PJ设计文档

题目1 MINI VIM编辑器

一、需求分析

1.normal模式下，需要实现：

①指令识别功能；

②相应的文件读写操作；

③字符串查找功能；

④退出功能；

⑤撤销与重做功能；

⑥翻页功能；

2.insert模式下，需要实现：

文本输入、处理与状态保存，关键在于如何处理输入、删除、换行等。

3.两个模式下都需要实现：

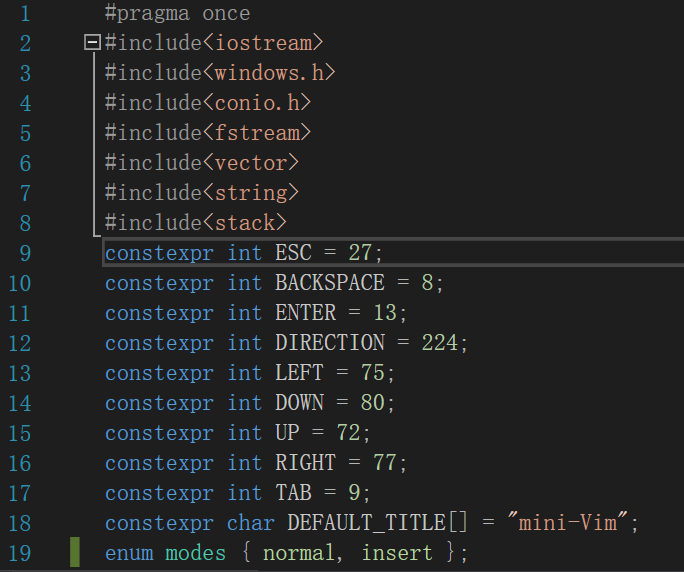
①将保存的文本显示到控制台缓冲区上

②实现光标的跳转（上下左右移动、删除、换行、normal模式下的命令）

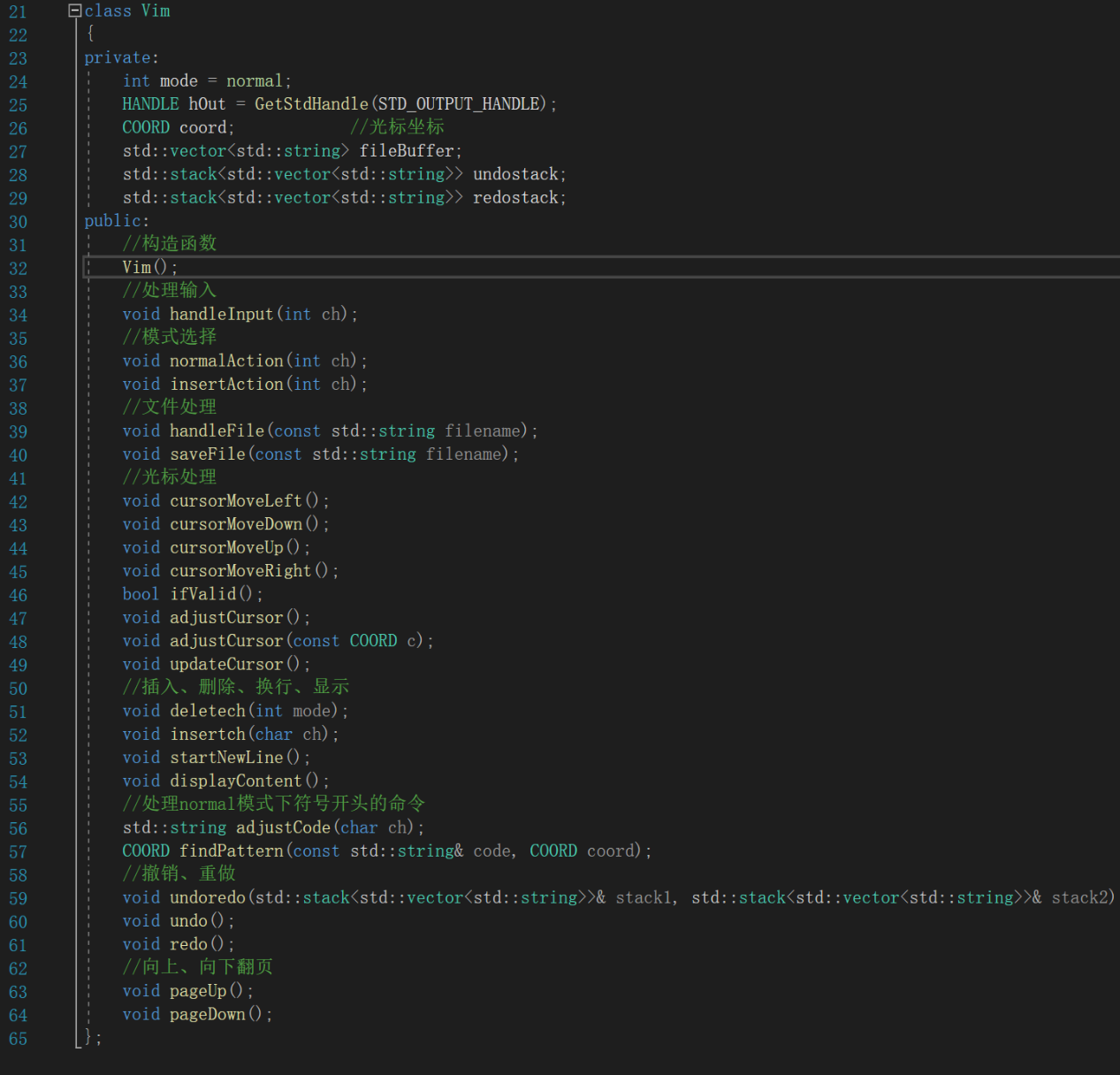
二、概要设计

基于上述分析，将整个程序分为三个cpp文件：InsertEditor.cpp（处理Insert模式）、NormalEditor（处理Normal模式）、miniVim.cpp（主程序），另用miniVim.h保存整个工程中需要的头文件、常量、类定义、相关函数接口。

1.miniVim.h



相应的头文件和常量定义



类Vim的定义，包含了对应的类函数的声明，相关功能已给出说明。

2.miniVim.cpp

(1)void handleinput(int ch)

根据当前的模式，将字符传递到对应模式下做进一步处理。

说明：根据当前mode的值用switch选择。

(2)void displayContents()

将存储在文本缓冲区中的内容显示在控制台缓冲区上。

说明：先清空控制台，再将文本输出到控制台上。

注意使用了printf而非cout，因为printf的效率远高于cout。

(3)void updateCursor()

将控制台缓冲区上显示的光标与内部实际的coord变量同步。

说明：用控制台函数SetConsoleCursorPosition设置。

(4)void adjustCursor()

调整当前光标的坐标，默认到当前文本的末尾。

说明：直接设置coord的值即可。

(5)void adjustCursor(COORD c)

调整当前光标的坐标到参数c指定的位置。

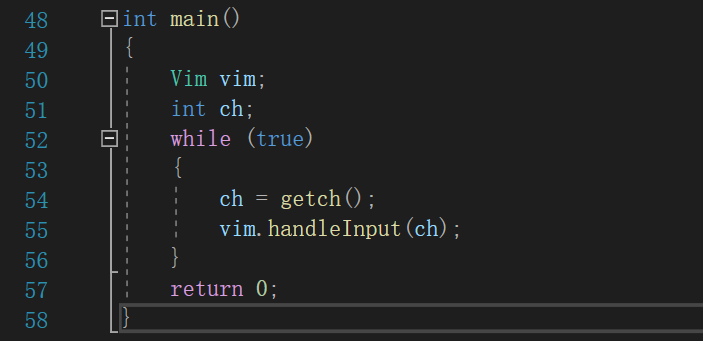
说明：直接设置coord的值即可。

(6)Vim()

构造函数，默认设置mode为normal，坐标为（0，0），文件创建一个空字符串，设置窗口标题为”mini-Vim”。

说明：设置窗口调用了控制台函数SetConsoleTitle。

整体流程：



创建Vim对象，进入循环，每次接受一个输入的字符，进行进一步处理。

说明：

这里用getch()接受输入，对比getchar()，两者的区别在于：

getch()每次只接受一个字符就结束，并且输入不在控制台缓冲区回显；

getchar()每次是从输入缓冲区中获得一个字符，并且输入会回显。

3.InsertEditor.cpp

(1)void cursorMoveLeft()

实现光标左移。

说明：只需要检测当前坐标是否大于0，是就将坐标的x值减一，否则不做任何操作。

(2)void cursorMoveDown()

实现光标下移。

说明：分为以下情况处理：

①当前光标正下方（即(x,y+1) )有文本：直接下移（++y）；

②当前光标正下方没有内容，且光标所在行不是最后一行：

将光标移到下一行的末尾。

(3)void cursorMoveUp()

实现光标上移。

说明：分为以下情况处理：

①当前光标正上方有文本：直接上移（--y）；

②当前光标正上方没有文本，且光标所在行不是第一行：

将光标移到上一行的末尾。

(4)void cursorMoveRight()

实现光标右移。

说明：只需要检测是否已经到当前行的末尾，还没到末尾就将坐标的x值加一。注意需要判断当前行是否已空。

(5)void startNewLine()

