组号: 03

上海大学计算机工程与科学学院

实 验 报 告

（数据结构 1）

学　　期：2022-2023 年 1 季

组　　长：刘彦辰

学　　号：21121319

指导教师：朱能军

成绩评定：_____（教师填写）

二〇二二年 12 月 13 日

小组信息			
姓名	学号	贡献比	签名
刘彦辰	21121319	80%	
李睿凤	21121906	10%	
车心宇	21121928	10%	

实验概述	
实验零	（熟悉上机环境、进度安排、评分制度；确定小组成员）
实验一	**约瑟夫问题变种**
实验二	**列车车厢重排问题**
实验三	**KMP 模式串查找**
实验四	?

任务分配

姓名	职责
刘彦辰	代码实现; 部分报告撰写
李睿凤	部分报告撰写
车心宇	部分报告撰写

I. 项目演示

1. 运行

在项目内的 **.EXE** 文件夹中, 我们提供了 win x64 平台下的可执行文件 **03_WordScanner.exe** 和测试文件.

运行 **03_WordScanner.exe**, 根据提示输入目标文件和包含查找单词的模式文件, 如图1-1所示.

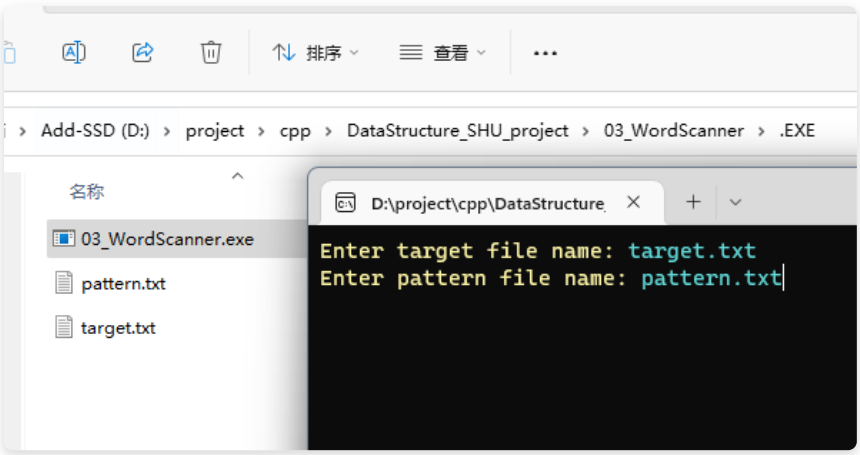

图 1-1

如果输入文件名不存在, 打开文件失败, 会出现如图1-2的提示:

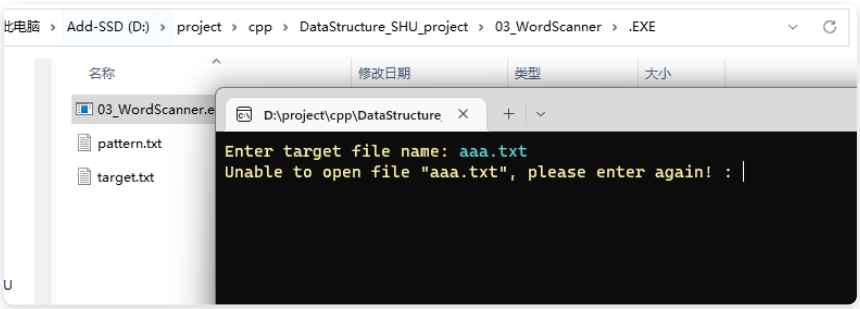

图 1-2

回车执行文字查找程序 (查找目标文件中所有的 "that", "of", "when"), 结果如图1-3所示.

