CHAPTER 3

第3章 字符串和矩阵

3.1 字符串

3.1.1 字符串的按需(堆)存储结构

```
//HString.h 字符串的类(HString 类)
#ifndef _HSTRING_H_
#define _HSTRING_H_
class HString                    //只一种类型(char),不需要模板
{
private://两个私有数据成员(见图 3-1)
    char *ch;                    //若是非空串,则按串长分配存储区; 空串 ch 为 NULL
    int length;                  //串长度
public://3 个构造函数,1 个析构函数,14 个公有成员函数
    HString()
    {                            //构造函数一,产生空串(见图 3-2)
        ch = NULL;
        length = 0;
    }
```

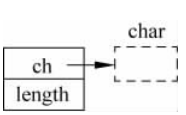

图 3-1 HString 类的两个数据成员

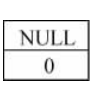

图 3-2 构造函数一生成的空串

```
    HString(const char * str)
    {//构造函数二,产生与字符串常量 str 字符相同的串(见图 3-3)
        length = strlen(str);
        ch = new char[length];
        assert(ch!= NULL);                  //分配串存储空间失败则退出
        for(int i = 0; i < length; i++)     //复制 str 到 ch
            ch[i] = str[i];
```

```
}
HString(const HString &S)  //const 限定 S 不能被改变
{//构造函数三,产生与字符串的类对象 S 相同的对象(见图 3-4)
    length = S.length;                  //复制串长
    ch = new char[length];              //分配与串 S 一样的存储空间
    assert(ch!= NULL);                  //分配串存储空间失败则退出
    for(int i = 0; i < length; i++)     //从第一个字符到最后一个字符
        ch[i] = S[i];                   //逐一复制字符(重载[ ])
}
```

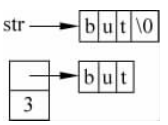

图 3-3　构造函数二生成的串

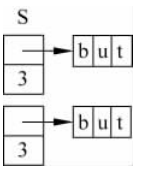

图 3-4　构造函数三生成的串

```
~HString()
{                                       //析构函数,清空串(见图 3-2)
    ClearString();
}
void ClearString()
{                                       //清空串(见图 3-2)
    if(ch!= NULL)                       //串不空
    {
        delete[ ]ch;                    //释放 ch 所指空间
        ch = NULL;                      //ch 不指向任何存储空间
    }
    length = 0;
}
void StrAssign(const char * str)
{//产生与字符串常量 str 字符相同的串(见图 3-3)
    ClearString();                      //释放原有存储空间
    length = strlen(str);               //str 的长度
    if(length)                          //str 的长度不为 0
    {
        ch = new char[length];          //分配串存储空间
        assert(ch!= NULL);              //分配串存储空间失败则退出
        for(int j = 0; j < length; j++) //分配串存储空间成功后,复制串 str[ ]
            ch[j] = str[j];
    }
}
void StrCopy(const HString &S)          //const 限定 S 不能被改变
{                                       //复制串 S(见图 3-4)
    ClearString();                      //释放原有存储空间
    length = S.length;                  //复制串长
    ch = new char[S.length];            //分配与串 S 一样的存储空间
    assert(ch!= NULL);                  //分配串存储空间失败则退出
    for(int i = 0; i < S.length; i++)   //从第一个字符到最后一个字符
        ch[i] = S[i];                   //逐一复制字符(重载[ ])
```

```
}
bool StrEmpty()const
{                                          //若串为空,则返回 true; 否则返回 false
    return length == 0;                    //空串标志
}
int StrCompare(const HString &S)const
{//若串>串 S,则返回值> 0; 若串 = 串 S,则返回值 = 0; 若串<串 S,则返回值< 0
    for(int i = 0; i < length && i < S.length; i++)  //在有效范围内
        if(ch[i]!= S[i])                   //逐一比较字符
            return ch[i] - S[i];           //不相等,则返回两字符 ASCII 码之差
    return length - S.length;              //在有效范围内字符相等,返回长度之差
}
int size()const
{                                          //返回串的元素个数,称为串的长度
    return length;
}
void Concat(const HString &S1, const HString &S2)
{                                          //返回由串 S1 和串 S2 连接而成的新串(见图 3-5)
    int i;
    ClearString();                         //释放串原有存储空间
    length = S1.length + S2.length;        //设置串的长度
    ch = new char[length];                 //分配串的存储空间
    assert(ch!= NULL);                     //分配串存储空间失败则退出
    for(i = 0; i < S1.length; i++)
    //将串 S1 的字符逐一复制给串
        ch[i] = S1[i];
    for(i = 0; i < S2.length; i++)     //将串 S2 的字符逐一复制给串(接在串 S1 的字符之后)
        ch[S1.length + i] = S2[i];
}
HString substr(int pos, int len)const
{//返回串的[pos]起长度为 len 的子串,不成功返回空串(见图 3-6)
    HString Sub;
    if(!(pos < 0 || pos > = length || len < = 0 || len > length - pos))
    //pos 和 len 的值合理
    {
        Sub.length = len;                  //串 Sub 的长度
        Sub.ch = new char[len];            //分配串 Sub 的存储空间
```

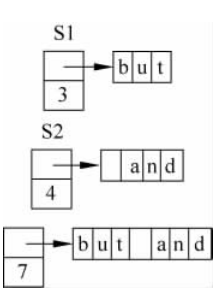

图 3-5　Concat()函数示例

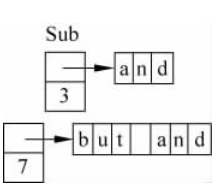

图 3-6　SubString()函数示例 pos＝4,len＝3

```
        assert(Sub.ch!= NULL);              //分配串存储空间失败则退出
        for(int i = 0; i < len; i++)
            Sub.ch[i] = ch[pos + i];
            //将串[pos]起长度为 len 的子串的字符逐一复制给串 Sub
    }
    return Sub;
}
bool StrInsert(int pos, const HString &S)
{//0≤pos≤length。在串的[pos]之前插入串 S。成功返回 true; 否则返回 false(见图 3-7)
    int i;
    if(pos < 0 || pos > length)  //Pos 不合法
        return false;
    if(S.length)                 //S 非空
    {
        char * p = new char[length + S.length];     //重新分配存储空间
        assert(p!= NULL);                           //重新分配存储空间失败则退出
        for(i = 0; i < pos; i++)                    //复制串的前半部分给 p
            p[i] = ch[i];
        for(i = 0; i < S.length; i++)               //复制 S 给 p
            p[pos + i] = S[i];
        for(i = pos; i < length; i++)               //复制串的后半部分给 p
            p[i + S.length] = ch[i];
        length += S.length;                         //更新串的长度
        delete[ ]ch;                                //释放原存储空间
        ch = p;                                     //指向新存储空间
    }
    return true;
}
bool StrDelete(int pos, int len)
{//从串中删除[pos]起长度为 Len 的子串(见图 3-8)
    int i;
    char * p;
    if(length < pos + len)              //pos 和 len 的值超出范围
        return false;
    p = new char[length - len];         //重新分配串的存储空间(减少)
    assert(p!= NULL);                   //重新分配存储空间失败则退出
    for(i = 0; i < pos; i++)            //复制串的前半部分给 p
        p[i] = ch[i];
    for(i = pos; i < length - len; i++) //复制串的后半部分给 p
```

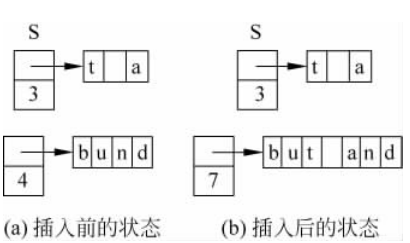

图 3-7　StrInsert()函数示例(pos＝2)

