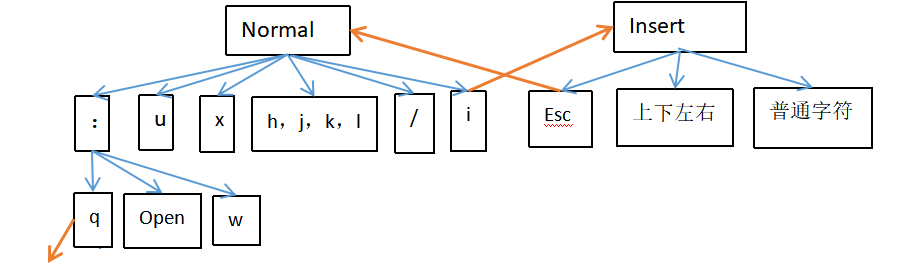
MINI VIM 编辑器实验报告

张志豪20307130085



1. 前言

写在前面，引用声明：此pj中许多的控制台函数操作引用自csdn文章https://blog.csdn.net/weixin\_30289831/article/details/99138150?ops\_request\_misc=&request\_id=&biz\_id=102&utm\_term=console%E5%87%BD%E6%95%B0c++&utm\_medium=distribute.pc\_search\_result.none-task-blog-2~all~sobaiduweb~default-0-99138150.first\_rank\_v2\_pc\_rank\_v29&spm=1018.2226.3001.4187

代码长度较长，可能是由于在每个函数中都考虑了文本出现换行或者翻页的特殊情况（也可以说是这份代码的特色），在处理细节的过程中也浪费了较多的内存，可以进一步改进。

1. 基本思路

在mini vim中有normal和insert两种模式，一开始作者试图将normal和insert分成两个源文件来写，在写的过程中发现二者是“纠缠”在一起的，因此最后只有一个源文件。

由于功能较多，作者认为开始时的架构非常困难。

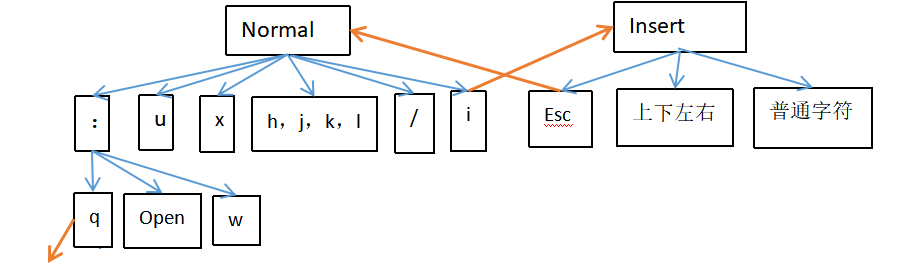
下面结合头文件来梳理思路。

1. #pragma once
2. #include<iostream>
3. #include<Windows.h>
4. #include<string.h>
5. #include<stdlib.h>
6. #include <conio.h>
8. **class** vim\_operator {
9. **private**:
10. //normal模式下的函数们
11. **bool** normal\_function(); //normal模式功能表函数
12. **bool** lastline();    //将指令输出在最后一行
13. **bool** withdraw();    //执行命令u
14. **bool** Delete();  //执行命令x
15. **bool** NormalMove(**const** **char**& a); //normal光标位置移动
16. **bool** OpenFile(**const** **char**\* filename);    //打开文件
17. **bool** WriteFile(**const** **char**\* filename);   //写入文件
18. **bool** search();  //执行搜索命令
19. **bool** BM(**char**\* t, **const** **int** m);  //利用BM算法匹配字符串
20. **bool** clean();   //清除控制台页面中的所有内容
22. //insert模式下的函数们
23. **bool** insert\_function(); //insert模式功能表函数
24. **bool** InsertMove();  //insert光标位置移动
25. **bool** grow();    //行数超过时的增长
26. **bool** Insert\_cin(**char** ch);   //insert模式下单字符的插入
28. //bonus
29. **bool** pageup();  //向上翻页
30. **bool** pagedown();    //向下翻页
31. //有用的数据和结构们
32. **char** mode;  //记录模式
33. **int** height;  //记录窗口高度
34. **int** row\_num;  //记录最后的一行的行号
35. **int** page\_num;  //页数的最大值
36. **const** **char**\* TempFileName = { "tempfile.txt" };  //临时文件的名称，撤销操作用
37. **const** **char**\* LastFileName= { "lastfile.txt" };
38. **int**\*\* num;  //存储每一行的元素个数,两个维度分别是页数和行数
39. **int**\*\* find\_enter;  //记录每一行的的回车符是不是在本行，因为在下一行时，再在本行输入，除了正常输入，还需要回车
40. **int** mod\_re; // 判断是撤销操作还是反撤销操作
41. **char** state\_judge;  //判断撤销是撤销insert还是x
42. **struct** feedback//用来记录撤销前的光标位置、字符、是撤销还是反撤销
43. {
44. COORD pre = { 0,0 };
45. **char** ch = 0;
46. **int** mod\_re = 1;
47. } Feedback;
49. **public**:
50. **bool** judge();   //用来判断属于哪一种模式
51. **void** renew();   //更新目前光标位置
52. vim\_operator() //构造函数
53. {
54. mode = 'n';//默认模式为normal模式
55. **HANDLE** hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);
56. CONSOLE\_SCREEN\_BUFFER\_INFO pBuffer;
57. GetConsoleScreenBufferInfo(hStdout, &pBuffer);
58. height = pBuffer.srWindow.Bottom - pBuffer.srWindow.Top + 1;
59. num = **new** **int**\* [100];
60. find\_enter = **new** **int**\* [100];
61. num[0] = **new** **int**[height];  //一页
62. find\_enter[0] = **new** **int**[height];
63. page\_num = 1;
64. row\_num = 0; //代表现在最后一行为第一行
65. mod\_re = 1;
66. //which\_back = 0; //0表示没有撤回，1表示之前操作为insert，2表示之前操作为/x
67. **for** (**int** i = 0; i < height; i++)
68. {
69. num[0][i] = 0;  //初始化每一行的元素个数都为0
70. find\_enter[0][i] = 0; //0表示该行为开头，即上行转下来时需要回车
71. }
72. }
73. };

最开始一个judge函数，用来判断vim属于两种模式中的哪一种，在构造函数中初始化为默认的normal模式。

Judge函数判定为mode==‘n’时进入normal模式下的功能表函数normal\_function，在normal\_function中利用char ch=\_getch（）读取缓冲区的字符，ch=‘/’或‘u’或‘x’或‘h，i，j，k’时进入对应的功能函数；ch=‘：’时进入指令函数lastline进一步判断是open、w、q中的哪一种指令；ch=‘i’时将状态mode改为‘i’返回judge函数并进入insert模式下的功能表函数insert\_function。

同样地，在mode==‘i’的insert模式功能表函数中也根据ch=\_getch（）判断是执行光标移动，还是退出insert模式，还是直接键入字符。



下面介绍一些程序中用到的几个属性，功能在注释中体现

1. **int** height;  //记录窗口高度
2. **int** row\_num;  //记录最后的一行的行号
3. **int** page\_num;  //页数的最大值
4. **const** **char**\* TempFileName = { "tempfile.txt" };  //临时文件的名称，撤销操作用
5. **const** **char**\* LastFileName= { "lastfile.txt" };
6. **int**\*\* num;  //存储每一行的元素个数,两个维度分别是页数和行数
7. **int**\*\* find\_enter;  //记录每一行的的回车符是不是在本行，因为在下一行时，再在本行输入，除了正常输入，还需要回车
8. **int** mod\_re; // 判断是撤销操作还是反撤销操作
9. **char** state\_judge;  //判断撤销是撤销insert还是x
10. **struct** feedback//用来记录撤销前的光标位置、字符、是撤销还是反撤销
11. {
12. COORD pre = { 0,0 };
13. **char** ch = 0;
14. **int** mod\_re = 1;
15. } Feedback;
16. 具体实现

