

C语言高级篇

第七讲

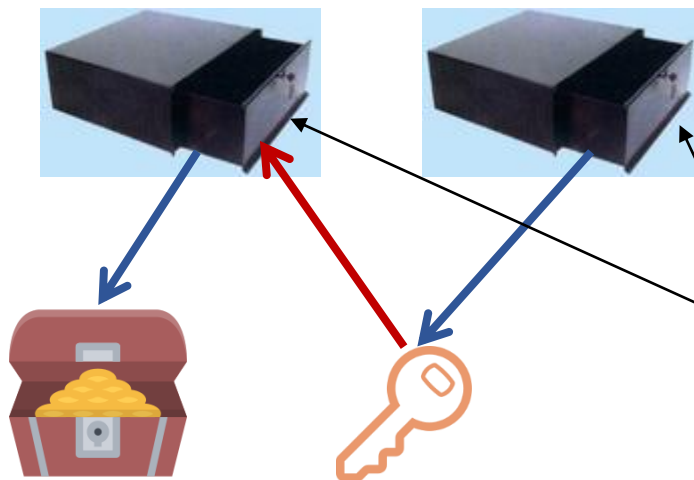
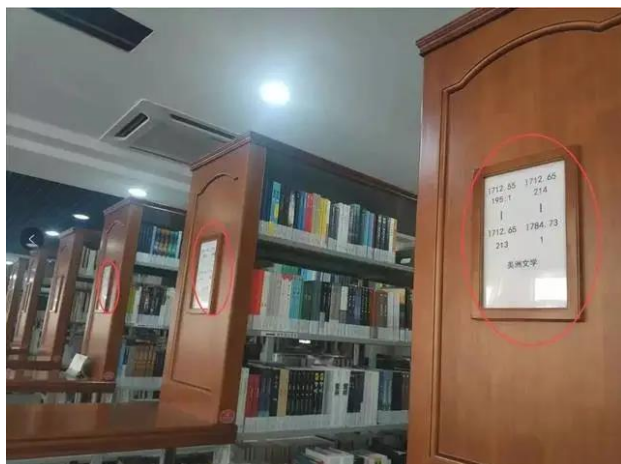
指针基础

Pointer Introduction

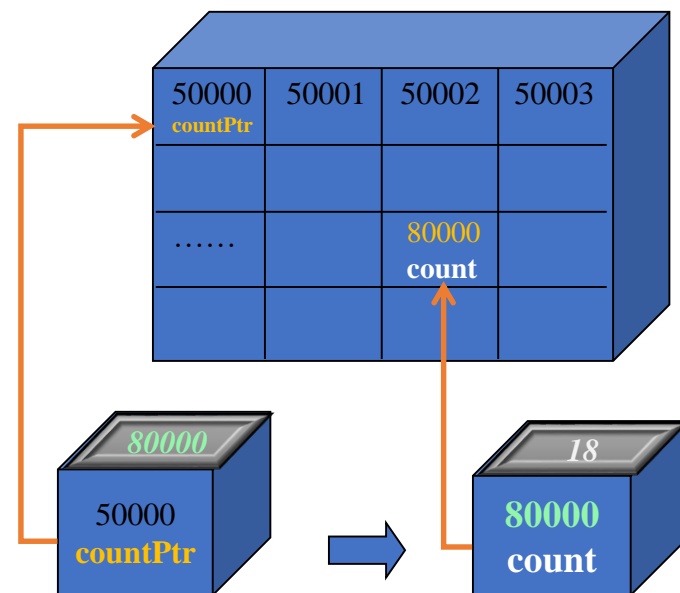


第七讲 指针基础

一句话印象：变量保存具体数值，指针保存变量的地址



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



count: 一般变量
countPtr: 指针变量

第七讲 指针基础

- 指针是 C语言一个最强大的特性
- 指针是 C语言的精髓之一，可以直接访问原始内存数据
- 使程序简洁、高效、紧凑
- 指针是 C语言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C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、图

第七讲 指针基础

1. 指针、地址、变量、内存之间的关系
2. 数据类型的回顾
3. 指针变量
4. 指针运算
5. void *指针与 memcpy 函数
6. malloc 函数与堆空间
7. 本章小结

7.1 指针、地址、变量、内存之间的关系

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

变量可以看作是内存空间的别名

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

内存（Memory）

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

7.1 指针、地址、变量、内存之间的关系

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, ch;  
aPtr = &a;  
a = 'c';  
*aPtr = 'x';
```

指针：是一种(类)数据类型。通常说的指针，是指针变量，其存储的值是其他变量的地址，其作用是指向内存中的某个实体。指针与地址密不可分，有时也把变量的地址称为该变量的指针。

- 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

&：取地址符

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，即，类型 char *

内存 (Memory)

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

7.1 指针、地址、变量、内存之间的关系

任意的地址都可以作为指针的值吗？

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（低地址端）。

```
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);
```



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

	地址偏移量 →															
地址base ↓	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xA	0xB	0xC	0xD	0xE	0xF
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							

变量在内存中的分配情况

哪些地址可作为合法的指针值？

7.1 指针、地址、变量、内存之间的关系

```
#include <stdio.h>
int main()
{
    int a = 0;
    printf("address of a: %p\n", &a);
    printf(" sizeof of a: %d\n", sizeof(a));
    return 0;
}
```

注意%X和%p输出的区别：两者都是16进制输出，但%p按照地址位数补前导0。

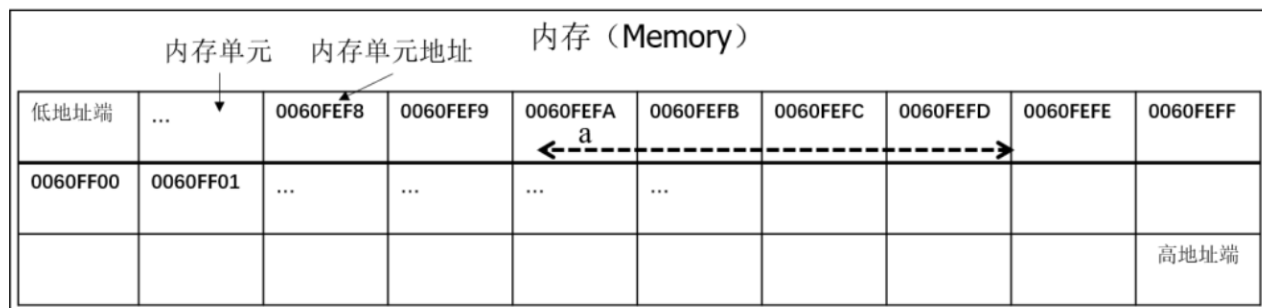
程序输出：

```
address of a: 000000000060FEFA
sizeof of a: 4
```

- 输出的第 1 行是 a 的地址值，这个地址值是用 16 进制表示的 64 位二进制数（8 字节）。读者运行时，这个结果可能是 32 位的（4 字节）。
- 在计算机中，用于表示地址值的二进制位数（或字节数）是固定宽度的，又称为 CPU 的寻址空间，一般是 4 字节（32 位）或 8 字节（64 位）。
- 上述结果表明，程序在运行时，变量 a 存放在以 000000000060FEFA 开始的连续 4 个字节的内存空间中（变量 a 占 4 个字节，见输出的第 2 行）。

7.1 指针、地址、变量、内存之间的关系*

内存、地址与变量示意图



一个 32 位的 CPU 或者操作系统，它用来寻址的地址位数是 32 位，寻址空间为 2 的 32 次方，即 4GB 的大小。如果计算机安装了超过 4GB 的内存，则多余的内存空间也是浪费。

大端存储模式是指将变量的高字节存放到内存空间的低地址处；小端存储模式是指将变量的低字节存放到内存空间的低地址处。Intel 系列处理器采用小端存储模式。网络传输（例如常用的 TCP/IP/UDP 协议）一般采用大端存储模式，即把接收到的第一个字节（存放在低地址）当作高字节。

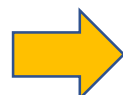
7.2 数据类型的回顾

语言中出现的每一个变量、常量以及表达式都是有类型的。请说出右侧每行表达式的类型。

- 因为计算机是个存储容量有限的离散系统，数学概念上的很多数都是无限且连续的（如实数），离散的系统无法精确表示数学上的所有数。数据类型定义了数据的存储长度（字节数）和编码方式，以有限的范围和精度来存储数学概念上的数。
- 对于一个字节的内存单元，里面存储的内容是 0xFF，如果不知道该内存单元属于那种数据类型则无法对其信息进行解释；如果它的数据类型是 signed char，则它表示的是整数-1；如果它的数据类型是 unsigned char，则它表示整数 255。



```
int a = 0;  
a + 0.1;  
a + 0.1f;  
double b = 2;  
b + 3 / 2 - 1.0;  
"abcd"[0];  
char c = 'C';  
c + 1;  
c + 1.0;
```



```
#include <stdio.h>  
int main()  
{  
    char a = 0xFF;  
    unsigned char b = 0xFF;  
    printf("%d vs %d\n", a, b);  
    return 0;  
}
```

7.3 指针变量

- 指针变量（也简称指针）是用来存放某个变量的地址的变量。
- 指针变量本质上和其他变量没有区别，只不过它的值不是通常意义上的数值，而是地址。
- 指针变量如果存放的是某一个变量的地址，那么说该指针指向那个变量。

在任何一个合法的 C 语言数据类型名后加上*号，就构成了一个指针类型，该类型的指针必须指向*号前的类型。例如 int*是一个指针类型，int*类型的指针必须指向 int 型变量（或者说 C 语言编译器会把 int*类型指针所指向的内存内容解释成为 int 型变量）。指针变量无论其指向的变量是什么类型，其自身的数据长度（sizeof表达式的值）都是固定的，因为都是存储的一个地址：在 32 位系统里占 4 个字节；在 64 位系统里占 8 个字节。

7.3.1 指针变量的定义

- 指针变量的定义如下：

<类型> * <变量名> [= <初始值>];

- 也可以连续定义多个同类型的指针变量：

<类型> * <变量名 1> [= <初始值 1>], * <变量名 2> [= <初始值 2>], ...;

其中*号说明<变量名>是一个指针，它指向的数据类型为<类型>。

```
int *pi;  
char * pc;  
float* pf;
```

定义三个不同类型的
指针变量

```
int *xPtr, x;  
int * yPtr, y;  
int* zPtr, z;
```

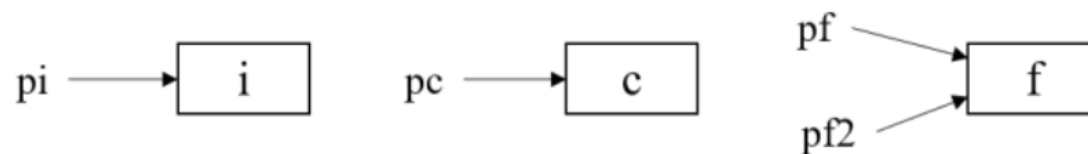
x, y和z，容易被误认
是指针变量。

```
int x, y, z;  
int *xPtr, *yPtr, *zPtr;
```

这种书写风格更好！

7.3.1 指针变量的定义

```
int i = 2;  
char c = 'a';  
double f = 2.0;  
int *pi = &i;  
char *pc = &c;  
double *pf = &f, *pf2 = pf;
```



变量名后有一个箭头表示该变量是一个指针（后文中很多 也简称为 ），箭头指向的方框表示指针所指向的变量。这种方式能形象地表示指针的指向关系，在分析指针相关的复杂算法时灵活采用这种示意图的方式能帮助读者更好地理解和设计算法。

7.3.2 指针与变量的关系

虽然常说指针指向某个变量，但指针变量的本质还是变量，它也会占用内存空间，也会有地址，与其他变量如 int, float 等没有任何本质区别，只不过指针的值表示的是内存空间的地址而已。

