目录

- 项目概述
- 一、工具功能清单
  - 1.1 核心功能
    - 基础功能
    - 帮助系统
  - 1.2 加密算法实现（第一版）
    - 1. 凯撒密码 (Caesar Cipher)
    - 2. 替换密码 (Substitution Cipher)
    - 3. 维吉尼亚密码 (Vigenère Cipher)
    - 4. 栅栏密码 (Rail Fence Cipher)
    - 5. Base64 编码
  - 1.3 后续版本扩展功能（预留设计）
    - 对称加密算法
    - 非对称加密算法
    - 其他功能
- 二、C/C++ 详细开发步骤
  - 2.1 开发环境准备
    - Windows 平台（推荐使用 MSYS2）
    - Linux 平台
    - macOS 平台
  - 2.1.1 配置 VSCode 开发环境（推荐）
    - 1. 安装 VSCode
    - 2. 安装必要的扩展
    - 3. 配置 C/C++ 扩展
    - 4. 配置 CMake Tools
    - 5. 配置调试（launch.json）
    - 6. 配置构建任务（tasks.json）
    - 7. VSCode 使用技巧
  - 2.1.2 零基础入门：如何开始写第一份代码
    - 步骤 1：创建项目目录
    - 步骤 2：创建项目基本结构
    - 步骤 3：编写第一份代码
    - 步骤 4：配置 CMake（首次配置）
    - 步骤 5：编译项目

        - 步骤 6：运行程序
        - 步骤 7：调试代码
        - 步骤 8：修改代码并重新编译
        - 步骤 9：常见问题排查
    - 2.2 项目结构设计
    - 2.3 添加加密功能（在基础框架之上）
        - 步骤 1：实现命令解析器
        - 步骤 3：实现第一个加密算法（凯撒密码）
        - 步骤 4：集成加密功能到主程序
        - 步骤 5：使用 Makefile 简化编译
    - 2.4 第二阶段：添加更多加密算法
    - 2.5 第三阶段：完善功能
        - 文件操作
        - 配置管理
        - 命令历史
    - 2.6 跨平台编译技巧
        - 处理不同操作系统的差异
        - 使用 CMake（推荐，比 Makefile 更跨平台）
- 三、开发过程中可能遇到的问题及解决方案
    - 3.1 编译和链接问题
        - 问题 1：找不到头文件
        - 问题 2：链接错误
        - 问题 3：Windows 和 Linux 换行符不一致
    - 3.2 用户使用问题
        - 问题 1：用户需要安装编译环境吗？
        - 问题 2：如何让用户方便地添加到环境变量？
        - 问题 3：跨平台兼容性问题
    - 3.3 依赖管理问题
        - 问题：如何管理第三方库？
    - 3.4 调试技巧
        - 使用 VSCode 调试（推荐）
        - 使用命令行 GDB（Linux/macOS，进阶）
        - 打印调试（简单但有效）
    - 3.5 性能优化
        - 第一版不需要过度优化
        - 基本优化原则
- 四、开发流程建议
    - 4.1 迭代式开发

    - 4.2 版本控制（Git）
    - 4.3 测试建议
        - 手动测试
        - 自动化测试（进阶）
- 五、示例配置文件
    - 5.1 字符映射表（mapping.json）
- 六、常见问题（FAQ）
    - Q1: 我应该用 C 还是 C++?
    - Q2: 需要使用面向对象编程吗?
    - Q3: 如何确保代码质量?
    - Q4: 项目可以用多久完成?
    - Q5: 学习资源推荐?
- 七、下一步行动
- 八、附录
    - 8.1 推荐的代码风格
    - 8.2 有用的命令行参数解析
    - 8.3 README 模板
- 特性
- 安装
    - 从源码编译
    - 下载预编译版本
- 使用方法
- 许可证
- 总结

CipherX CLI 工具开发指南

项目概述

CipherX 是一个交互式命令行加密/解密工具，支持多种加密算法，使用 C/C++ 实现，具有良好的性能和跨平台特性。

一、工具功能清单

1.1 核心功能

基础功能

- **交互式命令行界面 (REPL)**
  - 启动工具后进入交互模式
  - 支持命令提示符显示
  - 命令历史记录（上下键导航）
  - 优雅的退出机制
- **文本加密/解密**
  - 直接加密输入的文本字符串
  - 直接解密密文字符串
  - 支持中文、英文及常用标点符号
- **文件加密/解密**
  - 读取文件内容进行加密
  - 读取文件内容进行解密
  - 保存加密/解密结果到文件
- **配置管理**
  - 加载自定义字符映射表（JSON 格式）
  - 显示当前使用的映射配置
  - 保存当前配置到文件

帮助系统

- `help` 命令：显示所有可用命令
- `help <command>` 命令：显示特定命令的详细帮助

1.2 加密算法实现（第一版）

1. 凯撒密码 (Caesar Cipher)

- **描述**：最简单的替换加密，将每个字符按照固定位移量进行替换
- **参数**：位移量（shift）
- **命令示例**：

```
cipherx> encrypt caesar "Hello World" 3
Khoor Zruog
```

- **实现难度**：⭐ (非常适合初学者)

2. 替换密码 (Substitution Cipher)

- **描述**：使用自定义字符映射表进行一对一替换
- **参数**：映射表文件路径
- **命令示例**：

```
cipherx> load mapping.json
cipherx> encrypt substitution "Hello"
```

- **实现难度**：⭐ ⭐

3. 维吉尼亚密码 (Vigenère Cipher)

- **描述**：使用密钥字符串进行多表替换加密
- **参数**：密钥字符串
- **命令示例**：

```
cipherx> encrypt vigenere "Hello World" "KEY"
```

- **实现难度**：⭐ ⭐ ⭐

4. 栅栏密码 (Rail Fence Cipher)

- **描述**：将文本按 Z 字形排列后横向读取
- **参数**：栅栏数量（rails）
- **命令示例**：

```
cipherx> encrypt railfence "Hello World" 3
```

- **实现难度**：⭐ ⭐ ⭐

5. Base64 编码

- **描述**：将文本转换为 Base64 编码（严格来说是编码而非加密）
- **参数**：无
- **命令示例**：

```
cipherx> encode base64 "Hello World"
SGVsbG8gV29ybGQ=
```

- **实现难度**：⭐ ⭐

1.3 后续版本扩展功能（预留设计）

对称加密算法

- AES (Advanced Encryption Standard)
- DES (Data Encryption Standard)
- XOR 加密

非对称加密算法

- RSA（需要第三方库支持）

其他功能

- 密码强度分析
- 频率分析工具（用于破解简单密码）
- 批量文件处理
- 配置文件自动加载
- 命令脚本支持（从文件读取命令序列）

二、C/C++ 详细开发步骤

2.1 开发环境准备

Windows 平台（推荐使用 MSYS2）

为什么选择 MSYS2?