新建一行文本，加入文本缓冲区。

说明：分为以下情况处理：

①当前行在文本末尾：直接加入一行空文本（即””），并将光标下移。

②当前行不在末尾（包含光标在当前行内部和末尾的情况）：

将光标当前行分为光标左边和光标右边（包括光标）两部分，从当前行删去右边部分，调用vector的insert函数将右边部分插入到下一行。完成后，将光标下移，并将光标坐标的x设置为0，即将光标移动到新插入这行的行首。

(6)void deletech()

删除光标处的字符。

说明：若当前坐标为(x,y)，则我们应删去(x,y-1)处的字符，分为以下情况处理：

①x在当前行的内部（两边都有文本）

用[x,end)的内容替换[x-1,end)，效果即为删去了该字符并让该字符右侧的所有字符都左移。替换的具体实现为string的replace函数。最后让光标左移（仅insert模式下）

②x在行尾

直接用pop\_back()删去末尾，并让光标左移（仅normal模式下）。

③x在行首（仅normal模式下）

如果是在第一行则跳过，否则，将该行的所有内容添加到上一行，并删去本行的内容。添加具体实现为string的append函数，删除为vector的erase函数。将光标调整到上一行末尾（没有添加本行前）

④x超界

将光标调整回本行末尾。

(6)void insertch(char ch)

将给定的字符插入到光标位置。

说明：分为以下情况处理：

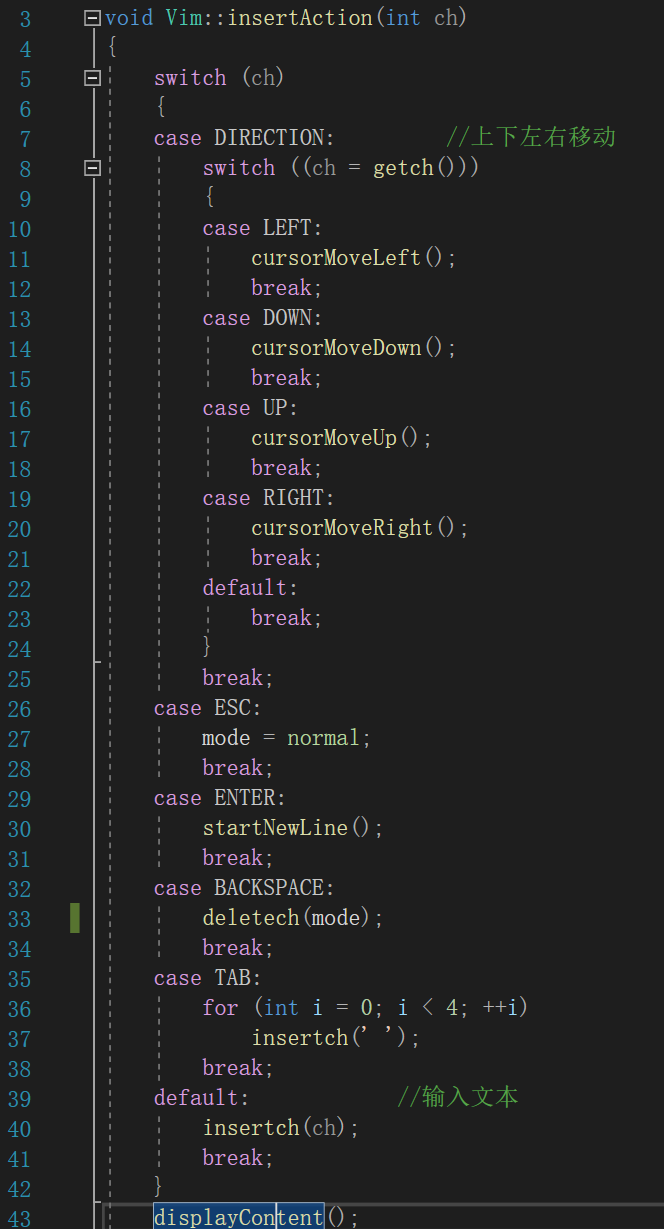
①若光标没有超出当前文本区

也分为两种情况，如果在本行内部，用stirng的insert函数插入，在末尾用push\_back直接插入。最后光标右移。

②光标超出当前文本区

需要新建一行，即文件缓冲区加入一空行。

整体流程：



主要流程在void insertAction(int ch)中，根据输入的字符选择对应的操作：

①当输入为方向键时，调用对应的光标移动函数。需要注意的是，方向键对应的ASCII码超出0~127的范围，所以用int保存ch更合适；另外，方向键实际上是两个ASCII码的组合，一个表示按下了方向键，另一个表示具体哪个方向（上下左右）。具体对应关系在miniVim.h中已给出。

②输入ESC时，切换模式。

③输入ENTER时，调用startNewLine函数。

④输入BACKSPACE时，调用deletech函数。

⑤输入TAB时，相当于输入四个空格。

⑥其余情况为输入正常字符，直接调用insertch函数插入到对应光标位置。

最后调用displayContent函数将内容回显在控制台缓冲区上，然后进入下一次循环。

4.NormalEditor.cpp

(1)void handleFile(const string filename)

打开filename对应的文件，如果成功打开则将文件内容加载到控制台缓冲区，否则什么都不做。

说明：文件的打开涉及到istream类的操作，创建一个istream类对象infile,使用infile.open(filename)打开对应文件，用infile.is\_open()判断是否成功打开。如果打开了，将文本缓冲区fileBuffer的内容清空，再用>>将文件内容输入到fileBuffer。终止条件为文件读到EOF。完成后，调整光标位置（adjustCursor()），并关闭文件（infile.close()）。

(2)void saveFile(const string filename)

将当前文本缓冲区（控制台缓冲区上的）内容保存到filename对应的文件，如果不存在该文件则会创建新文件，否则会覆盖该文件。

说明：文件的保存涉及到ostream类的操作，创建一个ostream类对象outfile，用filename初始化outfile（std::ostream outfile(filename)），然后将文本缓冲区fileBuffer的内容用<<输出到outfile对应的文件。完成后关闭文件（outfile.close()）。

(3)string adjustCode(char ch)

处理normal模式下以符号(:和/)开头的命令，将得到的命令返回。

说明：首先初始创建了一个CONSOLE\_SCREEN\_BUFFER\_INFO csbi，该类的对象存储了控制台窗口、光标相关信息，用GetConsoleScreenBufferInfo(hOut, &csbi)即可获得信息。将坐标coord的Y坐标调整到当前窗口底部（csbi.srWindow.Bottom），X坐标调整到0，输出符号后，X调整到1，然后更新光标(updateCursor())使显示与实际坐标同步。接下来读入指令，用一个string类的code保存指令，且code被初始化为之前输入的符号（:或/）。接下来基于以下情况处理：

①输入了BACKSPACE

需要删除命令行的一个字符，可以在InsertEditor.cpp的deletch()函数的基础上做对应的改动。注意由于命令行只有一行，不需要考虑跨行处理。

②输入了ENTER

结束输入，将得到的code返回。

③输入了方向键

由于方向键的ASCII码超界，用char读取为乱码，所以特殊处理，进入后用getch()接受方向键的第二个ASCII码，然后continue进行下一循环（表现为没有现象）。

④其余情况

即正常输入，参考InsertEditor.cpp的insertch()函数，不考虑跨行处理，基本是一样的。

完成本轮输入后，用displayContent()将原先的内容显示出来，将光标设置到底端最左部（这样保证每次code是从最左端开始的），用cout输出code。跳出循环后返回code。

(4)COORD findPattern(const string& code,COORD coord)

根据给定的光标位置和指定要查找的文本，从光标位置往后查找，返回对应文本开头的坐标。

说明：先根据code得到要查找的pattern，然后创建string text，用text记录当前fileBuffer的所有内容到一行方便查找（具体步骤为，将光标右边的内容和光标下面所有行的内容加入到text）。然后直接调用string的find方法在text中寻找pattern。找到后，则需要根据得到的text中的下标pos反推实际的二维坐标coord，只需要从0开始逐渐增加到pos，每次增加就让原本的坐标(x,y)对应后移一位（坐标对应位置后面还有内容时++x，否则x=0，++y）。得到结果后返回。

(5)void ifValid()

用于判断坐标是否越界以便调整。

说明：直接比较即可。

(6)void undoredo(stack<vector<string>>& stack1,stack<vector<string>>& stack2)

用于撤销重做。

说明：撤销和重做的实现方法为，直接用元素为fileBuffer对应类型的stack来存储相应的状态。用2个栈保存状态，undostack存储之前操作时保存的状态，用于撤销；redostack保存撤销前的状态，用于重做。

当调用该函数时，先将当前状态压入stack2，再将stack1的栈顶压出赋给当前状态。

(7)void undo()

用于撤销。

说明：调用undoredo(undostack,redostack)即可。

(8)void redo()

用于重做。

说明：调用undoredo(redostack,undostack)即可。

(9)void pageUp()

