向函数传递字符串

- 向函数传递字符串时
  - 既可用字符数组作函数参数
  - 也可用字符指针作函数参数

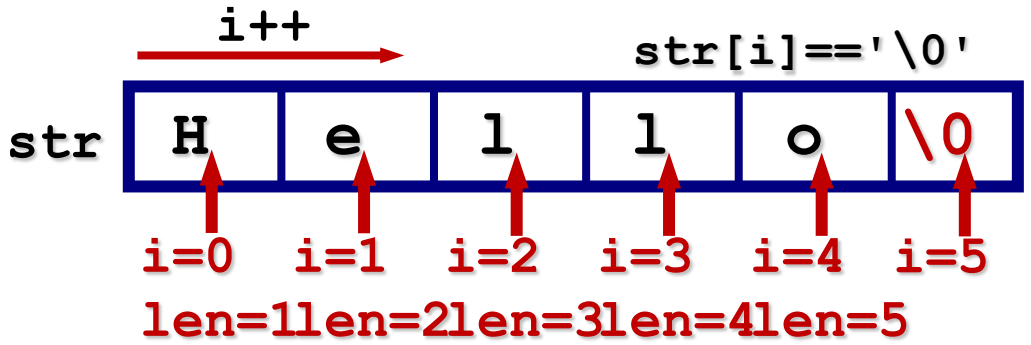
- Simulating Call by reference（模拟传引用调用）
  - 传字符串的首地址，而非字符串中的全部字符
  - `MyStrlen()`, `MyStrcpy()`

计算实际字符个数

指出MyStrlen的实现中区分char str[]和char *pStr的不必要性

```
unsigned int  MyStrlen(char str[])
{
   int  i;
   unsigned int len = 0;
   for (i=0; str[i]!='\0'; i++)
   {
        len++;
   }
   return len;
}
```

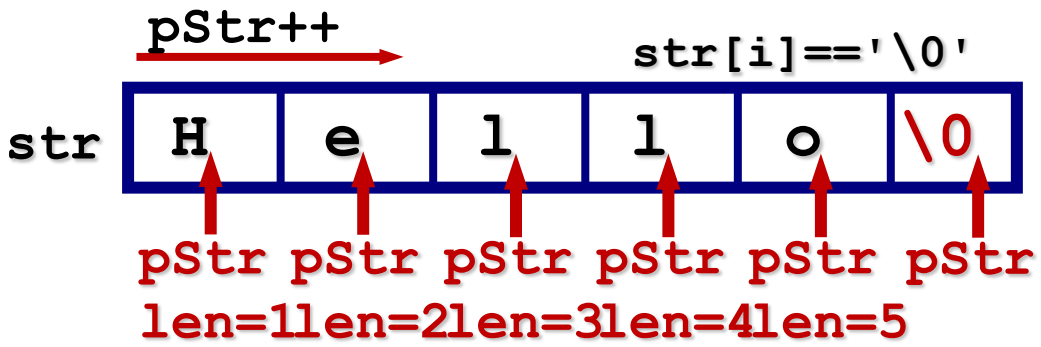
用字符数组实现



实际字符个数（不含'\0'）

```
unsigned int  MyStrlen(char *pStr)
{
   unsigned int  len = 0;
   for (; *pStr!='\0'; pStr++)
   {
        len++;
   }
   return len;
}
```

用字符指针实现



计算实际字符个数

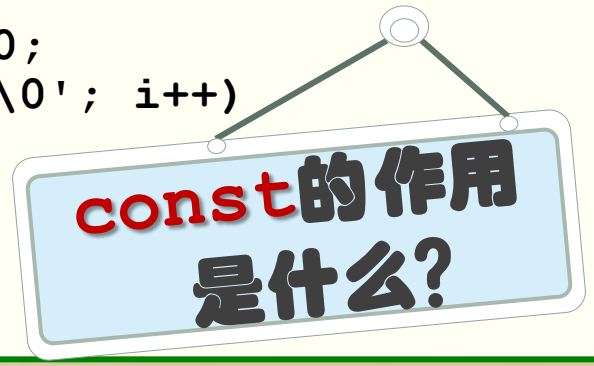
- 优化这个函数的实现

```
unsigned int  MyStrlen(char *pStr)
{
   unsigned int  len = 0;
   for (; *pStr!='\0'; pStr++)
   {
        len++;
   }
   return len;
}
```

