

数据结构与算法

串

串

4. 1 串类型的定义

4. 2 字符串的实现

 4. 2. 1 字符串的C实现

 4. 2. 2 字符串的C++实现

4. 3 字符串模式匹配算法

 4. 3. 1 简单字符串模式匹配算法

 4. 3. 2 首尾字符串模式匹配算法

 4. 3. 3 KMP字符串模式匹配算法

4. 4 实例研究——文本编辑

4.1 串类型的定义

- 现在的计算机硬件结构主要反映数值计算的需要，在处理字符串数据时比处理整数和浮点数要复杂得多。要有效地实现字符串的处理，就要根据具体情况使用适合的存储结构。
- 串是字符串的简称，它的每一个数据元素由一个字符组成。串是一种特殊的线性表。

4.1 串类型的定义

$n (n \geq 0)$ 个字符的有限序列

$s = "a_1, a_2, \dots, a_n"$

串名: s

串值: $a_i (1 \leq i \leq n)$

串长: n

4.1 串类型的定义

- 串是由零个或多个字符组成的有限连续序列。
- 简单地说，串就是一串字符。一般记为 $s = "a_1a_2a_3 \dots a_n"$ 。其中 S 是串的名字，用双引号括起来的字符序列是串的值， $a_i (1 \leq i \leq n)$ 可以是字母、数字或其它字符。 n 是串中字符的个数，称为串的长度， $n=0$ 时的串称为空串 (Null String)。

4.1 串类型的定义

- 子串：字符串中任意个连续的字符构成的子序列称为该字符串的子串。空串是任何串的子串。
- 主串：包含子串的字符串称为主串。

4.1 串类型的定义

- 位置：一个字符在序列中的序号称为该字符在串中的位置。子串在主串中的位置则以子串的第一个字符在主串中的位置来表示。
- 串相等：字符串的长度相等而且各对应位置上的字符都相同。

4.1 串类型的定义

- 空串：由零个字符组成的串，空串中不包含任何字符，长度为0。
- 空格串：由一个或多个空格组成的串。它不等于空串，它的长度是串中包含的空格数。字符串中空格也算在串的长度中。

串的基本操作

1. void Copy(String ©, const String &original)

初始条件：串original已存在。

操作结果：将串original复制得到一个串copy。

2. bool Empty() const

初始条件：串已存在。

操作结果：如串为空，则返回true，否则返回false。

3. int Length() const

初始条件：串已存在。

操作结果：返回串的长度，即串中的字符个数。

串的基本操作

4. void Concat(String &addTo, const String &addOn)

初始条件：串addTo和addOn已存在。

操作结果：将串addOn联接到串addTo的后面。

5. String SubString(const String &s, int pos, int len)

初始条件：串存在，且 $0 \leqslant pos < s.Length()$, $0 \leqslant len \leqslant s.Length() - pos$ 。

操作结果：返回从第pos个字符开始长度为len的子串。

6. int Index(const String &target, const String &pattern, int pos = 0)

初始条件：目标串target和串pattern都存在，模式串pattern非空，且 $0 \leqslant pos < target.Length()$ 。

操作结果：返回目标串target中第pos个字符后第一次出现的模式串pattern的位置。

C如何存储字符串

在c中，字符串是以NULL结束的字符数组。

```
#include <stdio.h>
void main()
{
    char string[256]; int i;
    for(i=0; i<26; i++) string[i] = 'A' + i;
    string[i] = NULL;
    printf("the string contains %s\n", string);
    char str[] = "hello";
    printf("%s\n", str);
}
```

'A' 如何区别于"A"

- 字符串是0个或多个ASCII字符组成的序列，并以NULL结束。
- 在c中使用字符时可以使用字符的ASCII码值或将字符放在单引号内，例如'A'；另一方面，使用双引号时，例如"A"，c将创建包含指定字符并以NULL字符结束的字符串。C将它们区别存储，所以字符常量和字符串常量并不是相同的。



