

# C语言高级篇

## 第七讲

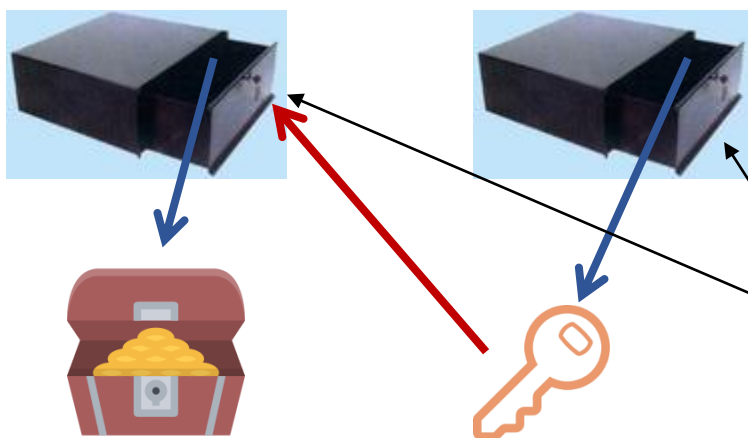
# 指针初步(1)

introduction to pointer (1)

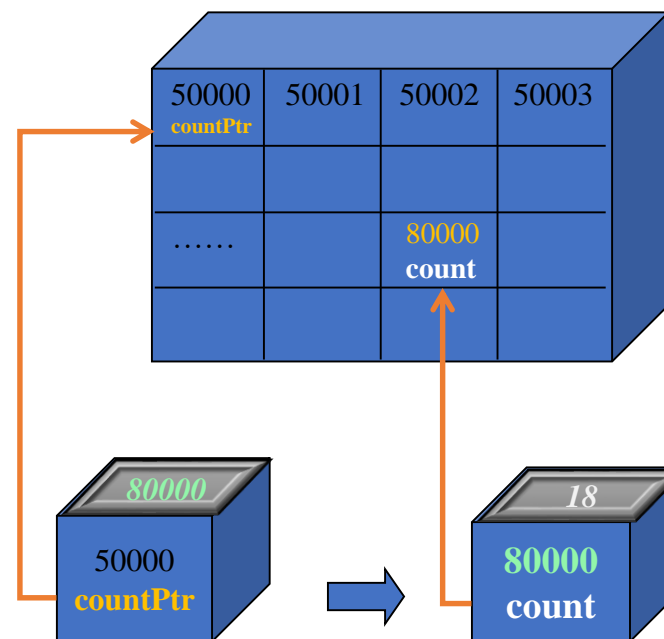


# 第七讲 指针初步

通常，变量直接包含特定值，而指针则包含变量的地址

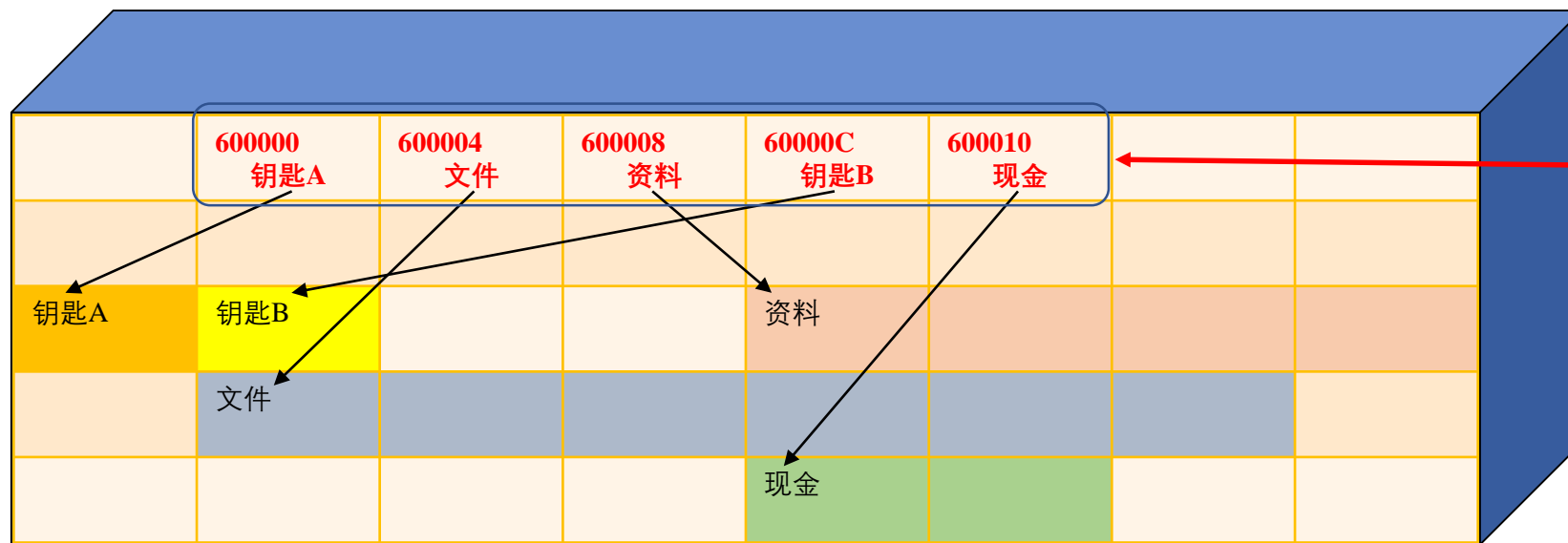
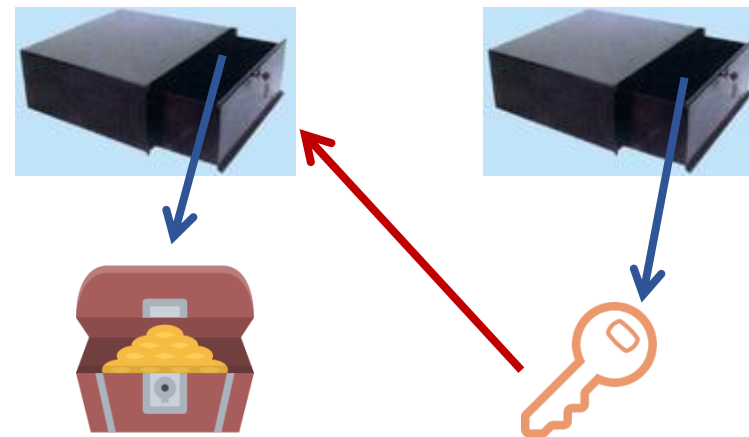


```
int count = 18;  
int *countPtr = &count;
```



count: 一般变量  
countPtr: 指针变量

# 第七讲 指针初步



这些内存里存储的不是通常的数据，而是数据的“索引”，根据索引即能访问相应的数据，如：地址600000里存放的是“钥匙A”（是某一数据的地址），然后就可以去访问“钥匙A”对应的数据。

# 第七讲 指针初步

## 指针特性

- 指针是 C 编程语言一个最强大的特性
- 指针是 C 的精髓之一
- 使程序简洁、高效、紧凑
- 指针是 C 中较难掌握的问题之一（即便是优秀的程序员，也可能对指针望而生畏！）
- 指针使程序可以模拟按引用调用，生成和操作动态数据结构
- 动态数据结构：动态的数据结构，如链表、队、栈、树、等等

# 第七讲 指针初步

---

## 学习要点

1. 了解变量在内存中的存储形式和地址的概念
2. 了解指针的概念和用法
3. 掌握指针变量的使用
4. 能用指针向函数传递参数，返回指针类型
5. 了解指针在函数中的使用
6. 熟悉指针运算
7. 了解指针、数组与字符串之间的密切关系
8. 指针在字符串处理中的应用

# 第七讲 指针初步

## 本讲结构

### 7.1 指针与地址

### 7.2 指针变量

#### 7.2.1 指针变量的定义与赋值

#### 7.2.2 通过指针访问数据

#### 7.2.3 作为函数参数的指针

#### 7.2.4 返回指针的函数

### 7.3 指针运算

#### 7.3.1 指针与整数的加减

#### 7.3.2 指针的比较

#### 7.3.3 指针强制类型转换和动态存储分配

### 7.4 指针与数组

#### 7.4.1 指针与一维数组

#### 7.4.2 字符指针与字符串

# 序：回顾变量与内存的关系

## 常用的数据实体：简单变量和数组

- 变量的属性：
  - 名称(name): 有效的标识符
  - 类型(type): int 等
  - 长度(size): 由类型决定
  - 值(value): 根据输入或赋值
  - 地址(address): 系统分配



```
int main()
{
    char c;
    short s = 0;
    int a = 55, b, sum;
    double d;
    int x[20];
    .....
    return 0;
}
```

内存 (Memory)

	.....	60FEF8 <b>d</b>	60FEF9	60FEFA	60FEFB	60FEFC	60FEFD	60FEFE	60FEFF
60FF00 <b>sum</b>	60FF01	...02	...03	...04 <b>b</b>	...05	...06	...07	...08 <b>a</b>	...09
...0A	...0B	...0C <b>s</b>	...0D	...0E	...0F <b>c</b>	...10	.....		

## 序：回顾变量与内存的关系

## 常用的数据实体：简单变量和数组

- 数组是具有相同名称和相同类型的一组连续内存空间。

```
int a[12] = {1, 3, 5, -2, -4, 6};  
for ( i=0; i<12; i++ )  
    printf("%d ", a[i]);  
printf("\n");
```



```
a[0]  a[1]  a[2]  a[3]  a[4]  a[5]  a[6]  a[7]  a[8]  a[9]  a[10] a[11]
```

1	3	5	-2	-4	6	0	0	0	0	0	0
---	---	---	----	----	---	---	---	---	---	---	---

## 内存 (Memory)

[illegible]



## 7.1 指针与地址

访问数据 { 数据实体的名称 — 直接访问 (通过变量)  
                  数据实体的地址 — 间接访问 (通过指针)

```
char a ;  
char *aPtr;  
aPtr = &a;  
a = 'c';  
*aPtr = 'x';
```

**指针：**数据实体的地址，其指向相应数据实体所在的内存空间

- 指向哪个数据实体：哪些可以访问
- 具有什么样的类型：什么操作规则

char类型变量a的地址是一个指向变量a的指针，其类型是一个指向char类型的指针。

内存 (Memory)

	.....	60FEF8 a	60FEF9	60FEFA	60FEFB	60FEFC	60FEFD	60FEFE	60FEFF
60FF00	60FF01	.....							

# 7.1 指针与地址

任意的地址都可以被称作指针吗?

