C 基础班讲义

朱景尧

目　录

1　第一个 c 语言的 hello world

1.1　include　头文件包含

头文件包含，写法　#include <　文件名 >,

1.2　main　函数

这个就是　C 语言程序的入口，所有的　　C 程序都是从　main 开始执行，一个　C 的源程序必须有一个 main 函数，也只能有一个　　main 函数

1.3　注释

// 注释一行

/* */　代表块注释，可以注释多行代码

1.4　{}括号和代码块

代表一个代码单元

1.5　声明

C 语言规定，所有的变量和函数必须先声明，然后才能使用　　.

1.6　C 语言自定义名字的要求

可以使用大小写字母，下划线，数字，但第一个字母必须是字母或者下划线

字母区分大小写

变量名最好用英文，而且要有所含义，通过变量的名称就能猜测变量的意思。

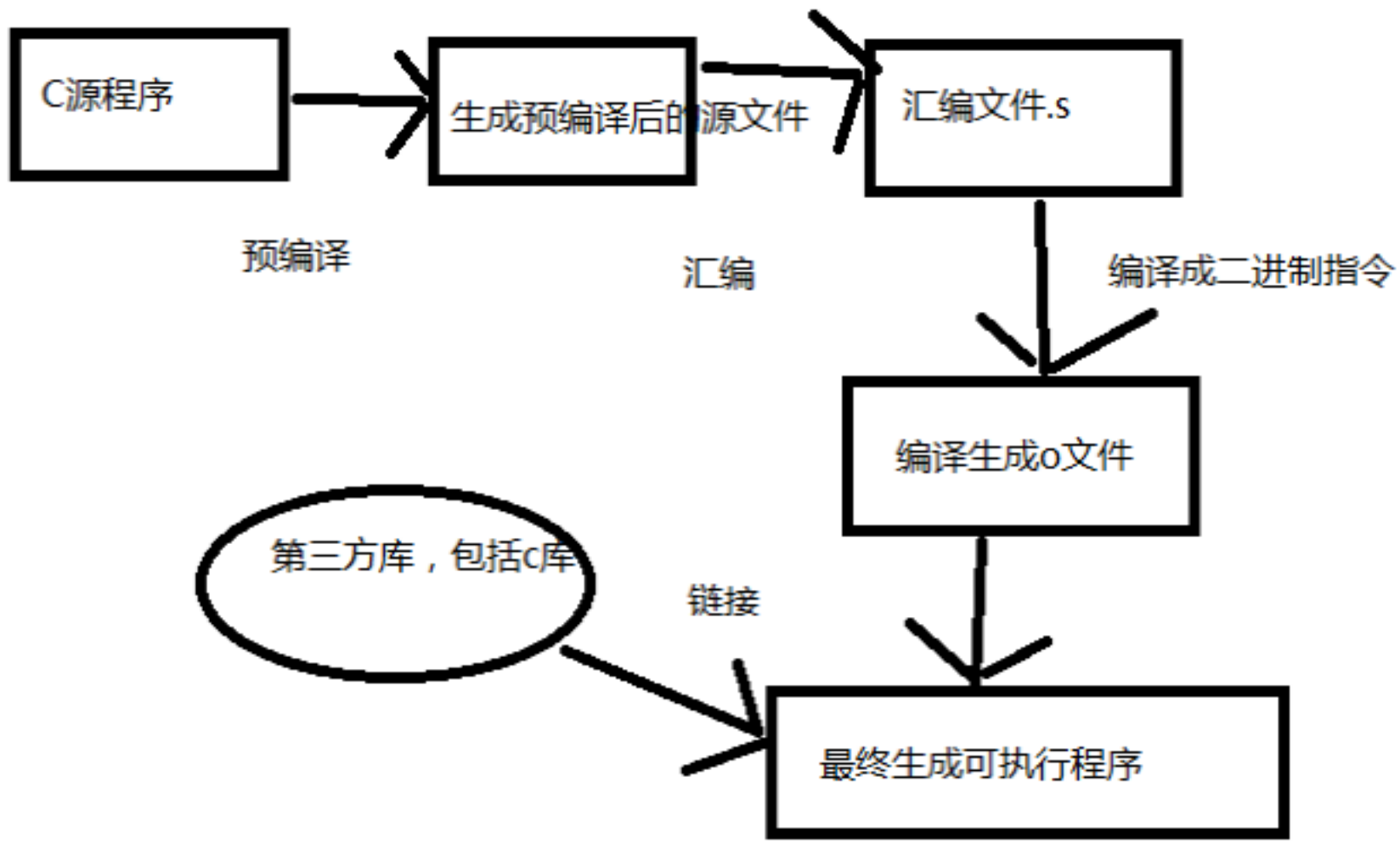
1.7 return 语句

在 C 语言当中任何函数遇到 return 代表这个函数停止，当 main 函数遇到 return ，代表整个程序退出

return 代表函数的返回值，如果返回类型是 void ，可以直接写 return，而不需要返回任何值

2 C 语言的编译

2.1 编译过程



2.2 gcc 编译选项

-o 代表指定输出文件名

-E 代表预编译

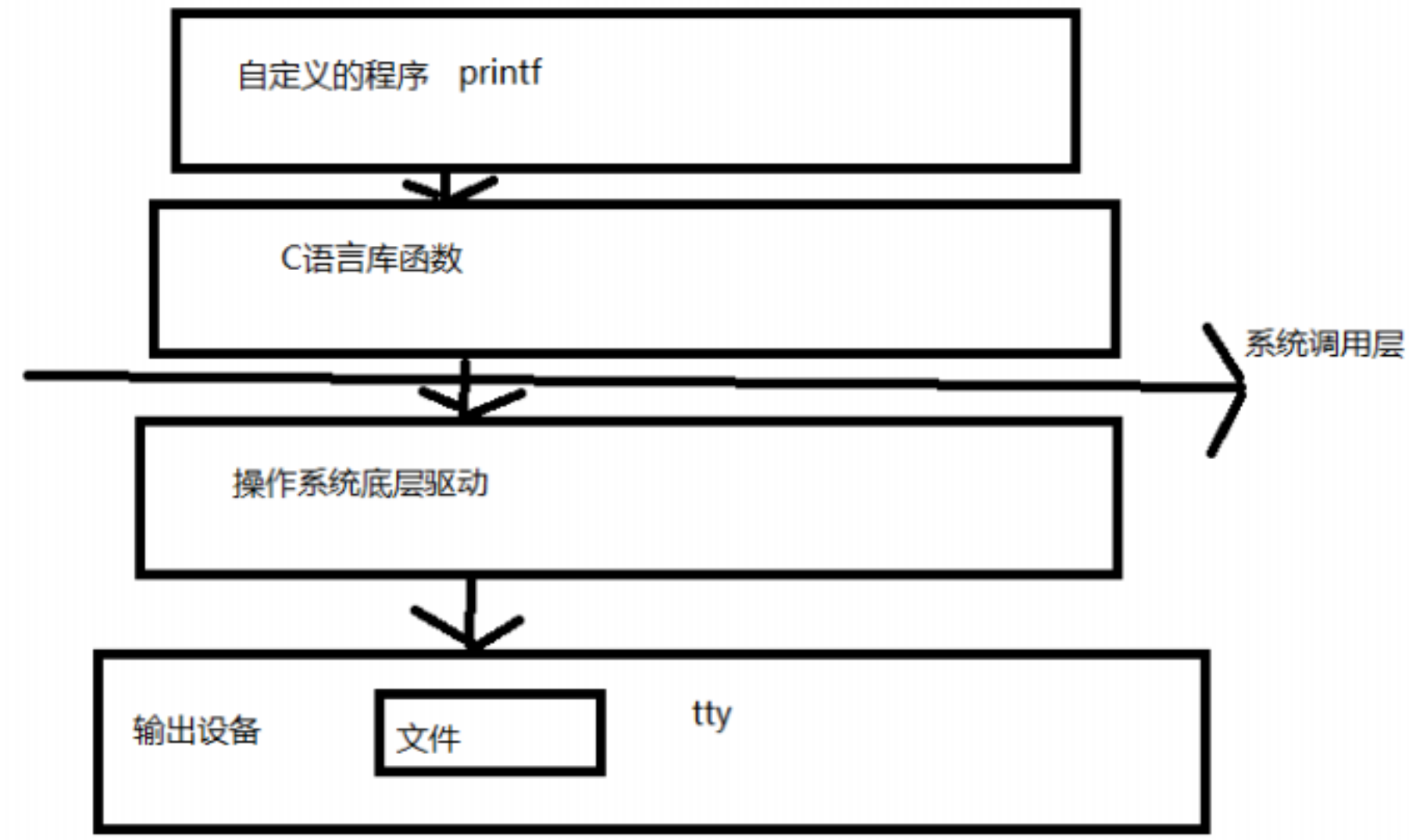
预编译处理 include 的本质就是简单的将 include 中的文件替换到 c 文件中

如果 include 包含的头文件在系统目录下， 那么就用 #include <>， 如果包含的文件在当前目录下，那么用 #inlclude —‖

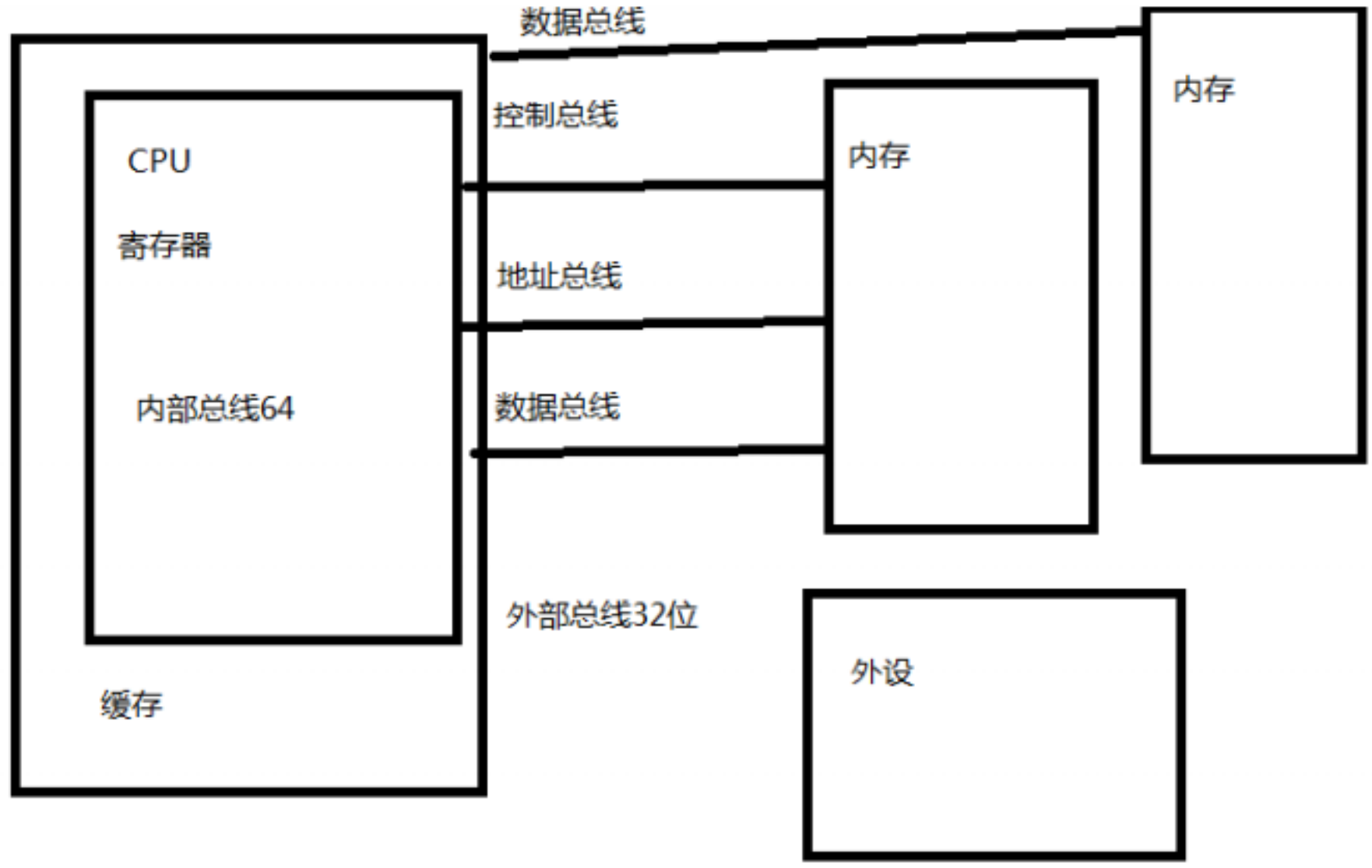
-S 代表汇编

-c 代表编译

2.3 printf 执行原理



2.4 CPU32 位 64 位简介



一个字节是 8 个 bit，一个字是两个字节，整型 4 个字节

8 位寄存器

AL 00000000 256

16 位寄存器

AX AL AH

0000000000000000

1111111111111111

四个通用寄存器

AX

BX

CX

DX

32 位寄存器

EAX 00000000000000000000000000000000
11111111111111111111111111111111

EBX

ECX

EDX

64 位寄存器

REAX

REBX

RECX

REDX

3 C 语言中的数据类型

重点： **

3.1 常量

常量就是在程序中不可变化的量 ,常量是不可被赋值的。

3.1.1 #define

#define 的本质就是简单的文本替换

通过 #define 定义的常量，在 C 语言里面一般叫宏定义

3.1.2 const

const 定义一个变量，但是这个变量的值只能在定义的时候赋予，之后就不可以修改。

对于 const 类型的变量， 一定要在定义的时候给变量赋初值， 不然定义之后就无法赋值了。

3.2 字符串常量

在 C 语言当中“ ”引用的字符串都是字符串常量，常量一旦定义也是不可以被修改的。

3.3 位，字节，字

3.3.1 位

计算机内部都是二进制的，一个二进制的位，就叫做一个 bit，就是一位

3.3.2 二进制

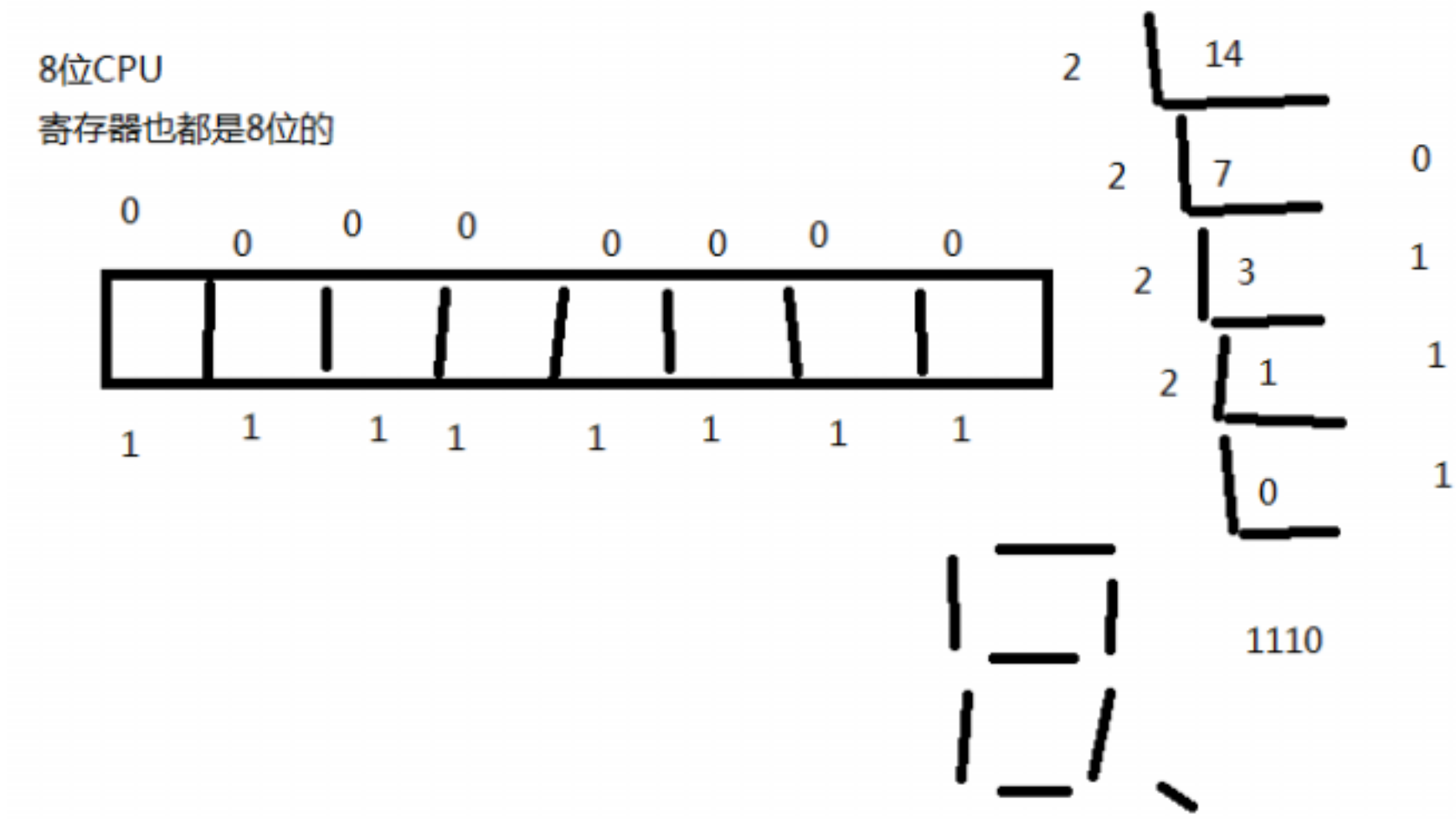
逢二加 1，在二进制表达数的时候是只有 0 和 1，而没有 2 这个数的

二进制最大表示的数，就是 2 几次幂

对于 8 位的 CPU 来讲，最大表达的数是 2 的 8 次幂

3.3.3 十进制

逢 10 加 1，只有从 0 到 9 的数，没有 10 这个数，



3.3.4 八进制

从 0 到 7，逢 8 加 1

在 C 语言中八进制是数字前面加 0

3.3.5 十六进制

0,1,2,3,4,5,6,7,8,9,a,b,c,d,e,f,10

逢 16 加 1，

在 C 语言当中表达一个十六进制数的方式，数字前面加 0x 前缀

3.3.6 字节

8 个 bit 为代表一个字节

3.4 sizeof 关键字

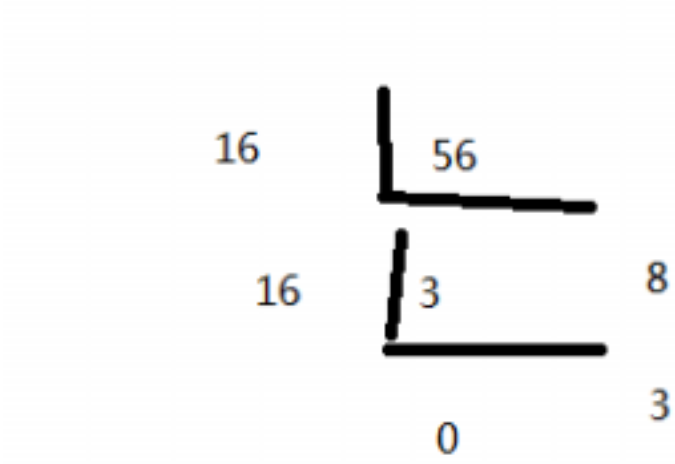
sizeof 与 size_t 类型

sizeof 是计算数据在内存当中占多大空间的，单位字节

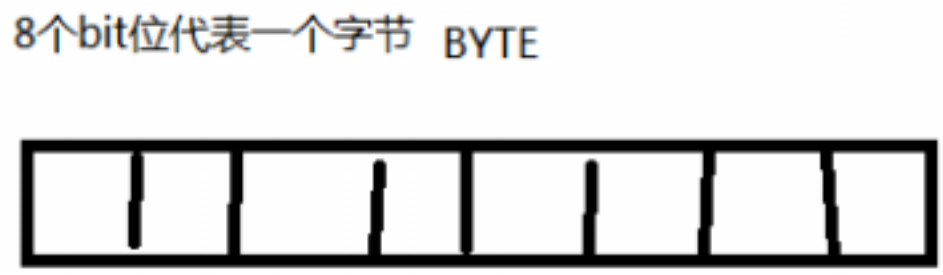
由于 sizeof 永远返回的是一个大于等于 0 的整数，所以如果用 int 来表示 sizeof 的返回值

就不合适， size_t 一般就是一个无符号的整数 .

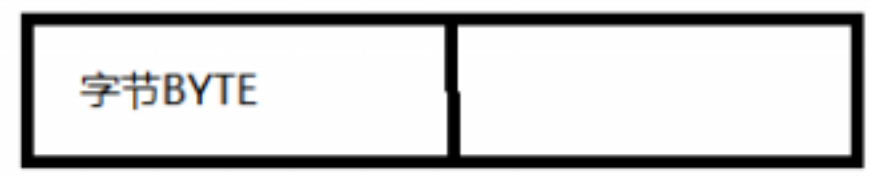
3.5 十进制，二进制，八进制，十六进制



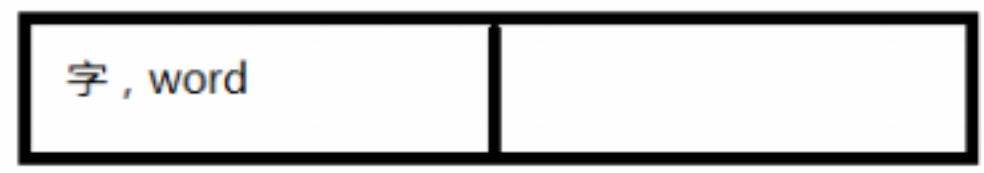
5GB，存贮表达方法
2Mb，bit



两个字节，代表一个字word



两个字，代表一个双字，DWORD



大写B，代表的是BYTE（字节）
小写b，代表的是bit（位）

3.6 int 类型

3.6.1 int 常量，变量

一个 int 型数据占据 4 个字节的内存大小，在 16 位操作系统下， int 是 2 个字节，在 32 和 64 位操作系统下， int 是 4 个字节。

int a;// 代表在内存当中开辟一个 4 个字节大小的空间

a = 10;// 代表 4 个字节的空间内容是常量 10

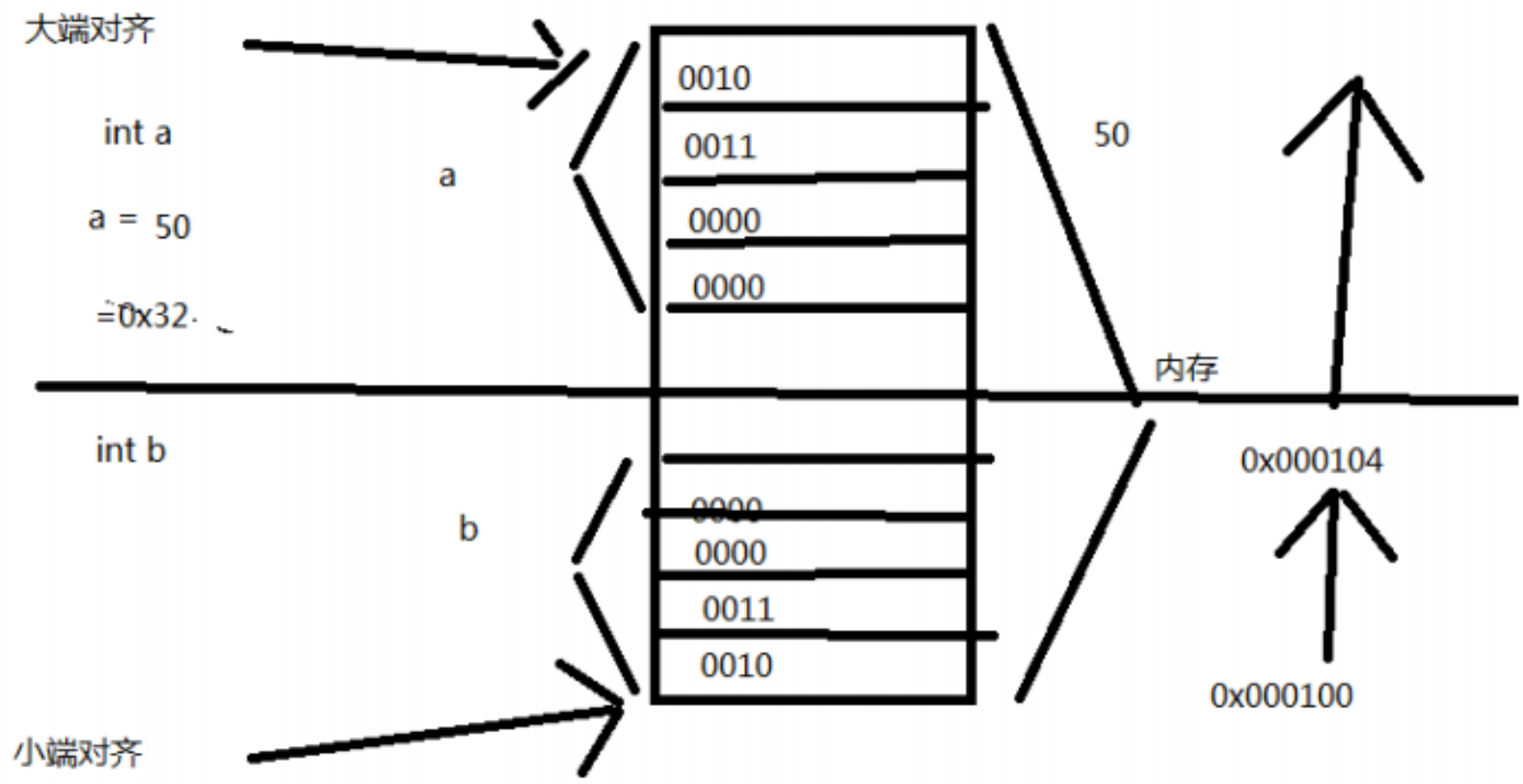
小端对齐和大端对齐

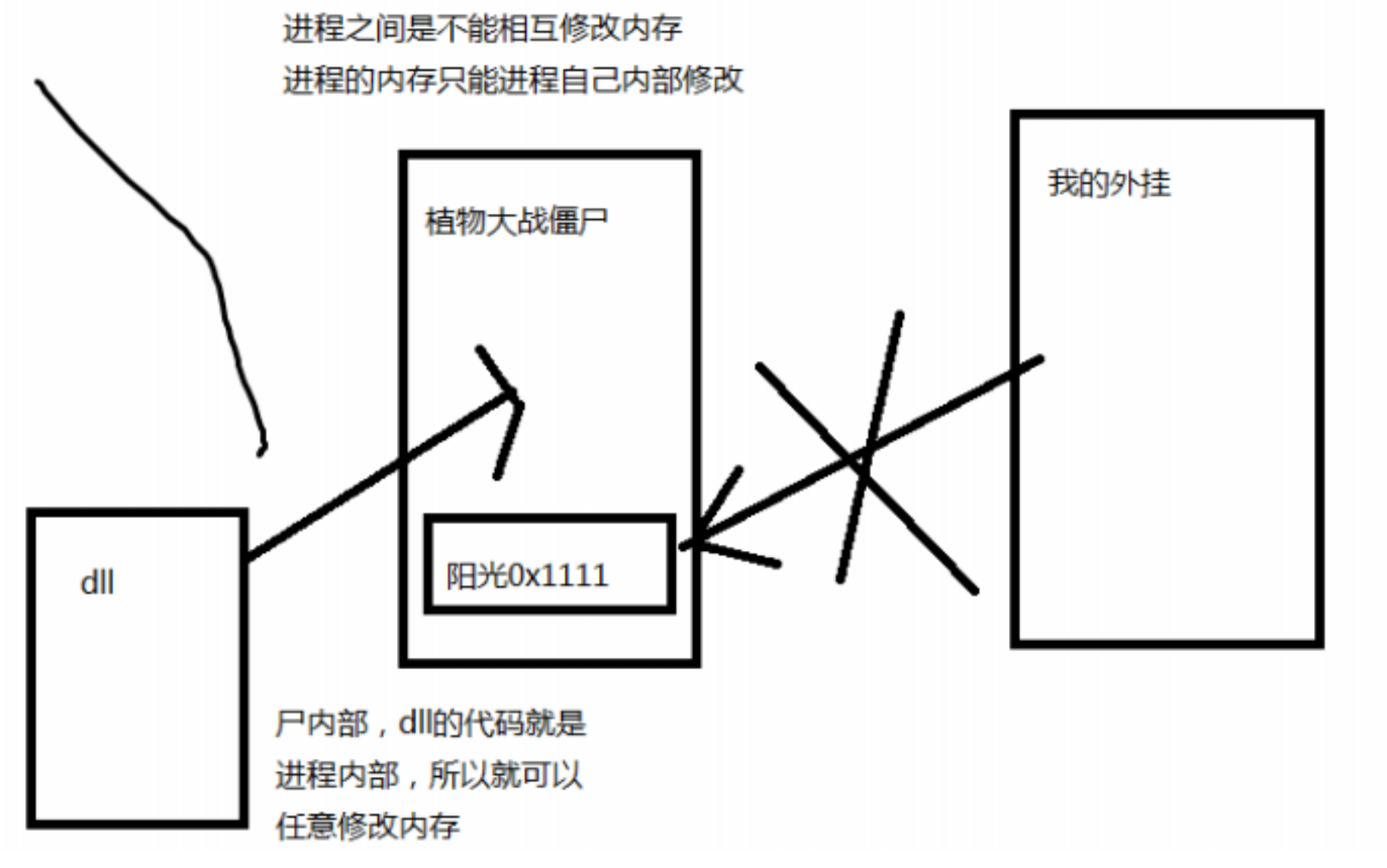
高地址放高位，底地址放低位 --- 小端对齐

高地址放低位，底地址放高位 —大端对齐

对于大型 unix CPU 都是按照大端对齐方式处理 int ,

但对于 x86 构架 CPU，还有 ARM，是小端对齐的


大端对齐
int a
a = 50
=0x32
a
0010
0011
0000
0000
50
内存
int b
b
0000
0000
0011
0010
0x000104
0x000100
小端对齐


进程之间是不能相互修改内存
进程的内存只能进程自己内部修改
植物大战僵尸
我的外挂
dll
阳光0x1111
尸内部，dll的代码就是
进程内部，所以就可以
任意修改内存

3.6.2 printf 输出 int 值

int a = 0x100;// 十六进制

printf(⁻%d ‖a);%d 的意思是按照 10 进制打印一个整数

%x

%X，输出十六进制的时候是用大写的 ABCDEF 还是小写的 abcdef ,

3.6.3 printf 输出八进制和十六进制

%o

3.6.4 short， long， long long ， unsigned int

在 32 位系统下：

short = 2 个字节

long 和 int 一样，是 4 字节

long long 是 8 个字节

在 64 位操作系统下

int， 4 个字节

long 在大多数 64 位系统下 8 个字节

unsigned int// 无符号整数

unsigned long// 无符号的长整数

unsigned short// 无符号短整数

9l,9L,9ll,9LL,9u,9ull,9ULL

3.6.5 整数溢出

当把一个大的整数赋值给小的整数，叫溢出。

int I = 0x12345678

short a = I;