1、准备工作

1. #include<iostream>
2. #include"pj\_vim\_operator.h"
3. #include<Windows.h>
4. #include<string.h>
5. #include<fstream>
6. #include<stdlib.h>
7. #include <conio.h>
8. **using** **namespace** std;
10. **struct** node
11. {
12. **int** x, y;
13. };
15. node get\_coordinate(**HANDLE** hStdout)//给定控制台界面，返回光标坐标
16. {
17. node coordinate;
18. CONSOLE\_SCREEN\_BUFFER\_INFO pBuffer;
19. GetConsoleScreenBufferInfo(hStdout, &pBuffer);
20. coordinate.x = pBuffer.dwCursorPosition.X;
21. coordinate.y = pBuffer.dwCursorPosition.Y;
22. **return** coordinate;
23. }
25. **void** vim\_operator::renew()  //更新cur指向现在光标位置
26. {
27. **HANDLE** hOut;
28. CONSOLE\_SCREEN\_BUFFER\_INFO INFO;
29. hOut = GetStdHandle(STD\_OUTPUT\_HANDLE);
30. GetConsoleScreenBufferInfo(hOut, &INFO); //得到现在屏幕缓冲区的光标位置
31. }

上述是一些准备过程，通过查阅资料直到光标的坐标由x和y组成，因此建立一个含有两个整数的结构node（后面发现其实用得不多），renew（）函数的作用仅仅是更新现在光标的位置。

2、模式判断函数

1. **bool** vim\_operator::judge()//用来判断属于哪一种模式
2. {
3. **while** (**true**)
4. {
5. **if** (mode == 'n')
6. {
7. **if** (!normal\_function())
8. **return** **false**;
9. }
10. **else** **if** (mode == 'i')
11. {
12. **if** (!insert\_function())
13. **return** **false**;
14. }
15. }
16. **return** **false**;
17. }

此函数为最顶层的逻辑判断，根据mode的值判断应该进入哪种模式的功能表函数，同时可以由于：q操作返回false而退出程序。

3、normal模式功能表

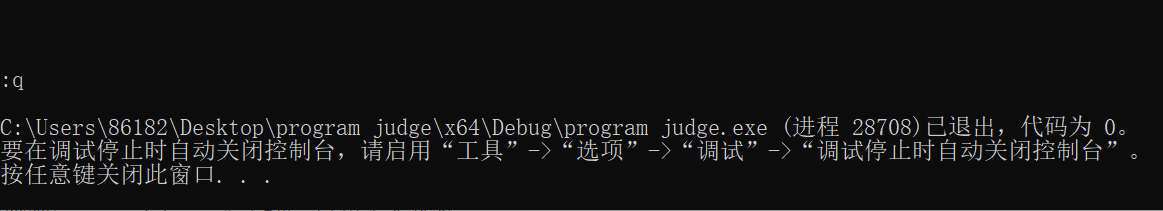
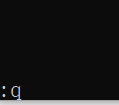
1. **bool** vim\_operator::normal\_function()//normal模式下的功能表函数
2. {
3. **while** (1)
4. {
5. **char** ch;
6. ch = \_getch(); //将缓冲区中的数据以字符的形式读出
7. **if** (ch == ':')
8. {
9. **if** (!lastline()) **return** **false**;
10. }
11. **else** **if** (ch == 'h' || ch == 'j' || ch == 'k' || ch == 'l')  //移动光标
12. {
13. **if** (!NormalMove(ch)) **return** **false**;
14. }
15. **else** **if** (ch == -32)
16. {
17. **if** (!InsertMove()) **return** **false**;
18. }
19. **else** **if** (ch == '/')
20. {
21. search();
22. }
23. **else** **if** (ch == 'x')
24. {
25. Feedback.ch = 0;
26. Feedback.mod\_re = 1;
27. Feedback.pre = { 0,0 };
28. state\_judge = 'x'; //确定该操作为撤销
29. Delete();
30. }
31. **else** **if** (ch == 'i')
32. {
33. state\_judge = 'i';  //确定该操作为插入
34. mode = 'i';
35. **return** **true**;
36. }
37. **else** **if** (ch == 'u')
38. {
39. **if** (!withdraw())
40. **return** **false**;
41. }
42. }
43. }

此函数为normal模式下的功能表函数，根据ch=\_getch（）判断进一步进入哪个函数。

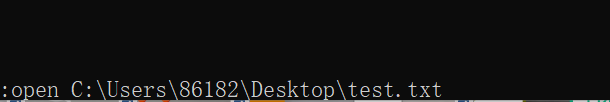
1. normal模式下输入‘：’，此时指令在最后一行打印，进入lastline（）函数
2. **bool** vim\_operator::lastline()//将指令输出在最后一行
3. {
4. **HANDLE** hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);
5. CONSOLE\_SCREEN\_BUFFER\_INFO pBuffer;
6. GetConsoleScreenBufferInfo(hStdout, &pBuffer);
7. COORD lowest;   //记录最底端开头的光标位置
8. COORD record = pBuffer.dwCursorPosition;
9. lowest.X = 0;
10. lowest.Y = pBuffer.srWindow.Bottom;
11. **int** page = pBuffer.srWindow.Bottom / height;
12. **int** row = pBuffer.srWindow.Bottom % height;
13. **char**\* temp = **new** **char**[num[page][row] + 1];  //记录原来最后一行
14. temp[num[page][row]] = 0;  //保证0结尾，使得temp成为字符串
15. SetConsoleCursorPosition(hStdout, lowest);
16. unsigned **long**\* num1 = **new** unsigned **long**;
17. ReadConsoleOutputCharacterA(hStdout, temp, num[page][row], lowest, num1);   //读取控制台屏幕缓冲区中最后一行的元素
18. **for** (**int** i = 0; i < num[page][row]; i++)
19. cout << " ";  //清空该行原有数据
20. SetConsoleCursorPosition(hStdout, lowest);
21. **int** count = 0;
22. cout << ':';
23. **char** str[5] = { 0 };
24. cin >> str;
25. **if** (str[0] == 'q' && str[1] == 0) //退出
26. {
27. DeleteFileA(TempFileName);
28. DeleteFileA(LastFileName);
29. **return** **false**;
30. }
31. **else** **if** (str[0] == 'o' && str[1] == 'p' && str[2] == 'e' && str[3] == 'n' && str[4] == 0)  // 文件读取
32. {
33. **char** file\_name[100];
34. cin >> file\_name;
35. **for** (count = 0; file\_name[count] != 0; count++);
36. count += 6;
37. //还原
38. SetConsoleCursorPosition(hStdout, lowest);
39. **for** (**int** i = 0; i < count; i++)
40. cout << " ";  //清空该行原有数据
41. SetConsoleCursorPosition(hStdout, lowest);
42. cout << temp;  //还原最后一行的元素
43. SetConsoleCursorPosition(hStdout, record);  //还原原有输出窗口光标位置
44. **if** (!OpenFile(file\_name))
45. **return** **false**;
46. }
47. **else** **if** (str[0] == 'w' && str[1] == 0)  //文件写入
48. {
49. **char** file\_name[100];
50. cin >> file\_name;
51. **for** (count = 0; file\_name[count] != 0; count++);
52. count += 3;
53. //还原
54. SetConsoleCursorPosition(hStdout, lowest);
55. **for** (**int** i = 0; i < count; i++)
56. cout << " ";  //清空该行原有数据
57. SetConsoleCursorPosition(hStdout, lowest);
58. cout << temp;  //还原最后一行的元素
59. SetConsoleCursorPosition(hStdout, record);  //还原原有输出窗口光标位置
60. **if** (!WriteFile(file\_name))
61. **return** **false**;
62. }
63. }

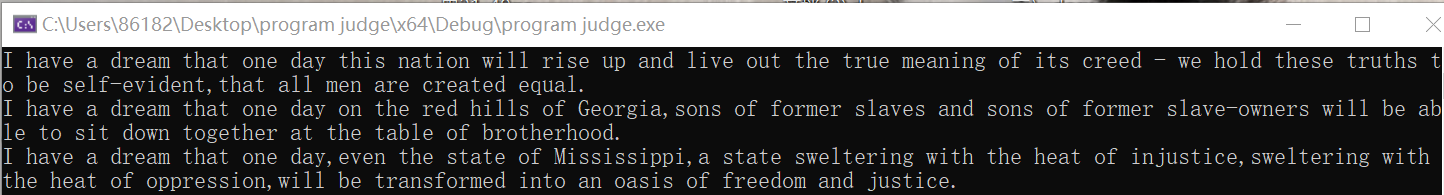
输入‘i’后，字符i保存在程序的缓冲区，此时调用 SetConsoleCursorPosition函数将光标移动到控制台最底端，输出一个字符‘i’。然后接着输入指令，用一个int型变量count记录指令本身的字符串长度，在指令输入完毕后输出count个空格（即清空最后一行），并将光标还原到控制台的最开始位置。