NO

一个合法的具有指针类型的数据必须指向一个完整的数据实体。

计算机的内存空间以字节为单位编址。对于单位长度为多字节的数据实体，其地址是其第一个字节的地址。

```
double a, b;  
int i;  
short x, y;  
unsigned int arr[6];  
char s[8];
```

```
printf("%X\n", &a);  
printf("%X\n", &b);  
printf("%X\n", &i);  
printf("%X\n", &x);  
printf("%X\n", &y);  
printf("%X\n", arr);  
printf("%X\n", s);
```

输出

61FEF8  
61FEF0  
61FEEC  
61FEEA  
61FEE8  
61FED0  
61FEC8

地址base	变量在内存中分配情况													
0x61FEC	...						s[0]	s[1]	s[2]	s[3]	s[4]	s[5]	s[6]	s[7]
0x61FED	arr[0]			arr[1]			arr[2]			arr[3]				
0x61FEE	arr[4]			arr[5]			y		x		i			
0x61FEF	b						a							
	0x0	0x2	0x4	0x6	0x8	0xA	0xC	0xE						
	地址偏移量													

哪些地址是合法的指针?

# 指针与地址

**指针**：数据实体的地址，其指向相应数据实体所在的内存空间。

地址的获取方法：

普通变量

`&a`

与“按位与”  
的二元运算  
符'&'区分开

数组元素

`&s[6]`

函数

`max`

数组

`s`

```
double a;  
char s[8];
```

```
int max(int x, int y, int z)  
{  
    int max = x;  
    if(y > max) max = y;  
    if(z > max) max = z;  
    return max;  
}
```

- 函数的代码也是存储在内存中的，因此其代码的入口地址也是一种指针。
- 函数名本身就是该函数入口代码在内存中的地址，是一种具有指针类型的数据。

## 7.2 指针变量

- 指针变量（也简称指针）是用来存放所指对象地址的变量。
- 定义语法：<类型> \* <变量名>;
  - \* 说明名为<变量名>的变量是一个指针
  - <类型> 是指针所指向的数据类型。

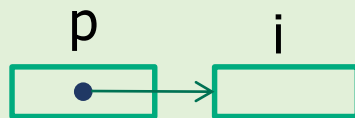
```
int *px;           // 指向整型的指针变量    也可写成int * px;
char *pc;          // 指向字符型的指针变量
char *acp[10];     // 由指向字符的指针构成的数组，即指针数组
int f( );          // 返回值为整型的函数
int *fpi( );       // 返回值为指向整型的指针的函数， 指针函数
```

## 7.2.1 指针变量的定义与赋值

### 如何使一个指针指向一个具体对象

常常通过&运算符使指针指向某个对象。单目（取地址）运算符 &：用来取变量或数组成员地址的运算符。

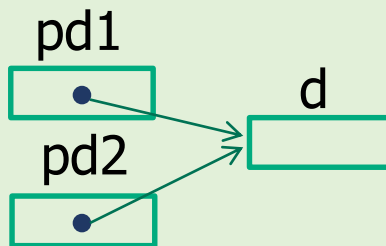
```
int i,*p;  
p = &i;
```



指针p指向了变量i

类型相同的指针变量之间可以相互赋值，不同类型的指针之间不能直接相互赋值。

```
double d, *pd1, *pd2;  
pd1 = &d;  
pd2 = pd1;
```



指针pd1和pd2都指向了变量d

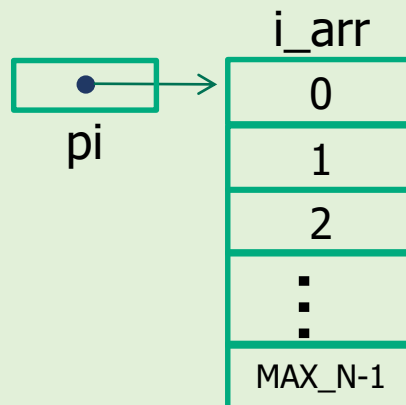
抽屉d的钥匙🔑给pd1，pd2又从pd1处复制一把钥匙🔑，pd1和pd2都能打开d。

## 7.2.1 指针变量的定义与赋值

### 指针指向一个数组

数组名的值是该数组下标为0的元素的地址，因此数组也可以直接赋值给类型相同的指针变量。

```
int i_arr[MAX_N], *pi;  
pi = i_arr;
```



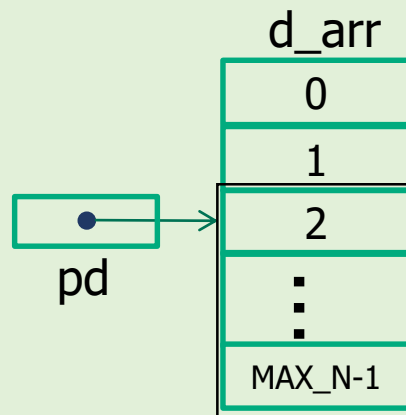
pi指向了数组i\_arr

or

pi指向了数组元素i\_arr[0]

数组元素等价于普通变量，其地址可以赋值给类型相同的指针变量。

```
double d_arr[MAX_N];  
double *pd;  
pd = &d_arr[2];
```



pd指向了数组元素d\_arr[2]

or

pd指向了数组d\_arr[]中从下标为2的元素开始的数组的后部

## 7.2.2 通过指针访问数据

- 指针表示的是数据的存储地址，而非数据本身。
- 通过指针访问数据时，必须在其左侧加上一元运算符 '\*', 复引用运算符。
- 单目（间接引用）运算符 \*：用来取某地址中内容的运算符。

```
int i = 10, y=20, *pi;
```

0x10000

i

10

pi

0x\*&^(!

pi的值未初始化，值不确定

y

20

初始化后的数据情况

## 7.2.2 通过指针访问数据

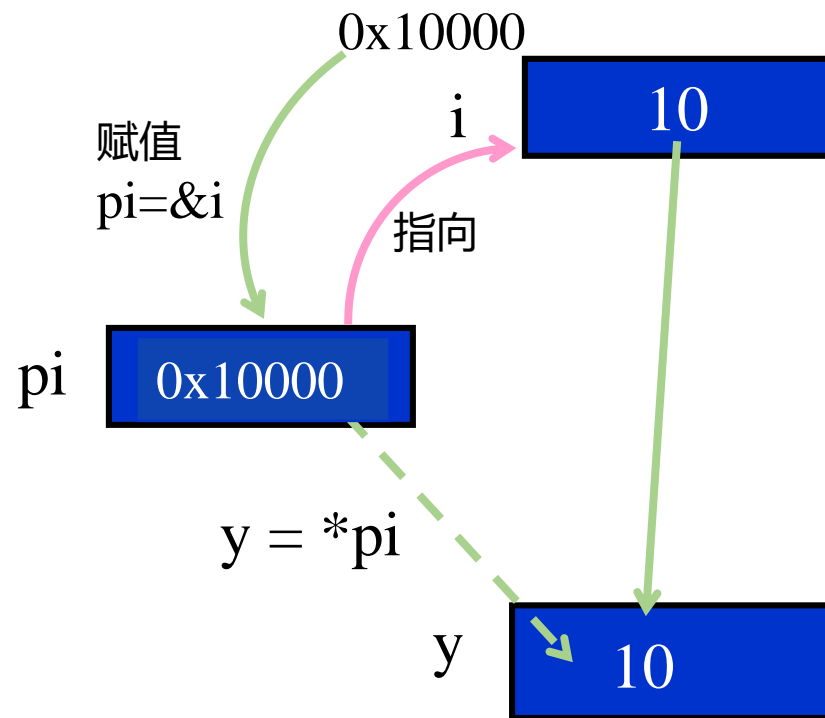
- 指针表示的是数据的存储地址，而非数据本身。
- 通过指针访问数据时，必须在其左侧加上一元运算符 '\*', 复引用运算符。
- 单目（间接引用）运算符 \*：用来取某地址中内容的运算符。

以后凡是对 i 的引用，都可用 \*pi 来代替

```
int i = 10, y=20, *pi;  
pi = &i;  
y = *pi;
```

将变量 i 的地址赋给指针变量 pi，称 pi 指向 i

取 pi 所指对象的值赋给 y，即取 pi 中所存地址中的内容



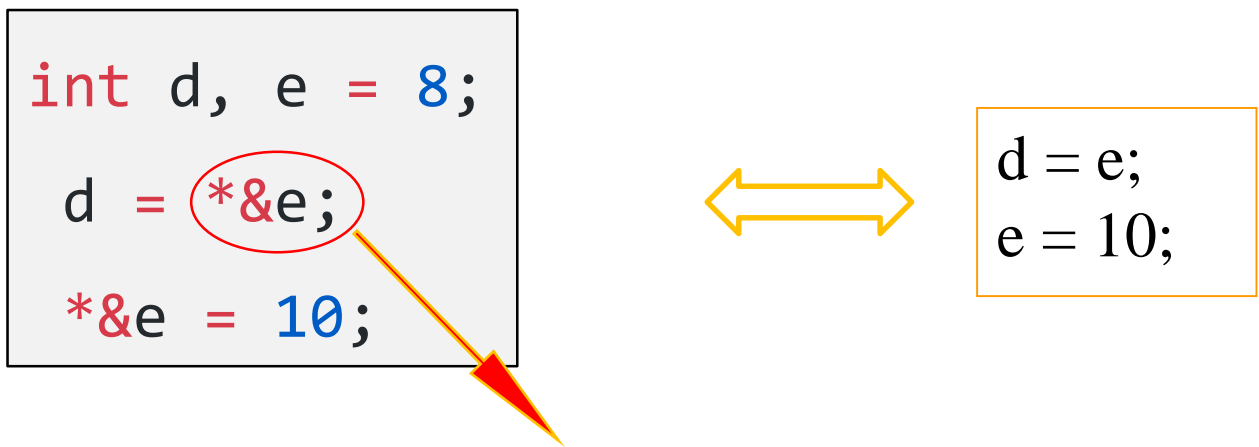
程序执行示意



## 7.2.2 通过指针访问数据

一元（间接访问）运算符'<'与取数据实体地址的运算符'&'互逆，运算符组合&\*<是合法的，但无任何意义。

```
int d, e = 8;  
d = *&e;  
*&e = 10;
```



\*&e 表示访问变量e的地址所指向的内容，等价于变量e

**例7-1-array 数据中的最长行。**从标准输入上读入多行正文数据，在标准输出上输出其中最长行的长度和内容。

输入

abcd  
abc123  
abcdefg  
xyz

输出

7: abcdefg

用数组完成：

- 数据结构设计

定义两个一维字符数组来存储新输入行及当前最长行

- 主算法设计

while(还有新输入行)

if(新行比以前保存的最长行更长)

保存新行及其长度；

输出所保存的最长行；

如何从标准输入中输入一行？如何判断输入结束？  
`while(gets(s)!=NULL)`  
...