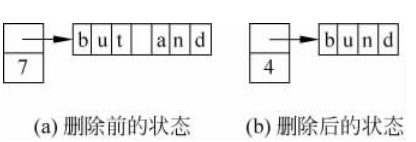

图 3-8　StrDelete()函数示例(pos＝2,len＝3)

```
        p[i] = ch[i + len];
    length -= len;                          //更新串的长度
    delete[ ]ch;                            //释放原存储空间
    ch = p;                                 //指向新存储空间
    return true;
}
int Index(const HString &S, int pos)const
{//若串中[pos]起存在与 S 相等的子串,
 //则返回第一个相等子串的起始位置;否则返回 -1(蛮力匹配算法)
    HString sub;
    if(pos >= 0)
        for(int i = pos; i <= length - S.length; i++)
        {//i 从串的[pos]到倒数第 S.length 个字符
            sub = substr(i, S.length);
            //子串 sub 是从串的[i]起,长度为 S.length 的子串
            if(sub.StrCompare(S) == 0) //子串 sub 等于模式串 S
                return i;               //返回模式串 S 的第一个字符在串中的位置
        }
    return -1;                          //串中不存在与模式 S 相等的子串
}
bool Replace(const HString &T, const HString &V)
{//初始条件: 串 T 和 V 存在,串 T 是非空串
 //操作结果: 用串 V 替换串中出现的所有与串 T 相等的不重叠的子串
    if(T.length == 0)                   //T 是空串
        return false;
    int i = 0;                          //从串的[0]起查找串 T
    while(i > -1)
    {
        i = Index(T, i);                //结果 i 为从上一个 i 之后找到的子串 T 的位置
        if(i > -1)                      //串中存在串 T
        {
            StrDelete(i, T.length);     //删除该串 T
            StrInsert(i, V);            //在原串 T 的位置插入串 V
            i += V.length;              //在插入的串 V 后面继续查找串 T
        }
    }
    return true;
}
char operator[ ](int i)const
{                                       //重载下标操作符
    return ch[i];
}
HString& operator = (const HString &L)
{                                       //重载赋值运算符
    if(this != &L)                      //没出现 L = L 的情况
    {
        if(ch != NULL)                  //不是空串
            delete ch;                  //释放 ch 原有存储空间
```

```
            ch = new char[L.length];          //根据 L.length 重新分配空间
            assert(ch!= NULL);                //存储分配失败则退出
            for(int i = 0; i < L.length; i++)
                ch[i] = L[i];                 //逐个复制数据
            length = L.length;
        }
        return * this;
    }
    void Output( ostream& out )
    {                                         //输出字符串
        for(int i = 0; i < length; i++)
            out << ch[i];
    }
};
ostream& operator << (ostream& out, HString& str)
{                                             //在类外重载输出流操作符
    str.Output(out);
    return out;
}
# endif

//Main3 - 1.cpp 验证串 HString 类的成员函数
# include "C.h"                               //文件包含宏命令
# include "HString.h"                         //HString 类
void main()
{
    int i;
    char c, * p = "God bye!", * q = "God luck!";
    HString t, s(p),r("Good luck!");
    //空串 t,同字符串数组 p 的串 s,用字符串常量构造的串 r
    cout <<"串 s 为"<< s << endl;             //输出串 s
    cout <<"s[4] = "<< s[4]<< endl;
    cout <<"串 r 为";
    r.Output(cout);                           //输出串 r
    HString u(r);                             //用 String 类对象 r 构造的串 u
    cout << endl <<"串 u 为"<< u << endl;     //输出串 u
    t.StrAssign(p);                           //将字符串 p 的内容赋给 t
    cout <<"串 t 为"<< t << endl;             //输出串 t
    cout <<"t 的串长为"<< t.size()<<",串空否?";
    cout << boolalpha << t.StrEmpty()<< endl;
    s.StrAssign(q);                           //将字符串 q 的内容赋给 s
    cout <<"串 s 为"<< s << endl;             //输出串 s
    i = s.StrCompare(t);                      //比较串 s 和串 t 的大小
    if(i < 0)
        c = '<';
    else if(i == 0)
        c = ' = ';
    else
        c = '>';
    cout <<"串 s"<< c <<"串 t"<< endl;
    r.Concat(t, s);                           //连接串 t 和串 s,得到串 r
```

```
    cout <<"串 t 连接串 s 产生的串 r 为"<< r << endl; //输出串 r
    s.StrAssign("oo");                              //将字符串"oo"赋给 s
    cout <<"串 s 为"<< s << endl;                   //输出串 s
    t.StrAssign("o");                               //将字符串"o"赋给 t
    cout <<"串 t 为"<< t << endl;                   //输出串 t
    r.Replace(t, s);                                //将串 r 中和串 t 相同的子串用串 s 代替
    cout <<"把串 r 中和串 t 相同的子串用串 s 代替后,串 r 为"<< r << endl;  //输出串 r
    s.ClearString();                                //清空串 s
    cout <<"串 s 清空后,串长为"<< s.size()<<",串空否?";
    cout << boolalpha << s.StrEmpty()<< endl;
    s = r.substr(5,4);                              //生成串 s 为从串 r 的[5]起的 4 个字符
    cout <<"串 s 为从串 r 的[5]起的 4 个字符,长度为"<< s.size()<<",串 s 为"<< s << endl;
    t.StrCopy(r);                                                  //由串 r 复制得串 t
    cout <<"由串 r 复制得串 t,串 t 为"<< t << endl;                //输出串 t
    t.StrInsert(5, s);                                             //在串 t 的[5]前插入串 s
    cout <<"在串 t 的[5]前插入串 s 后,串 t 为"<< t << endl;        //输出串 t
    t.StrDelete(0, 5);                                             //从串 t 的[0]起删除 5 个字符
    cout <<"从串 t 的[0]起删除 5 个字符后,串 t 为"<< t << endl; //输出串 t
    cout << t.Index(s, 0)<<"是从串 t 的[0]起,和串 s 相同的第一个子串的起始位置"<< endl;
    cout << t.Index(s, 1)<<"是从串 t 的[1]起,和串 s 相同的第一个子串的起始位置"<< endl;
}
```

程序运行结果:

```
串 s 为 God bye!
s[4] = b
串 r 为 Good luck!
串 u 为 Good luck!
串 t 为 God bye!
t 的串长为 8,串空否?false
串 s 为 God luck!
串 s>串 t
串 t 连接串 s 产生的串 r 为 God bye!God luck!
串 s 为 oo
串 t 为 o
把串 r 中和串 t 相同的子串用串 s 代替后,串 r 为 Good bye!Good luck!
串 s 清空后,串长为 0,串空否?true
串 s 为从串 r 的[5]起的 4 个字符,长度为 4,串 s 为 bye!
由串 r 复制得串 t,串 t 为 Good bye!Good luck!
在串 t 的[5]前插入串 s 后,串 t 为 Good bye!bye!Good luck!
从串 t 的[0]起删除 5 个字符后,串 t 为 bye!bye!Good luck!
0 是从串 t 的[0]起,和串 s 相同的第一个子串的起始位置
4 是从串 t 的[1]起,和串 s 相同的第一个子串的起始位置
```

HString 类有三个构造函数,其中两个是带参数的。可根据情况选择初始 HString 类的对象是空串、由 char 数组指定的串和由 HString 类的对象复制的串。

HString 类中存储字符串的方式和 C++语言设置的存储字符串方式不同,串尾没有 0 作为串结束标志,而是单独用一个私有数据成员 length 存放串长。

练习 3-1

改写 HString.h：增加一个数据成员表示串的存储空间。存储空间和串长不一定相等，存储空间不够时，按当前空间的倍数增加。比较两种方法的优劣。

3.1.2　STL 的串结构

C++的标准模板库(STL)提供了串类的操作，称为 string(串)。其实在第 2 章中就用到了 string(作为文件名和表达式字符串)。

```
//Algo3-1.cpp STL 中的串
#include "C.h"                          //文件包含宏命令
void main()
{
    int i;
    char c, *p = "God bye!", *q = "God luck!";
    string t, s(p),r("Good luck!"); //空串 t,同字符数组 p 的串 s,用字符串常量构造的串 r
    cout <<"s = "<< s << endl;
    cout <<"r = "<< r << endl;
    string u(r);                        //用 string 类对象 r 构造的串 u
    cout <<"u = "<< u << endl;
    t.assign(p);                        //将字符串 p 的内容赋给 t
    cout <<"t = "<< t << endl;
    cout <<"t 的串长为"<< t.size()<<",t 的存储容量为"<< t.capacity()<<",串空否?";
    cout << boolalpha << t.empty()<< endl;
    s.assign(q);                        //将字符串 q 的内容赋给 s
    cout <<"s = "<< s << endl;
    i = s.compare(t);                   //比较串 s 和串 t 的大小
    if(i < 0)
        c = '<';
    else if(i == 0)
        c = ' = ';
    else
        c = '>';
    cout <<"串 s"<< c <<"串 t"<< endl;
    r = t + s;                          //连接串 t 和串 s,得到串 r
    cout <<"r = t + s = "<< r << endl;
    s = "oo";
    t = "o";                            //将字符串"o"赋给 t
    i =- 2;
    while(true)                         //永真循环
    {
        i = r.find(t, i + 2);           //从 r 的[i + 2]起查找 t
        if(i >= 0)                      //在 r 的[i + 2]起找到了 t
            r.replace(i, 1, s);         //将在 r 中找到的 t 用 s 代替
        else                            //在 r 的[i + 2]起没找到 t
            break;                      //跳出 while 永真循环
    }
    cout <<"r = "<< r << endl;
    s.erase();                          //清空 s
    cout <<"s 清空后,串长为"<< s.size()<<",串空否?";
```

```
    cout << boolalpha << s.empty()<< endl;
    s = r.substr(5, 4);                    //生成串 s 为从串 r 的[5]起的 4 个字符
    cout <<"串 s 为从串 r 的[5]起的 4 个字符,s = "<< s << endl;
    t = r;                                 //由串 r 复制得串 t
    cout <<"由串 r 复制得串 t,t = "<< t << endl;
    t.insert(5, s);                        //在串 t 的[5]前插入串 s
    cout <<"在串 t 的[5]前插入串 s 后,串 t = "<< t << endl;
    t.erase(0, 5);                         //从串 t 的[0]起删除 5 个字符
    cout <<"从串 t 的[0]起删除 5 个字符后,t = "<< t << endl;
}
```

