



# 第四章 串

---

- 4.1 串类型的定义
- 4.2 串的实现
  - 4.2.1 定长顺序存储表示
  - 4.2.2 堆分配存储表示
  - 4.2.3 串的块链存储表示
- 4.3 串的模式匹配算法



## 4.1 串类型的定义

### 一、串及其基本概念

**串的定义：**串 (String) 是零个或多个字符组成的有限序列。

一般记作  $S = "a_1a_2a_3...a_n"$ ，其中  $S$  是串名，双引号括起来的字符序列是串值； $a_i$  ( $1 \leq i \leq n$ ) 可以是字母、数字或其它字符。

**串的长度：**串中所包含的字符个数称为该串的长度。

**空串：**长度为零的串称为空串 (Empty String)，它不包含任何字符。

**空格串：**将仅由一个或多个空格组成的串称为空格串 (Blank String)。注意：空串和空格串的不同，例如 “ ” 和 “” 分别表示长度为1的空格串和长度为0的空串。



## 4.1 串类型的定义

**子串与主串：**串中任意个连续字符组成的子序列称为该串的**子串**，包含子串的串相应地称为**主串**。通常将子串在主串中首次出现时的该子串的首字符对应的主串中的序号，定义为子串在主串中的**位置**。

例如，设A=“This is a string” B=“is” 则B是A的子串，A为主串。B在A中出现了两次，其中首次出现所对应的主串位置是3。因此，称B在A中的位置为3。

**相等：**当且仅当两个串的长度相等，并且每个对应位置的字符都相等时，称两个串相等。

例如，A=“BEIJING” B=“BEIJING”，则串A与串B相等。

**串常量和串变量：**串常量和整常数、实常数一样，在程序中只能被引用但不能改变其值，即只能读不能写。串变量的值在程序中可以改变，即可以读也可以写。



## 4.1 串类型的定义

### 二、串的抽象数据定义

串的抽象数据类型定义见教材P<sub>71</sub>

### 三、串的基本操作

对于串的基本操作，许多高级语言均提供了相应的运算或标准库函数来实现。下面仅介绍几种在C语言中常用的串运算。

定义下列几个变量：

```
char c[10];
```

```
char s1[ ]= "dirtreeformat", 定义一个仅仅足以存  
放初始化字符串以及空字符“\0”的一维数组
```

```
char *p = "file.txt", 定义一个指针，初值指向一个字  
符串常量
```



## 4.1 串类型的定义

### 1 求串长(length)

`int strlen(const char *s);` //求串的长度, 不包括结束符 ‘\0’。

例如: `printf(“%d”, strlen(s1));` 输出13

### 2 串复制(copy)

`char *strcpy(char *dest, const char *src);`

该函数将串src复制到串dest中, 并且返回一个指向串的开始处的指针。

例如: `strcpy(s3, s1);` //s3= “dirtreeformat”

### 3 联接(concatenation)

`char *strcat(char *dest, const char *src)`

该函数将串src复制到串dest的末尾, 并且返回一个指向串dest的开始处的指针。

例如: `strcat(s3, “/” )`

`strcat(s3, s2);` //s3= “dirtreeformat/file.txt”



## 4.1 串类型的定义

### 4 串比较 (compare)

```
int strcmp(const char *s1, const char *s2);
```

该函数比较串s1和串s2的大小，返回值是s1与s2第一个不同的字符差值。

例如：`result=strcmp(“baker”, “Baker”); //result>0`

`result=strcmp(“12”, “12”); //result=0`

`result=strcmp(“Jos”, “Joseph”); //result<0`

### 5 字符定位

```
char *strchr(const char *s, char c);
```

该函数是找c在字符串中第一次出现的位置，若找到则返回该位置，否则返回NULL。

例如：`p=strchr(s2, “.”); //p指向“file”之后的位置`

`if(p) strcpy(p, “.cpp”); //s2= “file.cpp”`



## 4.1 串类型的定义

- 6、串的定位 `char *strstr (const char *s1, const char *s2);`
- 在字符串s1中搜索字符串s2。如果搜索到，返回指针指向字符串s2第一次出现的位置；否则返回NULL。