用于向上翻页。

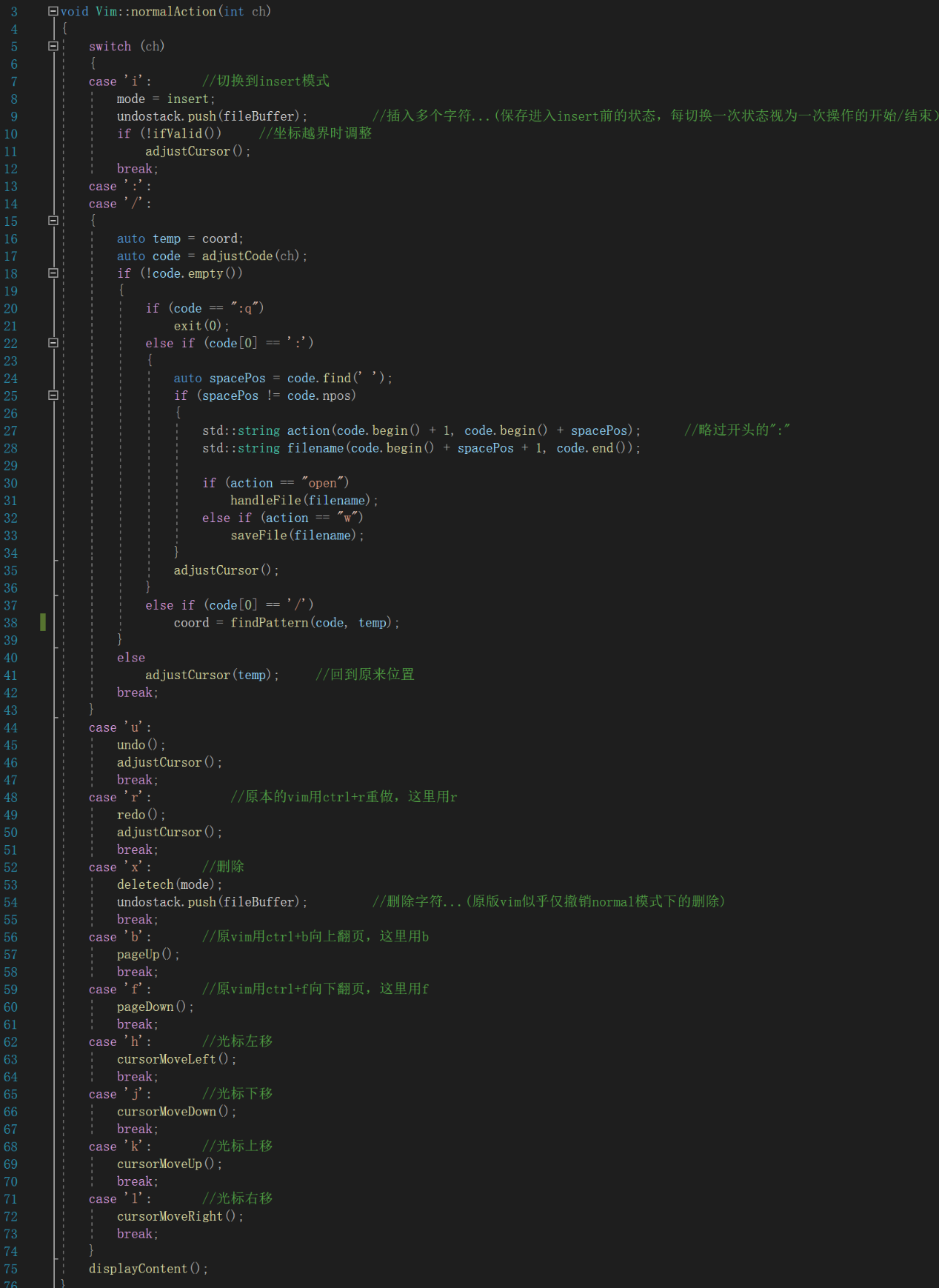
说明：直接调整光标。

(10)void pageDown()