程序运行结果:

```
s = God bye!
r = Good luck!
u = Good luck!
t = God bye!
t 的串长为 8,t 的存储容量为 31,串空否?false
s = God luck!
串 s>串 t
r = t + s = God bye! God luck!
r = Good bye! Good luck!
s 清空后,串长为 0,串空否?true
串 s 为从串 r 的[5]起的 4 个字符,s = bye!
由串 r 复制得串 t,t = Good bye! Good luck!
在串 t 的[5]前插入串 s 后,串 t = Good bye! bye! Good luck!
从串 t 的[0]起删除 5 个字符后,t = bye! bye! Good luck!
```

3.1.3 字符串的模式匹配算法

1. 朴素的模式匹配算法

朴素的模式匹配算法即 HString 类中 Index()成员函数,如图 3-9 所示:由主串的[pos]字符起,检验是否存在子串 S。首先令 i 等于 pos(i 为主串中当前待比较字符的位序),j 等于 0(j 为 S 中当前待比较字符的位序),如果主串的字符[i]与 S 的字符[j]相同,则 i、j 各加 1 继续比较,直至 S 的最后一个字符(找到)。如果在比较期间出现了不同(没找到),则令 i 等于 pos+1,j 等于 0,由 pos 的下一个位置起,继续查找是否存在子串 S。

该算法简单、容易理解。但主串的指针 i 总要回溯,特别是在如图 3-9 所示的有较多字符匹配而又不完全匹配的情况下,回溯得更多。这时,主串的每个字符要进行多次比较,显然效率较低。

2. KMP(D.E.Knuth-J.H.Morris-V.R.Pratt)算法和改进的 KMP 算法

如果能使主串的指针 i 不回溯,也就是使主串的每个字符只进行一次比较,效率会大为提高。这是可以做到的。仍以图 3-9 为例,当检测到主串中字符[i(终值)]与模式串 S 中字符[j(终值)]不匹配时,字符[i(终值)]之前的字符都是已知的。它们都与模式串 S 中相应的字符相同,故 i 可仍保持在终值处不动,j 回溯到子串 S 的某个字符处与[i]指示的字符继续进行比较。j 回溯到哪个字符是由子串 S 的模式决定的。KMP 算法根据子串 S 生成的

next 数组指示 j 应该回溯到的字符。next 数组的意义是这样的：如果 next[j]＝k，则当子串 S 的字符[j]与主串的字符[i]失配时，主串的字符[i]继续与 S 的字符[k]进行比较即可。设子串 S 为“abaabcac”，如图 3-10 所示。当 S 的字符[4]与主串的字符[i]失配时，主串的字符[$i-1$]一定是 a，和 S 的字符[3]相等。它也和 S 的字符[0]相等。这样，主串的字符[i]和 S 的字符[1]开始比较即可。所以，对于模式串“abaabcac”，next[4]＝1。

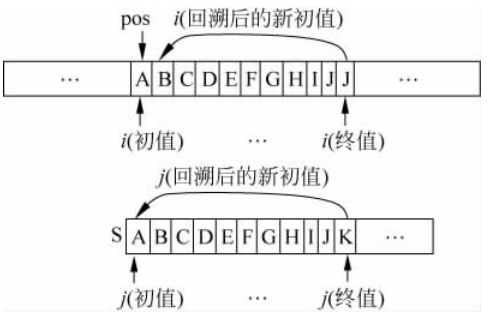

图 3-9　朴素的模式匹配算法

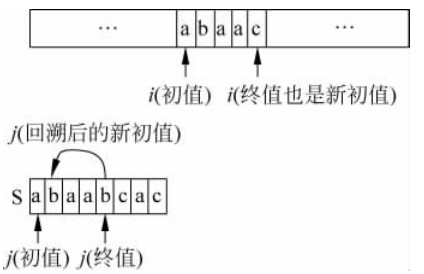

图 3-10　串“abaabcac”的数组 next[5]＝2 的由来

KMP 算法还有可改进之处。仍以图 3-10 为例：如果 S 的字符[4]与主串的字符[i]失配，则主串的字符[i]一定不是 b。这样，尽管主串的字符[$i-1$]是 a，和 S 的字符[0]相等，但主串的字符[i]肯定和 S 的字符[1](b)不相等。所以可令 next[4]＝0，使主串的字符[i]和 S 的字符[0]开始比较。这样使得模式串又向右移了一位，提高了匹配的效率。

Algo3-2.cpp 中的 KMP 类包括 KMP 算法和改进的 KMP 算法。KMP 类是带模板的，将主程序的第 1 行和第 2 行中的任意一行作为注释，就可以用于 string 类或 HString 类。二者的运行结果是一样的。

```
//Algo3-2.cpp KMP 和改进的 KMP 算法
#include "C.h"                          //文件包含宏命令
#include "HString.h"                    //HString 类
template<typename T> class KMP          //带模板的 KMP 类
{
private:
    vector<int> Next;
    void printNext()
    {                                   //输出 Next 向量
        for(int i=0; i<Next.size(); i++)
            cout<<Next[i]<<' ';
        cout<<endl;
    }
    void buildNext(T P, bool flag)
    {//KMP(flag=true)或改进的 KMP(flag=false)算法用到的模式串 P 的 next 表
        int i=0;                        //主串指针
        int j=-1;                       //模式串指针
        while(i<Next.size()-1)
            if(i<0 || P[i]==P[j])//初始或匹配
            {
                i++, j++;               //各+1 继续向后比较
                if(flag)                //KMP
                    Next[i]=j;
```

```
                else                    //改进的 KMP
                    Next[i] = (P[i]!= P[j]) ? j : Next[j];
            }
            else                        //失配
                j = Next[j];
    }
public:
    int Match(T P, T R, int pos, bool flag)
    {                                   //KMP(flag = true)或改进的 KMP(flag = false)算法
        T Q;
        Q = R.substr(pos,R.size() - pos);
        int n = Q.size(), i = 0;        //主串指针
        int m = P.size(), j = 0;        //模式串指针
        Next.assign(m, -1);             //分配 next 的长度(与模式串的相同)和初值(-1)
        buildNext(P, flag);             //构造模式串 P 的 Next 表
        while(j < m && i < n)           //自左向右逐个比对字符
        {
            if(0 > j || Q[i] == P[j])//若匹配,或 P 已移出最左侧(两个判断的次序不可交换)
                i++, j++;               //则转到下一字符
            else                        //不匹配
                j = Next[j];            //模式串右移
        }
        if(!flag)
            cout <<"改进的";
        cout <<"KMP 算法的 next 数组: ";
        printNext();                    //输出 Next 表
        return i - j + pos;
    }
};
void main()
{
    typedef string T;                   //主程序第 1 行
//  typedef HString T;                  //主程序第 2 行
    T P[2] = {"aaaab", "数据"}, Q[2] = {"aaabaaaab", "算法与数据结构"};  //可以比较汉字
    KMP < T > kmp;
    for(int j = 0; j < 2; j++)          //英文和汉字
        for(int i = 0; i < 2; i++)      //KMP 算法(i == 0)和改进的 KMP 算法(i == 1)
        {
            int k = 2;                  //pos 参数
            int pos = kmp.Match(P[j], Q[j], k, 0 == i);
            cout << Q[j]<<"从["<< k <<"]起和"<< P[j];
            if(Q[j].size()< pos + P[j].size())
                cout <<"不匹配。"<< endl;
            else
                cout <<"在主串的["<< pos <<"]处首次匹配。"<< endl;
        }
}
```

