4.2 串的表示和实现

在程序设计语言中，串只是作为输入或输出的常量出现，则只需存储此串的串值，即字符序列即可。但在多数非数值处理的程序中，串也以变量的形式出现。

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

4.2.1 定长顺序存储表示

```
#define  MAXSTRLEN  255
      // 用户可在255以内定义最大串长
typedef unsigned char  Sstring
                              [MAXSTRLEN + 1];
       // 0号单元存放串的长度
```

特点:

1）**串**的实际长度可在这个予定义长度的范围内随意设定，超过予定义长度的串值则被舍去，称之为“**截断**”。

2）按这种串的表示方法实现的串的运算时，其基本操作为“**字符序列的复制**”。

例如：串的联接算法中需分三种情况处理

```
Status Concat(SString S1, SString S2, SString &T) {
// 用T返回由S1和S2联接而成的新串。若未截断, 则返回TRUE，否则FALSE。
  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];  uncut = TRUE;
  }
  else if (S1[0] <MAXSTRSIZE) { // 截断
    T[1..S1[0]] = S1[1..S1[0]];
    T[S1[0]+1..MAXSTRLEN] =
      S2[1..MAXSTRLEN-S1[0]];
    T[0..MAXSTRLEN] = S1[0..MAXSTRLEN];
                  // T[0] == S1[0] == MAXSTRLEN
    uncut = FALSE;
  }
  else { // 截断(仅取S1)
    T[0..MAXSTRLEN] = S1[0..MAXSTRLEN];
    uncut = FALSE;
  }
return uncut;
} // Concat
```

算法：4.2

求子串：

```
Status SubString(SString &Sub, SString S, int pos, int len) {
    // 用Sub返回串S的第pos个字符起长度为len的子串。
    //其中，1≤pos≤StrLength(S)且0≤len≤StrLength(S)-pos+1
    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;          return OK;
}//SubString
```

算法：4.3

4.2.2 堆分配存储表示

```
typedef struct {
  char *ch;
     // 若是非空串，则按串长分配存储区，
     //  否则ch为NULL
  int  length;   // 串长度
} HString;
```

通常，C语言中提供的串类型就是以这种存储方式实现的。系统利用函数malloc( )和free( )进行串值空间的动态管理，为每一个新产生的串分配一个存储区，称串值共享的存储空间为**“堆”**。

C语言中的串以一个空字符为结束符，串长是一个隐含值。

这类串操作实现的算法为：

先为新生成的串分配一个存储空间，然后进行串值的复制。

```
Status StrInsert(HString &S, int pos, HString T) {
    // 1≤pos≤StrLength(S)+1。
     //在串S的第pos个字符之前插入T
    if (pos<1 || pos>S.length+1 ) return ERROR;
    // pos不合法
    if (T.length) {
    //T非空，则重新分配空间，插入T
     if (!(S.ch = (char *) malloc ( S.ch, ( S.length+T.length )
* sizeof ( char ))))
            exit (OVERFLOW);
```

```
for ( i=S.length-1;i>=pos-1; --1)
    //为插入T而腾出位置
    S.ch[i+T.length] = S.ch[i];
S.ch[pos-1..pos+T.length-2] = T.ch[0..T.length-1]； //插入T
    S.Length += T.length;
  }
  return OK;
} // StrInsert
```

算法：4.4

```
// = = = = ADT String的表示与实现 = = = =
// - - - - 串的堆分配存储表示 - - - -
Typedef struct{
char *ch ;   //若是非空串，则按串长分配存储区，否则ch为NULL
 int length;                    //串长度
}Hstring;
```

```
// - - - - 基本操作的函数原型说明 - - - -
Status StrAssign ( HString &T, char *chars );
//生成一个其值等于串常量chars的串T

 int StrLength ( HString S );
//返回S的元素个数,称为串的长度

 int StrCompare (HString S, HString T )
//若S>T，则返回值>0；若S=T，则返回值=0；若S<T，则返回值<0
```

Status ClearString (HString &S);

//将S清为空串，并释放S所占空间。

Status Concat (HString &T, HString S1, HString S2);

//用T返回由S1和S2联接而成的新串。

HString SubString (HString S, int pos, int len);

// 1≤pos≤StrLength(S)且0≤len≤StrLength(S)-pos+1。

//返回串S的第pos个字符起长度为len的子串。

```
// - - - - 基本操作的函数原型说明 - - - -
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 ) ;                    //释放T原有空间
    for ( i=0, c=chars; c; ++i, ++c );          //求chars的长度i
    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;
    }
    return OK;
}//StrAssign
```

```
Status ClearString ( HString &S );
    //将S清为空串。
 if ( S.ch )    { free ( S.ch );   S.ch = NULL; }
S.Length = 0;
 return OK;
}//ClearString
```

```
int StrLength ( HString S ) {
//返回S的元素个数,称为串的长度
 return S.length;
}// StrLength
int StrCompare (HString S, HString T )
    //若S>T，则返回值>0；若S=T，则返回值=0；若S<T，
    //则返回值<0
 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;
}//StrCompare
```

```
Status Concat(HString &T, HString S1, HString S2) {
  // 用T返回由S1和S2联接而成的新串
  if (T.ch)  free(T.ch);        // 释放旧空间
  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;
} // Concat
```

```
Status SubString(HString &Sub, HString S,
                                        int pos, int len) {
 // 用Sub返回串S的第pos个字符起长度为len的子串
  if (pos < 1  ||  pos > S.length
               ||  len < 0  ||  len > S.length-pos+1)
      return ERROR;
  if (Sub.ch)  free (Sub.ch);            // 释放旧空间
  if (!len)
     { Sub.ch = NULL;  Sub.length = 0; } // 空子串
  else {    Sub.ch = (char *)malloc(len*sizeof(char)); //完整子串
            Sub.ch[0..len-1] = S[pos-1..pos+len-2];
            Sub.length = len;
          }
  return OK;
} // SubString
```

4.2.3 串的块链存储表示

也可用链表来存储串值，由于串的数据元素是一个字符，它只有 8 位二进制数，因此用链表存储时，通常一个结点中可以存放一个字符，也可以存放多个字符。

$$\textbf{存储密度} = \frac{\text{数据元素所占存储位}}{\text{实际分配的存储位}}$$

```
//= = =串的块链存储表示= = =
#define  CHUNKSIZE  80  // 可由用户定义的块大小
 typedef  struct Chunk {  // 结点结构
   char  ch[CUNKSIZE];
   struct Chunk  *next;
 } Chunk;

 typedef struct {  // 串的链表结构
   Chunk *head, *tail; // 串的头和尾指针
    int   curlen;     // 串的当前长度
 } LString;
```

实际应用时，可以根据问题所需来设置结点的大小。

例如: 在编辑系统中，整个文本编辑区可以看成是一个串，每一行是一个子串，构成一个结点。即: 同一行的串用定长结构（80个字符），行和行之间用指针相联接。