1. 输入“：q”时的结果



1. 输入“：open C:\Users\86182\Desktop\test.txt”时的结果

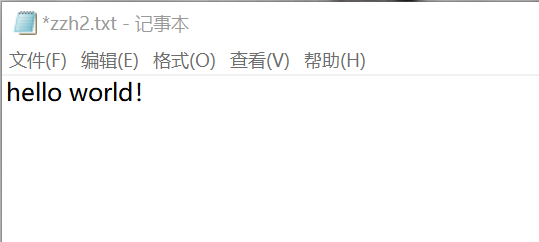


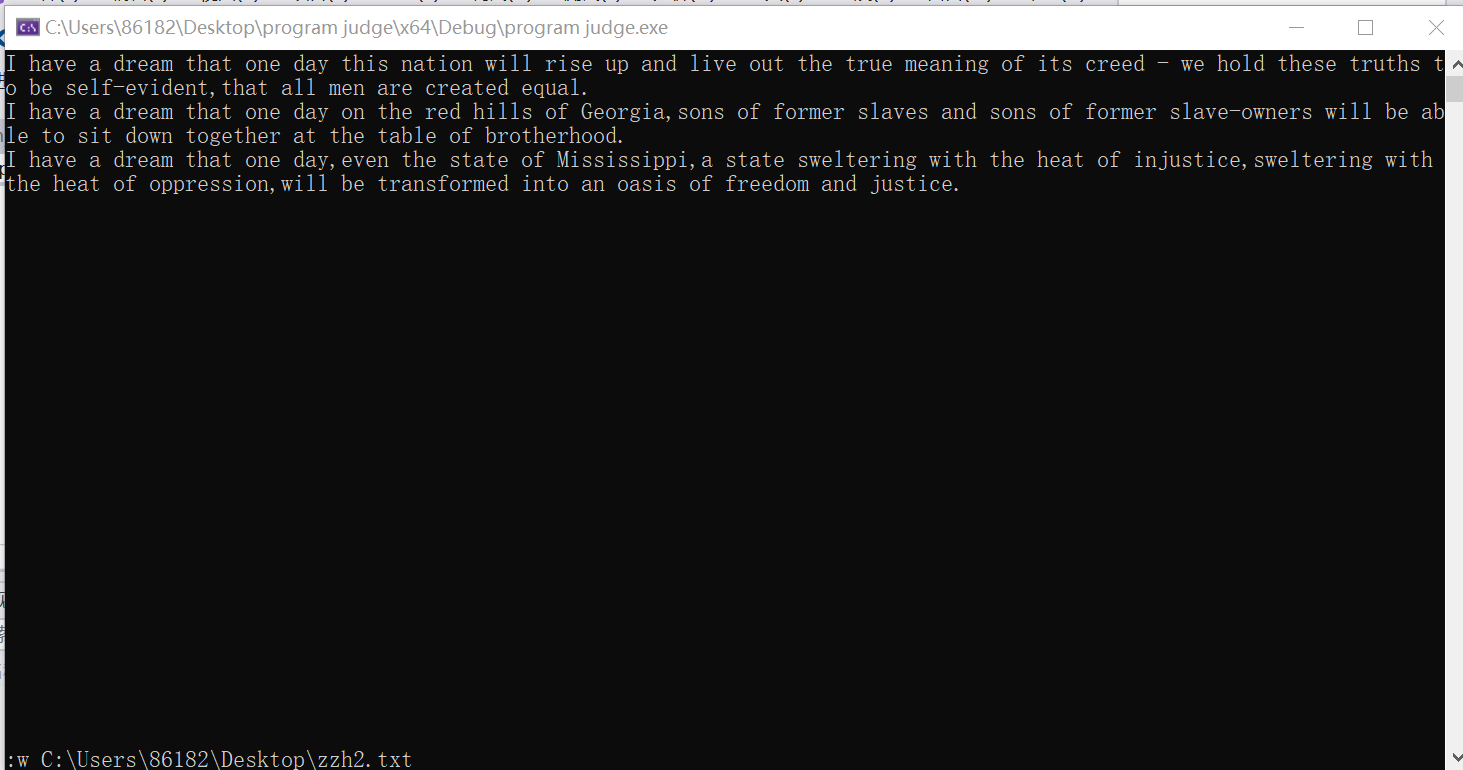


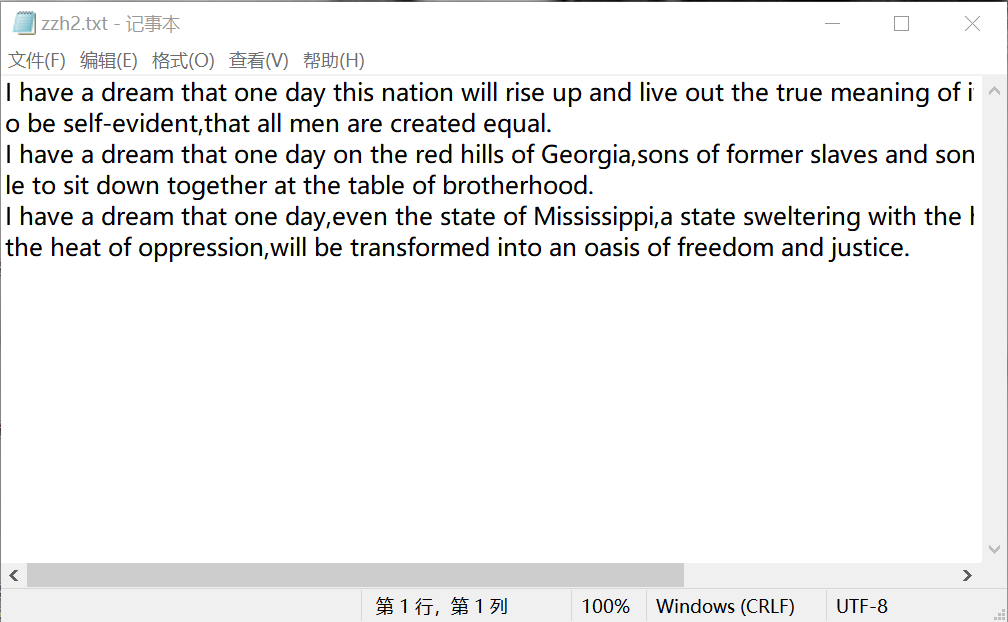
1. **bool** vim\_operator::OpenFile(**const** **char**\* filename)
2. {
3. **char** p;
4. **HANDLE** hStdout;
5. CONSOLE\_SCREEN\_BUFFER\_INFO scr;
6. hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);    // 获取标准输出句柄
7. GetConsoleScreenBufferInfo(hStdout, &scr);
8. COORD pos = scr.dwCursorPosition;
9. **int** page = pos.Y / height;
10. **int** row = pos.Y % height;
11. ifstream infile;
12. infile.open(filename);
13. **int** count\_char = 0;
14. **while** (!infile.fail())
15. {
16. p = infile.get();
17. //当文件中为‘\n’时，需要对控制台的row\_num 进行增加
18. **if** (p == '\n')
19. {
20. cout << '\n';
21. **if** (page \* height + row == row\_num)
22. row\_num++;
23. **if** (row\_num == page\_num \* height) //如果总的最后一行为最大页的最后一行，则需要新加页
24. **if** (!grow()) **return** **false**;
25. page += (row + 1) / height;
26. row = (row + 1) % height;
27. count\_char = 0;
28. }
29. **else**
30. {
31. **HANDLE** hStdout;
32. CONSOLE\_SCREEN\_BUFFER\_INFO scr;
33. hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);    // 获取标准输出句柄
34. GetConsoleScreenBufferInfo(hStdout, &scr);
35. **if** (count\_char == scr.srWindow.Right + 1)
36. {
37. **if** (page \* height + row == row\_num)
38. row\_num++;
39. **if** (row\_num == page\_num \* height) //如果总的最后一行为最大页的最后一行，则需要新加页
40. **if** (!grow()) **return** **false**;
41. page += (row + 1) / height;
42. row = (row + 1) % height;
43. count\_char = 0;
44. }
45. **else**
46. count\_char++;
47. cout << p;
48. num[page][row]++;
49. }
50. }
51. GetConsoleScreenBufferInfo(hStdout, &scr);
52. pos = scr.dwCursorPosition;
53. **if** (pos.X < 3)
54. pos.X = 0;
55. **else**
56. pos.X -= 3;
57. SetConsoleCursorPosition(hStdout, pos);
58. num[page][row]--;
59. infile.close();
60. **return** **true**;
61. }
62. **bool** vim\_operator::grow()
63. {
64. num[page\_num] = **new** **int**[height];
65. find\_enter[page\_num] = **new** **int**[height];
66. **for** (**int** i = 0; i < height; i++)
67. {
68. num[page\_num][i] = 0;  //初始化每一行的元素个数都为0
69. find\_enter[page\_num][i] = 0;
70. }
71. page\_num++;
72. **return** **true**;
73. }