如何比较两个字符串长度大小？需要计算字符串长度：  
`int str_len(char s[ ]);`

如何保存一个字符串？需要拷贝一个串至另一个串：  
`int str_copy(char s[], char t[]);`

## 用数组方式输出数据中最长行的长度和内容

```
#include <stdio.h>
#define MAXLINE 1024
int str_len(char s[ ]);
void str_copy(char s[ ], char t[ ]);

int main() /* find longest line */
{
    int len;          /* 当前行的长度 */
    int max;          /* 目前最长行的长度 */
    char line[MAXLINE]; /* 保存当前输入行 的内容*/
    char save[MAXLINE]; /* 保存最长行的内容 */
    max = 0;
    while( gets(line) != NULL ) {
        len = str_len(line);
        if( len > max ) {
            max = len;
            str_copy(save, line);
        }
    }
    if( max > 0 )
        printf("%d: %s", max, save);
    return 0;
}
```

```
int str_len(char s[])
{
    int i = 0;
    while(s[i] != '\0' )
        i++;
    return i;
}

void str_copy(char s[], char t[])
{
    int i = 0;
    while((s[i] = t[i])!= '\0')
        i++;
}
```

char \* gets(char s[])从标准输入中读入一行到数组 line 中，但换行符不读入，数组以 '\0' 结束。若输入结束或发生错误，则返回NULL

例7-1-pointer 数据中的最长行。输入：多行字符串。输出：最长行的长度和内容。

### 算法（用指针完成）：

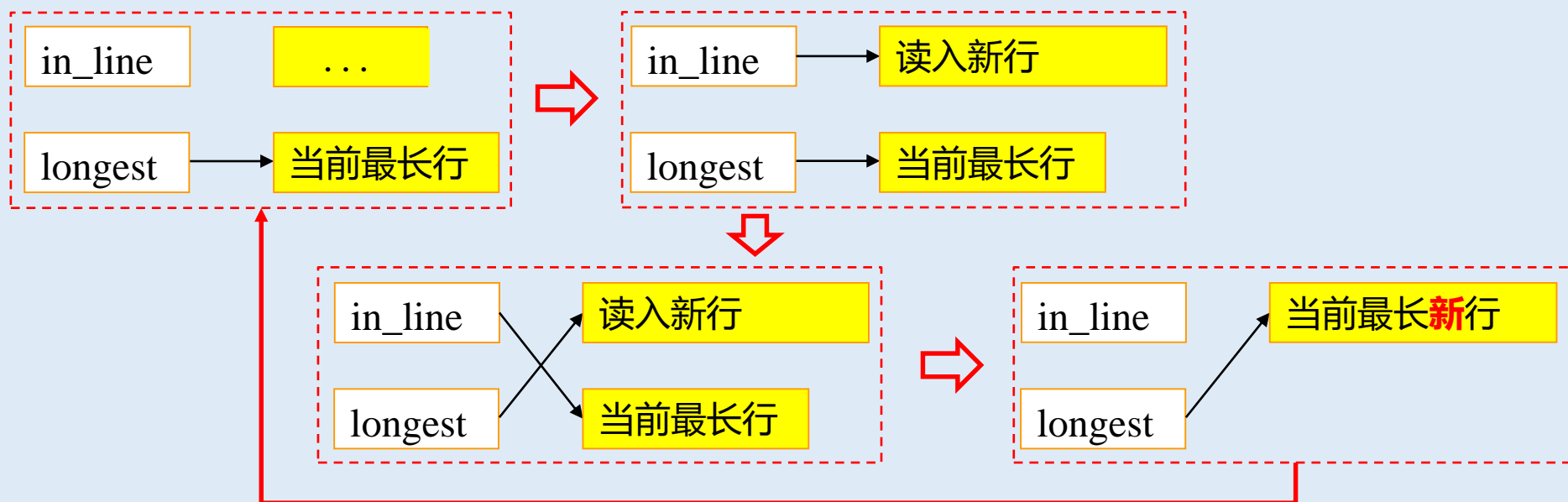
设指针变量 in\_line 和 longest 分别指向当前行（新行）和当前最长行

while(还有新输入行)

if( in\_line所指向的行比longest所指向的行长 )

交换in\_line和longest指针并保存新行长度；

输出 longest 所指内容



```

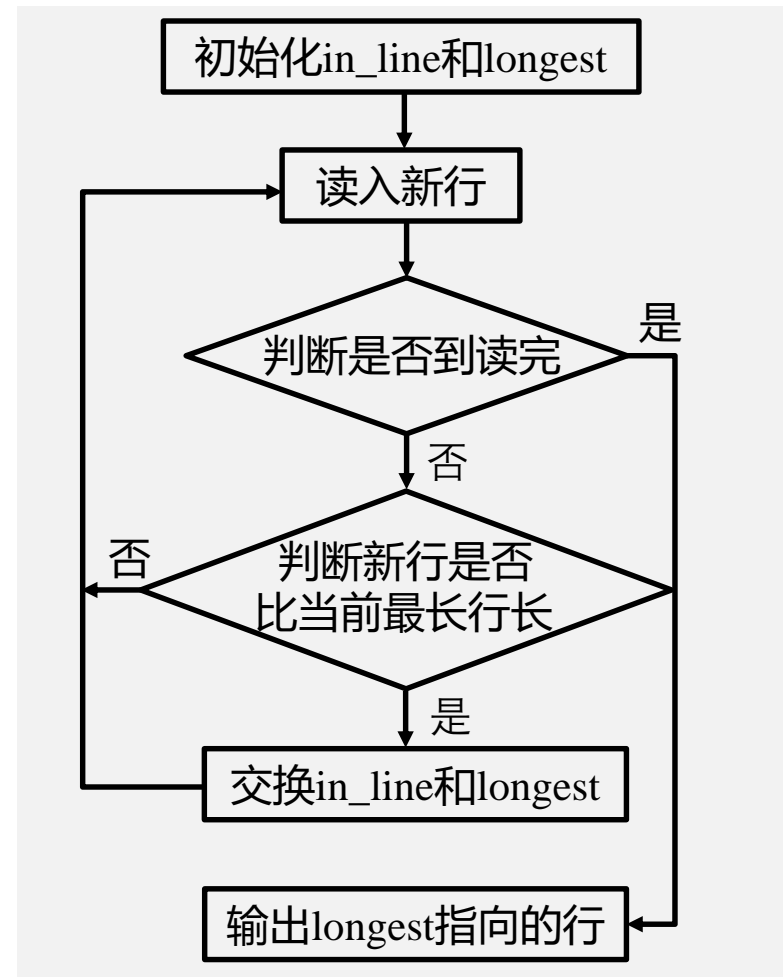
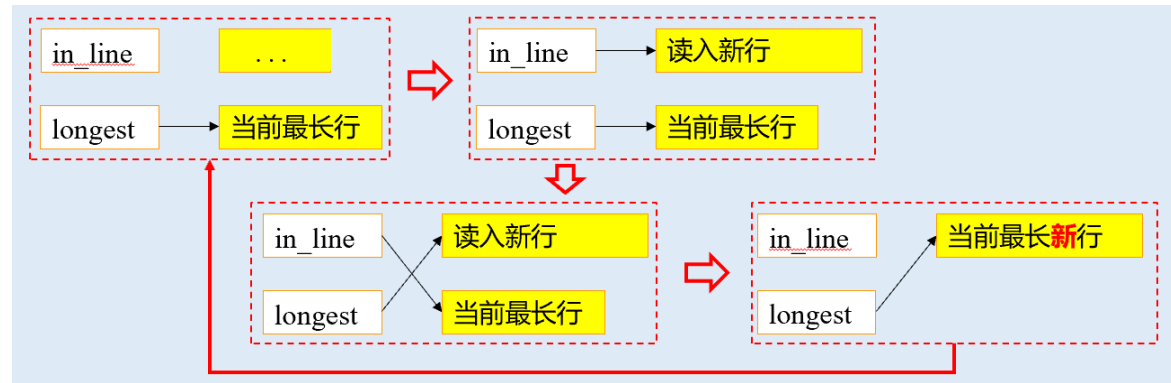
#include<stdio.h>
#include<string.h>
#define MAX 100
int main()
{
    char arr_1[MAX], arr_2[MAX] = "";
    char *in_line = arr_1, *longest = arr_2, *tmp;
    int max_len = 0, len;
    while(gets(in_line) != NULL)
    {
        len = strlen(in_line);
        if(len > max_len){
            max_len = len;
            tmp = in_line;
            in_line = longest;
            longest = tmp;
        }
    }
    printf("%d:%s\n", max_len, longest);
}

```

初始化指针使其分别指向一块空间

保存新行长度

交换指向当前行和所保存行的指针



## 两种方式的对比

```
#include<stdio.h>
#include<string.h>
#define MAX 100
int main()
{ char arr_1[MAX], arr_2[MAX] = "";
  char *in_line = arr_1, *longest = arr_2, *tmp;
  int max_len = 0, len;
  while(gets(in_line) != NULL)
  {
    len = strlen(in_line);
    if(len > max_len)
    {
      max_len = len;      tmp = in_line;
      in_line = longest; longest = tmp;
    }
  }
  printf("%d:%s\n", max_len, longest);
  ...
```

/\*指针版\*/

```
#include <stdio.h>
#define MAXLINE 1024
int str_len(char s[ ]);
void str_copy(char s[ ], char t[ ]);
int main( ) // find longest line length seen so far
{ int len, max;

  char line[MAXLINE];      // current input line
  char save[MAXLINE];      // longest line saved
  max = 0;
  while( gets(line) != NULL ){
    len = str_len(line);
    if( len > max ) {
      max = len;
      str_copy(save, line);
    }
  }
  if( max > 0) printf("%s", save);
  ...
```