用于向下翻页。

说明：直接调整光标。

整体流程：



主要流程在函数void NormalAction(int ch)中

分以下情况处理：

①输入i

切换到insert模式。此时需要将当前的状态保存到undostack中。坐标越界时调整坐标（虽然应该不会出现这种情况？）

②输入：或/

进入命令处理模式（调用adjustCode函数），注意先保存当前的坐标。然后检测获得的命令

(i)获得:q时，退出（exit(0)）。

(ii)获得:(不是:q)时，先根据空格的位置将命令分割为两部分，左边为action，右边为filename。根据action进一步处理：

若action=”open”，则调用handleFile(filename)打开对应文件。

若action=”w”，则调用saveFile(filename)保存到对应文件。

结束后调整光标位置。若检测到命令为空，则光标回退到原先的位置。（比如输入”:”，然后用BACKSPACE删除，此时光标直接回到原先位置）

③输入u

调用undo()撤销并调整光标。

④输入r

调用redo()撤销并调整光标。

⑤输入x

删除光标处字符，调用deletech()，并将删除后的状态加入到undostack。

⑥输入b、f

调用对应翻页函数进行上下翻页。

⑦输入h、j、k、l

调用对应光标移动函数移动。

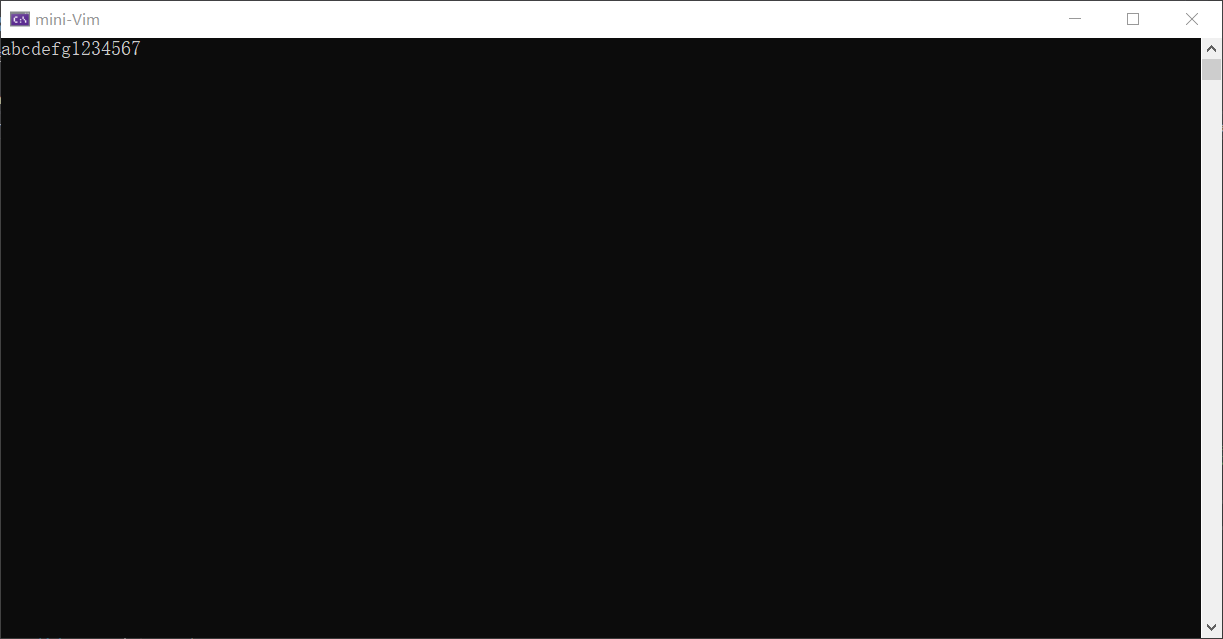
最后调用displayContent()将内容显示出来。

三、详细设计

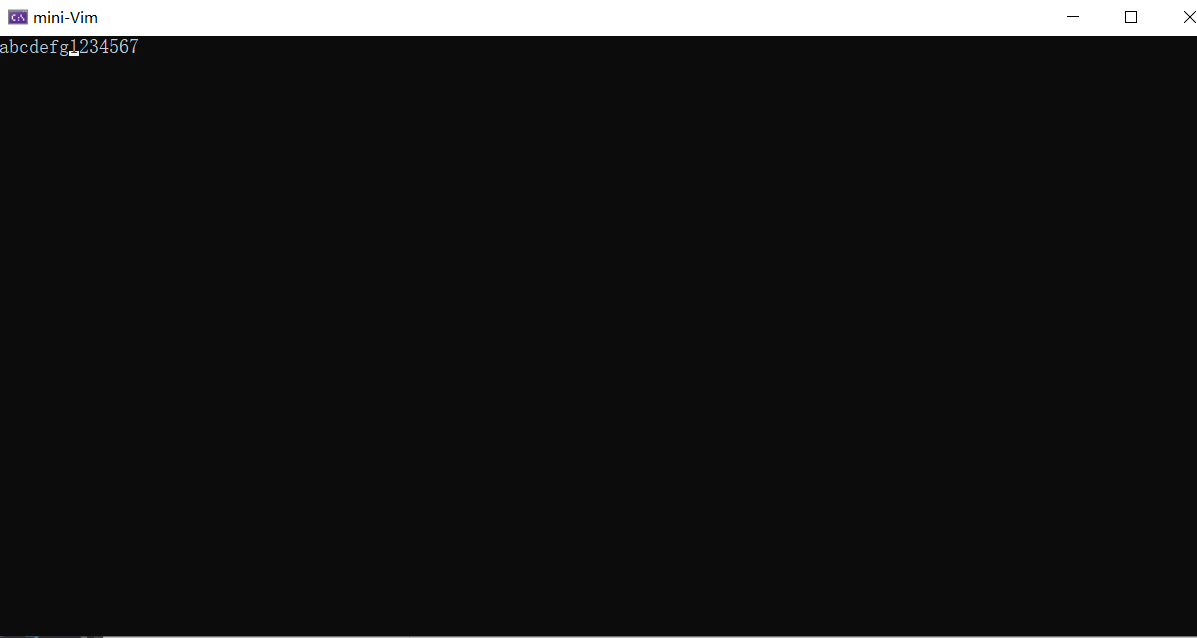
见提交的源代码。

四、调试分析

测试（一）：用insert模式输入一段文字



移动光标



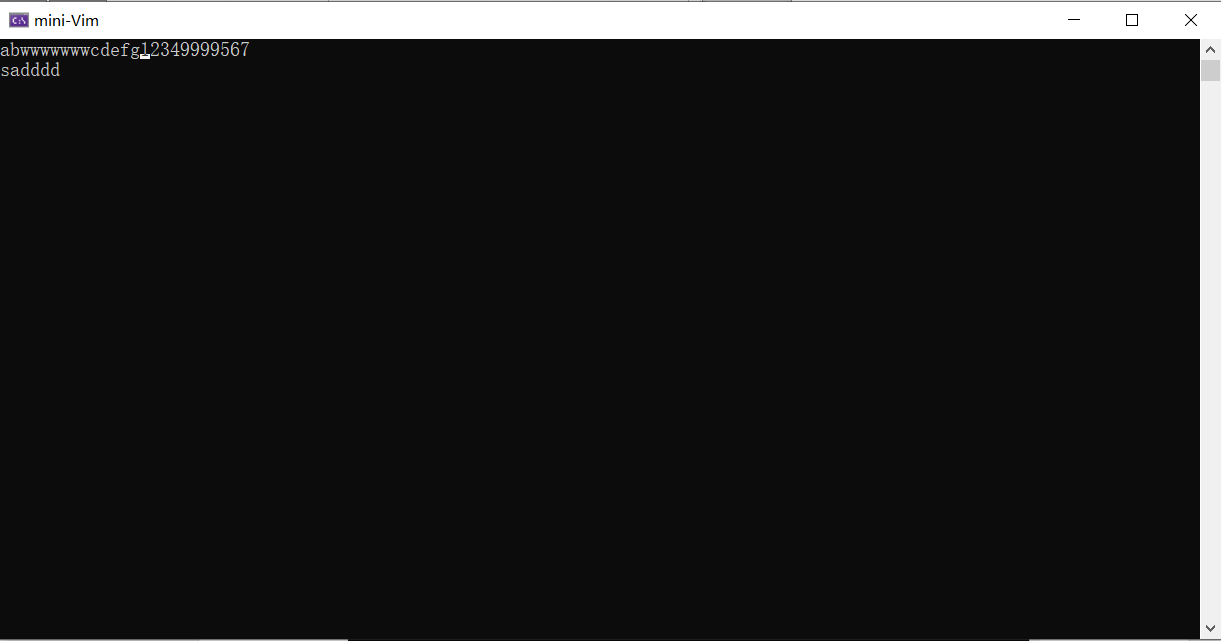
用回车键分割文字



利用上下左右，再输入一些文字



退格



总结：时间复杂度分析：

设行数为c，每行字符数n，则：

插入：O(n)（尾部O(1)）

回车：O(c+n)（最后一行O(1)）

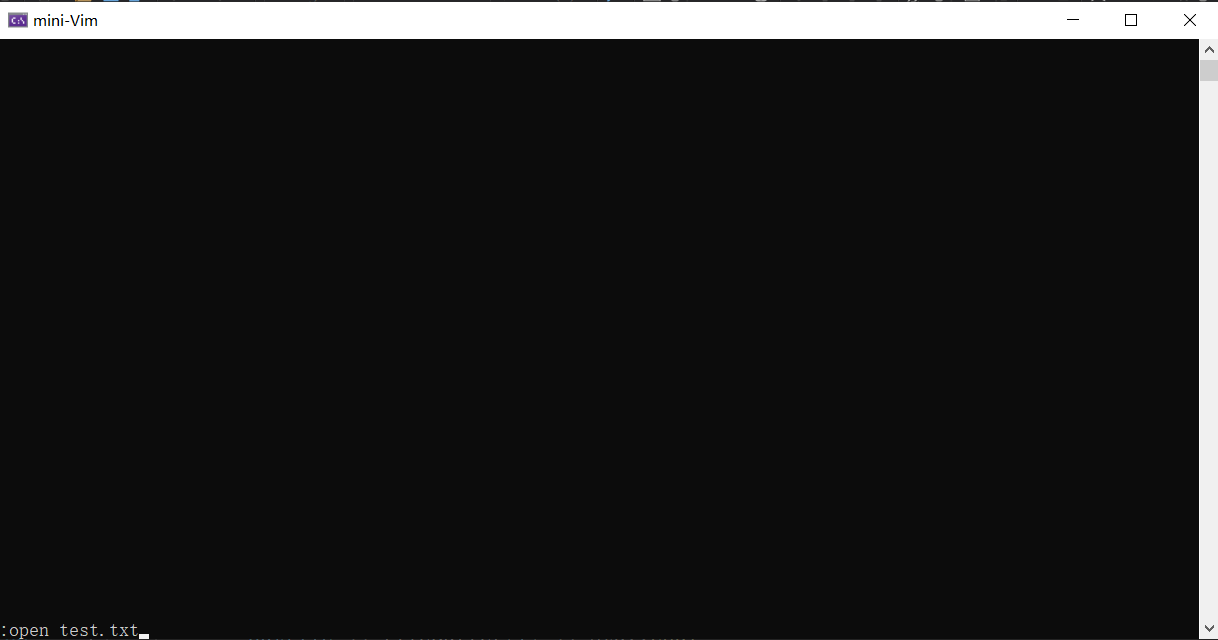
删除：O(n)（尾部O(1)，涉及换行时O(c+n)）

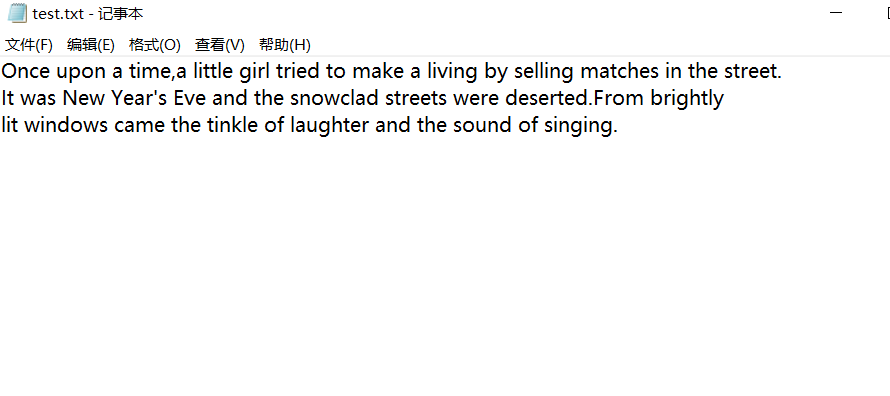
移动光标：O(1)

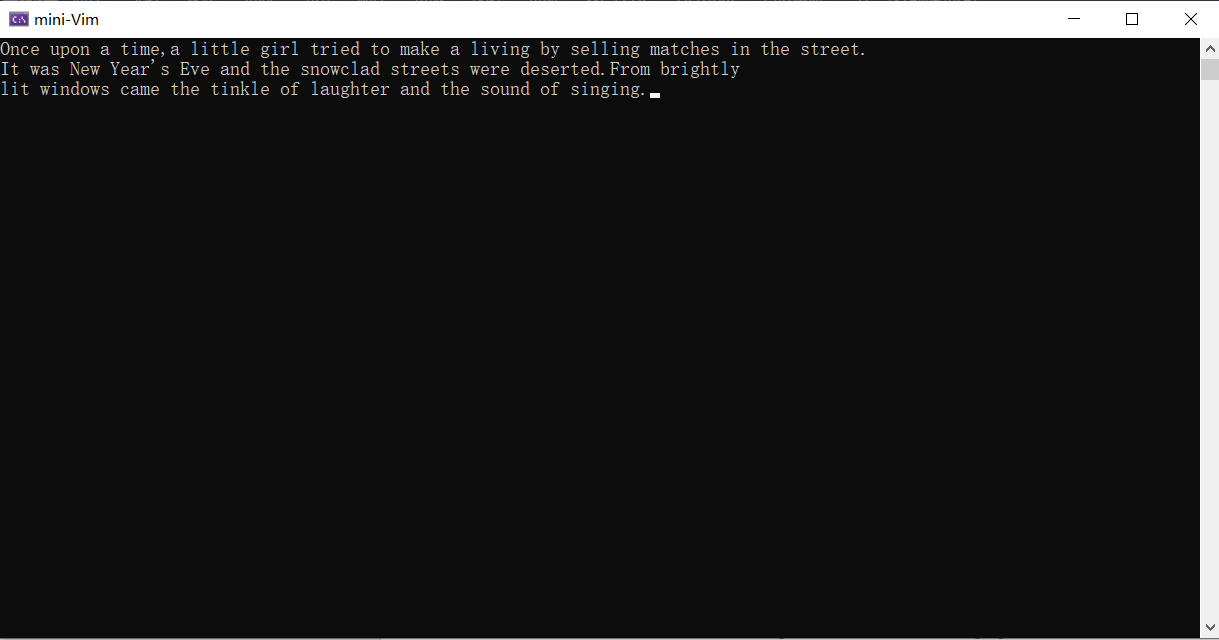
将内容写入控制台缓冲区：O(c)

问题：主要效率问题在于内容写入的操作上，每次都要执行一次内容写入。因为想将已经写入缓冲区的文字删除我只想到2种办法，要么使用system(“CLS”)清屏再全部输出，要么使用退格的方法，用空格键覆盖。前一种方法效率上略低，后一种方法处理起来麻烦得多（需要将后面的所有字符前移，乃至跨行处理等），我也没想好要怎么具体处理，就没有采用。

