第4章　串

4.1　串及其运算

4.2　串的存储结构

4.3　串运算的实现

介绍字符串的基本概念、基本运算、存储结构（顺序存储、链式存储）；串的基本运算实现：求串的长度、插入子串、删除子串、子串定位、子串删除等；串的模式匹配算法：**BF** 算法、**KMP** 算法。

串即字符串，是计算机非数值处理的主要对象之一。

应用：文字编辑中的查找替换；

编译器的词法扫描 .

4.1 串及其运算

串：由多个或零个字符组成的有限序列 ，记作：

$$S = \text{“}c_1c_2c_3\ldots c_n\text{”}\ (n>=0)$$

其中， S 是串名 ， $c_1c_2c_3\ldots c_n$ 是串值 ， c_i 是串中字符 ， n 是串的长度，表示串中字符的数目。串值必须用一对双引号括起来。

例如， **aString = “aString”;** 中，左边的 **aString** 是一个串变量名，而右边的字符序列 **"aString"** 是赋给它的值。

空串：含零个字符的串称为空串。空串是任意串的子串。

子串：串中任意个连续的字符组成的子序列；任意串是其自身的子串。

主串：包含子串的串。

4.1 串及其运算

字符在串中的位置：

和线性表类似，通常称字符在字符序列中的序号为该字符在串中的位序；

子串在串中的位置：

子串的第一个字符在主串中的位置；如子串 "man" 在主串 "commander" 中的位置为 4 。

空格串：由一个或多个空格组成的串；

串的表示：字符串常量约定用一对双引号括起来；

串的操作：以“串的整体”作为操作对象。

回顾 C 语言

字符串的实现两种方式：

1. 用字符数组实现

static char string[ ]="I love China!";// 静态存储

char string[ ]="I love China!";

const char string[ ]="I love China!";// 只读不能写

char a[5]= "abcd";

char a[5]="abcde";// 编译出错： " array bounds overflow"

char a[5]; a[5]= "abcd";

char string[ ]; string= "China!";

char string1[ ] ;

string1=string;

2. 用字符指针实现

(1)char * string;// 字符指针变量

string="I love China!" ;// 把存放字 2)

string

(2)char *string ="I love China!" ;

定义字符指针 string 时如果未使其指向一段空间： gets(string) 和 strcpy(string, 字符串) 是不合法的

提示：不能用赋值语句将一个字符串常量或者字符数组直接赋给一个字符数组。只有 2 个途径（1）可在定义时完成初始化；（2）可通过函数实现： strcpy(字符数组 string, 字符串

调用函数完成赋值时如果输入长度超过数组大小，则编译虽通过，执行仍然报错 " ***.exe 停止工作" 或警告报错 " buffer is too small" VS2015 后用 gets_s() 取代了

串的基本运算 (常用库函数) :

1. 字符串输出函数 **puts(str)** //str 为字符数组名或者字符指针 , 等价于 **printf(“%s”, str)**

2. 字符串输入函数 **gets(str)** // str 为字符数组名，注意 **scanf(“%s”,str)** 使用时存在的问题,

3. 串赋值 **strcpy (char * T, const char * S)**
将串 S 的值赋给串指针变量 T 。

T 和 S 分别指向一个字符数组

4. 串赋值 **strncpy(char * T, const char * S, int n)**
将串 S 中前 n 个字符赋给串指针变量 T 。

5. 串连接 **char * strcat(char * S, const char * T)**
将串 T 联接在串 S 末尾，并返回 S 指针。

6. 求串长 **int strlen(const char * S)**

7. 串比较 **int strcmp(const char * S, const char* T)**

S>T： 函数值大于 0 ； S<T： 函数值小于 0 ； S=T： 函数值等于 0

串的基本运算 (常用库函

8. 子数定位 **char * strstr(const char * S, const char * T)**

串 T 在串 S 中首次出现的位置（指针）。

9. 字符定位 **char * strchr(const char * S, char c)**

查找字符 c 在 S 中首次出现的位置。

10. 字符定位 **char * strrchr(const char * S, char c)**

查找字符 c 在 S 中最后一次出现的位置。

11. **strlwr(str) //str** 为字符数组名或者字符指针

将字符串 str 转换成小写字母;