在字符串常量中表示引号

创建字符串时程序必须把所需要的字符放在双引号内，有时候可能要求字符串中包含双引号，为此可以使用转义符"\\"。

```
#include <stdio.h>
void main()
{
    char string[ ]= "\\" Stop!\\", he said. ";
}
```

4.2 字符串的实现

- 字符串的C语言实现
- 字符串的C++实现

4. 2. 1字符串的C实现

- 串的顺序存储
 - 定长顺序存储结构，属于静态存储结构。
 - 堆式顺序存储结构，属于动态顺序存储结构。
- 串的链式存储：块链存储表示

4. 2. 1字符串的C实现

定长顺序存储结构

- 用一组地址连续的存储单元存储串值的字符序列。在串的定长顺序存储结构中，按照预定定义的大小，为每个定义的串变量分配一个固定长度的存储区。
- 串的实际长度一般小于等于定义的长度，超过定义长度的串值则被舍去，称为截断。

4. 2. 1字符串的C实现

定长顺序存储结构

直接使用定长的字符数组来定义， 数组的上界预先给出：

```
#define maxstrlen 256
typedef char sstring[maxstrlen];
sstring s;    //s是一个可容纳255个字符的顺序串。
```

4. 2. 1字符串的C实现

定长顺序存储结构

- 一般可使用一个不会出现在串中的特殊字符在串值的尾部来表示串的结束。例如，C语言中以字符‘\0’表示串值的终结，这就是为什么在上述定义中，串空间最大值maxstrlen为256，但最多只能存放255个字符的原因，因为必须留一个字节来存放‘\0’字符。
- 若不设终结符，可用一个整数来表示串的长度，那么该长度减1的位置就是串值的最后一个字符的位置。此时顺序串的类型定义和顺序表类似。

4. 2. 1字符串的C实现

堆式顺序存储结构

- 多数情况下串的操作以整体形式参与，参与运算的串变量之间的长度相差较大，在操作过程中长度变化也比较大。
- 为串标量设定固定大小空间的数组不尽合理；克服这个弊病的方法就是不限定串长的最大长度，即动态分配串值的存储空间。

4. 2. 1字符串的C实现

堆式顺序存储结构

- 仍以一组地址连续的存储单元存放串值字符序列，但它们的存储空间是在程序执行过程中动态分配得到的。
- C语言中，存在一个称为堆（Heap）的自由存储区，可以为每个新产生的串动态分配一块实际串长所需的存储空间。

4. 2. 1字符串的C实现

堆式顺序存储结构

- 以堆分配存储表示串的特点：串变量的存储空间是在程序执行过程中动态分配而得。
- C语言中的串类型按此实现，利用alloc()为新生成的串分配一个实际所需的存储空间。

4.2.1 字符串的C实现

块链存储表示

- 和线性表的链式存储结构相似，也可以采用链表方式存储串值。
- 串的链式存储结构包含字符域和指针域的结点链接结构。优点是插入和删除运算方便。
- 但是由于串的结构的特殊性——串的数据元素是一个字符，只有8位二进制数，如果每个结点存放一个字符，每个结点的指针域所占空间比字符域大，有效空间利用率不高。
- 为了提高链式存储结构的有效空间利用率，采用块链结构的存储方法。

4.2.1 字符串的C实现

块链存储表示

- 块链结构：每个结点存放若干个字符，以减少链表中的结点数量。除头指针外，还附设尾指针指示链表中最后一个结点，并给出当前串的长度。这些都是为了操作的方便。
- 串值的链式存储结构对某些操作，比如联接操作等有一定的方便，但是由于串的特殊性，使得串的链式存储结构存储量大而且操作复杂，不太实用。