- MSYS2 提供了完整的 Unix-like 环境
- 包含 g++, clang++, cmake 等现代化工具链
- 包管理器 pacman 方便安装和更新工具
- 与 Windows 集成良好，适合 C/C++ 开发

1. **安装 MSYS2**
   - 下载地址：https://www.msys2.org/
   - 下载安装程序并运行（推荐安装到 `C:\msys64`）
   - 安装完成后，打开 MSYS2 UCRT64 终端（推荐使用 UCRT64，兼容性更好）
2. **更新 MSYS2 系统**

```
pacman -Syu
```

如果提示关闭窗口，关闭后重新打开 MSYS2 终端，再次运行：

```
pacman -Su
```

3. **安装开发工具**

```
# 安装 GCC/G++ 编译器
pacman -S mingw-w64-ucrt-x86_64-gcc

# 安装 CMake
pacman -S mingw-w64-ucrt-x86_64-cmake

# 安装 Make
pacman -S mingw-w64-ucrt-x86_64-make

# 安装 GDB 调试器
pacman -S mingw-w64-ucrt-x86_64-gdb

# 安装 Git（如果还没有）
pacman -S git
```

4. **配置环境变量**
   为了在 Windows 命令提示符或 PowerShell 中使用这些工具，需要将 MSYS2 的 bin 目录添加到系统 PATH：
   - 右键"此电脑" → 属性 → 高级系统设置 → 环境变量
   - 在"系统变量"中找到 `Path`，点击编辑
   - 添加以下路径（根据实际安装路径调整）：

     ```
     C:\msys64\ucrt64\bin
     C:\msys64\usr\bin
     ```

5. **验证安装**
   打开新的 PowerShell 或命令提示符窗口：

```
gcc --version
g++ --version
cmake --version
gdb --version
```

如果显示版本信息，说明安装成功！

其他选择：

- **MinGW-w64**：轻量级选择，下载地址：https://www.mingw-w64.org/
- **Visual Studio Community**：功能强大的 IDE，包含 MSVC 编译器

Linux 平台

1. **安装 GCC/G++**

```
sudo apt update
sudo apt install build-essential
```

2. **验证安装**

```
gcc --version
g++ --version
```

macOS 平台

1. **安装 Xcode Command Line Tools**

```
xcode-select --install
```

2. **验证安装**

```
gcc --version
```

2.1.1 配置 VSCode 开发环境（推荐）

Visual Studio Code 是一个轻量级但功能强大的代码编辑器，非常适合 C/C++ 开发。

1. 安装 VSCode

- 下载地址：https://code.visualstudio.com/
- 下载并安装适合你系统的版本（Windows x64 User Installer 推荐）

2. 安装必要的扩展

打开 VSCode，点击左侧的扩展图标（或按 `Ctrl+Shift+X` ），搜索并安装以下扩展：

必装扩展：

1. **C/C++** (Microsoft)

- 提供智能代码补全、语法高亮、调试支持
- 扩展 ID: `ms-vscode.cpptools`

2. **CMake Tools** (Microsoft)
   - CMake 项目支持，可视化配置和构建
   - 扩展 ID: `ms-vscode.cmake-tools`
3. **CMake** (twxs)
   - CMake 语法高亮和智能提示
   - 扩展 ID: `twxs.cmake`

推荐扩展:

4. **Chinese (Simplified) Language Pack** (Microsoft)

- 中文界面（如果你喜欢中文）
- 扩展 ID: `MS-CEINTL.vscode-language-pack-zh-hans`

5. **Code Runner**
   - 快速运行代码片段
   - 扩展 ID: `formulahendry.code-runner`
6. **GitLens**
   - 增强 Git 功能
   - 扩展 ID: `eamodio.gitlens`

3. 配置 C/C++ 扩展

安装完 C/C++ 扩展后，需要告诉它使用哪个编译器。

1. 在 VSCode 中按 `Ctrl+Shift+P` 打开命令面板
2. 输入 `C/C++: Edit Configurations (UI)`
3. 在配置界面中设置：
   - **编译器路径**:
     - Windows (MSYS2): `C:/msys64/ucrt64/bin/g++.exe`
     - Linux: `/usr/bin/g++`
     - macOS: `/usr/bin/clang++`
   - **IntelliSense 模式**:
     - Windows: `windows-gcc-x64`
     - Linux: `linux-gcc-x64`
     - macOS: `macos-clang-x64`
   - **C++ 标准**: `c++11` 或更高

这会在项目目录下创建 `.vscode/c_cpp_properties.json` 文件。

示例配置文件 `.vscode/c_cpp_properties.json`（Windows + MSYS2）：

```
{
    "configurations": [
        {
            "name": "Win32",
            "includePath": [
                "${workspaceFolder}/**"
            ],
            "defines": [
                "_DEBUG",
                "UNICODE",
                "_UNICODE"
            ],
            "compilerPath": "C:/msys64/ucrt64/bin/g++.exe",
            "cStandard": "c17",
            "cppStandard": "c++11",
            "intelliSenseMode": "windows-gcc-x64"
        }
    ],
    "version": 4
}
```

4. 配置 CMake Tools

1. 按 `Ctrl+Shift+P`，输入 `CMake: Select a Kit`
2. 选择你安装的编译器工具链：
    - Windows: 选择 `GCC xxx ucrt64` 或类似选项
    - Linux/macOS: 选择系统的 GCC 或 Clang
3. 在 VSCode 底部状态栏会显示：
    - 当前选择的工具链
    - 构建类型（Debug/Release）
    - 构建按钮