例、串的定位`index(char *s, char *t, int pos)`

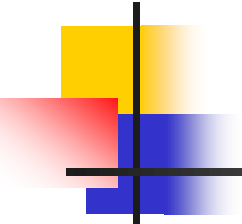
求串T在主串S中第pos个字符之后的位置。方法：从主串S的第pos个字符开始，取长度和串T相等的子串和T比较，若相等，则求得函数值为pos。否则子串位置增1继续与T比较，直至找到与T相等的子串或确定S中不存在和串T相等的子串为止。



## 4.1 串类型的定义

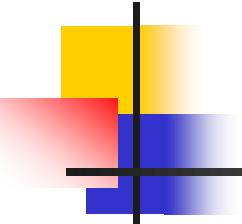
```
int index(char *s, char *t, int pos)
{  if( pos>0 )
    {  n = strlen( s );
        m = strlen( t );
        i = pos;
        while( i<n-m+1 )
        {  substr( sub, s, i, m );
            if( strcmp( sub, t ) != 0 )
                ++i;
            else return( i );
        }
    }
    return( 0 );
}
```



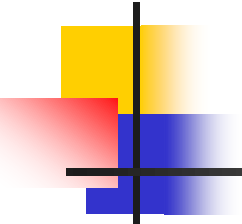


---

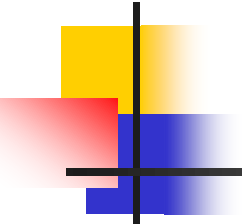
- `int strlen(const char *str)`
- `{`
- `assert(str != NULL);`
- `int len = 0;`
- `while (*str ++ != '\0')`
- `++ len;`
- `return len;`
- `}`




```
char *strcpy(char *strDes, const char *strSrc)
{
    assert((strDes != NULL) && (strSrc != NULL))
    ;
    char *address = strDes;
    while ((*strDes ++ = *strSrc ++ ) != '\0')
        NULL;
    return address;
}
```



```
char *strcat(char *strDes, const char *strSrc)
{
    assert((strDes != NULL) && (strSrc != NULL));
    char *address = strDes;
    while (*strDes != '\0')
        ++ strDes;
    while ((*strDes ++ = *strSrc ++ ) != '\0')
        NULL;
    return address;
}
```



```
int strcmp(const char *s, const char *t)
{
    assert(s != NULL && t != NULL);
    while (*s && *t && *s == *t)
    {
        ++ s;
        ++ t;
    }
    return (*s - *t);
}
```



```
char *strstr(const char *strSrc, const char *str)
{
    assert(strSrc != NULL && str != NULL);
    {
        const char *s = strSrc;
        const char *t = str;
        for (; *t != '\0'; ++ strSrc)
        {
            for (s = strSrc, t = str; *t != '\0' && *s == *t; ++s, ++
t)
                NULL;
            if (*t == '\0')
                return (char *) strSrc;
        }
        return NULL;
    }
}
```



# 复制指定长度字符串

---

- `char *strncpy(char *strDes, const char *strSrc,`
- `int count)`
- `{`
- `assert(strDes != NULL && strSrc != NULL);`
- `char *address = strDes;`
- `while (count -- && *strSrc != '\0')`
- `*strDes ++ = *strSrc ++;`
- `return address;`
- `}`



## 4.2 串的表现和实现

### 4.2.1 定长顺序存储表示

定长顺序存储表示,也称为静态存储分配的顺序表。它是用一组连续的存储单元来存放串中的字符序列。所谓定长顺序存储结构,是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 255
```

```
typedef char[maxstrlen] SString; // 0分量存放串长
```

```
Status SubString( SString &Sub, SString S, int pos, int len )  
{ if( pos < 1 || pos > S[0] || len < 0 || len > S[0]-pos+1 )  
    return ERROR;  
    Sub[1..len] = S[pos..pos+len-1];  
    Sub[0] = len;  
    retrun OK;  
}
```



## 4.2 串的表现和实现