- 从右往左读：指针变量，指向字符常量
- 保护指针变量指向的内容不被修改

```
unsigned int  MyStrlen( const  char *pStr)
{
   unsigned int  len = 0;
   for (; *pStr!='\0'; pStr++)
   {
        len++;
   }
   return len;
}
```

```
unsigned int  MyStrlen( const  char str[])
{
   int  i;
   unsigned int len = 0;
   for (i=0; str[i]!='\0'; i++)
   {
        len++;
   }
   return len;
}
```



The qualifier `const` can be applied to the declaration of any variable to specify that its value will not be changed

For an array, the `const` qualifier says that the elements will not be altered.

The `const` declaration can also be used with array arguments, to indicate that the function does not change that array

计算实际字符个数

- 进一步优化这个函数

```
unsigned int  MyStrlen(const char *pStr)
{
   unsigned int  len = 0;
   for (; *pStr!='\0'; pStr++)
   {
       len++;
   }
   return len;
}
```

*pStr!='\0'

*pStr!=0

*pStr为真

```
unsigned int  MyStrlen(const char *pStr)
{
   unsigned int  len = 0;
   for (; *pStr; pStr++)
   {
       len++;
   }
   return len;
}
```

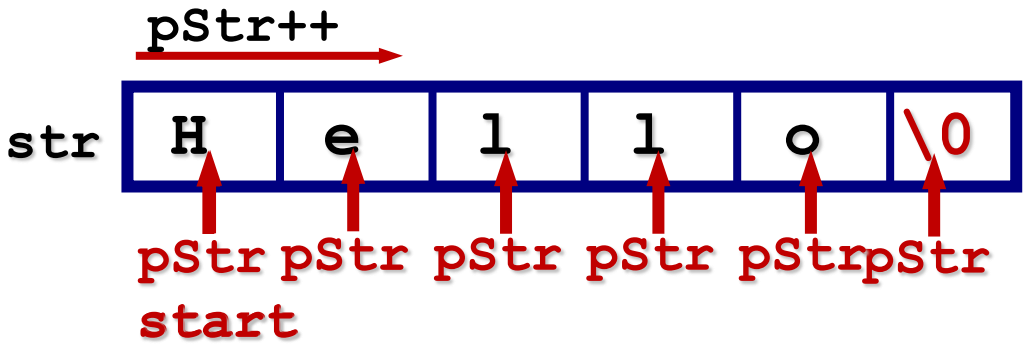
```
unsigned int  MyStrlen(const char *pStr)
{
   unsigned int  len = 0;
   while (*pStr++)
   {
       len++;
   }
   return len;
}
```

*和++都为一元运算符
结合性：right to left
pStr++相当于(pStr++)

计算实际字符个数

- 继续优化这个函数

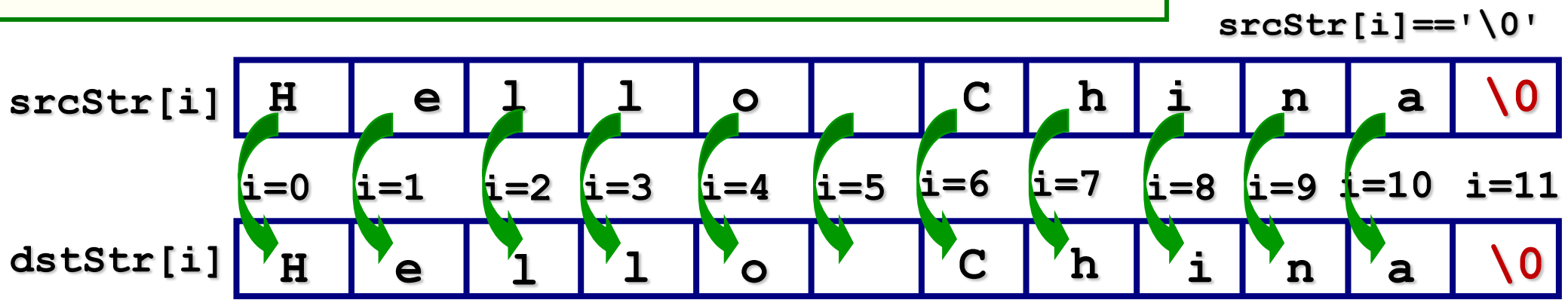
```
unsigned int  MyStrlen(const char *pStr)
{
   unsigned int  len = 0;
   while (*pStr++)
   {
        len++;
   }
   return len;
}
```




```
unsigned int  MyStrlen(const char *pStr)
{
   const char *start = pStr;
   while (*pStr)
   {
        pStr++;
   }
   return pStr - start;
}
```