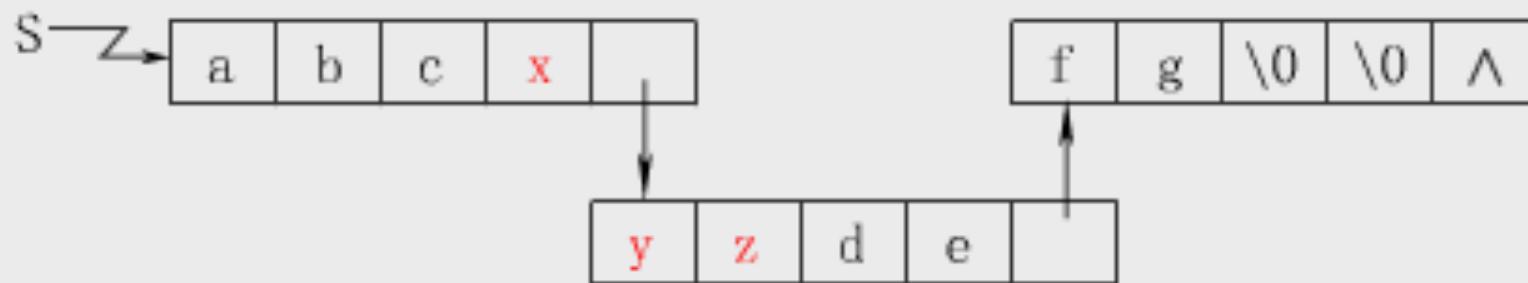
4. 2. 1 字符串的C实现



(a) 结点大小为1的链串S



(b) 结点大小为4的链串S



(c) 在图(b)中的第三个字符后插入“xyz”后的链串S

链串示意图

存储密度

若将多个字符存放在一个结点中，就可以提高存储密度。由于串中的字符数不一定是每个结点存放字符个数的整倍数，所以，需要在最后一个结点的空缺位置上填充特殊的字符。这种存储形式的优点是存储密度高于结点大小为1的存储形式；不足之处是做插入、删除字符的操作时，可能会引起结点之间字符的移动，算法实现起来比较复杂。

$$\text{存储密度} = \frac{\text{串值所占的存储单元}}{\text{实际分配的存储密度}}$$

4. 2. 2字符串的C++实现

- C++在头文件string中已含了一种安全的字符串实现
- 由于这个库没有包含在一些较老的C++编译器中，因此本节将设计自己的安全的String类，使用面向对象技术来克服C风格的串中存在的问题。

4. 2. 2字符串的C++实现

```
class String
{
protected:
    // 串实现的数据成员:
    char *strVal;           // 串值
    int length;              // 串长
public:
    // 抽象数据类型方法声明及重载编译系统默认方法声明:
    String();                // 构造函数
    virtual ~String();        // 析构函数
    String(const String &copy); // 复制构造函数
    String(const char *copy); // 从C风格串转换的构造函数
    String(LinkList<char> &copy); // 从线性表转换的构造函数
    int Length() const;       // 求串长度
    bool Empty() const;       // 判断串是否为空
    String &operator =(const String &copy); // 重载赋值运算符
    const char *CStr() const; // 将串转换成C风格串
    const char &String::operator [](int i) const; // 重载下标运算符
};
```

4.2.2 字符串的C++实现 - 串相关操作

```
String Read(istream &input);
    // 从输入流读入串
String Read(istream &input, char &terminalChar);
    // 从输入流读入串, 用terminalChar返回串结束字符
void Write(const String &s);
    // 输出串
void Concat(String &addTo, const String &addOn);
    // 将串addOn连接到addTo串的后面
void Copy(String &copy, const String &original);
    // 将串original复制到串copy
void Copy(String &copy, const String &original, int n);
    // 将串original复制n个字符到串copy
int Index(const String &target, const String &pattern,      int pos = 0);
    // 查找模式串pattern第一次在目标串target中从第pos个字符开始出现的位置
String SubString(const String &s, int pos, int len);
    // 求串s的第pos个字符开始的长度为len的子串
```