5. 配置调试（launch.json）

创建 `.vscode/launch.json` 文件来配置调试器：

Windows (MSYS2/MinGW) 配置：

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Debug CipherX",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}/build/cipherx.exe",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${workspaceFolder}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "miDebuggerPath": "C:/msys64/ucrt64/bin/gdb.exe",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ],
            "preLaunchTask": "CMake: build"
        }
    ]
}
```

Linux 配置:

```
{
    "version": "0.2.0",
    "configurations": [
        {
            "name": "Debug CipherX",
            "type": "cppdbg",
            "request": "launch",
            "program": "${workspaceFolder}/build/cipherx",
            "args": [],
            "stopAtEntry": false,
            "cwd": "${workspaceFolder}",
            "environment": [],
            "externalConsole": false,
            "MIMode": "gdb",
            "setupCommands": [
                {
                    "description": "Enable pretty-printing for gdb",
                    "text": "-enable-pretty-printing",
                    "ignoreFailures": true
                }
            ],
            "preLaunchTask": "CMake: build"
        }
    ]
}
```

6. 配置构建任务（tasks.json）

创建 `.vscode/tasks.json` 来定义构建任务：

```
{
    "version": "2.0.0",
    "tasks": [
        {
            "label": "CMake: build",
            "type": "shell",
            "command": "cmake",
            "args": [
                "--build",
                "${workspaceFolder}/build"
            ],
            "group": {
                "kind": "build",
                "isDefault": true
            },
            "problemMatcher": ["$gcc"],
            "detail": "Build the project using CMake"
        },
        {
            "label": "CMake: configure",
            "type": "shell",
            "command": "cmake",
            "args": [
                "-S",
                "${workspaceFolder}",
                "-B",
                "${workspaceFolder}/build"
            ],
            "problemMatcher": [],
            "detail": "Configure CMake project"
        },
        {
            "label": "CMake: configure (Windows MSYS2)",
            "type": "shell",
            "command": "cmake",
            "args": [
                "-S",
                "${workspaceFolder}",
                "-B",
                "${workspaceFolder}/build",
                "-G",
                "MinGW Makefiles"
            ],
```

```
            "problemMatcher": [],
            "detail": "Configure CMake project for Windows MSYS2"
        }
    ]
}
```

注意:

- **Windows MSYS2 用户**：使用 `CMake: configure (Windows MSYS2)` 任务
- **Linux/macOS 用户**：使用 `CMake: configure` 任务

7. VSCode 使用技巧

常用快捷键:

- `Ctrl+Shift+B`：构建项目
- `F5`：启动调试
- `Ctrl+F5`：运行不调试
- `F9`：设置/取消断点
- `F10`：单步执行（跳过函数）
- `F11`：单步执行（进入函数）
- `Shift+F11`：跳出函数
- `Ctrl+Shift+P`：命令面板
- `Ctrl+\`` ：打开终端

CMake Tools 快捷操作:

- 底部状态栏点击 "Build" 按钮直接构建
- 点击 "Debug" 或 "Release" 切换构建类型
- 点击 "🔧" 图标快速构建

调试技巧:

1. 在代码行号左侧点击设置断点（红点）
2. 按 `F5` 开始调试
3. 使用左侧调试面板查看变量、调用栈、监视表达式
4. 在"调试控制台"中输入变量名查看值

2.1.2 零基础入门：如何开始写第一份代码

这一节专门为完全没有开发过类似项目的初学者准备，手把手教你从零开始。

步骤 1：创建项目目录

1. **在合适的位置创建项目文件夹**
   例如在 `D:\Projects\` 下：

   ```
   # Windows (PowerShell 或 CMD)
   mkdir D:\Projects\cipherx
   cd D:\Projects\cipherx

   # Linux/macOS
   mkdir ~/Projects/cipherx
   cd ~/Projects/cipherx
   ```

2. **用 VSCode 打开项目文件夹**
   - 方法 1：在文件夹中右键 → "通过 Code 打开"（如果安装时勾选了此选项）
   - 方法 2：打开 VSCode → 文件 → 打开文件夹 → 选择 `cipherx` 目录
   - 方法 3：在终端中执行 `code .`（前提是 VSCode 已添加到 PATH）

步骤 2：创建项目基本结构

在 VSCode 中创建以下目录和文件：

1. **创建目录**
   - 在 VSCode 左侧资源管理器中，点击"新建文件夹"图标
   - 创建 `src` 目录
   - 创建 `build` 目录（稍后用于存放编译产物）
2. **创建文件**
   - 在 `src` 目录中创建 `main.cpp` 文件
   - 在项目根目录创建 `CMakeLists.txt` 文件
   - 在项目根目录创建 `.gitignore` 文件

你的目录结构现在应该是这样：

```
cipherx/
├── src/
│   └── main.cpp
├── build/
├── CMakeLists.txt
└── .gitignore
```

步骤 3：编写第一份代码

文件 1： `src/main.cpp` （你的第一个 C++ 程序）

在 VSCode 中打开 `src/main.cpp`， 输入以下代码：

```cpp
#include <iostream>
#include <string>

using namespace std;

void printWelcome() {
    cout << "\n";
    cout << "=====================================\n";
    cout << "  CipherX - Text Encryption Tool     \n";
    cout << "  Version 0.1.0                      \n";
    cout << "=====================================\n";
    cout << "\n";
    cout << "Type 'help' for available commands\n";
    cout << "\n";
}

void printHelp() {
    cout << "Available commands:\n";
    cout << "  help      Show this help message\n";
    cout << "  exit      Exit the program\n";
}

int main() {
    printWelcome();

    string command;

    while (true) {
        cout << "cipherx> ";
        getline(cin, command);

        if (command.empty()) {
            continue;
        }

        if (command == "help") {
            printHelp();
        } else if (command == "exit" || command == "quit") {
            cout << "Goodbye!\n";
            break;
        } else {
            cout << "Unknown command: '" << command << "'\n";
            cout << "Type 'help' for available commands.\n";
```