每次从文件读入一个字符，注意：当读入的字符为‘\n’时需要对控制台的row\_num加1，如果总的最后一行为最大页的最后一行，则需要利用grow（）函数新加页。当读入的字符非‘\n’时，对各个类属性进行更新后直接打印读入的字符，同时依然要注意是否需要新加页。

1. 输入“：w C:\Users\86182\Desktop\zzh2.txt”时的结果



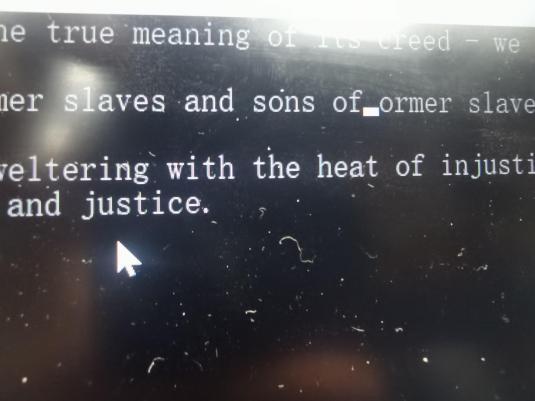
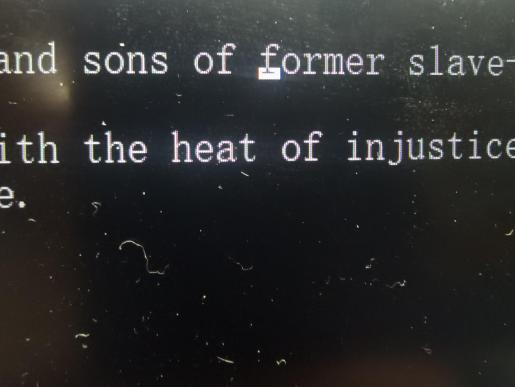




原本文件中的内容被覆盖

1. **bool** vim\_operator::WriteFile(**const** **char**\* filename)
2. {
3. ofstream outfile;
4. outfile.open(filename);
5. **HANDLE** hStdout;
6. CONSOLE\_SCREEN\_BUFFER\_INFO scr;
7. hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);    // 获取标准输出句柄
8. GetConsoleScreenBufferInfo(hStdout, &scr);
9. COORD before = scr.dwCursorPosition;
10. COORD pos = { 0,0 };
11. **while** (pos.Y <= row\_num)
12. {
13. **int** page = pos.Y / height;
14. **int** row = pos.Y % height;
15. **char**\* temp = **new** **char**[num[page][row] + 1]; //存储原有最后一行的元素
16. temp[num[page][row]] = 0;  //保证0结尾，使得temp成为字符串
17. unsigned **long** num1 = 0;
18. ReadConsoleOutputCharacterA(hStdout, temp, num[page][row], pos, &num1);//读取控制台屏幕缓冲区中第一行的元素
19. outfile << temp;
20. **delete**[]temp;
21. **if** (pos.Y != row\_num)  //防止最后一次多输出一行
22. outfile << endl;
23. pos.Y++;
24. }
25. SetConsoleCursorPosition(hStdout, before);  //还原光标位置
26. outfile.close();
27. **return** **true**;
28. }
29. normal模式下输入‘h’‘j’‘k’‘l’，实现光标的移动
30. **bool** vim\_operator::NormalMove(**const** **char**& ch)
31. {
32. **HANDLE** hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);
33. CONSOLE\_SCREEN\_BUFFER\_INFO pBuffer;
34. GetConsoleScreenBufferInfo(hStdout, &pBuffer);
35. COORD present;  //记录当前的光标位置
36. present.X = get\_coordinate(hStdout).x;
37. present.Y = get\_coordinate(hStdout).y;
38. **if** (ch == 'h')
39. {
40. present.X--;
41. SetConsoleCursorPosition(hStdout, present);
42. **return** **true**;
43. }
44. **else** **if** (ch == 'j')
45. {
46. present.Y++;
47. SetConsoleCursorPosition(hStdout, present);
48. **return** **true**;
49. }
50. **else** **if** (ch == 'k')
51. {
52. present.Y--;
53. SetConsoleCursorPosition(hStdout, present);
54. **return** **true**;
55. }
56. **else** **if** (ch == 'l')
57. {
58. present.X++;
59. SetConsoleCursorPosition(hStdout, present);
60. **return** **true**;
61. }
62. **else**
63. **return** **false**;
64. }
65. normal模式下输入‘x’，实现光标处字符的清除

实验结果如下：

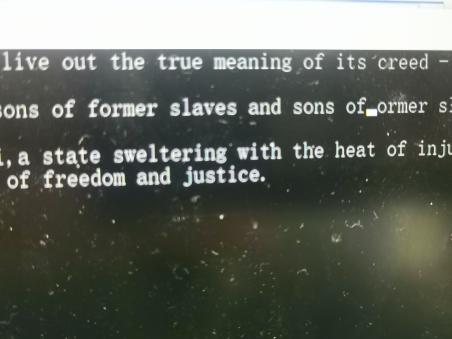
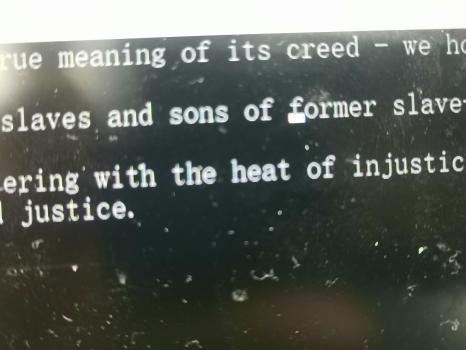


1. **bool** vim\_operator::Delete()
2. {
3. **HANDLE** hOut;
4. CONSOLE\_SCREEN\_BUFFER\_INFO hBuffer;
5. hOut = GetStdHandle(STD\_OUTPUT\_HANDLE);    // 获取标准输出句柄
6. GetConsoleScreenBufferInfo(hOut, &hBuffer);
7. COORD position = hBuffer.dwCursorPosition;
8. unsigned **long** num1 = 0;
9. **int** page = position.Y / height;
10. **int** row = position.Y % height;
11. **if** (position.X == 0 && num[page][row] == 0)
12. **return** **true**;
13. **char**\* temp = **new** **char**[num[page][row] - hBuffer.dwCursorPosition.X]; //存储光标后的元素，在后续进行输出
14. temp[num[page][row] - position.X - 1] = 0;  //保证0结尾，使得temp成为字符串
15. position.X += 1;
16. ReadConsoleOutputCharacterA(hOut, temp, num[page][row] - position.X, position, &num1);//读取控制台屏幕缓冲区中最后一行的元素
17. position.X -= 1;
18. **if** (Feedback.mod\_re == 1)  //为撤销操作做准备
19. {
20. Feedback.pre = position;
21. ReadConsoleOutputCharacterA(hOut, &Feedback.ch, 1, Feedback.pre, &num1);//读取光标前的第一个元素，为撤回操作做准备
22. }
23. SetConsoleCursorPosition(hOut, position);
24. cout << temp << ' ';
25. **delete**[]temp;
26. num[page][row]--;
27. position = hBuffer.dwCursorPosition;
28. **if** (position.X != 0)
29. position.X -= 1;
30. SetConsoleCursorPosition(hOut, position);
31. **return** **true**;
32. }