程序运行结果:

```
KMP 算法的 next 数组: -1 0 1 2 3
aaabaaaab 从[2]起和 aaaab 在主串的[4]处首次匹配。
改进的 KMP 算法的 next 数组: -1 -1 -1 -1 3
```

```
aaabaaaab 从[2]起和 aaaab 在主串的[4]处首次匹配。
KMP 算法的 next 数组：－1 0 0 0
算法与数据结构从[2]起和数据在主串的[6]处首次匹配。
改进的 KMP 算法的 next 数组：－1 0 0 0
算法与数据结构从[2]起和数据在主串的[6]处首次匹配。
```

运行 Algo3-2.cpp 也可以查找汉字字符串的匹配，汉字编码有以下两个特性。

(1) 两个字节(字符)表示一个汉字。

(2) 表示汉字的字符范围是从 128 到 255(转换成 unsigned char 类型)。

程序运行结果中 next[]的值为－1 时，并不是将主串的当前字符与模式串的字符[－1]进行比较(模式串也没有字符[－1])，而是主串当前字符的下一个字符与模式串的字符[0]进行比较。

由程序运行结果可见，改进的 KMP 算法的 next 数组的值都小于或等于 KMP 算法的 next 数组的值。值越小，模式串向右移动得越多，效率越高。

3. Boyer-Moore 算法

Boyer-Moore 算法有两个策略用于计算当模式串和主串不匹配时，主串不动，模式串应该向右移动几个字符。

(1) 坏字符策略。模式串一般较短，因此只包含字符集中的少数字符。如果主串当前与模式串比较的字符在模式串中没有，就称这个字符为“坏字符”。那就可以将模式串向右移动直到主串该字符的下一个字符与模式串的第一个字符对齐。为了提高效率，比较是从模式串的最右字符开始的，即由右向左比较。图 3-11 是应用坏字符策略的一个实例。模式串与主串的第一次比较是模式串的 m 与主串的空格比较，二者不等且模式串中没有空格(空格即是“坏字符”)，则模式串向右移动，使其第一个字符 r 与主串空格后面的字符 a 对齐再进行比较；如果主串的当前比较字符与模式串对应的字符不相等，而这个字符是模式串中的字符(不是坏字符)，则模式串向右移动，使得模式串中与该字符相同的字符中最右边的那个与该字符对齐，这样不会错过可能的匹配机会。如图 3-11 中模式串与主串的第 9 次比较，模式串的 m 与主串的 o 不等，但 o 是模式串中的字符，则令模式串中最右边的 o 与主串当前比较的 o 对齐，开始第 10 次比较。在模式串与主串不相等时主串的当前字符应该和模式串的哪个字符对齐的信息存于向量 BC 中，BC 的长度等于字符集的长度。图 3-12 显示了模式串为“room”时部分 BC[]的值(其余的值都是-1)。还可能出现的一种情况是 BC[]确定的位置在当前比较字符的右边，显然不能使模式串向左移。出现这种情况时要将模式

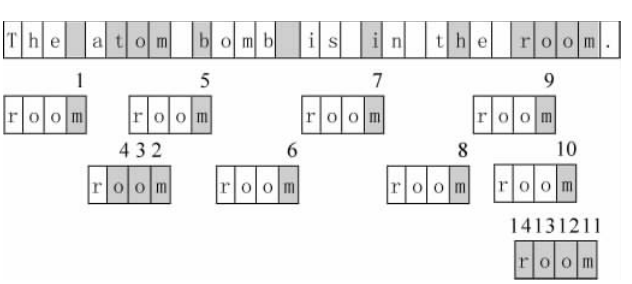

图 3-11　坏字符策略示例

串向右移动一个字符,如图 3-13 所示。当模式串的右端与主串的右端对齐的时候还没找到相匹配的串,可确定查找失败。坏字符策略在匹配过程中对主串一些字符跳过不去检查,如图 3-11 主串中白底色的那些字符;但也有一些字符会被检查多次,如主串中最后两个 o 均被检查了两次。

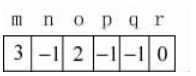

图 3-12　模式串为“room”时的部分 BC[]

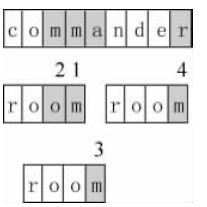

图 3-13　BC[]确定的位置在当前比较字符的右边(2)示例

(2) 好后缀策略。和坏字符策略一样,好后缀策略也是从模式串最右字符起开始与主串比较的。当主串和模式串的当前字符不相等时,模式串右移若干字符使得模式串左边和主串前面比较过的子串相同的子串(好后缀)与主串对齐,向量 GS 指示右移量。图 3-14 是应用好后缀策略的一个实例。

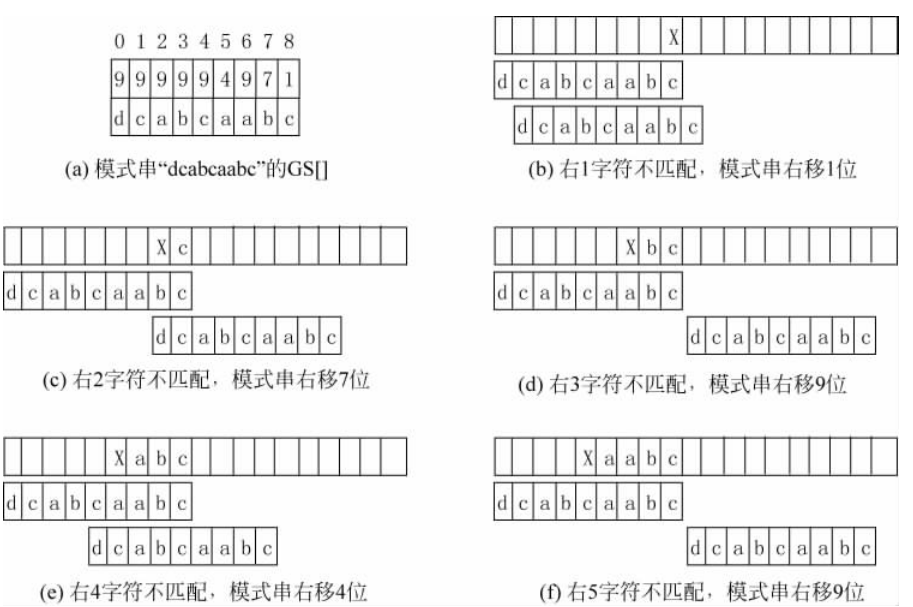

图 3-14　好后缀策略示例

以图 3-14(c)为例,模式串右数第二个字符与主串不匹配,模式串右移 7 位(等于 GS[7]),主串 X 右边的 c 与模式串左数第一个 c(好后缀)对齐继续比较。主串 X 右边的 c 不与模式串中间的 c 对齐的原因是 X 肯定不是 b,而模式串中间的 c 后面跟的恰是 b,所以模式串中间的 c 不是好后缀。

相比于坏字符策略仅考虑主串中失配的单字符(如图 3-11 第 9 次失配模式串仅右移一位),好后缀策略考虑的是整个后缀的匹配,看来好后缀策略比坏字符策略要好。但好后缀策略也有自己的软肋:当最右字符失配(没有后缀)时,只能右移一个字符,这时它不如坏字