```
Status Concat(SString &T, SString S1, Sstring S2)
{
    if( S1[0] + S2[0] <= maxstrlen ) //未截断
    {
        T[1..S1[0]] = S1[1..S1[0]];
        T[S1[0]+1..S1[0]+S2[0]] = S2[1..S2[0]];
        T[0] = S1[0] + S2[0]; return TRUE;
    }
    else if ( S1[0] < maxstrlen )// S2被部分截断
    {
        T[1..S1[0]] = S1[1..S1[0]];
        T[S1[0]+1..maxstrlen] = S2[1..maxstrlen-S1[0]];
        T[0] = maxstrlen; return FALSE;
    }
    else //S2全部被截断
    {
        T[0..maxstrlen] = S1[0..maxstrlen];
        return FALSE;
    }
}
```





## 4.2 串的表现和实现

### 4.2.2 堆分配存储表示

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而得，所以也称为动态存储分配的顺序表。在C语言中，利用和等动态存储管理函数，来根据实际需要动态分配和释放字符数组空间。

```
typedef struct{  
    char *ch;  
    int  length;  
}HString;
```



## 4.2 串的表现和实现

```
Status strassign(HString t, char *chars){  
    //生成一个其值等于串常量chars的串t  
    if(t.ch) free(t.ch);  
    for(i=0, c=chars; c; ++i, ++c) ; // 求chars的长度  
    if(!i) { t.ch=NULL; t.length=0; }  
    else  
    { if(!(t.ch=(char *)malloc(i*sizeof(char))))  
        exit(overflow);  
      t.ch[0..i-1] = chars[0..i-1];  
      t.length=i;  
    }  
}
```



## 4.2 串的表现和实现

```
int strlen(HString s){ return s.length; }
```

```
Status clearstring(HString s)
{ if(s.ch) { free(s.ch); s.ch=NULL;}
  s.length=0;
}
```

```
int strcmp(HString s, HString t)
{ for(i=0; i<s.length && i<t.length; ++i)
    if(s.ch[i] != t.ch[i])
        return(s.ch[i] - t.ch[i]);
  return s.length-t.length;
}
```



## 4.2 串的表现和实现

```
Status concat(HString t, HString s1, HString s2)
{
    if(t.ch) free(t.ch);
    if(!(t.ch=(char*)malloc((s1.length+s2.length)*sizeof(char))))
        exit(overflow);
    t.ch[0..s1.length-1]=s1.ch[0..s1.length-1];
    t.length=s1.length+s2.length;
    t.ch[s1.length..t.length-1]=s2.ch[0..s2.length-1];
    return OK;
}
```



## 4.2 串的表现和实现