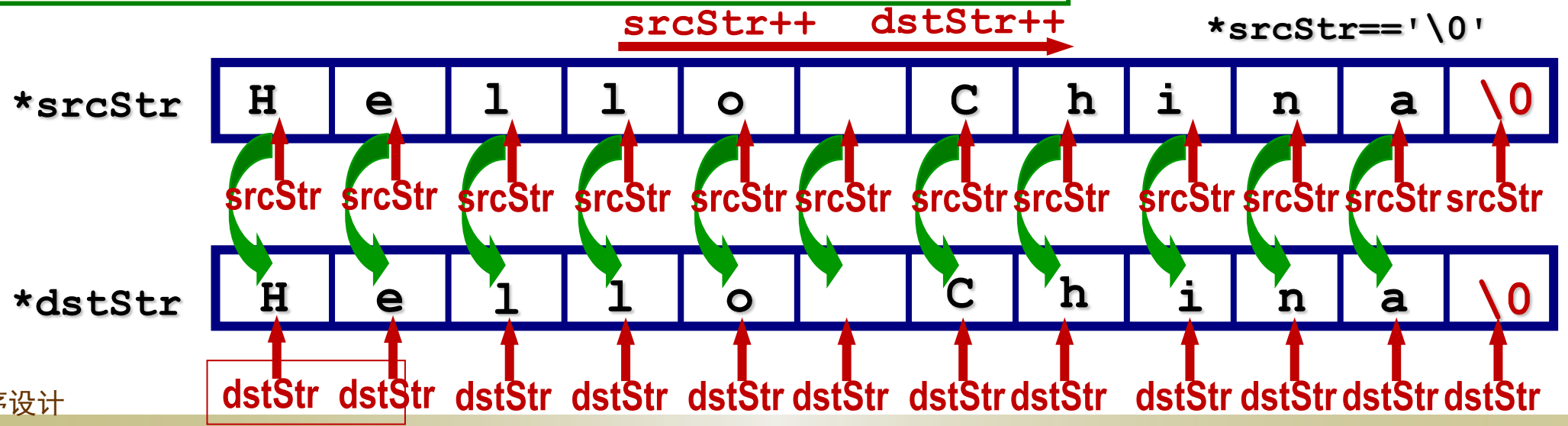
用字符数组编程实现字符串复制

```
void  MyStrcpy(char dstStr[], char srcStr[])
{
   int  i = 0;
   while (srcStr[i] != '\0')
   {
       dstStr[i] = srcStr[i];
       i++;
    }
   dstStr[i] = '\0';
}
```



用字符指针编程实现字符串复制

```
void  MyStrcpy(char *dstStr, char *srcStr)
{
   while (*srcStr != '\0')
   {
        *dstStr = *srcStr;
        srcStr++;
        dstStr++;
   }
   *dstStr = '\0';
}
```



用字符指针编程实现字符串复制

```
void  MyStrcpy(char *dstStr, char *srcStr)
{
   while (*srcStr != '\0')
   {
        *dstStr = *srcStr;
        srcStr++;
        dstStr++;
   }
   *dstStr = '\0';
}
```

```
void  MyStrcpy(char *dstStr, char *srcStr)
{
   while (*srcStr)
   {
        *dstStr++ = *srcStr++;
   }
   *dstStr = '\0';
}
```

```
void  MyStrcpy(char *dstStr, const char *srcStr)
{
   while (*dstStr++ = *srcStr++)
   {
        ;
   }
}
```

循环在赋值空字符之后才会终止

关于程序的效率的几点建议

- 不要一味地追求程序的效率，应当在满足正确性、可靠性、健壮性、可读性等质量因素的前提下，设法提高程序的效率
- 不要追求紧凑的代码，因为紧凑的代码并不一定能产生高效的机器码
- 以提高程序的全局效率为主，提高局部效率为辅
- 在优化程序的效率时，应先找出限制效率的“瓶颈”，不要在无关紧要之处优化
- 先优化数据结构和算法，再优化执行代码
- 时间效率和空间效率对立时应分析哪个更重要，作出适当的折衷