4.2.2 字符串的C++实现 - 串相关操作

```
bool operator ==(const String &first, const String &second);  
    // 重载关系运算符==  
bool operator <(const String &first, const String &second);  
    // 重载关系运算符<  
bool operator >(const String &first, const String &second);  
    // 重载关系运算符>  
bool operator <=(const String &first, const String &second);  
    // 重载关系运算符<=br/>bool operator >=(const String &first, const String &second);  
    // 重载关系运算符>=br/>bool operator !=(const String &first, const String &second);  
    // 重载关系运算符!=
```

串类部分成员函数的实现

1. 串构造函数(1)

```
String::String(const char *inString)
// 操作结果：从C风格串转换构造新串——转换构造函数
{
    length = strlen(inString);           // 串长
    strVal = new char[length + 1];        // 分配存储空间
    strcpy(strVal, inString);            // 复制串值
}
```

串类部分成员函数的实现

2. 串构造函数(2)

```
String::String(LinkList<char> &copy)
// 操作结果：从线性表转换构造新串——转换构造函数
{
    length = copy.Length();           // 串长
    strVal = new char[length + 1];    // 分配存储空间
    for (int i = 0; i < length; i++)
    { // 复制串值
        copy.GetElem(i + 1, strVal[i]);
    }
    strVal[length] = '\0';            // 串值以'\0'结束
}
```

串类部分成员函数的实现

3. 将C++串转换为C语言串

```
const char *String::CStr() const
    // 操作结果: 将串转换成C风格串
{
    return (const char *)strVal; // 串值类型转换
}
```

4. 串比较实现

```
bool operator ==(const String &first, const String &second)
{ // 操作结果: 重载关系运算符 ==
    return strcmp(first.CStr(), second.CStr()) == 0;
}
```

```
bool operator <(const String &first, const String &second)
{ // 操作结果: 重载关系运算符<
    return strcmp(first.CStr(), second.CStr()) < 0;
}
```

串类部分成员函数的实现

5. 进一步串操作示例

```
void Concat(String &addTo, const String &addOn)
    // 操作结果：将串addOn连接到addTo串的后面
{
    const char *cFirst = addTo.CStr(); // 指向第一个串
    const char *cSecond = addOn.CStr(); // 指向第二个串
    char *copy = new char[strlen(cFirst) + strlen(cSecond) + 1];
    // 分配存储空间
    strcpy(copy, cFirst);           // 复制第一个串
    strcat(copy, cSecond);          // 连接第二个串
    addTo = copy;                   // 串赋值
    delete []copy;                  // 释放copy
}
```

4.3 字符串模式匹配算法

- 子串的定位操作称为串的模式匹配。结果分为匹配成功和匹配不成功2种。
- 在串匹配中，一般将主串称为目标串，子串称之为模式串。设S为目标串，T为模式串，且不妨设：
 $S = "s_0 s_1 s_2 \dots s_{n-1}"$ $T = "t_0 t_1 \dots t_{m-1}"$

4.3 字符串模式匹配算法

- 串的匹配实际上是对合法的位置 $0 \leq i \leq n-m$ 依次将目标串中的子串 $s[i..i+m-1]$ 和模式串 $t[0..m-1]$ 进行比较，若 $s[i..i+m-1] = t[0..m-1]$ ，则称从位置*i*开始的匹配成功，亦称模式*t*在目标*s*中出现；
- 若 $s[i..i+m-1] \neq t[0..m-1]$ ，则称从位置*i*开始的匹配失败。

4.3 字符串模式匹配算法

- 上述的位置*i*又称为位移，当
 $s[i..i+m-1] = t[0..m-1]$ 时，*i*称为有效位移；当
 $s[i..i+m-1] \neq t[0..m-1]$ 时，*i*称为无效位移。
- 这样，串匹配问题可简化为是找出某给定模式T在一给定目标T中首次出现的有效位移。