```
Status substr(HString sub, HString s, int pos, int len)
{
    if( pos<1 || pos>s.length || len<0 || len>s.length-pos+1 )
        return ERROR;
    if(sub.ch) free(sub.ch);
    if(!len) { sub.ch=NULL; sub.length=0; }
    else
    {
        sub.ch=(char *)malloc(len*sizeof(char));
        sub.ch[0..len-1]=s[pos-1..pos+len-2];
        s.length=len;
    }
    return OK;
}
```



## ■ C 风格字符串

使用 字符 '\0' 终止的一维字符数组。C 风格的字符串起源于 C 语言，在 C++ 中继续支持。

## ■ C++ 中的 String 类

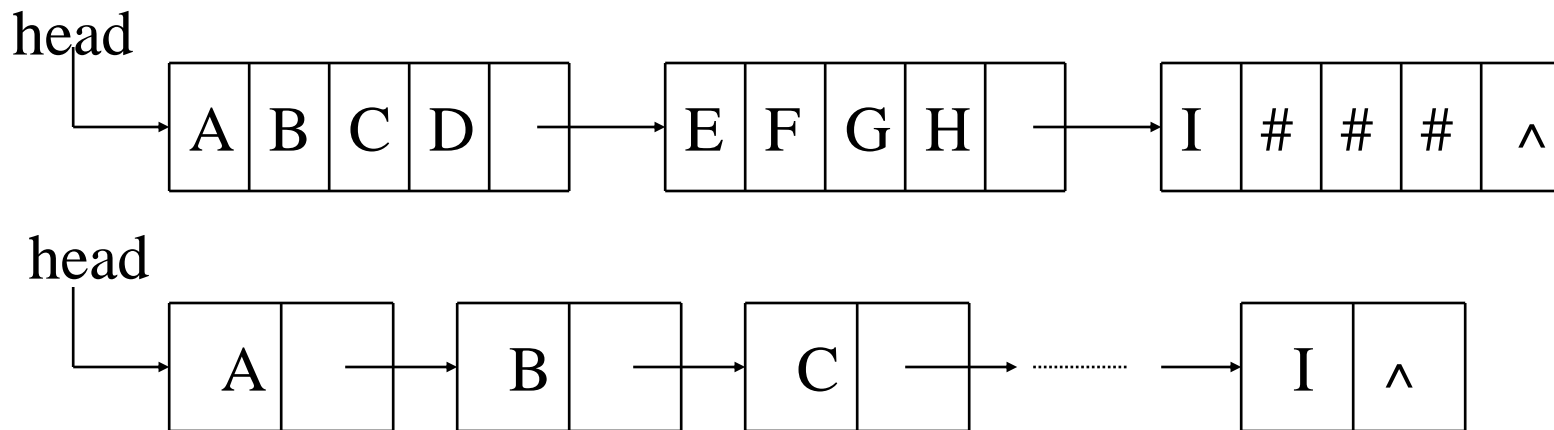
C++ 标准库提供了 **string** 类，支持长度可变的字符串。提供了许多基本操作符号，例如 `"=`", `"=="`, `"+="`，可以通过调用**string**类的成员函数实现字符串追加、赋值、比较、插入、取子串、长度等等。**string** 类能够自动将c风格的字符串转换成**string**对象，支持所有c风格的字符串的操作方法，可以与c字符串混用。

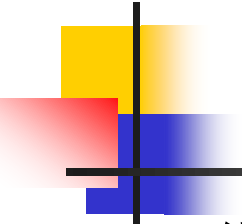
## 4.2 串的表现和实现

### 4.2.3 串的块链存储表示

顺序串上的插入和删除操作不方便，需要移动大量的字符。因此，可用单链表方式来存储串值，串的这种链式存储结构简称为链串。

一个链串由头指针唯一确定。这种结构便于进行插入和删除运算，但存储空间利用率太低。



- 
- 优点是操作方便；不足之处是存储密度较低。所谓存储密度为：

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

若要将多个字符存放在一个结点中，就可以缓解这个问题。举例：



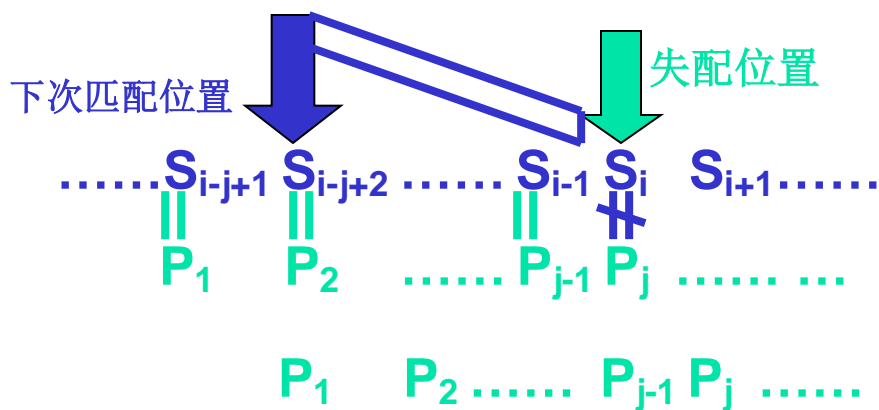


## 4.3 串的模式匹配算法

### 4.3.1 求子串位置的定位函数Index( )

**模式匹配：**求子串（模式）在主串的位置。

**基本方法：**从指定位置开始逐个比较主串与模式的字符，一旦发现出现字符不匹配，则整个模式相对于原来的位置右移一位。如下图所示：



e.g:  $S = a b c a b c a b c d$

$P = a b c a b c d$

$a b c a b c d$

模式右移一位



## 4.3 串的模式匹配算法

最基本的模式匹配程序：