当一个 int 赋值给 short ， 会将高位抛弃,

3.7 char 类型

3.7.1 char 常量，变量

char 是字符型，代表一个字节的内存

char 在内存当中，有符号最大 7f,

无符号，最大 ff

unsigned char

char 的本质就是一个字节，一个 BYTE

3.7.2 printf 输出 char

%c

3.7.3 不可打印 char 转义符

\a, 警报

\b 退格

\n 换行

\r 回车

\t 制表符

斜杠

单引号

双引号

\?问号

3.7.4 char 和 unsigned char

char 取值范围为 -128 到 127

unsigned char 为 0-255

3.8 浮点 float,double,long double 类型

3.8.1 浮点常量，变量

float f = 2.5;

double f1 = 3.1415926

3.8.2 printf 输出浮点数

%f,%Lf

3.9 类型限定

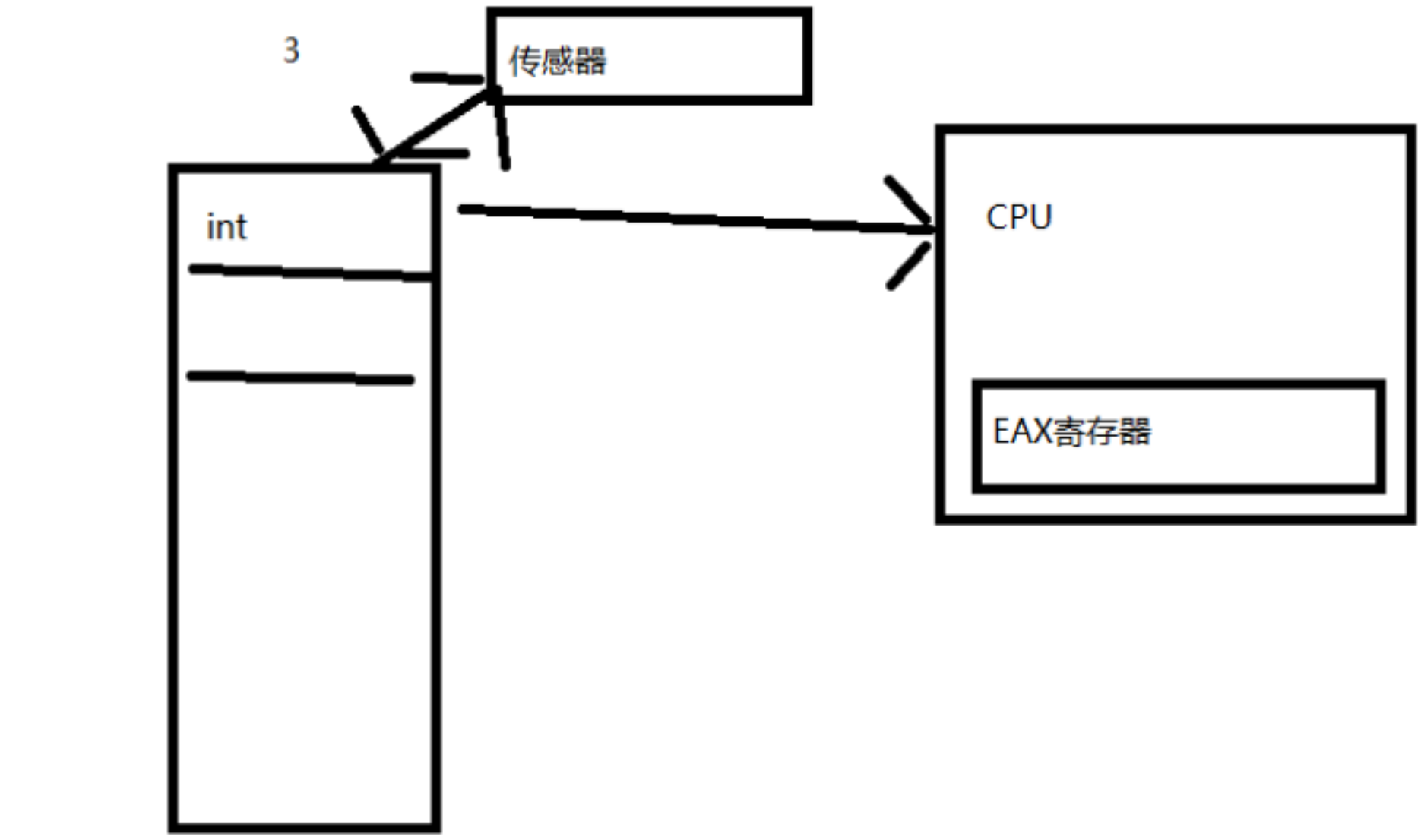
3.9.1 const

const 限定一个变量的值不可以改变

const 伪常量，做到尽量不直接修改，如果是间接修改，就不一定了。

3.9.2 volatile

告诉编译器不要自作聪明的给我优化代码，把我的变量优化的寄存器里面计算，只要是 volatile 类型变量，每一步都需要从内存当中读取。



3.9.3 register

register 告诉编译器， 这个变量只是用寄存器就好， 提高效率， 所以说 register 只是一个建议，而不是必须的结果。

4 字符串格式化输出和输入

重点： *

4.1 字符串在计算机内部的存储方式

字符串是内存中一段连续的 char 空间，以 \0 结尾

字符串就是 0 结尾的连续 char 的内存

4.2 printf 函数， putchar 函数

printf 格式字符

字符	对应数据类型	含义
d	int	接受整数值并将它表示为有符号的 十进制 整数
hd	Short int	短整数

hu	Unsigned short int	无符号短整数
o	unsigned int	无符号 8 进制整数
u	unsigned int	无符号 10 进制 整数
x / X	unsigned int	无符号 16 进制 整数， x 对应的是 abcdef ， X 对应的是 ABCDEF
f	float 或 double	单精度浮点数 或 双精度浮点数
e / E	double	科学计数法 表示的数， 此处 "e" 的大小写代表在输出时用的 “ e” 的大小写
c	char	字符 型。可以把输入的数字按照 ASCII 码相应转换为对应的字符
s / S	char * / wchar_t *	字符串 。输出字符串中的字符直至字符串中的空字符（字符串以 0,结尾，这个 '\0' 即空字符）
p	void *	以 16 进制形式输出 指针
%	%	输出一个百分号

printf 附加格式

字符	含义
l	附加在 d,u,x,o 前面， 表示长整数
-	左对齐
m(代表一个整数)	数据最小宽度
0	将输出的前面补上 0 直到占满指定列宽为止 不可以搭配使用 “-”
N(代表一个整数)	宽度至少为 n 位 不够以空格填充

printf 是打印一个字符串

putchar 是打印一个字符

4.3 scanf 函数与 getchar 函数

5　运算符表达式和语句

重点： ***

5.1　基本运算符

5.1.1　=

数据对象：泛指数据在内存的存储区域

左值：表示可以被更改的数据对象

右值：能赋给左值的量

5.1.2　+

5.1.3　-

5.1.4　*

5.1.5　/

5.1.6　%

取模，取余数

5.1.7　+=

int a = 10;

a = a + 5;

可以简写成　a += 5;

5.1.8 -=

a = a – 5;a -= 5;

5.1.9 *=

a = a * 5;a *= 5;

5.1.10 /=

5.1.11 %=

5.1.12 ++

笔试面试特别容易考，也是特别容易出错的地方

5.1.13 --

5.1.14 逗号运算符

int I = 6 + 5, 3 + 2

逗号表达式先求逗号左边的值，然后求右边的值，整个语句的值是逗号右边的值。

5.1.15 运算符优先级

优先级	运算符	结合性
1	++ （后缀）， --（后缀），()（调用函数）， {} （语句块）， .,->	从左到右
2	++(前缀), --（前缀）, +（前缀）,-（前缀）,!（前缀）,~（前缀）,sizeof,*(取指针值)， &（取地址），（type ）（类型转化）	从右到左
3	* ， / ， %	从左到右
4	+ ， -	从左到右
5	<< >>	从左到右
6	< > <= >=	从左到右
7	== !=	从左到右
8	&	从左到右
9	^	从左到右

10	|	从左到右
11	&&	从左到右
12	||	从左到右
13	?	从右到左
14	=,*=,%=,+=,-=,<<=,>>=,&=,|=,^=	从右到左
15	,(逗号运算符)	从左到右

5.2 复合语句

{} 代码块

5.3 类型转化

```
double  d =    10.2 ;
    int   a = d;   //  隐式转换
```

```
int   a1 =   3;
int   a2 =   2;
double  mid = (   double )a1 / (    double )a2;  //  显示的强转
    printf(    "mid = %f\n"    , mid);
```

6 条件分支语句

重点： ***

6.1 关系运算符

在 C 语言中 0 代表假，非 0 代表真，

在计算机程序判断是根据一个条件返回是真还是假来做不同的操作。

6.1.1 <

6.1.2 <=

6.1.3 >

6.1.4 >=

6.1.5 ==

一个 = 号在 C 语言里面是赋值的，不是比较的，但是很多初学者爱犯一个严重的错误，就是用 = 号来比较两个数是否相等

6.1.6 !=

!=

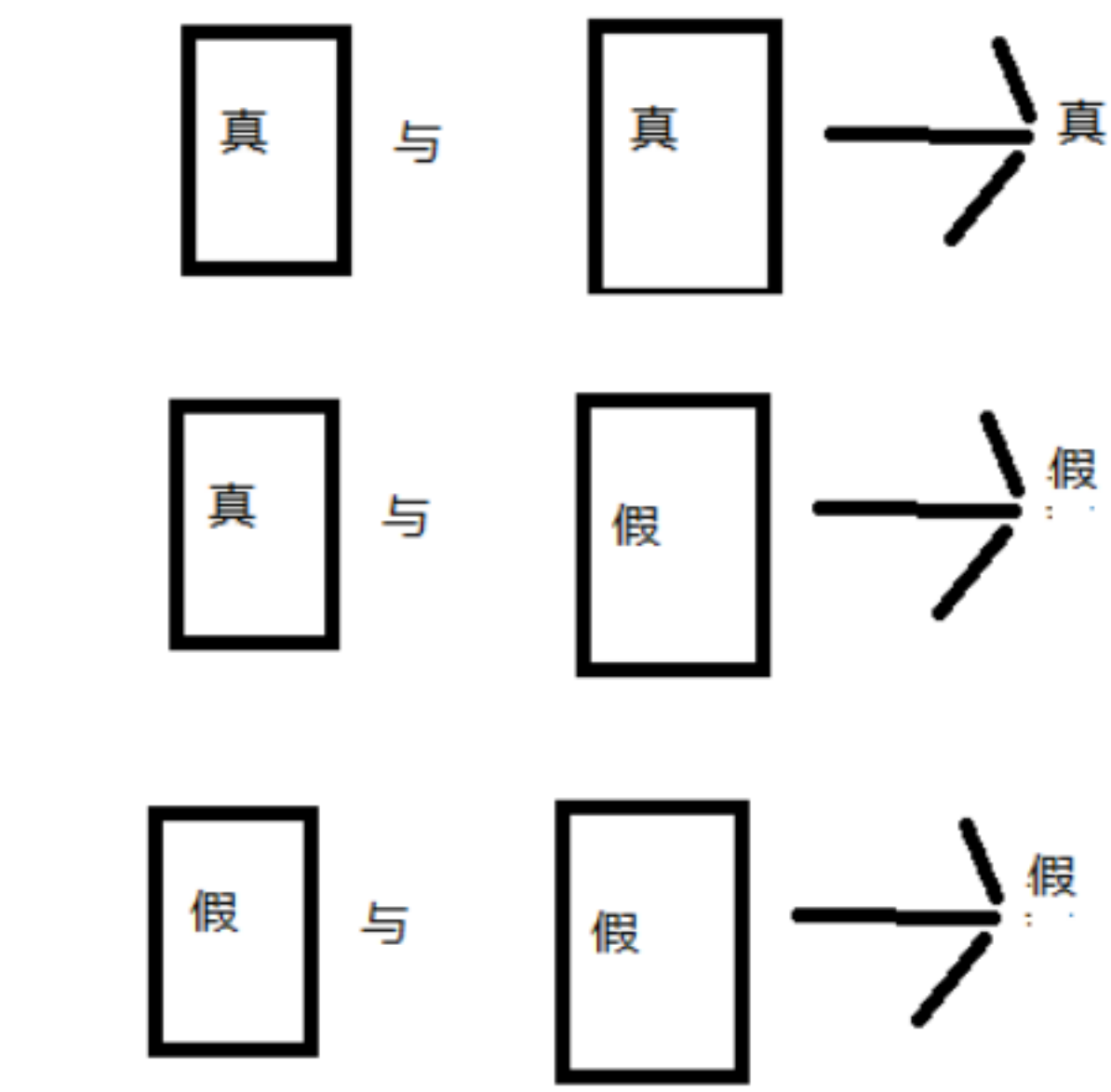
6.2 关系运算符优先级

前四种相同，后两种相同，前四种高于后两种优先级

6.3 逻辑运算符

6.3.1 &&

逻辑与

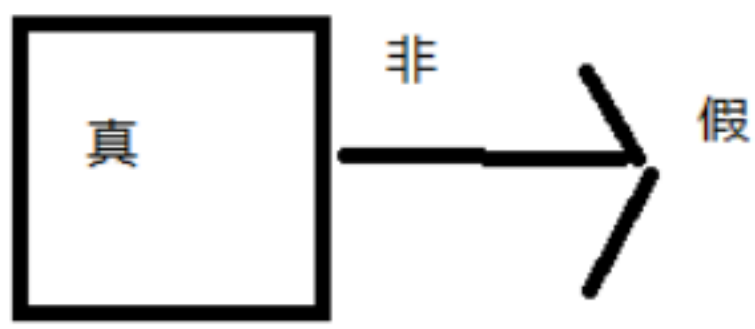


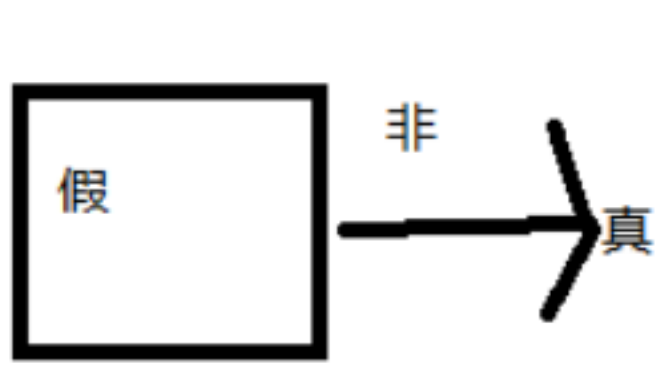
6.3.2 ||

逻辑或

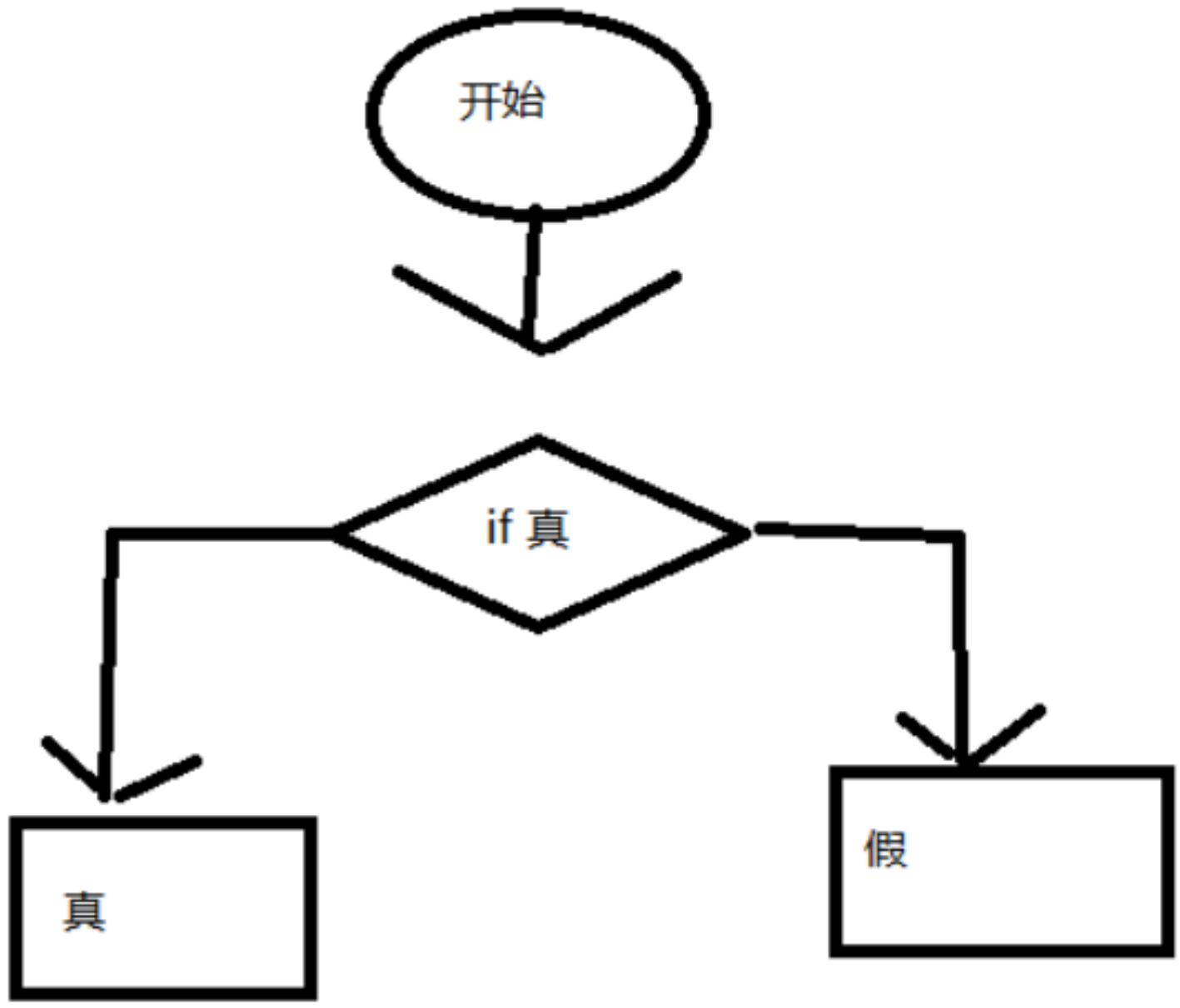
6.3.3 !

逻辑非





6.4 if



6.5 if else

两路分支 if 语句，只可能执行一路，不可能同时执行两路，也不可能两路都没有被执行

6.6 if else if

if else if 这种于语法可以实现多路的分支，但只有一路被执行。

else 永远是和最近的一条 if 语句配对。

6.7 switch 与 break,default

switch 是为多重选择准备的，遇到 break 语句， switch 就终端执行

6.8 条件运算符？

一个求绝对值的例子

```
int   i = -    8;
int   x = (i <       0) ? -i: i;
```

?号用法

当？号　前面括号内容为真的时候，执行？号之后冒号之前的语句，否则执行冒号之后的语句

6.9　goto 语句与标号

尽量不要在程序当中使用　goto 语句,

7　循环语句

重点： ***

7.1　while

while （条件），如果条件为真，那么循环就执行，否则循环退出

7.2　continue

continue　意思是跳过下面语句，继续执行循环

7.3　break

break 中断循环,

7.4　do while

do

　　语句

while (条件);

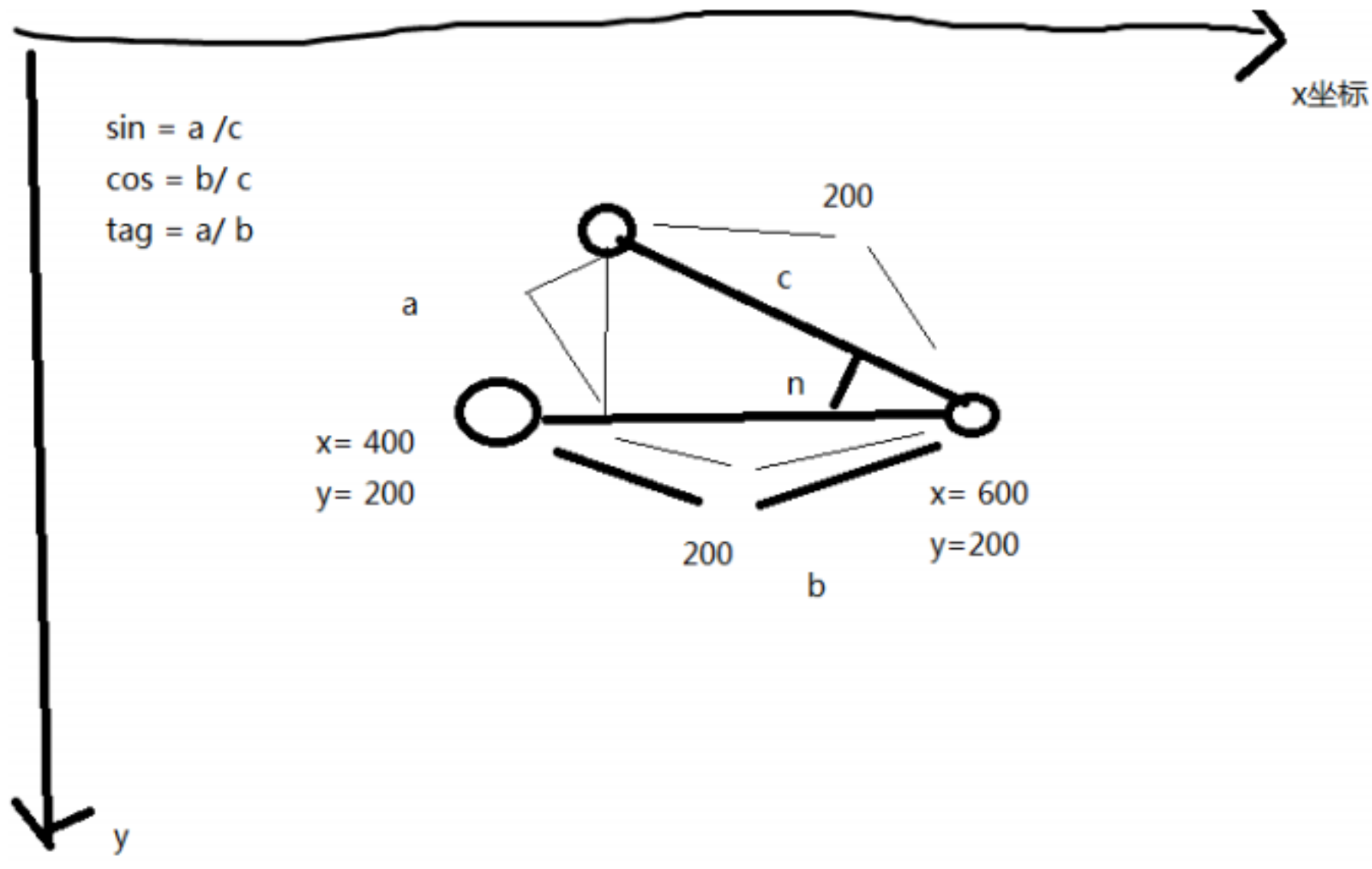
对于 do while 来讲，至少能执行一次,

7.5　for

可以指定循环次数的语句

7.6 循环嵌套

特别需要掌握的技术,



8 整数在计算机内部的存储方式

重点： ****

8.1 原码

将最高位做为符号位（ 0 代表正， 1 代表负），其余各位代表数值本身的绝对值

+7 的原码是 00000111

-7 的原码是 10000111

+0 的原码是 00000000

-0 的原码是 10000000

8.2 反码

一个数如果值为正，那么反码和原码相同

一个数如果为负，那么符号位为 1，其他各位与原码相反

+7 的反码 00000111

-7 的反码 11111000

-0 的反码 11111111

8.3 补码

原码和反码都不利于计算机的运算，如：原码表示的 7 和-7 相加，还需要判断符号位。

正数：原码，反码补码都相同

负数：最高位为 1，其余各位原码取反，最后对整个数 + 1

-7 的补码： =

10000111 （原码）

111111000 （反码）

11111001(补码)

+0 的补码为 00000000

-0 的补码也是 00000000

补码符号位不动，其他位求反，最后整个数 + 1 ，得到原码

用补码进行运算，减法可以通过加法实现

7-6=1

7 的补码和 -6 的补码相加： 00000111 + 11111010 = 100000001

进位舍弃后，剩下的 00000001 就是 1 的补码

-7+6 = -1

-7 的补码和 6 的补码相加： 11111001 + 00000110 = 11111111

11111111 是-1 的补码

9 数组

重点： **

内存连续 ， 并且是同一种数据类型的变量 ,C 语言的数组小标好是从 0 开始的，到 n-1.

9.1 一维数组定义与使用

类型 变量名称 [数组元素的个数];