```
        }
    }

    return 0;
}
```

代码说明：

- `#include <iostream>`：包含输入输出流库（用于 cout, cin）
- `#include <string>`：包含字符串类库
- `using namespace std;`：使用标准命名空间，简化代码
- `printWelcome()`：显示欢迎信息的函数
- `printHelp()`：显示帮助信息的函数
- `main()`：程序入口函数
- `while (true)`：无限循环，直到用户输入 exit
- `getline(cin, command)`：读取用户输入的一整行

文件 2：`CMakeLists.txt`（CMake 构建配置）

在项目根目录的 `CMakeLists.txt` 中输入：

```
cmake_minimum_required(VERSION 3.10)
project(CipherX VERSION 0.1.0)

# 设置 C++ 标准为 C++11
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# 包含源代码目录
include_directories(${PROJECT_SOURCE_DIR}/src)

# 源文件列表
set(SOURCES
    src/main.cpp
)

# 生成可执行文件
add_executable(cipherx ${SOURCES})
```

配置说明：

- `cmake_minimum_required`：要求的最低 CMake 版本

- `project`： 项目名称和版本
- `set(CMAKE_CXX_STANDARD 11)`： 使用 C++11 标准
- `include_directories`： 指定头文件搜索路径
- `set(SOURCES ...)`： 列出所有源文件
- `add_executable`： 生成名为 cipherx 的可执行文件

注意：可执行文件默认会生成在 `build/` 目录中

文件 3： `.gitignore` （Git 忽略文件）

```
# 编译产物
build/
*.o
*.exe
cipherx

# IDE 配置（可选，有些人喜欢共享 VSCode 配置）
# .vscode/

# 操作系统文件
.DS_Store
Thumbs.db

# 调试文件
*.dSYM/
```

步骤 4：配置 CMake（首次配置）

1. **打开 VSCode 集成终端**
   - 按 `Ctrl+`` 或 菜单：终端 → 新建终端
2. **创建构建目录并配置 CMake**
   Windows (MSYS2)：

   ```
   # 确保在项目根目录
   cd D:\Projects\cipherx

   # 配置 CMake（使用 MinGW Makefiles）
   cmake -S . -B build -G "MinGW Makefiles"
   ```

   Linux/macOS：

```
# 确保在项目根目录
cd ~/Projects/cipherx

# 配置 CMake
cmake -S . -B build
```

3. **如果使用 CMake Tools 扩展**
   - 按 `Ctrl+Shift+P`
   - 输入 `CMake: Configure`
   - 选择你的编译工具链（Kit）
   - CMake Tools 会自动完成配置

步骤 5：编译项目

方法 1：使用终端命令

```
# 编译
cmake --build build

# Windows 下编译后会生成 build/cipherx.exe
# Linux/macOS 下会生成 build/cipherx
```

方法 2：使用 VSCode 快捷键

- 按 `Ctrl+Shift+B` （构建）
- 或点击底部状态栏的 "Build" 按钮

方法 3：使用 CMake Tools

- 点击底部状态栏的 " 🔧 Build" 按钮

步骤 6：运行程序

方法 1：在终端中运行

```
# Windows
.\build\cipherx.exe

# Linux/macOS
./build/cipherx
```

方法 2：使用 VSCode 调试

- 按 `F5` 启动调试
- 或按 `Ctrl+F5` 运行不调试

预期输出:

```
=====================================
  CipherX - Text Encryption Tool
  Version 0.1.0
=====================================

Type 'help' for available commands

cipherx> help
Available commands:
  help      Show this help message
  exit      Exit the program
cipherx> exit
Goodbye!
```

步骤 7：调试代码

1. **设置断点**
   - 在 `main.cpp` 的 `main()` 函数中，找到 `printWelcome();` 这一行点击行号左侧，设置红色断点
2. **启动调试**
   - 按 `F5` 开始调试
   - 程序会在断点处暂停
3. **调试操作**
   - 查看左侧"变量"面板，可以看到当前作用域的变量
   - 按 `F10` 单步执行（逐行执行）
   - 按 `F11` 进入函数内部
   - 按 `F5` 继续运行到下一个断点
   - 在"调试控制台"输入变量名查看值
4. **常见调试场景**
   - 在 `while (true)` 循环内设置断点，观察每次循环的 `command` 变量值
   - 在条件分支处设置断点，验证程序逻辑是否正确

步骤 8：修改代码并重新编译

现在尝试添加一个新功能：

在 `main.cpp` 中添加版本信息命令：

```
// 在 main() 函数的 while 循环中，添加新的条件分支
if (command == "help") {
    printHelp();
} else if (command == "version") {
    cout << "CipherX Version 0.1.0\n";
    cout << "Built with C++11\n";
} else if (command == "exit" || command == "quit") {
    cout << "Goodbye!\n";
    break;
} else {
    // ... 原有代码
}
```

同时更新 `printHelp()` 函数：

```
void printHelp() {
    cout << "Available commands:\n";
    cout << "  help      Show this help message\n";
    cout << "  version   Show version information\n";
    cout << "  exit      Exit the program\n";
}
```

保存文件后：

1. 按 `Ctrl+Shift+B` 重新编译
2. 按 `Ctrl+F5` 运行
3. 输入 `version` 测试新功能

恭喜！你已经完成了第一个 CLI 程序，并学会了如何修改和调试代码！ 🎉

步骤 9：常见问题排查

问题 1：编译失败，提示 "cmake: command not found"

- 确认 CMake 已正确安装并添加到 PATH
- Windows 用户确认 MSYS2 的 bin 目录已添加到系统 PATH
- 重启 VSCode 和终端

问题 2：找不到 g++ 或编译器

- 检查编译器是否正确安装： `g++ --version`
- 确认 `.vscode/c_cpp_properties.json` 中的 `compilerPath` 正确

问题 3：程序运行后中文显示乱码

- Windows 用户在终端执行： `chcp 65001` （切换到 UTF-8）
- 或在代码中添加以下内容：