12. **strupr(str) //str** 为字符数组名或者字符指针

将字符串 str 转换成大写字母;

注：除了 puts 和 gets 函数位于 stdio.h 之外，其余 10 个函数位于 string.h 中

串的其他常用基本运算:

【1】取子串　substr(string * Sub, string *T, int i, int len);

从 T 中第 i 个字符开始的 len 个字符构成子串 Sub 。

【2】串插入　insert(string * S, int i, String T)

将串 T 插入到串 S 的第 i 个字符之后。

【3】删除子串　delete(string *S, int i, int j)

删除串 S 中第 i 个字符开始的连续 j 个字符。

【4】串替换　int replace(string *s, int start, string * t, string * v),

即要求在主串 s 中，从位置 start 开始查找是否存在子串 t，若主串 s 中存在子串 t，则用子串 v 替换子串 t 且函数返回 1，若主串 s 中不存在子串 t，则函数返回 0.

举例：

```
#include <string.h>
#include <stdio.h>
void main( )
{ char string[80];
  strcpy( string, "Hello world from " );
  strcat( string, "strcpy " );
  strcat( string, "and " );
  strcat( string, "strcat!" );
  printf( "String = %s\n", string );
 }
```

输出结果：

String = Hello world from strcpy and strcat!

4.2 串的存储结构

串的存储方法与线性表类似，只是每个结点是单个字符。

1. 顺序存储

每个串预先分配一个固定长度的存储区域，实际串长可在所分配的固定长度区域内变动，在串值后加入”\0”表示结束。顺序存储适合对串中的字符随机访问。

```
#define maxsize 256
typedef struct
{ char ch[maxsize];
  int len;
}seqstring;
 seqstring *s;
```

2．链式存储:

每个结点存放一个字符:

```
typedef struct node
{  char ch;
   struct node*next;
}linkstring;
```

链串的每个结点存放一个字符，存储密度较低。

为了提高存储密度，每个结点也可以存放多个字符。

每个结点存放 4 个字符:

```
#define chunksize 4
typedef struct node
{  char ch[chunksize];
   struct node *next;
}linkstring;
```

2. 链式存储:

为了方便串的操作，还可以设置头尾双指针及串的长度等。

每个结点存放 4 个字符为例：

```
#define chunksize 4
typedef struct node
{  char ch[chunksize];
   struct node*next;
}
```

```
typedef struct
{  chunk *head,*tail;
   int length;
}linkstring;// 链串定义
```

```
linkstring *s;
```

4.3 串运算的实现

4.3.1 基本运算的实现（串的顺序存储）

顺序串的类型定义:

```
#define  maxsize  256
typedef  struct
{ char  ch[maxsize];
   int len;
}seqstring;
seqstring *S;
```

注意：串中字符的序号从1开始，在数组中从下标0开始存放。

顺序串操作示例：赋值、输出、统计长度

```
#include<malloc.h>
#include<stdio.h>
#include <string.h >
#define maxsize 256
typedef struct
{char ch[maxsize];
 int len;
}seqstring;
void main( )
{ seqstring *str;
   str=(seqstring *)malloc(sizeo
   strcpy(str->ch,"I love my sch
   str->len=strlen(str->ch);
   puts(str->ch);
   printf("%d\n",str->len);
}
```

或：

```
void main( )
{ seqstring str;
   strcpy(str.ch,"I love my
school!");
   str.len=0;
   while(str.ch[str.len]!='\0')
     str.len++;// 统计字符串长度
   printf("str 长度是:  %d\
n",str.len);
   puts(str.ch);
}
```

gets(str->ch);

printf("%s\n",str->ch);

例 1. 串连接：将两个串首尾相接，连成一个新串 S 。

seqstring* strconnection(seqstring* S, seqstring*S1, seqstring* S2)

```
// 用 S 指向由 S1 和 S2 联接而成的新串 , 并返回
    {    int i;
         if( S1->len+S2->len > maxsize-1)
           { printf(" 上溢 "); return NULL;
           }
         else
           { for (i=0;i< S1-> len; i++)
             S-> ch[i]=S1->ch[i]; // 将 S1 串传给 S

             for (i=0;i< S2->len; i++)
             S-> ch[S1->len+i]=S2->ch[i]; // 将 S2 串添加到 S 之

             S-> len=S1->len+S2->len; // 串长度是两串长度之和
             S-> ch[S->len]='\0'; // 在 S 末
           }
    }//返回值类型可以是 int 型 0,1
```

strcpy(S->ch, S1->ch);