9.2 数组在内存的存储方式

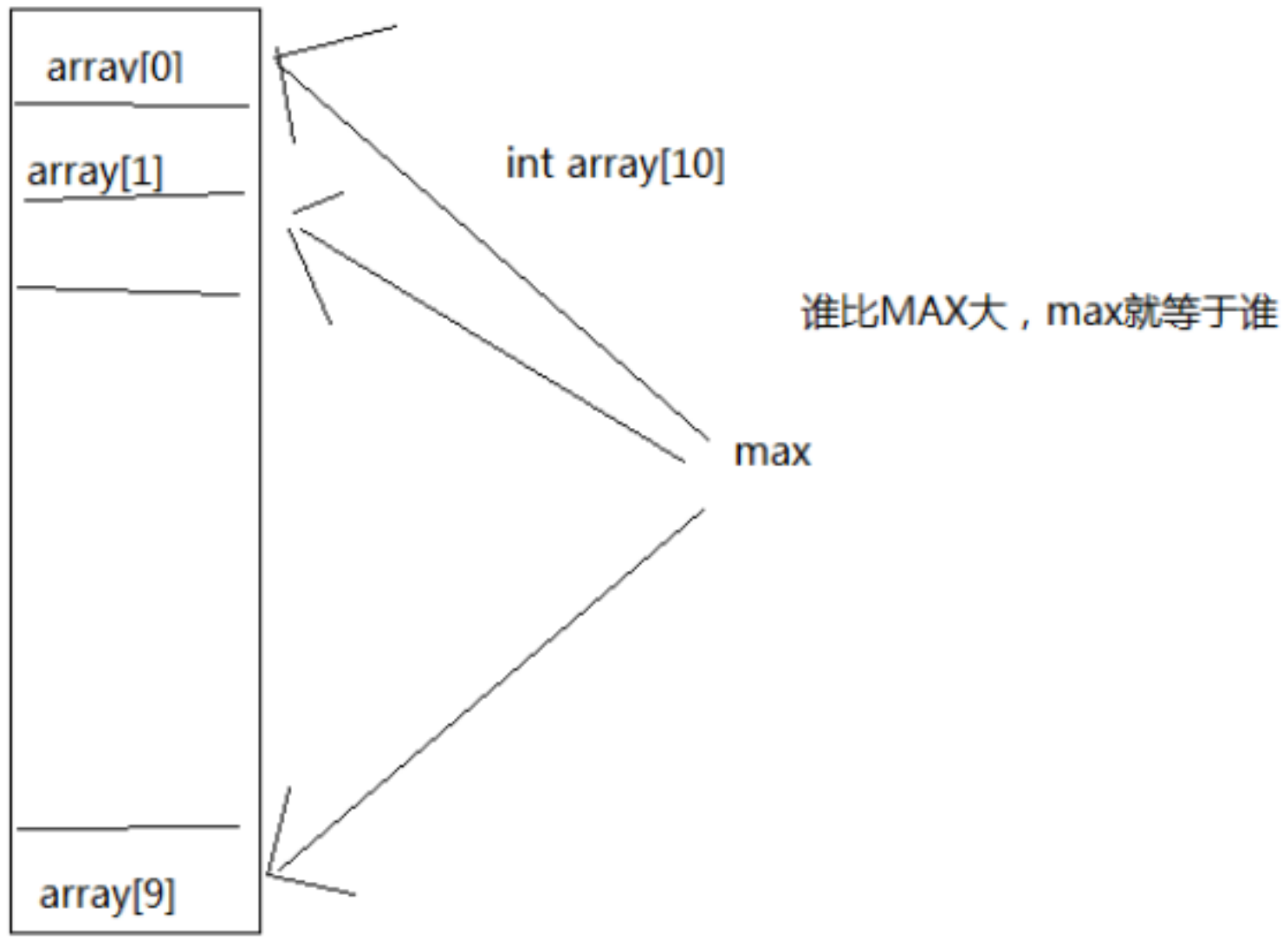
在内存当中是连续的内存空间地址。

9.3 一维数组初始化

int array[10] = {0};// 将数组所有元素都初始化为 0

int array[10] = {0,1,2,3,4,5,6,7,8,9}

数组中找最大值思路



数组中找第二大值思路

第一次循环先找到数组
当中最大的那个值

第二次循环，找最大，
但是小于第一次循环当
中找到的最大那个值

max=前两个元素当中最大那个

smax=前两个当中第二大那个

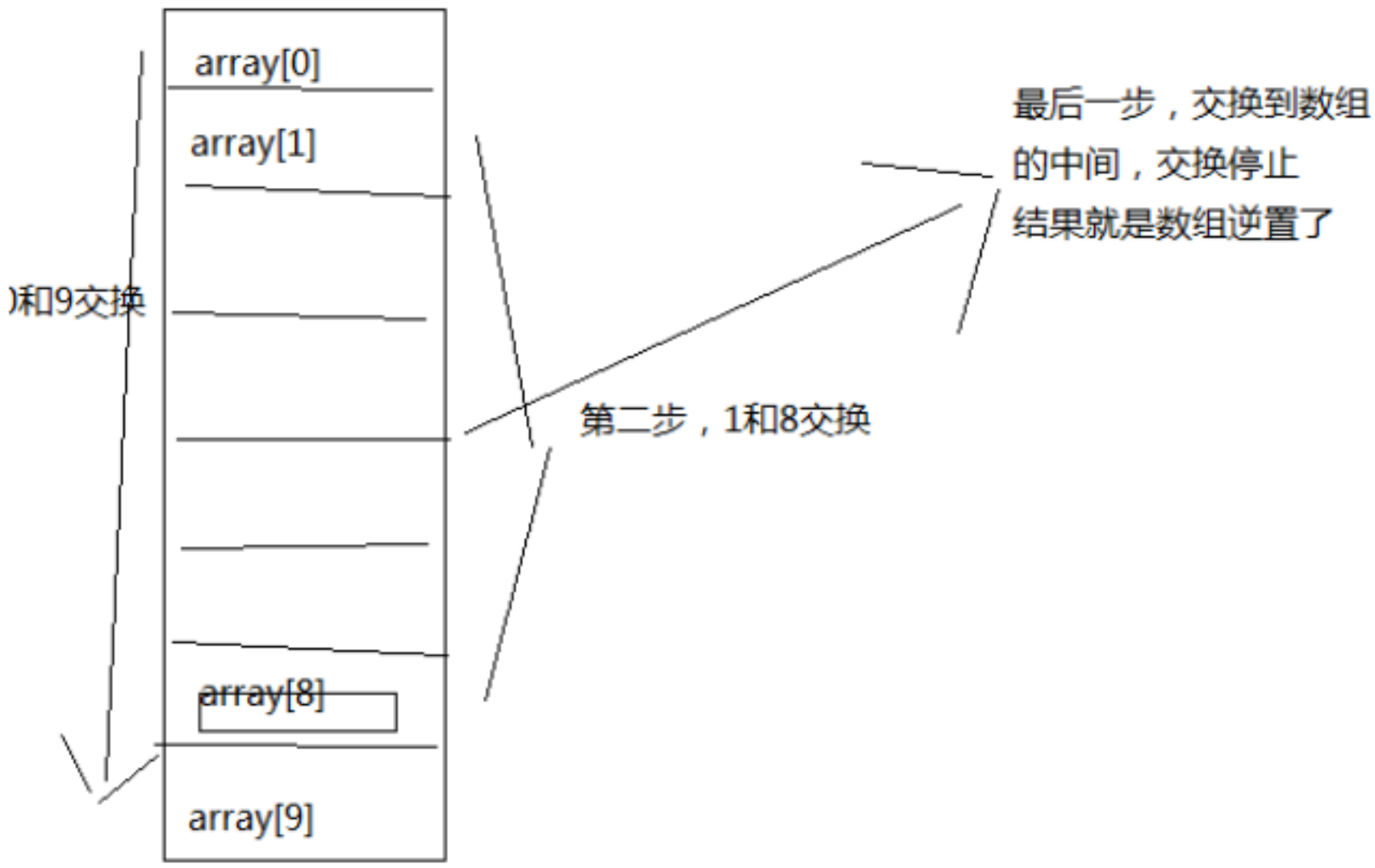
一次循环

max=最大值

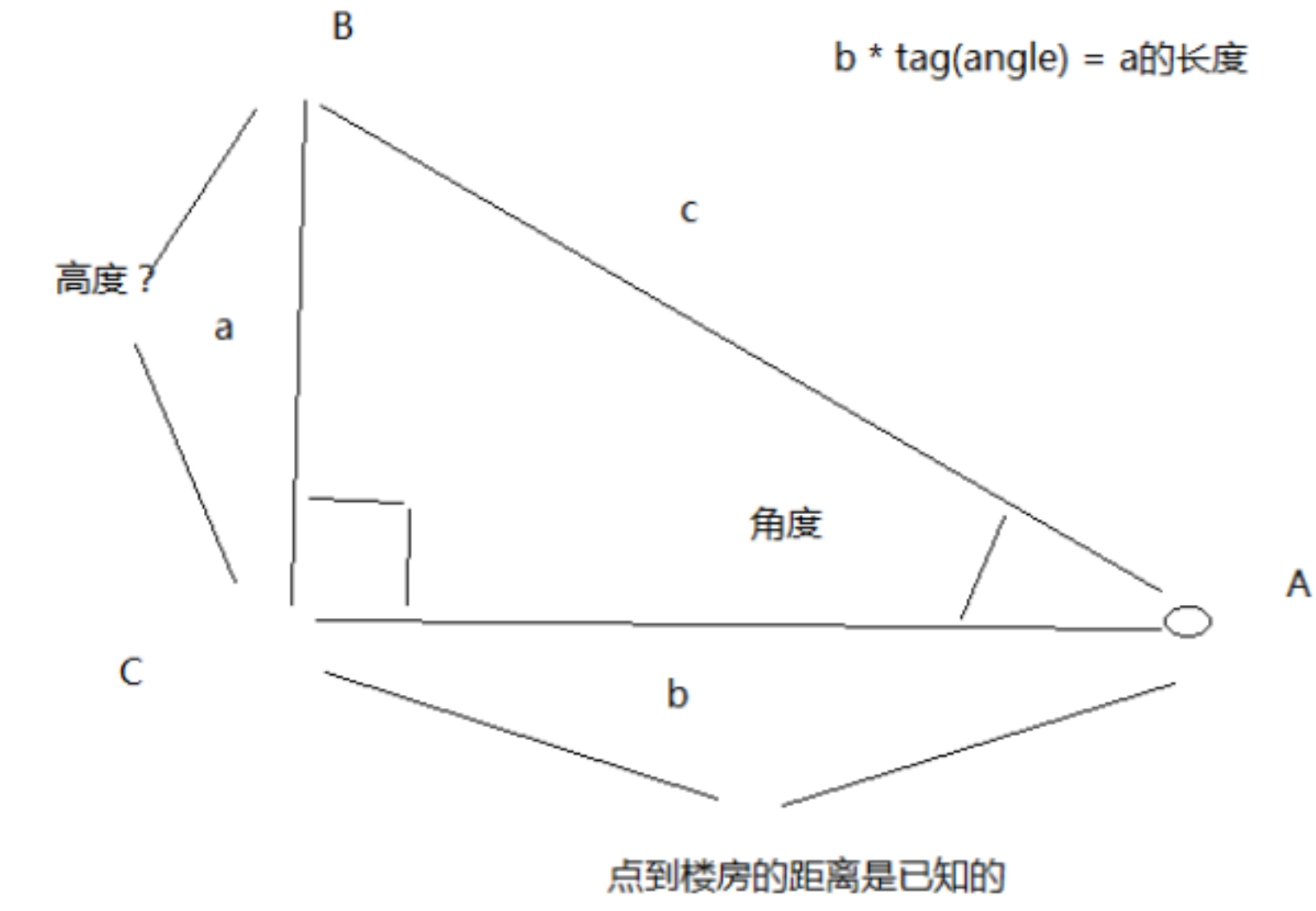
smax=第二个值

从array[2]比较，

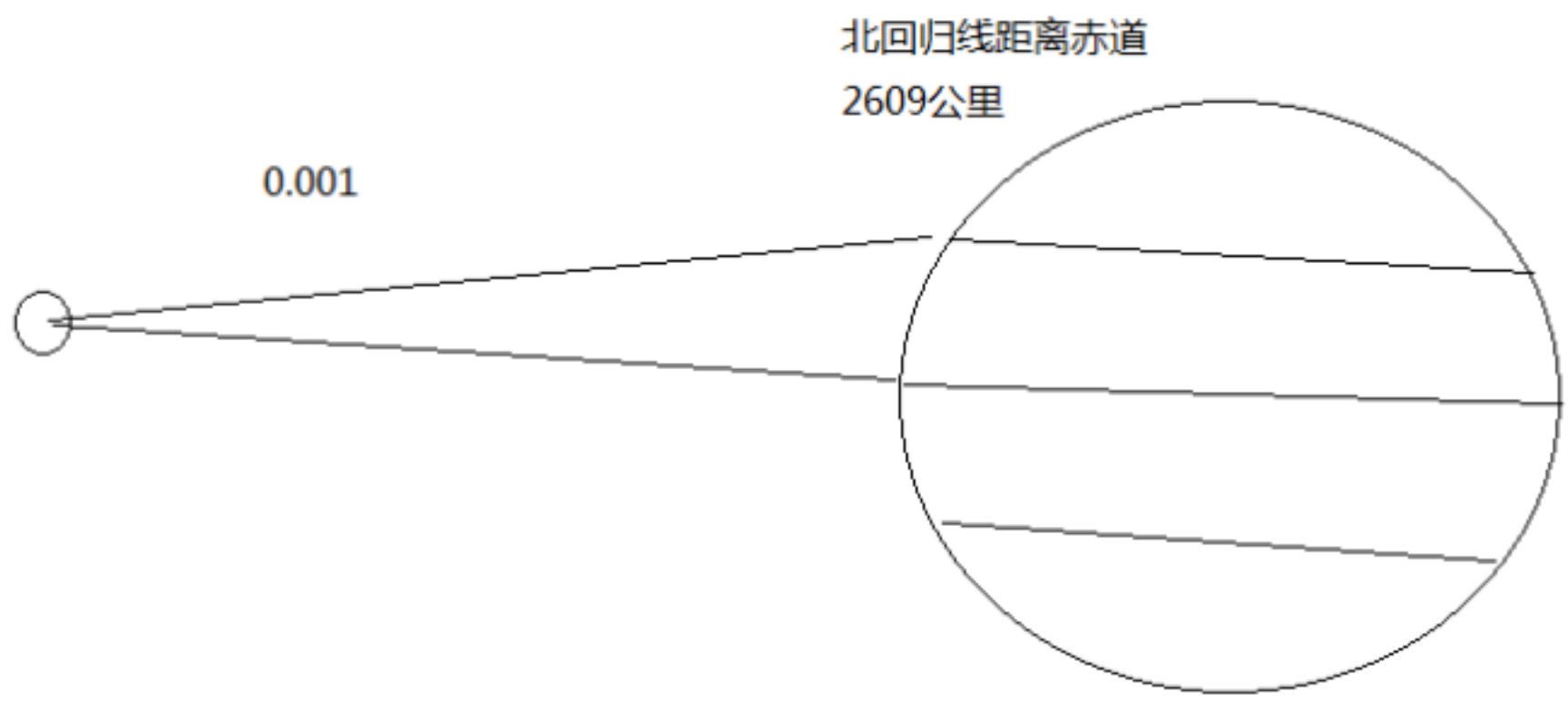
逆置数组思路



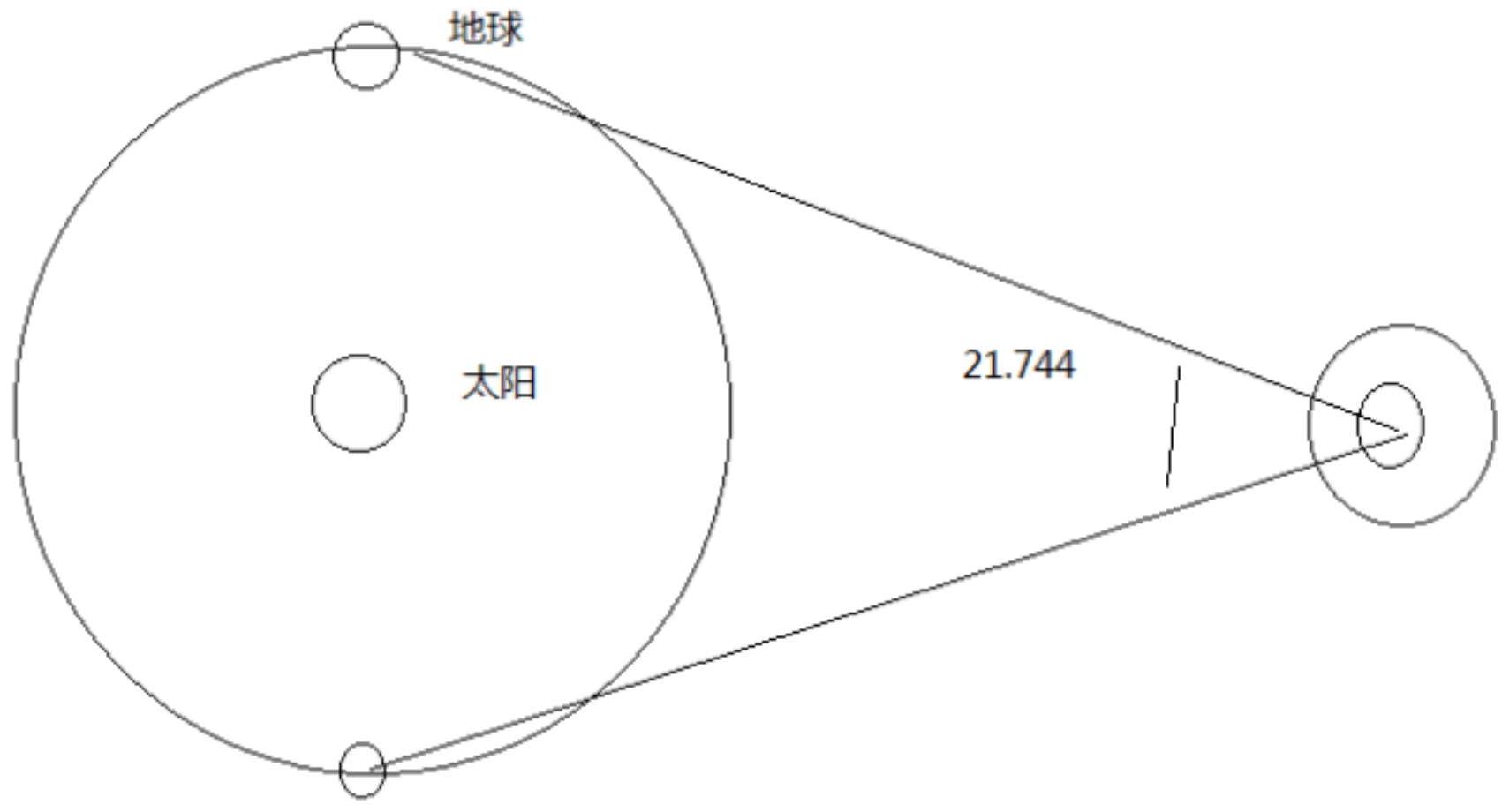
测量楼宇高度的说明



测量地球太阳距离的说明



测量太阳木星距离的说明



9.4 二维数组定义与使用

```
int array[ 3][ 4]; //12 个元素的二维数组
```

9.5 二维数组初始化

```
int a[ 3][ 4] = { { 1, 2, 3, 4 }, { 5, 6, 7, 8 }, { 9, 10, 11, 12 } };
```

10 字符串与字符数组

重点： ***

10.1 字符数组定义

char buf[100];

对于 C 语言字符串其实就是一个最后一个元素为 '\0' 的 char 数组 .

10.2 字符数组初始化

```
char buf[] = "hello world";
```

10.3 字符数组使用

10.4 随机数产生函数 rand 与 srand

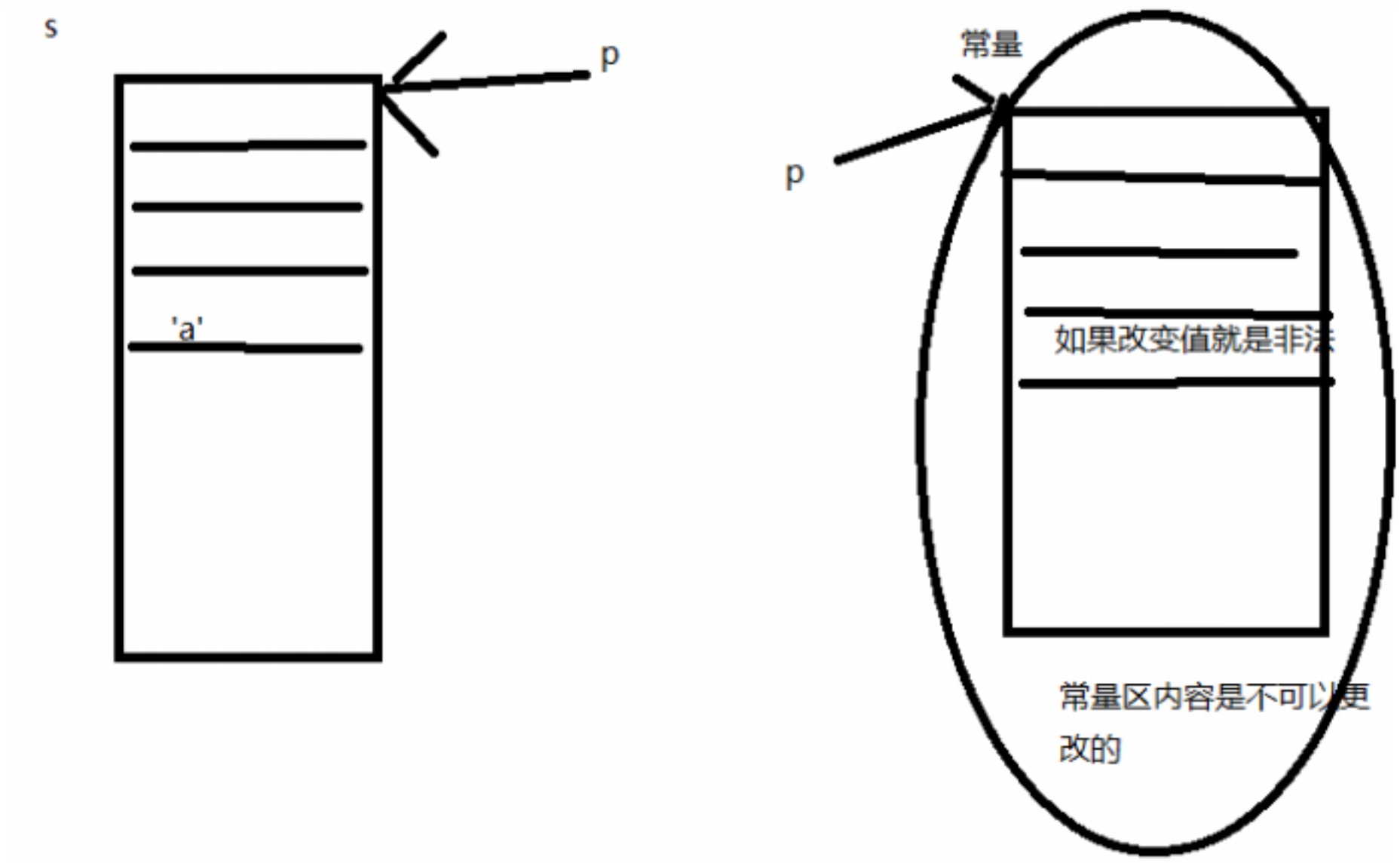
头文件 stdlib.h

```
#include <time.h>
int t = (int)time(NULL);
srand(t);
for (int i = 0; i < 10; i++)
{
    printf("%d\n", rand());
}
```

10.5 用 scanf 输入字符串

```
char buf[100] = {0};

scanf("%s", buf);

scanf("请输入 i 的值%d", &i);
```

10.6 字符串的结束标志



10.7 字符串处理函数

10.7.1 gets

gets 没有解决缓冲区溢出的问题 .

10.7.2 fgets 函数

gets 函数不检查预留缓冲区是否能够容纳用户实际输入的数据。 多出来的字符会导致内存溢出， fgets 函数改进了这个问题。

由于 fgets 函数是为读取文件设计的，所以读取键盘时没有 gets 那么方便

```
char s[ 100] = { 0 };
fgets(s, sizeof (s), stdin );
```

10.7.3 puts 函数

puts 函数打印字符串，与 printf 不同， puts 会在最后自动添加一个 ’ \n ’

```
char  s[] =     "hello world"     ;
puts(s);
```

10.7.4 fputs 函数

fputs 是 puts 的文件操作版本,

```
char  s[] =     "hello world"     ;
fputs(s,     stdout  );
```

10.7.5 strlen ， 字符串长度

strlen 返回字符串的长度，但是不包含字符串结尾的 \0‘

char buf[10]

sizeof(buf);// 返回的是数组 buf 一共占据了多少字节的内存空间 .

10.7.6 strcat ， 字符串追加

char str1[100];

char str2[100];

strcat(str1, str2);// 把 str2 追加到 str1 的后面

str1 一定要有足够的空间来放 str2 ，不然会内存溢出 .

10.7.7 strncat ， 字符串有限追加

strncat(str1, str2, sizeof(str1) －strlen(str1) - 1);

10.7.8 strcmp ， 字符串比较

strcmp(a, －str);// 如果两个参数所指的字符串内容相同，函数返回 0

10.7.9 strncmp ， 字符串有限比较

strncmp(str , －exit ,4);

10.7.10 strcpy 字符串拷贝

strcpy(str ，"hello world ");// 存在溢出的问题,

10.7.11 strncpy 字符串有限拷贝

strcpy(str ，"hello world ", 7);

10.7.12 sprintf ，格式化字符串

printf 是向屏幕输出一个字符串

sprintf 是向 char 数组输出一个字符串，其他行为和 printf 一模一样

sprintf 也存在缓冲区溢出的问题

10.7.13 strchr 查找字符

strchr(str ，'c');

返回值是字符 'c' 在字符串 str 中的位置

10.7.14 strstr 查找子串

10.7.15 strtok 分割字符串

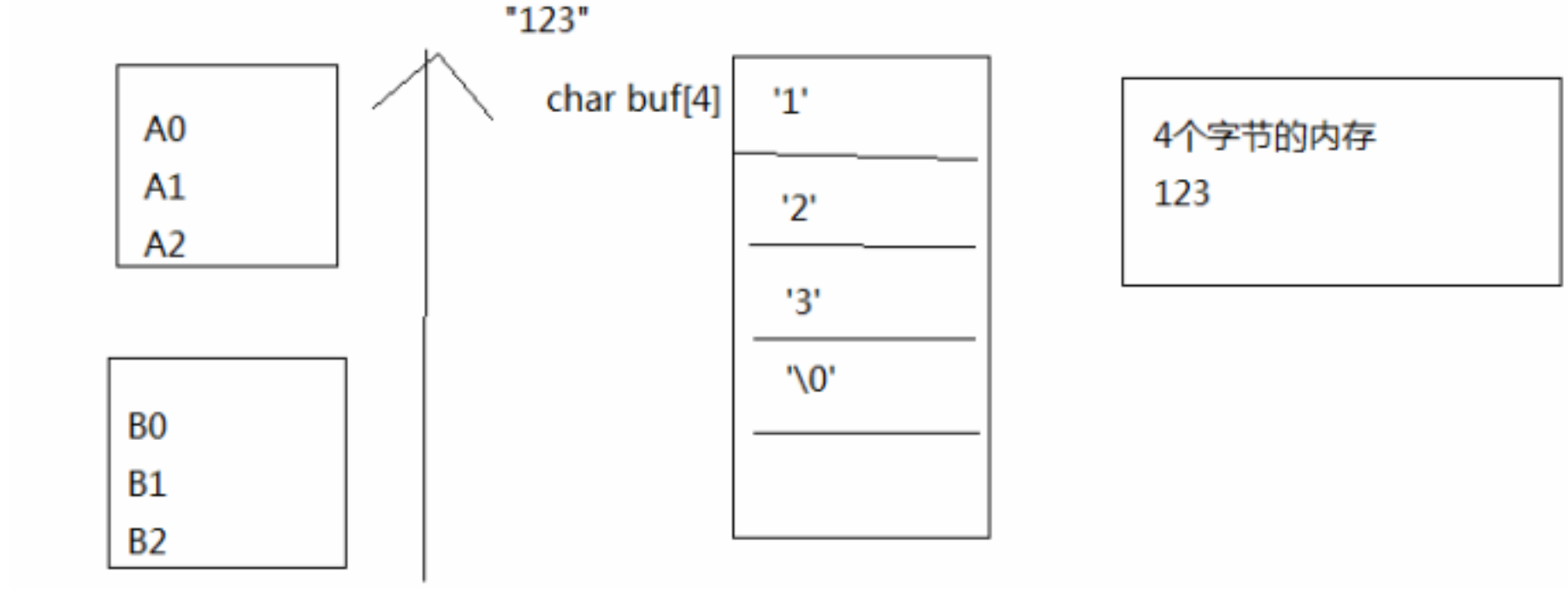
字符在第一次调用时 strtok() 必需给予参数 s 字符串，往后的调用则将参数 s 设置成 NULL 每次调用成功则返回指向被分割出片段的指针

```
char buf[] = "abc@defg@igk";
char *p = strtok(buf, "@");;
while (p)
{
    printf( "%s\n" , p);
    p = strtok( NULL, "@");
}
```

10.7.16 atoi 转化为 int

10.7.17 atof 转化为 float

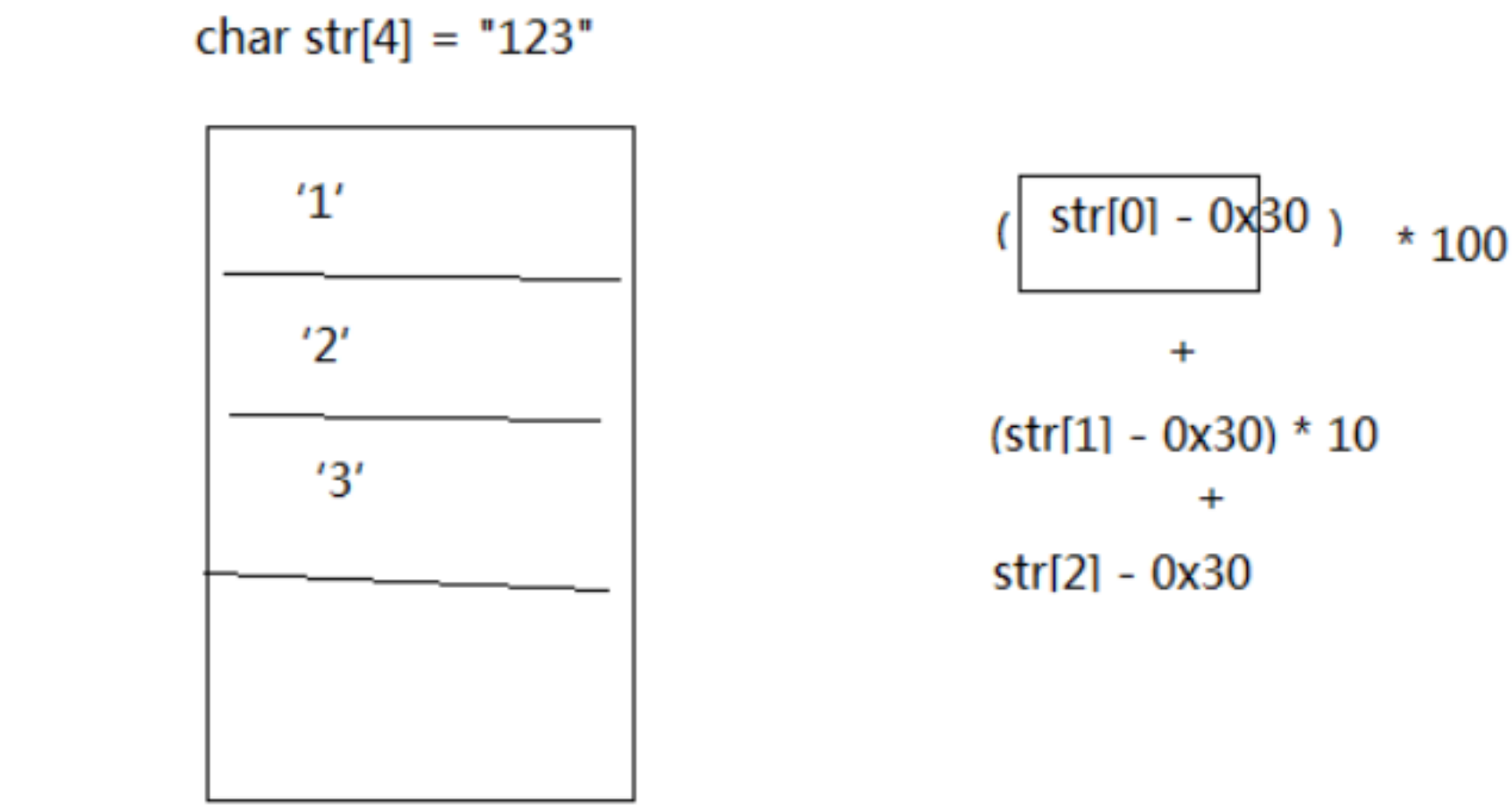
10.7.18 atol 转化为 long



作业说明：

不可以用任何已有的函数，完全自己写代码，完成十进制字符串转化为十进制的整数

作业思路：



11 函数

重点 ***

11.1 函数的原型和调用

在使用函数前必须定义或者声明函数

```
double  circle(     double  r);
int   main()
{
      double  length = circle(        10);
      printf(    "length = %f\n"      , length);
      return    0;
}

double  circle(     double   r )
{
      return    2 *   3.14  *    r ;
}
```

11.2 函数的形参与实参

在调用函数的时候，函数大多数都有参数，主调函数和被调用函数之间需要传递数据。

在定义函数时函数名后面括弧中的变量名称为 “ 形式参数 ”，简称形参。 在调用函数时，函数名后面括号中的变量或表达式称为 “ 实际参数 ” ，简称实参。

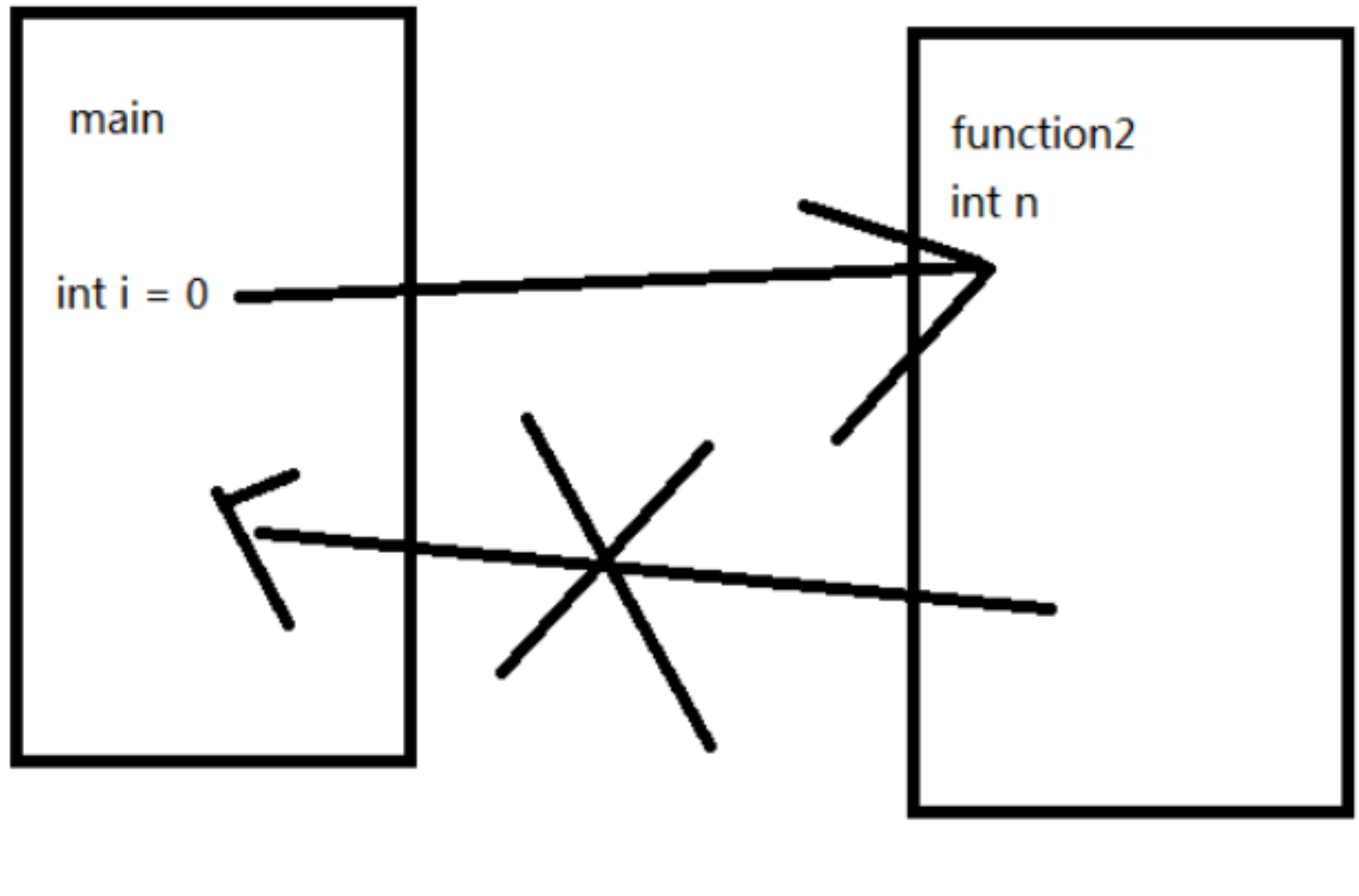
1 形参在未出现函数调用时，他们并不占用内存单元，只有在发生函数调用的时候形参才被分配内存，函数调用完成后，形参所占的内存被释放

2 实参可以是变量，常量或者表达式

3 在定义函数时，一定要指定形参的数据类型

4 形参与实参的数据类型一定要可兼容

5 在 C 语言中，实参与形参的数据传递是 “ 值传递 ” ，即单向传递，只由实参传递给形参，而不能由形参传递给实参。



11.3 函数的返回类型与返回值

1 函数的返回值通过函数中的 return 获得，如果函数的返回值为 void 可以不需要 return 语句。

2 函数 return 语句中的返回值数据类型应该与函数定义时相同。

3 如果函数中没有 return 语句，那么函数将返回一个不确定的值。

11.4 main 函数与 exit 函数

在 main 函数当中遇到 return 语句，代表整个程序结束。在子函数当中遇到 return 代表子函数结束。

不论程序的任何位置调用 exit 函数，代表整个程序退出。

函数的学习难点是递归

11.5 函数的递归

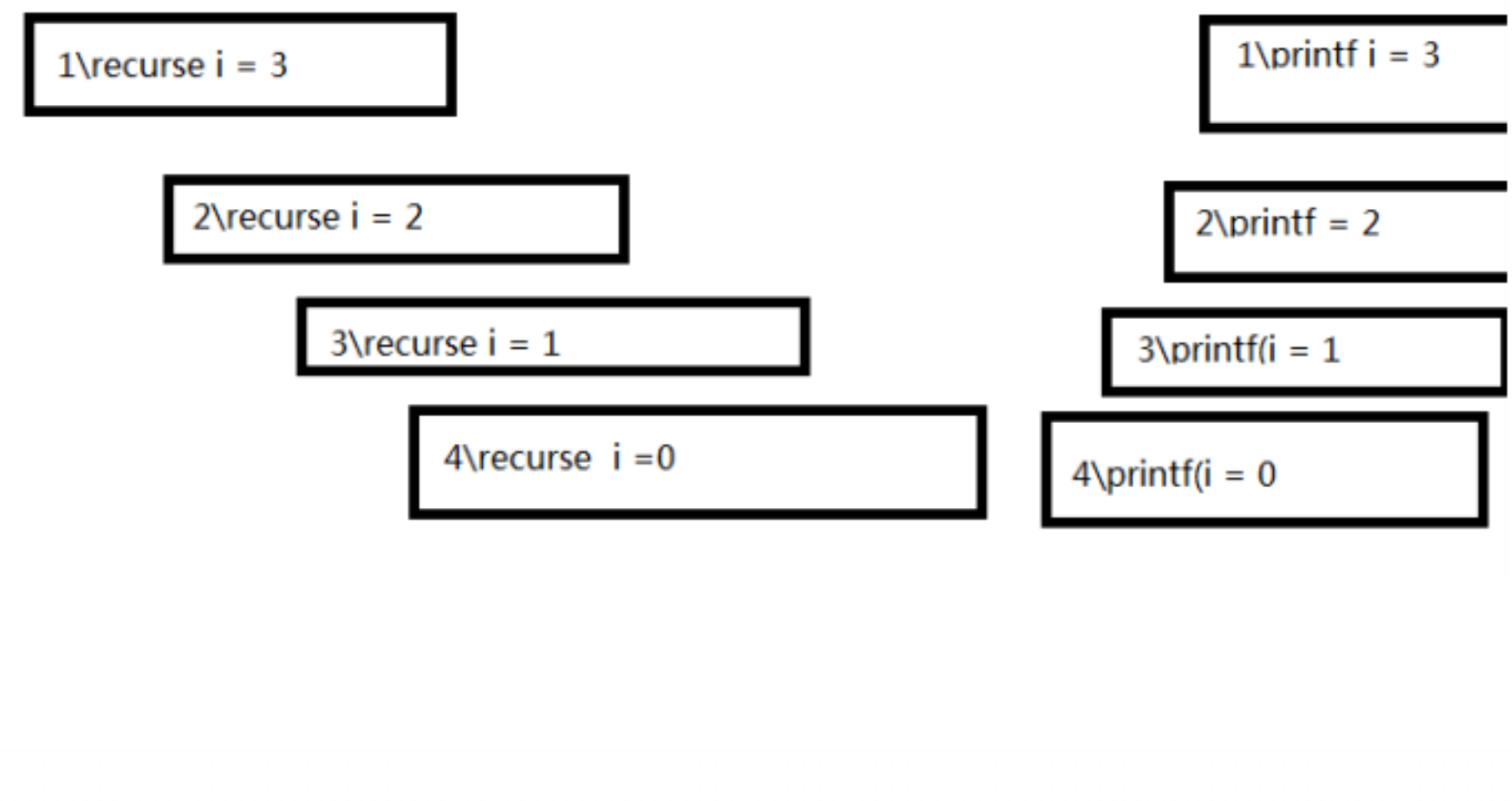
函数可以调用自己，这就叫函数的递归

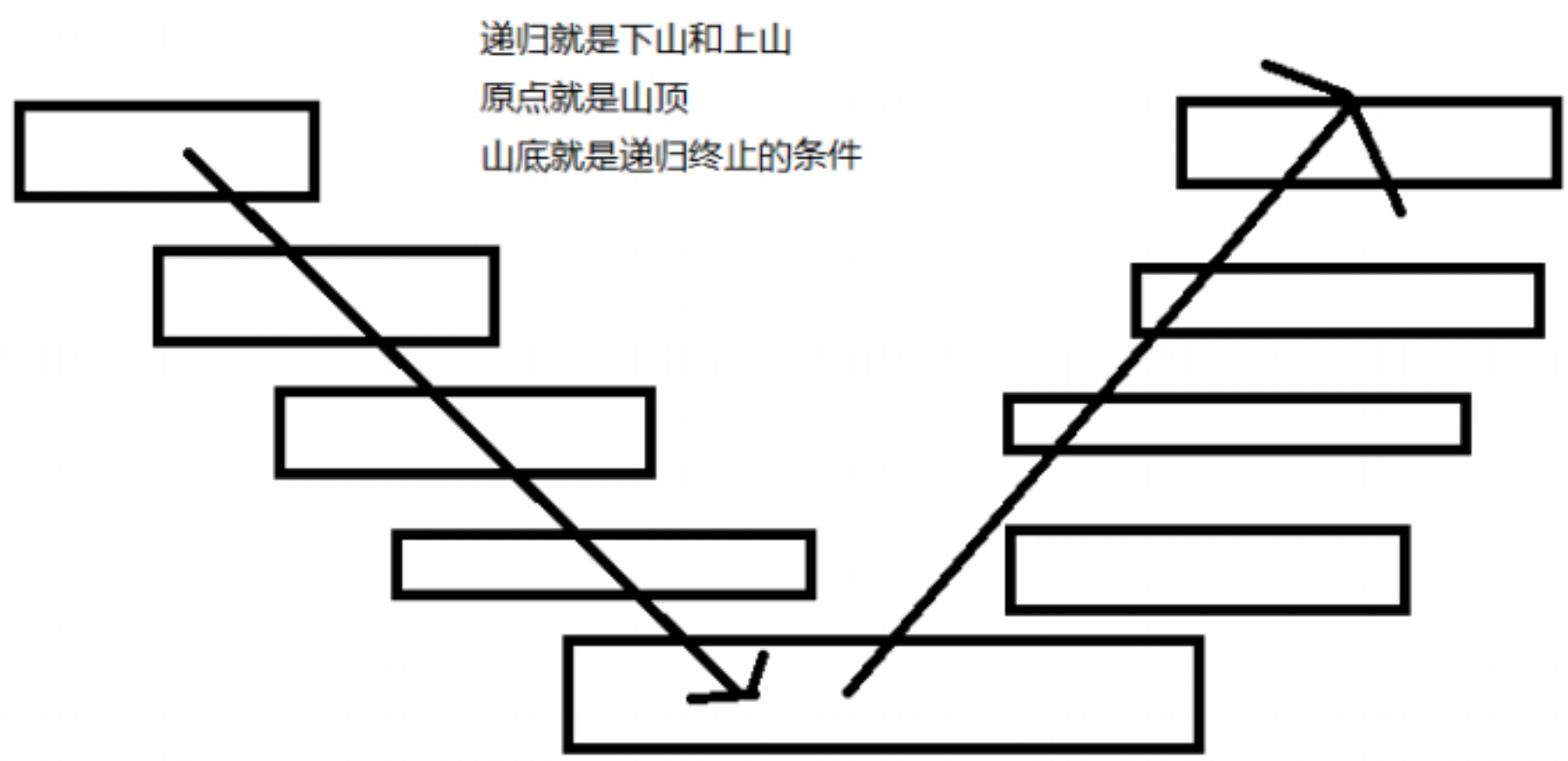
```
void  recurse(   int    i )
{
      if   ( i  >   0)
      {
            recurse(  i  -    1);
      }
      printf(    "i = %d\n"    ,   i );
}

int   main()
{
      recurse(  10);
      return    0;
}
```