/\*数组版\*/

与数组实现方式相比，指针实现方式减少了每当发现新的更长行时所进行的字符数组拷贝(通过调用函数str\_copy)。显然指针实现方式代码执行速度要快。

## 7.2.3 作为函数参数的指针

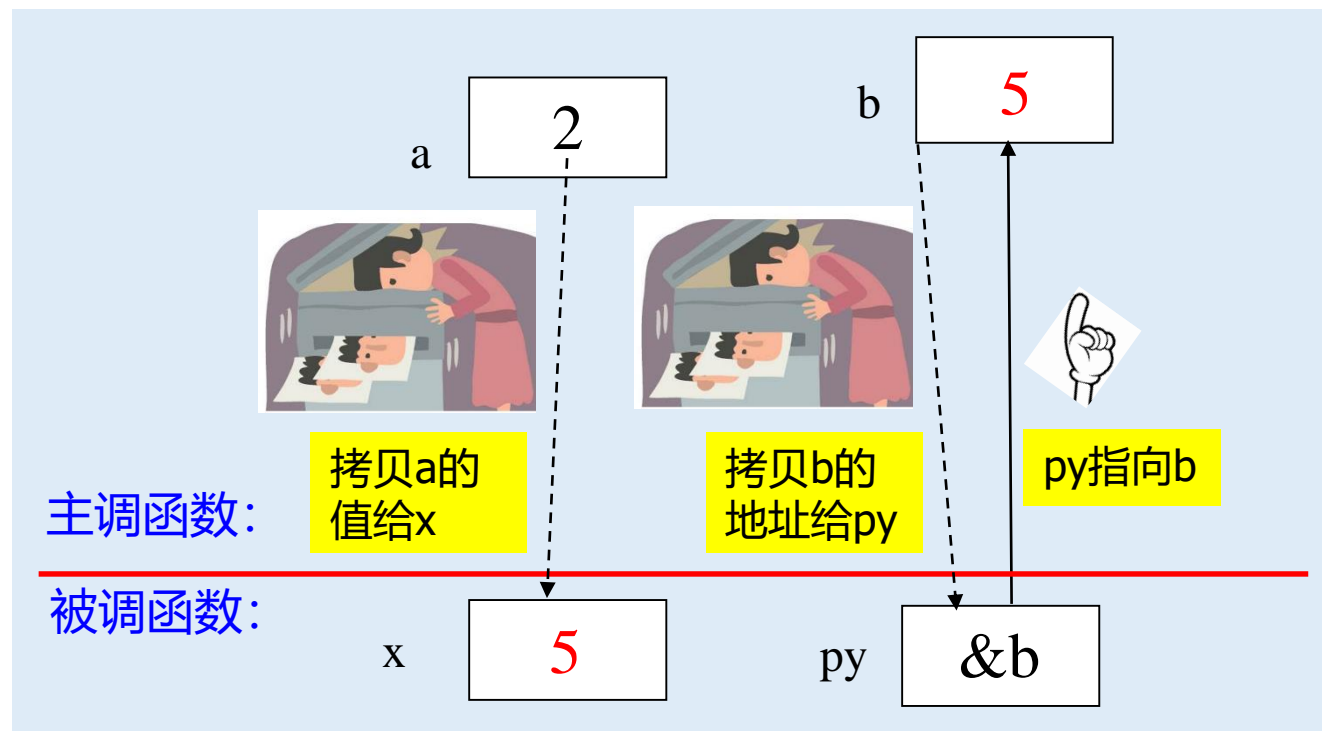
- C语言中的函数参数是**值传递**的，函数调用不会改变实参变量的值，一般通过**返回值**或者**修改全局变量**的方式实现。
- 定义指针作为函数参数，可以**以指针的方式改变函数外部的变量值**。
- 当一个函数需要同时**向外部传递多个数值**而又不希望改变全局变量时，可使用指针类型的函数参数。

```
int a = 2, b = 3, c = 4;
c = sum(a, &b);
...
```

实参传递的是变量的地址

```
int sum(int x, int *py)
{
    x += *py;
    *py = x;
    return x;
}
```

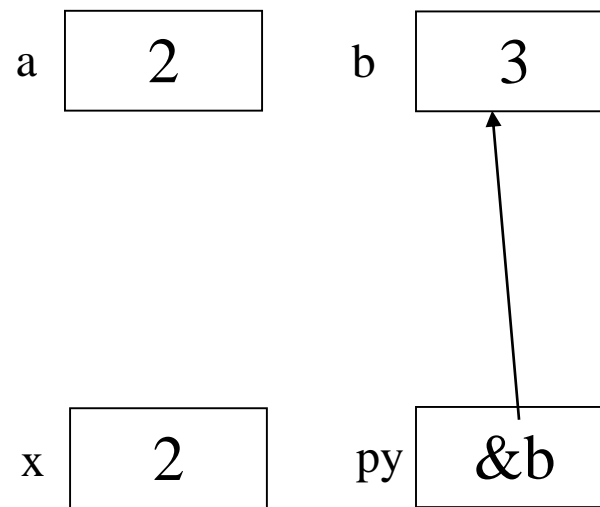
形参定义为指针



## 7.2.3 作为函数参数的指针

- C语言中的函数参数是值传递的，函数调用不会改变实参变量的值，一般通过返回值或者修改全局变量的方式实现。
- 定义指针作为函数参数，可以以指针的方式改变函数外部的变量值。
- 当一个函数需要同时向外部传递多个数值而又不希望改变全局变量时，可使用指针类型的函数参数。

```
int a = 2, b = 3, c = 4;  
c = sum(a, &b);  
...  
  
int sum(int x, int *py)  
{  
    x += *py;  
    *py = x;  
    return x;  
}
```





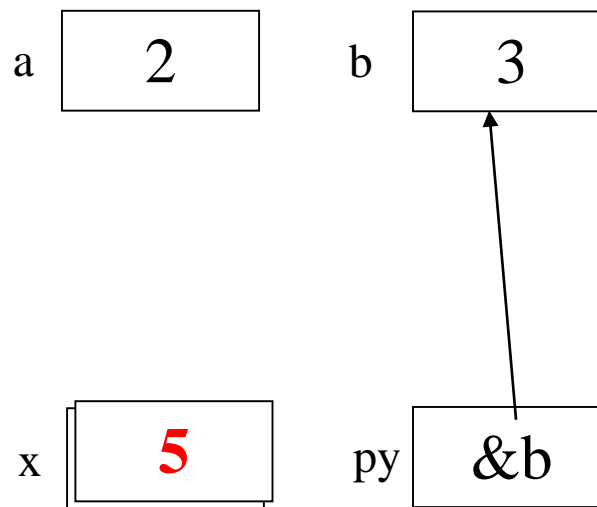
## 7.2.3 作为函数参数的指针

- C语言中的函数参数是值传递的，函数调用不会改变实参变量的值，一般通过返回值或者修改全局变量的方式实现。
- 定义指针作为函数参数，可以以指针的方式改变函数外部的变量值。
- 当一个函数需要同时向外部传递多个数值而又不希望改变全局变量时，可使用指针类型的函数参数。

```
int a = 2, b = 3, c = 4;  
c = sum(a, &b);  
...
```

```
int sum(int x, int *py)  
{  
    x += *py;  
    *py = x;  
    return x;  
}
```

当执行完这条语句，x is?



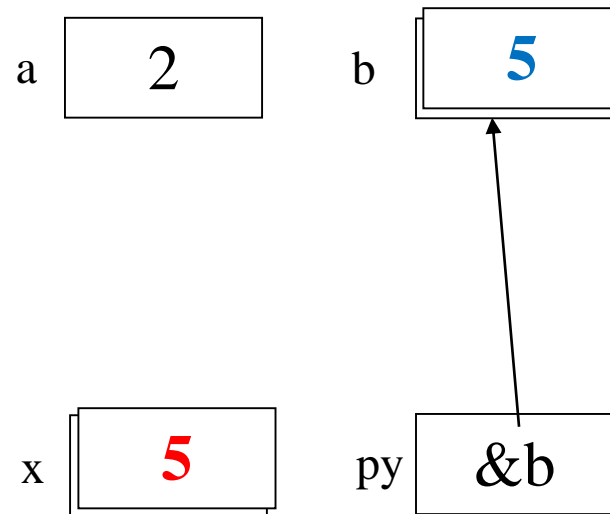
## 7.2.3 作为函数参数的指针

- C语言中的函数参数是**值传递**的，函数调用不会改变实参变量的值，一般通过**返回值**或者**修改全局变量**的方式实现。
- 定义指针作为函数参数，可以**以指针的方式改变函数外部的变量值**。
- 当一个函数需要同时**向外部传递多个数值**而又不希望改变全局变量时，可使用指针类型的函数参数。

```
int a = 2, b = 3, c = 4;  
c = sum(a, &b);  
...  
  
int sum(int x, int *py)  
{  
    x += *py;  
    *py = x;  
    return x;  
}
```

当指令执行完这句，b is?

当指令执行完这句，c is ?



## 7.2.3 作为函数参数的指针

```
int pn, nn, sum;
double average;
...
data_stat(&pn, &nn, &sum, &average);
...
```

**例7-2 数据统计函数。**设计一个函数data\_stat(), 从标准输入上读入数量不定的整数。统计输入数据中正数、负数的数量, 以及全部非 0 数据的总和及平均值。输出的平均值中保留小数点两位, 其余为整数。

```
void data_stat(int *p_num, int *n_num, int *sum, double *avg)
{
    int v;

    *p_num = *n_num = *sum = 0;
    while(~scanf("%d", &v)) {
        if(v>0) (*p_num)++;
        else if(v<0) (*n_num)++;
        *sum += v;
    }
    *avg = (double) *sum / (*p_num + *n_num);
}
```

当产生多个结果时, 通过指针类型的参数, 使作为实际参数的外部变量保存计算结果。

(\*p\_num)++和(\*n\_num)++  
中括号必不可少

输入样例:	输出样例:
1	3
2	1
-3	4
0	1.00
4	

**⚡ 注意:** 样例正确, 程序不一定对。比如, 程序中用整型, 样例也正确, 因为样例的平均值刚好为整数。但测试数据的平均值不一定是整数 (大部分不是)。

## 7.2.3 作为函数参数的指针

```
int a[10], d[10];  
int b;
```

```
char *strcpy(char dest[], char src[]);  
char *strcpy(char *dest, char *src);
```

- 数组名 (a)、数组元素地址 (&a[5]) 和 普通变量地址 (&b) 都是指针类型的数据，只要数据类型匹配，都可以作为实际参数传递给函数中指针类型的形式参数。
- 若函数的参数列表里有数组，则其是指针类型，当函数调用时，作为实际参数的数组向函数传递的是数组第一个元素的地址。
- 无论是形式参数还是实际参数，指针和数组在语法上都是等价的，可以互换。
- 一个指针类型的参数所要求的到底是一个数组还是一个普通变量的地址，需要由函数的定义来解释。当函数被调用时，实际参数必须符合函数定义的要求，这一点是由程序员而非编译系统来保证的。

```
int a = 2, b = 3, c = 4;  
c = sum(a, &b);  
...  
  
int sum(int x, int *py)  
{  
    x += *py;  
    *py = x;  
    return x;  
}
```

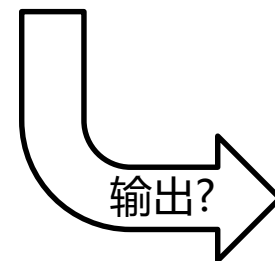
## 7.2.4 返回指针的函数——查找字符

- 指针不但可以用作函数的参数，也可以作为函数的返回值。

```
char *strchr(char *s, int c);  
char *strrchr(char *s, int c);
```

字符串s中存在字符c，strchr()和strrchr()分别返回字符c在s中第一次和最后一次出现的位置的指针。若无字符c则返回NULL。

```
char a[] = "this is test";  
printf("%x\n", strchr(a, 's'));  
printf("%c\n", *strchr(a, 's'));  
printf("%x\n", strrchr(a, 's'));  
printf("%c\n", *strrchr(a, 's'));
```



地址				...	..43	44	45	46	47	48	49	4a	4b	4c	...	
	...	t	h	i	s		i	s		t	e	s	t	\0	...	

