限性**：尝试对 `mainwindow.cpp` 等依赖 Qt 库的文件运行 ESBMC 时，由于无法解析 Qt 头文件和复杂的类结构，工具无法运行。这说明形式化工具在大型应用框架中的落地难度较大。

4. 结果分析与评估

4.1 误报率评估

- **ChatGPT**：检出率最高，但误报率（或称“噪音率”）也最高。它倾向于给出大量“最佳实践”建议，其中很多在当前上下文中并非真正的错误。需要开发者具备较强的甄别能力。
- **Cppcheck**：在标准 C++ 代码上的误报率较低，但在涉及 Qt 框架特性（如信号槽、复杂的对象生命周期管理）时，容易出现“未使用函数”或“内存泄漏”（误判 Qt 父子对象管理）的误报。
- **ESBMC**：在能够成功运行的模块上，误报率极低。它的数学基础保证了结果的精确性，但其适用范围（Recall）受限于代码的复杂度和依赖库。

4.2 误报原因分析

- **代码复杂度与框架依赖**：LedgerApp 使用了 Qt 框架。Cppcheck 等静态分析工具往往难以理解 Qt 的父子对象内存管理机制（Parent-Child Ownership），因此可能会错误地报告内存泄漏，或者无法

识别通过 `connect` 连接的槽函数调用，从而报告“未使用函数”。

- **上下文缺失**：ChatGPT 虽然理解能力强，但如果只提供部分代码片段，它可能会基于通用规则进行推断，从而产生误报。例如，它不知道 `QSqlQuery::bindValue` 内部已经处理了转义，因此建议手动验证输入。
- **工具原理限制**：静态分析基于模式匹配和数据流分析，难以处理复杂的运行时状态；而形式化验证虽然精确，但受限于状态空间爆炸，难以处理大规模代码。

5. 结论

5.1 各工具检测能力对比

工具	检出能力	误报率	易用性	适用场景
Cppcheck	中	低	高	CI/CD 集成，日常快速扫描，发现低级错误。
ChatGPT	高	高	高	代码审查辅助，理解复杂逻辑， 提供重构建议。
ESBMC	高 (特定模块)	极低	低	关键算法模块验证，高安全需求场景。

5.2 工具使用建议

- **Cppcheck**：建议作为日常开发的“守门员”。虽然它对 Qt 框架的理解有限，但它能有效捕捉标准的 C++ 错误（如越界、未初始化）。建议通过配置屏蔽特定的 Qt 相关误报。
- **ChatGPT**：建议在代码审查（Code Review）阶段使用。它可以作为“第二双眼睛”，帮助发现逻辑漏洞和设计缺陷。但必须人工复核其结果，不能盲目采信。
- **ESBMC**：建议仅用于核心算法或不依赖复杂框架的独立模块。对于 GUI 应用，全量使用形式化验证成本过高且不切实际。

5.3 工具组合推荐

推荐组合：Cppcheck + ChatGPT

- **理由**：Cppcheck 能够快速、自动化地发现语法和基础逻辑层面的错误，且误报可控。ChatGPT 则能从语义层面理解代码，发现更深层次的逻辑缺陷，并提供修复代码。两者结合，既保证了检测的覆盖率，又兼顾了效率。对于 Qt 这样的大型框架项目，这种组合比试图配置复杂的 Clang-Tidy 或形式化工具更具性价比。