假如第二个 for 循环中循环条件改为 i<= S2->len 则无需后面那句末尾赋‘ \0’操作

strcat(S->ch, S2->ch);

第二个串的’ \0’ 一起复制

完整代码

:

```
#include "stdio.h"
#include "string.h"
#include "malloc.h"
#define  maxsize  256
typedef  struct
{ char ch[maxsize];
   int len;
}seqstring;
seqstring* strconnection(seqstring*, seqstring*,seqstring*);
void main( )
{ seqstring * s;
   seqstring* s1;
   seqstring* s2;
   s1 = (seqstring *)malloc(sizeof(seqstring));
   s2 = (seqstring *)malloc(sizeof(seqstring));
   s = (seqstring *)malloc(sizeof(seqstring));
   strcpy(s1->ch,"I  love  ");      s1->len=strlen (s1->ch );
   strcpy(s2->ch, "lantian");       s2->len=strlen (s2->ch);
   s=strconnection(s, s1,s2);
   printf("%s\n",s);
}
```

→ 或者 printf(“%s\n”,s->ch)

例2. 取子串：从主串的第i个字符起，取j个字符构成子串。

seqstring* substr (seqstring*sub, seqstring * s, int i, int j)

```
{   int k;
    if (i<1 || i>s->len || j<=0 || j>s->len ||j>s->len-i+1)
       {  printf(“ 越界”  );
          return NULL;
       }
else
  { for (k=0; k<j; k++)
        sub->ch[k]=s->ch[i-1+k]; // 将子串的字符串给 T
    sub->len=j; // 将子串的长度赋给 sub
    sub->ch[sub->len]=‘\0’; // 在 sub 末尾赋 ‘ \0’
    return sub;
   }
```

本算法可以不要返回值， void

例 3.1 插入子串算法（在第 i 个字符位置插入串 t ）

```
void stringinsert(seqstring*S, int i, seqstring*t)
{  int k;
   if(i<1||i>S->len)
     {printf(" 非法插入 \n");
      return;
      }

   for (k=S->len;k>=i-1;k--)
       S->ch[k+t->len]=S->ch[k];

   for(k=0;k<=t->len-1;k++)
     S->ch[k+i-1]=t->ch[k];

   S->len=S->len+t->len;
}
```

```
#define  maxsize  256
typedef  struct
{ char ch[maxsize];
   int len;
}seqstring;
```

例 4. 删除子串算法

（删除从 i 个字符开始的 m 个字符构成的子串）

```
void strDelete(seqstring*S,int i,int m)
{   int k;
    if(i<1||i>S->len)
        {printf(" 非法删除 \n");return;}
    else
{   for (k=i+m-1;k<=S->len;k++)
        S->ch[k-m]=S->ch[k];
    if(i+m-1<=S->len)
        S->len=S->len-m;
    else
        {S->ch[i-1]='\0';
         S->len=i-1;
        }
    }
}
```

```
#define  maxsize  256
typedef  struct
{ char ch[maxsize];
  int len;
}seqstring;
```