## 7.2.4 返回指针的函数——查找字符

- 指针不但可以用作函数的参数，也可以作为函数的返回值。

```
char * strchr(char *s, int c);  
char * strrchr(char *s, int c);
```

字符串s中存在字符c，strchr()和strrchr()分别返回字符c在s中第一次和最后一次出现的**位置**的**指针**。若无字符c则返回NULL。

```
//自己实现strchr()函数  
char *my_strchr(const char *s, int c)  
{  
    if(s == NULL)    return NULL;  
    while(*s != '\0')  
    {  
        if( *s == (char) c )  
            return (char *) s;  
        s++;  
    }  
    return NULL;  
}
```

strrchr()函数  
怎么实现？  
请读者思考。

## 7.2.4 返回指针的函数——删除换行符

**例7-3 删除换行符。** 当使用函数fgets()读入一行数据时，如果输入数据包含换行符，并且缓冲区【这里指定义中存放输入的数组】足够大，则该换行符会被保存在缓冲区中。删除该缓冲区中可能包含的换行符。

```
...
#define N 50
int main()
{
    char string[N], *p;
    fgets(string, N, stdin);
    printf("%d\n", strlen(string));
    if((p = strchr(string, '\n')) != NULL)
        *p = '\0';
    printf("%d\n", strlen(string));
    return 0;
}
```

判断缓冲区是否包含换行符

输入:  
12345abcde[enter]  
输出:

11

10

?

fgets输入时把\n也作为字符输入string

1	2	3	4	5	a	b	c	d	e	\n	\0	...
---	---	---	---	---	---	---	---	---	---	----	----	-----

把字符串string中的\n修改为\0

1	2	3	4	5	a	b	c	d	e	\0	\0	...
---	---	---	---	---	---	---	---	---	---	----	----	-----

## 7.2.4 返回指针的函数—查找子串

**例7-4 字符串替换：**从标准输入读入的数据中，每行最多包含一个字符串“\_xy\_”。将输入行中的“\_xy\_”替换为“\_ab\_”，在标准输出上输出替换后的结果。

输入示例 abc\_xy\_ef.xyz

输出示例 abc\_ab\_ef.xyz

```
char *strstr(char *s, char *s1);
```

函数strstr() 的功能是检查一个字符串中是否包含某个特定的子串。

算法设计：

while (还有新输入行)

if (不含有指定字符串)

将输入字符串直接输出，继续读取数据；

else 输出“替换”后的字符串

fgets(buf, BUFSIZ, stdin) != NULL

(p = strstr(buf, str)) == NULL

当strstr()的返回值不为NULL时，p指向输入数据中的子串“\_xy\_”及其后的内容



## 7.2.4 返回指针的函数—查找子串

**[例7-4]字符串替换：**从标准输入读入的数据中，每行最多包含一个字符串“\_xy\_”。将输入行中的“\_xy\_”替换为“\_ab\_”，在标准输出上输出替换后的结果。

```
...
int main()
{
    char buf[BUFSIZ], *p, *str = "_xy_";
    while(fgets(buf, BUFSIZ, stdin) != NULL) {
        p = strstr(buf, str);
        if(p == NULL) {
            printf("%s", buf); //无匹配，输出原串
            continue;
        }
        *p = '\0'; ←
        printf("%s_ab_%s", buf, p+strlen(str));
    }
    return 0;
}
```

输入示例 abc\_xy\_e

buf	a	b	c	_	x	y	_	e	\n	\0
-----	---	---	---	---	---	---	---	---	----	----

被替换的字符不输出  
其它字符照常输出

用指针指向开始输出的位置

buf	a	b	c	\0	x	y	_	e	\n	\0
-----	---	---	---	----	---	---	---	---	----	----

↑  
p

↑  
p+strlen(str)

从子串“\_xy\_”的第一个字符将输入分为两部分

## 7.3 指针运算

- 指针和整型量可以进行加减（如上一页中的  $p + \text{strlen}(\text{str})$ ）。若  $p$  为指针，则  $p+n$  和  $p-n$  是合法的，同样  $p++$  也是合法的，它们的结果同指针所指对象类型有关。如果  $p$  是指向数组某一元素的指针，则  $p+1$  及  $p++$  为数组下一元素的指针。

$p$	$a[0]$
$p+1$	$a[1]$
$p+2$	$a[2]$

注意：  $p++$  和  $p+1$  的区别

- $p++$  是  $p = p+1$ ，结果为  $p$  指向下一元素；
- $p+1$  表示下一元素的指针，但  $p$  本身不变。

```
int a[5] = {10, 11, 12, 13, 14};
```

```
int *pi;
```

这时  $pi$  指向  $a[1]$

```
pi = &a[1];
```

这时  $pi$  指向  $a[2]$ ， $*pi$  为  $a[2]$ ，即 12

```
pi++;
```

这时  $pi$  指向  $a[4]$ ， $*pi$  为  $a[4]$ ，即 14

```
pi += 2;
```

```
*(pi-2) = *(pi-4);
```

这时  $*(pi-4)$  为  $a[0]$ ， $*(pi-2)$  为  $a[2]$ ，则  $a[2]$  变为 10

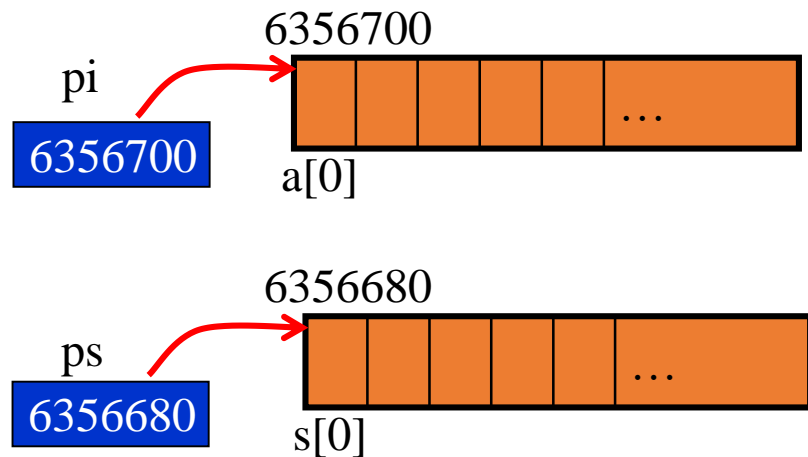
## 7.3.1 指针与整数的加减

- 与指针相加的整数表示的是元素的个数，而不是指针数值的增量。
- 指针数值的改变是以其所指向的数据类型的长度为单位的。

pi的值增加了  
 $1 * 4 = 4$

ps的值增加了  
 $3 * 2 = 6$

```
int a[10], *pi;  
pi=a;  
pi=pi+1;  
short s[10], *ps;  
ps=s;  
ps=ps+3;
```



## 例7-5 指针的加减运算

- 与指针相加的整数表示的是元素的个数，而不是指针数值的增量。
- 指针数值的改变是以其所指向的数据类型的长度为单位的。



```
#include <stdio.h>
#define MAX_N 10
int main()
{
    int *pi, a[MAX_N], i;
    short s[MAX_N], *ps;
    printf("%d %d\n", sizeof(a), sizeof(s));

    pi = a; //pi指向a[0]
    ps = s; //ps指向s[0]
    printf("pi: %d, ps: %d\n", pi, ps);

    for(i = 0; i < MAX_N; i++, pi++, ps++)
    {
        *pi = 100+i;
        *ps = i+10;
    }
    printf("pi: %d, ps: %d\n", pi, ps);
    printf("a: %d, %d, %d, %d\n", a[0], a[1], a[2], a[5]);
    printf("s: %d, %d, %d, %d\n", s[0], s[1], s[2], s[5]);
    printf("pi: %d, ps: %d\n", pi, ps);

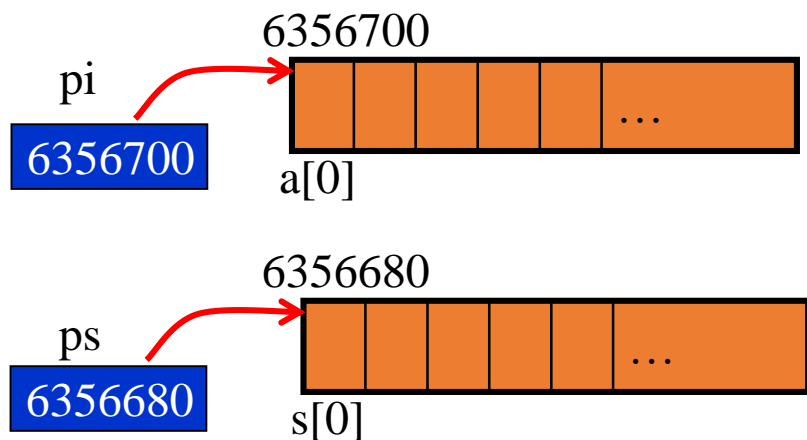
    pi = pi-5; //pi指向a[5]
    ps -= 5; //ps指向s[5]
    printf("pi: %d, ps: %d\n", pi, ps);
    printf("*pi: %d, *(pi+2): %d\n", *pi, *(pi+2));
    printf("*ps: %d, *(ps+2): %d\n", *ps, *(ps+2));

    return 0;
}
```

输出?

观察输出结果，并分析原因

## 例7-5 指针的加减运算



观察输出结果，  
并分析原因

40 20

pi: 6356700, ps: 6356680

pi: 6356740, ps: 6356700

a: 100, 101, 102, 105

s: 10, 11, 12, 15

pi: 6356740, ps: 6356700

pi: 6356720, ps: 6356690

\*pi: 105, \*(pi+2): 107

\*ps: 15, \*(ps+2): 17

输出

```
#include <stdio.h>
#define MAX_N 10
int main()
{
    int *pi, a[MAX_N], i;
    short s[MAX_N], *ps;
    printf("%d %d\n", sizeof(a), sizeof(s));

    pi = a; //pi指向a[0]
    ps = s; //ps指向s[0]
    printf("pi: %d, ps: %d\n", pi, ps);

    for(i = 0; i < MAX_N; i++, pi++, ps++)
    {
        *pi = 100+i;
        *ps = i+10;
    }
    printf("pi: %d, ps: %d\n", pi, ps);
    printf("a: %d, %d, %d, %d\n", a[0], a[1], a[2], a[5]);
    printf("s: %d, %d, %d, %d\n", s[0], s[1], s[2], s[5]);
    printf("pi: %d, ps: %d\n", pi, ps);

    pi = pi-5; //pi指向a[5]
    ps -= 5; //ps指向s[5]
    printf("pi: %d, ps: %d\n", pi, ps);
    printf("*pi: %d, *(pi+2): %d\n", *pi, *(pi+2));
    printf("*ps: %d, *(ps+2): %d\n", *ps, *(ps+2));

    return 0;
}
```

6356680+1 is 6356682

pi: 6356700  
pi++: pi is 6356704  
means 6356700+1 is 6356704

## 7.3.1 指针与整数的加减

```
char *strcpy(char dest[ ], char src[ ])
{
    int i;
    for(i=0; (dest[i] = src[i]) != '\0'; i++)
        ;
    return dest;
}
```

数组版本

```
char *strcpy(char *dest, char *src)
{
    char *d = dest;
    while((*d++ = *src++) != '\0')
        ;
    return dest;
}
```