符策略好。

Algo3-3.cpp 对于同一组模式串和主串，分别采用坏字符、好后缀、坏字符＋好后缀算法，结果是坏字符＋好后缀算法移动的次数最少。

```
//Algo3 - 3.cpp 用类实现 BM 算法的程序
#include "C.h"                              //文件包含宏命令
#include "HString.h"                        //HString 类
const int N = 256;                          //字符集扩展到汉字,ASCII 码 0～255(坏字符用到)
template < typename T > class BMMatching    ///带模板的 BM 算法类
{
private://4 个私有数据成员,3 个私有成员函数
    T P, Q;                                 //模式串,主串
    vector < int > BC, GS, SS;              //3 个整型辅助向量
    int p;                                  //模式串长度
    void buildBC(T P)
    {                                       //构造 BC 表,坏字符策略用到
        BC.assign(N, -1);                   //分配 BC 表,与字符表等长,初值为 - 1
        for(int j = 0; j < p; j++)          //自左向右扫描模式串 P
            BC[(unsigned char)P[j]] = j;    //unsigned char 避免 ASCII 码有负值
    }
    void buildSS(T P)
    {                                       //构造最大匹配后缀长度表,构造 GS 表用到
        SS.assign(p);                       //分配 SS 表,与模式串等长
        SS[p - 1] = p;                  //对最后一个字符而言,与之匹配的最长后缀就是整个 P 串
        //以下,从倒数第二个字符起自右向左扫描 P,依次计算出 SS[ ]其余各项
        for(int lo = p - 1, hi = p - 1, j = lo - 1; j >= 0; j -- )
            if(lo < j && SS[p - hi + j - 1]<= j - lo)  //情况一
                SS[j] = SS[p - hi + j - 1];  //直接利用此前已计算出的 SS[ ]
            else
            {                               //情况二
                hi = j;
                lo = __min(lo, hi);
                while(0 <= lo && P[lo] == P[p - hi + lo - 1])
                    lo -- ;                 //逐个对比处于(lo, hi]前端的字符
                SS[j] = hi - lo;
            }
    }
    void buildGS(T P)
    {                                       //构造 GS 表,好后缀策略用到
        int i, j;
        buildSS(P);
        GS.assign(p, p);                    //分配 GS 表,与模式串等长,初值为串长
        for(i = 0, j = p - 1; j >= 0; j -- )  //逆向逐一扫描各字符 P[j]
        {
            if(j + 1 == SS[j])              //若 P[0, j] = P[p - j - 1, p),则
                while(i < p - j - 1)        //对于 P[p - j - 1]左侧的每个字符 P[i]而言
                    GS[i++] = p - j - 1;    //p - j - 1 都是 GS[i]的一种选择
        }
        for(j = 0; j < p - 1; j++)
        //正向扫描各字符 P[j]——画家算法：GS[j]不断递减,直至最小
```

```
            GS[p - SS[j] - 1] = p - j - 1;    //p - j - 1 必是其 GS[p - SS[j] - 1]值的一种选择
        cout <<"GS = ";
        for(i = 0; i < p; i++)
            cout << GS[i]<<' ';
        cout << endl;
    }
public:
    int match(T P, T Q, int pos, bool Bad, bool Good)
    {                                    //Boyer - Morre 算法(完全版,兼顾坏字符与好后缀)
        if(Bad || Good)                  //不能全为 false
        {
            p = P.size();                //求得模式串的串长
            if(Bad)                      //坏字符策略
                buildBC(P);              //构造 BC 表
            if(Good)                     //好后缀策略
                buildGS(P);              //构造 GS 表
            int i = pos;           //模式串相对于主串的起始位置(初始时[0]与主串[pos]对齐)
            while(Q.size()>= i + p)      //模式串最末尾的字符没有移出主串
            {                            //不断右移(距离可能不止一个字符)模式串
                int j = p - 1;           //从模式串最末尾的字符开始
                while(P[j] == Q[i + j])  //自右向左比对,不等即退出 while 循环
                    if( -- j < 0)  //相等则继续比对左边字符,直至最左字符后跳出 while 循环
                        break;
                if(j < 0)                //完全匹配
                    break;               //跳出外层 while 循环,返回匹配位置
                else                     //否则,适当地移动模式串
                {
                    int B = INT_MIN, G = INT_MIN;   //初值取整型最小值
                    if(Bad)                         //坏字符
                        B = __max(1, j - BC[(unsigned char)Q[i + j]]);
                        //根据 BC 表求得位移量,最少右移 1
                    if(Good)                        //好后缀
                        G = GS[j];                  //根据 GS 表求得位移量
                    i += __max(G, B);               //位移量根据 B 和 G 选择大者
                    cout <<"i = "<< i << endl;
                }
            }
            return i;
        }
        return Q.size();                            //超出范围
    }
};
void main()
{
    typedef string T;
//  typedef HString T;
    BMMatching < T > BM;
    T P = "dcabcaabc", Q = "asdrcabdcabcaabcytu";
    cout <<"主串: "<< Q <<" 子串: "<< P << endl;
    int index, pos = 1;
```

```
    bool B, G;
    for(int flag = 0; flag < 3; flag++)
    {
        B = false, G = false;                          //设置初值
        switch(flag)
        {
        case 0: B = true; cout <<"坏字符: "<< endl; break;
        case 1: G = true; cout <<"好后缀: "<< endl; break;
        case 2: B = true; G = true; cout <<"坏字符加好后缀: "<< endl; break;
        }
        index = BM.match(P, Q, pos, B, G);
        if(Q.size()< index + P.size())
            cout <<"由["<< pos <<"]开始的主串和模式串匹配不成功。"<< endl;
        else
            cout <<"由["<< pos <<"]开始的主串在["<< index <<"]和模式串首次匹配。"<< endl;
    }
}
```

程序运行结果：

```
主串: asdrcabdcabcaabcytu 子串: dcabcaabc
坏字符:
i = 3
i = 4
i = 6
i = 7
由[1]开始的主串在[7]和模式串首次匹配。
好后缀:
GS = 9 9 9 9 9 4 9 7 1
i = 2
i = 3
i = 7
由[1]开始的主串在[7]和模式串首次匹配。
坏字符加好后缀:
GS = 9 9 9 9 9 4 9 7 1
i = 3
i = 7
由[1]开始的主串在[7]和模式串首次匹配。
```

4. Karp-Rabin 算法

十进制数 $1234=1\times10^3+2\times10^2+3\times10^1+4\times10^0$，Karp-Rabin 算法把字符串看成整数：$abcd=a\times R^3+b\times R^2+c\times R^1+d\times R^0$。其中 R 的值根据字符种类确定：如果字符限制在 26 个英文小写字母，则可设 $R=26$，$a=0$，$b=1$，$c=2$，$d=3$，……。这样，不同个数、不同组合的字符串都有各自的数值，称为“指纹”。两个指纹相同的字符串是相等的。但还有一个问题：如果字符种类多，字符串又长，则指纹的值很容易超过整型或长整型的最大值。解决这个问题的方法是找一个适当大的整数对指纹求余。但这又引出了新的问题：两个不同的指纹求余后变得相同了，这时指纹相同不保证字符串相同。但经过指纹的筛选所剩字符