示例：子串定位（模式匹配）

模式匹配是串处理中最重要的运算之一。

定义：　在串中寻找子串（第一个字符）在串中的位置；

在模式匹配中，被查找的子串称为模式串，主串称

为目标串。

示例：　目标串　**T : “Beijing”**

模式串　**P : “jin”**

匹配结果　**= 4**

Brute-Force 算法

朴素的模式匹配算法（穷举的模式匹配算法）：

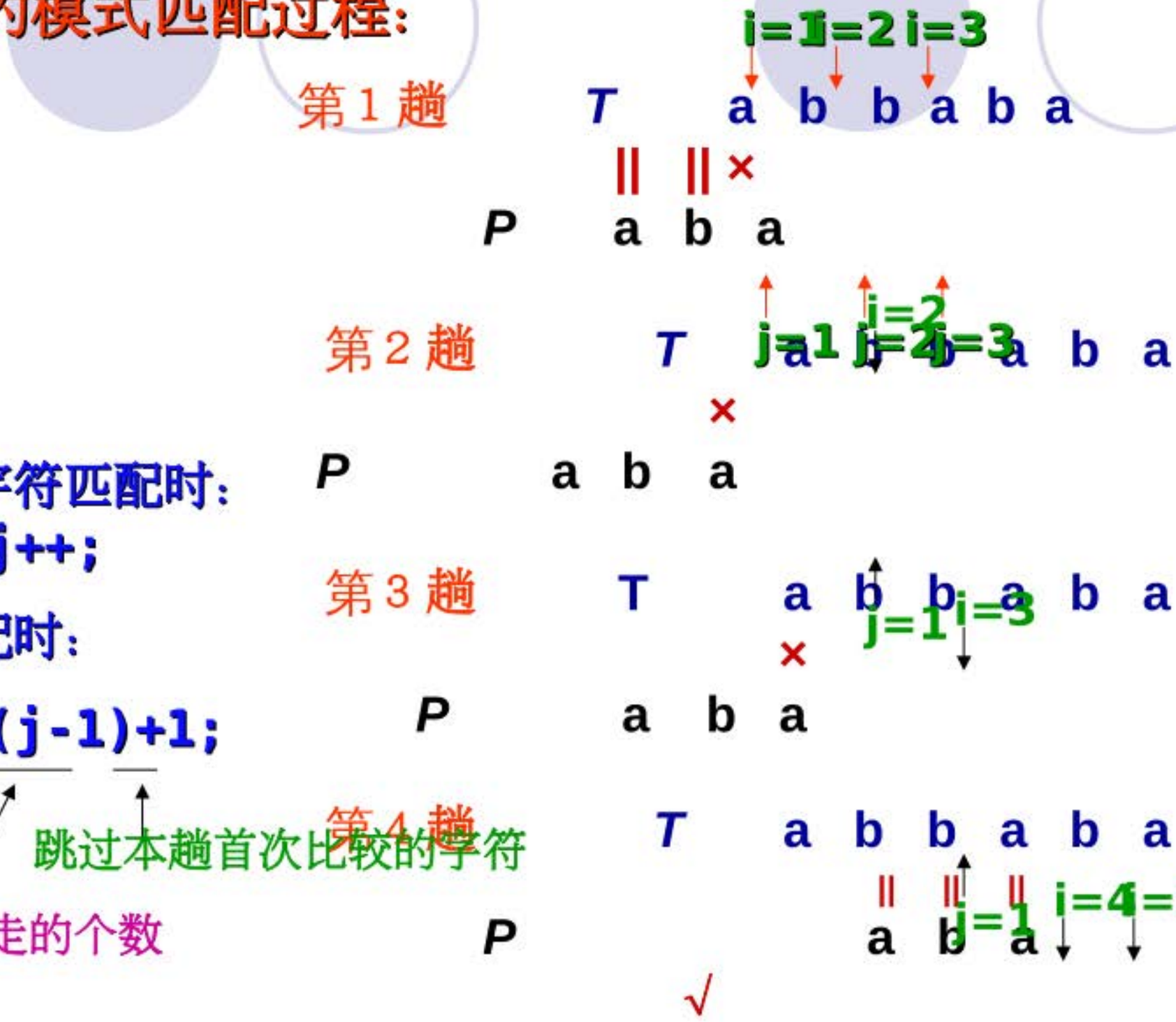
设 T="$s_1, s_2, \ldots, s_n$"（主串），P="$t_1, t_2, \ldots, t_m$"（模式串）

其中 0 < **m≤n**；

思想：从目标串 T 的第 **pos** 个字符开始与模式串 T 中的第一个字符进行比较，如果相同，则继续比较后面的字符，否则，从目标串的下一个字符开始与模式串的第一个字符进行下一轮比较。重复此过程，直至匹配成功或者直至匹配结束，失败告终。