指针版本

从右向左结合



\*p++ 等价于 \*(p++), 由于是后置++运算, 先执行\*p, 然后p再自增1 (地址增加)。  
(\*p)++ 表示p先与\*结合, ++作用于变量p所指向的变量 (是变量值增加)。

## 7.3.1 指针与整数的加减

```
while((*d++ = *s++) != '\0')  
    ;
```

\*s++ 等价于 \*(s++), 由于是后置++运算, 先执行\*s, 然后s再自增1

“约”等价

```
while ((*d = *s) != '\0')  
{  
    d++;  
    s++;  
}
```

- 表达式(\*d = \*s), 先把\*s赋值给\*d, 然后返回\*d
- 接着执行表达式 \*d != '\0'
- 若表达式 \*d != '\0' 成立, 执行 d++ 和 s++ 两条语句

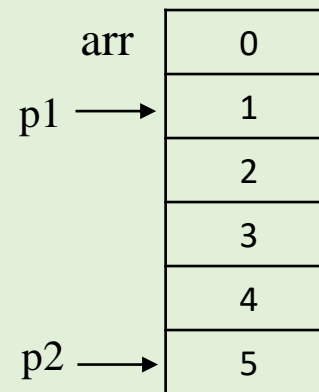
执行完后, 左边代码的指针比右边多一个位置

## 7.3.1 指针与整数的加减

`*p++` 等价于 `*(p++)`

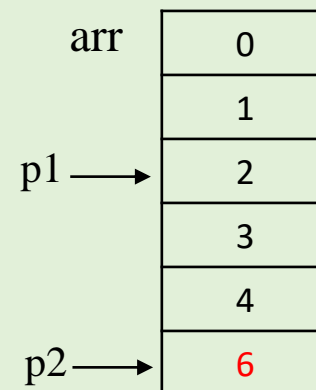
`(*p)++` 表示p先与\*结合, ++作用于变量p所指向的变量

```
int a, b, arr[ ] = {0, 1, 2, 3, 4, 5};  
int *p1 = &arr[1], *p2 = &arr[5];
```



```
a = *p1++;  
b = (*p2)++;
```

a is 1  
b is 5  
arr[5] is 6

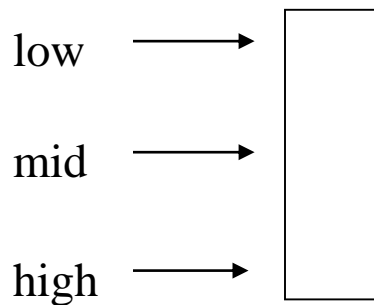




## 7.3.1 指针与整数的加减

注意：两指针不能相加？

计算中间指针：



$$\text{mid} = (\text{low} + \text{high}) / 2;$$

哪个用法正确？

$$\text{mid} = \text{low} + (\text{high} - \text{low}) / 2;$$

## 7.3.1 指针与整数的加减

- **指针相减**: 两个指向同一数组成员的指针可进行相减, 其结果为两指针间相差元素的个数。

### 例7-6

```
int *i_low, *i_hi, a[MAX_N];  
double *d_low, *d_hi, d[MAX_N];
```

```
i_low = a;    i_hi = &a[2];  
d_low = d;    d_hi = &d[2];
```

```
printf("i_hi = %d\ni_low = %d\n", i_hi, i_low);  
printf("i_hi - i_low = %d\n\n", i_hi - i_low);  
printf("d_hi = %d\nd_low = %d\n", d_hi, d_low);  
printf("d_hi - d_low = %d\n\n", d_hi - d_low);
```

请运行程序,  
并理解该输出  
结果

输出

```
i_hi = 6356704  
i_low = 6356696  
i_hi - i_low = 2
```

**704 - 696 = 2**

```
d_hi = 6356632  
d_low = 6356616  
d_hi - d_low = 2
```

**32 - 16 = 2**

指针相减的结果只取决于两指针间相隔元素的数量, 而与数组元素所占字节数无关。

## 7.3.1 指针与整数的加减

---

$$0 + 1 = ?$$

## 7.3.2 指针的比较

两指针间的比较：两个指向同一类型的指针，可以进行 “==, >, <” 等关系运算，其实就是地址的比较。

```
int array[N], *p, *q;  
p = &array[0];  
q = &array[3];  
... // we can compare p with q
```

### 1) 指针与 0 的比较

- ◆ 在指针未指向任何实际的存储单元时，或指针所指向的存储单元已经不存在时，通常将其标记为无效指针（0）。
- ◆ 为了表示指针的 0 在类型上不同于整数类型的 0，在C的标准头文件中定义了一个等于 0 的符号常量NULL。
- ◆ 数值为 0 或 NULL 的指针不指向任何内容，称为空指针（重要的表示）。
- ◆ 数值 0 是唯一可以不将整数转换为指针类型而直接赋给指针变量的整数值。
- ◆ 为了避免和常量值相区别，通常用用NULL表示空指针。

## 1) 指针与 0 的比较

例7-7 多行数据的平均值。从标准输入上读入N ( $0 < N < 1000000$ ) 行数据，每行含有  $n_i$  个由空格符分隔的实数。在标准输出上输出每个输入行的行号、数据的个数以及该行数据平均值，每行输入数据对应一个输出行，行号与数据个数间以冒号和一个空格分隔，数据个数与数据平均值之间以空格符分隔。

输入样例

输出样例

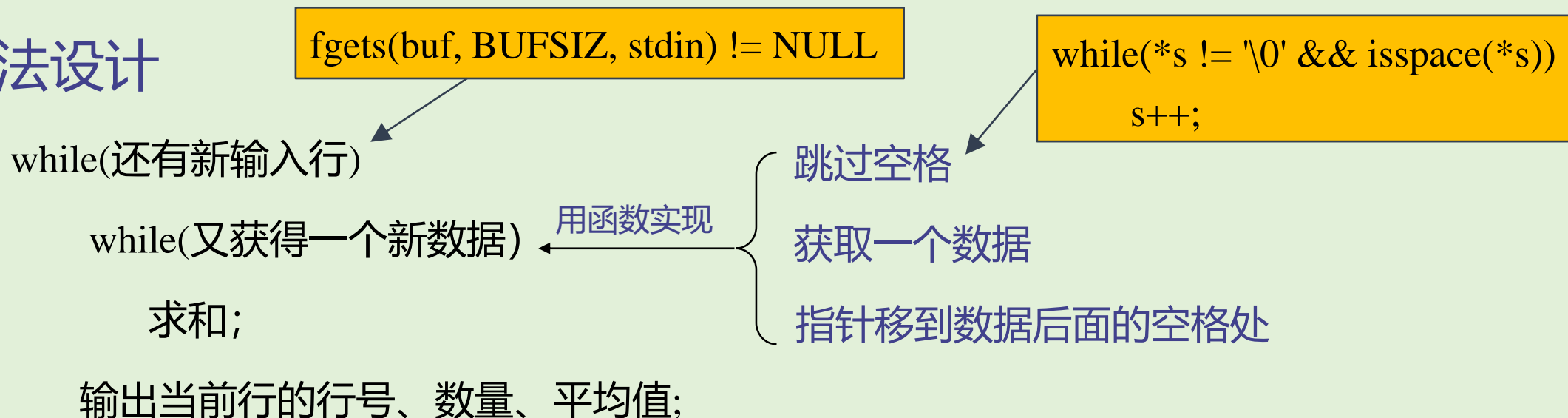
```
c7-7.in - 记事本
文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)
1.1 2.2
1 2 3
1 2 3 4
1.1 2.2 3.3

6( Windows (CRLF) UTF-8

C:\alac\example\chap7>c7-7 < c7-7.in
1: 2 1.650000
2: 3 2.000000
3: 4 2.500000
4: 3 2.200000

C:\alac\example\chap7>
```

### 算法设计



## 例7-7 多行数据的平均值。

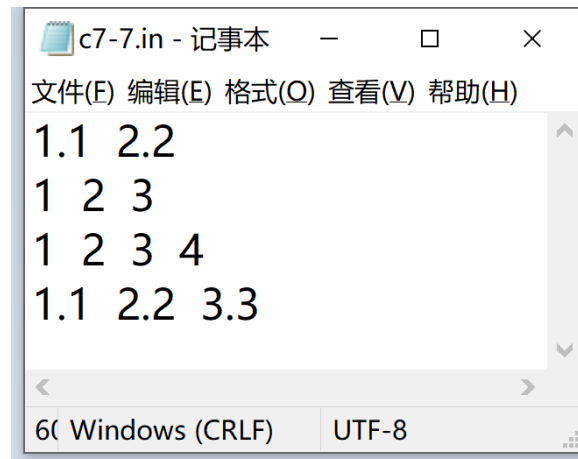
这是一个处理以“行”为单位的输入数据的基本框架，值得掌握！

BUFSIZ是一个常用的说明缓冲区大小的符号常量，实际数量取决于编译系统的版本和编译选项，常见的数值为512或4096。

```
int main()
{
    int i, n;
    double d, subsum;
    char buf[BUFSIZ], *p;

    for(i=1; fgets(buf, BUFSIZ, stdin) != NULL; i++)
    {
        subsum = 0;
        for(n=0, p=buf; (p = get_value(p, &d)) != NULL; n++)
            subsum += d;
        if(n>0)
            printf("%d:%d %f\n", i, n, subsum/n);
    }
    return 0;
}
```

读出buf[]中实数数据的各字段，并把指针指到实数后面的第一个空白位



```
c7-7.in - 记事本
文件(E) 编辑(E) 格式(O) 查看(V) 帮助(H)
1.1 2.2
1 2 3
1 2 3 4
1.1 2.2 3.3
6( Windows (CRLF) UTF-8
```

```
char *get_value(char *s, double *d)
{
    while(*s != '\0' && isspace(*s))
        s++; // 跳过空格，指针指到非空格处
    if(sscanf(s, "%lf", d) != 1)
        return NULL;
    while(*s != '\0' && !isspace(*s))
        s++; // 指针移到实数后的空白位
    return s;
}
```

为什么不用scanf输入一行的多个数据？

空格和\0都作为空白符处理，不能进行“行”(hang)处理

这个程序非常实用，记下来，牢固掌握！

## 2) 指向同一数组的两个指针进行比较

c7-8 子串逆置。从标准输入上读入以空格分隔的字符串  $s$  和  $t$ ，将  $s$  中首次与  $t$  匹配的子串逆置后再输出  $s$ ，当  $s$  中无与  $t$  匹配的子串时直接输出  $s$ 。

输入样例：  
abc123defg 123  
abc123defg 234

输出样例：  
abc321defg  
abc123defg

思路：两个一维字符数组存储  $s$  和  $t$ ，两个指针标记匹配子串中逆序位置。

### ● 算法设计

获得两个字符串

if(存在匹配项)

逆序：逆序的起始位置和终止位置

输出

if(( p = strstr(str, substr) ) != NULL)

p

p+strlen(substr)-1

用函数实现

while(还有逆序字符)