11.5.1 递归的过程分析

```
void  up_down( int      n)
{
      printf(    "in %d, location %p\n"       ,   n, &  n);
      if   ( n <   4)
```

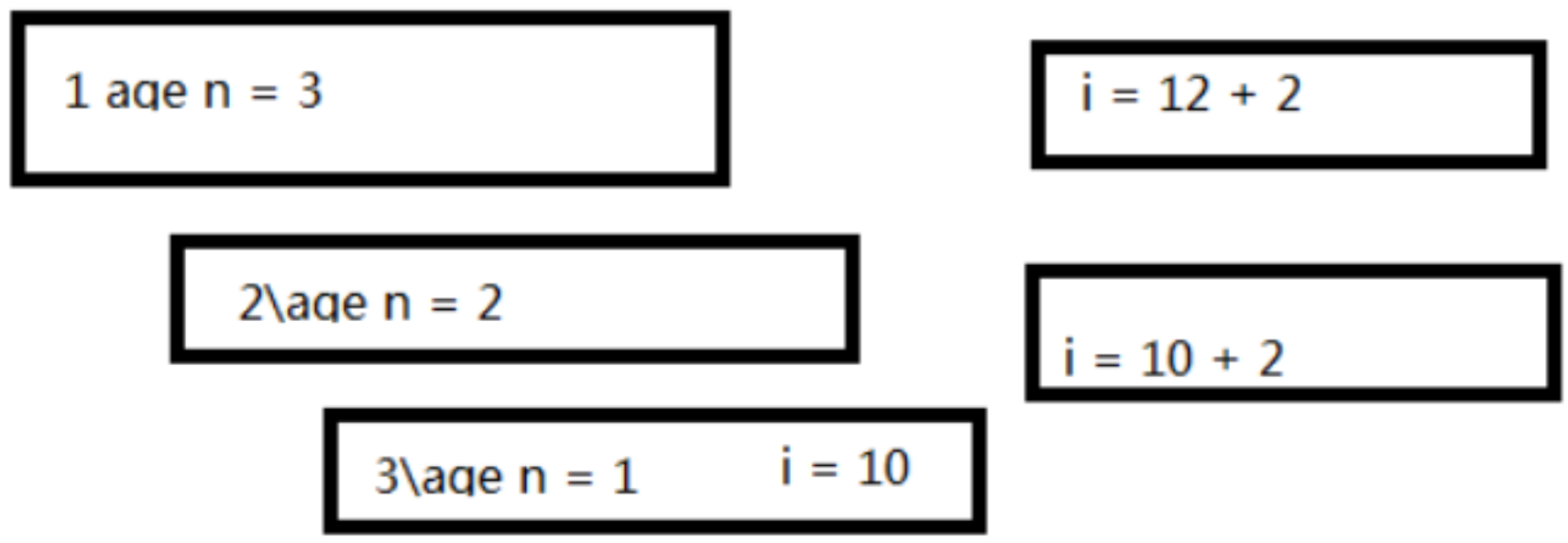
```
        up_down((n +  1));
    printf(   "out %d, location %p\n"      ,  n, & n);
}

int  main()
{
    up_down(1);
    return   0;
}
```





有 n 个人排成一队，问第 n 个人多少岁，他回答比前面一个人大 2 岁，再问前面一个人多少岁，他回答比前面一个人大 2 岁，一直问到最后问第一个人，他回答 10 岁

```
int   age( int   n)
{
    int   i;
    if   ( n ==   1)
        i =   10;
    else
        i = age(   n -   1) +   2;
    return   i;
}
```



将 10 进制数转化为二进制数的例子

234 在十进制下为 2 * 10 的 2 次方 + 3 * 10 的 1 次方 + 4*10 的 0 次方。

奇数的二进制最后一位一定是 1，偶数的二进制最后一位一定是 0。

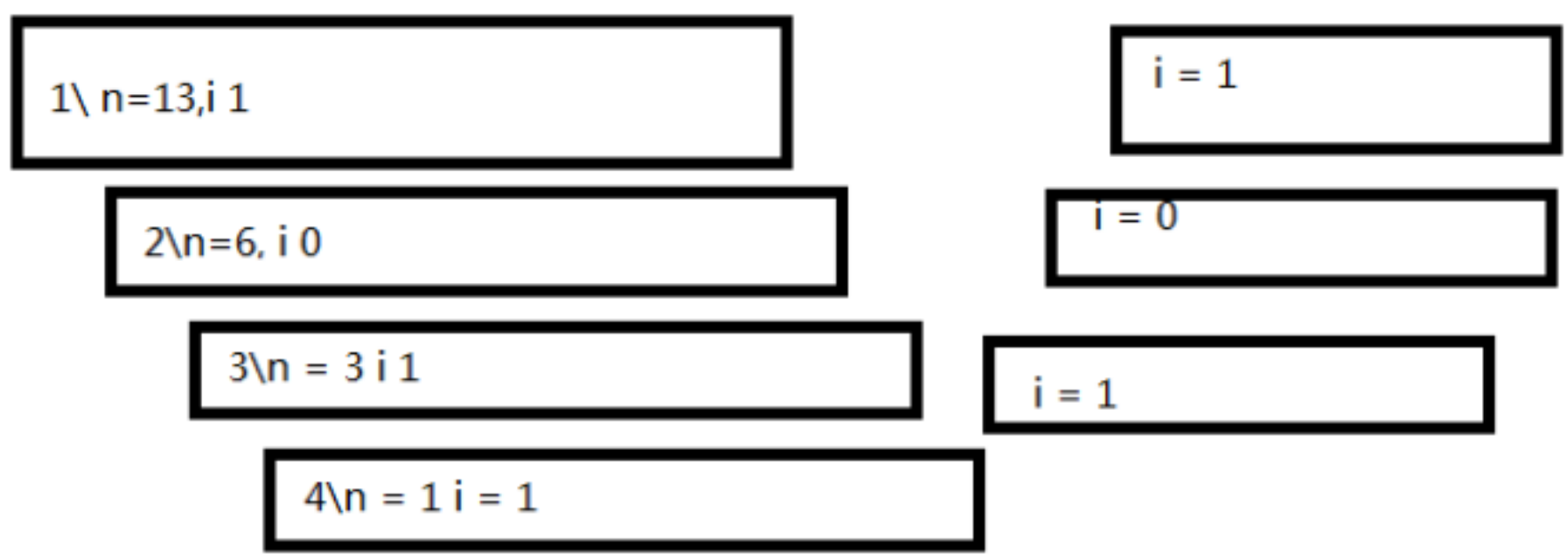
可以通过 number % 2 得到二进制形式的最后一位，如果要将一个完整的整数转化为二进制就需要用到递归函数。

在递归调用之前，计算 number % 2 的值，然后在递归调用语句之后进行输出，这样计算出的第一个数值反而在最后一个输出。

为了得出下一个数，需要把原数除以 2，这种计算相当于十进制下把小数点左移一位，如果此时得出的数是偶数， ，则下一个二进制的数值是 0，如果得出的是奇数，那么下一个二进制数为 1。

直到被 2 除的结果小于 2，就停止递归。

```
void to_binary( unsigned int n)
{
    unsigned int i = n % 2;
    if ( n >= 2)
        to_binary( n / 2);
    printf( "%c", i + 0x30);
}
```



斐波那契数列例子

斐波那契数列指的是这样一个数列 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, ...

第 0 项是 0，第 1 项是第一个 1。

这个数列从第 2 项开始，每一项都等于前两项之和。

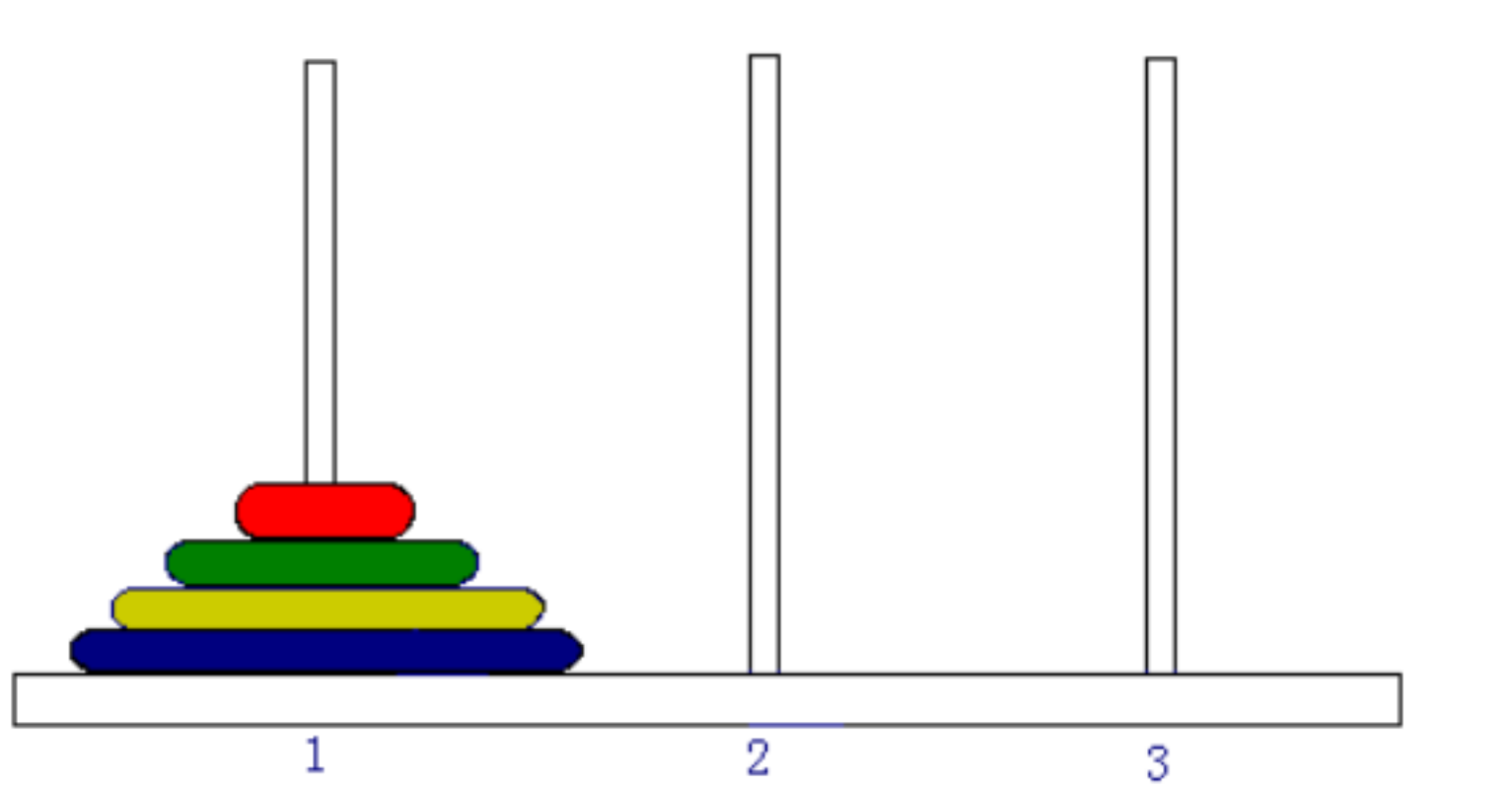
```
int fib( int n)
{
    if ( n == 0)
```

```
        return    0;
    if   ( n ==   1)
        return    1;
    if   ( n >   1)
        return   fib(    n -    1) + fib(     n -    2);
}
```

汉诺塔的例子

有三根针 1,2,3 。1 针上有 4 个盘子，盘子大小不等，大的在下，小的在上，

要求把这 4 个盘子从 1 针一到 3 针，在移动过程中可以借助 2 针，每次只允许移动一个盘子，并且在移动过程中在三根针上都要保持大盘子在下，小盘子在上。



可以分为以下步骤：

1、将 1 针上 n-1 个盘借助 3 针移动到 2 针上；

2、将 1 针上剩下的一个盘移动到 3 针上；

3、将 n-1 个盘从 2 针借助 1 针移动到 3 针上。

```
void  hanoi(  int   n,  int   one,  int   two,  int   three )
{
```

```
    if  ( n ==  1)
        printf(   "%d->%d\n",  one,  three );
    else
    {
        hanoi( n -  1,  one,  three ,  two);
        printf(   "%d->%d\n",  one,  three );
        hanoi( n -  1,  two,  one,  three );
    }
}
```

11.5.2　递归的优点

递归给某些编程问题提供了最简单的方法

11.5.3　递归的缺点

一个有缺陷的递归会很快耗尽计算机的资源，递归的程序难以理解和维护。

11.6 多个源代码文件程序的编译

11.6.1　头文件的使用

如果把　main　函数放在第一个文件中，而把自定义函数放在第二个文件中，那么就需要在第一个文件中声明函数原型。

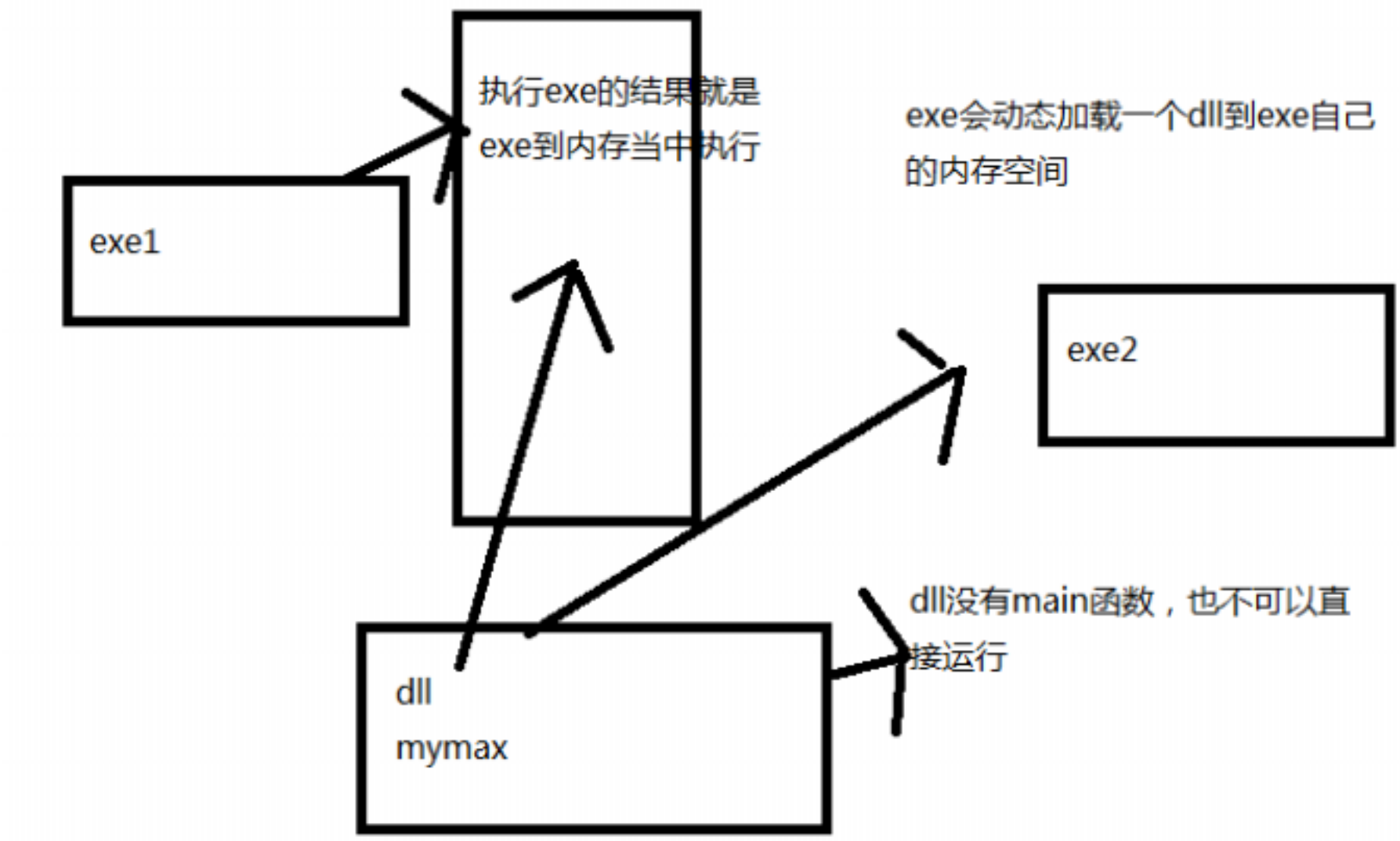
如果把函数原型包含在一个头文件里，那么就不必每次使用函数的时候都声明其原型了。把函数声明放入头文件是很好的习惯。

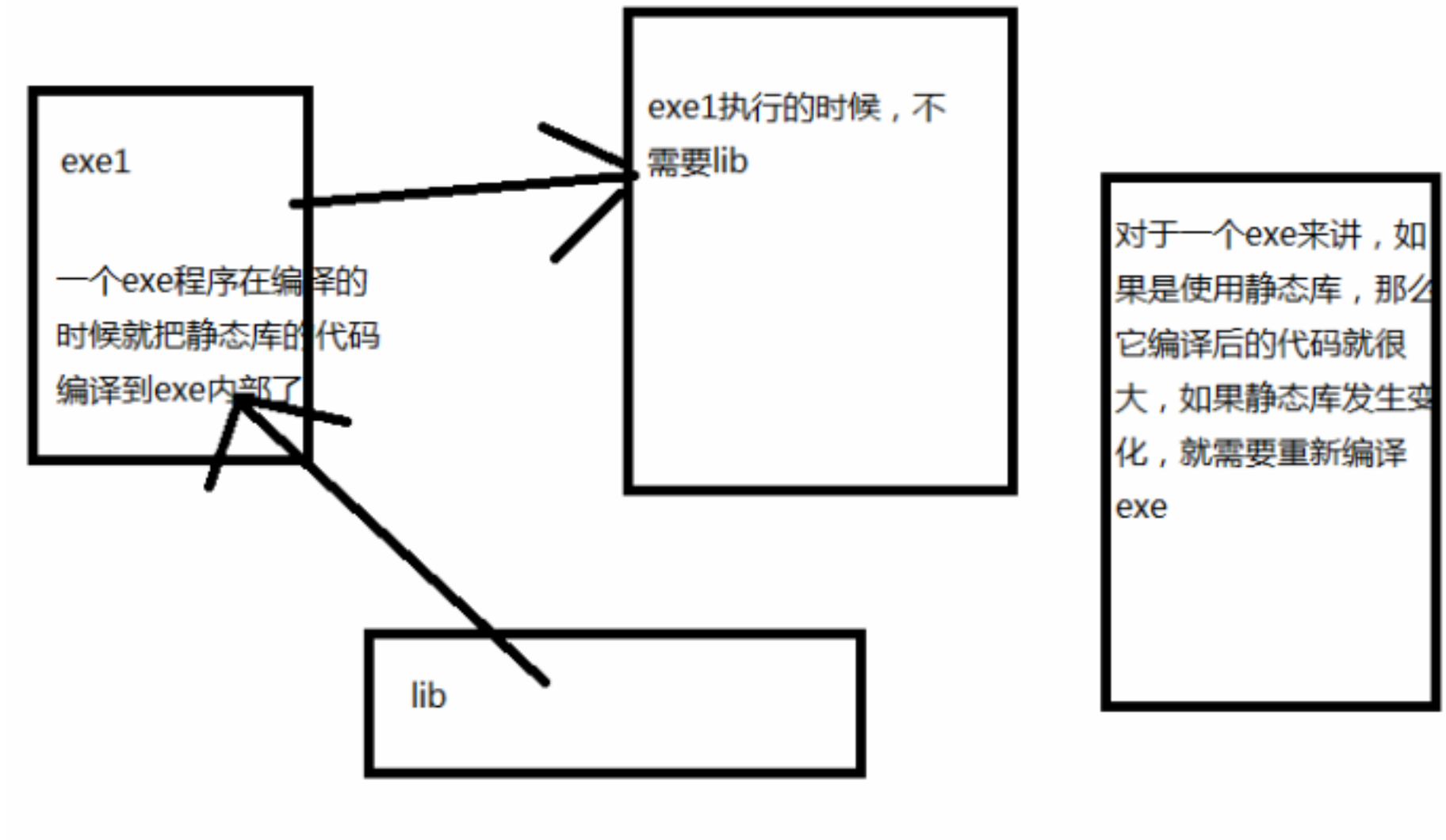
11.6.2　#include　与#define　的意义

11.6.3　#ifndef　与#endif

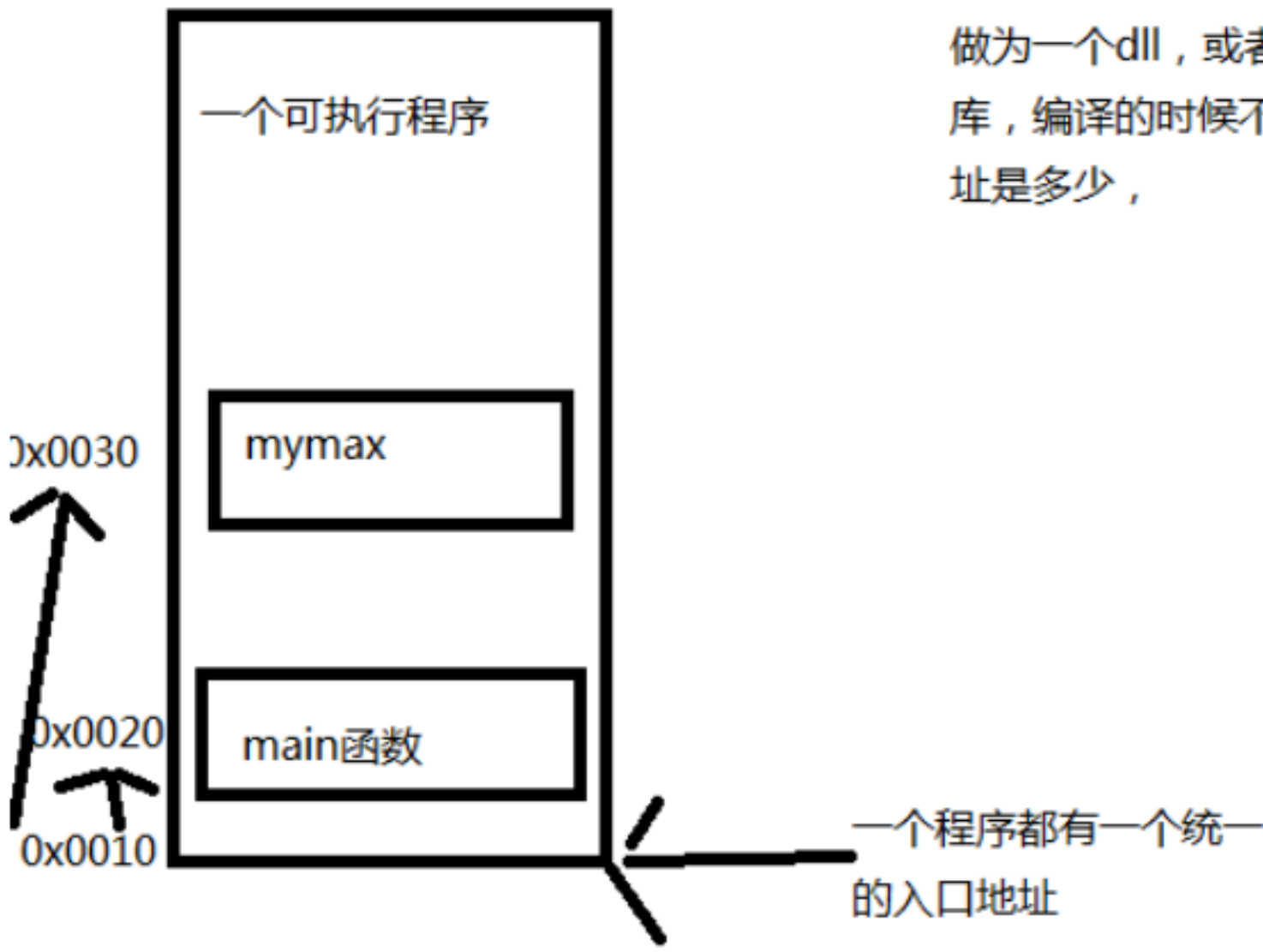
11.7 函数的二进制封装

11.7.1 exe 加载 dll 的说明





11.7.2 动态库中代码与位置无关的说明图



11.7.3 linux 编写 so 文件的方式

1 首先 gcc 编译的时候要加 -fPIC 选项， -fPIC 是告诉 gcc 生成一个与位置无关的代码

2gcc 链接的时候要加 -shared 选项，意思是生成一个 so 共享库。

对于 linux 或者 unix，一个 so 文件，文件扩展名必须是 so，文件名的前三个字母必须是 lib

11.7.4 linux 使用 so

gcc 链接的时候需要加 -L. 代表从当前目录下找相关的 so 文件， -l 文件名（但不包括文件名开头的 lib 和扩展名 so）

例如编译一个 main.o 文件，要用到当前目录下的 liba.so 库

gcc –o main.out –L. –la main.o

11.7.5 配置 profile 文件可以在当前目录下查找 so 文件

linux 不在当前目录下寻找可执行程序，同时也不早当前目录下找 so 库文件

修改用户配置文件的方法

1

cd

2

vi .bash_profile

3

export LD_LIBRARY_PATH = $LD_LIBRARY_PATH:.