4.3.1 简单字符串模式匹配算法

- 一种简单直观的模式匹配算法是布鲁特-福斯算法，简称BF算法。
- 算法的基本思想是从主串的指定位置开始和模式的第一个字符比较，若相等则继续比较下一个字符；否则从主串的下一个开始再重新和模式的字符比较，直到最后结束。
- 在最好的情况下算法的时间复杂度为 $O(n+m)$ ，在最坏的情况下时间复杂度为 $O(n*m)$ 。

4.3.1 简单字符串模式匹配算法

第1趟 S a b b a b a
 P a b a

穷举的模式
匹配过程

第2趟 S a b b a b a
 P a b a

第3趟 S a b b a b a
 P a b a

第4趟 S a b b a b a
 P a b a
 Ö

4.3.1 简单字符串模式匹配算法

第1趟 S a b b a b a
 P a b a

穷举的模式
匹配过程

第2趟 S a b b a b a
 P a b a

第3趟 S a b b a b a
 P a b a

第4趟 S a b b a b a
 P a b a
 Ö

4.3.1 简单字符串模式匹配算法

```
int SimpleIndex(const String &T, const String &P, int pos = 0)
    // 操作结果：查找模式串P第一次在目标串T中从第pos个字符开始出现的位置
{
    int startPos = pos, i = pos, j = 0;
    while (i < T.Length() && j < P.Length()) {
        if (T[i] == P[j]) { // 继续比较后续字符
            i++; j++;
        }
        else { // 指针回退，重新开始新的匹配
            i = ++startPos; j = 0;
        }
    }
    if (j >= P.Length())
        return startPos; // 匹配成功
    else
        return -1; // 匹配失败
}
```

4.3.2 首尾字符串模式匹配算法

- 在简单字符串模式匹配算法中，分析匹配执行时间的最坏情况是每趟匹配过程都是在比较到模式串的最后一个字符时才发现不能匹配。
- 为避免在每趟匹配的最后一个字符时才发现不能匹配，可采用从模式串的两头分别进行比较的方法。

4.3.2 首尾字符串模式匹配算法

```
int FrontRearIndex(const String &T, const String &P, int pos = 0)
    // 操作结果：查找模式串P第一次在目标串T中从第pos个字符开始出现的位置
{
    int startPos = pos;
    while (startPos <= T.Length() - P.Length()) {
        int front = 0, rear = P.Length() - 1; // 模式串的首尾部字符位置
        while (front <= rear) {
            if (T[startPos + front] != P[front] || T[startPos + rear] != P[rear])
                break; // 模式串的首部或尾部字符不匹配，退出内循环
            else {front++; rear--;} // 首尾部字符匹配，重新定位新的首尾部字符
        }
        if (front > rear) return startPos; // 匹配成功
        else ++startPos; // 首部或尾部字符不匹配，重新查找匹配起始点
    }
    return -1; // 匹配失败
}
```

4.3.3 KMP字符串模式匹配算法

- 造成BF算法慢的主要原因是回溯，而这些回溯并不是必要的。
- 改进算法叫做KMP算法，在 $O(n+m)$ 的时间数量级上完成串的模式匹配操作。
- 改进的主要思想是，每一趟匹配过程中出现字符比较不等时，不需要回溯 i 指针，而是利用已经得到的部分匹配结果将模式串向右滑动尽可能远的位置，继续进行比较。
- 改进算法的最大特点是不用回溯，在处理外部输入文件时比较有优势。

4.3.3 KMP字符串模式匹配算法

KMP算法分析

- 一般情况，假设主串为 $s_1s_2\cdots s_n$ ，模式串为 $p_1p_2\cdots p_m$ 。为了避免回溯问题，需要解决：匹配过程中，主串第*i*个字符与模式串中第*j*个字符比较不等时，主串中的第*i*个字符应该与模式串中的哪个字符比较。也就是说模式串应该向右移动多大的步长。
- 假设已经知道了模式串应该向右移动多大的步长，也就是知道了 $\text{next}[j]$ 的值，那么可以设计匹配算法。