交换数据

起始位置移动

终止位置移动

## 2) 指向同一数组的两个指针进行比较

c7-8 子串逆置。从标准输入上读入以空格分隔的字符串  $s$  和  $t$ ，将  $s$  中首次与  $t$  匹配的子串逆置后再输出  $s$ ，当  $s$  中无与  $t$  匹配的子串时直接输出  $s$ 。

输入样例：  
abc123defg 123  
abc123defg 234

输出样例：  
abc321defg  
abc123defg

解题思路：

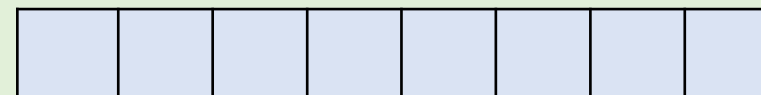
$first < last$

while(还有逆序字符)

交换数据

起始位置移动

终止位置移动



first

last

$first < last$

$first++$ ,  $last--$ ;



## 2) 指向同一数组的两个指针进行比较

c7-8 子串逆置。从标准输入上读入以空格分隔的字符串  $s$  和  $t$ ，将  $s$  中首次与  $t$  匹配的子串逆置后再输出  $s$ ，当  $s$  中无与  $t$  匹配的子串时直接输出  $s$ 。

输入样例：  
abc123defg 123  
abc123defg 234

输出样例：  
abc321defg  
abc123defg

解题思路：

$first < last$

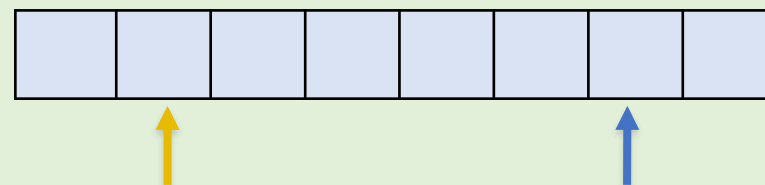
while(还有逆序字符)

交换数据

起始位置移动

终止位置移动

$first++$ ,  $last--$ ;



first

last

$first < last$

## 2) 指向同一数组的两个指针进行比较

**c7-8 子串逆置。**从标准输入上读入以空格分隔的字符串  $s$  和  $t$ ，将  $s$  中首次与  $t$  匹配的子串逆置后再输出  $s$ ，当  $s$  中无与  $t$  匹配的子串时直接输出  $s$ 。

输入样例：  
abc123defg 123  
abc123defg 234

输出样例：  
abc321defg  
abc123defg

解题思路：

$first < last$

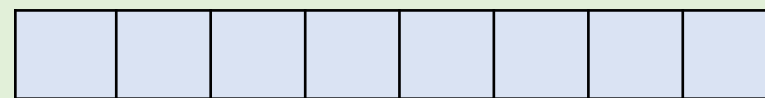
while(还有逆序字符)

交换数据

起始位置移动

终止位置移动

$first++$ ,  $last--$ ;



first

last

$first < last$

## 2) 指向同一数组的两个指针进行比较

**c7-8 子串逆置。**从标准输入上读入以空格分隔的字符串  $s$  和  $t$ ，将  $s$  中首次与  $t$  匹配的子串逆置后再输出  $s$ ，当  $s$  中无与  $t$  匹配的子串时直接输出  $s$ 。

输入样例：  
abc123defg 123  
abc123defg 234

输出样例：  
abc321defg  
abc123defg

解题思路：

$first < last$

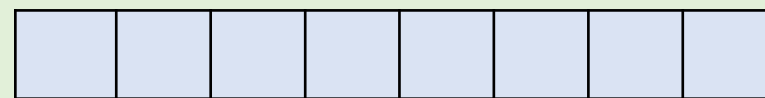
while(还有逆序字符)

交换数据

起始位置移动

终止位置移动

$first++$ ,  $last--$ ;



first

last

$first < last$

## 2) 指向同一数组的两个指针进行比较

**c7-8 子串逆置。**从标准输入上读入以空格分隔的字符串  $s$  和  $t$ ，将  $s$  中首次与  $t$  匹配的子串逆置后再输出  $s$ ，当  $s$  中无与  $t$  匹配的子串时直接输出  $s$ 。

输入样例：  
abc123defg 123  
abc123defg 234

输出样例：  
abc321defg  
abc123defg

解题思路：

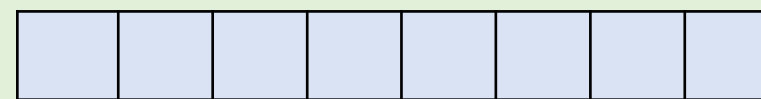
first < last 不成立

while(还有逆序字符)

交换数据

起始位置移动

终止位置移动



first

last

first > last

## 2) 指向同一数组的两个指针进行比较

**c7-8 子串逆置。**从标准输入上读入以空格分隔的字符串 *s* 和 *t*，将 *s* 中首次与 *t* 匹配的子串逆置后再输出 *s*，当 *s* 中无与 *t* 匹配的子串时直接输出 *s*。

输入样例：  
abc123defg 123  
abc123defg 234

输出样例：  
abc321defg  
abc123defg

```
#include<stdio.h>
#include<string.h>
void rev(char *, char *);
int main(){
    char str[BUFSIZ], substr[BUFSIZ], *p;
    scanf("%s%s", str, substr);
    if((p = strstr(str, substr)) != NULL)
        rev(p, p + strlen(substr)-1);
    puts(str);
    return 0;
}
```

```
void rev(char* first, char* last)
{
    int tmp;
    while(first < last)
    {
        tmp = *last;
        *last = *first;
        *first = tmp;
        first++, last--;
    }
}
```

**这个程序非常实用，记下来，牢固掌握，灵活运用！**

## \*7.3.3 指针强制类型转换和动态存储分配

### 1) 指针的强制类型转换

- 当需要不同类型指针互相赋值时，可通过强制类型转换改变对指针类型的解释，以保证所需操作在语法上的正确性。即在指针前加上圆括号括起来的目标类型。

- 常见的非法指针运算包括：

指针间的加、乘、除，指针加减浮点数

指针移位操作

指针的位运算

不同类型指针间的直接赋值

```
int *ia, *ip, n, arr[3][6];  
short s, sa[16], *id;  
ip = (int *) sa;  
ia = (int *) arr;  
id = (short *) &n;  
ip = (int *) id;
```

## \*7.3.3 指针强制类型转换和动态存储分配

### 2) 使用malloc动态申请空间

- 在C中可以使用标准库函数malloc动态申请一块内存空间（以字节为单位），用于初始化指针变量，即动态存储分配。

```
void* malloc(size_t size);  
void free(void* mem);
```

size\_t是在<stdio.h>中定义的一种整数类型，等价于 unsigned int。参数size表示字节数。

- C语言定义了通用指针类型void\*。具有void\*类型的指针可以赋给任何类型的指针变量，具有void\*类型的指针变量可以接受和保存任意类型的指针。

## 2) 使用malloc动态申请空间

- 使用malloc初始化指针变量的常见用法

```
char *s;  
int *intptr;  
s = (char *) malloc(32); /* s指向大小为32个字节（字符）的空间*/  
s = (char *) malloc(strlen(p)+1); /* s指向能正好存放字符串p的空间*/  
intptr = (int *) malloc( sizeof(int) *10); /* ptr指向能存放10个整型的空间*/  
...  
free(s); free(intptr);
```

运算符sizeof用来计算所在系统中某种类型或类型变量所占的长度（以字节为单位）。

- 使用malloc申请到的动态空间在不用时应使用函数free释放，如：free(s)
- 使用malloc和free函数要用库函数 `#include <stdlib.h>`



## 2) 使用malloc动态申请空间

```
char *s;  
int *intptr;  
s = (char *) malloc(strlen(p)+1); /* s指向能正好存放字符串p的空间*/  
intptr = (int *) malloc( sizeof(int) *10); /* ptr指向能存放10个整型的空间*/  
...  
free(s); free(intptr);
```

**注意：**由malloc申请的动态空间，必须要用free函数来释放。

由malloc申请的动态空间不及时释放，是造成许多软件出现内存泄漏的主要原因！

**内存泄漏(memory leak)：**指软件在长时间运行过程中造成内存越来越少，最终可能导致系统内存耗尽而导致软件性能下降或不能使用的现象。

## 2) 使用malloc动态申请空间

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    int *intptr, n, i;
    scanf("%d", &n);
    intptr = (int *) malloc(sizeof(int)*n);
    // 不要这样用 int a[n];

    for(i=0; i<n; i++)
        scanf("%d", intptr+i);
    for(i=0; i<n; i++)
        printf("%d", intptr[i]);
    printf("\n");
    free(intptr);
    return 0;
}
```

输入

5

1 2 3 4 5

输出

?

## 2) 使用malloc动态申请空间

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
int main()
{
    char *s;
    int n, i;
    scanf("%d", &n);
    s = (char *) malloc(n); //不要这样 char s[n];
    scanf("%s", s);
    printf("%d, %d\n", strlen(s), sizeof(s));
    for (i = 0; * (s + i) != '\0'; i++)
        printf("%c", * (s + i));
    printf("\n");
    free(s);
    return 0;
}
```

输入

9

abcde

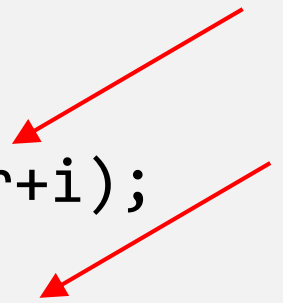
输出

?