```
printf("&i = %p\t", &i);
printf("&c = %p\t", &c);
printf("&f = %p\n", &f);
printf("&pi = %p\t", &pi);
printf("&pc = %p\t", &pc);
printf("&pf = %p\n", &pf);
printf("sizeof(i) = %d\t", sizeof(i));
printf("sizeof(c) = %d\t", sizeof(c));
printf("sizeof(f) = %d\n", sizeof(f));
printf("sizeof(pi) = %d\t", sizeof(pi));
printf("sizeof(pc) = %d\t", sizeof(pc));
printf("sizeof(pf) = %d\n", sizeof(pf));
```

&i = 000000000061FE14 &c = 000000000061FE13 &f = 000000000061FE08
&pi = 000000000061FE00 &pc = 000000000061FDF8 &pf = 000000000061FDF0
sizeof(i) = 4 sizeof(c) = 1 sizeof(f) = 8
sizeof(pi) = 8 sizeof(pc) = 8 sizeof(pf) = 8

程序输出

变量在内存中的存储情况

61FDF0	61FDF1	61FDF2	61FDF7
pf							
61FDF8	61FDF9	61FDFA	61FDFF
pc							
61FE00	61FE01	61FE02	61FE07
pi							
61FE08	61FE09	61FE0A	61FE0B	61FE0C	61FE0D	61FE0E	61FE0F
f							
61FE10	61FE11	61FE12	61FE13	61FE14	61FE15	61FE16	61FE17
			c	i			
...							

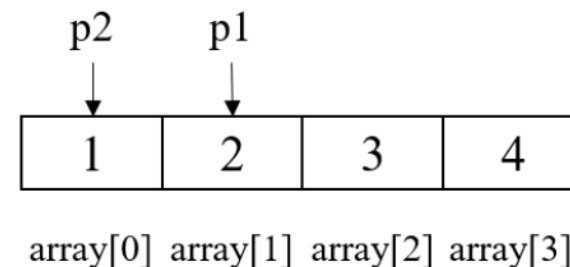
7.3.3 指针的初始化与赋值

- 指针在定义时最好进行初始化，或者赋值为空指针（null pointer），空指针的数值一般为0。在C语言标准头文件<stdio.h>中定义了一个宏 NULL 来表示空指针，其宏定义为：

```
#define NULL ((void *)0)
```

- 先暂时忽略 0 前面的(void *)，可简单认为 NULL 的值就是 0。未经初始化的指针称为野指针，其值为随机值，因此所指向的内容不是合法的内存空间，一定要赋值为合法的地址（比如已经定义的变量或数组元素的地址）后再使用。

```
int array[4] = {1, 2, 3, 4};  
int *p1 = &array[1];  
int *p2 = array; // 等价于 p2 = &array[0];
```



- 数组名作为表达式的值是数组第一个元素的地址，可以赋值给同类型的指针。因此数组名 array 可以赋值给 int* 型指针 p2，赋值后 p2 指向数组 array 的第一个元素。

7.3.3 指针的初始化与赋值

- 指针类型也可以用来定义数组：

<类型>* <数组名>[数组长度];

例如, `int *a[100]`; 定义了一个长度为 100 的指针数组, 数组元素是 `int*` 型指针。指针数组的主语为数组, 本质为一个数组, 数组元素为指针类型。后续还将介绍数组指针, 需要把这两者区分开来。

- 指针既然是一种合法的数据类型, 当然可以作为函数参数和函数返回值, 例如下面两个函数声明:

```
void func1(int *p1, int *p2);  
float *func2(float *a, float *b);
```

第一个函数的两个参数类型都是 `int*` 型指针; 第二个函数的两个参数是 `float*` 型指针, 返回值是 `float*` 型指针。指针作为函数参数和返回值的用法将在 7.3.7 节和 7.3.9 节介绍。

7.3.3 指针的初始化与赋值

- C 语言是一门强类型语言，在赋值时编译器会检查赋值运算符两边的数据类型，如果类型一致或类型兼容（所谓兼容，即能通过编译器的隐式类型转换完成右值类型向左值类型的临时转换）才允许赋值，否则编译器会报编译错误。例如：

```
double a = 0;  
int *b = a; // 编译错误，提示不兼容的类型
```

将浮点数解释为地址确实没有实际意义，逻辑上也讲不通

```
int a = 0;  
int *b = a; // 编译警告，提示发生了隐式类型转换
```

把变量 a 的类型从 double 改为 int 后，第 2 行的编译错误消失了。因为编译器进行了隐式类型转换，将 a 的值从 int 类型隐式转换为了指针类型（将整数值解释为地址还是有一定合理性的）。虽然有一定合理性，但还是存在隐患，因此编译器给出一个警告。

7.3.3 指针的初始化与赋值

再看一个不同类型指针之间赋值的例子：

```
int a = 3;  
char *pc = &a; // 编译警告，提示指针类型之间不兼容
```

上述代码第 2 行会产生一个编译警告，因为不同指针类型之间发生了赋值。a 是一个整型变量，它的地址&a 具有 int* 的类型，而 pc 是一个 char* 型指针，赋值符两边的指针类型不一致。&a 的值被隐式类型转换为 char* 类型，进而赋值给 pc。

上述代码执行后，char* 型指针 pc 指向了 int 型变量 a 的第一个字节。通过 pc 访问它指向的内存空间，会解释成 1 个字节的 char 型变量，但它其实是 int 型变量 a 的第一个字节内容。

7.3.3 指针的初始化与赋值

1. 当赋值运算符或算数运算符两边连接的表达式类型不一致时，编译器会使用隐式类型转换规则进行表达式值的类型提升或向目标类型进行转换。例如：
 - ✓ 浮点类型向整数类型通过截断小数实现隐式类型转换
 - ✓ 整数类型可以无损地隐式转换为数学上等价的浮点类型（需要在浮点数精度之内）
 - ✓ 数组名作为表达式可以隐式转换为指向数组首元素的指针
2. 有些类型之间不存在隐式类型转换（编译器禁止这种行为），例如：浮点数类型与指针类型之间无法进行隐式类型转换（把一个大象类型的值解释成一个小鸟类型的值是没有逻辑和意义的）
3. 隐式类型转换的对象是表达式的值，不会影响变量的类型和自身数值，例如

```
double b = 1.2; int a = b;
```

第 2 行，将变量 b 的值赋值给 a 时，发生了隐式类型转换，b 的值（浮点数 1.2）转换为 int 型（结果是整数 1）后赋值给了整型变量 a。变量 b 的类型和自身的值不会受到影响。

7.3.4 通过指针访问数据

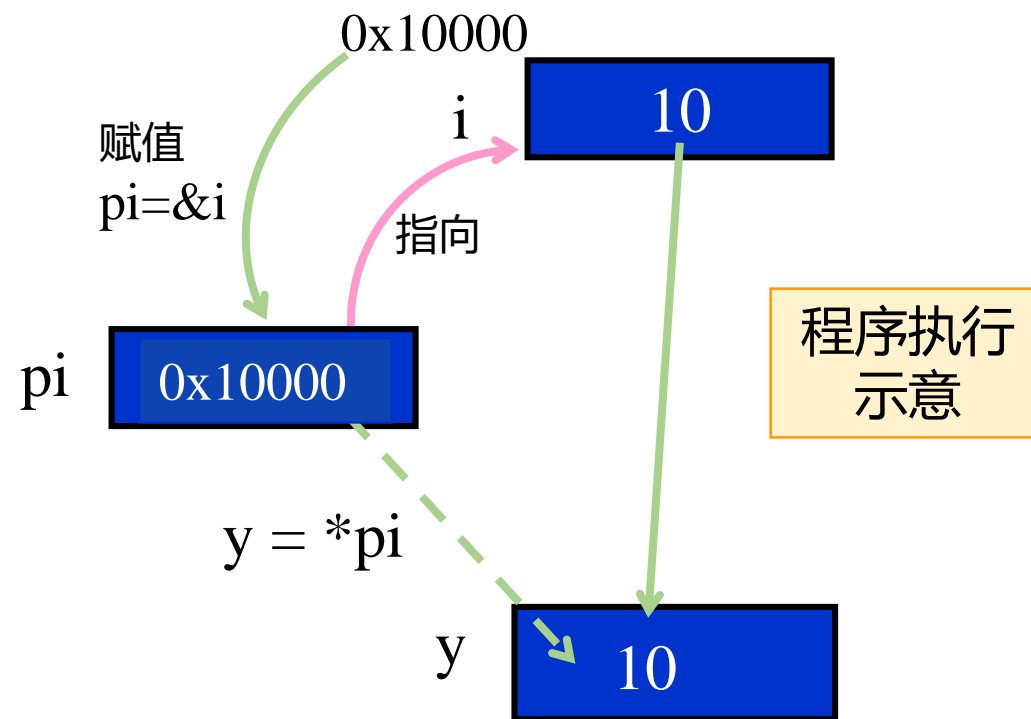
- 指针表示的是数据的存储地址，而非数据本身。
- 通过指针访问数据时，必须在其左侧加上一元运算符 '*'，解引用运算符。
- 单目运算符 *：用来取某地址中内容的运算符。

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



7.3.4 通过指针访问数据

解引用运算符 '*' 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;
```



```
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

以下代码会引发运行时错误 (run-time error)，这是一个严重的错误。因为指针pd 定义时并未初始化，后续也没有赋值语句对 pd 进行赋值，因此它是一个野指针（不知道指向何处）。对野指针进行解引用是十分危险的，因为它指向的内存空间不可访问或者没有合法分配。

```
double *pd;  
*pd = 5.0;
```

7-4 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？可以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

【例6-10解法】数据中的最长行 输入若干行字符串，输出最长行的长度和字符串