串不多了,再逐一检验它们是否真与模式串匹配也不会太耗时。

另外一个问题是十进制数 $001234=0\times10^5+0\times10^4+1\times10^3+2\times10^2+3\times10^1+4\times10^0=1234$,这在数学上是没有问题的,但处理字符串就有问题了。例如,上例 abcd=a×R^3+b×R^2+c×R^1+d×R^0= 0×26^3+1×26^2+2×26^1+3×26^0=bcd。解决这个问题有两种方法:①把 a~z 的值设为 1~26,这样,abcd=a×R^3+b×R^2+c×R^1+d×R^0= 1×26^3+2×26^2+3×26^1+4×26^0,bcd= b×R^2+c×R^1+d×R^0= 2×26^2+3×26^1+4×26^0,可以区分二者;②因为对指纹求余后,有了多义性,故要对指纹相同的字符串再做一次检查,这个检查可以区分系数为 0 的前缀字符。本书采用的是第二种方法。

```
//Algo3-4.cpp Karp-Rabin 算法
#include "C.h"                                  //文件包含宏命令
#include "HString.h"                            //HString 类
const int R=255;                                //包括汉字的基数
const int M=INT_MAX/R/2;                        //求余的模(尽量大些)
template<typename T> class Karp_Rabin           //带模板的 Karp_Rabin 类
{
private://4 个私有成员函数,4 个私有数据成员
    int p, Dp;                                  //模式串的长度,字符串最高位要乘的系数
    T P, Q;                                     //模式串,主串
    void CountDp()                              //计算 Dp
    {                                           //计算基数 R 的(p-1)次方 %M
        Dp=1;                                   //Dp 累乘的初值
        for(int i=1; i<p; i++)
            Dp=(R*Dp)%M;                        //累乘 p-1 次,每次取模
    }
    int Digit(char S)
    {                                           //将字符 S 转为正的整型数值
        return int((unsigned char)S);
    }
    bool Check(int i)
    {//指纹相同时,从 Q[i]起逐位与 P 比对以确认是否真正匹配
        for(int j=0; j<p; j++, i++)
            if(P[j] != Q[i])                    //模式串与主串的指定子串的某位不同
                return false;                   //提前退出,不匹配
        return true;                            //结束 for 循环,匹配
    }
    void Update(int &hashQ, int k)              //两次求余避免做减法后是负数
    {//基于前一指纹计算起始位在 T[k],串长为 p 的 T 子串指纹的算法
        hashQ=((hashQ-Digit(Q[k-1])*Dp)%M+M)%M;
        //在前一指纹基础上,去除首位 T[k-1]
        hashQ=(hashQ*R+Digit(Q[k+p-1]))%M;      //添加末位 T[k-1+p]
    }
public://一个公有成员函数
    int Match(T P1, T Q1, int pos)
    {                                           //Karp-Rabin 串匹配算法
        P=P1, Q=Q1;                             //给三个私有数据成员赋值
        p=P.size();                             //p 是模式串的长度
        int t=Q.size();                         //t 是主串的长度
        CountDp();                              //根据 p、M、R 求得 Dp
```

```
        int hashP = 0, hashQ = 0;                    //模式串和主串指纹的初值
        for (int i = 0; i < p; i++)
        {
            hashP = (hashP * R + Digit(P[i])) % M;   //计算模式串的指纹
            hashQ = (hashQ * R + Digit(Q[i + pos])) % M;  //计算主串(前 p 位)的指纹
        }
        for(int k = pos; k <= t - p; k++)
        {                                            //查找
            if(hashQ == hashP)
            {
                cout <<"k = "<< k <<",指纹匹配,";
                if(Check(k))                         //在指纹匹配的情况下,确认字符串匹配
                {
                    cout <<"找到"<< endl;
                    return k;                        //返回
                }
            }
            Update(hashQ, k + 1);                    //计算下一个 hashQ
        }
        cout <<"没找到"<< endl;
        return k;                                    //k > t - p,表示无匹配
    }
};
void main()
{
//  typedef string T;                                //主程序第 1 行
    typedef HString T;                               //主程序第 2 行
    T p = "数据", q = "算法与数据结构";
//  T p = "cabcaabc", q = "asdrcabcaabcytu";
    cout <<"主串 q = "<< q <<",模式串 p = "<< p << endl;
    Karp_Rabin < T > K;
    int pos = 3;                                     //主串查找的起始位置
    int indes = K.Match(p, q, pos);
    cout <<" indes = "<< indes << endl;
    if(q.size()>= indes + p.size())
        cout <<"由["<< pos <<"]开始的主串在["<< indes <<"]首次与模式串匹配"<< endl;
}
```

程序运行结果：

```
主串 q = 算法与数据结构,模式串 p = 数据
k = 6,指纹匹配,找到
indes = 6
由[3]开始的主串在[6]首次与模式串匹配
```

练习 3-2

改写 BMMatching 类中的成员函数 match(),使得主串与模式串比较时,若在模式串的右数第一个字符处失配则用坏字符策略,在其余处失配用好后缀策略。自己组织一些主串和模式串,将这种算法和 Algo3-3.cpp 的三种算法做比较。

3.2 矩阵

3.2.1 多维数组的顺序存储结构

C 语言的数据存储区是一维的,但它能够表示多维数组,MuArray.h 是用一维向量 vector 实现多维数组的存储。它和 C 语言的存储方式是一样的。

```
//MuArray.h 用 vector 实现多维数组的类(MuArray 类)
#ifndef _MUARRAY_H_
#define _MUARRAY_H_
template<typename T> class MuArray
{
private://一个私有成员函数,4 个私有数据成员(见图 3-15)
    vector<T> base;                          //数组元素
    int dim;                                 //数组维数
    vector<int> bounds;                      //数组维界
    vector<int> constants;                   //数组映像函数常量基址
    bool Locate(va_list ap, int &off)const   //Value()、Assign()调用此函数
    {//若 ap 指示的各下标值合法,则求出该元素在 base 中的相对地址 off
        int i, ind;
        off=0;                               //初值
        for(i=0; i<dim; i++)
        {
            ind=va_arg(ap, int);             //逐一读取各维的下标值
            if(ind<0 || ind>=bounds[i])      //各维的下标值不合法
                return false;
            off+=constants[i]*ind;           //相对地址=各维的下标值*本维的偏移量之和
        }
        return true;
    }
public://3 个公有成员函数
    void Array(int Dim, ...)
    {//构造函数,若维数 Dim 和各维长度合法,则构造相应的数组对象(见图 3-16)
        int elemtotal=1, i;                  //elemtotal 是数组元素总数,初值为 1(累乘器)
        va_list ap;                          //变长参数表类型,在 cstdarg 中
```

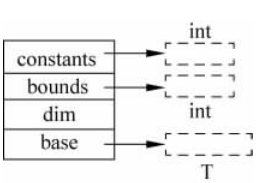

图 3-15 MuArray 类的 4 个数据成员

注:因为 vector 有指针的特性,故此处用指针表示 vector

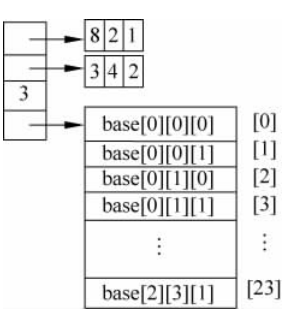

图 3-16 构造参数为(3,3,4,2)的数组对象存储实例

```
        assert(Dim > 0);                    //数组维数不合法则退出
        dim = Dim;                          //数组维数
        va_start(ap, Dim);                  //变长参数"..."从形参 Dim 之后开始
        for(i = 0; i < dim; i++)
        {
            bounds.push_back(va_arg(ap, int));
            //在尾部插入元素,逐一将变长参数赋给 bounds[i]
            assert(bounds[i]> 0);           //数组界值不合法则退出
            elemtotal * = bounds[i];        //数组元素总数 = 各维长度之乘积
        }
        va_end(ap);                         //结束提取变长参数
        base.assign(elemtotal, 0);          //给 base 分配 elemtotal 个空间,其值均为 0
        constants.assign(dim, 1);           // 给 constants 分配 dim 个空间,其值均为 1
        for(i = dim - 2; i >= 0; i -- )
            constants[i] = bounds[i + 1] * constants[i + 1];  //每一维的偏移量
    }
    bool Value(T &e, int n, ...)const
    {//...依次为各维的下标值,若各下标合法,则 e 被赋值为矩阵相应的元素值
        va_list ap;                         //变长参数表类型,在 cstdarg 中
        int off;
        va_start(ap, n);                    //变长参数"..."从形参 n 之后开始
        if(Locate(ap, off) == false)        //调用 Locate(),求得变长参数所指单元的相对地址 off
            return false;
        e = base[off];                      //将变长参数所指单元的值赋给 e
        return true;
    }
    bool Assign(T e, ...)
    {//...依次为各维的下标值,若各下标合法,则将 e 的值赋给矩阵指定的元素
        va_list ap;                         //变长参数表类型,cstdarg 中
        int off;
        va_start(ap, e);                    //变长参数"..."从形参 e 之后开始
        if(Locate(ap, off) == false)        //调用 Locate(),求得变长参数所指单元的相对地址 off
            return false;
        base[off] = e;                      //将 e 的值赋给变长参数所指单元
        return true;
    }
};
#endif
```

MuArray.h 中有些成员函数的形参有“...”,它代表变长参数表,即“...”可用若干个实参取代。这很适合含有维数不定的数组的函数。因为如果是二维数组,参数中要包括二维的长度,两个整型量;而如果是三维数组,则参数中要包括三维的长度,三个整型量。随着所构造的数组的维数不同,参数的个数也不同。这就必须使用变长参数表才能解决参数个数不定的问题。Algo3-5.cpp 是采用变长参数表的一个实例。

```
//Algo3 - 5.cpp 变长参数表(函数的实参个数可变)编程示例
#include "C.h"                  //文件包含宏命令
typedef int T;                  //定义 T 为整型
T Max(int num, ...)             //函数功能:返回 num 个数中的最大值
```

```
{//"..."表示变长参数表,位于形参表的最后,前面必须至少有一个固定参数
    va_list ap;                  //定义 ap 是变长参数表类型(C 语言的数据类型),在 cstdarg 中
    T m, n;
    assert(num > 0);             //若 num 不合法则退出
    va_start(ap, num);           //ap 指向固定参数 num 后面的实参表
    m = va_arg(ap, T);           //读取 ap 所指的实参,其类型为 T,将其赋给 m,ap 向后移
    for(int i = 1; i < num; i++) //从第二个数到最后一个数
    {
        n = va_arg(ap, T);       //依次读取 ap 所指的实参,将其赋给 n,ap 向后移
        if(m < n)
            m = n;               //m 中存放最大值
    }
    va_end(ap);          //与 va_start()配对,结束对变长参数表的读取,ap 不再指向变长参数表
    return m;                    //将最大值返回
}
void main()
{
    cout <<"1.最大值为"<< Max(4, 7, 9, 5, 8)<< endl;  //在 4 个数中求最大值,ap 最初指向 7
    cout <<"2.最大值为"<< Max(3, 17, 36, 25)<< endl;
    //在三个数中求最大值,ap 最初指向 17
}
```