测试（二）：打开文件test.txt



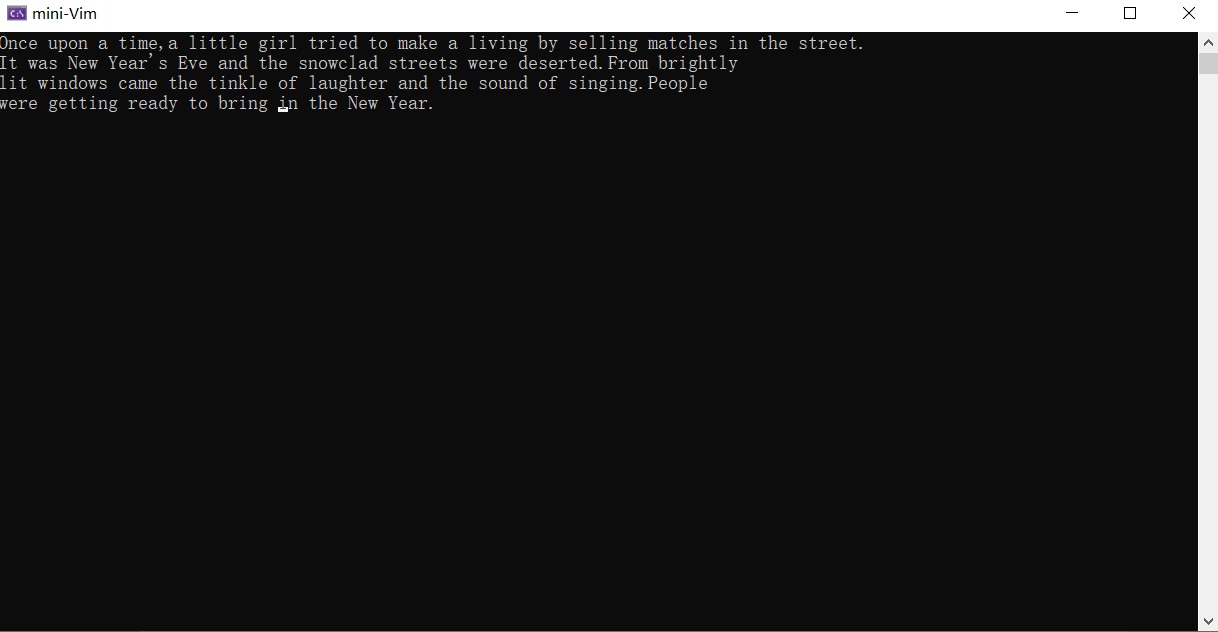


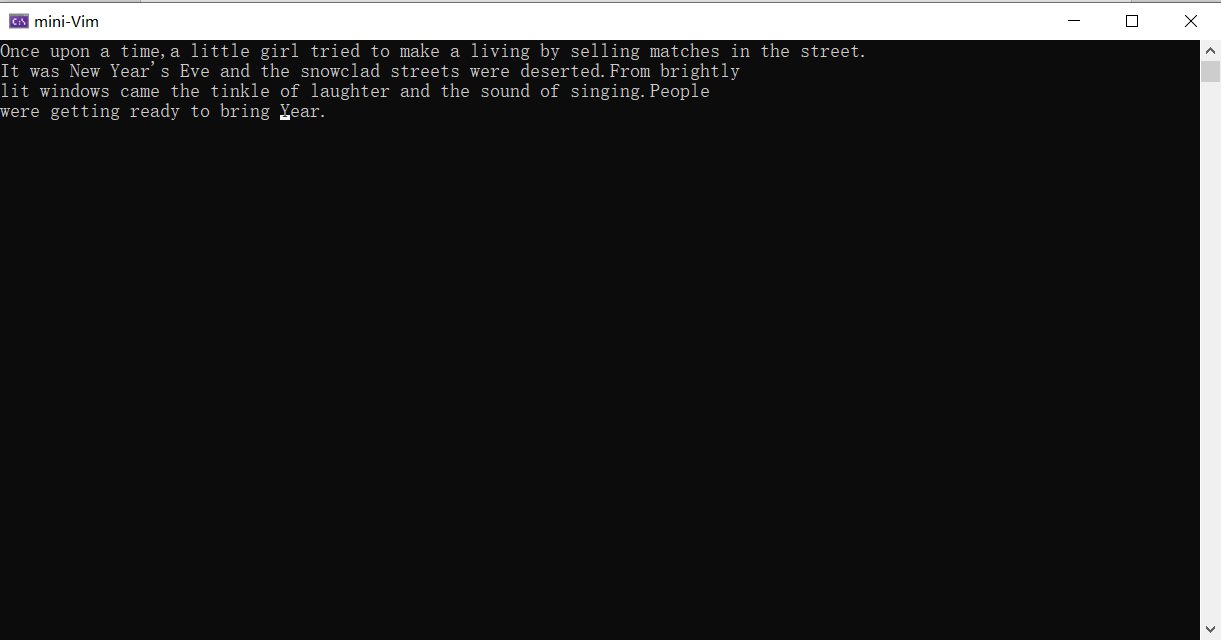


调到insert模式输入一些内容

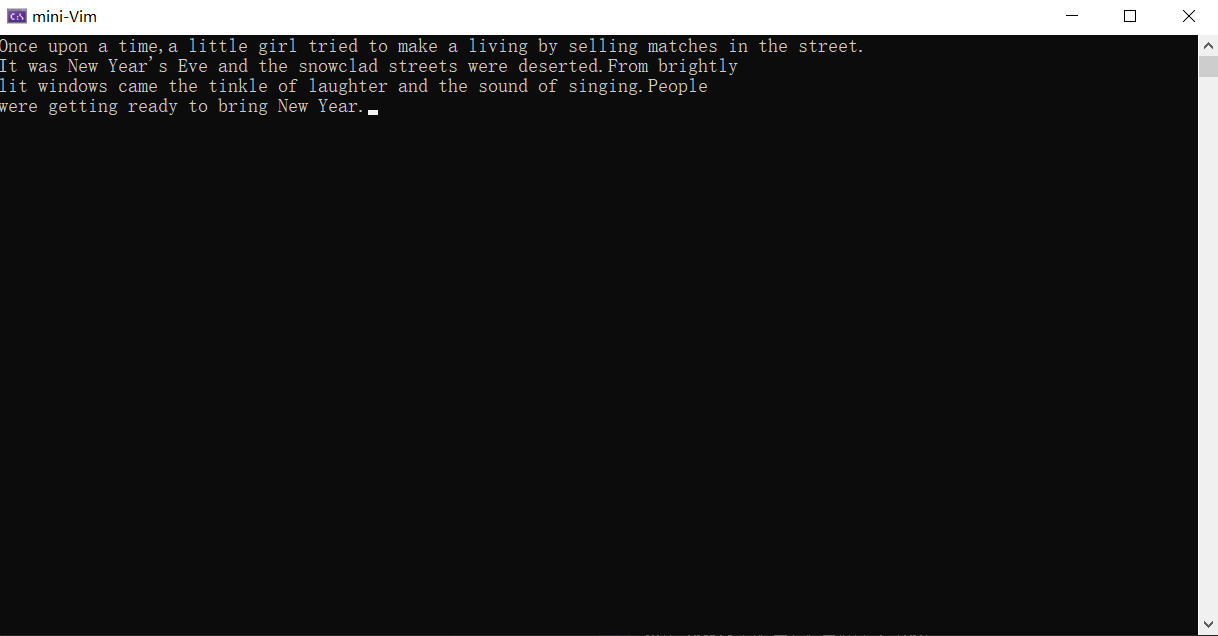


移动光标，用x删除一些内容

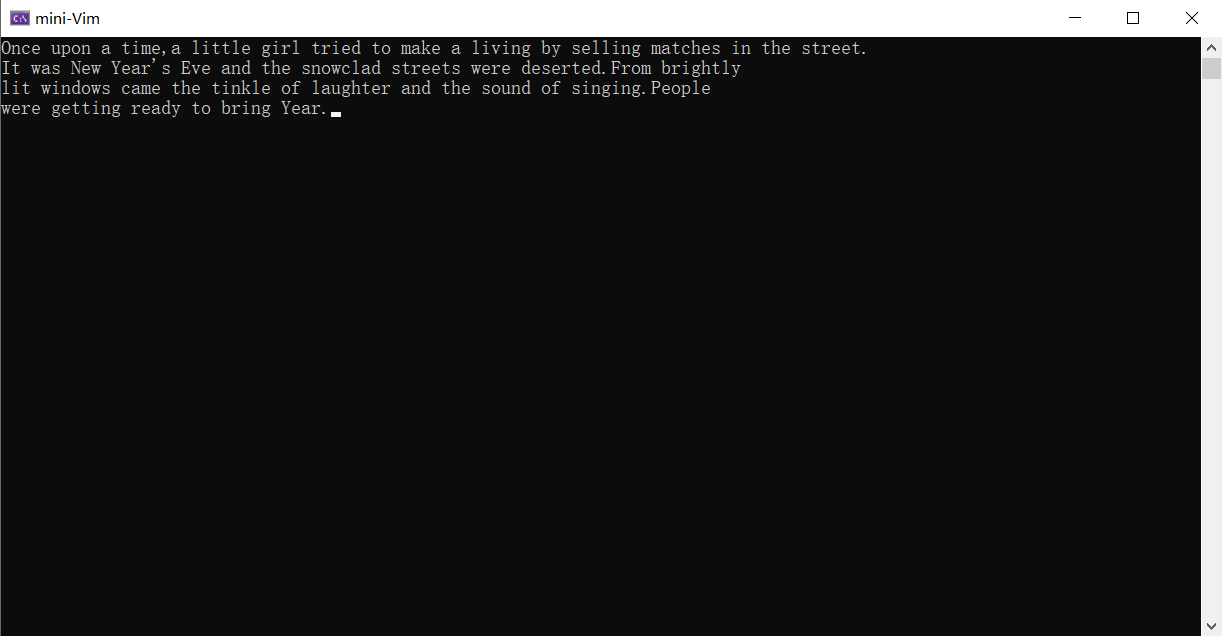




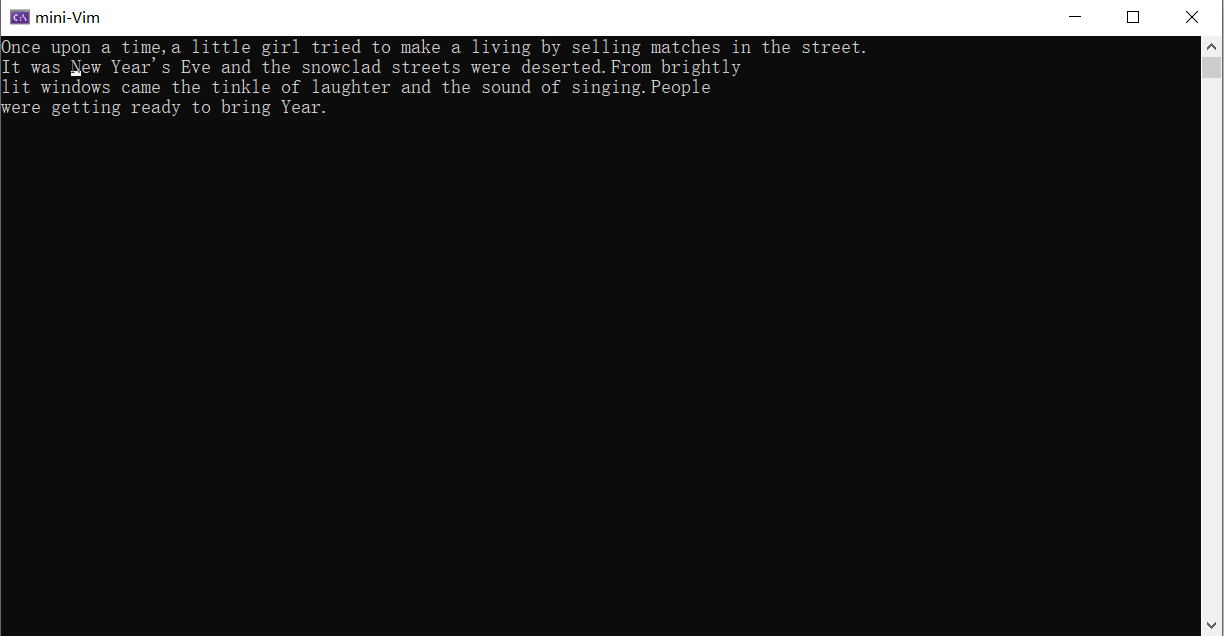