```
//二维数组所有元素初始化为'\0'
char arr[2][MAX_N] = {{""}};
int in = 1, longest = 0;
int max_len = 0, len, tmp;
while (gets(arr[in]) != NULL)
{
    len = strlen(arr[in]);
    if (len > max_len)
    {
        max_len = len;
        tmp = in;
        in = longest;
        longest = tmp;
    }
}

printf("%d: %s\n", max_len, arr[longest]);
```

longest始终
记录最长行

程序算法分析（示例）：

目前最长：arr[longest = 0] = "abcd"

输入前：arr[in = 1] = "ab"

第2行更短，新的输入存入第2行，即下一条：

输入后：arr[in = 1] = "abdefghi"

若 len > max_len（新输入的字符串更长），

则 $\text{max_len} \leftarrow \text{len}$ ，in 和 longest 交换数据，字符数组变为：

输入前：arr[in = 0] = "abcd"

目前最长：arr[longest = 1] = "abdefghi"

保持长的第2行，下一次新输入存入第1行，即：

待输入：arr[in = 0] = "??"

若输入字符串比目前最长的字符串短，不做处理，接着检查下一个输入字符串，存入当前行。

输入文件结束时，输出结果。

本设计比较巧妙。不需要拷贝数组（字符串）。longest记录已输入的最长行，in表示将要输入的行。

7-4

算法（用指针完成）：

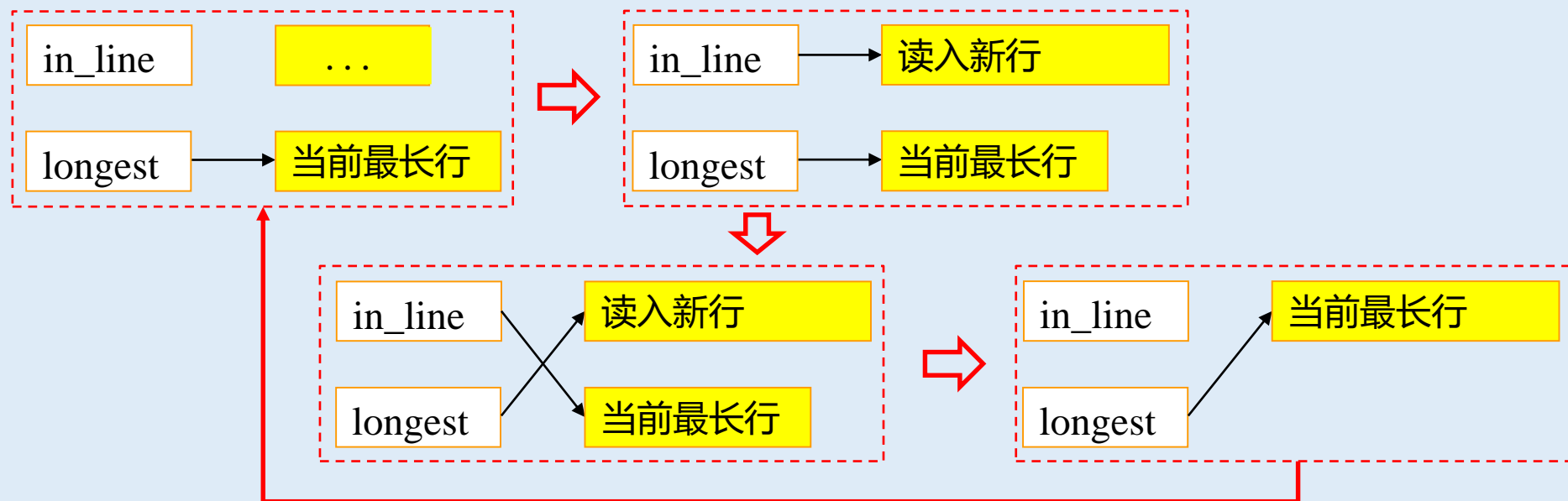
设指针变量 `in_line` 和 `longest` 分别指向当前行（新行）和当前最长行

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



```

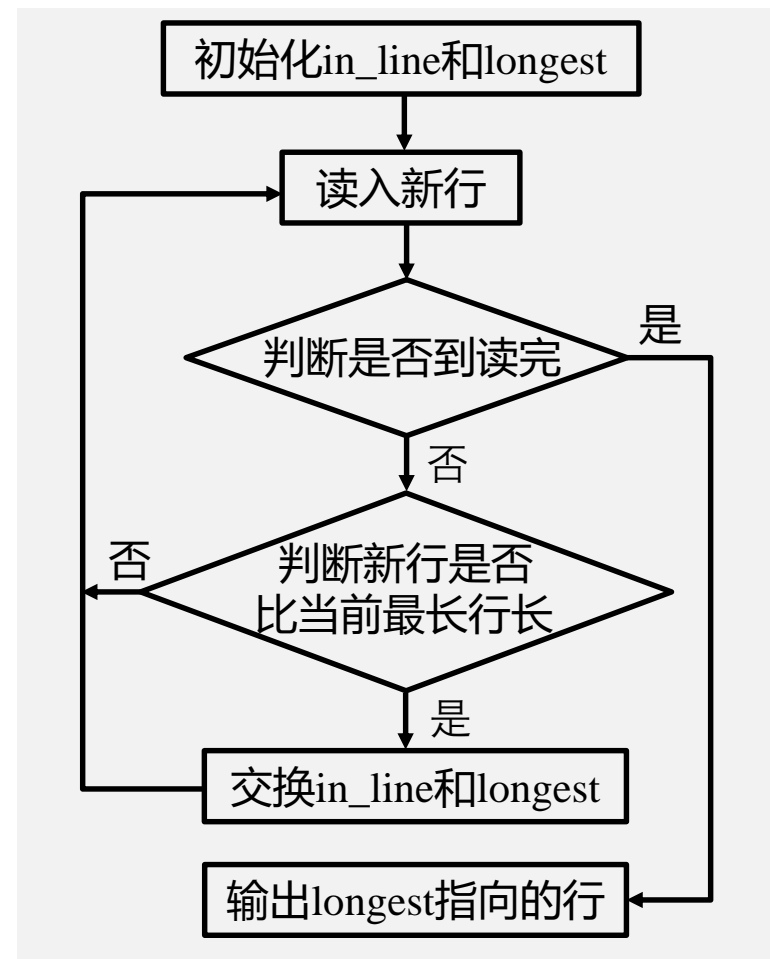
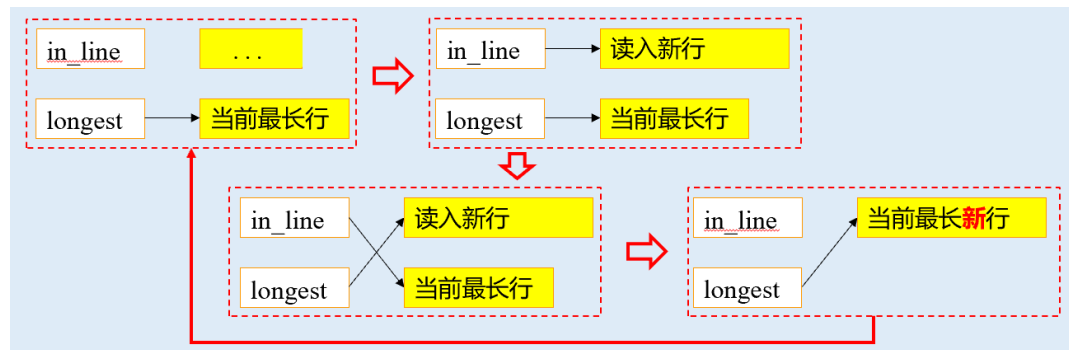
#include<stdio.h>
#include<string.h>
#define MAX 100
int main()
{
    char arr_1[MAX], arr_2[MAX] = "";
    char *in_line = arr_1, *longest = arr_2, *tmp;
    int max_len = 0, len;
    while(gets(in_line) != NULL)
    {
        len = strlen(in_line);
        if(len > max_len)
        {
            max_len = 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
{
    int len, max; // 当前行, 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;
}
```

/*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。

7.3.5 常量指针和指针常量

关键字 `const` 可以用来修饰指针变量，根据它的位置有三种修饰方法：

1. `const <类型> *<变量名>;`
2. `<类型> *const <变量名>;`
3. `const <类型> *const <变量名>;`

1. 常量指针 `const <类型> *<变量名>;`

常量指针（指向常量数据的指针），意思是该指针指向的内容只读（不可更改），但指针本身的值，即指针指向可以更改，例如下面的代码第 4 行会导致编译错误。

```
char str1[] = "abc";  
char str2[] = "ABC";  
const char *pc = str1; // pc 指向字符串 str1 的第一个字符  
*pc = 'A'; // 会产生编译错误：常量指针指向的内容不可更改  
pc = str2; // 指针本身的值可以更改
```

7.3.5 常量指针和指针常量

将一个 const 常量指针赋值给一个同类型的非 const 指针时，编译器会给出一个警告，因为有可能通过非 const 指针修改它指向的变量。例如：

```
char str[] = "abc";  
const char *pc = str; // pc 指向字符串 str 的第一个字符  
char *pc2 = pc; // 编译警告，常量指针赋值给了非常量指针  
*pc2 = 'A'; // pc2 是普通指针，可以修改指向的变量  
printf("%s\n", str); // 输出 str 的内容： Abc
```

将常量指针 pc 赋值给了非常量指针 pc2，引起编译警告，因为后续可能会通过 pc2 更改它指向的内容，如第 4 行。虽然 pc2 和 pc 指向同样的内容，但由于 pc 是常量指针，因此不能通过 pc 修改它指向的内容（编译错误）。

7.3.5 常量指针和指针常量

2. 指针常量 <类型> *const <变量名>;

指针常量，意思是指针本身是个常量（右值，只读变量），它的值在赋值后不能更改，即不能更改指向，例如下面的代码第 3 行会导致编译错误。

```
char str1[] = "abc";  
char *const p = str1;  
p = "hello"; // 编译错误：不能给只读变量 p 赋值
```

3. 常量 '指针常量' const <类型> *const <变量名>;

常量 '指针常量'，它的值（指向位置）不能被修改，它指向的内容也不能被修改。例如下面的代码第 3~4 行会导致编译错误。

```
char str1[] = "abc";  
const char *const p = str1;  
p = "hello"; // 编译错误：不能给只读变量 p 赋值  
p[0] = 'A'; // 编译错误：常量指针指向的内容不可更改
```

7.3.6 char*型指针与字符串

- 字符串是以 '\0' 结束的字符数组。例如 `char s[100] = "hello"` 定义了一个字符串，数组 `s` 的内容为字符串 "hello"（实际的存储内容最后还有一个 '\0'）。数组名 `s` 作为表达式的值（例如出现在赋值符的右侧）是该字符串首字符的地址，可以通过隐式类型转换为 `char*` 类型。
- 若有一个 `char*` 型指针 `p` 指向一个合法字符串的首字符，则通过 `p` 可以遍历访问这个字符串的每个字符 ('\0' 为结束标志)，即 `p` 完全能代表这个字符串。在没有上下文歧义的情况下，可以简称指向合法字符串的 `char*` 型指针为字符串。例如定义 `char *p = s`，则 `p` 也是字符串，而且和 `s` 是同一个字符串（指向同一个字符数组）。

```
char *p = NULL, s[20] = "point to me";  
p = "point to me"; // p 指向字符串常量 "point to me"  
p = s; // p 指向字符串 s 的内容  
s = "point to you"; // 编译错误, s 是数组名, 是左值, 不能被赋值  
p = "point to you"; // p 指向另一个字符串常量 "point to you"
```

7.3.6 char*型指针与字符串

- 用双引号引起来的字符串常量保存在程序内存中的常量区，它们的内容不能更改。字符串常量作为表达式的值是其首字符的地址，具有 `const char*` 类型，可以将它赋值给字符型指针变量。当通过指针试图更改字符串常量的内容时，程序就会崩溃。例如：

```
char *p = "Hello" ; // p 指向字符串“Hello”  
*p = 'A' ; // 编译没有错误，但发生运行时错误
```

- 字符串既可以用 `char*` 指针表示，也可以用 `char` 型数组表示。区别在于前者只是一个指针，指向的内容保存在别的地方中（例如其他某个 `char` 型数组，某一段常量区的字符串）；后者是一个数组，拥有字符串内容的数据实体空间。例如，执行这两条语句：

```
char *s; scanf("%s", s);
```

会导致运行时错误，因为指针 `s` 指向的空间并没有被分配（野指针）。正确的做法应该是

```
char s[100]; scanf("%s", s);
```

即，定义一个 `char` 型数组来确保字符串的存储空间。

7.3.6 char*型指针与字符串

- 对于字符串常量，本质是一个 '\0' 结尾的字符数组，存放在程序的静态存储区，字符串常量作为表达式的值是该字符数组的首地址，可隐式转换为 `const char *` 类型。
- 同样内容的字符串常量不管在代码中出现多少次，都是同一个对象（内存）
- `printf("a constant character string\n");` 传递给函数的是字符串常量的首地址。

注意：char数组和char*型指针使用时容易混淆！（见下例）

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

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

word = "you can't do this"; ✗
// 错误，word是数组名，不能赋值，正确做法为：
strcpy(word, "...");
```