4

保存退出

5

. .bash_profile

11.8 作业描述

int sum1(int n);//n = 10 0,1,2,3,4,5,6,7,8,9,10 的和

要求不可以用循环，只可以用递归

int sum2(int n);//n = 10 从 0 到 n 范围内所有素数的和

要求不可以用循环，只可以用递归

12 指针

重点： *****

12.1 指针

12.1.1 指针的概念

指针也是一个变量 ,做为指针变量的值是另一个变量的地址。

指针存放的内容是一个地址，该地址指向一块内存空间

12.1.2 指针变量的定义

可以定义一个指向一个变量的指针变量。

int *p;// 表示定义一个指针变量。

*p;// 代表指针所指内存的实际数据

切记，指针变量只能存放地址，不能将一个 int 型变量直接赋值给一个指针。

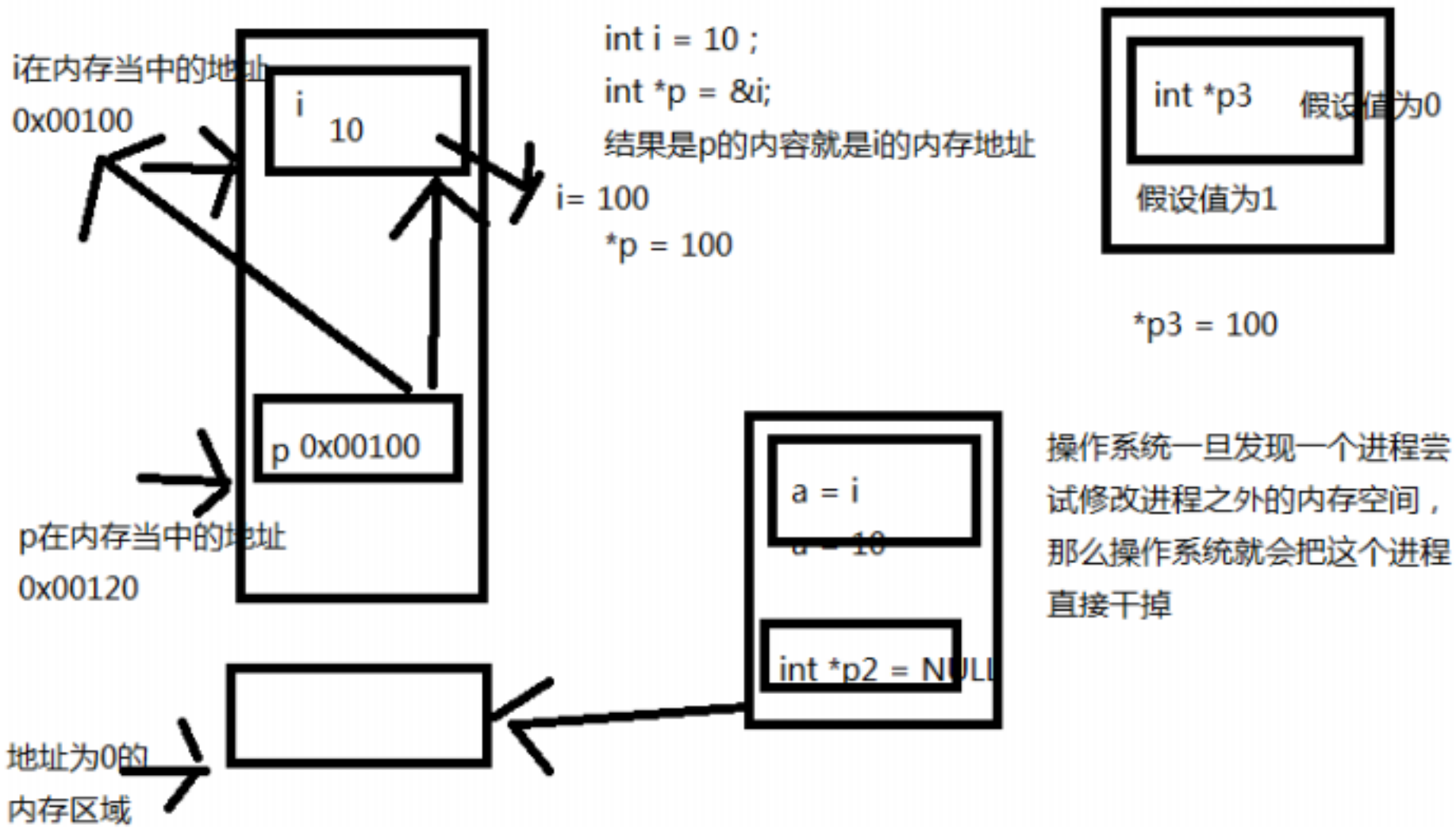
int *p = 100;

12.1.3 NULL

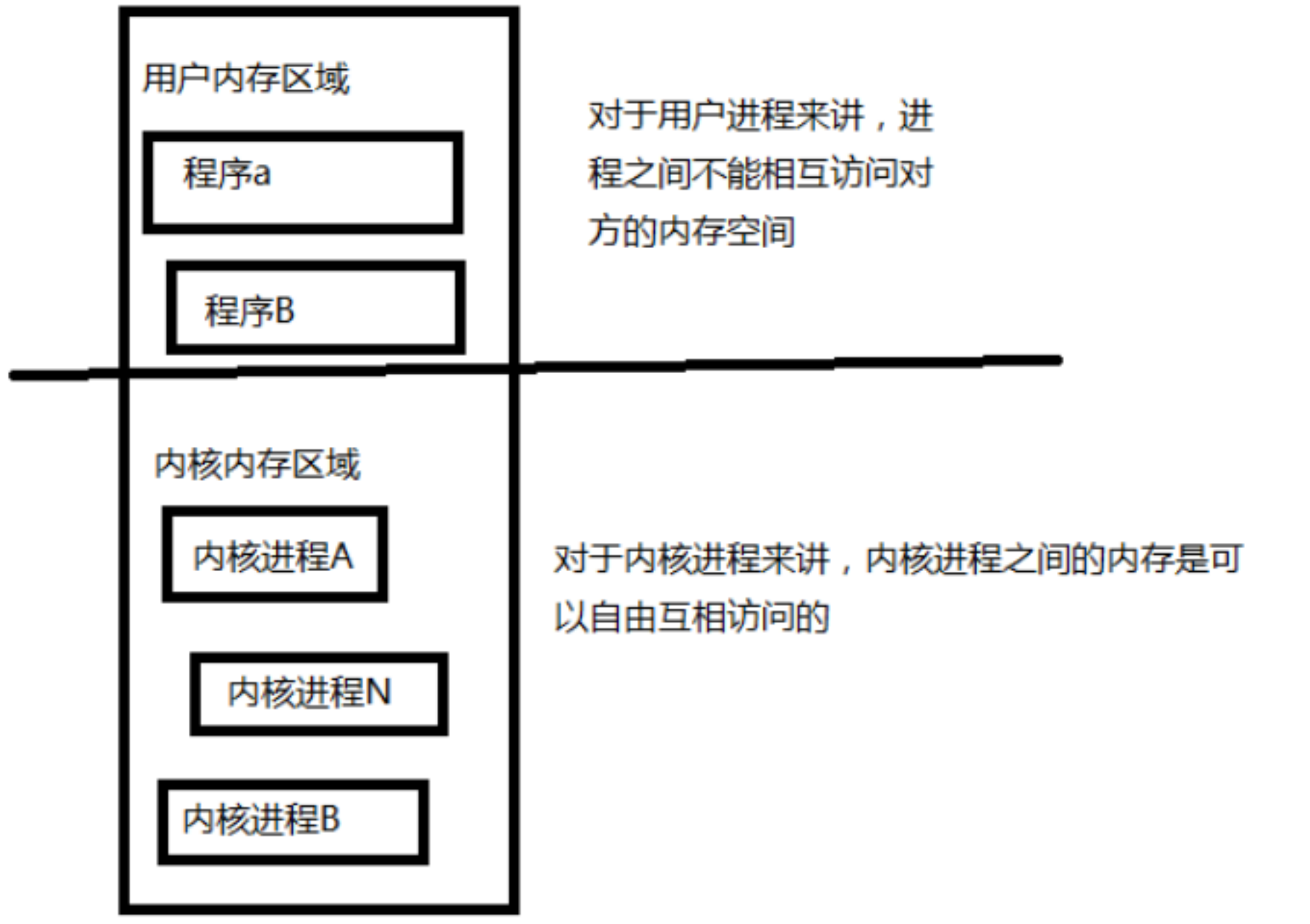
一个指向 NULL 的指针，我们称之为空指针，意味着这个指针不指向任何一个变量。

12.1.4 野指针

定义之后没有初始化值的指针


i在内存当中的地址
0x00100
i
10
int i = 10 ;
int *p = &i;
结果是p的内容就是i的内存地址
i= 100
*p = 100
p 0x00100
p在内存当中的地址
0x00120
int *p3
假设值为0
假设值为1
*p3 = 100
a = i
a = 10
int *p2 = NULL
操作系统一旦发现一个进程尝试修改进程之外的内存空间，那么操作系统就会把这个进程直接干掉
地址为0的
内存区域

12.1.5 & 取地址运算符


用户内存区域
程序a
程序B
对于用户进程来讲，进程之间不能相互访问对方的内存空间
内核内存区域
内核进程A
内核进程N
内核进程B
对于内核进程来讲，内核进程之间的内存是可以自由互相访问的

12.1.6 无类型指针

定义一个指针变量，但不指定它指向具体哪种数据类型。可以通过强制转化将 void * 转化为其他类型指针，也可以用 (void *) 将其他类型指针强制转化为 void 类型指针。

void *p

在 C 语言当中，可以将任何一种地址赋值给 void * 指针

12.1.7 指针的兼容性

指针之间赋值比普通数据类型赋值检查更为严格，例如：不可以把一个 double * 赋值给 int *

12.1.8 指针与数组的关系

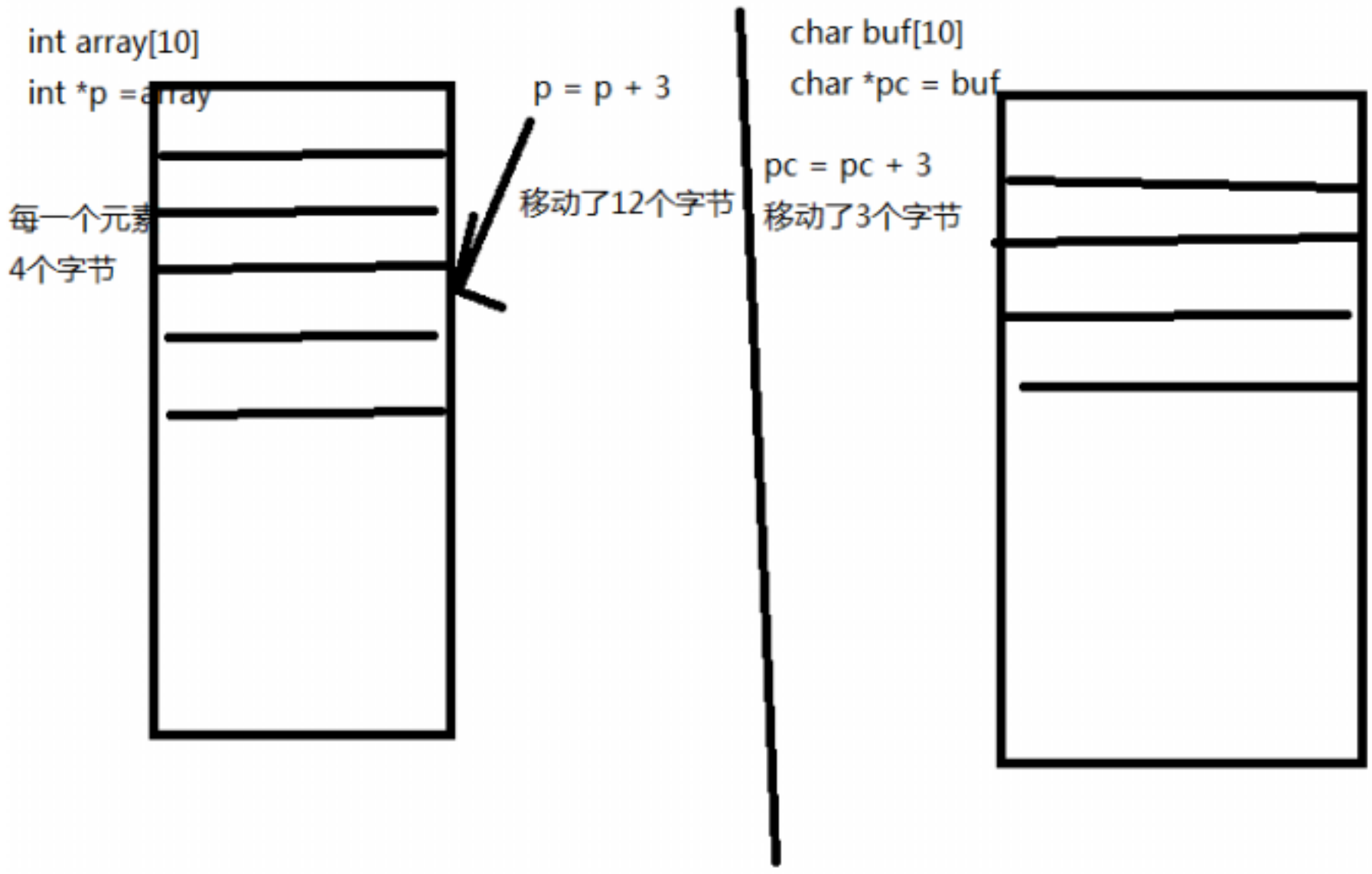
一个变量有地址，一个数组包含若干个元素，每个元素在内存中都有地址。

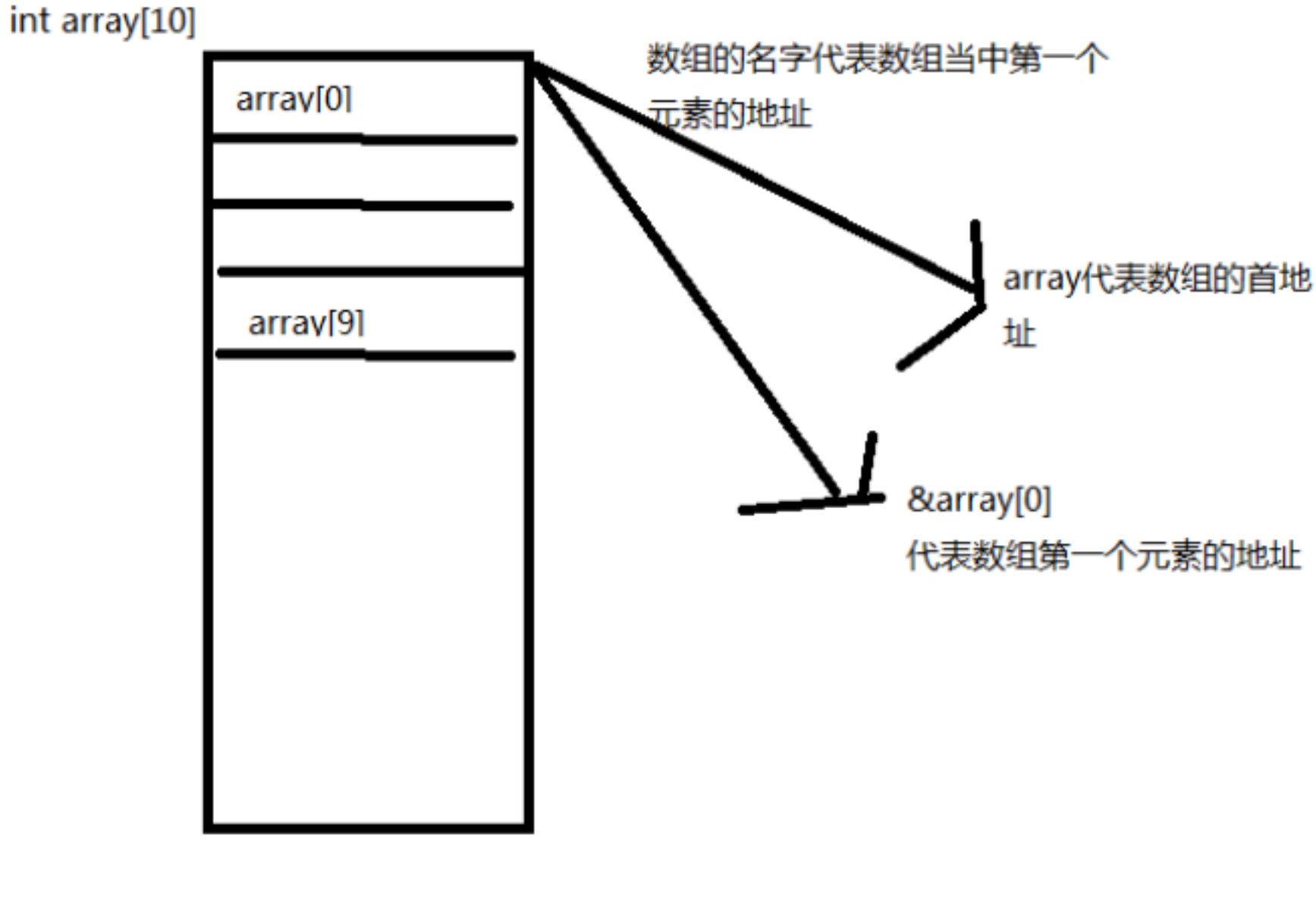
int a[10];

int *p = a;

比较 p 和&a[0] 的地址是否相同

在 C 语言当中数组的名称代表数组的首地址，如果取数组名称的地址，C 语言认为就是取数组的首地址。





12.1.9　通过指针使用数组元素

通过指针 计算，不是把指针当做一个整数，计算结果，而是指针在内存当中移动

p + 1 代表 &a[1]，也可以直接使用 p[1] 表示 a[5]

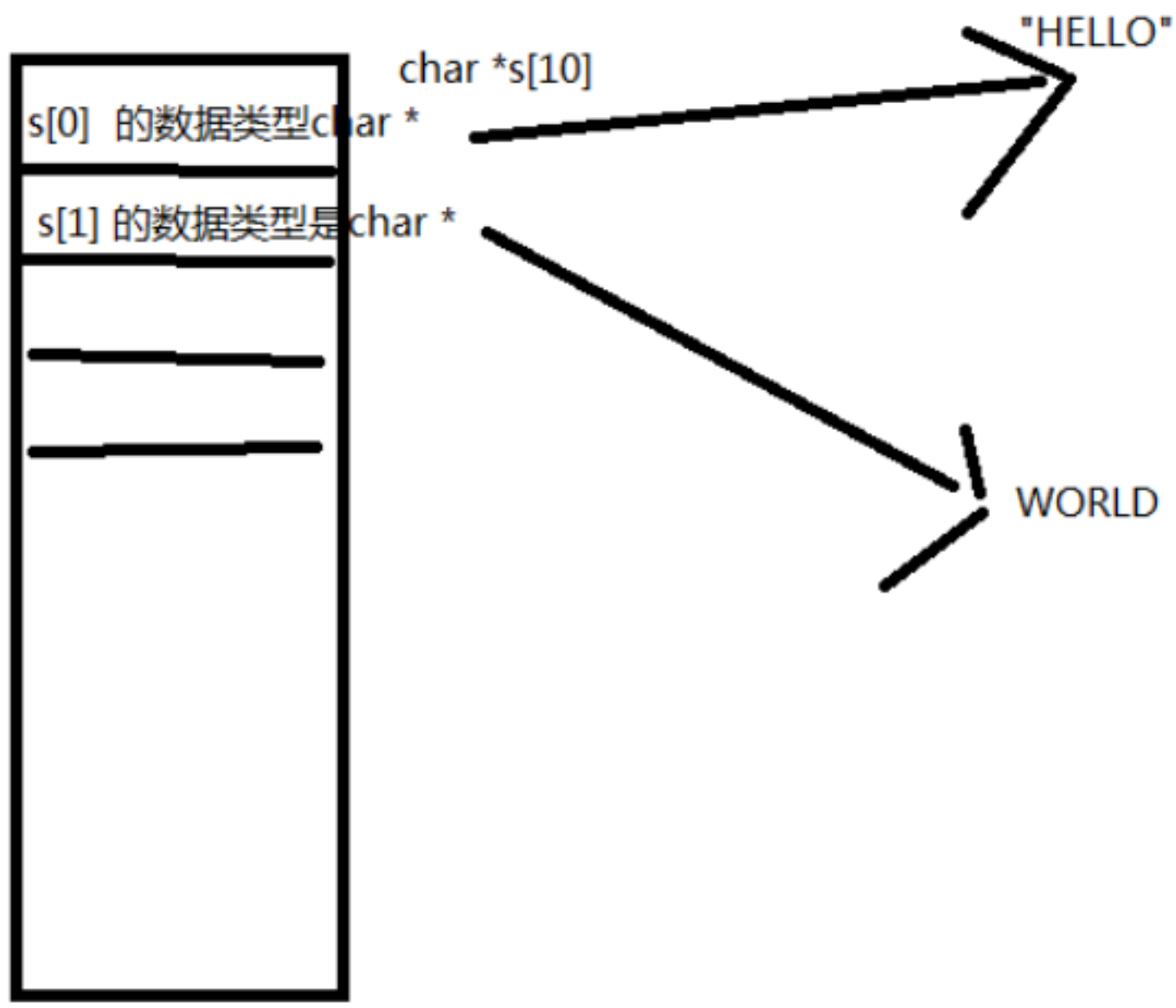
p + 5 代表 &a[5]

p++

在 C 语言里面数组名称是个常量，值是不可改变的

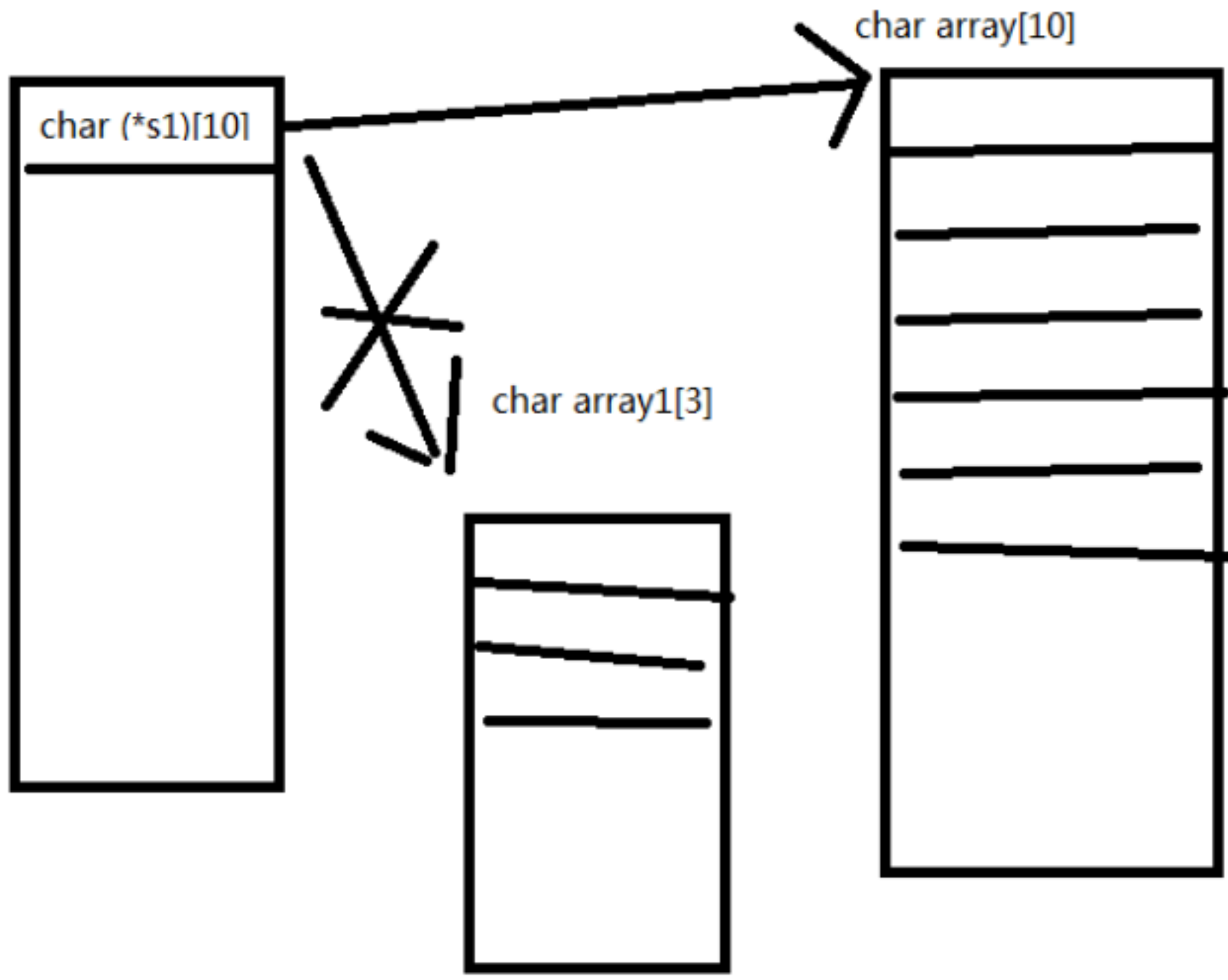
12.1.10 指针数组

int *p[5];



12.1.11 数组指针

int (*P)[5];

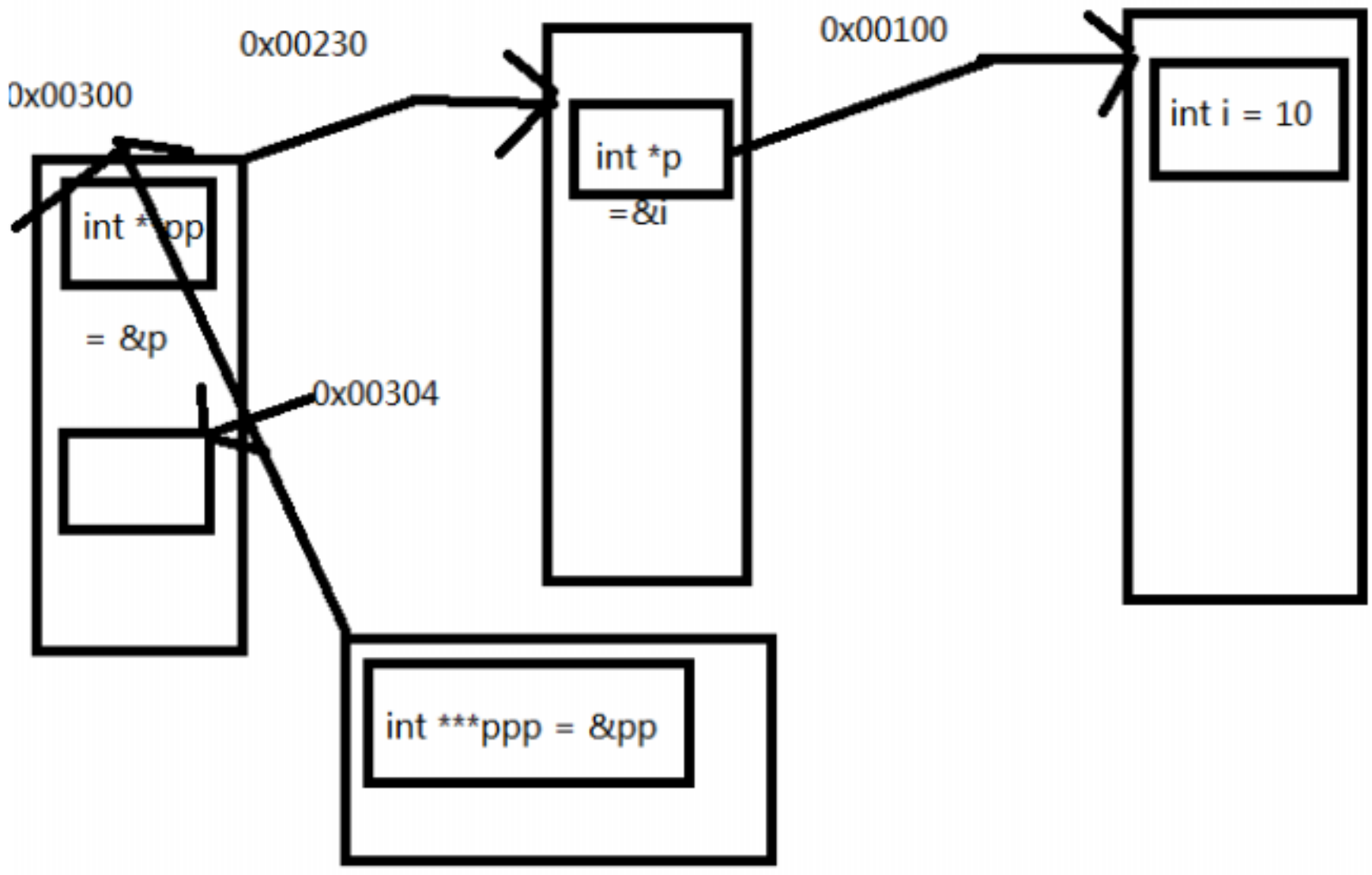


12.1.12 指向指针的指针（二级指针）

指针就是一个变量，既然是变量就也存在内存地址，所以可以定义一个指向指针的指针。

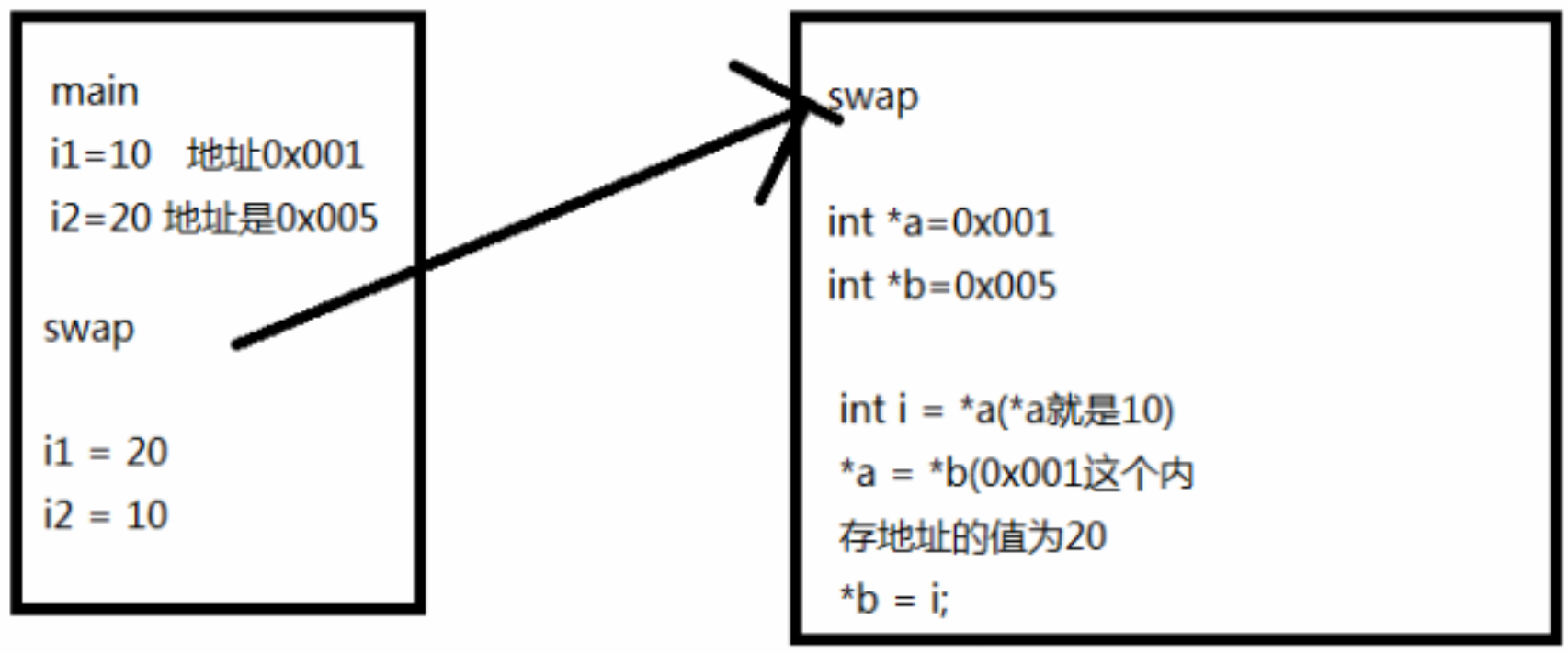
```
        int    i   =  10;
int     *p1   =  &i;
        int    **p2   =  &p1;
        printf(    "%d\n" ,   **p2);
```

以此类推可以定义　3 级甚至多级指针。



12.1.13 指针变量做为函数的参数

函数的参数可以是指针类型。 ，它的作用是将一个变量的地址传送给另一个函数。



正常情况下C语言函数的实参时不能通过形参修改的
但可以通过将实参的地址传递给函数的参数来达到修
改实参的目的

12.1.14 一维数组名作为函数参数

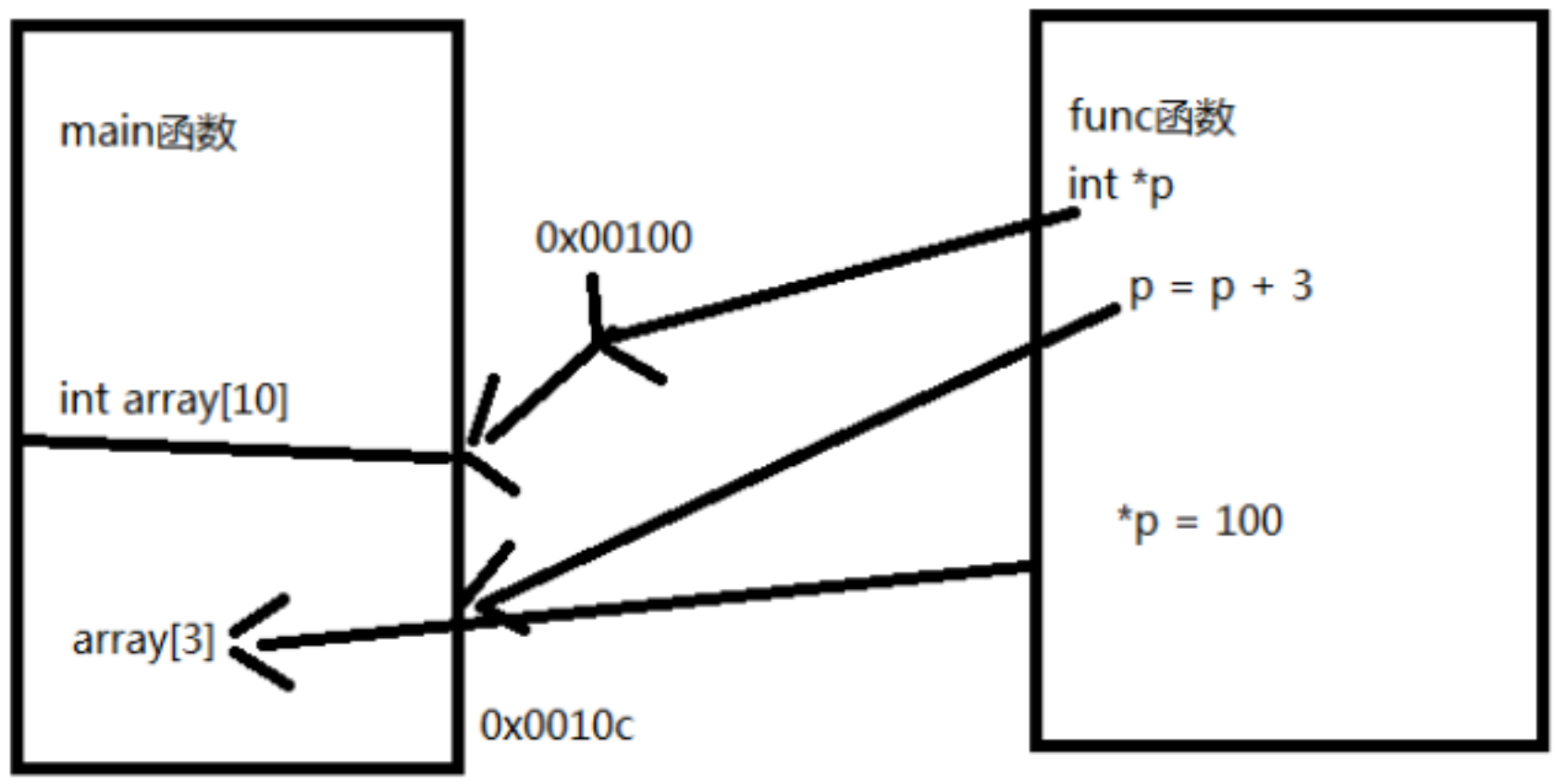
当数组名作为函数参数时，　　C 语言将数组名解释为指针

当数组名作为函数参数传递给被调用函数时，被调用函数是不知道数组有多少元素的

int func(int array[10]);

相当于传递是一个地址，那么就可以通过地址来修改实参的值。

只要传递是数组名，那么形参一定可以通过地址修改实参的值。



12.1.15 二维数组名作为函数参数

二维数组做函数参数时可以不指定第一个下标。

int func(int array[][10]);