对于删除操作，用一个字符型指针存储光标后的元素，将光标退回一位，再将字符指针存储的字符输出，即可达到删除字符的效果。同时，删除操作的过程中还需要为撤回操作做铺垫，即将Feedback中的mod\_re赋值成1，pre赋值为光标位置position，这样可以在撤回时判定上一步操作为删除字符。

7、normal模式下输入‘u’，实现撤销操作和反撤销操作

实验结果如下：



可以看到，按下一次u时撤销了上一步的删除自护操作，再次按下u时撤销了上一步的撤销操作，因此字符又被删除了。

1. **bool** vim\_operator::withdraw()
2. {
3. **if** (state\_judge == 'i')  //撤销的是insert模式输入
4. {
5. **if** (mod\_re == 1)  //第一次撤销（奇数次撤销），真正的撤销
6. {
7. WriteFile(LastFileName);
8. clean();
9. OpenFile(TempFileName);
10. mod\_re = 2;
11. }
12. **else**  //第二次（偶数次）撤销，即还原操作
13. {
14. WriteFile(TempFileName);
15. clean();
16. OpenFile(LastFileName);
17. mod\_re = 1;
18. }
19. }
20. **else** **if** (state\_judge == 'x')  //撤销的是x命令删除当前光标数据
21. {
22. **HANDLE** hOut;
23. hOut = GetStdHandle(STD\_OUTPUT\_HANDLE);
24. SetConsoleCursorPosition(hOut, Feedback.pre);
25. **if** (Feedback.mod\_re == 1)  //第一次撤销（奇数次撤销），真正的撤销
26. {
27. Insert\_cin(Feedback.ch);
28. Feedback.mod\_re = 2;
29. SetConsoleCursorPosition(hOut, Feedback.pre);
30. }
31. **else**  //第二次（偶数次）撤销，即还原操作
32. {
33. Delete();
34. Feedback.mod\_re = 1;
35. }
36. }
37. **return** **true**;
38. }

个人认为，撤销操作时此次大作业的难点之一。实在是没有想出什么好办法，于是在封装类的时候建立了两个临时文件

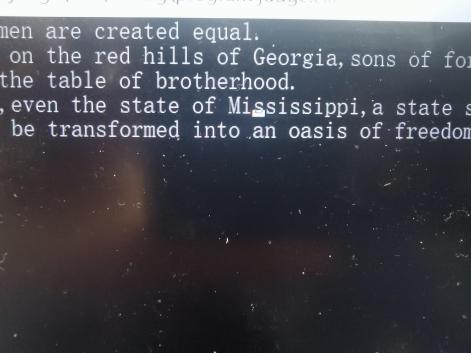
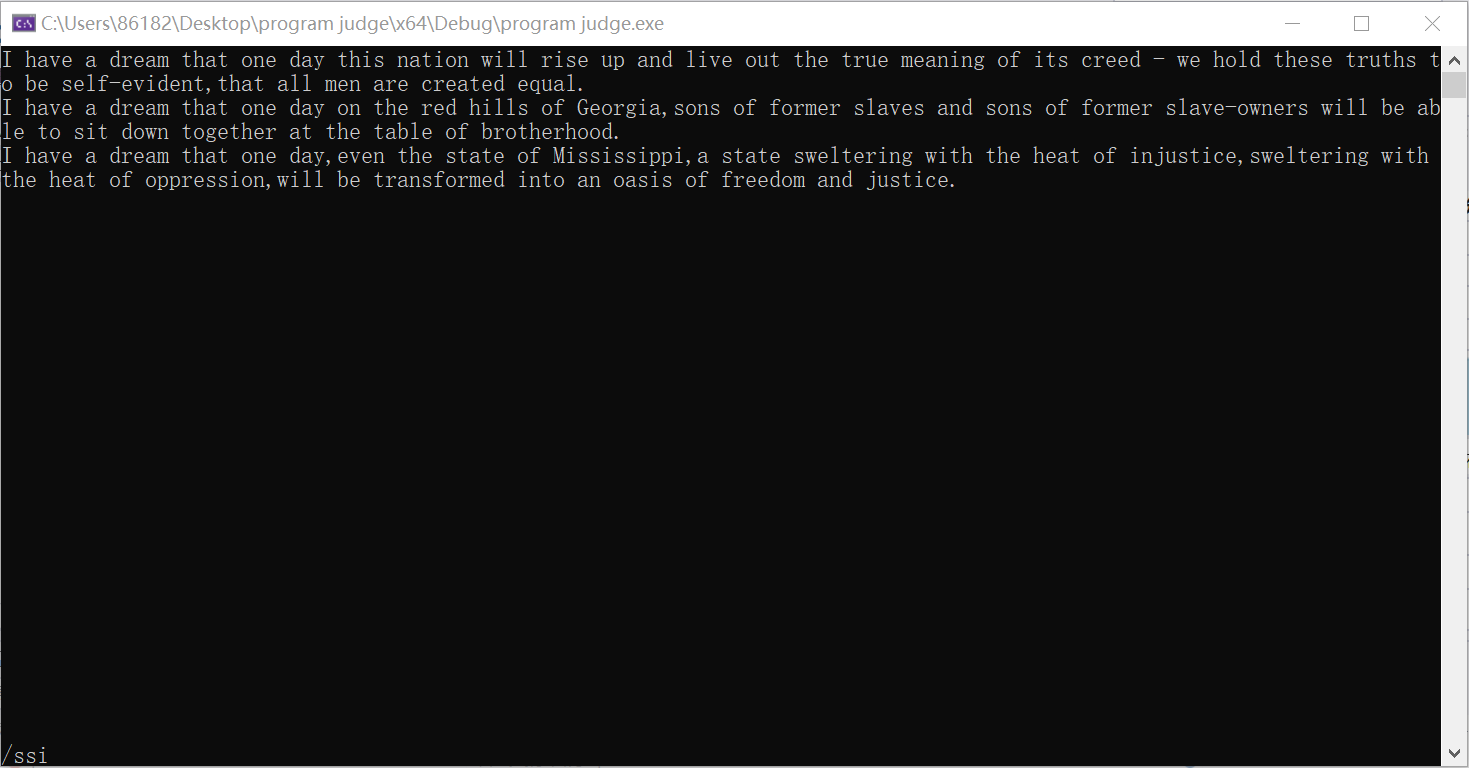
1. **const** **char**\* TempFileName = { "tempfile.txt" };  //临时文件的名称，撤销操作用

**const** **char**\* LastFileName= { "lastfile.txt" };

两个临时文件分别用来记录此时的控制台文本内容和执行撤回操作前的控制台文本内容。在执行的过程中要区分是“奇数次撤销”还是“偶数次”撤销，这就需要用到类属性Feedback。

1. normal模式下按下‘/’，然后输入匹配串pattern，实现查找功能

实验结果：



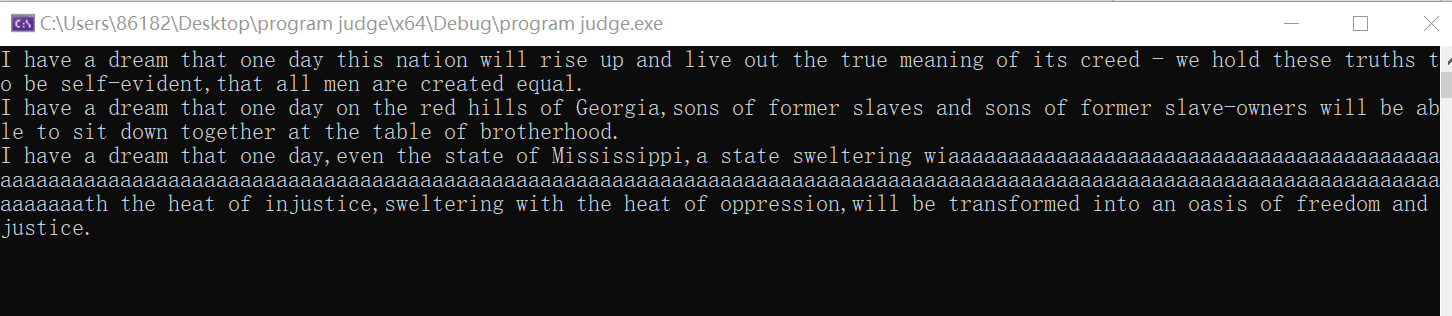
将光标移动到任意位置（图中移动到了Mississippi的开头处），输入指令“/ssi”，即查找最近的一处ssi字符串，结果光标移动到了ssi的开头s处。