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

a[0] = 'c'; // ✓

*p = 'c'; ✗
// 错误，不能修改字符串常量内容
```

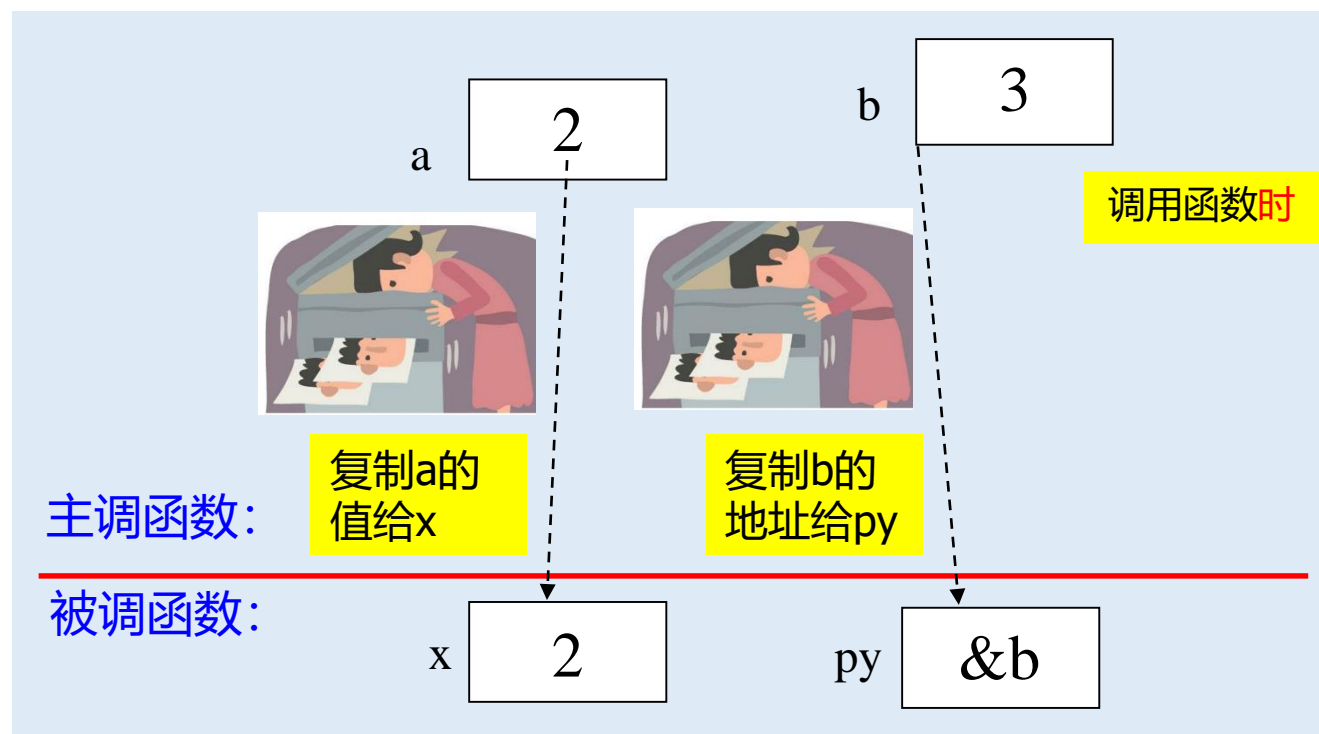
7.3.7 指针作为函数参数

- 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;  
}
```

实参传递的是
变量的地址

形参定义为指针



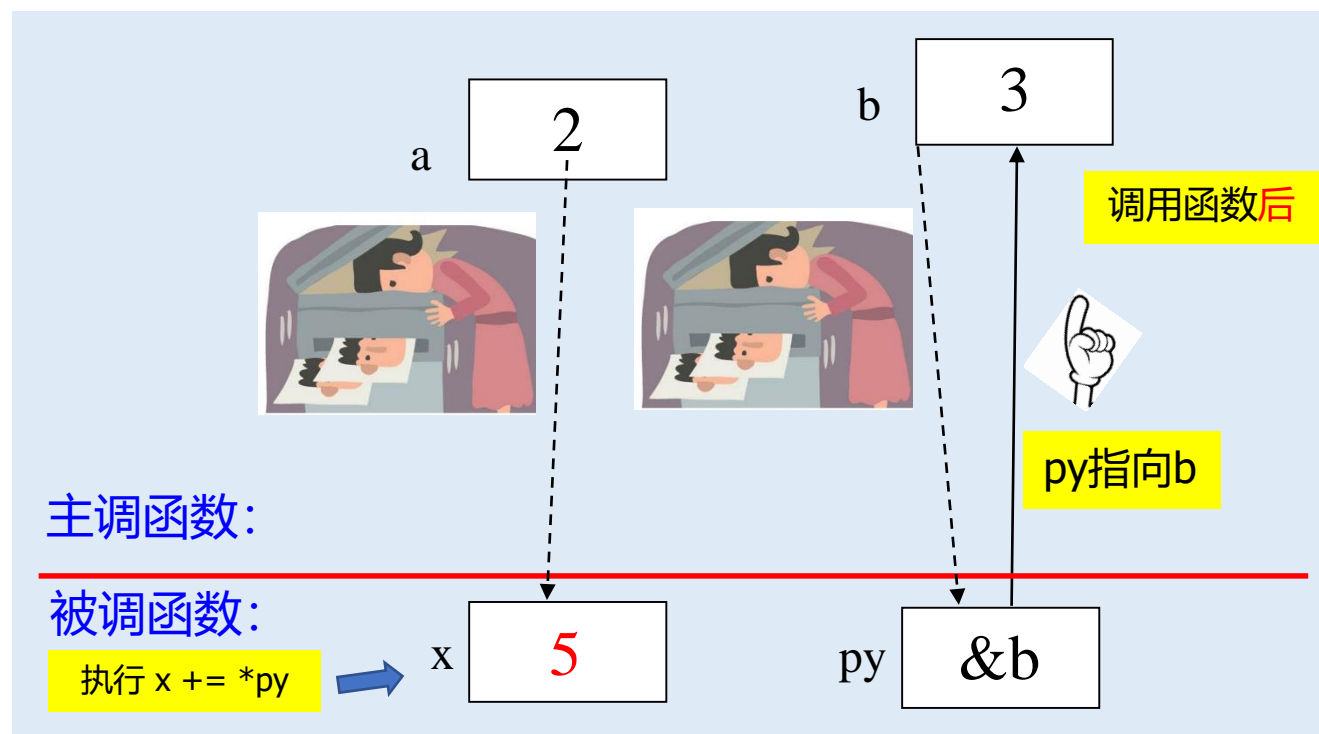
7.3.7 指针作为函数参数

- 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;  
}
```

实参传递的是
变量的地址

形参定义为指针



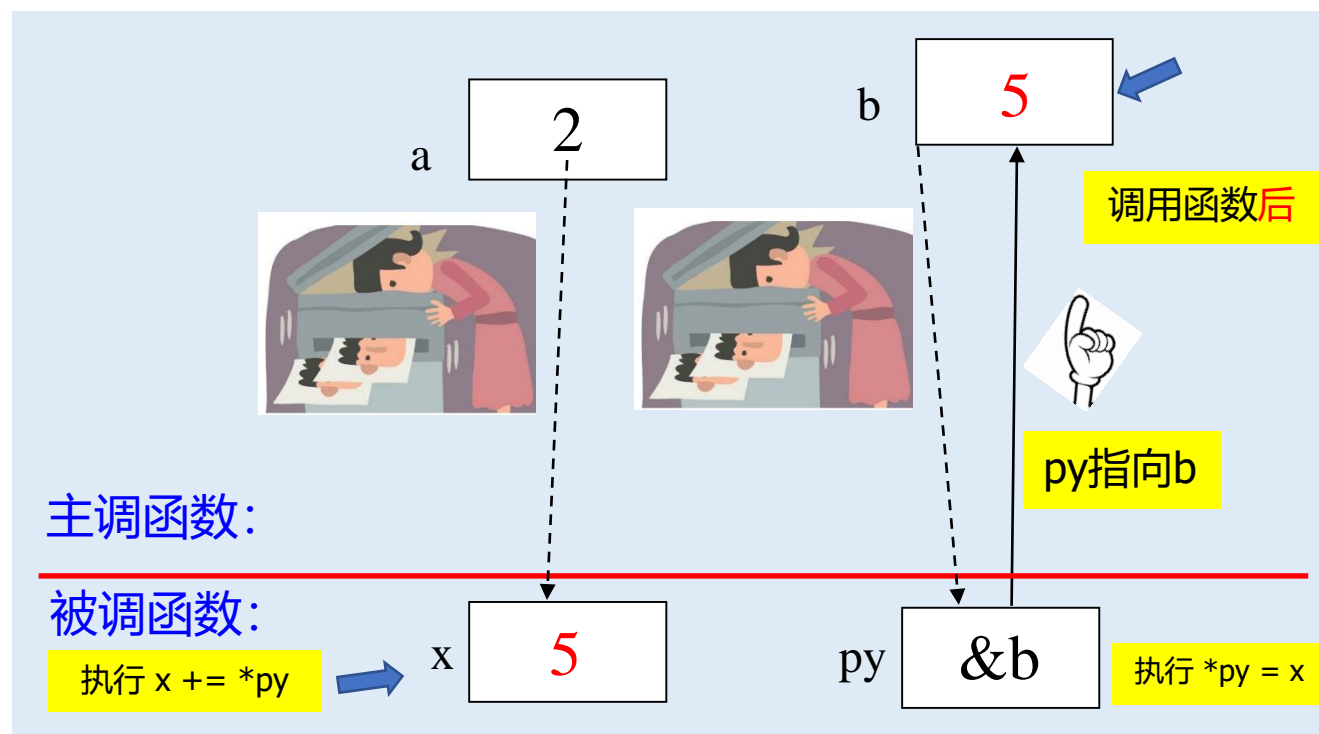
7.3.7 指针作为函数参数

- 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;  
}
```