用u撤销一定次数

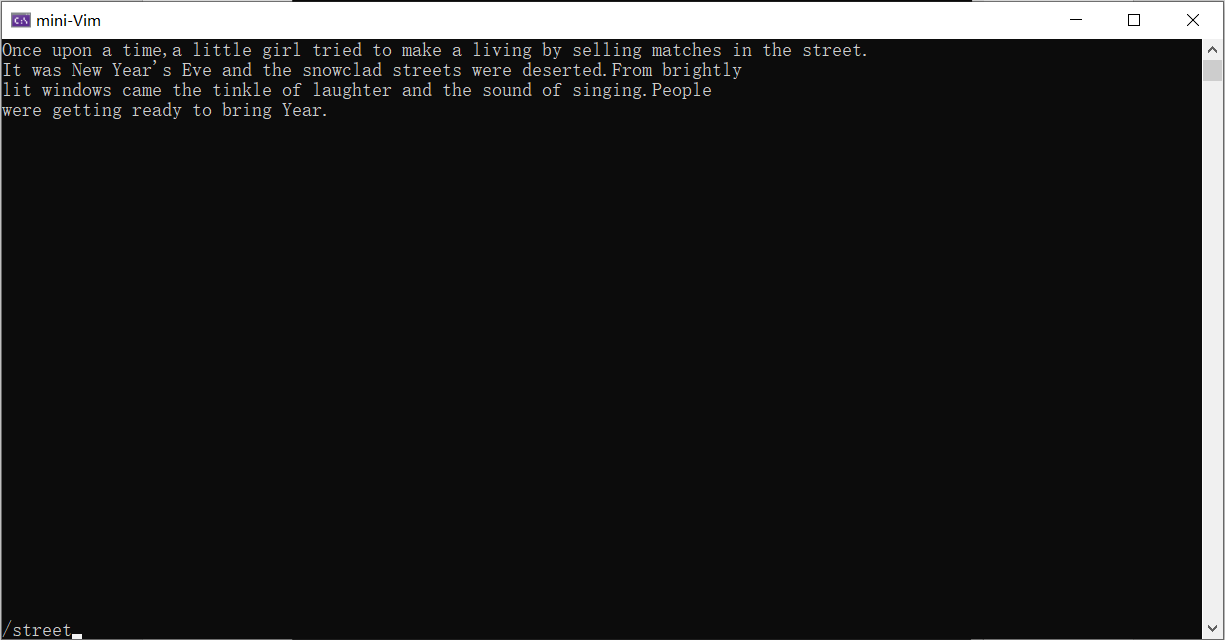


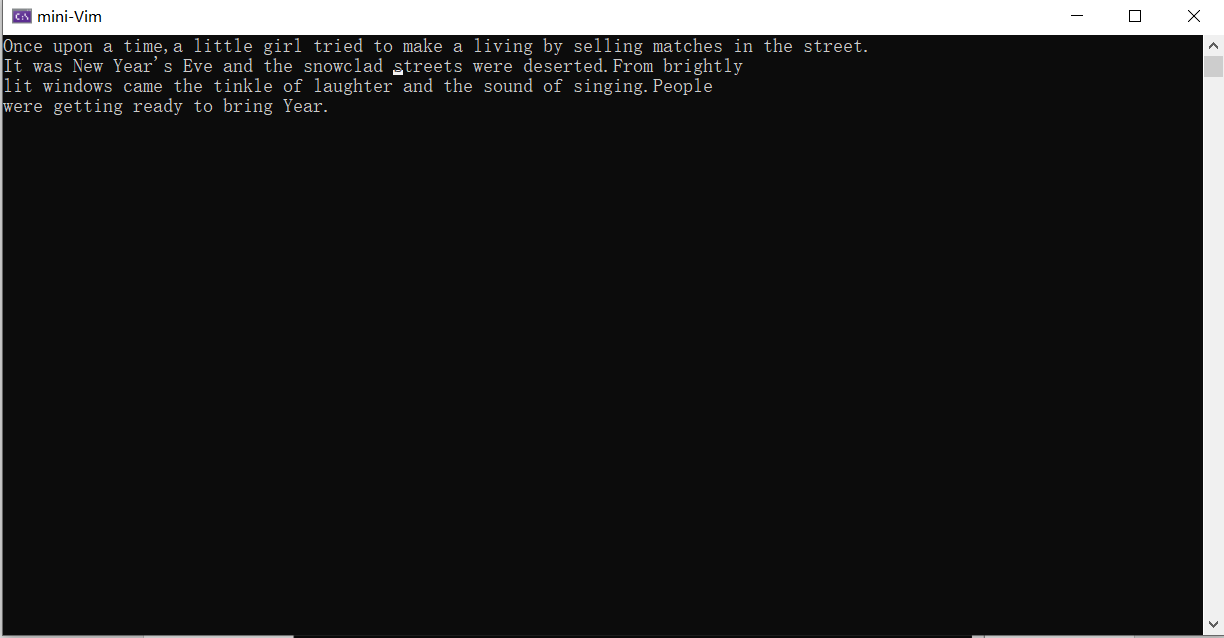
用r重做



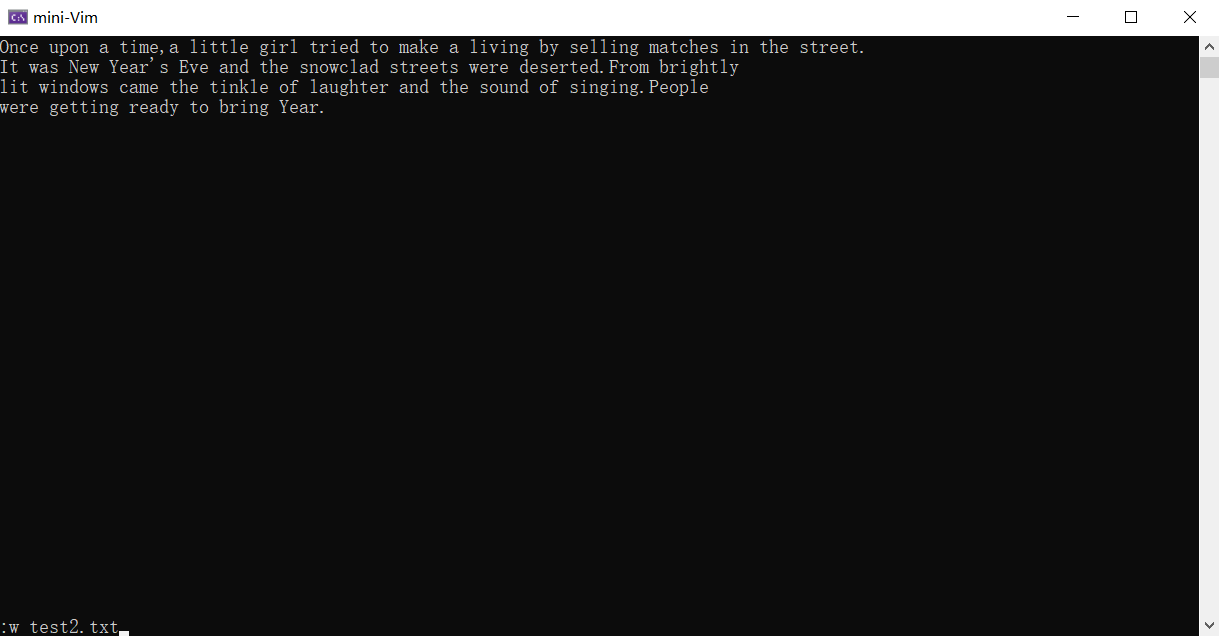
用/查找文本

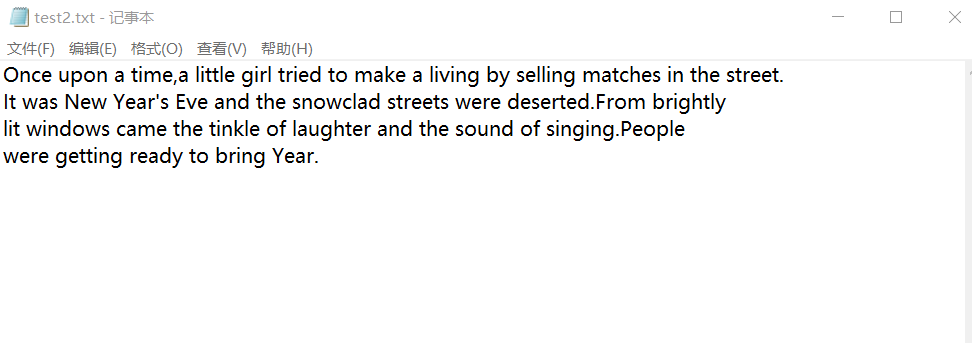




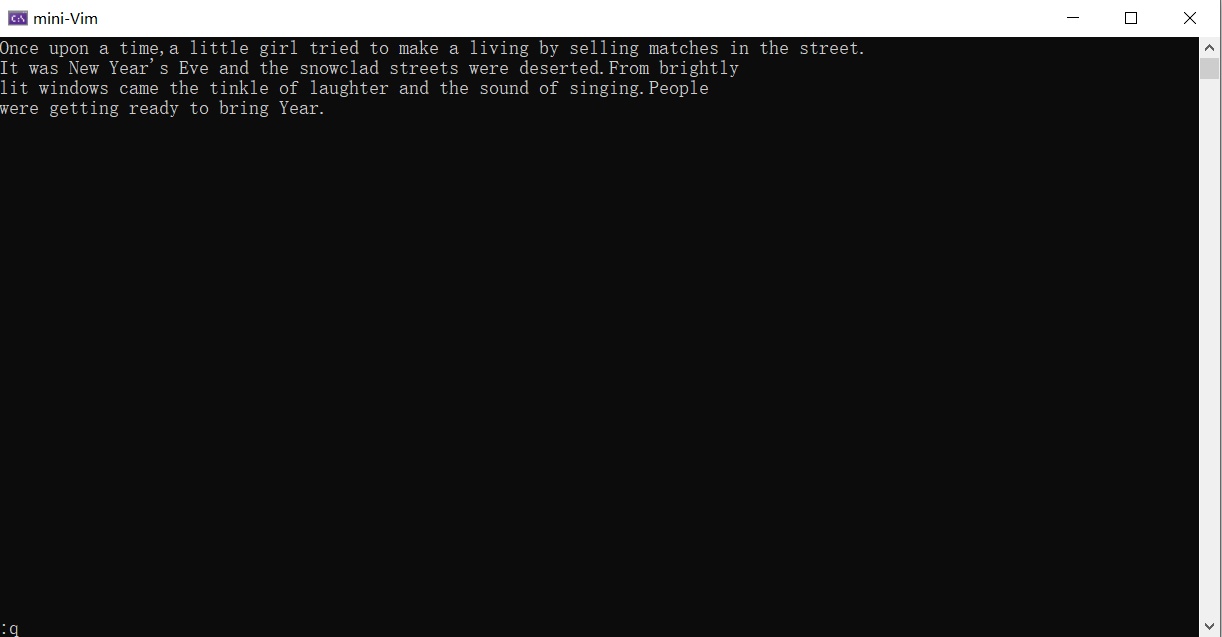


