**Level1终端原始模式**

文本编辑器需要对用户输入进行精确控制，这与我们平时使用的命令行程序截然不同。在正常的"熟模式"（cooked mode）下，终端会进行各种预处理：缓冲整行输入、处理退格键、响应Ctrl+C等信号。但文本编辑器需要即时响应每个按键，这就需要切换到"原始模式"（raw mode）。

这个阶段体现了系统编程的核心思想 - 理解并控制底层资源。我们需要理解终端的工作原理，学会使用termios结构体来配置终端行为。同时，资源管理也是关键 - 我们必须确保程序退出时恢复原始设置，避免用户的终端处于不可用状态。

错误处理在这个阶段尤为重要，因为系统调用可能因为各种原因失败。采用防御性编程的思想，对每个可能失败的操作都进行检查和处理。

**如何实现题目中所说的：在正常的"熟模式"（cooked mode）下，终端会进行各种预处理：缓冲整行输入、处理退格键、响应Ctrl+C等信号。但文本编辑器需要即时响应每个按键，这就需要切换到"原始模式"（raw mode）。？**

**tcgetattr函数**

头文件：#include <termios.h>

函数原型：int tcgetattr(int fd, struct termios \*termios\_p);

|  |  |  |
| --- | --- | --- |
| fd | int | 文件描述符（这里用 STDIN\_FILENO 表示标准输入） |
| optional\_actions | int | 控制何时应用新设置（关键选项见下表） |
| termios\_p | struct termios\* | 包含终端设置的结构体指针 |

| 选项常量 | 值 | 行为 |
| --- | --- | --- |
| TCSANOW | 0 | 立即应用设置 |
| TCSADRAIN | 1 | 等待所有输出完成后应用 |
| TCSAFLUSH | 2 | 等待输出完成 + 清空输入缓冲区 |

说明：tcgetattr函数用于获取与终端相关的参数。参数fd为终端的文件描述符，返回的结果保存在termios结构体中

**termios 结构体**

1.终端的行为控制器：通过标志位控制输入/输出处理方式

2.原始模式的关键：通过禁用规范模式实现即时输入

3.系统编程基础设施：所有终端交互程序的基础

4.资源管理范例：需要遵循"获取-修改-恢复"的模式

**Level2：原始输入和输出处理**

这个阶段我们开始构建用户界面的基础设施。与图形界面不同，终端界面需要我们手动控制每一个细节：光标位置、屏幕清除、文本渲染等。这里的核心思想是抽象化低层细节，将复杂的ANSI转义序列封装成简单易用的函数。

缓冲输出是这个阶段的重要概念。直接向终端写入会导致闪烁和性能问题，因此我们建议采用append buffer的设计模式：先将所有输出内容收集到缓冲区，然后一次性写入(避免出现中间态,用户要么看到完整的新状态，要么看到完整的旧状态)。这体现了批处理优化的思想(也可以理解成原子化地更新屏幕)。

特殊键处理建议使用有限状态机的设计思想。不同的转义序列需要通过状态来区分，比如箭头键的序列是ESC[A、ESC[B等，我们需要逐步解析这些序列。

**LinuxC语言基础知识：**

**如何实现系统调用的io操作？**

**open/close：暂略**

**读取与写入一个文件的数据：**

**读取：read（）：**记得free掉内存

**写入：write（）：**

**编辑器选择 write 的原因：**

1. 需要精确控制每个字节（如转义序列）

2. 避免标准库缓冲区的干扰

3. 确保即时输出（无缓冲）

fd：文件描述符 \*buf：文件里读到的存入内存中 count：读取的字节数

**如何获取终端窗口大小以得到所需行、列数指针？**

**ioctr函数**

查询终端屏幕尺寸以确定行指针与列指针的方法：

通过 ioctl 系统调用查询终端尺寸

**Ioctr：**三个参数：1、文件描述名指针 2、iocontrol命令 3、对应命令的参数

也可以用write（）函数来实现功能

**Enum函数：**

枚举函数的使用：更加直观地表现数组之中的内容

**Atexit函数：**

确保在程序退出时恢复终端原始设置，退出函数，以另外一个函数作为参数，如果本程序退出（不论是否为正常退出，均执行参数函数内容）

**Unix/Linux 系统中的几个标准文件描述符常量：**

表示标准输入输出流（Standard Output）。

文件描述符：

|  |  |  |
| --- | --- | --- |
| 文件描述符 | 宏常量 | 用途 |
| 0 | STDIN\_FILENO | 标准输入（键盘） |
| 1 | STDOUT\_FILENO | 标准输出（屏幕） |
| 2 | STDERR\_FILENO | 标准错误（屏幕） |

**底层原理：**

在 Unix/Linux 系统中：

所有 I/O 操作都通过文件描述符（File Descriptor）完成

每个进程启动时会自动打开 3 个文件描述符：

0：输入 1：输出 2：错误

本质是内核维护的打开文件表的索引

**使用方法：**

**以编辑器中的清屏代码为例：**

write(STDOUT\_FILENO, "\x1b[2J", 4);

write(STDOUT\_FILENO, "\x1b[H", 3);