```
// 在文件开头添加
#ifdef _WIN32
#include <windows.h>
#endif

// 在 main() 函数开头调用
int main() {
    #ifdef _WIN32
    SetConsoleOutputCP(CP_UTF8);
    SetConsoleCP(CP_UTF8);
    #endif
    // ... 其余代码
}
```

问题 4：调试时提示找不到 gdb

- 确认 gdb 已安装： `gdb --version`
- 检查 `.vscode/launch.json` 中的 `miDebuggerPath` 是否正确

2.2 项目结构设计

```
cipherx/
├── src/                    # 源代码目录
│   ├── main.cpp            # 主程序入口
│   ├── cli.cpp             # CLI 交互逻辑
│   ├── cli.h               # CLI 头文件
│   ├── cipher/             # 加密算法模块
│   │   ├── cipher_base.h   # 加密算法基类
│   │   ├── caesar.cpp      # 凯撒密码实现
│   │   ├── caesar.h
│   │   ├── substitution.cpp
│   │   ├── substitution.h
│   │   ├── vigenere.cpp
│   │   ├── vigenere.h
│   │   └── ...
│   ├── utils/              # 工具函数
│   │   ├── string_utils.cpp
│   │   ├── string_utils.h
│   │   ├── file_utils.cpp
│   │   └── file_utils.h
│   └── config/             # 配置管理
│       ├── config.cpp
│       └── config.h
├── include/                # 公共头文件
├── tests/                  # 测试代码
├── docs/                   # 文档
├── examples/               # 示例配置文件
│   └── mapping.json
├── build/                  # 编译输出目录（不提交到 git）
├── Makefile               # Make 构建脚本
├── CMakeLists.txt         # CMake 构建脚本（可选）
├── .gitignore             # Git 忽略文件
├── README.md              # 项目说明
├── Guidance.md            # 本开发指南
└── LICENSE                # 许可证
```

2.3 添加加密功能（在基础框架之上）

前提条件:

- 你已经完成了 2.1.2 节的零基础入门指南，有一个能运行的基本 CLI 程序

- 你熟悉了 VSCode 的编译和调试流程

如果你还没有创建基本框架，请先返回 **2.1.2 节**完成基础搭建。

现在我们在已有的框架基础上，添加真正的加密解密功能。

步骤 1：实现命令解析器

文件： `src/utils/string_utils.h`

```
#ifndef STRING_UTILS_H
#define STRING_UTILS_H

#include <string>
#include <vector>

// 分割字符串
std::vector<std::string> split(const std::string& str, char delimiter = ' ');

// 去除字符串首尾空格
std::string trim(const std::string& str);

#endif
```

文件： `src/utils/string_utils.cpp`

```cpp
#include "string_utils.h"
#include <sstream>
#include <algorithm>

std::vector<std::string> split(const std::string& str, char delimiter) {
    std::vector<std::string> tokens;
    std::string token;
    std::istringstream tokenStream(str);

    while (std::getline(tokenStream, token, delimiter)) {
        if (!token.empty()) {
            tokens.push_back(token);
        }
    }

    return tokens;
}

std::string trim(const std::string& str) {
    size_t start = str.find_first_not_of(" \t\n\r");
    if (start == std::string::npos) {
        return "";
    }

    size_t end = str.find_last_not_of(" \t\n\r");
    return str.substr(start, end - start + 1);
}
```

步骤 3：实现第一个加密算法（凯撒密码）

文件： `src/cipher/caesar.h`

```
#ifndef CAESAR_H
#define CAESAR_H

#include <string>

class Caesar {
public:
    // 加密函数
    static std::string encrypt(const std::string& text, int shift);

    // 解密函数
    static std::string decrypt(const std::string& text, int shift);

private:
    // 对单个字符进行位移
    static char shiftChar(char c, int shift);
};

#endif
```

文件： src/cipher/caesar.cpp

```cpp
#include "caesar.h"

std::string Caesar::encrypt(const std::string& text, int shift) {
    std::string result = text;

    for (size_t i = 0; i < result.length(); i++) {
        result[i] = shiftChar(result[i], shift);
    }

    return result;
}

std::string Caesar::decrypt(const std::string& text, int shift) {
    // 解密就是反向位移
    return encrypt(text, -shift);
}

char Caesar::shiftChar(char c, int shift) {
    // 处理大写字母 A-Z
    if (c >= 'A' && c <= 'Z') {
        return 'A' + (c - 'A' + shift + 26) % 26;
    }
    // 处理小写字母 a-z
    else if (c >= 'a' && c <= 'z') {
        return 'a' + (c - 'a' + shift + 26) % 26;
    }
    // 其他字符不变
    else {
        return c;
    }
}
```

步骤 4：集成加密功能到主程序

更新 `src/main.cpp`，添加对 encrypt 和 decrypt 命令的支持：

```cpp
#include <iostream>
#include <string>
#include <sstream>
#include "cipher/caesar.h"
#include "utils/string_utils.h"

// ... (保留之前的函数)

int main() {
    printWelcome();

    string command;
    int defaultShift = 3;  // 默认位移量

    while (true) {
        cout << "cipherx> ";
        getline(cin, command);

        if (command.empty()) {
            continue;
        }

        vector<string> tokens = split(command, ' ');

        if (tokens[0] == "help") {
            printHelp();
        }
        else if (tokens[0] == "encrypt") {
            if (tokens.size() < 2) {
                cout << "Usage: encrypt <text> [shift]\n";
                continue;
            }

            // 获取要加密的文本（可能包含空格）
            size_t textStart = command.find(tokens[1]);
            string text = command.substr(textStart);

            string encrypted = Caesar::encrypt(text, defaultShift);
            cout << "Encrypted: " << encrypted << "\n";
        }
        else if (tokens[0] == "decrypt") {
            if (tokens.size() < 2) {
                cout << "Usage: decrypt <text> [shift]\n";
```