1. **int** Find\_Distance(**char**\* t, **const** **char** ch, **const** **int** len) //BM算法求移动距离
2. {
3. **int** i = len - 1;
4. **if** (ch == t[i])
5. **return** len;
6. i--;
7. **while** (i >= 0)
8. {
9. **if** (ch == t[i])
10. **return** len - 1 - i;
11. **else**
12. i--;
13. }
14. **return** len;
15. }
17. **bool** vim\_operator::BM(**char**\* t, **const** **int** m)  // 此BM算法只适用与控制台，虽然原理是一样的
18. {
19. **int** i = m - 1;
20. **int** j = m - 1;
21. **HANDLE** hOut;
22. hOut = GetStdHandle(STD\_OUTPUT\_HANDLE);
23. **char**\* info = **new** **char**[m + 1];
24. CONSOLE\_SCREEN\_BUFFER\_INFO INFO;
25. GetConsoleScreenBufferInfo(hOut, &INFO); //得到现在屏幕缓冲区的光标位置
26. COORD now = INFO.dwCursorPosition;
27. COORD position = now;
28. unsigned **long**\* num1 = **new** unsigned **long**;
29. ReadConsoleOutputCharacterA(hOut, info, m, position, num1);//读取控制台屏幕缓冲区中的m个元素
30. **while** (**true**)
31. {
32. **if** (info[i] == t[j])
33. {
34. i--;
35. j--;
36. }
37. **else** **if** (j < 0)
38. {
39. //找到匹配，设置光标指向匹配的第一个元素首字母
40. SetConsoleCursorPosition(hOut, position);
41. **return** **true**;
42. }
43. **else**
44. {
45. **if** (position.Y > row\_num)
46. {
47. **int** page2 = row\_num / height;
48. **int** row = row\_num % height;
49. COORD posi = { num[page2][row], row\_num };
50. SetConsoleCursorPosition(hOut, posi);
51. **return** **true**;
52. }
53. i = Find\_Distance(t, info[i], m);
54. j = m - 1;
55. //设置下一次检索的位置
56. **int** page = position.Y / height;
57. **int** row = position.Y % height;
58. position.Y += (position.X + i) / num[page][row];
59. position.X = (position.X + i) % num[page][row];
60. i = j;
61. ReadConsoleOutputCharacterA(hOut, info, m, position, num1);
62. }
63. }
64. **return** **false**;
65. }
67. **bool** vim\_operator::search()
68. {
69. **HANDLE** hOut;
70. CONSOLE\_SCREEN\_BUFFER\_INFO scr;
71. hOut = GetStdHandle(STD\_OUTPUT\_HANDLE);    // 获取标准输出句柄
72. GetConsoleScreenBufferInfo(hOut, &scr);
73. COORD before = scr.dwCursorPosition;
74. COORD pos = { 0,scr.srWindow.Bottom };  //将光标移到当前窗口缓冲区最下方
75. **int** page = scr.srWindow.Bottom / height;
76. **int** row = scr.srWindow.Bottom % height;
77. **char**\* temp = **new** **char**(num[page][row] + 1);  //存储原有最后一行的元素
78. temp[num[page][row]] = 0;  //保证0结尾，使得temp成为字符串
79. SetConsoleCursorPosition(hOut, pos);
80. unsigned **long**\* num1 = **new** unsigned **long**;
81. ReadConsoleOutputCharacterA(hOut, temp, num[page][row], pos, num1);//读取控制台屏幕缓冲区中最后一行的元素
82. **for** (**int** i = 0; i < num[page][row]; i++)
83. cout << " ";
84. SetConsoleCursorPosition(hOut, pos);
85. cout << '/';
86. //存储pattern
87. **int** count = 0;
88. **char** pattern[100] = { 0 };
89. cin >> pattern;
90. **while** (pattern[count++] != 0);
91. count--;
92. SetConsoleCursorPosition(hOut, pos);
93. **for** (**int** i = 0; i < count + 2; i++)
94. cout << " ";  //清空该行原有数据
95. SetConsoleCursorPosition(hOut, pos);
96. cout << temp;  //还原最后一行的元素
97. SetConsoleCursorPosition(hOut, { 0,0 });  //还原原有输出窗口光标位置
98. SetConsoleCursorPosition(hOut, before);  // 还原原有光标位置
99. **if** (!BM(pattern, count))
100. **return** **false**;
101. **else**
102. **return** **true**;
103. }

查找功能很好理解，但是实现起来比较麻烦（作者调试了好长时间）。由于期末考试KMP算法给作者留下了心理阴影，这里使用BM算法进行查找。与之前的lastline（）函数相同，指令“/pattern”也需要被打印在控制台底部，然后清除指令字符串，将光标移动到匹配的字符处，如果找不到匹配则将光标移动到整段字符的末端。

1. insert模式下按下方向键，按照方向移动光标
2. **bool** vim\_operator::InsertMove()
3. {
4. **HANDLE** hStdout;
5. CONSOLE\_SCREEN\_BUFFER\_INFO INFO;
6. hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);
7. GetConsoleScreenBufferInfo(hStdout, &INFO); //得到现在屏幕缓冲区的光标位置
8. COORD present = INFO.dwCursorPosition;
9. **int** page = present.Y / height;
10. **int** row = present.Y % height;
11. **char** a = \_getch();
12. **if** (a == 75) //左移
13. {
14. **if** (present.X > 0)
15. present.X -= 1;
16. SetConsoleCursorPosition(hStdout, present);
17. }
18. **else** **if** (a == 80) // 下移
19. {
20. **if** (present.Y < row\_num)
21. {
22. present.Y += 1;
23. **if** (row == height - 1)
24. {
25. page += 1;
26. row = 0;
27. }
28. **else**
29. row += 1;
30. **if** (present.X > num[page][row])
31. present.X = num[page][row];
32. }
33. SetConsoleCursorPosition(hStdout, present);
34. }
35. **else** **if** (a == 72)  // 上移
36. {
37. **if** (present.Y > 0)
38. {
39. present.Y -= 1;
40. **if** (row == 0)
41. {
42. page -= 1;
43. row = height - 1;
44. }
45. **else**
46. row -= 1;
47. **if** (present.X > num[page][row])
48. present.X = num[page][row];
49. **if** (present.X < 0)  // 防止数组数据出现其他情况
50. present.X = 0;
51. }
52. SetConsoleCursorPosition(hStdout, present);
53. }
54. **else** **if** (a == 77)  // 右移
55. {
56. **if** (present.X < num[page][row])
57. present.X += 1;
58. **else** **if** (num[page][row] == 1)
59. {
60. **if** (present.Y < row\_num)
61. {
62. present.X = 0;
63. present.Y += 1;
64. }
65. }
66. SetConsoleCursorPosition(hStdout, present);
67. }
68. **else** **if** (a == 73) // page\_up时不尽光标会改变，屏幕缓冲区所
69. {
70. pageup();
71. }
72. **else** **if** (a == 81)
73. {
74. pagedown();
75. }
76. **else**
77. **return** **false**;
78. **return** **true**;
79. }
80. insert模式下按下非esc的按键，直接在光标处进行字符插入操作