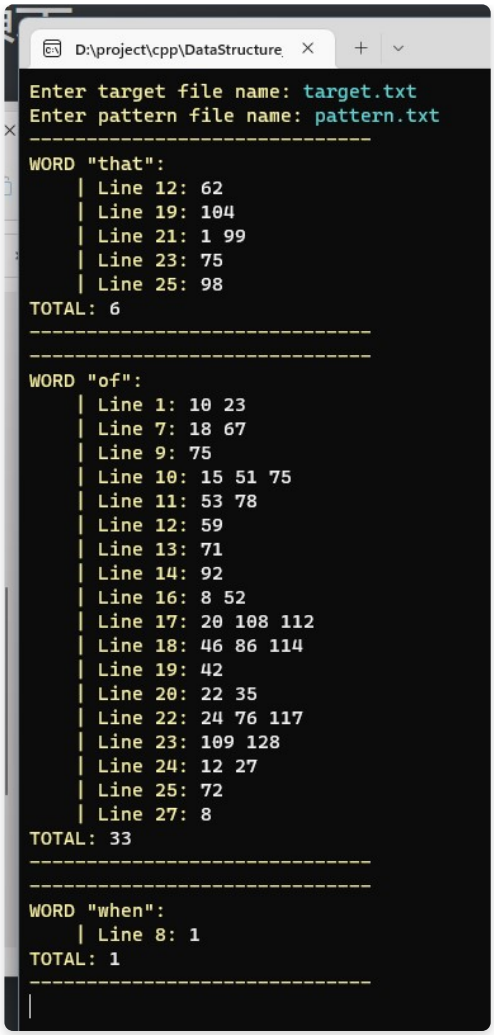

图 1-3

图1-3展示了目标文件中所有包含模式文件中模式串的位置, 例如在目标文件的第10行的第 15, 51 和 75 个字符处出现了模式串 "of".

每个模式串在目标文件中的总出现次数也在最后表明, 例如单词 "that" 在目标文件中共出现 6 次.

2. 测试文件说明

测试文件应为 .txt 格式文本文件.

测试文件需与 **03_WordScanner.exe** 置于同一目录下.

模式文件中的模式串需顶格书写并以回车分割, 如:

```
apple
 of banana
when
```

即表示查找三个模式串 "apple", " of banana" 以及 "when".

II. KMP 算法

问题描述：

对于两个字符串 P、T，找到 T 在 S 中第一次出现的起始位置，若 T 未在 S 中出现，则返回-1。

算法分析：

KMP 算法解决的问题是在字符串中的模式定位问题，也就是关键字搜索，是将$Brute-Force$算法的改进版本。我们称字符串为 P, 模式串为T。

$Brute-Force$ 算法是从左到右一个一个匹配，若过程中出现不匹配的字符，则将模式串右移并重新匹配。而KMP 的重点在于当出现不匹配的字符时，获得下一次匹配过程中 T 的初始比较字符的位置 (j) 。即：利用已经部分匹配这个有效信息，保持 i 指针不回溯，通过修改 j 指针，让模式串尽量地移动到有效的位置。

若模式串中最前面的 k 个字符和 j 之前的最后 k 个字符是一样的（如下图1）

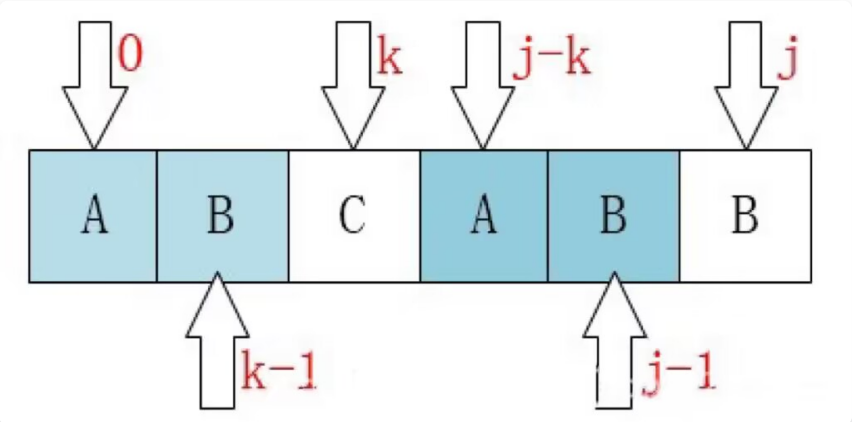

图 1-1

图1 字符串中已匹配的的子串的前缀和后缀

则可以直接把 j 移动到 k，而不需再比较前面的 k 个字符。

```
当T[i] != P[j]时
有T[i-j ~ i-1] == P[0 ~ j-1]
由P[0 ~ k-1] == P[j-k ~ j-1]
必然：T[i-k ~ i-1] == P[0 ~ k-1]
```

我们要计算每一个位置 j 对应的 k，用数组 $next$ 来保存，$next[j]=k$，表示当$T[i]!=P[j]$时，j 指针的下一个位置。

$next$ 数组表示的是: T 字符串中第 N 位的最长公共前后缀。

$next[j-1]$ 表示的是：前 j 位的最长公共前后缀。

$j=0$ 时候，定义$next[0]=0$，然后每次重新计算第 j 位最长公共前后缀时候，因为 $next[j-1]$ 描述的是前 j 位的最长公共前后缀，现在只要将第 j 位的字符 $T[j]$ 和第 $D[j-1]$ 的字符 $T[D[j-1]]$ 进行比较，若相同就是最长公共前后缀加1，若不相同就是0。