12.1.16 指向二维数组的指针

int a[3][5]

a	二维数组名称，数组首地址

a[0], *(a + 0), *a	0 行， 0 列元素地址
a + 1	第 1 行首地址
a[1], *(a + 1)	第 1 行， 0 列元素地址
a[1] + 2, *(a + 1) + 2, &a[1][2]	第 1 行， 2 列元素地址
*(a[1] + 2), *(*(a + 1) + 2), a[1][2]	第 1 行， 2 列元素的值

12.1.17 指向常量的指针与指针常量

const char *p;// 定义一个指向常量的指针

char *const p;// 定义一个指针常量，一旦初始化之后其内容不可改变

12.1.18 const 关键字保护数组内容

如果将一个数组做为函数的形参传递，那么数组内容可以在被调用函数内部修改，有时候不希望这样的事情发生，所以要对形参采用 const 参数

func(const int array[])

12.1.19 指针做为函数的返回值

char *func();// 返回值为 char * 类型的函数

12.1.20 指向函数的指针

指针可以指向变量，数组，也可以指向一个函数。

一个函数在编译的时候会分配一个入口地址，这个入口地址就是函数的指针，函数名称就代表函数的入口地址。

函数指针的定义方式： int (*p)(int);// 定义了一个指向 int func(int n) 类型函数地址的指针。

1 定义函数指针变量的形式为：函数返回类型 (* 指针变量名称)(参数列表)

2 函数可以通过函数指针调用

3int(* P)() 代表指向一个函数，但不是固定哪一个函数。

```
void  man()
{
    printf(    "抽烟\n" );
```

```
        printf(    "喝酒\n" );
        printf(    "打牌\n" );
}

void  woman()
{
        printf(    "化妆\n" );
        printf(    "逛街\n" );
        printf(    "网购\n" );
}

int   main()
{
        void (*p)();
        int   i =     0;
        scanf(  "%d", &i);
        if   (i ==     0)
              p = man;
        else
              p = woman;
        p();
        return    0;
}
```

12.1.21 把指向函数的指针做为函数的参数

将函数指针做为另一个函数的参数称为回调函数

```
int   max( int    a,   int    b)
{
        if   ( a >   b)
              return    a;
        else
              return    b;
}

int   add( int    a,   int    b)
{
        return    a +   b;
}
```

```
void  func(   int  (* p)( int  ,   int  ),   int    a,   int    b)
{
      int   res =     p( a,   b);
      printf(    "%d\n" , res);

}

int   main()
{
      int    i =      0;
      scanf(  "%d", &i);
      if   (i ==     0)
            func(max,    10,   20);
      else
            func(add,    10,   20);
      return    0;
}
```

12.1.22 指针运算

赋值： int *p = &a;

求值： int I = *p;

取指针地址　int **pp = &p;

将一个整数加 (减)给指针 :p + 3; p　−3;

增加（减少）指针值　p++,p--

求差值　,p1 −p2, 通常用于同一个数组内求两个元素之间的距离

比较　p1 == p2,　通常用来比较两个指针是否指向同一个位置。

12.1.23 指针小结

定义	说明
Int i	定义整形变量
int *p	定义一个指向 int 的指针变量
Int a[10]	定义一个 int 数组
Int *p[10]	定义一个指针数组，其中每个数组元素指向一个 int 型变量的地址
Int func()	定义一个函数，返回值为 int 型
Int *func()	定义一个函数，返回值为 int * 型
Int (*p)()	定义一个指向函数的指针，函数的原型为无参数，返回值为 int
Int **p	定义一个指向 int 的指针的指针，二级指针

13 字符指针与字符串

重点： ****

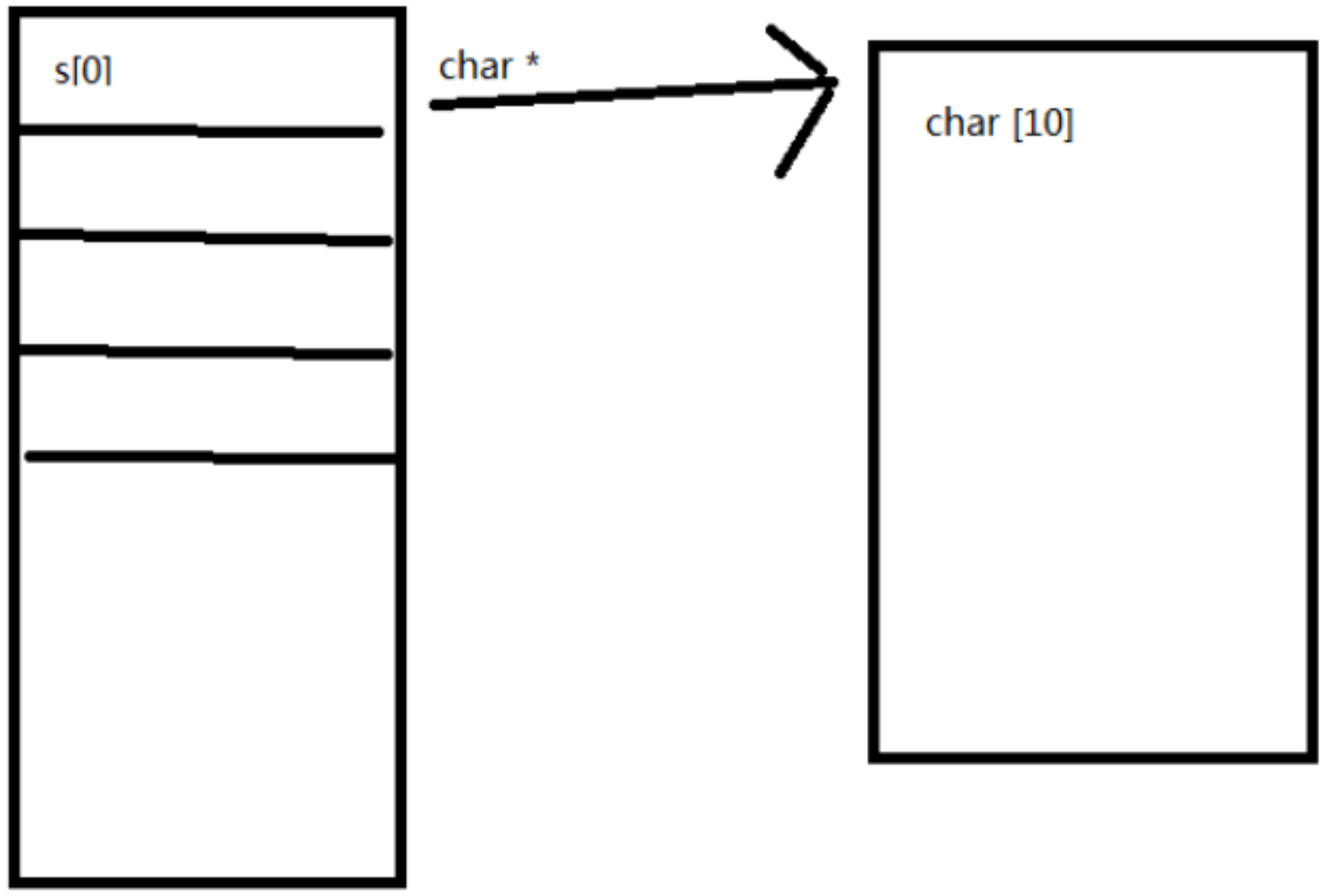
13.1 指针和字符串

在 C 语言当中，大多数字符串操作其实就是指针操作。

```
char  s[] =      "hello word"     ;
char  *p = s;
p[ 0] =    'a'   ;
```

13.2 通过指针访问字符串数组

13.3 函数的参数为 char *

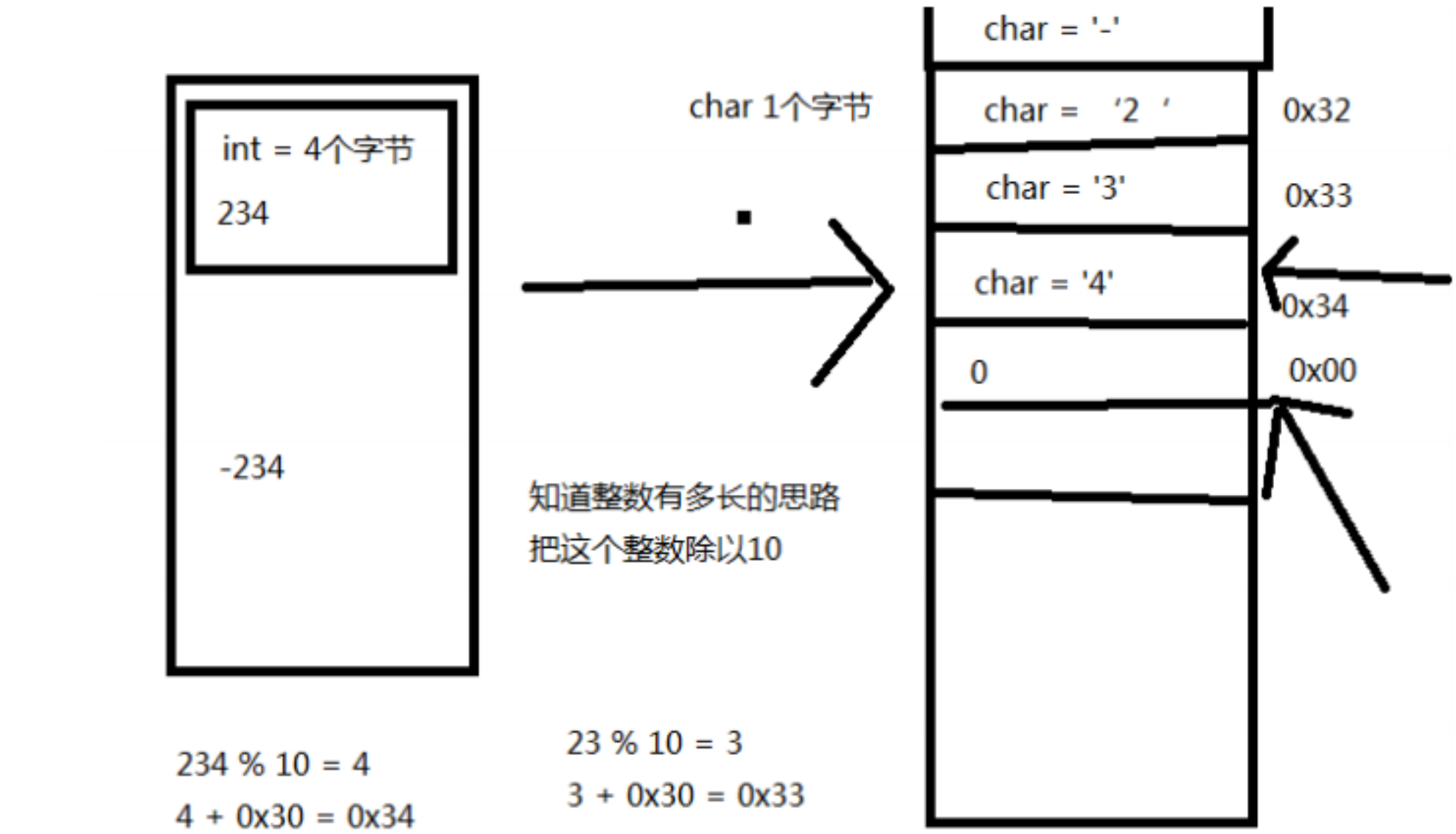


13.4 指针数组做为 main 函数的形参

```
int main(int argc, char *argv[]);
```

作业

```
// 不可以使用任何 C语言库函数，只可以自己写函数实现 int 到字符串的转化
void itoa( int i , char * s)
{
}
```



14 内存管理

重点： *****

14.1 作用域

一个 C 语言变量的作用域可以是代码块 作用域，函数作用域或者文件作用域。

代码块是 {} 之间的一段代码。

14.1.1 auto 自动变量

一般情况下代码块内部定义的变量都是自动变量。当然也可以显示的使用 aotu 关键字

14.1.2 register 寄存器变量

通常变量在内存当中，如果能把变量放到 CPU 的寄存器里面，代码执行效率会更高

register int I;

14.1.3 代码块作用域的静态变量

静态变量是指内存位置在程序执行期间一直不改变的变量，一个代码块内部的静态变量只

能被这个代码块内部访问。

14.1.4 代码块作用域外的静态变量

代码块之外的静态变量在程序执行期间一直存在，但只能被定义这个变量的文件访问

14.1.5 全局变量

全局变量的存储方式和静态变量相同，但可以被多个文件访问

14.1.6 外部变量与 extern 关键字

extern int I;

14.1.7 全局函数和静态函数

在 C 语言中函数默认都是全局的，使用关键字 static 可以将函数声明为静态

14.2 内存四区

14.2.1 代码区

代码区 code，程序被操作系统加载到内存的时候，所有的可执行代码都加载到代码区，也

叫代码段，这块内存是不可以在运行期间修改的。

14.2.2 静态区

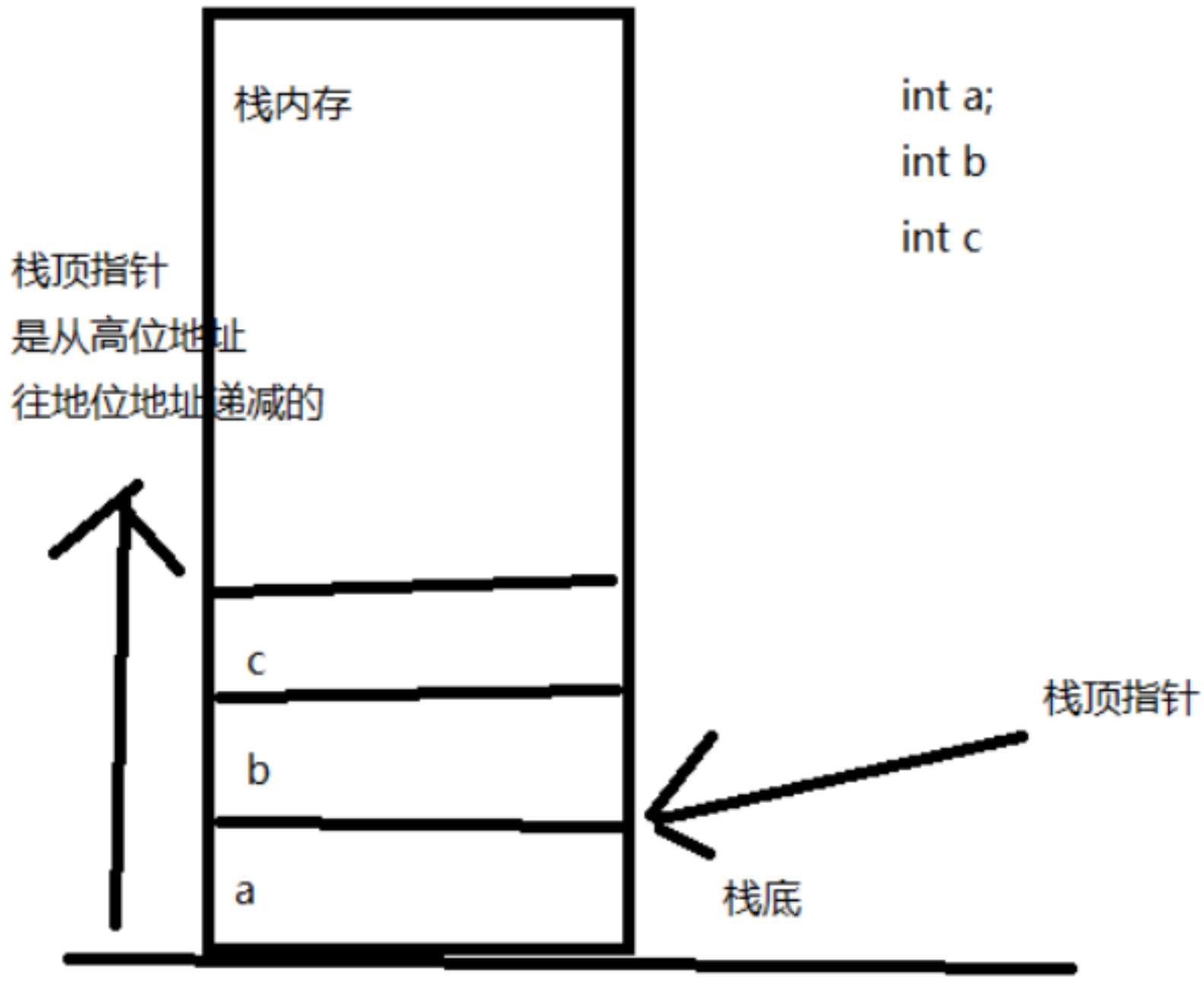
所有的全局变量以及程序中的静态变量都存储到静态区 ,比较如下两段代码的区别

```
int a = 0;
int main()
{
    static int b = 0;
    printf( "%p, %p\n" , &a, &b);
    return 0;
}
```

```
int a = 0;
static int b = 0;
int main()
{
    printf( "%p, %p\n" , &a, &b);
    return 0;
}
```

14.2.3 栈区

栈 stack 是一种先进后出的内存结构，所有的自动变量，函数的形参都是由编译器自动放出栈中，当一个自动变量超出其作用域时，自动从栈中弹出。



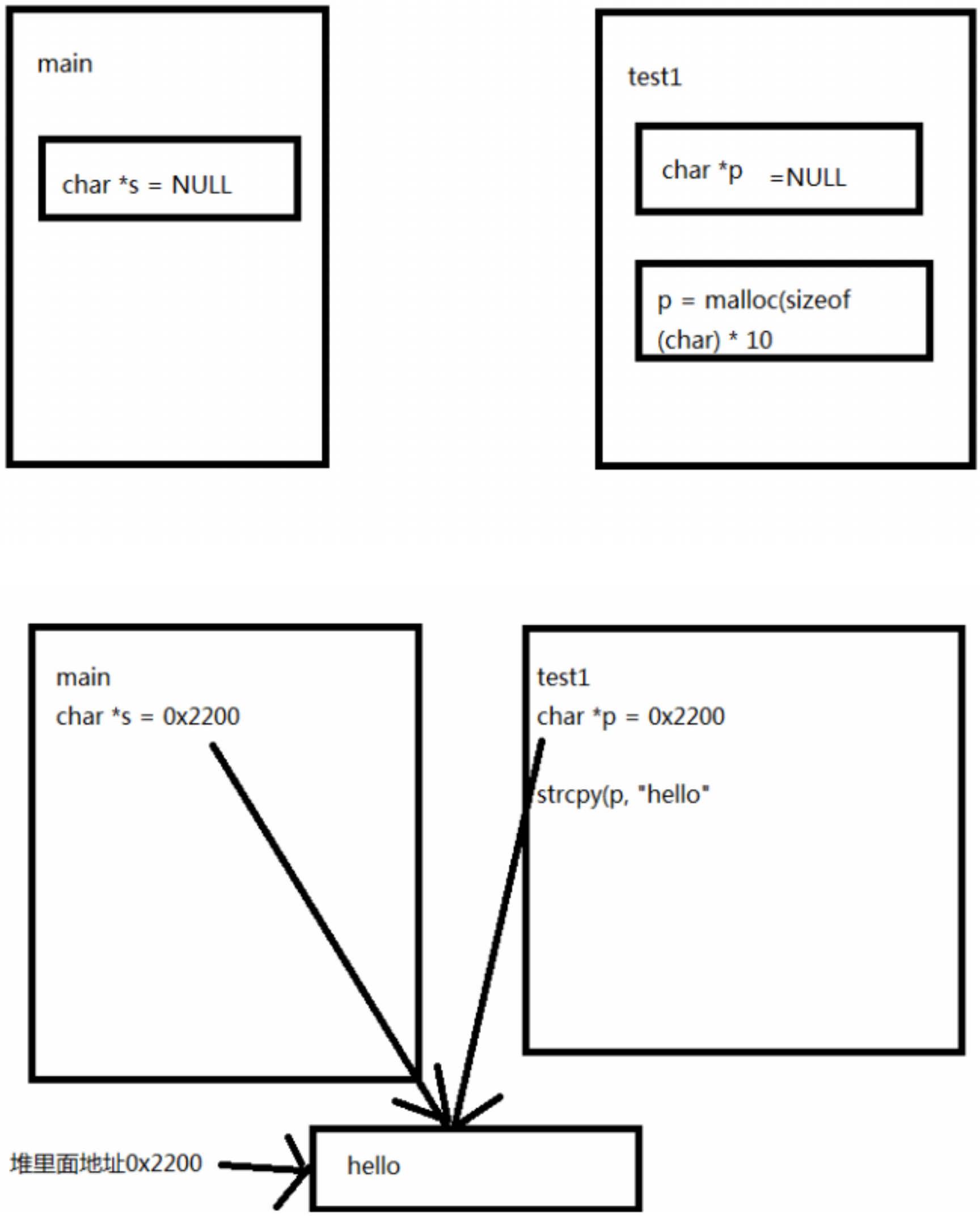
14.2.4 堆区

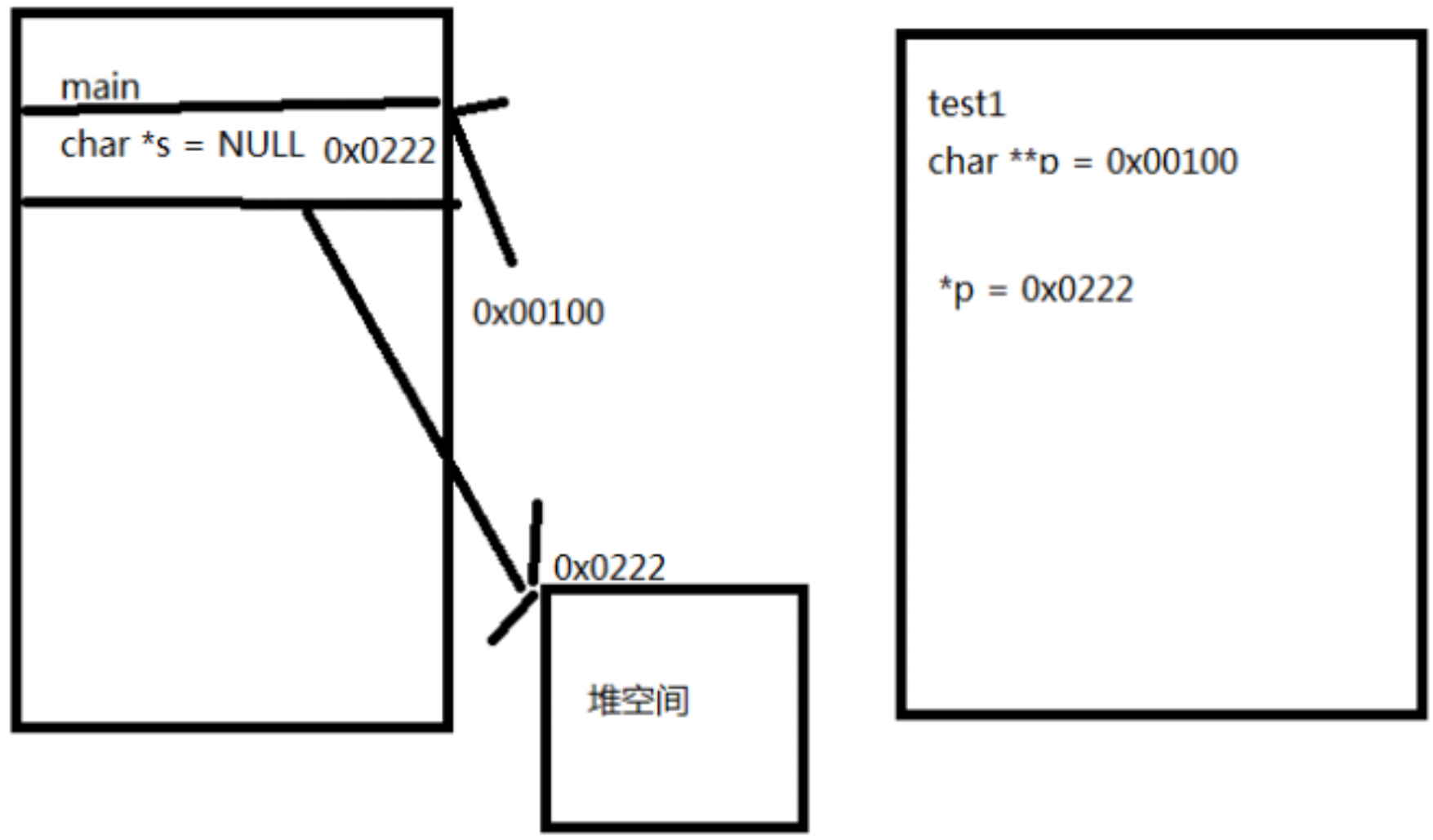
堆 heap 和栈一样，也是一种在程序运行过程中可以随时修改的内存区域，但没有栈那样先进后出的顺序。

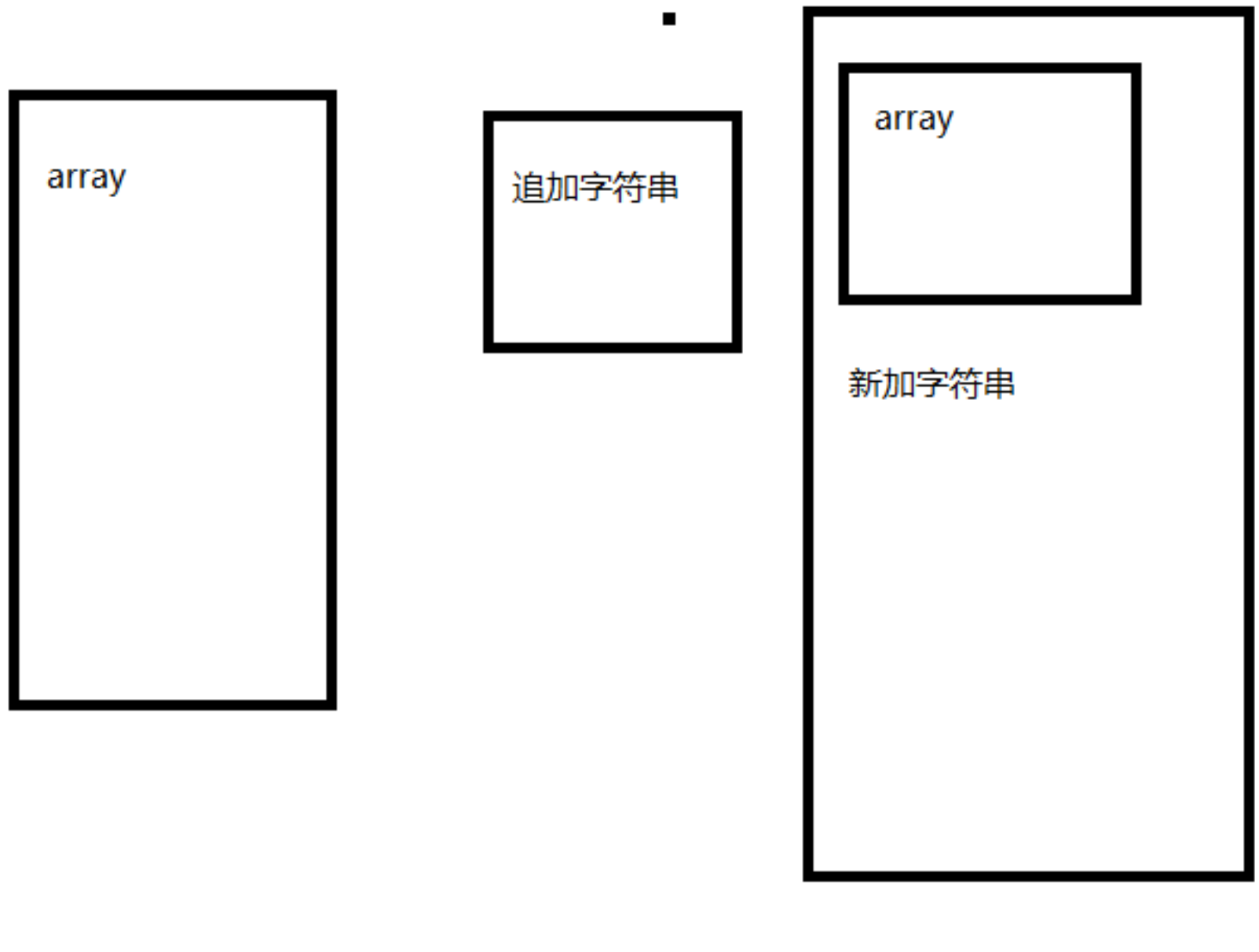
堆是一个大容器，它的容量要远远大于栈，但是在 C 语言中，堆内存空间的申请和释放需要手动通过代码来完成。

14.3 堆的分配和释放

14.3.1 malloc




main
char *s = NULL 0x0222
0x00100
0x0222
堆空间
test1
char **p = 0x00100
*p = 0x0222


array
追加字符串
array
新加字符串

14.3.2 free

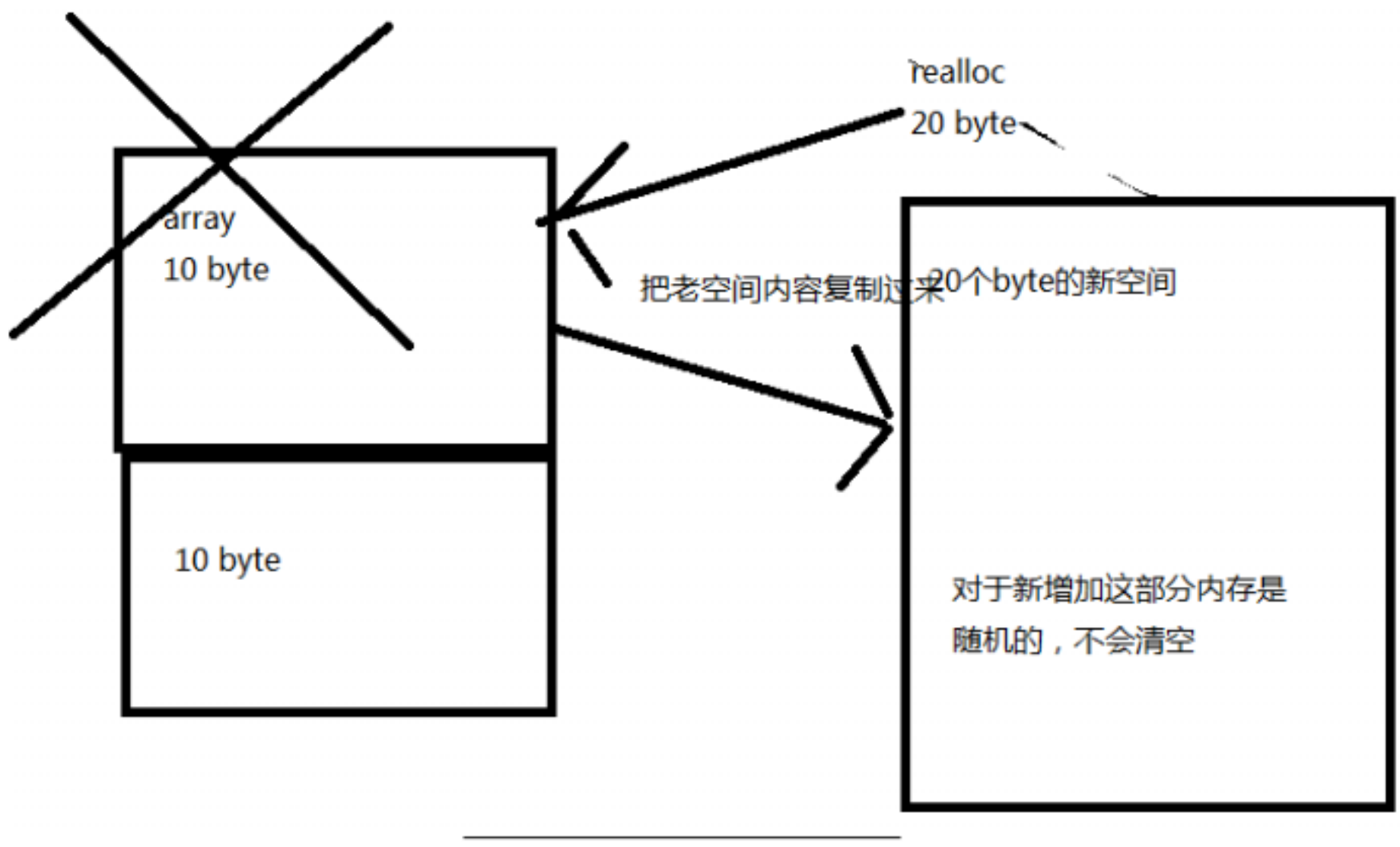
14.3.3 calloc:

第一个参数是所需内存单元数量，　第二个参数是每个内存单元的大小　（单位：字节），calloc 自动将分配的内存置　0

```
int *p = ( int *)calloc( 100, sizeof ( int )); // 分配 100 个 int
```