实参传递的是
变量的地址

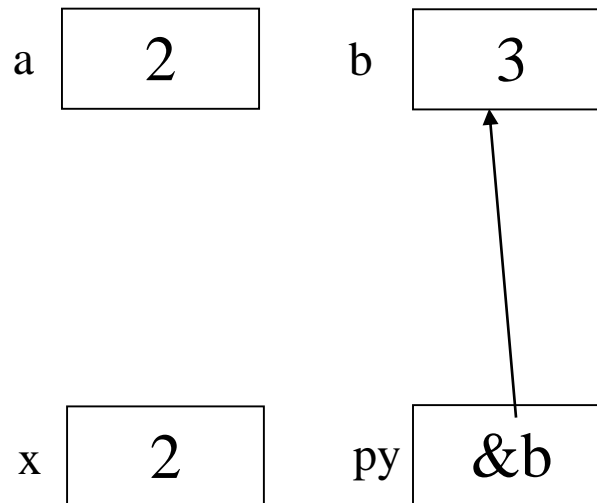
形参定义为指针



7.3.7 指针作为函数参数

- 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;  
}
```



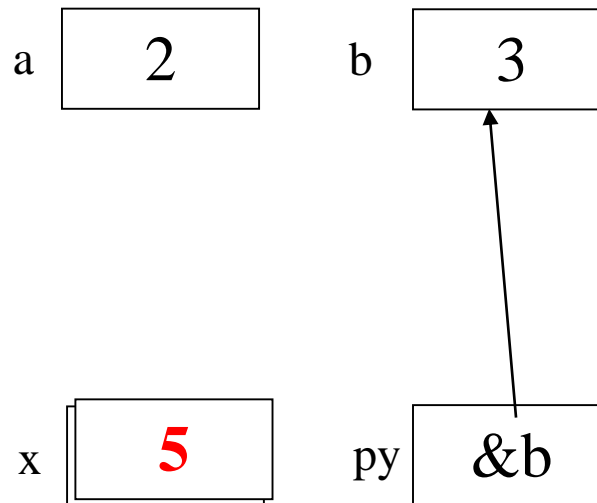
7.3.7 指针作为函数参数

- 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?



7.3.7 指针作为函数参数

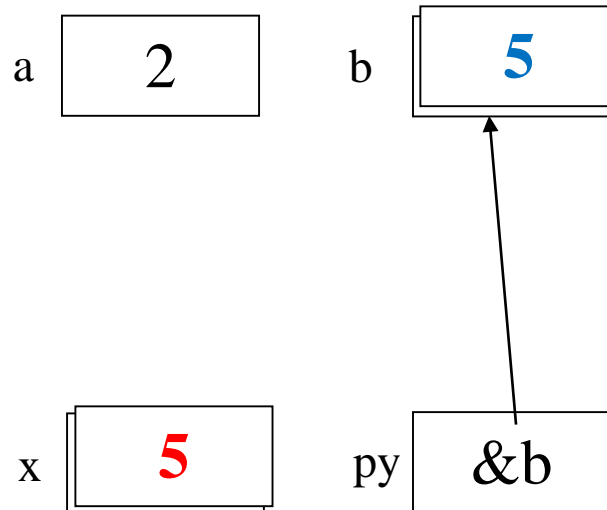
- C语言中的函数参数是值传递的，函数调用不会改变实参变量的值，一般通过返回值或者修改全局变量的方式实现。
- 定义指针作为函数参数，可以以指针的方式改变函数外部的变量值。
- 当一个函数需要同时向外部传递多个数值而又不希望改变全局变量时，可使用指针类型的函数参数。

```
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 ?



7.3.7 指针作为函数参数

- 形式参数和实际参数是独立的数据实体，两者不管名字是否一样，都没有任何关系。唯一发生联系的地方，是在函数调用时，发生了实际参数向形式参数的值传递。
- 如果真的有修改实际参数的需求时怎么办？例如希望设计一个 swap 函数，函数调用后互换两个实际参数的值。此功能可以用如下程序实现：

```
#include <stdio.h>
void swap(int *pa, int *pb)
{
    int tem = *pa;
    *pa = *pb;
    *pb = tem;
}
```

```
int main()
{
    int a = 3, b = 4;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```


7.3.7 指针作为函数参数

```
#include <stdio.h>
void swap(int *, int *);

int main()
{
    int a = 3, b = 4;
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}

void swap(int *pa, int *pb)
{
    int tem = *pa;
    *pa = *pb;
    *pb = tem;
}
```

输出结果：a = 4, b = 3

将实参的地址传递（复制）给形参，形参和实参指向同一个内存空间，通过对形参的操作实现对实参的修改。

```
#include <stdio.h>
void swap_fail(int, int);

int main()
{
    int a = 3, b = 4;
    swap_fail(a, b);
    printf("a = %d, b = %d\n", a, b);
    return 0;
}

void swap_fail(int x, int y)
{
    int temp = 0;
    temp = x;
    x = y;
    y = temp;
}
```

输出结果：a = 3, b = 4

形参和实参分别占用不同的内存单元，它们相互独立，无法直接实现数据同步

7.3.7 指针作为函数参数

从标准输入上读入多个学生成绩（至少读入 1 个，至多不超过 1000 个，成绩是 0~100 的实数），写一个函数 gradeAnalyse，功能包括：接收读入的成绩，计算平均成绩和标准差。输出保留到小数点两位。

问题分析：因为函数需要返回两个统计量，所以依靠程序的返回值是不行的，可以引入两个指针参数，函数计算的两个统计量写入到参数指针指向的变量中。另外，函数要接收读入的成绩，成绩个数可能比较多，把所有成绩都以值传递方式传入函数也不适合，用一个指针参数指向数组元素即可。具体程序见下页。

7.3.7 指针作为函数参数

```
#include <stdio.h>
#include <math.h>
#define ARRAY_SIZE 1000
void gradeAnalyse(const float *, int, float *, float *);
int main()
{
    int n = 0;
    float data[ARRAY_SIZE], average, std;
    while (scanf("%f", &data[n]) > 0)
        n++;
    gradeAnalyse(data, n, &average, &std);
    printf("%.2f, %.2f\n", average, std);
    return 0;
}
```

```
void gradeAnalyse(const float *pData, int n,
float *pAver, float *pStd)
{
    int i;
    float d, sum = 0.0f, std = 0.0f;
    for (i = 0; i < n; i++)
        sum += pData[i];
    *pAver = sum / n;
    for (i = 0; i < n; i++)
        std += pow(pData[i] - *pAver, 2);
    *pStd = sqrt(std / n);
}
```

7.3.8 数组名和指针作为函数参数的关系

当需要在函数内部访问外部的数组时，可以声明函数参数为数组，在函数调用时传递数组名给函数，就可以在函数体内访问该数组的所有元素。例如下面这个程序：

```
#include <stdio.h>
void fill_array(int array[10], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        array[i] = i;
}
void print_array(int array[10], int n)
{
    int i;
    for (i = 0; i < n; ++i)
        printf(i ? " %d" : "%d", array[i]);
    printf("\n");
}
```

```
int main()
{
    int a[10];
    fill_array(a, 10);
    print_array(a, 10);
    return 0;
}
```

输出：

```
0 1 2 3 4 5 6 7 8 9
```

7.3.8 数组名和指针作为函数参数的关系

- 在函数原型: `void fill_array(int array[10], int n)`和`void print_array(int array[10], int n)`中, 都定义了一个 `int` 型数组 `array` 作为形式参数。
- 在 C 语言中, 数组为函数的形式参数, 等价于指向数组元素类型的指针为函数的形式参数。即上述两个函数原型分别等价于 `void fill_array(int *array, int n)`和 `void print_array(int *array, int n)`
- 因为 C 语言编译器不检查形式参数中数组的长度 (忽略它), 因此以下三种函数声明均等价:

```
void fill_array(int array[10], int n);  
void fill_array(int array[], int n);  
void fill_array(int *array, int n); // 用这种形式更常见
```

- 当函数调用时, 将数组名作为实际参数传递给形式参数 `array`, 因为数组名作为表达式的值是数组第一个元素的地址, 这样进入函数体后, 形式参数 `array` 就指向了实际数组的第一个元素, 可以将它视为原数组进行下标访问和修改。

7.3.8 数组名和指针作为函数参数的关系

- 函数形式参数 array 和实际参数数组是两个不同的对象。在 fill_array 函数内部是不能通过 sizeof(array) 来获取实际参数数组长度的，因为形式参数 array 本质上是一个指针，所以 sizeof(array) 返回的是指针类型的长度（4 字节或 8 字节）。这就需要第二个参数 n 来传递实际的数组长度。

```
#include <stdio.h>
void dummy_array(int array[]);
int main()
{
    int array[10] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    printf("%d\n", sizeof(array));
    dummy_array(array);
    return 0;
}
void dummy_array(int array[10])
{
    printf("%d\n", sizeof(array));
}
```

想一想左边代码运行结果？
第二行输出 10？ 40？ 4？ 8？

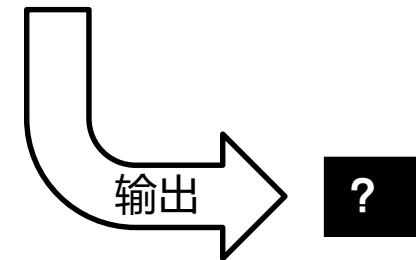
7.3.9 指针作为函数返回值

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		62fe40	..41	..42	..43	..44	..45	..46	..47	..48	..49	..4a	..4b	..4c	...	
	...	t	h	i	s		i	s		t	e	s	t	\0	...	

7.3.9 指针作为函数返回值

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;  
}
```

strchr()函数
怎么实现？
请读者思考。

7.3.9 指针作为函数返回值

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;
}
```

输入:
12345abcde[enter]

输出:
11
?

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	...
---	---	---	---	---	---	---	---	---	---	----	----	-----

7.3.9 指针作为函数返回值

从标准输入上读入一行字符串 buf 和一个字符串 str，把 buf 中出现的所有 str 子串全部替换为"I love c"并输出。保证每行字符数不超过 1000。

问题分析：

- 核心操作还是需要在 buf 中定位 str 的位置。可以使用标准库函数：

```
char *strstr(const char *_Str, const char *_SubStr);
```

strstr 函数在字符串 _Str 中寻找子串 _SubStr，如果找到则返回在 _Str 中指向 _SubStr 子串的指针，否则返回空指针。

- 本题稍显复杂的是，需要替换所有的子串而不只是第一个找到的，因此替换完成后还需要继续往后查找，直到找不到为止。

7.3.9 指针作为函数返回值

为了输出原字符串中不用替换的字符，设计一个辅助函数：

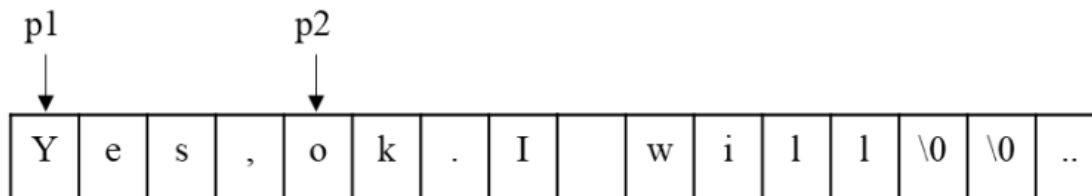
```
void print_str(const char *p1, const char *p2);
```

用于打印字符串中两个位置[p1, p2)之间的字符，注意，这是一个半闭半开区间，即包括 p1 但不包括 p2 所指向的位置。其代码如下：

```
void print_str(const char *p1, const char *p2)
{
    while (p1 < p2)
    {
        putchar(*p1);
        p1++;
    }
}
```

判断 p1 指向的字符是否在 p2 指向的字符之前

p1 指向自己当前位置的下一个元素（字符）位置



7.3.9 指针作为函数返回值

有了辅助函数 `print_str`，可以设计出主程序如右所示：

读入一行字符串，包括行尾的换行符

读入一行不包括行尾换行符的字符串
作为待查找的子串

`p1` 跳过要替换的字符串，指向后面的位置

```
#include <stdio.h>
#include <string.h>
char buf[1005];
char str[1005];
void print_str(const char *p1, const char *p2);
int main()
{
    char *p1 = buf, *p2 = 0;
    fgets(buf, 1000, stdin);
    scanf("%[^\\n]", str);
    while (p2 = strstr(p1, str))
    {
        print_str(p1, p2);
        printf("I love c");
        p1 = p2 + strlen(str);
    }
    fputs(p1, stdout);
    return 0;
}
```

7.4 指针运算

- 合法的指针运算包括指针加减整数、指针自增、自减以及指针相减、指针比较、对指针类型进行强制类型转换等。这些指针运算的主要目的是为了在一段连续内存空间中（例如数组）操纵指针的位置从而进行高效的数据访问。
- 注意指针运算的结果一定要指向合法的内存空间（例如在数组合法空间范围内），否则会发生“指针越界”。访问越界后的指针指向的内存空间，轻则程序崩溃，重则发生逻辑错误，埋下重大隐患。