实验结果：



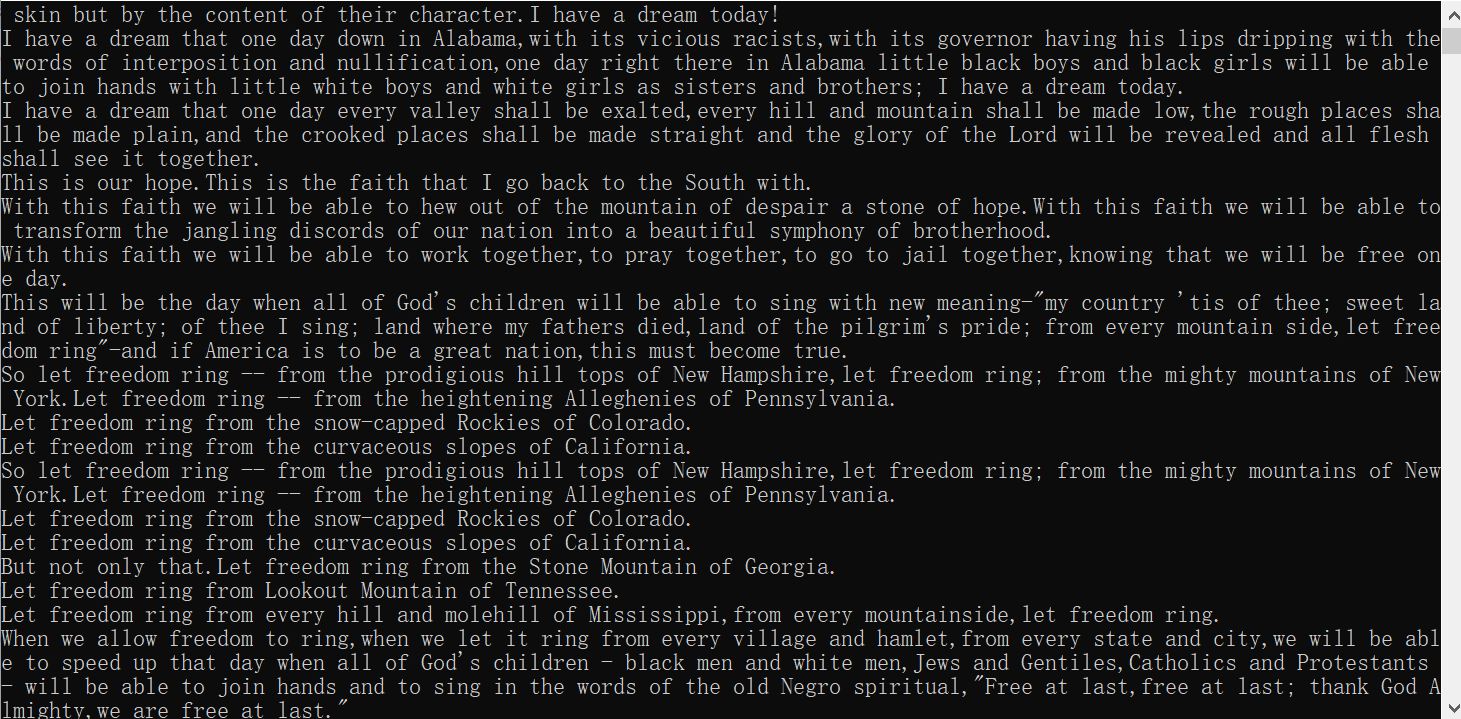
可以看到在任意位置可以插入任意多的字符，如果到达控制台的右边界会自动换行，如果控制台的一页写满会自动翻页。

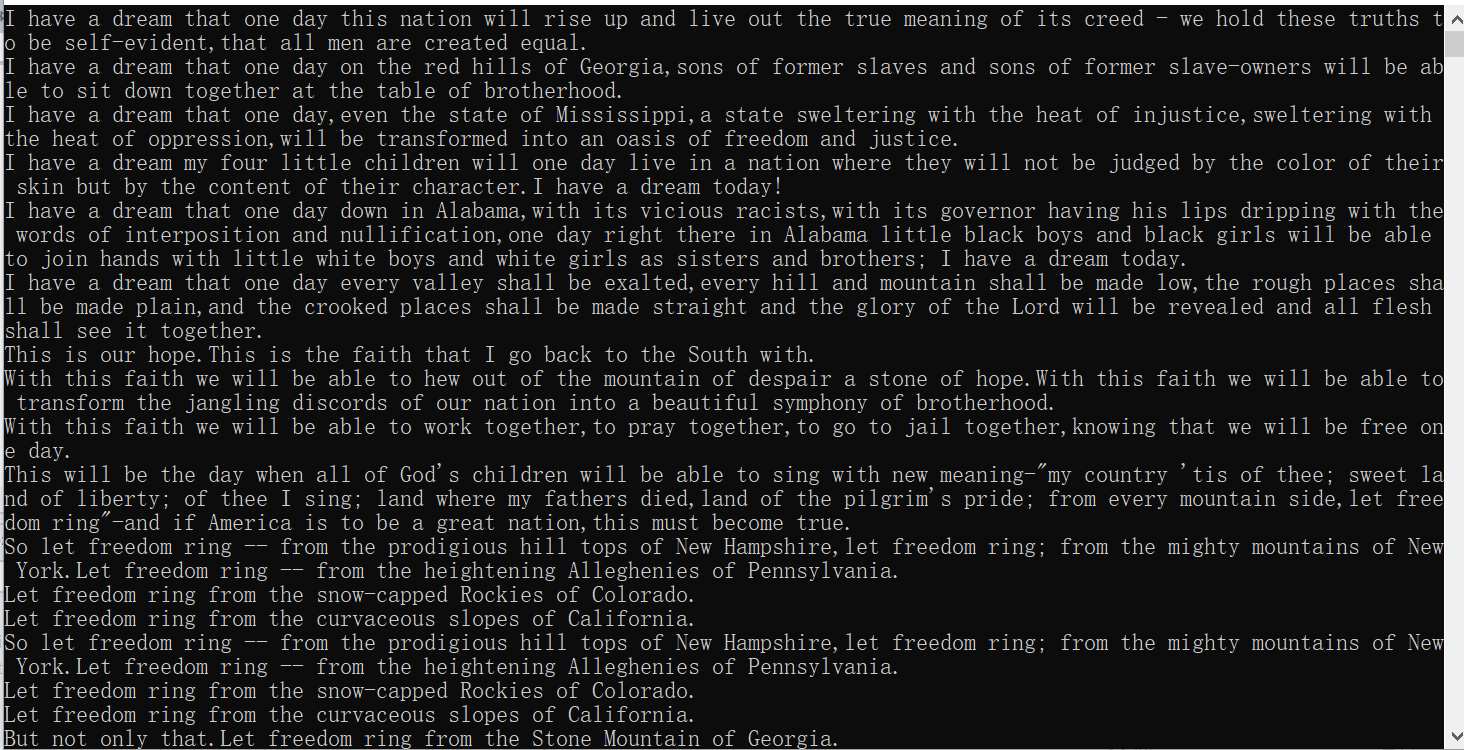
1. **bool** vim\_operator::Insert\_cin(**char** ch)
2. {
3. **HANDLE** hOut;
4. CONSOLE\_SCREEN\_BUFFER\_INFO hBuffer;
5. hOut = GetStdHandle(STD\_OUTPUT\_HANDLE);    // 获取标准输出句柄
6. GetConsoleScreenBufferInfo(hOut, &hBuffer);
7. unsigned **long** num1 = 0;
8. COORD position = hBuffer.dwCursorPosition;
9. **int** page = position.Y / height;
10. **int** row = position.Y % height;
11. //这个是更改find\_enter数组，为了自动转行而设置
12. **if** (position.X == 0 && !(page == 0 && row == 0)) //防止第一行出现问题
13. {
14. **if** (row != 0)
15. row -= 1;
16. **else**
17. {
18. page -= 1;
19. row = height;
20. }
21. }
22. page = position.Y / height;
23. row = position.Y % height;
24. **char**\* temp = **new** **char**[num[page][row] + 1 - hBuffer.dwCursorPosition.X]; //存储光标后面的元素
25. temp[num[page][row] - position.X] = 0;  //保证0结尾，使得temp成为字符串
26. ReadConsoleOutputCharacterA(hOut, temp, num[page][row] - position.X, position, &num1);//读取光标后面的元素
27. **char** temp\_one = 0;
28. **if** (num[page][row] - position.X != 0)
29. {
30. temp\_one = temp[num[page][row] - position.X - 1];
31. temp[num[page][row] - position.X - 1] = 0;
32. }
33. cout << ch << temp;
34. **if** (num[page][row] == hBuffer.srWindow.Right + 1) // 判断输入该字符后，该行最后一个字符会不会转行
35. {
36. cout << endl;
37. **if** (position.Y == row\_num)
38. {
39. row\_num++;
40. **if** (row\_num == page\_num \* height) //如果总的最后一行为最大页的最后一行，则需要新加页
41. **if** (!grow()) **return** **false**;
42. }
43. **if** (row != height)
44. row += 1;
45. **else**
46. {
47. row = 0;
48. page += 1;
49. }
50. Insert\_cin(temp\_one);
51. }
52. **else**
53. {
54. num[page][row]++;
55. cout << temp\_one;
56. }
57. position = hBuffer.dwCursorPosition; //插入时候的光标位置
58. **if** (position.X == hBuffer.srWindow.Right)
59. {
60. position.X = 0;
61. position.Y += 1;
62. }
63. **else**
64. position.X += 1;
65. SetConsoleCursorPosition(hOut, position);
66. **delete**[]temp;
67. **return** **true**;
68. }