用:w保存到test2.txt





用:q退出





总结：设文件有c行，一行字符数n

打开文件：O(c)

保存文件：O(c)

删除：O(n)（尾部O(1)，涉及换行O(c+n)）

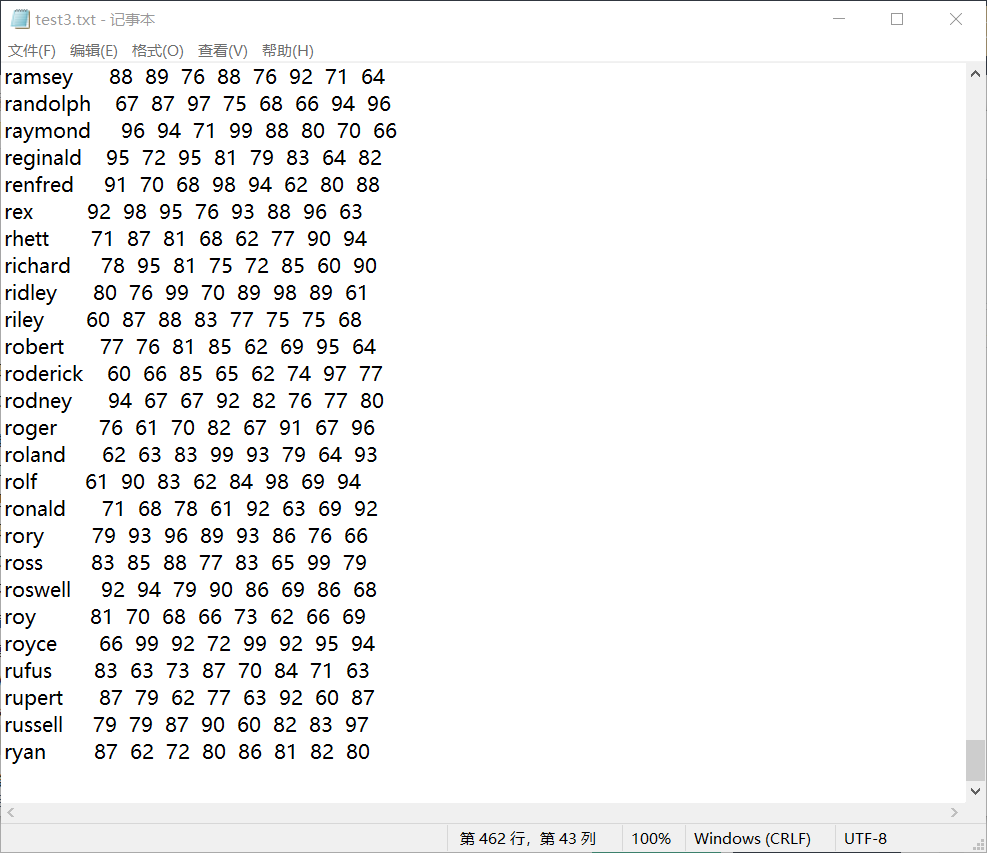
查找：string的find方法的复杂度可以参考<https://www.zhihu.com/question/392846851>，暂且认为最差情况O(ml)，平均情况O(m+l)。其中pattern长m，text长l。

撤销/重做：O(c)