4.3.3 KMP字符串模式匹配算法

KMP匹配算法

- 设 s 为目标串， t 为模式串，设 i 指针和 j 指针分别指向目标串和模式串中待比较的字符，开始时都为0
- 如果 $s_i = t_j$ ，则 i 和 j 分别加1；反之 i 不变， j 的值回到 $\text{next}[j]$ ，再对 s_i 和 t_j 进行比较。
- 依次类推，直到出现下面2种情况之一。
 - j 回到某个 $\text{next}[j]$ 时有 $s_i = t_j$ ，则指针值各加1，继续匹配。
 - j 回到0，并且匹配失败，此时 $i+1$ ，开始 s_{i+1} 和 t_0 进行比较。

4.3.3 KMP字符串模式匹配算法

next[j]的求解

- 回到模式串应该向右移动多大的步长这个问题。如果主串第 i 个字符与模式中第 j 个字符不等，若有 $\text{next}[j]=k$ ，那么 k 满足下面的约束。
 - $p_1 \cdots p_{k-1} = p_{j-k} \cdots p_{j-1}$
 - $0 < k < j$ ，并且没有 $k' > k$ 满足上面的等式
- $\text{next}[j]$ 的值只与模式串本身有关，可以用递推的方式求得其所有值。把串既作为模式串又作为主串，在 $\text{next}[1]=-1$ 的条件下递推求得所有的 $\text{next}[j]$ 值。

4.3.3 KMP字符串模式匹配算法

S $s_1 \cdots s_{i-j-1} s_{i-j}$ \parallel $s_{i-j+1} s_{i-j+2} \cdots s_{i-1} s_i$ \parallel $s_{i+1} \cdots s_n$
P $\quad p_1 \quad p_2 \quad \cdots \quad p_{j-1} \quad p_j \quad p_{j+1} \cdots p_m$

则有 $s_{i-j+1}s_{i-j+2}\cdots s_{i-1} = p_1p_2\cdots p_{j-1}$ (1)

为使模式 P 与目标 S 匹配，必须满足

$$p_1p_2\cdots p_{j-1}p_j\cdots p_m = s_{i-j+1}s_{i-j+2}\cdots s_{i-1}s_i\cdots s_{i-j+m}$$

如果 $p_1\cdots p_{j-2} \neq p_2p_3\cdots p_{j-1}$ (2)

由(1)(2)则立刻可以断定

$$p_1\cdots p_{j-2} \neq s_{i-j+2}s_{i-j+3}\cdots s_{i-1}$$

下一趟必不匹配

4.3.3 KMP字符串模式匹配算法

同样，若 $p_1 p_2 \cdots p_{j-3} \neq p_3 p_4 \cdots p_{j-1}$

则再下一趟也不匹配，因为有

$$p_1 \cdots p_{j-3} \neq s_{i-j+3} \cdots s_{i-1}$$

直到对于某一个“k”值，使得

$$p_1 \cdots p_k = p_{j-k} p_{i-k+1} \cdots p_{j-1}$$

且

$$p_1 \cdots p_{k-1} = p_{j-k+1} p_{j-k+2} \cdots p_{j-1}$$

则

$$\begin{aligned} p_1 \cdots p_{k-1} &= s_{i-k+1} s_{i-k+2} \cdots s_{i-1} \\ &\quad \parallel \quad \parallel \quad \parallel \\ &\quad p_{j-k+1} p_{j-k+2} \cdots p_{j-1} \end{aligned}$$

模式右滑 $j-k$ 位

4.3.3 KMP字符串模式匹配算法

next[j]的求解

假设当模式中第 j 个字符与主串中相应字符不匹配时，可以拿第 k 个字符来继续比较，则令 next[j]=k

next 函数定义：

-1

$\text{next}[j] = \max \{k \mid 0 < k < j \text{ 且 } p_0 \cdots p_{k-1} = p_{j-k} \cdots p_{j-1}\}$