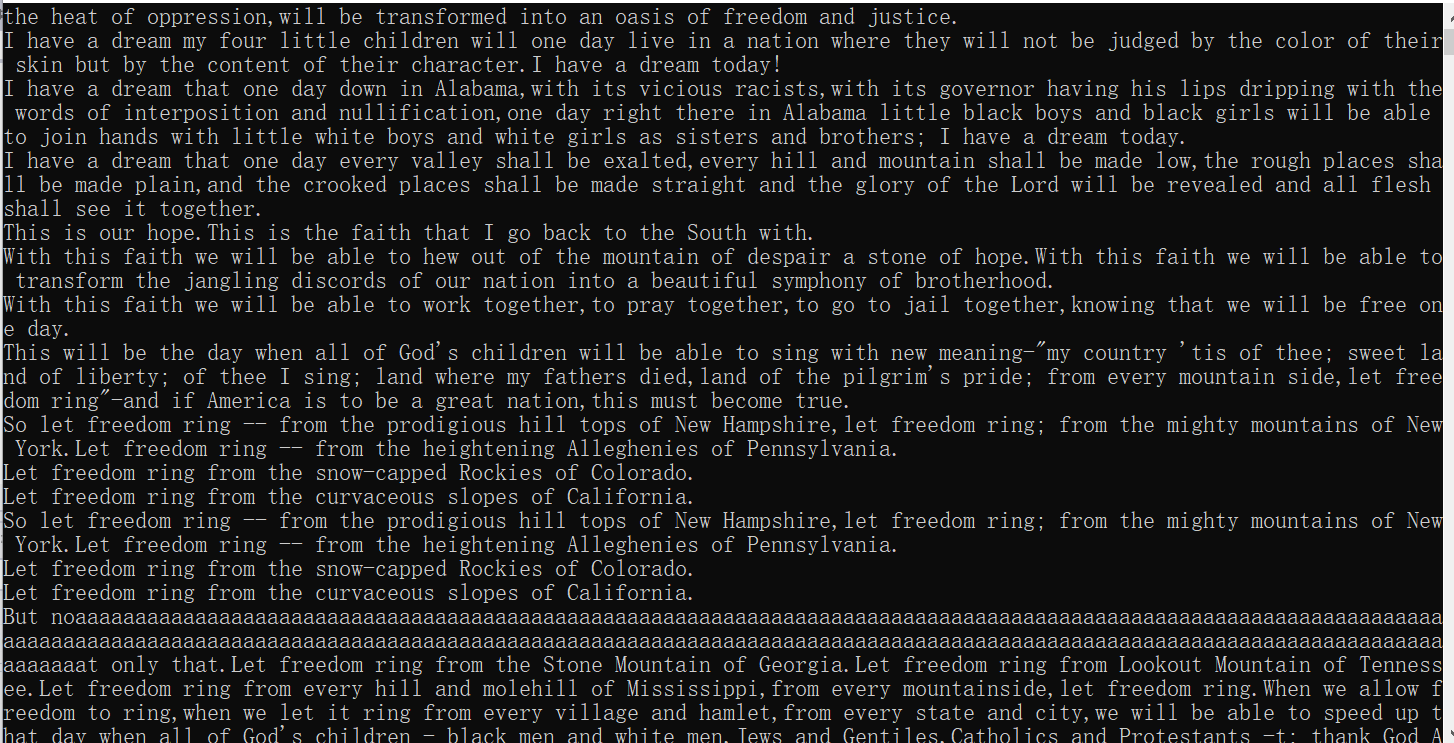
插入字符和删除字符异曲同工，先储存光标位置后面的字符，在光标处打印新输入的字符，然后将储存的字符连接在后面。同样也需要注意换行和翻页问题。

1. （bonus）在insert模式下按下PgUp、PgDn实现翻页，如果内容超过一页也会自动翻页。

实验结果：







读入一段较长的文本，在第一张图中显然控制台的左上角不是文本的开头，此时在insert模式下按下PgUp即可向上翻页到达文本的开头（如图二）；同样地，在insert模式下的任意光标位置开始插入大段字符，可以自动翻页（如图三）

1. **bool** vim\_operator::pageup()
2. {
3. **HANDLE** hOut;
4. CONSOLE\_SCREEN\_BUFFER\_INFO scr;
5. hOut = GetStdHandle(STD\_OUTPUT\_HANDLE);    // 获取标准输出句柄
6. GetConsoleScreenBufferInfo(hOut, &scr);
7. COORD position = scr.dwCursorPosition;
8. **int** page = position.Y / height;
9. **int** row = position.Y % height;
10. **if** (page != 0)
11. {
12. page -= 1;
13. position.Y -= height;
14. **if** (position.X > num[page][row])
15. position.X = num[page][row];
16. SetConsoleCursorPosition(hOut, position);
17. }
18. SMALL\_RECT rect{ 0,page \* height, scr.srWindow.Right, (page + 1) \* height };
19. SetConsoleWindowInfo(hOut, **true**, &rect);
20. **return** **true**;
21. }
23. **bool** vim\_operator::pagedown()
24. {
25. **HANDLE** hOut;
26. CONSOLE\_SCREEN\_BUFFER\_INFO scr;
27. hOut = GetStdHandle(STD\_OUTPUT\_HANDLE);    // 获取标准输出句柄
28. GetConsoleScreenBufferInfo(hOut, &scr);
29. COORD position = scr.dwCursorPosition;
30. **int** page = position.Y / height;
31. **int** row = position.Y % height;
32. **if** (page != page\_num)
33. {
34. page += 1;
35. position.Y += height;
36. **if** (position.Y > row\_num)
37. position.Y = row\_num;
38. **if** (position.X > num[page][row])
39. position.X = num[page][row];
40. SetConsoleCursorPosition(hOut, position);
41. }
42. SMALL\_RECT rect{ 0,page \* height, scr.srWindow.Right, (page + 1) \* height };
43. SetConsoleWindowInfo(hOut, **true**, &rect);
44. **return** **true**;
45. }
46. 代码特点&问题解决
47. 代码特点
48. 利用自建临时文件实现撤销操作
49. 每个可能出现换行和翻页的函数都加以考虑
50. 及时释放内存（新开空间的释放和临时文件的释放）
51. 问题解决

第一次写这么大的程序，遇到了成堆的问题，在此简述几个主要问题及解决。

1. 在输入指令后键入回车以执行指令，由于指令是通过“cin”输入进去的，而cin以回车结束时程序将换行，将指令顶到倒数第二行，同时控制台的第一行也被顶到上一页中。

为了解决此问题，在每次涉及到最后一行键入指令后，都把光标直接移动到

（0，0）的位置（此过程需要耗费一定时间，肉眼可以看出来控制台的内容先向上窜了一行又立即恢复）

 SetConsoleCursorPosition(hOut, { 0,0 });  //还原原有输出窗口光标位置

1. 每次存储光标后的字符串时，都新开了空间char temp=new char()，理论上程序可以自动清除没有用的空间，但是只有在所有程序都结束时才清除，在没写delete[]temp的时候程序会报空间错误。
2. 由于没学过c++，类函数之间的调用不太熟练。开始的时候，作者把每个函数都定义为void型（毕竟要返回值也没用），但是在功能表函数normal\_function和insert\_function中调用其他函数时会报错，因此就把多有的类函数都改成了bool型。
3. 总结与反思
4. 总结

第一次写这么大的程序，能够实现许多功能已经很满足了。在完成作业的过程中通过查阅资料对于控制台函数有了较深的理解，而我们写的vim编辑器可以看作一些软件的底层逻辑，让我们了解了一些平日里“拿来就用”的软件功能具体是怎样实现的。

1. 反思

和身边同学相比我的代码偏长（手动流泪），可能是因为许多函数最开始的预处理部分都是相同的，单独拿出来写一个函数会省一些行数。同时，新建临时文件的做法不知道是否可取，可能会在整个过程中占用较多内存。