算法实现：

next数组

```
int* getNext(String pStr){
    int* next = new int[pStr.length()];
    for(int i = 0 ; i < pStr.length(); i++){
        if(i == 0){
            next[i] = 0;//next数组的第一个设置为0
        }else{
            int j = next[i - 1];
            if(pStr.charAt(j) == pStr.charAt(i)){
                next[i] = next[i - 1] + 1;
            }else{
                next[i] = 0;
            }
        }
    }
    return next;
}
```

KMP 函数

KMP 函数返回一个 `vector<int>` 类型对象, 其中储存了目标字符串中所有匹配了模式串的位置.

```
while(i < P.length()){
    if(P[i] == T[j]){//如果相同，两个下标同时往后延
        i++;
        j++;
        if(j == T.length()){
            result.push_back(i - j);
             j = next[j - 1];
        }
    }else{ //如果不相等，移动相应位置
        if(j == 0){ //如果模式串(T)下标为0，证明一开始都没有匹配成功
            i++; //字符串（P）下标往后移动一位，模式串（T）不变
        } else { //如果模式串（T）下标不是0
            j = next[j - 1]; // j从next数组的j-1开始匹配，i不变
        }
    }
    return result;
}
```

时间复杂度分析

设字符串 P 长度为n，模式串 T 长度为 m。

算法 `getNext(T)` 时间复杂度为 $O(m)$ 。算法`KMP(P,T)` 时间复杂度为 $O(m+n)$ 。

相比 Brute-Force 算法效率有了明显提高。

使用 KMP 统计单词出现次数

1. 将 $patternFile$ 的每个待查找的单词存入`vector<String> pattern`中。
2. 将 $targetFile$ 的文本内容分行存入`vector<String> file`中。
3. 每个待查找单词分行进行 KMP 算法查找，并输出。

```
for every pattern in patternVector
for every line in StringVector
    result = KMP(line, pattern)
    if result is not empty
       output result
```

III. C++ 文件读写

文章文件读取

1. 创建ifstream类的对象`targetFile`。
2. 输入将要打开的文件名到`targetName`中。
3. 调用`ifstream::open()`函数，打开该文件，在操作前调用`ifstream::fail()`函数进行检测，如果打开失败则要求重新输入。
4. 创建vector类的对象`file`,用来存储目标文件的内容。
5. 利用`String::getline()`逐行读取打开的文件，并存储在String变量`splitedLines`中。
6. 用`vector::push_back()`将`splitedLines`置于`file`的末尾。
7. 重复5、6操作，直到读完文件。

目标词汇文件读取

1. 创建ifstream类的对象`patterntFile`。
2. 输入将要打开的文件名到`patternName`中。
3. 调用`ifstream::open()`函数，打开该文件，在操作前调用`ifstream::fail()`函数进行检测，如果打开失败则要求重新输入。
4. 创建vector类的对象`pattern`,用来存储目标文件的内容。
5. 循环利用`String::getline()`逐行读取打开的文件，并存储在String变量`temp`中。
6. 用`vector::push_back()`将`temp`置于`pattern`的末尾。
7. 重复5、6操作，直到读完文件。

IV. String 类

1. 概述

`String` 的实现可以基于数组实现的线性表 `arrayList` .

自从 C++ 11 标准, `std::string` 需满足末尾为 `\0` 的规范.

为对标 `std::string` , 我们没有继承数组实现的线性表, 重新写了一份.

需实现的基础功能如图4-1所示.