```
void print_str(const char *p1, const char *p2)
{
    while (p1 < p2)
    {
        putchar(*p1);
        p1++;
    }
}
```

代码回顾

7.4 指针运算

指针和整型量可以进行加减，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

7.4.1 指针加减整数

若 n 是一个整型量， p 是一个指针，则 $p + n$ 和 $p - n$ 是合法的指针运算表达式，表达式的类型是与 p 同类型指针，表达式的值由下式确定：

$$\text{value}(p + n) = \text{value}(p) + n * \text{sizeof}(T)$$

$$\text{value}(p - n) = \text{value}(p) - n * \text{sizeof}(T)$$

其中 $\text{value}(\text{<指针>})$ 表示 <指针> 的值（即所表示的实际地址）， T 代表指针指向的数据类型。

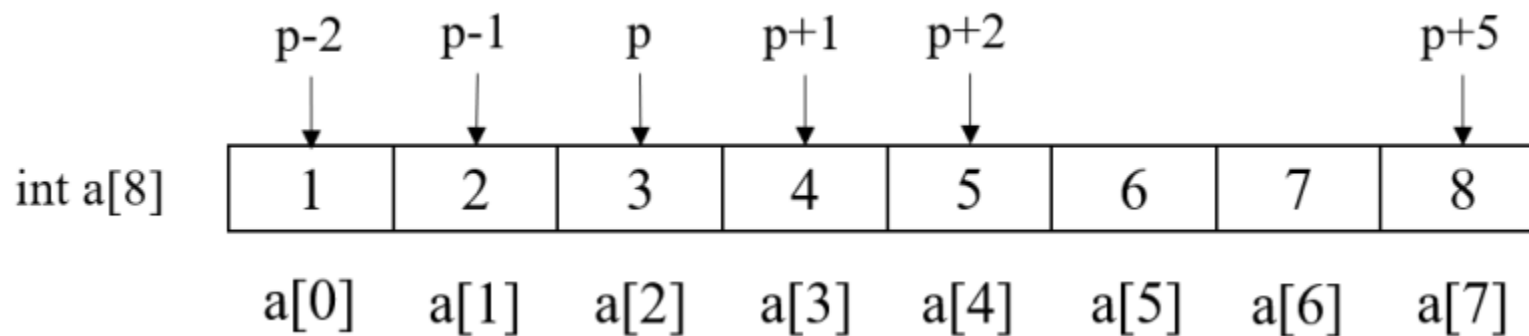
```
int a = 0;
int *p = &a;
printf(" p: %p\n", p);
printf("p+1: %p\n", p+1);
```

程序输出

```
p: 000000000061FE14
p+1: 000000000061FE18
```

7.4.1 指针加减整数

- 与指针相加的整数表示的是以其所指向的数据类型的长度为单位的偏移量，而不是指针数值（地址值）的偏移量。
- 如果指针指向的是数组元素，则与整数加减是以元素为单位移动指针。
- 指针与整数的加减运算使指针能够在内存空间上以它所指向的数据类型长度为单位进行移动，提高内存空间的访问效率。移动后的指针指向的内存空间是否合法或者有意义，需要编程者自身保证。



7.4.1 指针加减整数

指针既然可以与整数进行加减运算，那么作用于指针变量的++、--、+=、-=运算符也是合法的：

$p++$ 等价于 $p = p + 1$ ； $p--$ 等价于 $p = p - 1$ ； $p += n$ 等价于 $p = p + n$ ； $p -= n$ 等价于 $p = p - n$ 。

下面的例子展示了指针加减整数的运算：

```
int a[5] = {10, 11, 12, 13, 14};
int *pi = &a[1]; // pi 指向 a[1]
pi++; // pi 指向 a[2], *pi 为 a[2], 即 12
pi += 2; // pi 指向 a[4], *pi 为 a[4], 即 14
*(pi-2) = *(pi-4); // *(pi-4)为 a[0], *(pi-2)为 a[2], 相当于 a[2] = a[0];
const char *p = "BUAA" + 1; // p 指向常量字符串"UAA"
```

只要是合法的指针，用*号对它进行解引用就是访问它指向的变量。上述代码中， $*(pi-2) = *(pi-4)$ 的赋值语句，因为解引用运算符的优先级高于加减法，因此一定要括号来保证先进行指针加减再解引用。

7.4.1 指针加减整数

因为自增、自减运算符++、--与解引用运算符*的优先级相同，连用时按照从右向左的顺序结合。
例如*p++等价于*(p++)，由于自增运算符出现在指针右侧，意味着先对指针 p 进行解引用，再对 p 指针进行自增。如果要对 p 指向的变量值进行自增则需要写成(*p)++。

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



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）。

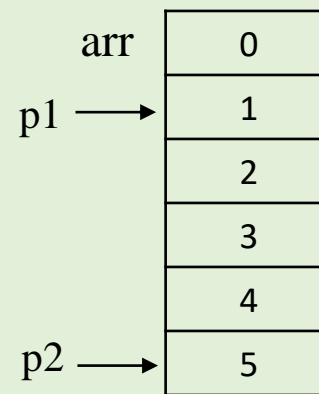
(*p)++ 表示p先与*结合，++作用于变量p所指向的变量（是变量值增加）。

7.4.1 指针加减整数

*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

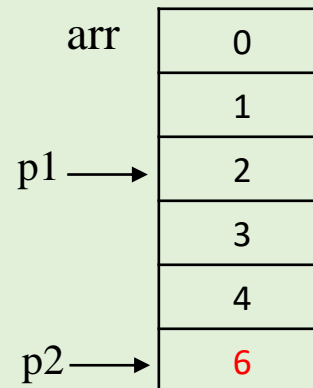
```
int a, b, arr[ ] = {0, 1, 2, 3, 4, 5};  
int *p1 = &arr[1], *p2 = &arr[5];
```



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



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



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

7.4.1 指针加减整数

自定义一个字符串查找函数，实现标准 C 函数 strstr 的功能。

问题分析：设计函数的接口和功能：char *_StrStr(const char *buf, const char *str)，从字符串 buf 中寻找字符串 str，如果找到，以指针形式返回 str 在 buf 中第一次出现的位置，否则返回空指针。为简化，不考虑输入的边界条件，保证 buf 和 str 不为空字符串。

```
#include <stdio.h>
#define N 1000
char *_StrStr(const char *, const char *);
int main()
{
    char buf[N + 5], str[N + 5];
    char *pbuf = buf, *pstr = str;
    gets(pbuf); // 保证输入不超过定义的字符数组宽度
    gets(pstr); // 保证输入不超过定义的字符数组宽度
    char *p = _StrStr(pbuf, pstr);
    p ? printf("%d\n", p - pbuf) : printf("no found\n");
    return 0;
}
```

7.4.1 指针加减整数

自定义一个字符串查找函数，实现标准 C 函数 strstr 的功能。

```
char *_StrStr(const char *buf, const char *str)
{
    const char *pbuf, *pstr;
    do
    {
        pbuf = buf++; // pbuf 指向文本开始位置
        pstr = str;    // pstr 指向子串的开始位置
        while (*pbuf == *pstr && *pbuf)
        {
            pbuf++;
            pstr++;
        }
        // while 循环判断源字符串中当前的子串是否与目标匹配
    } while (*pbuf && *pstr); // 直到遍历完
    return *pstr ? NULL : (char *)buf - 1;
}
```

经典的高效字符串匹配算法有 KMP (Knuth-Morris-Pratt) 算法、BM (Boyer-Moore) 算法等，留给感兴趣的读者做进一步深入研究。

7.4.2 指针与下标

- $*(p+n)$ 这种形式在指针操作中很常见，因此 C 语言编译器为它设计了一个等价写法： $p[n]$ 。是不是很熟悉？这就是访问数组元素的写法！把数组名看作数组首元素的地址（指针类型），下标 n 是以数组元素为单位的地址偏移量。
- 这也解释了为什么 C 语言数组下标从 0 开始： $p[0]$ 等价于 $*p$ ，即数组的第一个元素。如果指针 p 指向数组的中间元素，那么负下标如 $p[-1]$ 、 $p[-2]$ 也是合法的，它们分别等价于 $*(p-1)$ 和 $*(p-2)$ ，只要保证指针指向的位置没有超出数组范围即可。
- 若 p 是一个指针， n 是一个整型量，请牢记关于指针的下列两个等价写法：

$p[n]$ 等价于 $*(p+n)$

$\&p[n]$ 等价于 $p+n$

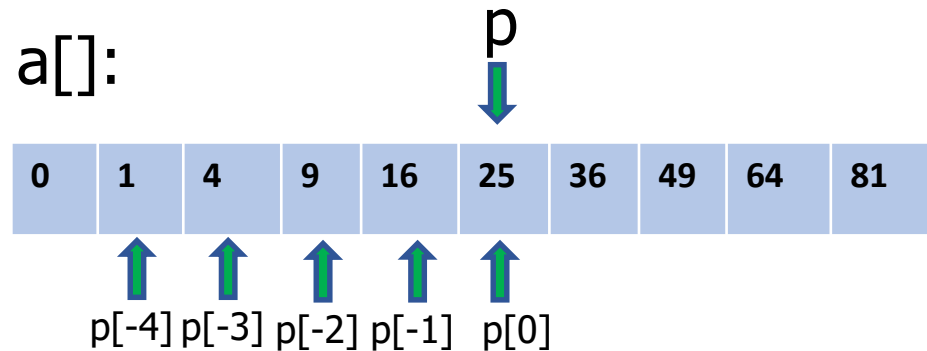
7.4.2 指针与下标

- 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]);
```

$p[n]$ 等价于 $*(p + n)$

$\&p[n]$ 等价于 $p + n$



输出:
25 16 9 4 1

7.4.2 指针与下标

- 数组名作为表达式，**可以看作指向数组第一个元素的指针**

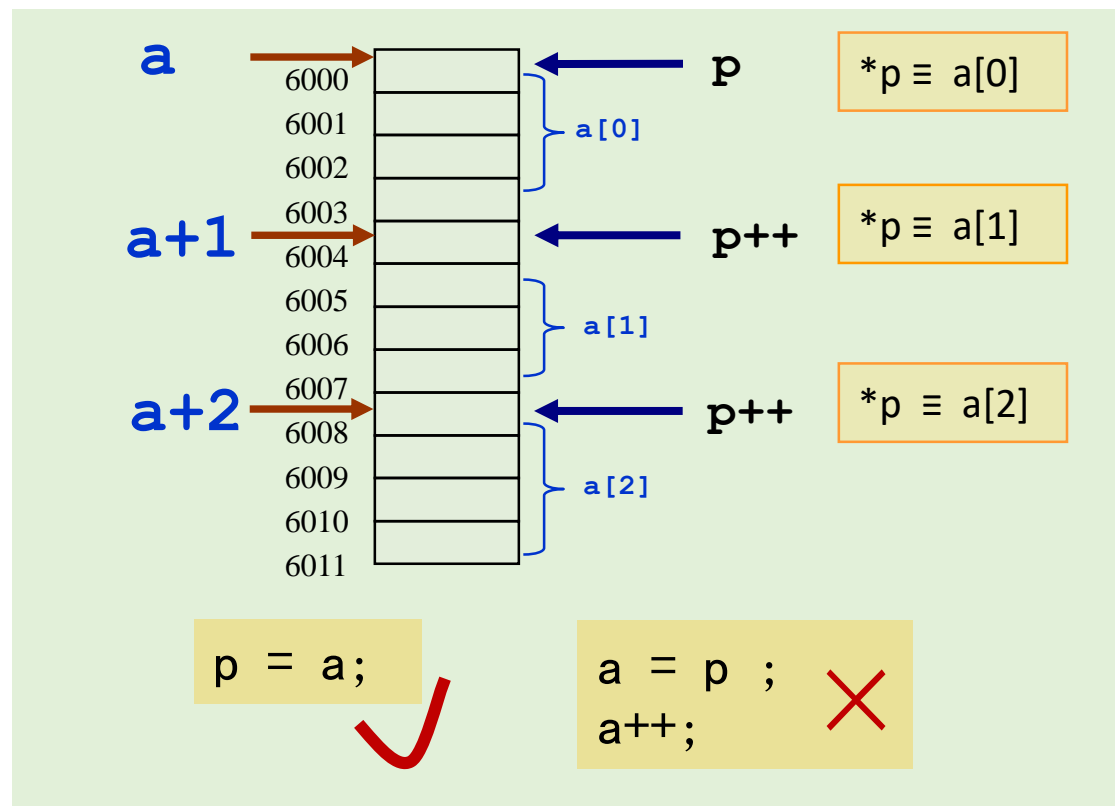
- ◆ 但数组名是常量，不能被赋值或修改其值

- 因此指针可赋值为数组名

- ◆ `int *p, a[10];`
- ◆ `p = a;` //p指向数组a的第一个元素

- 数组元素的几种等价引用形式

- ◆ `a[i]` 数组名+下标
- ◆ `*(a+i)` 数组名+偏移量，最后解引用
- ◆ `p[i]` 指针+下标
- ◆ `*(p+i)` 指针+偏移量，最后解引用