14.3.4 realloc

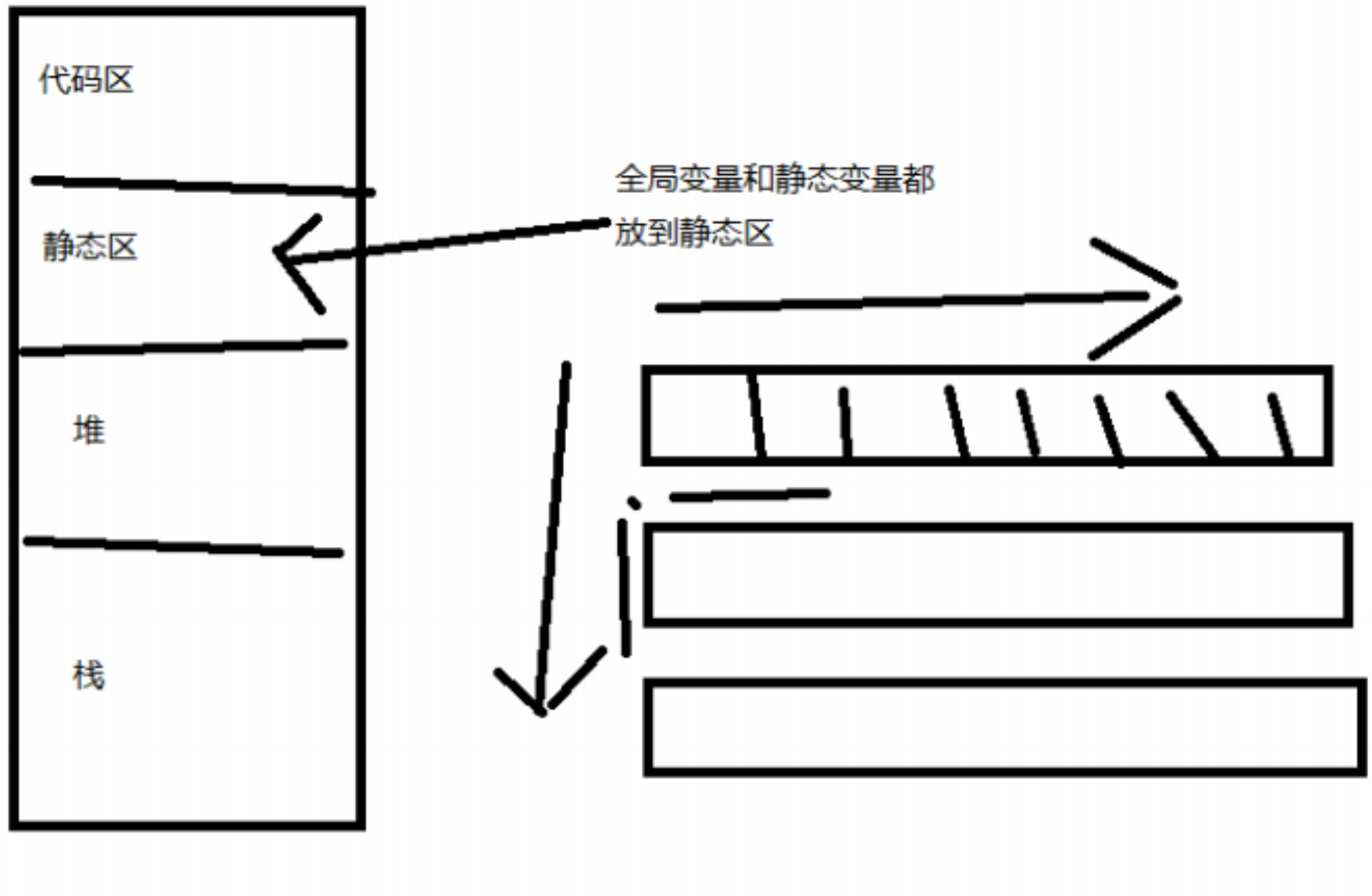
重新　分配大小。



在 C 语言当中，内存初始化是个好习惯，

除了全局变量和静态变量以外，其他变量　　C 语言编译不会帮你初始化，只有自己通过显示的初始化。


堆
操作系统会将物理内存
地址转化为虚拟内存地
址
物理的硬件内存条，
或者磁盘的虚拟内存
物理内存页越大，使用率
就越低


代码区
静态区
堆
栈
全局变量和静态变量都
放到静态区

15 结构体，联合体，枚举与　typedef

重点： ****

15.1 结构体

15.1.1　定义结构体　struct　和初始化

```
struct     man
{
     char  name[ 100];
     int   age;
};
struct     manm = {     "tom" ,   12 };

struct     manm = { .name =      "tom" , .age =      12 };
```

15.1.2　访问结构体成员

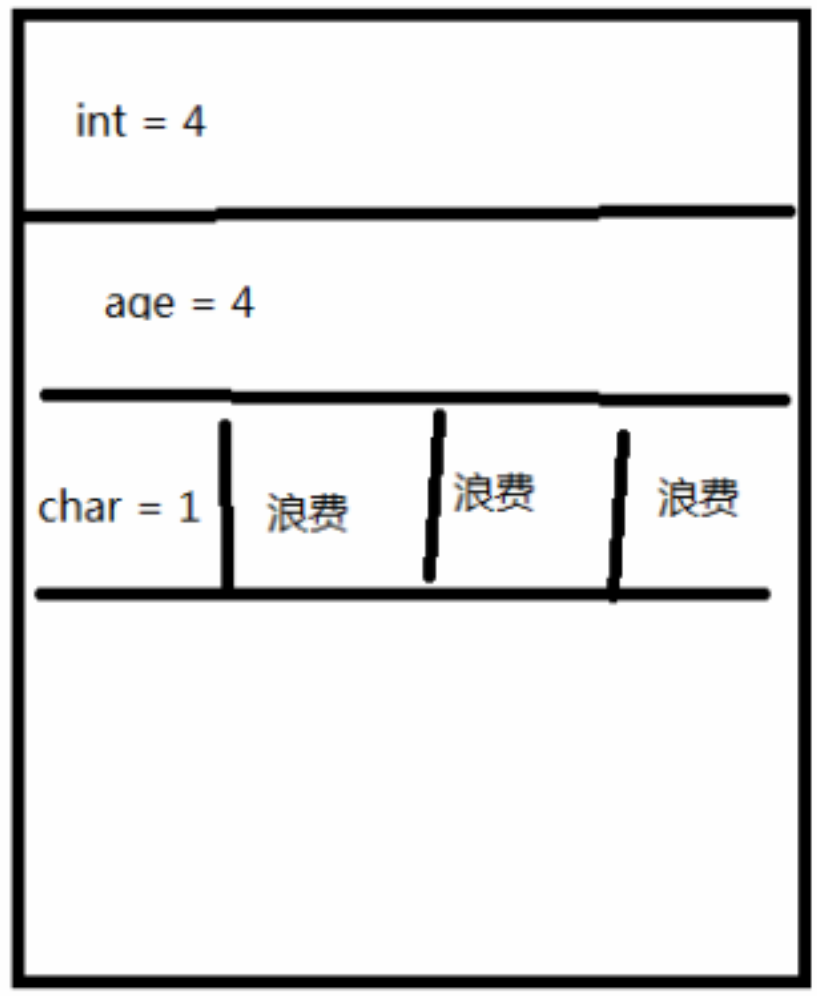
.操作符

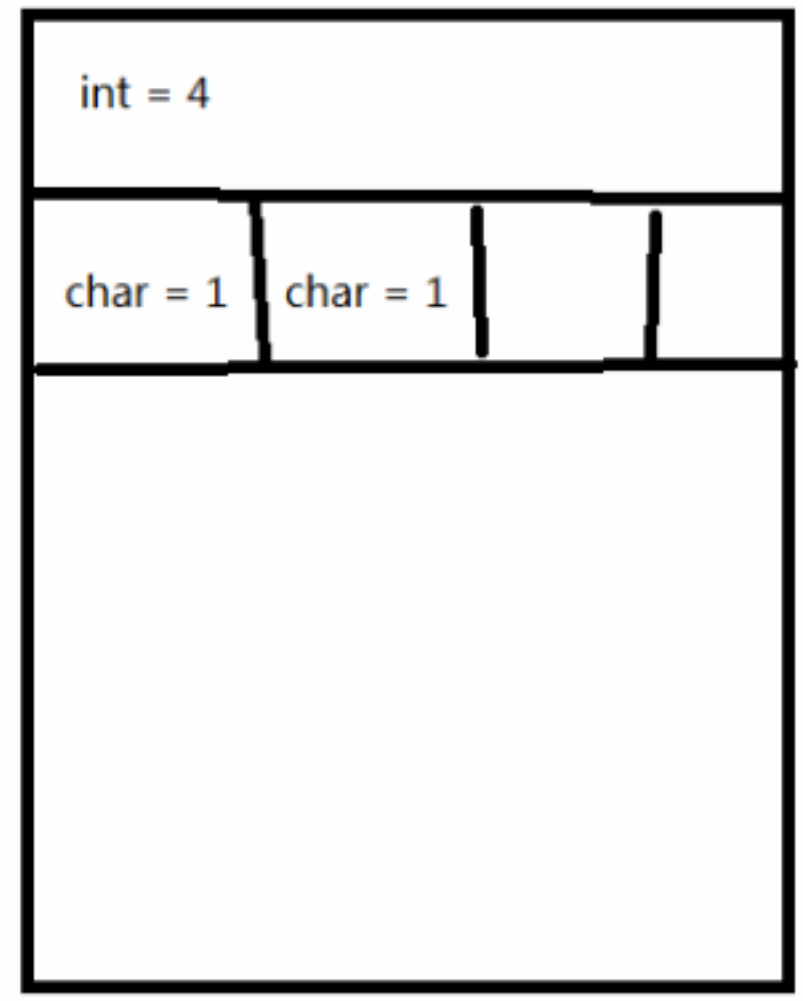
15.1.3　结构体的内存对齐模式

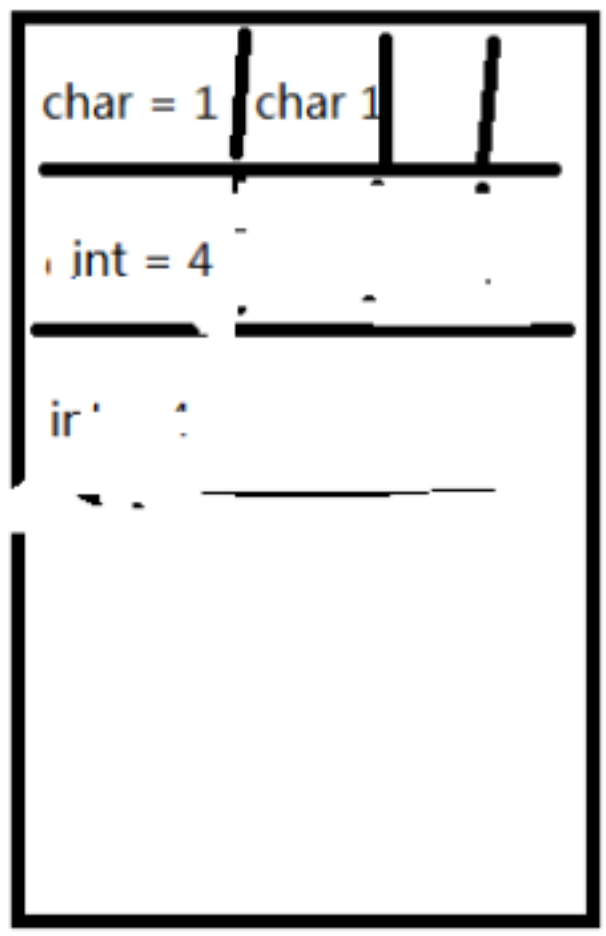
结构在内存的大小是和结构成员最长的那个元素相关的

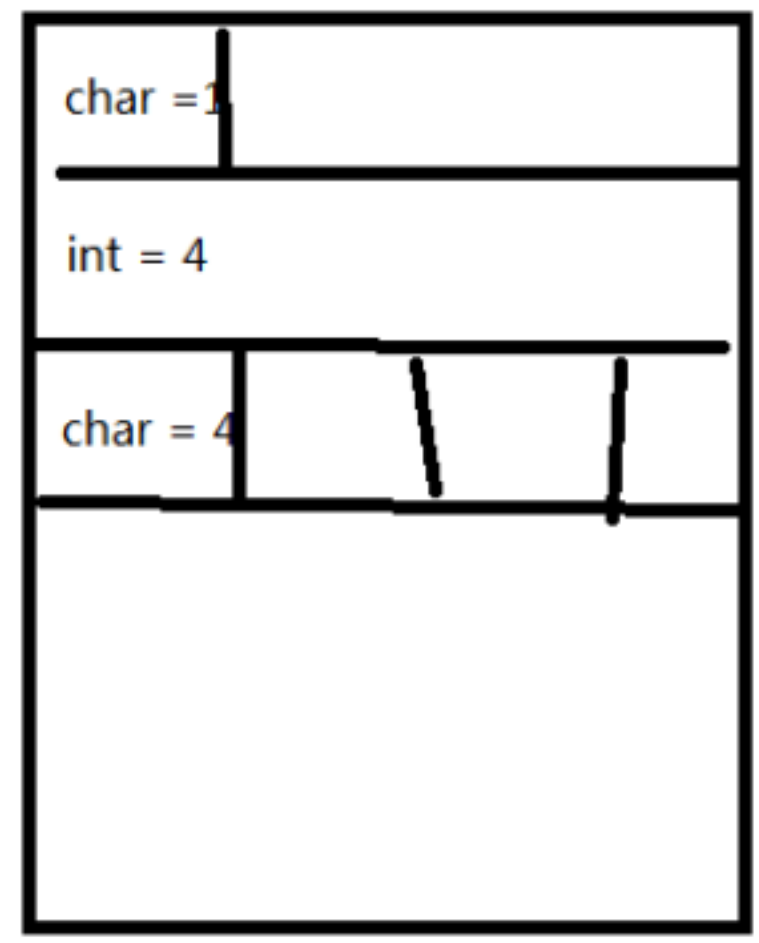
编译器在编译一个结构的时候采用内存对齐模式

```
struct     man{
     char  a;
     int   b;
};
```


int = 4
age = 4
char = 1
浪费
浪费
浪费


int = 4
char = 1
char = 1

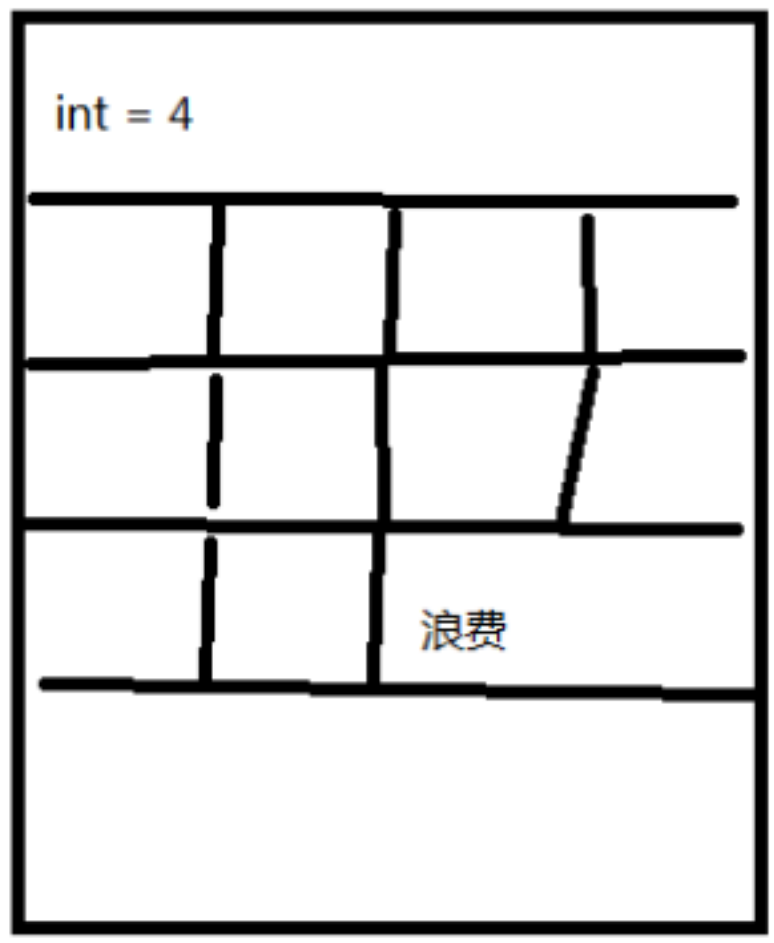

char = 1
char 1
int = 4

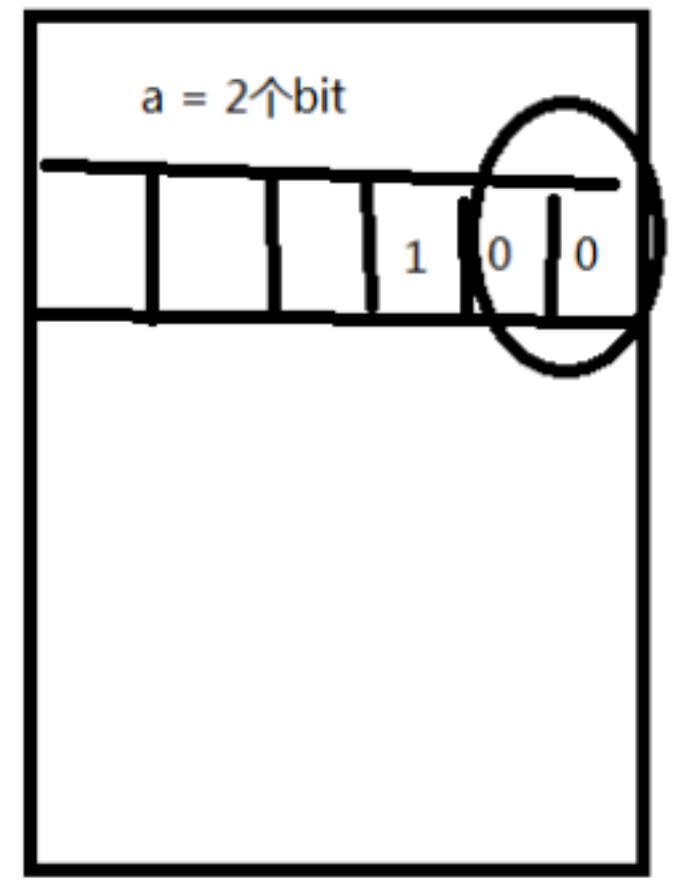

char =1
int = 4
char = 4

15.1.4 指定结构体元素的位字段

定义一个结构体的时候可以指定具体元素的位长

```
struct  test {
    char a :  2; // 指定元素为 2位长，不是 2个字节长
};
```





15.1.5 结构数组

```
struct  manm[ 10] = { {  "tom" ,  12 }, {  "marry" ,  10 }, {  "jack" ,  9 } };
```

15.1.6 嵌套结构

一个结构的成员还可以是另一个结构类型

```
struct  names{
    char  first[  100];
    char  last[  100];
};

struct  man{
```

```
        struct    namesname;
        int   age;
    };
    struct    manm = { {     "wang",   "wu" },    20 };
```

15.1.7　结构体的赋值

struct name a = b;

结构的赋值其实就是两个结构内存的拷贝

如果结构体成员有指针元素，那么就不能直接赋值,

15.1.8　指向结构体的指针

-> 操作符

15.1.9　指向结构体数组的指针

15.1.10 结构中的数组成员和指针成员

一个结构中可以有数组成员，也可以有指针成员，如果是指针成员结构体成员在初始化和赋值的时候就需要提前为指针成员分配内存。

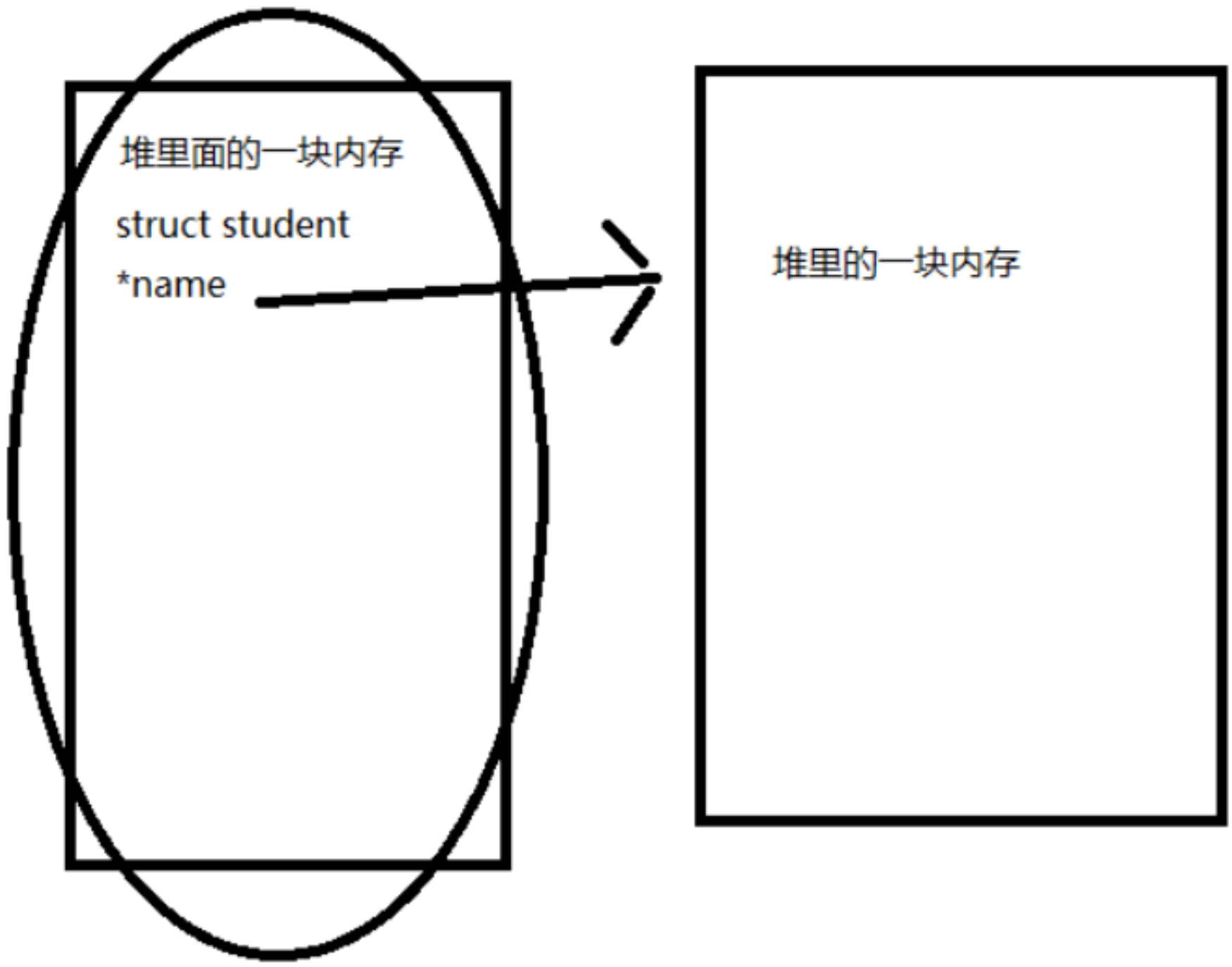
```
struct    man
{
    char  name[100];
    int   age;
};
```

```
struct    man
{
    char  *name;
    int   age;
};
```

15.1.11 在堆中创建的结构体

如果结构体有指针类型成员，同时结构体在堆中创建，那么释放堆中的结构体之前需要提前释放结构体中的指针成员指向的内存。

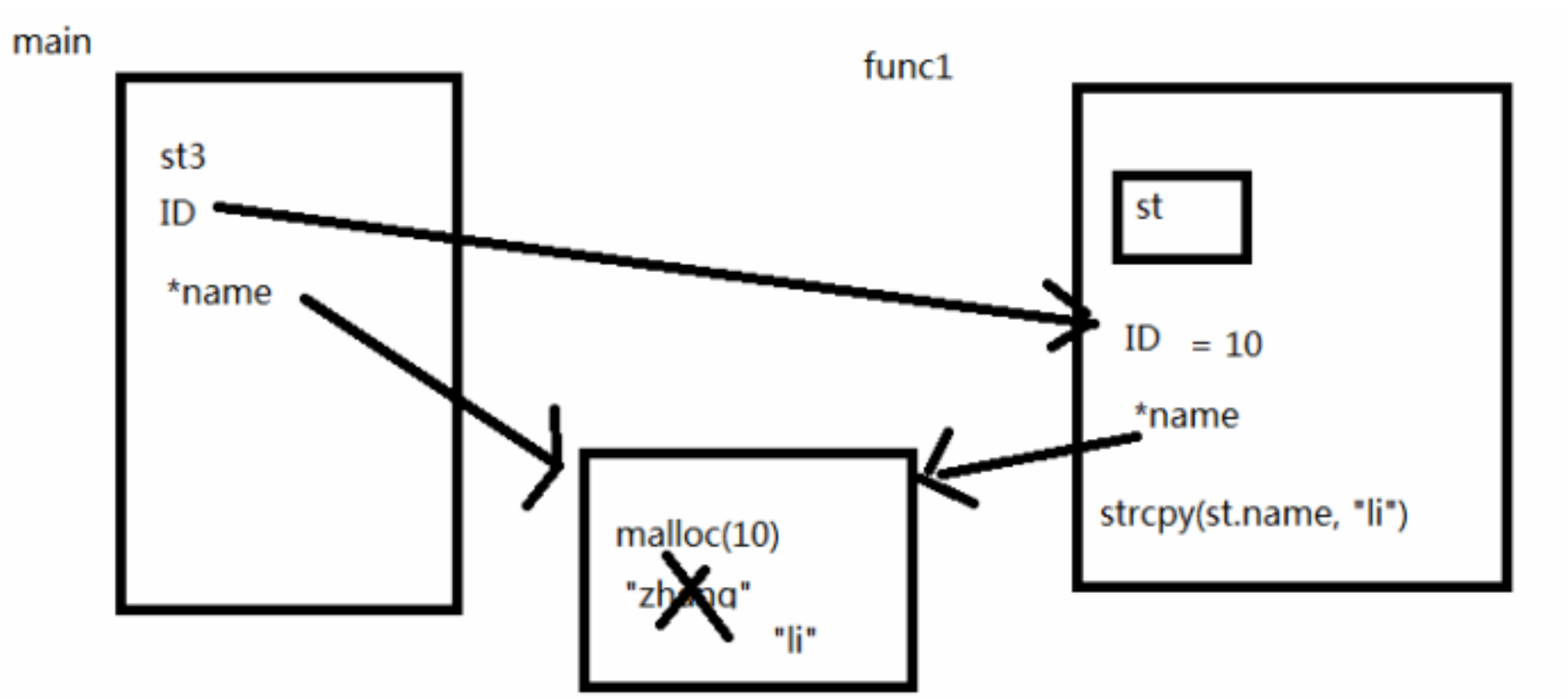
```
struct    man
{
    char  *name;
    int   age;
};

    struct    man*s = malloc(    sizeof  ( struct    man) *    2);
    s[ 0].name = malloc(    10 *    sizeof  ( char ));
    s[ 1].name = malloc(    10 *    sizeof  ( char ));
```