```
int Index( SString S, SString T, int pos )
// 在主串S的第POS个字符之后，寻找模式T的匹配位置
{   i = pos ; j = 1;
    while ( i <= S[0] && j <= T[0] ) // 0号单元存放串长
    {   if ( S[i] == T[j] ) { ++i ; ++j ; }
        else { i = i - j + 2; j = 1; }
    }
    if ( j > T[0] ) return i-T[0] // T在S中的匹配起始位置
    else return 0;
}
```

## 4.3 串的模式匹配算法


### 4.3.2 Knuth-Morris-Pratt 模式匹配算法 (KMP 算法)

说明基本模式匹配算法在最坏情况下时间复杂度为  $O(n * m)$  的实例 ( $n$ 为主串长度,  $m$ 为模式长度)。每比较 $m$ 次, 移动模式一次。最后在主串的  $n-m+1$  找到模式串, 比较  $(n-m+1) * m$  次。



## 4.3 串的模式匹配算法

说明 **KMP** 算法的实例：

  
S = abcabcabcd  
P = abcabcd

S = abcabcabcd  
P = abcabcd

右移一位，仍失配

S = abcabcabcd  
P = abcabcd

又右移一位，仍失配

S = abcabcabcd  
P = abcabcd

三次比较省去？

再右移一位，三次比较之后，再进行断点处的比较，比较成功！

本次比较省去？

本次比较省去？

问题：能否省去上述五次比较，直接进行 **S7** 和 **P4** 之间的比较呢？

## 4.3 串的模式匹配算法

$S = \text{abcabcabcd}$   
 $P = \text{abcabcd}$

失配点

直接寻找新匹配位置

$S = \text{abcabcabcd}$   
 $P = \text{abcabcd}$

• 分析：当  $S_i$  和  $P_j$  发生失配时， $S_{i-j+1} S_{i-j+2} \dots S_{i-1} = P_1 P_2 \dots P_{j-1}$

失配点

$\dots S_{i-j+1} S_{i-j+2} \dots S_{i-1} S_i S_{i+1} \dots$   
 $\parallel \parallel \dots \parallel \parallel$   
 $P_1 P_2 \dots P_{j-1} P_j \dots$

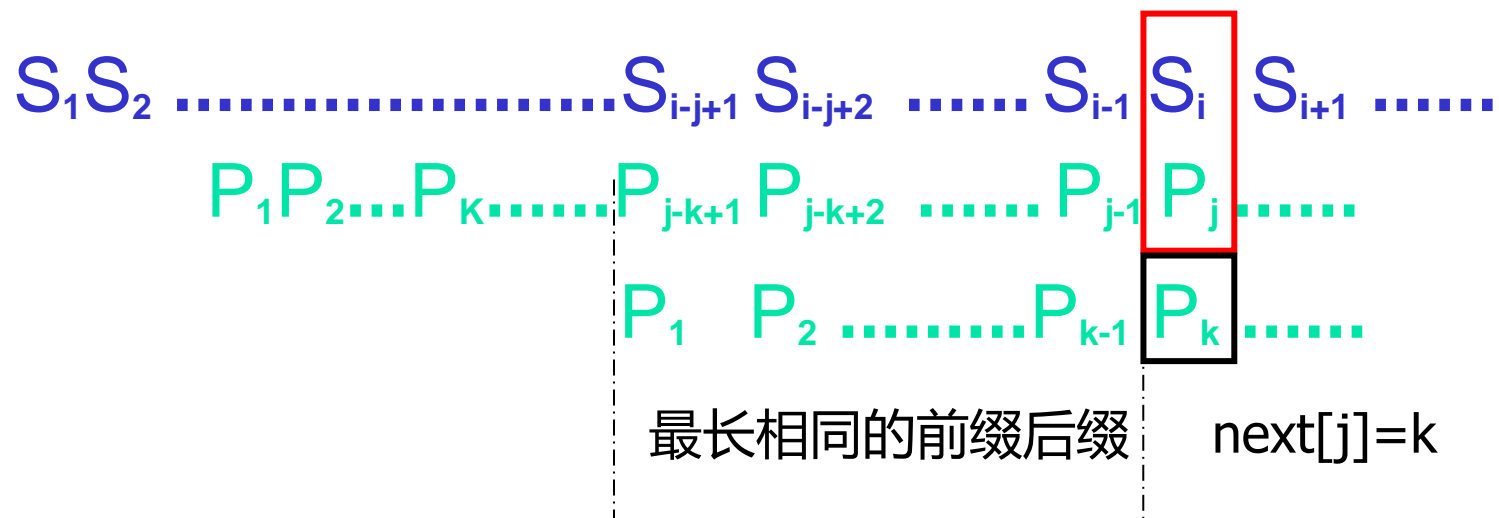
失配点

$\dots S_{i-k} S_{i-k+1} \dots S_{i-1} S_i S_{i+1} \dots$   
 $\parallel \dots \parallel \parallel$   
 $P_1 \dots P_{k-1} P_k \dots$

如果：  $P_1 P_2 \dots P_{k-1} = P_{j-k+1} P_{j-k+2} \dots P_{j-1}$ ;

可以直接比较  $S_i$  和  $P_k$







$$P_1 P_2 \dots P_K \dots P_{j-k+1} P_{j-k+2} \dots P_{j-1} P_j P_{j+1}$$