关于数组和指针关系的进一步阐述，详见下一章节

7.4.2 指针与下标

```
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));  
...
```

指针指向数组的首元素

指针下标访问数组元素

按偏移量解引用形式访问

数组可以按指针形式访问

关于数组和指针关系的进一步阐述，详见下一章节

7.4.2 指针与下标

数组下标访问元素

```
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]);
```

指针访问数组元素

```
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! 数组名是常量，不能改变它的值

关于数组和指针关系的进一步阐述，详见下一章节

7.4.3 指针相减

只有同类型并且指向同一数组内的元素的指针才可以相减，结果是一个有符号整数，表示两个指针指向元素之间的下标差。

```
#include <stdio.h>
int main()
{
    int a[10] = {0};
    int *p1 = &a[4]; // p1 = a + 4
    int *p2 = a;      // p2 = a;
    int *p3 = a + 9; // p3 = a + 9
    printf("p2 - p1 = %d\np3 - p1 = %d\n", p2 - p1, p3 - p1);
    return 0;
}
```

输出:

```
p2 - p1 = -4
p3 - p1 = 5
```

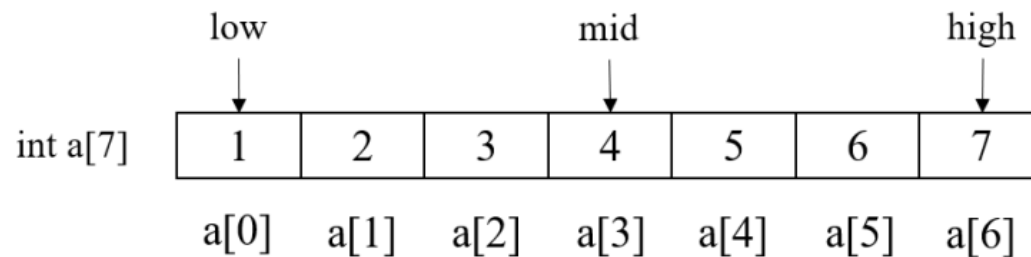
7.4.3 指针相减

- 指针相减的结果只取决于两指针间相隔元素的数量和方向，而与数组元素本身所占的字节数无关。如果指针 $p1 - p2$ 的结果为正，表示在数组中 $p2$ 指向的元素排在 $p1$ 指向元素的前面。
“前”与“后”分别对应低地址端和高地址端，数组名作为表达式的值是数组第一个元素的地址，对应数组空间的最低地址端。
- 只有当两个指针指向同一数组内容时，“前后”及间隔元素个数才有意义，否则两个指针相减没有太大的意义，唯一能得到的信息就是根据符号得到地址的前后关系。
- 指针相加没有任何意义，编译器禁止这种行为，会报编译错误。

$mid = (low + high) / 2;$

获取中间元素指针，哪个用法正确？

$mid = low + (high - low) / 2;$



$$high - low = 6 \quad mid = low + (high - low) / 2 = low + 3$$

7.4.4 其他指针运算

- 除了上述合法的指针运算外，其他关于指针的数学运算均为非法。包括指针乘法、指针与浮点数的加减、指针之间的相加、移位运算等，编译器均会报错，因为这些运算没有实际的物理意义。
- 指针可以参与逻辑运算，当指针为 NULL 指针时表示逻辑假，否则表示逻辑真。

```
char s[100], t[10];  
...  
char *p = strstr(s, t);  
if(p) //如果在 s 中找到 t  
...  
else  
...
```

if (p) 等价于 if (p != NULL)。

7.4.5 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
```

- ◆ 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，通常将其标记为空指针NULL。
- ◆ 空指针NULL在C的标准头文件中是一个值等于 0 的宏定义。
- ◆ 野指针：指针指向没有分配的空间，造成访问隐患，称为野指针，例如定义时未初始化。
- ◆ 指针经常和NULL使用==, !=进行比较，判断指针是否为空（无效）

7.4.5 指针的比较

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

```
fgets(buf, BUFSIZ, stdin) != NULL
```

while(还有新输入行)

while(又获得一个新数据) ← 用函数实现

求和;

输出当前行的行号、数量、平均值;

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

跳过空格

获取一个数据

指针移到数据后面的空格处

7.4.5 指针的比较

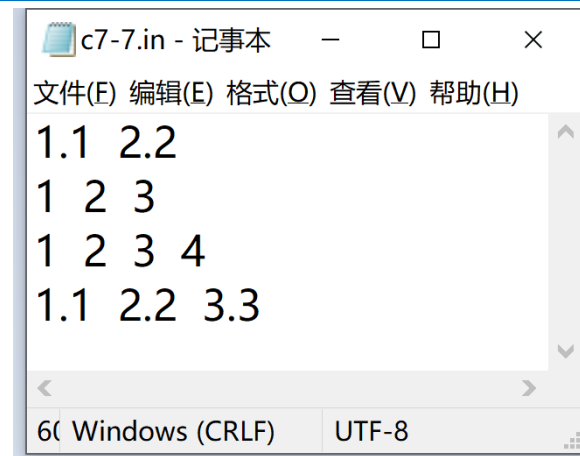
这是一个处理以“行”为单位的输入数据的基本框架，值得掌握！

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！

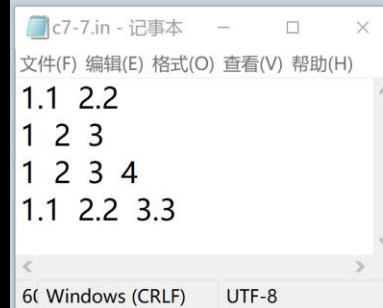
7.4.5 指针的比较

(改进版的程序)

```
int i, n=0;
double d, subsum;
char buf[BUFSIZ], *p;

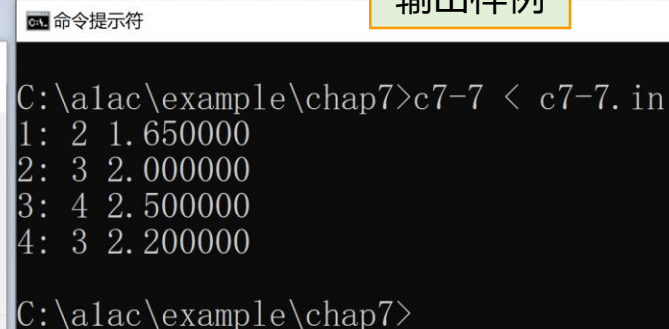
for(i=1; fgets(buf, BUFSIZ, stdin) != NULL; i++)
{
    subsum = 0, n=0;
    p = buf;
    while(p != NULL && sscanf(p, "%lf", &d) != NULL)
    {
        subsum += d;
        n++;
        if((p=strstr(p, " ")) != NULL) // 7-7比较啰嗦, 用strstr找空格更好
            p++; // 指针移到下一个读写数据的开始位置
    }
    if(n > 0)
        printf("%d:%d %f\n", i, n, subsum/n);
}
```

输入样例



```
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
```

输出样例



```
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>
```

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

这么写法更简练!

7.4.5 指针的比较

写一个函数 `void strRev(char *s)`, 对字符串 `s` 进行逆序操作。例如若 `s` 的内容是 "helloworld", 则函数执行后 `s` 的内容变为 "dlrowolleh"。

```
void strRev(char *s)
{
    char *plow = s, *phigh = s;
    while (*phigh) // while(*phigh!='\0')
        phigh++;
    phigh--; // phigh 指向 s 的最后一个字符
    while (plow < phigh)
        swap(plow++, phigh--);
}
```

```
void swap(char *plow, char *phigh)
{
    char tmp = *plow;
    *plow = *phigh;
    *phigh = tmp;
}
```

设置两个指针 `plow` 和 `phigh` 分别指向字符串 `str` 的首字符和最后一个字符, 然后互换 `plow` 和 `phigh` 指向的字符内容, 接着 `plow` 自加, `phigh` 自减, 直到 `plow >= phigh`。

7.4.5 指针的比较

写一个函数 `void strRev(char s[])`，对字符串 `s` 进行逆序操作。例如若 `s` 的内容是 "helloworld"，则函数执行后 `s` 的内容变为 "dlrow olleh"。

`plow`, `phigh` 确定后
用 `while` 实现逆序：

`while`(还有逆序字符)

交换数据

起始位置移动

终止位置移动

`plow < phigh`

`plow++`, `phigh--`;



`plow`

`phigh`

`plow < phigh`

7.4.5 指针的比较

写一个函数 `void strRev(char s[])`，对字符串 `s` 进行逆序操作。例如若 `s` 的内容是 "helloworld"，则函数执行后 `s` 的内容变为 "dlrow olleh"。

`plow`, `phigh` 确定后
用 `while` 实现逆序：

`plow < phigh` 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



`phigh` `plow`

`plow >= phigh`

`plow++`, `phigh--`;

7.4.5 指针的比较

```
void strRev(char *s)
{
    char *pLOW = s, *pHIGH = s;
    while (*pHIGH)
        pHIGH++;
    pHIGH--; // pHIGH 指向 s 的最后一个字符
    while (pLOW < pHIGH)
        swap(pLOW++, pHIGH--);
}
```

7.4.5 指针的比较

: 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;
}
```

为何不直接调用strRev?
而要重写rev函数?

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

7.4.6 指针的强制类型转换

- 指针都有类型，指针类型决定了如何对它所指向的内存空间内容进行解释和操作，决定了指针运算时如何映射到实际的地址。
- 不同类型指针之间不要互相赋值，但是任何情况都有例外，在编写一些底层相关的代码时，经常需要对同一内存空间的内容做出不同的解释，这种需求一般通过不同指针类型之间的赋值来完成。
- C 语言编译器做了折中处理，如果语法检查发现有不同类型指针之间的赋值，编译器会给出一个警告，提示这里要小心，如果编程者确信自己就是要这么做，可以通过强制类型转换的方式，避免编译器的警告。

```
int a = 0x00112233;  
char b[] = "hello";  
char *pa = (char *)&a;  
int *pb = (int *)b;
```

- 由于加上了强制类型转换，编译器认为编程者已经清楚了自己的行为，故不再给出警告。
- 指针的强制类型转换（隐式类型转换同样）不会更改指针的值，只是改变了对指针指向内容的解释

7.4.6 指针的强制类型转换