```
                continue;
            }

            size_t textStart = command.find(tokens[1]);
            string text = command.substr(textStart);

            string decrypted = Caesar::decrypt(text, defaultShift);
            cout << "Decrypted: " << decrypted << "\n";
        }
        else if (tokens[0] == "exit" || tokens[0] == "quit") {
            cout << "Goodbye!\n";
            break;
        }
        else {
            cout << "Unknown command. Type 'help' for available commands.\n";
        }
    }

    return 0;
}
```

步骤 5：使用 Makefile 简化编译

文件： `Makefile`

```
# 编译器
CXX = g++

# 编译选项
CXXFLAGS = -std=c++11 -Wall -Wextra -I./src

# 目标可执行文件
TARGET = cipherx

# 源文件
SOURCES = src/main.cpp \
          src/cipher/caesar.cpp \
          src/utils/string_utils.cpp

# 对象文件
OBJECTS = $(SOURCES:.cpp=.o)

# 默认目标
all: $(TARGET)

# 链接生成可执行文件
$(TARGET): $(OBJECTS)
         $(CXX) $(CXXFLAGS) -o $(TARGET) $(OBJECTS)

# 编译源文件
%.o: %.cpp
         $(CXX) $(CXXFLAGS) -c $< -o $@

# 清理编译产物
clean:
         rm -f $(OBJECTS) $(TARGET)
         rm -f cipherx.exe  # Windows

# 安装到系统（可选）
install: $(TARGET)
         cp $(TARGET) /usr/local/bin/

.PHONY: all clean install
```

使用方法:

```
# 编译
make

# 清理
make clean

# 重新编译
make clean && make

# 安装到系统（Linux/macOS，需要 sudo）
sudo make install
```

2.4 第二阶段：添加更多加密算法

按照相同的模式，实现其他加密算法：

1. 在 `src/cipher/` 目录下创建对应的 `.h` 和 `.cpp` 文件
2. 实现 `encrypt` 和 `decrypt` 函数
3. 在主程序中添加新的命令处理逻辑
4. 更新 Makefile 添加新的源文件

示例：替换密码（Substitution Cipher）

需要支持 JSON 配置文件，可以使用简单的 JSON 解析库，推荐初学者使用：

- **nlohmann/json**：单头文件，易于集成
- 下载地址：https://github.com/nlohmann/json

2.5 第三阶段：完善功能

文件操作

文件： `src/utils/file_utils.h`

```
#ifndef FILE_UTILS_H
#define FILE_UTILS_H

#include <string>

class FileUtils {
public:
    // 读取文件内容
    static std::string readFile(const std::string& filename);

    // 写入文件内容
    static bool writeFile(const std::string& filename, const std::string& content);

    // 检查文件是否存在
    static bool fileExists(const std::string& filename);
};

#endif
```

配置管理

支持从 JSON 文件加载字符映射表等配置。

命令历史

在 Windows 平台可以使用 `conio.h`，Linux/macOS 可以使用 `readline` 库或自己实现简单的历史记录。

2.6 跨平台编译技巧

处理不同操作系统的差异

```
#ifdef _WIN32
    // Windows 特定代码
    #include <windows.h>
    #define CLEAR_SCREEN "cls"
#else
    // Linux/macOS 特定代码
    #include <unistd.h>
    #define CLEAR_SCREEN "clear"
#endif
```

使用 CMake（推荐，比 Makefile 更跨平台）

文件： `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.10)
project(CipherX VERSION 0.1.0)

# 设置 C++ 标准
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# 包含目录
include_directories(${PROJECT_SOURCE_DIR}/src)

# 源文件
set(SOURCES
    src/main.cpp
    src/cipher/caesar.cpp
    src/utils/string_utils.cpp
)

# 可执行文件
add_executable(cipherx ${SOURCES})

# Windows 下不显示控制台窗口（可选）
# if(WIN32)
#     set_target_properties(cipherx PROPERTIES WIN32_EXECUTABLE TRUE)
# endif()
```

编译步骤：

```
# 创建构建目录
mkdir build
cd build

# 生成构建文件
cmake ..

# 编译
cmake --build .

# 或者使用 make（Linux/macOS）
make
```

三、开发过程中可能遇到的问题及解决方案

3.1 编译和链接问题

问题 1：找不到头文件

错误信息： `fatal error: xxx.h: No such file or directory`

解决方案：

- 检查 `#include` 路径是否正确
- 确保编译时使用了 `-I` 选项指定包含目录
- 在 Makefile 中正确设置 `CXXFLAGS`

问题 2：链接错误

错误信息： `undefined reference to 'xxx'`

解决方案：

- 确保所有 `.cpp` 文件都被编译并链接
- 检查 Makefile 中的 `SOURCES` 列表
- 检查函数声明和定义是否一致

问题 3：Windows 和 Linux 换行符不一致

解决方案：

- 使用 Git 配置自动转换：

```
git config --global core.autocrlf true  # Windows
git config --global core.autocrlf input  # Linux/macOS
```

3.2 用户使用问题

问题 1：用户需要安装编译环境吗?

答案：不需要!

解决方案 - 分发预编译的二进制文件：

1. **为每个平台编译独立的可执行文件**
   - Windows: `cipherx.exe`
   - Linux: `cipherx` (ELF binary)
   - macOS: `cipherx` (Mach-O binary)
2. **使用静态链接（重要！）**
   在编译时添加静态链接选项，这样可执行文件不依赖系统库：

   ```
   # Linux 静态编译
   g++ -static -o cipherx src/*.cpp

   # Windows (MinGW)
   g++ -static-libgcc -static-libstdc++ -o cipherx.exe src/*.cpp
   ```

   注意：完全静态编译可能导致文件较大，但用户可以直接运行。
3. **GitHub Releases 发布**
   在 GitHub 仓库创建 Release，上传不同平台的可执行文件：

   ```
   cipherx-v0.1.0-windows-x64.exe
   cipherx-v0.1.0-linux-x64
   cipherx-v0.1.0-macos-x64
   ```

4. **提供安装脚本（可选）**
   Windows (`install.bat`):

```
@echo off
copy cipherx.exe C:\Windows\System32\
echo CipherX installed successfully!
pause
```

Linux/macOS (`install.sh`):

```
#!/bin/bash
sudo cp cipherx /usr/local/bin/
sudo chmod +x /usr/local/bin/cipherx
echo "CipherX installed successfully!"
```

问题 2：如何让用户方便地添加到环境变量?