程序运行结果:

```
1.最大值为 9
2.最大值为 36
```

```
//Main3 - 2.cpp 验证多维数组 MuArray 类的成员函数
#include "C.h"                                          //文件包含宏命令
#include "MuArray.h"                                    //MuArray 类
void main()
{
    int i, j, k, dim = 3, bound1 = 3, bound2 = 4, bound3 = 2; //[3][4][2]数组
    int e;
    MuArray< int > A;                                   //多维数组 MuArray 类对象
    A.Array(dim, bound1, bound2, bound3);               //构造 3×4×2 的三维数组
    cout << bound1 <<"页"<< bound2 <<"行"<< bound3 <<"列矩阵元素如下: "<< endl;
    for(i = 0; i < bound1; i++)
    {
        for(j = 0; j < bound2; j++)
        {
            for(k = 0; k < bound3; k++)
            {
                A.Assign(i * 100 + j * 10 + k, i, j, k);
                                                //将 i×100 + j×10 + k 赋值给 A[i][j][k]
                A.Value(e, dim, i, j, k);               //将 A[i][j][k]的值赋给 e
                cout <<"A["<< i <<"]["<< j <<"]["<< k <<"] = "<< setw(3)<< e <<"  ";
                //输出 e
            }
            cout << endl;
```

```
        }
        cout << endl;
    }
}
```

程序运行结果：

```
3 页 4 行 2 列矩阵元素如下:
A[0][0][0] =   0  A[0][0][1] =  1
A[0][1][0] =  10  A[0][1][1] =  11
A[0][2][0] =  20  A[0][2][1] =  21
A[0][3][0] =  30  A[0][3][1] =  31

A[1][0][0] = 100  A[1][0][1] = 101
A[1][1][0] = 110  A[1][1][1] = 111
A[1][2][0] = 120  A[1][2][1] = 121
A[1][3][0] = 130  A[1][3][1] = 131

A[2][0][0] = 200  A[2][0][1] = 201
A[2][1][0] = 210  A[2][1][1] = 211
A[2][2][0] = 220  A[2][2][1] = 221
A[2][3][0] = 230  A[2][3][1] = 231
```

3.2.2　矩阵的压缩存储

```
//RLSMatrix.h 三元组行逻辑链接顺序表的类(RLSMatrix 类)
#ifndef _RLSMATRIX_H_
#define _RLSMATRIX_H_
template < typename T > struct Triple          //三元组类型结构体(见图 3-17)
{
    int i, j;                                  //行下标,列下标
    T e;                                       //非零元素值
};
template < typename T > class RLSMatrix        //三元组行逻辑链接顺序表的类
{
private://三个私有数据成员(见图 3-18)
```

图 3-17　三元组存储结构

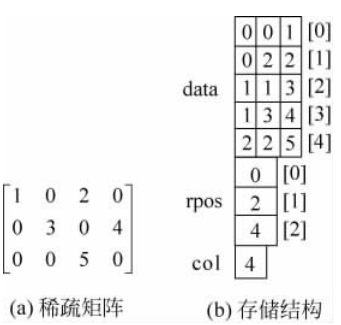

图 3-18　RLSMatrix 类存储稀疏矩阵的实例

```
    vector<Triple<T>>data;              //非零元三元组向量
    vector<int> rpos;                   //各行第一个非零元素的位置向量
    int col;                            //矩阵的列数
public://8 个公有成员函数,1 个构造函数
    RLSMatrix()
    {//构造函数,构造 0 行 0 列的空矩阵(见图 3-19)
        col = 0;
    }
    void ClearSMatrix()(见图 3-19)
    {//清空矩阵
        data.clear();
        rpos.clear();
        col = 0;
    }
    void MakeMatrixFromFile(string FileName)
    {//由文件创建稀疏矩阵
        int i, j, k;
        ifstream fin(FileName.c_str()); //打开存放稀疏矩阵数据的文件
        fin>>k>>col>>j;                 //由文件读入稀疏矩阵的行数、列数、非零元素数
        rpos.assign(k, 0);              //给 rpos[ ]赋初值 0(每行的第一个非零元素的初始位置)
        data.resize(j);                 //开辟存储空间
        for(i = 0; i<j; i++)            //依次输入 j 个非零元素
        {
            fin>>data[i].i>>data[i].j>>data[i].e;
            //由文件读入稀疏矩阵的一个非零元素
            if(data[i].i<0 || data[i].i>=rpos.size() || data[i].j<0 || data[i].j>=col)
            {
                cout<<"矩阵的行或列超出范围。"<<endl;
                exit(0);
            }
            if(i>0)                     //从第二个元素开始判断
                if(data[i].i<data[i-1].i || data[i].i==data[i-1].i &&
                data[i].j<=data[i-1].j)
                {
                    cout<<"元素没有按顺序输入。"<<endl;
                    exit(0);
                }
        }
        fin.close();                    //关闭文件
        for(i = 0; i<data.size(); i++)          //对于每个非零元素,按行统计,并记入 rpos[ ]
            for(j = data[i].i+1; j<rpos.size(); j++)  //从非零元素所在行的下一行起
                rpos[j]++;              //每行第一个非零元素的位置+1
    }
    void CopySMatrix(const RLSMatrix&M)
    {//由稀疏矩阵 M 复制得到稀疏矩阵
        col = M.col;
        data = M.data;
        rpos = M.rpos;
    }
    void TransposeSMatrix(const RLSMatrix&M)
    {//求稀疏矩阵 M 的转置矩阵
```