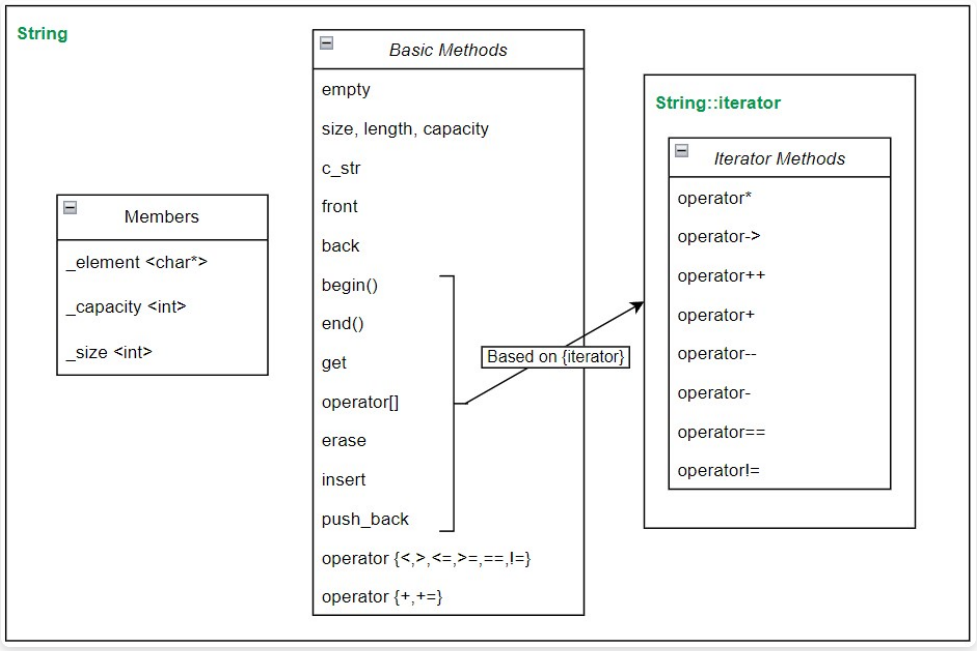

图 4-1

2. 末尾空字符

为满足末尾为 `\0` , 我们采用的策略是在管理容量 (capacity) 以及动态扩容时多预留一个位置;

同时每次插入结束手动将 `_element[_size]` 赋值为 `\0` .

这样最大程度避免了多余的操作.

3. 迭代器

为了符合STL规范, 我们额外设计了 `String` 的迭代器 `String::iterator` ;

在设计迭代器时应该注意不要继承 `std::iterator` , 而是自己定义各种 tag.

```
typedef std::bidirectional_iterator_tag iterator_category;
typedef char value_type;
typedef ptrdiff_t difference_type;
typedef char* pointer;
typedef char& reference;
```

4. 输入 / 输出

除了基础方法之外, 我们还额外添加了对输入输出运算符的重载:

```cpp
std::ostream& operator<<(std::ostream& out, const String& str);
std::istream& operator>>(std::istream& in, String& str);
```

为了方便读取一行内容 (类似 `std::getline`), 我们增加了 `String` 的成员函数 `getline` (这里有待改进, 应该重载 `std::getline` , 日后有空我会修改):

```cpp
std::istream& getline(std::istream& in) {
    std::string temp;
    std::getline(in, temp);
    (*this) = temp;
    return in;
}
```

由于我们会进行文件读写, 应当对 `std::istream` 和 `std::ifstream` 的继承关系熟练掌握. 图4-2 表明 `std::istream` 派生出 `std::ifstream` .

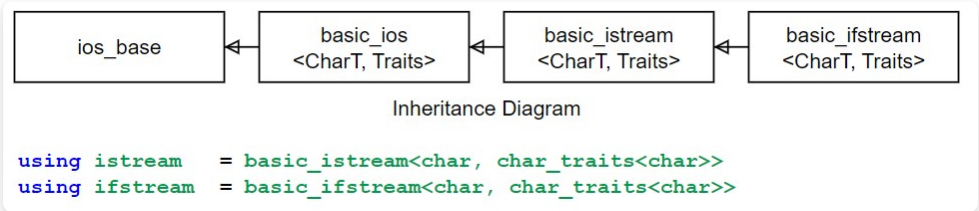

图 4-2

因此, `std::ifstream` 定义的对象可以传给 `std::istream` 的参数, 参考示例:

```cpp
#include <iostream>
#include <fstream>

std::ifstream myInputFile; // {myInputFile} is an std::ifstream object

void foo(std::istream& _obj); // declaration for some function

foo(myInputFile); // valid function call, pass {myInputFile} to {_obj}
```

V. 彩色输出

Tools 文件夹内包含一些个人编写的工具, 其中在 scui 命名空间内重写了 ostream 类和 istream 类, 写了 `cout` 和 `cin` 函数用来控制彩色输入 / 输出.

考察 `scui::cout()` 的声明和注释.

```cpp
// In "Tools/ColorIO.h"

/**
* @param fColor <7> white(DEFAULT), <0> black, <1> blue, <2> green, <3> red, <4> cyan, <5> pink, <6> yellow.
* @param bColor <0> black(DEFAULT), <1> blue, <2> green, <3> red, <4> cyan, <5> pink, <6> yellow, <7> white.
* @param fIntensity <true>(DEFAULT), <false>.
* @param bIntensity <false>(DEFAULT), <true>.
*/
scui::ostream& scui::cout(int fColor = 7, int bColor = 0, bool fIntensity = true, bool bIntensity = false);
```

可以看到, 通过直接调用以上函数能够进行彩色输出, 例如:

```cpp
#include "Tools/ColorIO.h"

scui::cout(7, 1) << "Hello";
```

可以在控制台内输出文字为高亮白色, 底色为蓝色的字符串 "Hello".

具体实现不在这里多做展开了, 可以通过阅读 Tools 文件夹内的相关文件了解.