Windows 方案:

1. 手动方式：将 `cipherx.exe` 复制到 `C:\Windows\System32\`
2. 推荐方式：创建专用目录

```
C:\Program Files\CipherX\cipherx.exe
```

然后添加到 PATH：

- 右键"此电脑" → 属性 → 高级系统设置 → 环境变量
- 在"系统变量"中找到 `Path` ， 点击编辑
- 添加 `C:\Program Files\CipherX\`

Linux/macOS 方案:

1. 复制到 `/usr/local/bin/` （推荐）

```
sudo cp cipherx /usr/local/bin/
sudo chmod +x /usr/local/bin/cipherx
```

2. 或添加到用户目录

```
mkdir -p ~/.local/bin
cp cipherx ~/.local/bin/
echo 'export PATH="$HOME/.local/bin:$PATH"' >> ~/.bashrc
source ~/.bashrc
```

问题 3：跨平台兼容性问题

常见问题：

- 文件路径分隔符不同（Windows 用 `\`，Linux/macOS 用 `/`）
- 控制台编码不同（中文显示问题）

解决方案：

1. **统一使用 `/` 作为路径分隔符**
   C++ 标准库会自动处理
2. **处理中文编码**
   Windows 控制台默认使用 GBK，可以在程序开始时设置：

```
#ifdef _WIN32
#include <windows.h>

void setupConsole() {
    // 设置控制台代码页为 UTF-8
    SetConsoleOutputCP(CP_UTF8);
    SetConsoleCP(CP_UTF8);
}
#endif
```

3.3 依赖管理问题

问题：如何管理第三方库?

方案 1：单头文件库（推荐初学者）

- 使用 nlohmann/json 这样的单头文件库
- 直接将 `.h` 文件复制到项目中
- 不需要额外编译和链接

方案 2：包管理器

- **vcpkg**（微软开发，跨平台）

```
# 安装 vcpkg
git clone https://github.com/Microsoft/vcpkg.git
cd vcpkg
./bootstrap-vcpkg.sh  # Linux/macOS
.\bootstrap-vcpkg.bat  # Windows

# 安装库
./vcpkg install nlohmann-json
```

- **Conan**（Python 编写，跨平台）

3.4 调试技巧

使用 VSCode 调试（推荐）

如果你按照 **2.1.1 节**配置了 VSCode 环境，调试非常简单：

1. 确保已配置 launch.json（参考 2.1.1 节第 5 步）

2. 设置断点

- 在代码行号左侧点击，出现红色圆点即为断点
- 或者将光标放在某行，按 `F9` 切换断点

3. 启动调试

- 按 `F5` 开始调试（会自动编译）
- 或点击左侧调试图标，然后点击绿色播放按钮

4. 调试控制

- `F5`：继续运行（到下一个断点或程序结束）
- `F10`：单步跳过（执行当前行，不进入函数内部）
- `F11`：单步进入（进入函数内部）
- `Shift+F11`：单步跳出（跳出当前函数）
- `Ctrl+Shift+F5`：重启调试
- `Shift+F5`：停止调试

5. 查看变量

- **变量面板**：自动显示当前作用域的所有变量
- **监视面板**：添加你想持续观察的表达式
- **调用堆栈**：查看函数调用链

- **调试控制台**：输入变量名或表达式查看值

6. 调试示例

假设你想调试凯撒密码的加密过程：

```
// 在 caesar.cpp 中
std::string Caesar::encrypt(const std::string& text, int shift) {
    std::string result = text;  // 在这一行设置断点

    for (size_t i = 0; i < result.length(); i++) {
        result[i] = shiftChar(result[i], shift);  // 也可以在这里设置断点
    }

    return result;
}
```

调试步骤：

1. 在 `std::string result = text;` 行设置断点
2. 按 `F5` 启动调试，运行程序
3. 在程序中执行加密命令： `encrypt "Hello"`
4. 程序会在断点处暂停
5. 查看"变量"面板，可以看到 `text` 和 `shift` 的值
6. 按 `F10` 单步执行，观察 `result` 的变化
7. 按 `F11` 进入 `shiftChar` 函数内部，查看字符转换过程

7. 常见调试场景

场景 1：程序崩溃

- 启用"所有异常时中断"选项
- 调试器会在异常发生的准确位置停止
- 查看调用堆栈找出问题根源

场景 2：变量值不符合预期

- 在变量赋值处设置断点
- 使用"监视"面板添加复杂表达式
- 单步执行观察值的变化

场景 3：无限循环

- 在循环内设置断点
- 查看循环变量的值
- 检查循环退出条件

8. VSCode 调试技巧

- **条件断点**：右键断点 → "编辑断点" → 设置条件（如 `i == 5`）
- **日志点**：不中断程序，仅输出消息到调试控制台
- **内联值显示**：调试时代码中直接显示变量值
- **数据断点**：监视变量值的改变

使用命令行 GDB（Linux/macOS，进阶）

如果你不用 VSCode 或需要远程调试：

```
# 编译时添加调试信息
g++ -g -o cipherx src/*.cpp

# 使用 gdb 调试
gdb ./cipherx

# GDB 基本命令
(gdb) run              # 运行程序
(gdb) break main       # 在 main 函数设置断点
(gdb) break file.cpp:10  # 在指定文件的第 10 行设置断点
(gdb) next             # 单步执行（不进入函数）
(gdb) step             # 单步执行（进入函数）
(gdb) print variable   # 打印变量值
(gdb) display variable # 每次停止时自动打印变量
(gdb) continue         # 继续运行
(gdb) quit             # 退出
```

打印调试（简单但有效）

```
#include <iostream>

#ifdef DEBUG
#define LOG(x) std::cout << "[DEBUG] " << x << std::endl
#else
#define LOG(x)
#endif

// 使用
LOG("Variable value: " << myVar);
```

编译时添加 `-DDEBUG` 启用调试输出。

3.5 性能优化

第一版不需要过度优化

- 代码清晰可读比性能更重要
- 先确保功能正确，再考虑优化

基本优化原则

1. **避免不必要的字符串拷贝**

   ```
   // 不好
   void process(std::string str) { }

   // 好
   void process(const std::string& str) { }
   ```

2. **使用合适的数据结构**
   - 频繁查找：使用 `std::map` 或 `std::unordered_map`
   - 顺序遍历：使用 `std::vector`
3. **预分配内存**