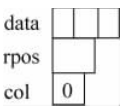

图 3-19　空矩阵

```
    int i, j, k;
    vector<int> colm;
    col = M.rpos.size();                    //M 转置的列数 = M 的行数
    if(M.data.size())                       //矩阵 M 非空
    {
        data.resize(M.data.size());         //开辟存储空间
        rpos.assign(M.col, 0);
        //给 rpos[ ]赋初值 0(每行的第一个非零元素的初始位置)
        for(i = 0; i<data.size(); i++)  //对于每个非零元素,按行统计,并记入 rpos[ ]
            for(j = M.data[i].j + 1; j<rpos.size(); j++)
            //从非零元素所在行的下一行起
                rpos[j]++;                  //每行第一个非零元素的位置 + 1
        colm = rpos;
        for(i = 0; i<data.size(); i++)  //对于 M 中的每一个非零元素
        {
            j = M.data[i].j;                //在 M 转置中的行数
            k = colm[j]++;                  //在 M 转置中的序号,colm[j] + 1
            data[k].i = j;                  //将 M.data[i]行列对调赋给 data[k]
            data[k].j = M.data[i].i;
            data[k].e = M.data[i].e;
        }
    }
}
bool AddSMatrix(const RLSMatrix &M, const RLSMatrix &N)
{//求两稀疏矩阵的和 = M + N
    int p, q, up, uq;
    if(M.rpos.size()!= N.rpos.size() || M.col!= N.col)     //M、N 两稀疏矩阵行或列数不同
        return false;
    col = M.col;                            //设置稀疏矩阵的列数
    rpos.resize(M.rpos.size());             //开辟存储空间
    for(int k = 0; k<rpos.size(); k++)  //对于每一行,k 指示行号
    {
        rpos[k] = data.size();              //矩阵第 k 行的第一个元素的位置
        p = M.rpos[k];                      //p 指示矩阵 M 第 k 行当前元素的序号
        q = N.rpos[k];                      //q 指示矩阵 N 第 k 行当前元素的序号
        if(k<rpos.size() - 1)               //不是最后一行
        {
            up = M.rpos[k + 1];             //下一行的第一个元素的位置是本行元素的上界
            uq = N.rpos[k + 1];
        }
        else                                //是最后一行
        {
            up = M.data.size();             //给最后一行设上界
            uq = N.data.size();
        }
        while(p<up && q<uq)                 //矩阵 M、N 均有第 k 行元素未处理
            if(M.data[p].j<N.data[q].j)
            //矩阵 M 当前元素的列<矩阵 N 当前元素的列
                data.push_back(M.data[p++]);
                //将 M 的当前元素值赋给矩阵,p 向后移
```

```
            else if(M.data[p].j > N.data[q].j)
            //矩阵 M 当前元素的列>矩阵 N 当前元素的列
                data.push_back(N.data[q++]);
                //将 N 的当前元素值赋给矩阵,q 向后移
            else                          //矩阵 M 当前元素的列 = 矩阵 N 当前元素的列
            {
                if(M.data[p].e + N.data[q].e!= 0)     //和不为 0
                {
                    data.push_back(M.data[p]);        //将 M 的当前元素值赋给矩阵
                    data[data.size() - 1].e += N.data[q].e;
                    //将 N 的当前元素值中成员 e 的值加入其中
                }
                p++, q++;                             //p、q 均向后移,无论和是否为 0
            }                                         //以下两个循环最多执行一个
        while(p < up)                 //N 第 k 行元素已处理完,M 还有第 k 行的元素未处理
            data.push_back(M.data[p++]);  //将 M 的当前值赋给矩阵,p 向后移
        while(q < uq)                 //M 第 k 行元素已处理完,N 还有第 k 行的元素未处理
            data.push_back(N.data[q++]);  //将 N 的当前值赋给矩阵,q 向后移
    }
    return true;
}
bool SubtSMatrix(const RLSMatrix &M, const RLSMatrix &N)
{//求两稀疏矩阵的差 = M - N
    int i;
    RLSMatrix N1;
    N1.CopySMatrix(N);                //由矩阵 N 复制矩阵 N1
    for(i = 0; i < N1.data.size(); i++)//对于 N1 的每一元素,其值乘以 - 1,N1 =- N
        N1.data[i].e *= -1;
    return AddSMatrix(M, N1);         //M + ( - N) = M - N
}
bool MultSMatrix(const RLSMatrix &M, const RLSMatrix &N)
{//一种求稀疏矩阵乘积 M×N 的方法(不使用临时数组,利用 N 的转置矩阵 S)
    int i, j, q, p, up, uq;
    Triple<T> t;                      //临时单元
    RLSMatrix S;                      //存 N 的转置矩阵
    if(M.col!= N.rpos.size())         //矩阵 M 和 N 无法相乘
        return false;
    rpos.resize(M.rpos.size());       //开辟存储空间
    col = N.col;                      //列数 = N 的列数
    S.TransposeSMatrix(N);            //S 是 N 的转置矩阵
    for(i = 0; i < rpos.size(); i++)  //对于乘积的每一行
        for(j = 0; j < col; j++)      //对于乘积的每一列,求 data[i][j].e
        {
            t.e = 0;                  //[i][j].e 的初值为 0
            p = M.rpos[i];            //p 指示矩阵 M 在 i 行的第一个非零元素的位置
            q = S.rpos[j];        //q 指示矩阵 S 在 j 行(N 在 j 列)的第一个非零元素的位置
            if(i < M.rpos.size() - 1) //不是最后一行
                up = M.rpos[i + 1];   //下一行的第一个元素的位置是本行元素的上界
            else                      //是最后一行
                up = M.data.size();   //给最后一行设上界
            if(j < S.rpos.size() - 1) //不是最后一行
```

```
                uq = S.rpos[j + 1];    //下一行的第一个元素的位置是本行元素的上界
            else                       //是最后一行
                uq = S.data.size();    //给最后一行设上界
            while(p < up && q < uq)    //p、q 分别指示矩阵 M、S 中第 i、j 行元素
                if(M.data[p].j < S.data[q].j)
                //矩阵 M 当前元素的列<矩阵 S(N)当前元素的列(行)
                    p++;               //p 向后移
                else if(M.data[p].j > S.data[q].j)
                //矩阵 M 当前元素的列>矩阵 S(N)当前元素的列(行)
                    q++;               //q 向后移
                else                   //矩阵 M 当前元素的列 = 矩阵 S(N)当前元素的列(行)
                    t.e += M.data[p++].e * S.data[q++].e;
                    //两值相乘并累加到 t.e,p、q 均向后移
            if(t.e!= 0)                //[i][j].e 不为 0
            {
                t.i = i, t.j = j;
                data.push_back(t);
            }
        }
    return true;
}
void PrintSMatrix()const
{                                      //按矩阵形式输出
    int k = 0;                         //非零元计数器,初值为 0
    vector<Triple<T>>::const_iterator p = data.begin();
    //常量指针,p 指向第一个非零元素
    for(int i = 0; i < rpos.size(); i++)        //所有行
    {
        for(int j = 0; j < col; j++)            //每行的所有元素
            if(k < data.size() && p->i == i && p->j == j)
            //p 所指非零元素为当前循环在处理元素
            {
                cout << setw(3)<<(p++)->e;  //输出 p 所指元素的值,p 指向下一个元素
                k++;                        //计数器 + 1
            }
            else                            //p 所指元素不是当前循环在处理元素
                cout << setw(3)<< 0;        //输出 0
        cout << endl;
    }
}
};
#endif

//Main3 - 3.cpp 验证稀疏矩阵 RLSMatrix 类的成员函数
#include "C.h"                              //文件包含宏命令
#include "RLSMatrix.h"                      //RLSMatrix 类
void main()
{
    bool f;
    RLSMatrix<int> A, B, C, D;
    A.MakeMatrixFromFile("F3 - 1.txt");     //由文件 F3 - 1.txt 创建矩阵 A(见图 3-20)
```

```
    B.MakeMatrixFromFile("F3 - 2.txt");          //由文件 F3 - 2.txt 创建矩阵 B(见图 3-21)
    C.MakeMatrixFromFile("F3 - 3.txt");          //由文件 F3 - 3.txt 创建矩阵 C(见图 3-22)
```

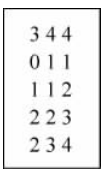

图 3-20　F3-1.txt 内容

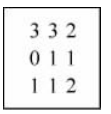

图 3-21　F3-2.txt 内容

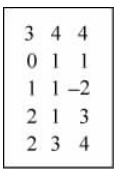

图 3-22　F3-3.txt 内容

```
    cout <<"矩阵 A = "<< endl;
    A.PrintSMatrix();                            //输出矩阵 A
    cout <<"矩阵 B = "<< endl;
    B.PrintSMatrix();                            //输出矩阵 B
    cout <<"矩阵 C = "<< endl;
    C.PrintSMatrix();                            //输出矩阵 C
    D.CopySMatrix(A);                            //由矩阵 A 复制矩阵 D
    cout <<"由矩阵 A 复制的矩阵 D = "<< endl;
    D.PrintSMatrix();                            //输出矩阵 D
    D.ClearSMatrix();                            //清空矩阵 D
    f = D.AddSMatrix(A, C);                      //D = A + C
    if(f)                                        //D = A + C 成功完成
    {
        cout <<"矩阵 D = A + C = "<< endl;
        D.PrintSMatrix();                        //输出矩阵 D
    }
    B.ClearSMatrix();                            //清空矩阵 B
    f = B.SubtSMatrix(D, A);                     //矩阵相减,B = D - A
    if(f)                                        //B = D - A 成功完成
    {
        cout <<"矩阵 B = D - A = "<< endl;
        B.PrintSMatrix();                        //输出矩阵 B
    }
    D.ClearSMatrix();                            //清空矩阵 D
    D.TransposeSMatrix(C);
    cout <<"矩阵 D = C 的转置 = "<< endl;
    D.PrintSMatrix();                            //输出矩阵 D
    A.ClearSMatrix();                            //清空矩阵 A
    f = A.MultSMatrix(C, D);
    if(f)                                        //A = C × D 成功完成
    {
        cout <<"矩阵 A = C × D = "<< endl;
        A.PrintSMatrix();                        //输出矩阵 A
    }
}
```

程序运行结果：

```
矩阵 A =
  0  1  0  0
  0  2  0  0
  0  0  3  4
```

```
矩阵 B =
  0  1  0
  0  2  0
  0  0  0
矩阵 C =
  0   1  0  0
  0  -2  0  0
  0   3  0  4
由矩阵 A 复制的矩阵 D =
  0  1  0  0
  0  2  0  0
  0  0  3  4
矩阵 D = A + C =
  0  2  0  0
  0  0  0  0
  0  3  3  8
矩阵 B = D - A =
  0   1  0  0
  0  -2  0  0
  0   3  0  4
矩阵 D = C 的转置 =
  0   0  0
  1  -2  3
  0   0  0
  0   0  4
矩阵 A = C × D =
  1  -2   3
 -2   4  -6
  3  -6  25
```