0

当 $j=0$ 时

当此集合不空时
其他情况

运用KMP算法的匹配过程

第1趟	目标	a c a b a a b a a b c a c a a b c	
	模式	a b a a b c a c	
	x		next(1) = 0
第2趟	目标	a c a b a a b a a b c a c a a b c	
	模式	a b a a b c a c	
	x		next(0) = -1
第3趟	目标	a c a b a a b a a b c a c a a b c	
	模式	a b a a b c a c	
	x		next(5) = 2
第4趟	目标	a c a b a a b a a b c a c a a b c	
	模式	a b a a b c a c	Ö

4.4 实例研究——文本编辑

- 文本编辑程序用于源程序的输入和修改，公文书信、报刊和书籍的编辑排版等。
- 常用的文本编辑程序有Edit、WPS、Word等。
- 文本编辑的实质是修改字符数据的形式和格式。
- 虽然各个文本编辑程序的功能不同，但基本操作是一样的，都包括串的查找、插入和删除等。
- 为了编辑方便，可以把文本当作一个字符串，文本分为若干行。我们把文本串的子串，行是页的子串，称为文本串，页是文本的子串。

4.4 实例研究——文本编辑操作

- R: 读取文本文件到缓冲区中，缓冲区中以前的任何内容将丢失，
当前行是文件的第一行。
- W: 将缓冲区的内容写入文本文件，当前行或缓冲区均不改变。
- I: 插入单个新行，用户必须在恰当的提示符的响应中键入新行并提供其行号。
- D: 删除当前行并移到下一行。
- F: 从当前行开始，查找包含有用户请求的目标串的第一行。
- C: 将用户请求的字符串修改成用户请求的替换字符串，仅在当前行中有效。
- Q: 退出编辑器。

Ö

4.4 实例研究——文本编辑操作

H: 显示解释所有命令的帮助消息，程序也接受?作为H的替代者。

N: 下一行，在缓冲区中进一行。

P: 上一行，在缓冲区中退一行。

B: 开始，到缓冲区的第一行。

E: 结束，到缓冲区的最后一行。

G: 到缓冲区中用户指定的行号。

V: 查看缓冲区的全部内容，显示到终端上。

Ö

主程序

主程序声明一个称为text的Editor对象并重复运行text的Editor方法，从用户处得到命令并处理这些命令。下面是得到的程序：

```
int main(void)
// 操作结果：读输入文件各行到文本缓存中，执行简单的行
// 编辑，并写文本缓存到输出文件中
{
    char infName[256], outfName[256];
    cout << "输入文件名(缺省: file_in.txt)：" ;
    strcpy(infName, Read(cin).CStr());
    Editor text(infName);
    string cmd;
    while (cmd != "q") {
        cout << "命令：" ;
        cin >> cmd;
        if (cmd == "w") {
            cout << "输出文件名：" ;
            cin >> outfName;
            text.write(outfName);
        } else if (cmd == "r") {
            cout << "输出文件名：" ;
            cin >> outfName;
            text.read(outfName);
        } else if (cmd == "l") {
            cout << text;
        } else if (cmd == "s") {
            cout << "输入搜索字符串：" ;
            cin >> str;
            cout << "输出替换字符串：" ;
            cin >> rep;
            text.replace(str, rep);
        }
    }
}
```

主程序

```
if (strlen(infName) == 0) {          // infName为空
    strcpy(infName, "file_in.txt");
}
cout << "输出文件名(缺省: file_out.txt):";
strcpy(outfName, Read(cin).CStr());
if (strlen(outfName) == 0) {          // outfName为空
    strcpy(outfName, "file_out.txt");
}
Editor text(infName, outfName);      // 定义文本对象
while (text.GetCommand()) {          // 接收并执行用户操作命令
    text.RunCommand();
}
system("PAUSE");                   // 调用库函数system()
return 0;                          // 返回值0, 返回操作系统
}
```