显然，这种传统的字符串模式匹配算法是对主串和模式串双双自左向右，一个一个字符比较，如果不匹配，主串和模式串的位置指针都要回溯。

朴素的模式匹配过程：



单个字符匹配时：

```
i++;j++;
```

不匹配时：

```
i=i-(j-1)+1;
j=1;
```

跳过本趟首次比较的字符

指针游走的个数

例 5．顺序串的朴素模式匹配

```
int Index (seqstring *T, seqstring * P, int pos)
 {  int i=pos; // 位序从 pos 开始
    int j=1;

    while(i<=T->len && j<=P->len)
        { if (T->ch[i-1]= =P->ch[j-1])
            {i++;
             j++;} // 继续比较后面的字符
         else
          {i=i-j+2;
            j=1;} // 本趟不匹配，设置下一趟匹配的起始位序
        }
    if (j>P->len)
        return(i-P->len);
    else
        return(0); // 匹配不成功
}
```

目标串

模式串

每趟 i 初始值

每趟匹配失败时的 i ，j 值

朴素匹配情况：

i=1	第一趟匹配	a b a b c a b c a c b a b a b c a c	i=3,j=3
i=2	第二趟匹配	a b a b c a b c a c b a b a b c a c	i=2,j=1
i=3	第三趟匹配	a b a b c a b c a c b a b a b c a c	i=7,j=5
i=4	第四趟匹配	a b a b c a b c a c b a b a b c a c	i=4,j=1
i=5	第五趟匹配	a b a b c a b c a c b a b a b c a c	i=5,j=1
i=6	第六趟匹配	a b a b c a b c a c b a b a b c a c	匹配成功，返回值 6

链串的朴素模式匹配算法:

```
LinkString*Index (LinkString*T,  LinkString*P, LinkString*pos)
// 从 pos 所指示的位置开始查找模式串 P 在目标串 T 中首次出现的位置

{    LinkString*shift,*Tq,*Pq;
     shift=pos;
    Tq=shift; Pq=P;
    while(Tq!=NULL&& Pq!=NULL)
{  if(Tq ->data== Pq->data)
  { Tq = Tq ->next;
    Pq = Pq ->next; // 继续比较后续结点中的字符
            }
        else{ shift=shift->next; // 匹配不成功，设置下一趟匹配的起始位
置
                Tq =shift;
    Pq=P;
            }
     }
    if(Pq==NULL) return shift; // 匹配成功
    else NULL; // 匹配不成功
}
```

BF 时间复杂度分析:

最好情况下:每趟不成功匹配都发生在模式串P的首字符与T中相应字符的比较，设从T的第i个位置开始与P模式匹配成功的概率为pi，则前i-1趟共比较了i-1次，第i趟成功匹配比较次数为m，则总的比较次数为i-1+m。所以，

$$\sum_{i=1}^{n-m+1}(i-1+m)p_i = \frac{1}{n-m+1}\sum_{i=1}^{n-m+1}(i-1+m) = \frac{1}{2}(n+m)$$

此时，平均时间复杂度为O(n+m);

最坏情况下:每趟不成功匹配都发生在模式串P的最后一个字符与T中相应字符的比较，每一趟都比较了m次才能够确定对应的字符是否均相等。由于n>>m，所以时间复杂度是O(m*n)。

$$\frac{1}{n-m+1}\sum_{i=1}^{n-m+1}(i*m) = \frac{1}{2}m(n-m+2)$$

KMP 算法：

KMP 算法是一种改进的字符串匹配算法，由 **D.E.Knuth**，**J.H.Morris** 和 **V.R.Pratt** 等人提出。

KMP 算法的核心思想：每当一趟匹配过程中出现字符比较不相等时，不需回溯 i 值，而是利用已经得到的“部分匹配”的结果将模式向右“滑动”尽可能远的一段距离后，继续进行比较，尽可能减少模式串与主串的字符比较次数以达到快速匹配的目的。

具体实现通过一个 **next( )** 函数实现，函数本身包含了模式串的局部匹配信息。

KMP 算法的时间复杂度 **O(m+n)**。

本算法的要求：理解算法思想，算法实现调通。

```
int  KMP(seqstring*T, seqstring*P,int next[], int pos)
{
    int i = pos, j = 0; // 下标从 0 开始
    while (i <T->len&&j<P->len)
        if (j==-1||T->str[i] ==P->str[j])
        {
            i++; j++;
        } // 继续比较后面的字符
        else
        {
            j=next[j];
        } // 本趟不匹配，设置下一趟匹配的起始位序
    if (j >=P->len)  return i - P->len; // 匹配成功，返回匹配首字母的下标
    else
     return(-1); // 匹配不成功
}
```