   ```
   std::string result;
   result.reserve(text.length());  // 预分配空间
   ```

四、开发流程建议

4.1 迭代式开发

版本 0.1.0：基本框架 + 凯撒密码

- 实现 CLI 交互框架
- 实现凯撒密码加密/解密
- 编写基本文档

版本 0.2.0：添加更多算法

- 实现替换密码
- 实现维吉尼亚密码
- 支持 JSON 配置文件

版本 0.3.0：文件操作

- 支持从文件读取文本
- 支持保存结果到文件

版本 0.4.0：用户体验优化

- 命令历史记录
- Tab 自动补全
- 更友好的错误提示

版本 1.0.0：正式发布

- 完善文档
- 多平台测试
- 发布二进制文件

4.2 版本控制（Git）

```
# 初始化仓库
git init

# 添加 .gitignore
echo "build/" >> .gitignore
echo "*.o" >> .gitignore
echo "*.exe" >> .gitignore
echo "cipherx" >> .gitignore

# 提交代码
git add .
git commit -m "Initial commit: project structure"

# 创建开发分支
git checkout -b develop

# 功能完成后合并到主分支
git checkout main
git merge develop

# 打标签
git tag -a v0.1.0 -m "Version 0.1.0: Basic Caesar cipher"
```

4.3 测试建议

手动测试

创建测试用例文档，每次修改后逐一测试：

测试用例 1：加密英文
输入：encrypt "Hello World"
期望输出：Khoor Zruog

测试用例 2：解密
输入：decrypt "Khoor Zruog"
期望输出：Hello World

测试用例 3：中文字符
输入：encrypt "你好世界"
期望输出：你好世界（保持不变）

自动化测试（进阶）

可以使用 Google Test 框架，但初学者可以先专注于功能实现。

五、示例配置文件

5.1 字符映射表（mapping.json）


```json
{
  "name": "Custom Substitution Cipher",
  "description": "A custom character mapping",
  "mapping": {
    "a": "z",
    "b": "y",
    "c": "x",
    "d": "w",
    "e": "v",
    "f": "u",
    "g": "t",
    "h": "s",
    "i": "r",
    "j": "q",
    "k": "p",
    "l": "o",
    "m": "n",
    "n": "m",
    "o": "l",
    "p": "k",
    "q": "j",
    "r": "i",
    "s": "h",
    "t": "g",
    "u": "f",
    "v": "e",
    "w": "d",
    "x": "c",
    "y": "b",
    "z": "a"
  }
}
```

六、常见问题（FAQ）

Q1: 我应该用 C 还是 C++?

A: 推荐使用 C++。C++ 的 `std::string` 和 STL 容器能大大简化开发，比 C 的字符数组更安全易用。

Q2: 需要使用面向对象编程吗?

A: 第一版可以不用。使用简单的函数和命名空间即可。随着项目复杂度增加，可以逐步引入类和对象。

Q3: 如何确保代码质量?

A:

- 编写清晰的注释
- 使用有意义的变量名
- 保持函数简短（每个函数只做一件事）
- 多测试，包括边界情况

Q4: 项目可以用多久完成?

A:

- 基础版（凯撒密码）：1-2 天
- 添加 3-4 种算法：3-5 天
- 完善功能和文档：1-2 周

Q5: 学习资源推荐?

A:

- **C++ 入门**：《C++ Primer》
- **命令行工具开发**：参考其他 CLI 工具源码（如 `git`， `curl`）
- **加密算法**：维基百科，《图解密码技术》
- **项目管理**：GitHub 官方文档

七、下一步行动

1. **立即开始**
   - 创建项目目录结构
   - 编写最小可运行版本
   - 提交到 GitHub
2. **第一周目标**
   - 实现 CLI 交互框架
   - 实现凯撒密码
   - 编写 README
3. **第一个月目标**
   - 实现 3-5 种加密算法
   - 支持文件操作
   - 发布 v0.1.0
4. **长期规划**
   - 收集用户反馈
   - 持续添加新功能
   - 优化性能和用户体验

八、附录

8.1 推荐的代码风格

```
// 文件头注释
/**
 * @file caesar.cpp
 * @brief Implementation of Caesar cipher
 * @author Your Name
 * @date 2024-01-01
 */

// 函数注释
/**
 * @brief Encrypt text using Caesar cipher
 * @param text The text to encrypt
 * @param shift The shift amount
 * @return Encrypted text
 */
std::string Caesar::encrypt(const std::string& text, int shift) {
    // 实现...
}
```

8.2 有用的命令行参数解析

如果需要支持类似 `cipherx encrypt -f input.txt -o output.txt` 的命令行参数：

```cpp
// 简单的参数解析示例
int main(int argc, char* argv[]) {
    if (argc > 1) {
        // 命令行模式
        std::string command = argv[1];
        if (command == "encrypt" && argc >= 3) {
            std::string text = argv[2];
            // 处理加密...
        }
    } else {
        // 交互模式
        // 进入 REPL...
    }
}
```

8.3 README 模板

```
# CipherX

一个交互式命令行加密/解密工具。

## 特性

- 支持多种加密算法
- 交互式命令行界面
- 跨平台支持（Windows/Linux/macOS）

## 安装

### 从源码编译

\`\`\`bash
git clone https://github.com/yourusername/cipherx.git
cd cipherx
make
\`\`\`

### 下载预编译版本

从 [Releases](https://github.com/yourusername/cipherx/releases) 下载对应平台的版本。

## 使用方法

\`\`\`bash
$ cipherx
cipherx> help
cipherx> encrypt "Hello World"
\`\`\`

## 许可证

MIT License
```

总结

这份指南为你提供了从零开始开发 CipherX CLI 工具的完整路线图。记住：

1. **从简单开始**：先实现一个可运行的最小版本
2. **迭代开发**：逐步添加功能，不要一次做太多
3. **重视文档**：良好的文档让项目更容易维护
4. **多测试**：每次修改后都要测试
5. **持续学习**：遇到问题多查资料，多参考优秀项目

祝你开发顺利！如有问题，欢迎随时查阅本指南。

文档版本：1.0
最后更新：2024-01-01
维护者：SmlCoke