文本编辑类Editor

```
// Editor类包含以字符串（行）为数据元素的双向链表textBuffer
class Editor
{
private:
    // 文本编辑类的数据成员:
    DblLinkedList<String> textBuffer;      // 文本缓存
    int curLineNo;                          // 当前行号
    ifstream infile;                      // 输入文件
    ofstream outfile;                     // 输出文件
    char userCommand;                     // 用户命令
```

文本编辑类Editor

```
// 辅助函数:  
StatusCode NextLine(); // 转到下一行  
StatusCode PreviousLine(); // 转到前一行  
StatusCode GotoLine(); // 转到指定行  
StatusCode InsertLine(); // 插入一行  
StatusCode ChangeLine();  
  
// 替换当前行的指定文本串  
void ReadFile(); // 读入文本文件  
void WriteFile(); // 写文本文件  
void FindString(); // 查找串  
void View(); // 查看(显示)缓冲区  
  
public:  
    // 方法声明:  
    Editor(char infName[], char outfName[]); // 构造函数  
    bool GetCommand(); // 读取操作命令字符userCommand  
    void RunCommand(); // 运行操作命令  
};
```

练习

一个串的长度为 $n(n>0)$ ，它的子串有多少个？后缀子串有多少个？

答案： $n(n+1)/2+1$; $n+1$;

注意前缀子串、真子串（除本身以外的其他子串）的概念。};

练习

设串S和T都采用堆式存储，设计一个算法，求它们的最长公共子串

```
void maxsubstr(Hstring & S, Hstring & T, Hstring & R) {
    int i=0, j=0, maxi=0, ad;
    while(i+k<S.n) {
        j=0;
        while(j<T.n) {
            if(S.ch[i]==T.ch[j]) {
                k=1;
                while(i+k<S.n && j+k<T.n)
                    if(S.ch[i+k]==T.ch[j+k]) k++;
                    else break;
                if(k>maxi) {ad=i; maxi=k;}
                j=j+k;
            }
            else j++;
        }
        i++;
    }
    for(i=ad, j=0; j<maxi; i++, j++)
        R.ch[j]=S.ch[i];
    R.ch[j]='\0'; R.n=maxi; R.maxsize=S.maxsize;
}
```

时间复杂度 $O(n*m)$

练习

把一个字符串中所有字符循环右移形成的新词称为原词的轮转词。比如 $T_1=abcd$, $T_2=cdab$, 互为轮转词。设计一个尽可能简单的算法，判断两个字符串是否为轮转词。

思路：长度不同，不能为轮转词；长度相同，先生成一个大字符串 $T_3=T_2+T_2$ ，判断 T_1 是否为 T_3 的子串即可。

练习

把一个字符串中所有字符循环右移形成的新词称为原词的轮转词。比如 $T_1=abcd$, $T_2=cdab$, 互为轮转词。设计一个尽可能简单的算法，判断两个字符串是否为轮转词。

思路：长度不同，不能为轮转词；长度相同，先生成一个大字符串 $T_3=T_2+T_2$ ，判断 T_1 是否为 T_3 的子串即可。

设计一个递归算法，将整数字符串转换为整数（“1234\0” \rightarrow 1234）

练习

设计一个递归算法，将整数字符串转换为整数
（“1234\0” ->1234）

思路：先对低位直接转换，再针对前面部分递归执行转换

```
int stringToInt(char * s, int start, int finish)
{
    if (start > finish) return 0;
    int num = s[finish];
    if (start == finish) return num -
        48;
    return stringToInt(s, start,
        finish - 1)*10 + num - 48;
}

int stringToIntmain(char *s)
{
    int i, j, k;
    char sign;
    sign = (s[0] == '-' ) ? '-' : '+';
    i = j = (s[0] == '-' ) ? 1 : 0;
    while (s[i] != '\0') i++;
    k = stringToInt(s, j, i-1);
    if (sign != '-') return k;
    else return -k;
}
```