```
int b = 0x12345678;  
char *p = (char *)&b;  
*p++ = 0xAB;  
*p = 0xCD;  
printf("%X\n", b);
```

读者可能会发现，计算机存储数据的顺序跟人们阅读的顺序相反？下例分析具体原因。

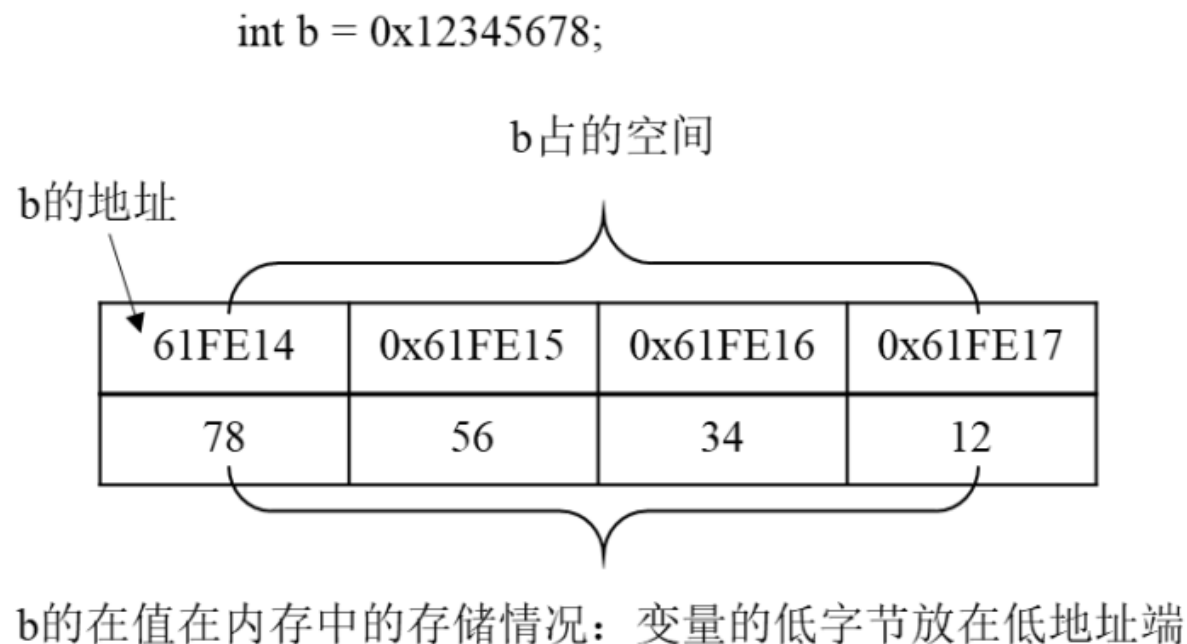
- 第 1 行，定义了一个 int 型变量 b，初始化其内容为：0x12345678。
- 第 2 行，定义了一个 char* 型指针 p，通过强制类型转换使其指向了变量 b。
- 第 3 行，对指针 p 进行解引用，访问 p 指向的内存空间。因为 p 是 char* 类型，所以编译器将 p 指向的内存空间解释为 1 个字节的 char 型变量，将它的内容改写为 0xAB。由于 p 指向的内存空间也是变量 b 的第一个字节，因此 b 的第一个字节被改写为 0xAB。然后把 p 加 1，指向内存中的下一个字节。
- 第 4 行，b 的第二个字节被改写为 0xCD。
- 第 5 行，以 16 进制输出 b 的内容，输出结果 1234CDAB。

7.4.6 指针的强制类型转换

将一个 int 型数据由小端字节序转换为大端字节序。

问题分析：C 语言在 Intel 架构的 CPU 下编译出来的指令为小端存储字节序，即变量的低字节放到低地址端，如例 7-13 中，`int b = 0x12345678`，则变量 b 的值在内存的存放情况（假设 b 的地址为 0x61FE4）见下图所示。

TCP/IP 网络通信协议采用大端存储字节序，即接收到的多字节流中低地址端存放的是高字节的数据（跟人们从左向右阅读的习惯一致）。因此通过 C 语言编程向网络传输数据时（例如通过 TCP/IP 协议）需要进行字节序转换。



7.4.6 指针的强制类型转换

将一个 int 型数据由小端字节序转换为大端字节序。

本例中，需要将 int 型数据的内存布局做调整。可以设置两个单字节类型指针 pL 和 pH（例如 char*），分别指向 int 型数据的低字节和高字节，互换内容后 pL++，pH--直到 pL >= pH。据此写出代码如下：

```
int a = 0x12345678;
char tmp;
char *pL = (char *)&a;
char *pH = (char *)&a + sizeof(int) - 1;
while(pL < pH)
    swap(pL++, pH--);
printf("%#x", a);
```

定义一个 int 型变量 a，后续对它进行字节序调整

通过高低字节互换内容，完成字节序的调整。
swap函数见例 7-12

输出：

0x78563412

7.5 void* 指针与 memcpy 函数

- C 语言中还有一个通用的指针类型 `void*`，用于指向原始的内存单元 (raw memory)。任何具体类型的指针都可以隐式类型转换为 `void*` 指针而不会有编译器警告，反之不行。
- `void*` 指针不对其所指向的内容做任何高层次的解释和假设：“只知道自己是内存地址，至于里面放的数据如何解释不归我管”。因此，对 `void*` 指针解引用也就没有意义，不要这么做！
- `void*` 指针与整数的加减代表着实际地址的加减，表示以字节为单位的指针移动；`void*` 指针之间的减法返回一个有符号整数，表示它们实际地址之差。(C 标准不支持，但有些扩展的编译器可能可以)
- 如果编程者知道 `void*` 指针指向的具体内容，或者编程者即将使用它指向的空间为自己所用，那么可以通过强制类型转换将它赋值给一个具体类型的指针，方便后续操作

```
int a = 123; // int* 是猫科动物， &a 是猫
void *pv = &a; // int* 赋值给 void*，没有问题，因为猫是动物 (pv 是动物)
int *pi = (int *)pv; // 通过强制类型转换恢复猫的身份
printf("%d\n", *pi);
```

7.5 void* 指针与 memcpy 函数

- 试想如果需要写一个函数，将一个内存空间的数据复制到另一个内存空间，如何设计这个函数的原型？用 int* 还是 double* 指向源和目标内存空间？
- 其实内存空间内容的复制是不需要类型信息的，只要告诉函数从哪个地址开始，复制多少字节的数据即可，根本和这些内容如何解释没有关系。
- 此时 void* 就派上用场了，它只是一个原始指针（raw pointer），其他所有类型的指针做实参时都可以隐式类型转换为 void * 类型而无需特殊处理，有效简化了函数的接口设计

```
void _MemCpy(void *dest, const void *src, unsigned long long n)
{
    unsigned long long i;
    for (i = 0; i < n; i++)
        *(char *)dest++ = *(char *)src++; /*(char *)dest = *(char *)src, dest++, src++;
}

```

7.5 void* 指针与 memcpy 函数

```
#include <stdio.h>
void _MemCpy(void *, const void *, unsigned long long);
int main()
{
    int a[] = {1, 2, 3, 4, 5};
    int i, b[10] = {0};
    _MemCpy(b, a, sizeof(a));
    for (i = 0; i < 10; i++)
    {
        printf("%d", b[i]);
        printf((i + 1) % 5 == 0 ? "\n" : " ");
    }
    return 0;
}
```

输出:

```
1 2 3 4 5
0 0 0 0 0
```

C 语言标准库提供了一个函数来完成内存之间的复制，其函数原型如下。功能是从 _Src 指向的内存空间拷贝 _Size 字节的内容到 _Dst 指向的空间，并将 _Dst 指针值返回。

```
void *memcpy(void * _Dst, const void * _Src, size_t _Size);
```

7.6 malloc函数与堆空间

- C 程序在运行时所占用的内存空间主要分为三部分：**静态存储区**、**栈空间**以及**堆空间**。静态存储区主要存放程序的代码段、字符串常量以及全局变量和静态变量。
- 程序中的**局部变量存放在栈空间**（函数调用栈），随着函数的调用和返回，栈空间不断增长和减小。一旦函数返回，对应的栈空间就会被释放和再利用。
- 堆空间也是动态变化的，并且它的大小几乎不受任何限制（当然受计算机内存大小的限制）。在堆空间上申请一段内存空间为程序所用，需要调用 C 语言标准库函数：

```
void *malloc(size_t _Size);
```

- malloc 函数接收一个参数 _Size，表示需要申请的内存空间的字节数。如果内存申请成功则返回指向这片可用内存空间（一共有 _Size 个字节）的指针（首地址）；否则返回一个 NULL 指针表示分配空间失败。由于 malloc 不需要知道这片内存空间用来存放何种数据，因此它返回一个 void *型指针。

7.6 malloc函数与堆空间

- 堆空间一旦申请成功，它将一直存在，直到程序结束。如果用 malloc 申请的堆空间不再使用的话，尽早归还它（释放空间），因为内存是稀缺资源，不断申请而不释放的话会导致计算机的可用内存不断减少，引发内存泄露（memory leak）。释放 malloc 申请的堆空间用标准库中的 free 函数：

```
void free(void *_Memory);
```

- 调用时，使用 malloc 函数返回的指针作为它的实际参数。注意，free 函数只能释放由 malloc 函数申请的堆空间，千万不能传给它局部或全局变量空间的地址。另外，一旦用 free 释放了指针指向的堆空间，之后就不能再通过指针访问它了。好的习惯是释放后给指针赋值 NULL，表明它现在是一个无效指针。

7.6 malloc函数与堆空间

```
double *d = (double *)malloc(1000000 * sizeof(double));  
d[0] = 1;  
// ... 数组相关的数据处理  
free(d); // 释放申请的堆空间  
d = NULL; // 指针置空, 避免后面的错误访问
```

- 第 1 行, malloc 函数的参数是指内存空间的字节数, 因此需要乘以 sizeof(数据类型)得到所需大小的内存空间 (存储 1000000 个 double 类型元素)。这里的目的是利用 malloc 申请的堆空间作为 double 型数组使用, 因此通过强制类型转换将 malloc 返回的内存空间首地址赋值给 double*指针 d, 方便后续将 d 视为一个长度为 1000000 的 double 数组直接处理。
- 第 4 行, 释放指针 d 指向的堆空间, 释放后这段空间便无法使用。需要注意的是指针 d 的值不受影响, 只不过它指向的内容已经不合法了 (归还给系统了)。
- 第 5 行, 将指针 d 赋值为 NULL, 表明它是一个无效指针, 防止后续误用 (否则是一个野指针)

7.6 malloc函数与堆空间

```
#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

12345

7.6 malloc函数与堆空间

```
#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

5, 4(或8)

abcde

7.7 本章小结

- 本章讲述了有关指针的基本概念和语法、指针运算和表达式、应用场景与注意事项，以及指针使用规则背后的深层原因与哲学思想。
- 本章中反复强调了类型这一概念：C 语言中所有的数字、常量、变量、表达式等，都是有类型的，类型对应着内存的布局 and 解释。知道了这一点，就能很好理解编译器的行为以及强制类型转换的意义，正所谓“心中有类型，指针如有神”。
- 指针和其他一般变量没有本质区别，只是存放的数据表示内存空间的地址而已。通过指针，可以高效地访问和修改内存空间的内容，这也是 C 语言执行效率很高、比较适合硬件编程和开发底层驱动程序的原因所在。