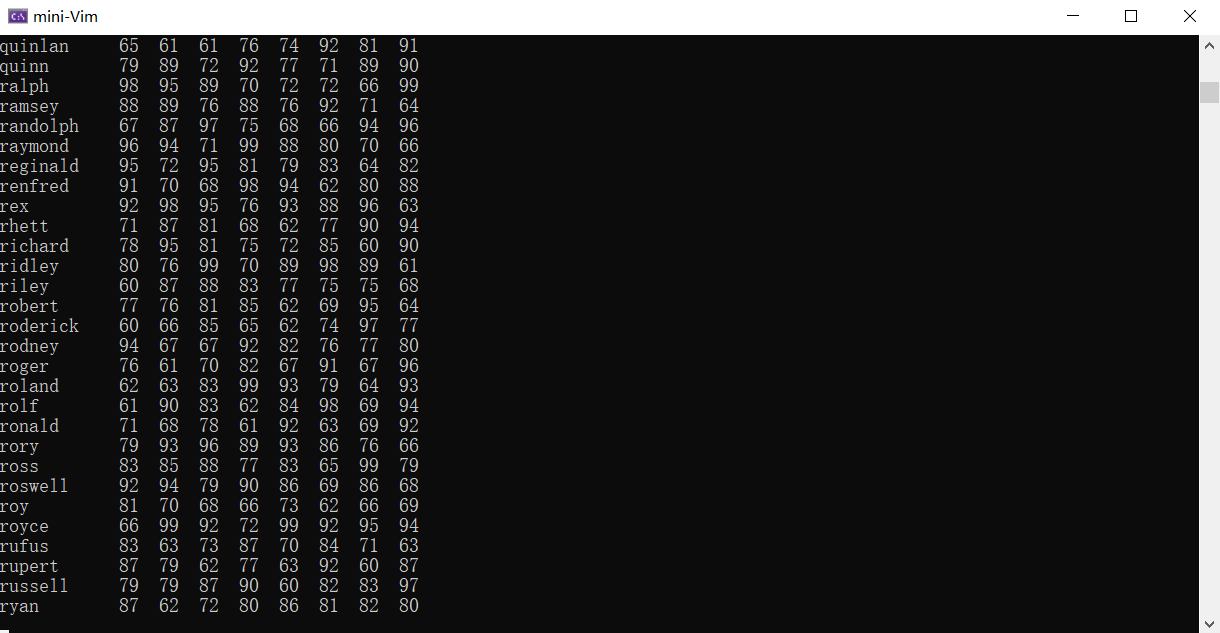
问题：①find方法复杂度比较高，直接使用了string类的find算法。为什么没有使用KMP算法？KMP算法每次查找都要重新计算一遍next数组，在短文本上表现不一定好，所以没有采用。因为mini-Vim目标并非用于超长文本处理，且随机输入很少出现最坏情况，对于普通长度文本来讲普通的find算法完全足够。如果确实追求这方面效率，可以使用C++17中的search函数，使用functional库中的boyer\_moore\_searcher（BM算法）以提高效率。

②撤销和重做用到栈是毫无疑问的，但直接保存状态这个做法在多次操作的情况下可能对空间造成一定负担，有待优化。

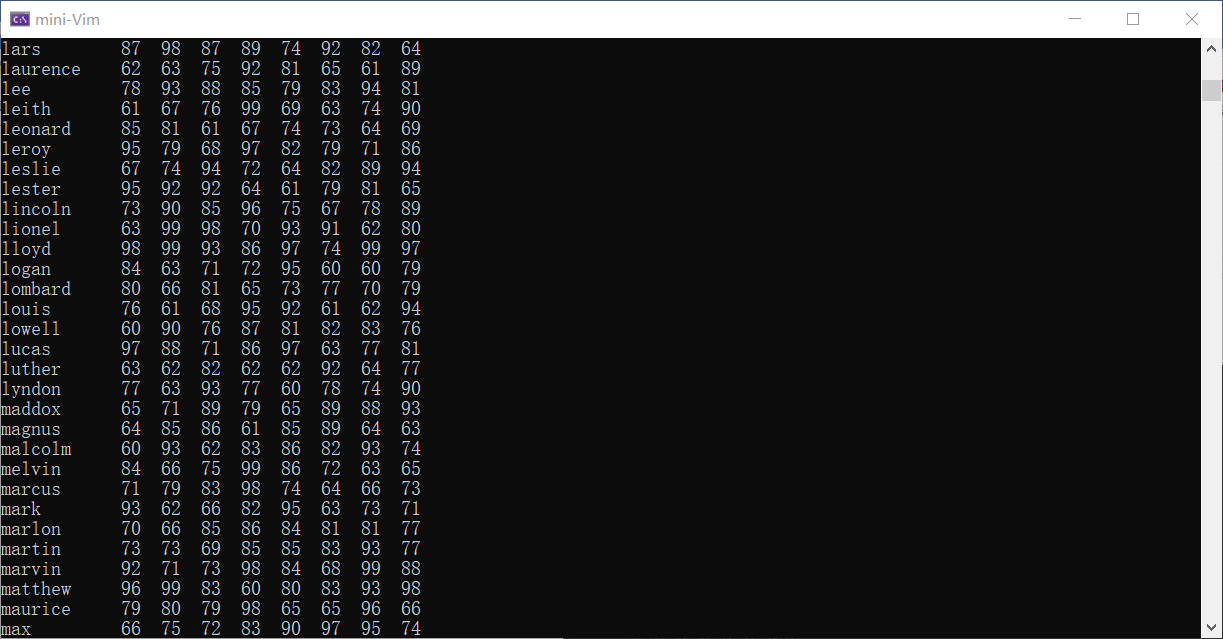
测试（三）：打开test3.txt



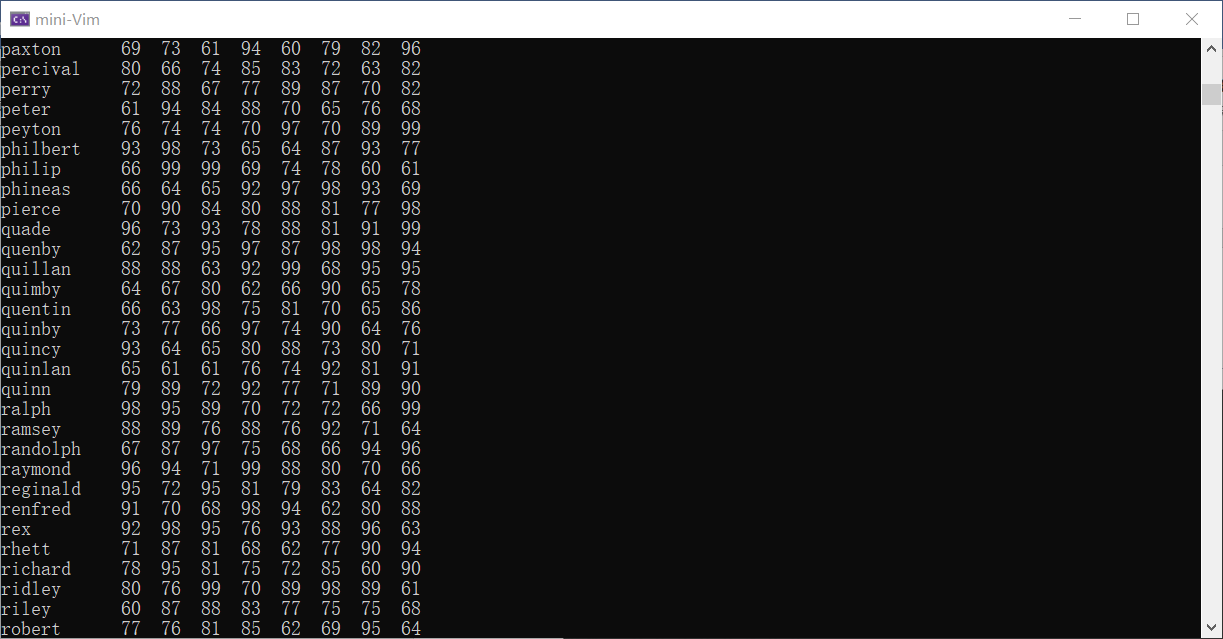
test3.txt:共462行，约20000字



向上翻页几次



向下翻页几次



总结：

翻页：O(1)

问题：翻页只是做了一个简单的光标移动，实际使用的体验不是太好。需要做进一步优化。

五、课程设计总结

（一）一些其他的问题

①关于使用system(“CLS”)再全部输出的问题

这样做最大的问题就在于：在文本内容较多时，刷新会有明显的闪屏，且由于输出非常消耗时间，每次做任何操作都会有屏幕的闪烁、滚动。这是目前最大的问题。

改进方法：可以再建造一个缓冲区，利用双缓冲区显示减少显示问题。

②关于使用vector<string>保存内容

常规使用上效率确实已经足够，但这个方案并不是最优的。参考<https://www.catch22.net/tuts/neatpad/piece-chains>给出的方案，这种方案在撤销、重做、插入、删除上都有一定优势，只是在处理换行上有待斟酌。

③一些细节的实现

细节的实现均参考了原版vim，比如：

上下左右（左右不超出本行，上在上一行比本行短时会跳到上一行行末，下同理，但下不会跳到空行上），回车（能分割本行），回退（能将本行接到上一行）命令的删除（输入:或\后再删除，光标会回退到原先的位置），撤销与重做的保存时机（原版的撤销仅有2种情况：一是insert插入后切换到normal模式算一次保存，此时撤销会删去insert插入的全部文本；二是normal模式下的删除算一次保存，此时撤销会插回删去的文本）

一些未实现的细节：(i)撤销重做时，光标也回到上一次的位置(ii)原版vim限定了输入的文本不能和最后一行重叠，最后一行只用于命令行输入

（二）感想

文本编辑器是一个看着简单但做起来细节繁琐，优化较难的项目。本次PJ是我第二次做大作业，过程虽然艰难，但我也收获了许多。我最大的收获应该是增强了我查找资料的能力，学会在Github、Stackoverflow等平台学习；第二个收获是在编程的命名规范上有很大的收获，做到严格按功能区分头文件和各个cpp文件，命名一目了然，注释比较齐全；第三是学习了相关的系统编程内容，对控制台函数有了一定的了解。

六、参考资料

[1] <https://github.com/SirIsaacNeutron/vimperor>

[2] <https://github.com/stoneteen/si-mini-vim-c->

[3] <https://www.catch22.net/tuts/neatpad/piece-chains>

[4] <https://zhuanlan.zhihu.com/p/268726520>

[5] <https://www.runoob.com/linux/linux-vim.html>