## 7.4 指针与数组

### 指针与数组关系非常密切

```
int *intptr, n, i;  
...  
for(i=0; i<n; i++)  
    scanf("%d", intptr+i);  
for(i=0; i<n; i++)  
    printf("%d", intptr[i]);
```



## 7.4.1 指针与一维数组

- 数组名可以看作是一个指针常量，指向数组起始位置（第一个元素的位置）

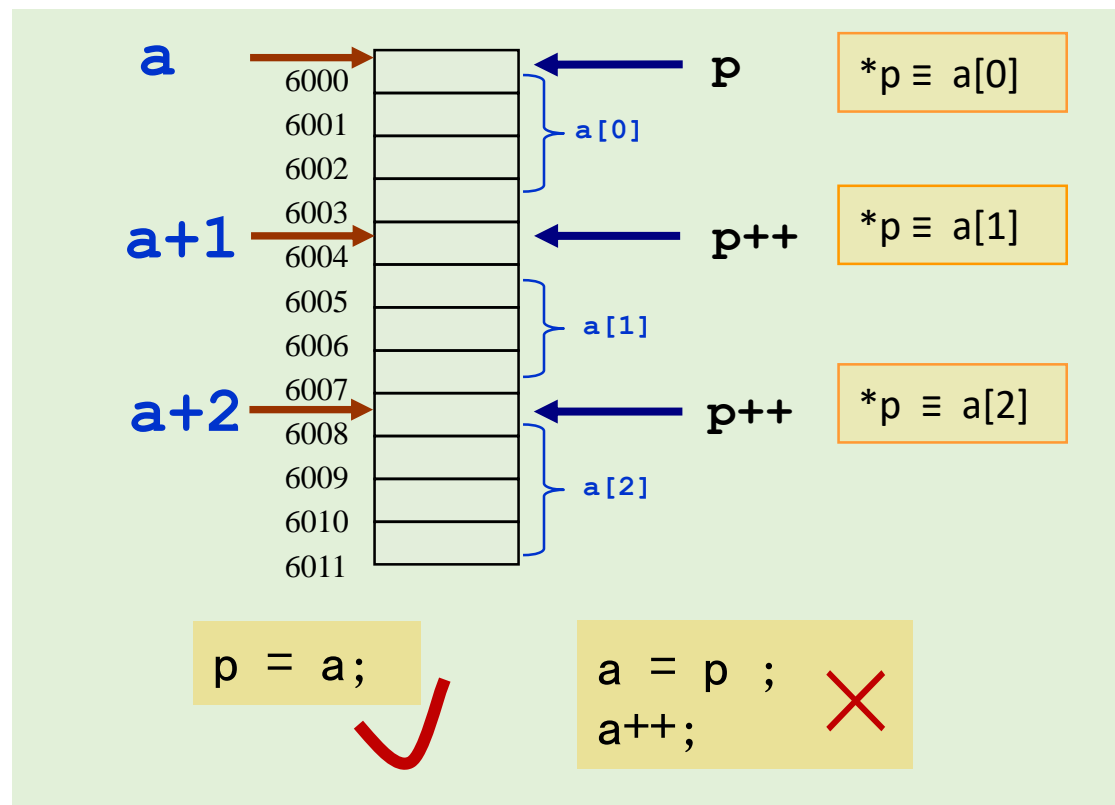
- ◆ 不能修改这个指针的值(指向)
- ◆ 可以定义函数的参数为数组

- 指针也可赋值为数组名

- ◆ `int *p, a[10];`
- ◆ `p = a;`

- 数组元素的几种等价引用形式

- ◆ `a[i]` 数组+下标
- ◆ `*(a+i)` 数组+偏移量
- ◆ `p[i]` 指针+下标
- ◆ `*(p+i)` 指针+偏移量



**数组名和指针（变量）有区别：**数组名可以赋值给对应的指针类型，数组名实际为常量（不可被赋值），一般的指针则是变量。

## 例：输入输出数组的全部元素

### 下标法

```
int  a[10];
int  i;
for(i=0; i<10; i++)
    scanf("%d", &a[i]);

for(i=0; i<10; i++)
    printf("%d ", a[i]);
```

### 指针法

```
int  a[10];
int  *p, i;

for(p=a; p<(a+10); p++)
    scanf("%d", p);

for(p=a; p<(a+10); p++)
    printf("%d ", *p);
```

for循环头能否为 for( ; a<(p+N); a++)

**No!** 数组名是数组首元素的地址，是指针常量

# 指针与数组互换的举例

```
int a[ ]={1, 2, 3, 4, 5, 6};  
int *p;  
p = a;  
p[0] = p[2]+p[3];  
*(p+3) += 2;  
printf("%d, %d\n", *p, *(a+3));  
...
```

指针指向数组

指针可以按数组形式访问

也可以按偏移量形式访问

数组可以按指针形式访问

输出

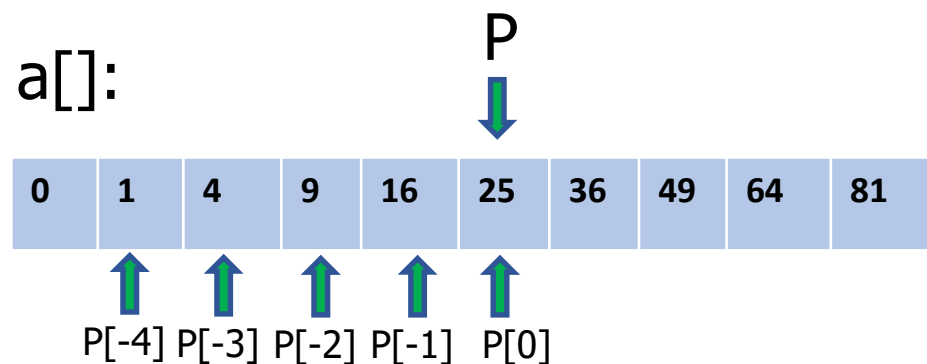
?

# 利用指针实现数组的负数下标

在C程序中可以利用指针与数组的可互换性来支持数组的负数下标:

- 语法合法
- 语义是否合法取决于是否越界

```
int  a[10], *p = &a[5];  
int  i;  
for (i=0; i<10; i++)  
    a[i] = i*i;  
for (i=0; i<5; i++)  
    printf("%d  ", p[-i]);
```



输出:  
25 16 9 4 1



## 7.4.2 字符指针与字符串

- 对于字符串常量，本质是一个 '\0' 结尾的字符数组，C编译程序会自动为它分配一个空间来存放这个常量，字符串常量的值是该字符数组的首地址，其类型是字符指针char \*（严格说是 const char \* const）。
- printf("a constant character string\n"); 传递给函数的是字符串第一个字符的指针。

**注意：**字符数组和字符指针使用时容易混淆！（见下例）

```
char *char_ptr, word[20];
```

```
char_ptr = "point to me";  
// 正确，把字符串的首地址赋给指针变量。 ✓
```

```
word = "you can't do this";  
// 错误，word是常量，正确做法为：strcpy(word, "..."); ✗
```

```
char a[]="hello", *p="hello";  
// p 指向常量字符串
```

```
a[0] = 'c'; // 正确 ✓
```

```
*p = 'c';  
// 错误，不能修改常量值 ✗
```

## 7.4.2 字符指针与字符串

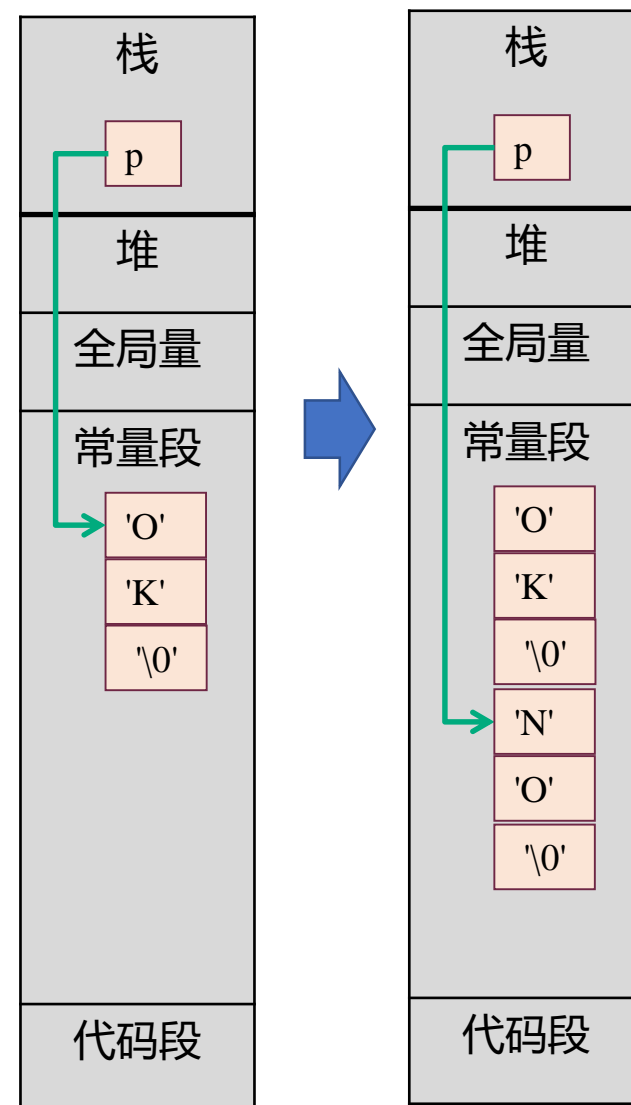
执行字符指针变量赋值语句时发生了什么？

```
char *p = "OK";
```

- 在常量区申请合适的内存空间，将字符串"OK"依次存入内存单元（包括字符串尾部的 '\0' ）
- 返回常量字符串中首字符地址，将其赋值给字符指针 p
- 对 p 赋新值 

```
p = "NO";
```

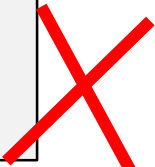
  - 不是修改原字符串的值，而是让指针变量 p 指向新字符串的首字符



# 指针类型与数组类型的差异

- 数组是一片连续的存储空间，而指针只是一个保存地址的存储单元，未经正确赋值之前不指向任何合法的存储空间，不能通过它进行任何数据访问。


```
double d,*dp;  
*dp = 5.678;  
*dp = 100;
```



运行时错误

指针未初始化  
就使用

```
char *string;  
scanf("%s", string);  
strcpy(string, "Hello");
```



- dp无所指，**野指针**，当要向一个指针指向的空间赋值时，一定要检查有没有给这个指针分配空间。也可以做以下提醒：定义指针时如果没有初始化，就让它指向NULL，即空指针，表示不能操作空指针所指向的空间。
- 数组名是一个**指针常量**而不是一个变量，是与一片固定的存储空间相关联的。可以对数组元素赋值而不可以对数组变量本身赋值。而指针变量本身是一个变量，可以根据需要进行赋值，从而指向任何合法的存储空间。通过数组所能访问的数据的数量在数组定义时就已确定，而通过指针所能访问的数据的数量取决于指针所指向的存储空间的性质和规模。

# 小 结

---

- 变量在内存中的存储形式和地址的概念
- 指针的概念、用法、指针运算
- 指针向函数传递参数，返回指针类型
- 指针、数组与字符串之间的密切关系
- 指针与数组的区别与联系
- 指针操作数组的方法

## 课堂作业：读程序观察数据变化

执行完 sum 函数后，a, b, c 的值分别是什么？

```
int a = 2, b = 3, c = 4 ;  
c = sum(a, &b);  
...  
  
int sum(int x, int *py)  
{  
    x += *py;  
    *py = x;  
    return x;  
}
```

## 课堂作业：程序填空

**c7-8 子串逆置。**从标准输入上读入以空格分隔的字符串s和t，将s中首次与t匹配的子串逆置后再输出s，当s中无与t匹配的子串时直接输出s

输入样例：  
abc123defg 123  
abc123defg 234

输出样例：  
abc321defg  
abc123defg

```
#include<stdio.h>
#include<string.h>
void rev(char *, char *);
int main()
{
    char str[BUFSIZ], substr[BUFSIZ], *p;
    scanf("%s%s", str, substr);
    if((p = strstr(str, substr)) != NULL)
        rev(p, p + strlen(substr)-1);
    puts(str);
    return 0;
}
```

```
void rev(char* first, char* last)
{
    int tmp;
    while(first < last)
    {
        _____
        _____
        _____
        _____
    }
}
```