1. `"\x1b[2J"` 是 ANSI 转义序列，表示"清除整个屏幕"

`\x1b` = ESC 字符（ASCII 27）

`[2J` = 清除屏幕命令

2. `4` 表示写入 4 个字节（ESC + [ + 2 + J）

3. 直接写入 `STDOUT\_FILENO` 确保立即生效

**Level3：实现文本查看器**

这个阶段我们从底层的输入输出处理转向数据模型的设计。一个文本编辑器的核心是如何在内存中表示和操作文本数据。我们推荐选择行导向的数据结构，因为大多数文本编辑操作都是以行为单位进行的。

分离渲染逻辑是这个阶段的重要思想。原始文本（chars）和渲染文本（render）的分离，让我们能够处理Tab制表符、语法高亮等复杂的显示需求，而不影响原始数据的完整性。这体现了单一职责原则。

视口和滚动机制引入了MVC模式的思想 - 数据（文本内容）、视图（屏幕显示）和控制（用户输入）相互分离。用户看到的只是整个文档的一个"窗口"，我们需要管理这个窗口与底层数据的映射关系。

状态栏的设计体现了信息反馈的用户体验原则。用户需要知道当前的状态：在哪个文件、第几行、文件是否被修改等。这些信息有助于用户建立心理模型。

**渲染**

渲染（Rendering） 在文本编辑器中的含义是将原始文本内容转换为适合屏幕显示的形式。这个过程主要解决两个问题：

1.特殊字符处理：比如将制表符(\t)转换为适当数量的空格

2.显示优化：准备可直接输出到屏幕的字符序列

**目标：**

显示文件内容并支持滚动

**实现思路：**

建立行存储结构

实现视口管理

添加滚动逻辑

**实现步骤：**

首先通过对行数据结构的进行定义，以将原始文本与渲染文本进行分离：typedef struct erow

而后我们通过文件加载来显示窗口：editorOpen（）

**Level 4: 实现文本编辑器**

这是从"只读查看器"到"可编辑器"的关键跃升。这个阶段的核心思想是数据一致性和用户体验的平衡。每次编辑操作都可能影响多个数据结构：文本内容、光标位置、渲染缓存、脏标志等，我们需要确保这些状态始终保持同步。

原子操作的概念在这里非常重要。每个编辑操作（插入字符、删除字符、插入行等）都应该是原子的 - 要么完全成功，要么完全失败，不能留下不一致的中间状态。

脏标志模式体现了状态管理的设计思想。我们需要追踪文档是否被修改，这不仅影响用户界面的显示，还关系到文件保存和退出确认的逻辑。

用户确认机制体现了防错设计的思想。当用户执行可能导致数据丢失的操作时（如退出未保存的文档），系统应该提供适当的警告和确认机制。

内存管理在这个阶段变得更加重要。频繁的插入和删除操作需要动态调整内存分配，我们需要考虑内存效率和操作性能的平衡。

**通过行操作实现本阶段的内容，通过以下函数实现：**

1.将制表符转换为空格，并计算渲染后的位置editorUpdateRow

2.在指定位置插入一行editorInsertRow

3.释放行内存editorFreeRow

4.删除指定行editorDelRow

5.在行中插入字符editorRowInsertChar

6.删除行中的字符editorRowDelChar

**而编辑器操作则是以行操作为基础进行实现：**

1.在光标位置插入字符editorInsertChar(int c)

2.插入新行editorInsertNewline()

3.删除字符（退格键）editorDelChar()

4.移动光标editorMoveCursor(int key)

**memmove函数**

memmove函数是C语言标准库中的一个函数，用于从一个内存位置复制数据到另一个内存位置。这个函数在处理重叠内存区域时特别有用，因为它能够在源区域被覆盖之前将重叠区域的字节拷贝到目标区域中。如果没有重叠，memmove的行为与memcpy相同。

函数声明和参数：

memmove函数的声明如下：

void \*memmove(void \*str1, const void \*str2, size\_t n)

其中，参数str1是指向目标数组的指针，str2是指向数据源的指针，而n是要复制的字节数。

返回值：

函数返回一个指向目标存储区str1的指针。

**总结**

**代码运行流程举例分析：**

**场景：用户输入字符'a'并回车**

main() 进入主循环

editorReadKey() 检测到'a'键

editorInsertChar('a') 被调用

调用 editorRowInsertChar()

触发 editorUpdateRow()

editorReadKey() 检测到回车键

editorInsertNewline() 被调用

可能调用 editorInsertRow()

editorRefreshScreen() 重绘界面

调用 editorDrawRows() 输出内容

调用 editorDrawStatusBar() 更新状态

**设计模式分析**

1.观察者模式：主循环监听按键事件

2.装饰器模式：editorUpdateRow对原始内容进行渲染装饰

3.状态模式：通过E.dirty等标志管理编辑状态

1. MVC分离：

Model: erow结构存储数据

View: editorDraw\*系列函数

Controller: 主循环和编辑函数