$$P_1 P_2 \dots P_{k-1} P_k$$

$$P_1 \dots P_{k'-1} P_{k'}$$

• 递推方法求特征值next函数值，设next[1]=0, 假设next[j]=k,

• (1) 若  $P_k = P_j$  , 则  $p_1 p_2 \dots p_k = p_{j-k+1} \dots p_j$ ,

•  $\text{next}[j+1] = k+1 = \text{next}[j] + 1$

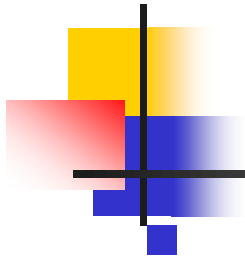
• (2) 若  $P_k \neq P_j$  , 则  $p_1 p_2 \dots p_k \neq p_{j-k+1} \dots p_j$ ,

• 若  $\text{next}[k] = k'$  , 且  $P_j = P_{k'}$  , 则  $p_1 p_2 \dots p_{k'} = p_{j-k'+1} \dots p_j$

•  $\text{next}[j+1] = k' + 1 = \text{next}[k] + 1$

• 若  $P_j \neq P_{k'}$  以此类推，直至成功或者不存在  $k'$

• 则  $\text{next}[j+1] = 1$



j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
Next[j]	0	1	1	2	2	3	1	2

- 匹配过程中产生失配，主串指针*i*不变，模式串指针*j*退回到next[j] 值的位置，若相等则指针*i*和*j*同时加1，否则指针*j*退到下一个next[next[j]] 值，以此类推。（当*j*为0时，指针*i*和*j*同时加1）



j	1	2	3	4	5	6	7	8
模式串	a	b	a	a	b	c	a	c
Next[j]	0	1	1	2	2	3	1	2

第一趟

↓ i=2

**S=acabaabaabcacaabc**

**P=ab**

↑ j=2 next[2]=1

第二趟

↓ i=2

**S=acabaabaabcacaabc**

**P=a**

↑ j=1 next[1]=0

第三趟

↓ i=3    ↓ i=8

**S=acabaabaabcacaabc**

**P=abaabc**

↑ j=6 next[6]=3

第四趟

↓ i=8    ↓ i=14

**S=acabaabaabcacaabc**

**P=abaabcac**

↑ j=2    ↑ j=9 成功

## 4.3 串的模式匹配算法

- 利用 **NEXT[j]** 函数值寻找模式的程序:

```
int Index_KMP( SString S, SString T, int pos )  
// 在主串S的第POS个字符之后，寻找模式T的匹配位置  
// 已知 NEXT 函数值，T 非空，1<=pos<=Strlength(S)  
{ i = pos ; j = 1;  
  while ( i <= S[0] && j <= T[0] )  
  { if ( j == 0 || S[i] == T[j] ) { ++i ; ++j ; }  
    else j = next[ j ];  
  }  
  if ( j > T[0] ) return i-T[0] // T在S中的匹配起始位置  
  else return 0;  
} // Index_KMP
```