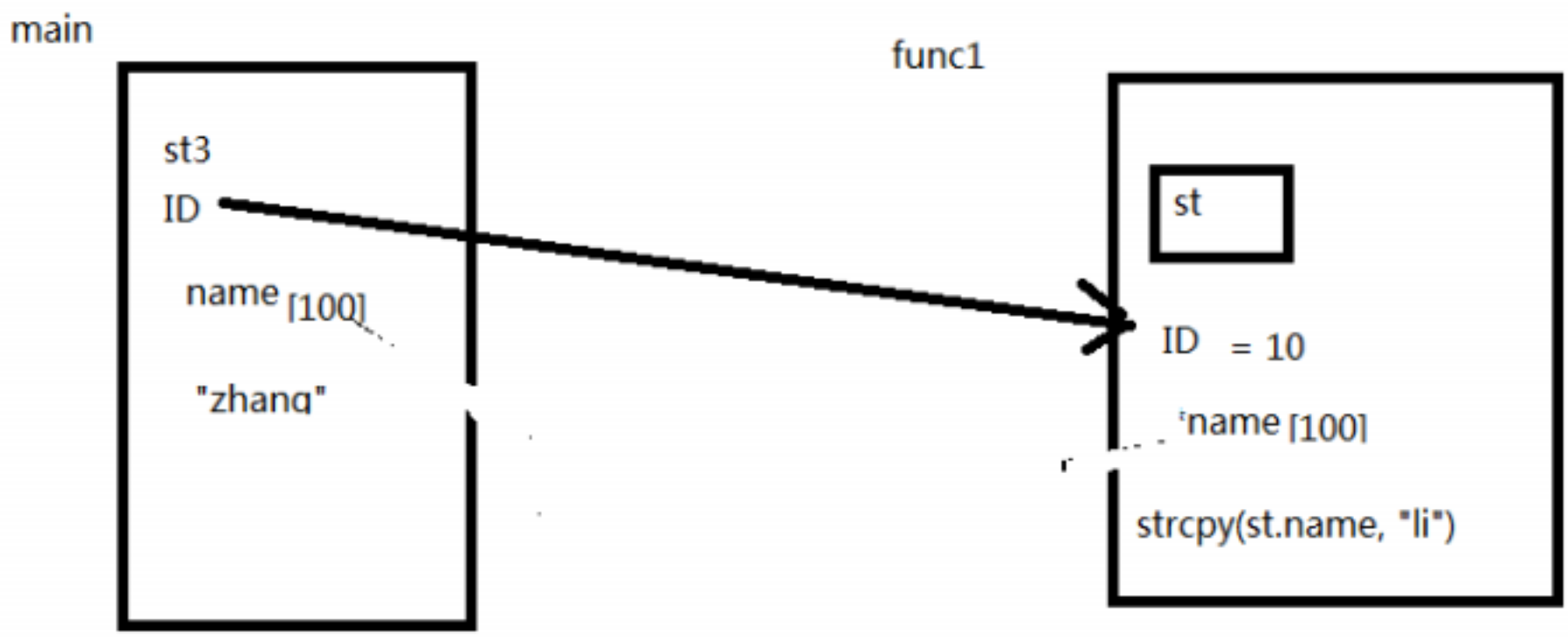


15.1.12 将结构作为函数参数

将结构作为函数参数

将结构指针作为函数参数



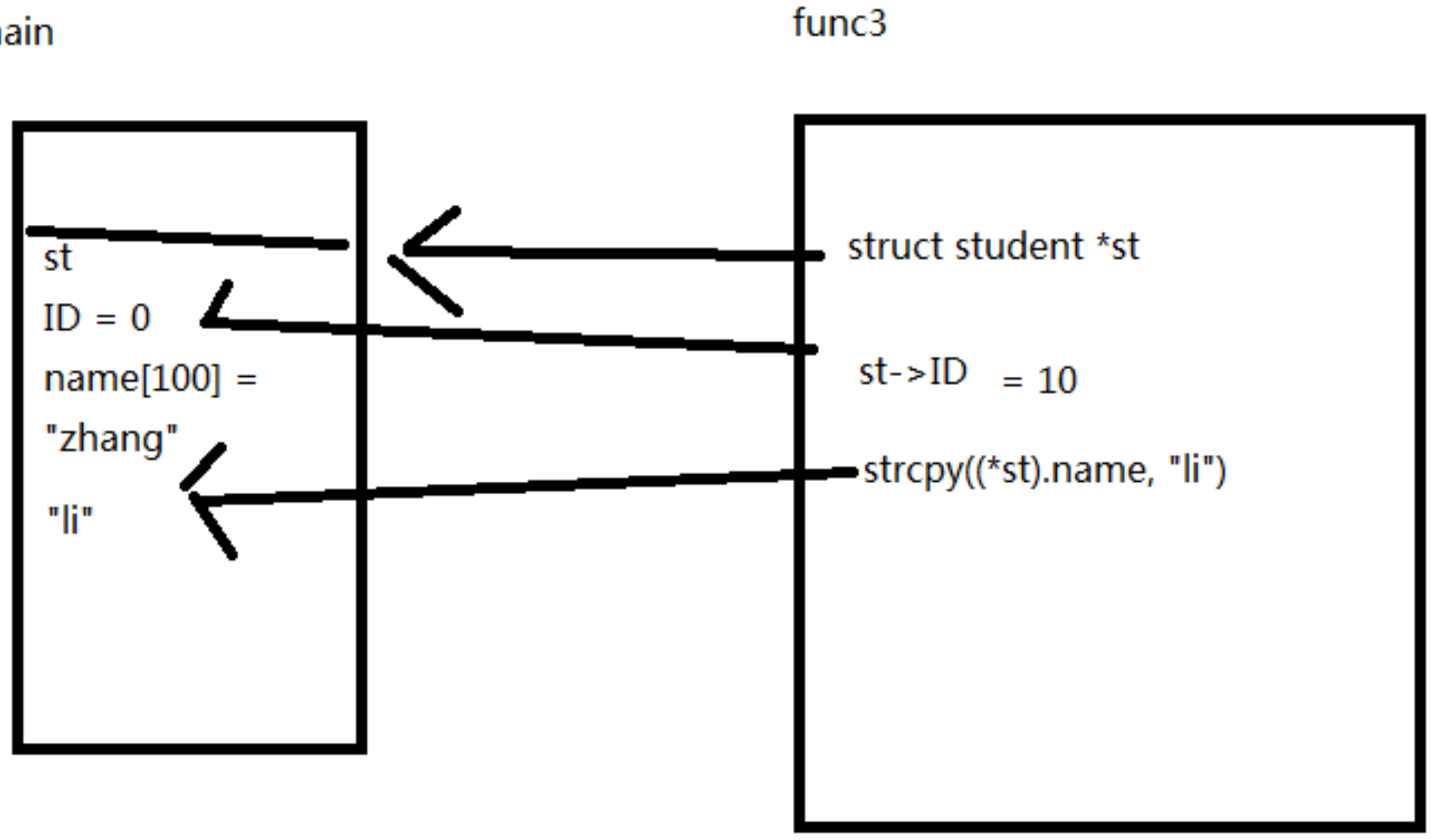


15.1.13 结构，还是指向结构的指针

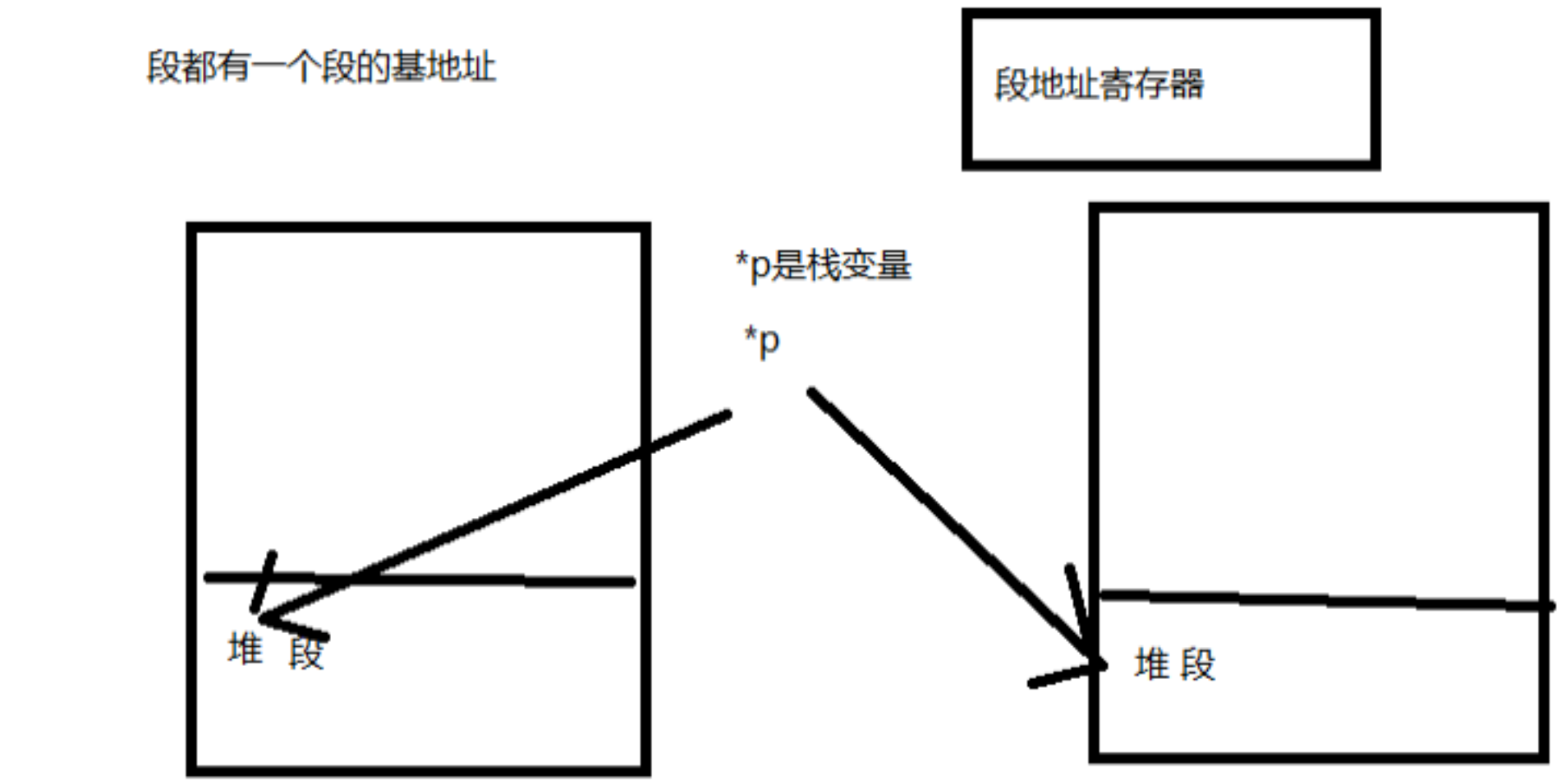
在定义一个和结构有关的函数，到底是使用结构，还是结构的指针？

指针作为参数，只需要传递一个地址，所以代码效率高

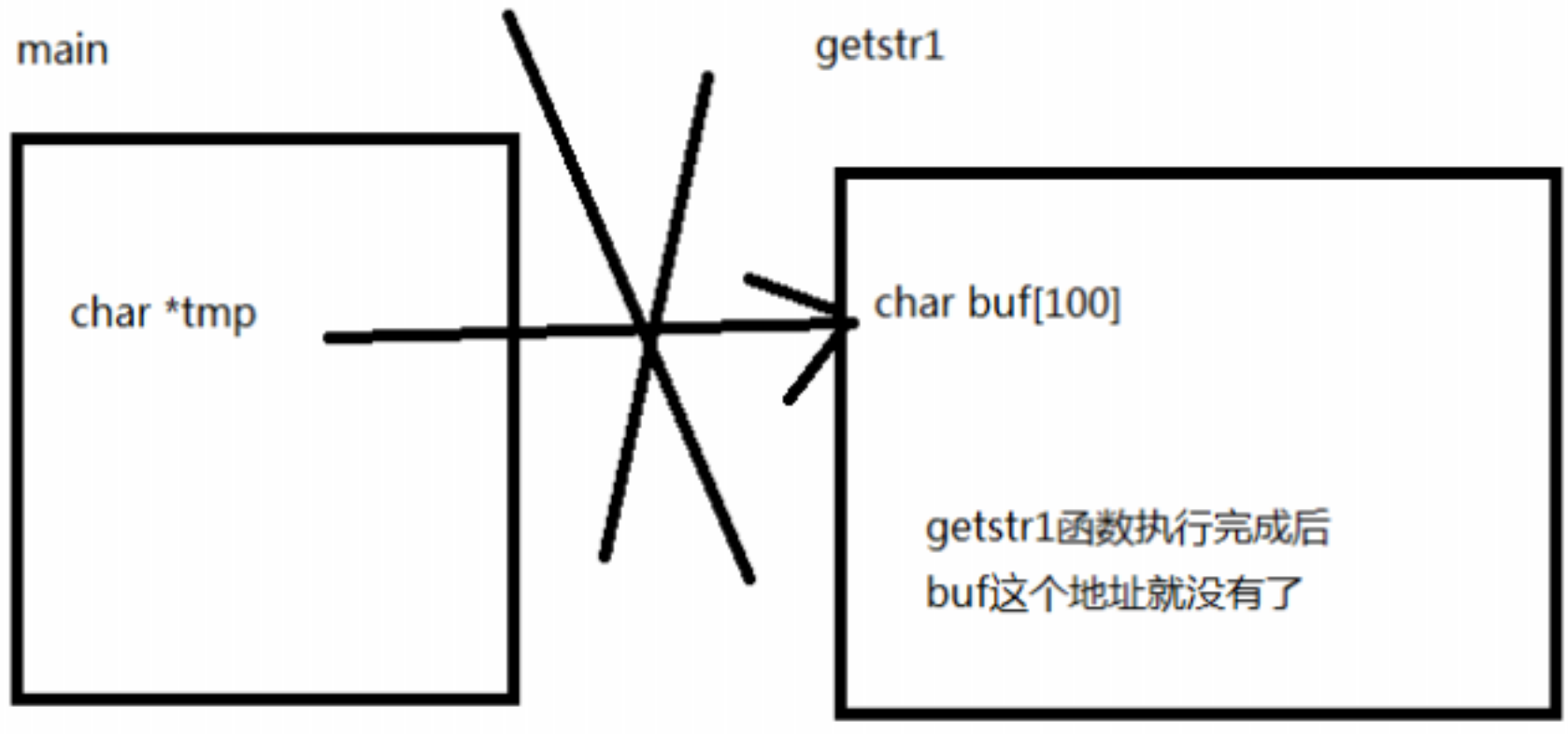
如果一个结构体变量做为函数的参数，效率极低。同时老的 C 编译器都不支持传递结构变量，只支持传递结构指针。

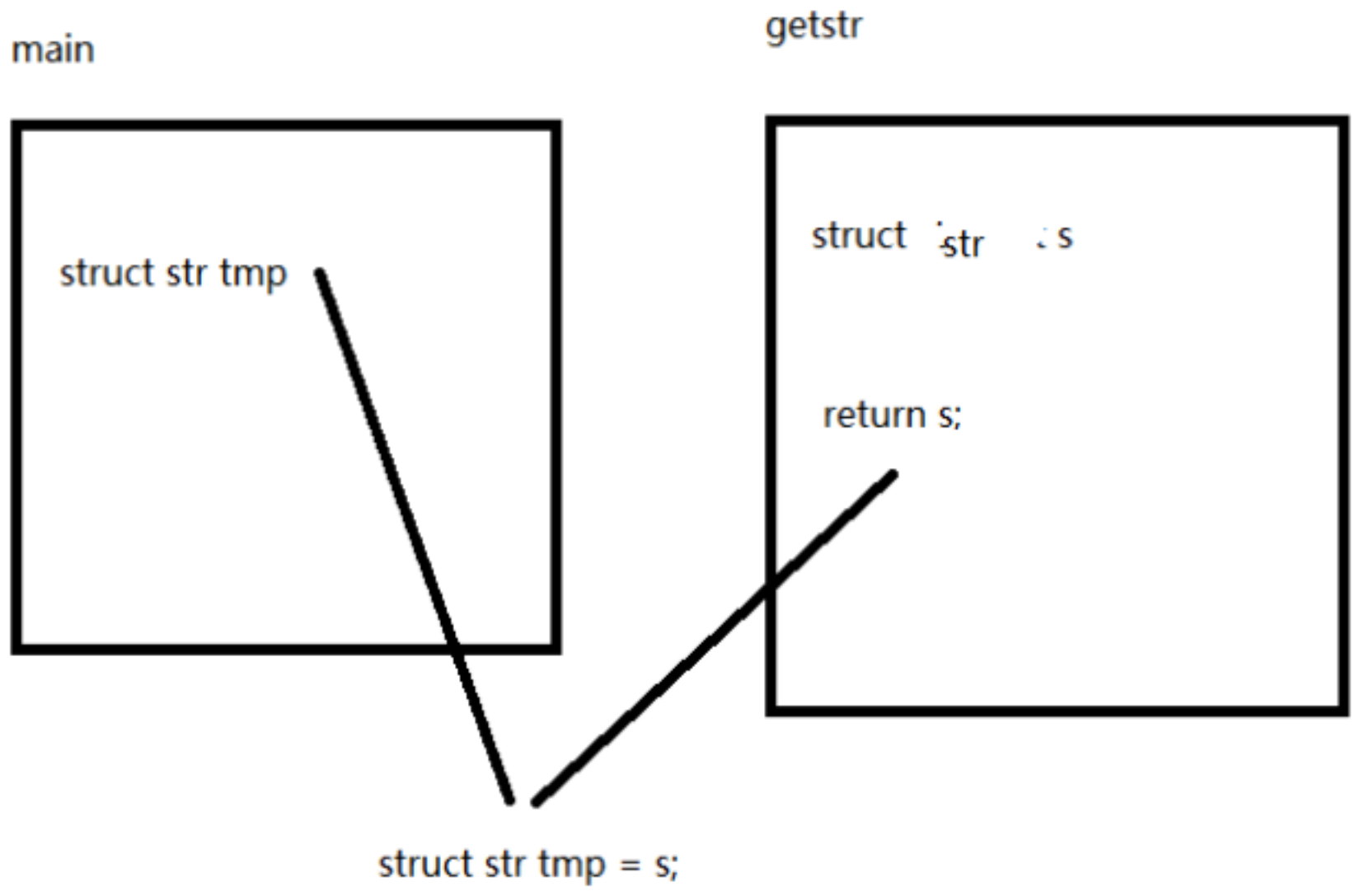


15.1.14 远指针与近指针



变量基于基地址的偏移地址





15.2 联合体

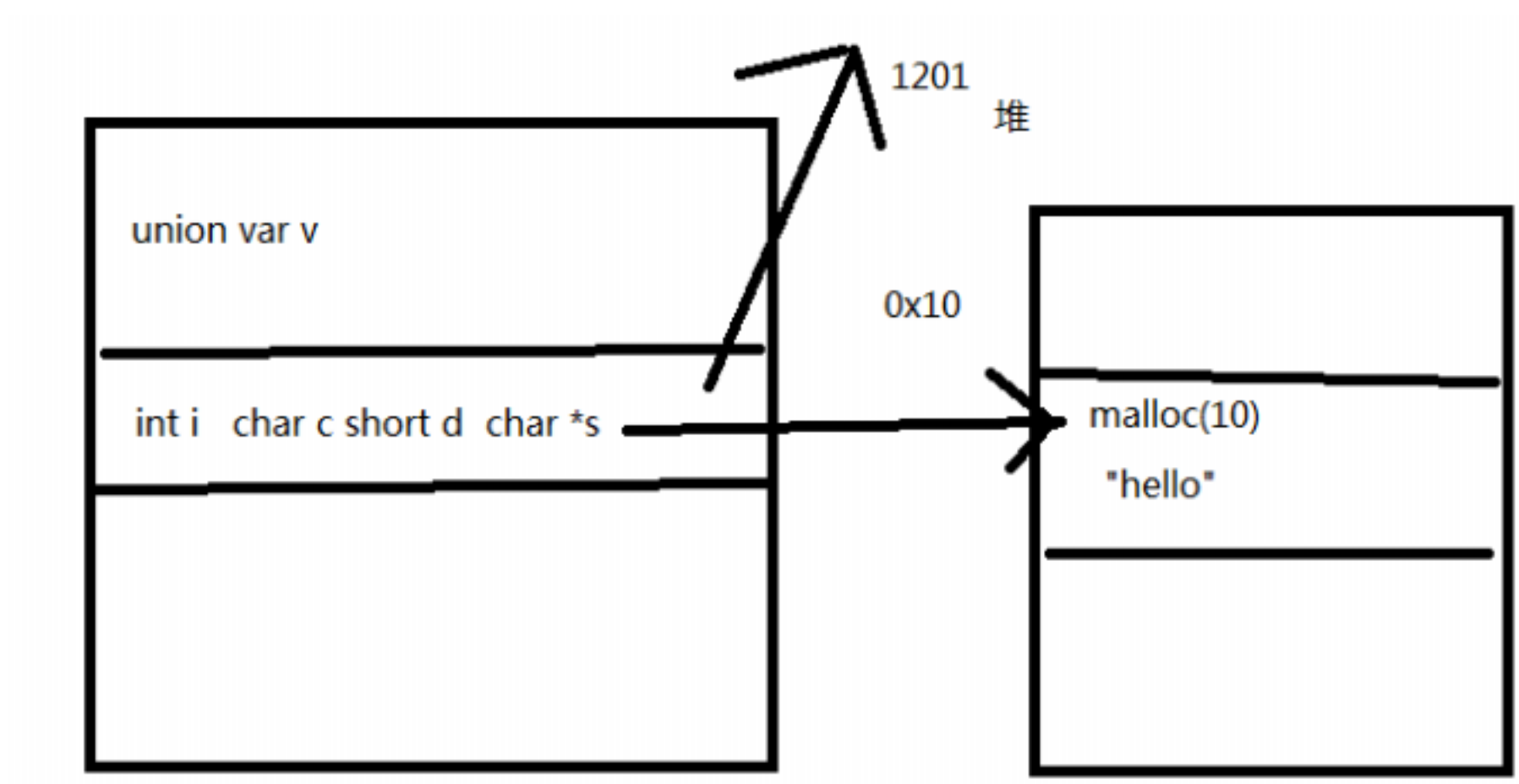
联合 union 是一个能在同一个存储空间存储不同类型数据的类型。

联合体所占的内存长度等于其最长成员的长度，也有叫做共用体。

联合体虽然可以有多个成员，但同一时间只能存放其中一种。

对于联合体来讲最基本的原则是，一次只操作一个成员变量，如果这个变量是指针，那么一定是处理完指针对应的内存之后再来使用其他成员。

```
union variant {
    int ivalue;
    char cvalue;
    double dvalue;
};

int main()
{
    union variant var;
    var.cvalue = 12;
    printf( "%d\n" , var.ivalue);
    printf( "%p, %p, %p\n" , &(var.cvalue), &(var.ivalue),
&(var.dvalue));
    return 0;
}
```



15.3 枚举类型

15.3.1 枚举定义

可以使用枚举 (enumerated type) 声明代表整数常量的符号名称, 关键字 enum 创建一个新的枚举类型。

实际上, enum 常量是 int 类型的。

枚举的本质就是 int 型的常量。

```
enum spectrum { red, yellow, green, blue, white, black };
enum spectrum color;
color = black;
if (color != red)
```

15.3.2 默认值

默认时，枚举列表中的常量被指定为 0,1,2 等

```
enum spectrum { red, yellow, green, blue, white, black };
printf( "%d, %d\n", red, black );
```

指定值

可以指定枚举中具体元素的值

```
enum spectrum { red = 10, yellow = 20, green, blue, white, black };
printf( "%d, %d\n", red, black );
```

15.4 typedef

typedef 是一种高级数据特性，它能使某一类型创建自己的名字

```
typedef unsigned char BYTE
```

1 与#define 不同， typedef 仅限于数据类型，而不是能是表达式或具体的值

2typedef 是编译器处理的，而不是预编译指令

3typedef 比#define 更灵活

直接看 typedef 好像没什么用处，使用 BYTE 定义一个 unsigned char。使用 typedef 可以增加程序的可移植性。

15.5 通过 typedef 定义函数指针

```
typedef const char *(* SUBSTR)(const char *, const char *);
```

```
const char *getsubstr( const char * src , const char * str )
{
    return strstr( src , str );
}

const char *func( const char *(* s)( const char *, const char *),
const char * src , const char * str )
```

```
const char *(*p[ 3])( const char *, const char *);
```

在程序当中如果是定义一个可读的常量，适合用 #define

如果定义的是一个具体的数据类型，那么 typedef 更加适合。

如果是定义一个函数指针，那么基本就 typedef 吧.

16 文件操作

重点： ***

不论操作什么类型的文件，第一步先打开一个文件，第二步，读写文件，第三步关闭文件。

16.1 fopen

r 以只读方式打开文件，该文件必须存在。

r+ 以可读写方式打开文件，该文件必须存在。用 r+ 写文件时候，从文件开始位置写入

rb+ 读写打开一个二进制文件，允许读写数据，文件必须存在。

rw+ 读写打开一个文本文件，允许读和写。

w 打开只写文件，若文件存在则文件长度清为 0，即该文件内容会消失。若文件不存在则建立该文件。

wb

w+ 打开可读写文件，若文件存在则文件长度清为零，即该文件内容会消失。若文件不存在则建立该文件。

a 以附加的方式打开只写文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾，即文件原先的内容会被保留。 （EOF 符保留） ,如果文件不存在， a 的行为和 w 是一样的

a+ 以附加方式打开可读写的文件。若文件不存在，则会建立该文件，如果文件存在，写入的数据会被加到文件尾后，即文件原先的内容会被保留。 （原来的 EOF 符不保留）

-b 只对 windows 有效，对于 unix 来讲是无效，

16.2 二进制和文本模式的区别

1.在 windows 系统中，文本模式下，文件以 "\r\n" 代表换行。若以文本模式打开文件，并用 fputs 等函数写入换行符 "\n" 时，函数会自动在 "\n" 前面加上 "\r" 。即实际写入文件的是 "\r\n" 。

2.在类 Unix/Linux 系统中文本模式下，文件以 "\n" 代表换行。所以 Linux 系统中在文本模式和二进制模式下并无区别。

对于 GBK 编码的汉字，一个汉字两个字节，对于 utf8 来讲一个汉字 3 个字节，但如果英文字母都是一个字节

16.3 fclose

fclose 关闭 fopen 打开的文件

16.4 getc 和 putc 函数

```
int main()
{
    FILE *fp = fopen( "a.txt" , "r" );
    char c;
    while ((c = getc(fp)) != EOF)
    {
        printf( "%c", c);
    }
    fclose(fp);
    return 0;
}
```

```
int main()
{
    FILE *fp = fopen( "a.txt" , "w" );
    const char *s = "hello world" ;
    int i;
    for (i = 0; i < strlen(s); i++)
    {
        putc(s[i], fp);
    }
    fclose(fp);
    return 0;
```

	}

16.5 EOF 与 feof 函数文件结尾

程序怎么才能知道是否已经到达文件结尾了呢？ EOF 代表文件结尾

如果已经是文件尾， feof 函数返回 true 。

16.6 fprintf,fscanf,fgets,fputs 函数

这些函数都是通过 FILE * 来对文件进行读写。

fscanf 不会读取行尾的 ‘\n’,fgets 会将行尾的 ‘\n’ 读取到 buf 里面

不论 fprintf 还是 fputs 都不会自动向行尾添加 \n, 需要代码中往 buf 的行尾写 \n 才可以达到换行的目录

16.7 stat 函数

#include <sys/stat.h>

函数的第一个参数代表文件名，第二个参数是 struct stat 结构。

得到文件的属性，包括文件建立时间，文件大小等信息。

16.8 fseek 函数

函数设置文件指针 stream 的位置。如果执行成功， stream 将指向以 fromwhere 为基准，偏移 offset （指针偏移量）个字节的位置，函数返回 0。如果执行失败则不改变 stream 指向的位置，函数返回一个非 0 值。

实验得出，超出文件末尾位置，还是返回 0。往回偏移超出首位置，还是返回 0，请小心使用。

第二个参数负数代表向前移动，整数代表向后移动。

第一个参数 stream 为文件指针

第二个参数 offset 为偏移量，单位：字节，正数表示正向偏移，负数表示负向偏移

第三个参数 origin 设定从文件的哪里开始偏移 ,可能取值为： SEEK_CUR、 SEEK_END 或 SEEK_SET

SEEK_SET： 文件开头

SEEK_CUR： 当前位置

SEEK_END： 文件结尾

```
fseek(fp, 3, SEEK_SET);
```

16.9 ftell 函数

函数 ftell 用于得到文件位置指针当前位置相对于文件首的偏移字节数。在随机方式存取文件时，由于文件位置频繁的前后移动，程序不容易确定文件的当前位置。

```
long len = ftell(fp)
```

16.10 fgetpos,fsetpos 函数

fseek 与 ftell 返回的是 long 类型，如果文件很大， 超过 long 的范围， 那么该函数会有问题，fgetpos 与 fsetpos 函数可以处理更大的文件类型

返回值：成功返回 0，否则返回非 0

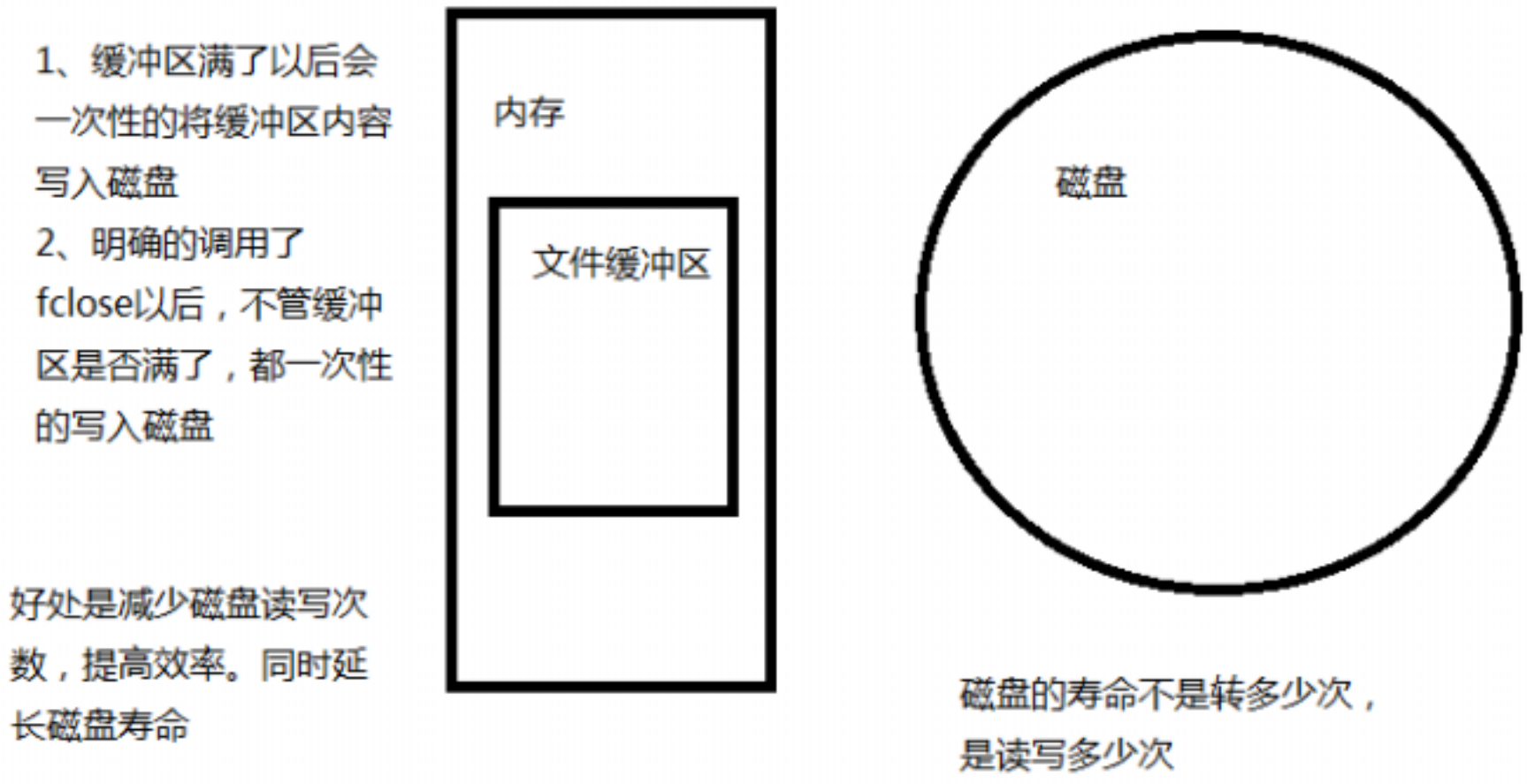
```
fpos_t ps = 0;
fgetpos(fp, &ps);
```

```
fpos_t ps = 2;
fsetpos(fp, &ps);
```

16.11 fflush 函数

fflush 函数可以将缓冲区中任何未写入的数据写入文件中。

修改配置文件，希望修改实时生效，那么每次修改完成之后我们 fflush 一次



16.12 fread 和 fwrite 函数

size_t fread (void *buffer , size_t size, size_t count, FILE *stream) ;

size_t fwrite(const void* buffer , size_t size, size_t count, FILE* stream);

注意：这个函数以二进制形式对文件进行操作，不局限于文本文件

返回值：返回实际写入或读取的数据块数目

只要读取到文件最后，没有完整的读取一个数据块出来， fread 就返回 0

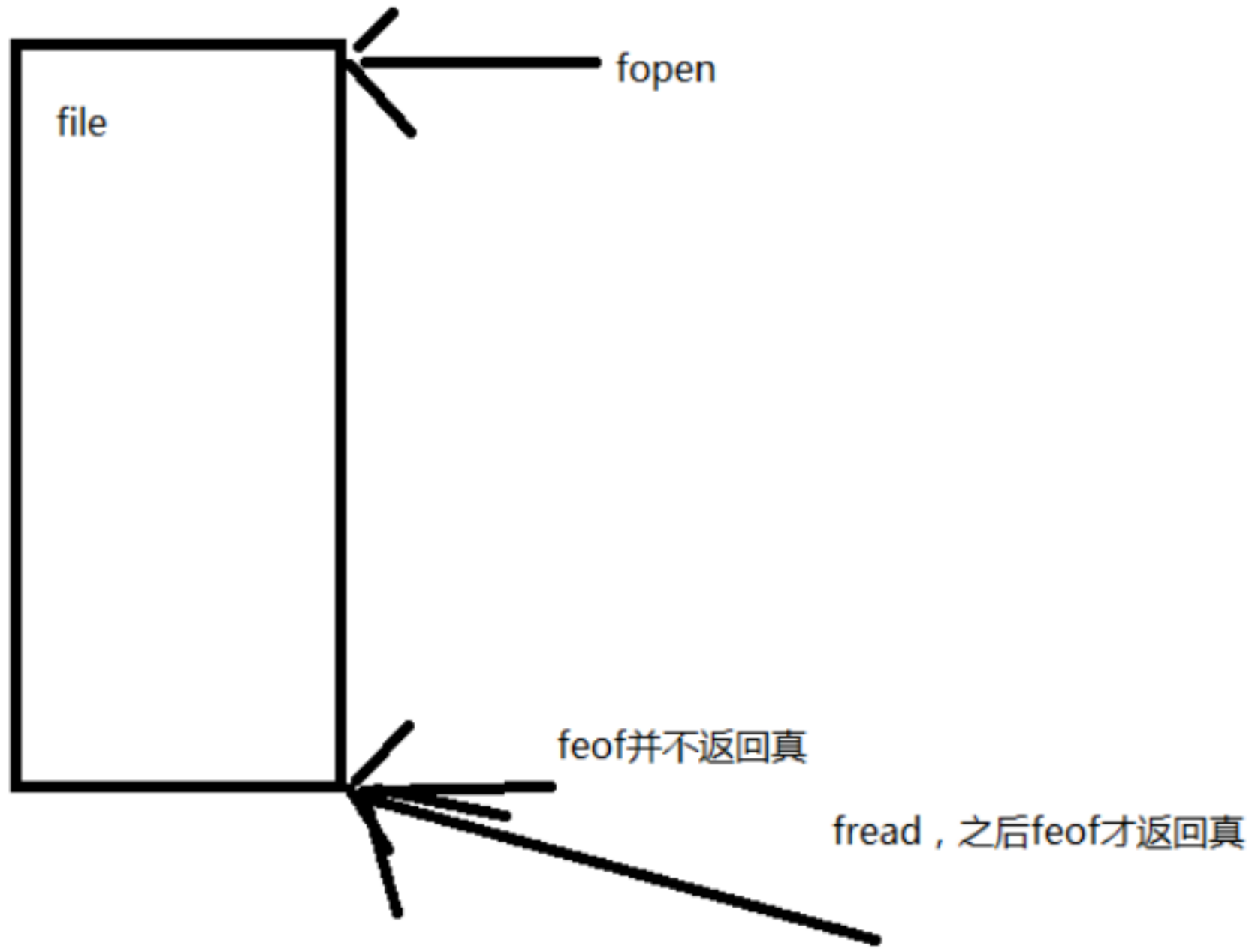
第一个参数代表 void * ，写入或者读取的缓冲区

第二个参数是代表写入或读取的时候一个单位的大小

第三个参数是代表写入或读取几个单位

第四个参数是 FILE *

16.13 fread 与 feof



注意以下两段代码的区别

```
while   (!feof(p))
{
    fread(&buf,      1,   sizeof  (buf), p);
}
```

```
while   (fread(&buf,       1,   sizeof  (buf), p))
```

16.14 作业

有一个文本 a.txt ， 内容一行是一个数字

有多少随机的

要求你写程序读取 a.txt, 将文件内容小到大排序， 写入 b.txt

17 基础数据结构与算法

重点： **** ，如果基础数据结构还不扎实的话，建议这段时间仔细的复习一下。

17.1 什么是数据结构

数据 (data) 是对客观事物符号表示，在计算机中是指所有能输入的计算机并被计算机程序处理的数据总称。

数据元素（ data element ）是数据的基本单位，在计算机中通常做为一个整体进行处理。

数据对象（ data object ）是性质相同的数据元素的集合，是数据的一个子集。

数据结构（ data structure ）是相互之间存在一种或多种特定关系的数据元素的集合。

数据类型（ data type ）是和数据结构密切关系的一个概念，在计算机语言中，每个变量、常量或者表达式都有一个所属的数据类型。

抽象数据类型（ abstract data type ADT ）是指一个数据模型以及定义在该模型上的一组操作，抽象数据类型的定义仅取决于它的一组逻辑性， 与其在计算机内部如何表示以及实现无关。

17.2 什么是算法

算法是对特定问题求解的一种描述，它是指令的有限序列，其每一条指令表示一个或多个操作，算法还有以下特性：

有穷性

一个算法必须总是在执行有限步骤后的结果，而且每一步都可以在有限时间内完成。

确定性

算法中每一条指令都有确切的含义，读者理解时不会产生二义性，在任何条件下，算法只有唯一的一条执行路径，即相同的输入只能得出相同的输出。

可行性

一个算法是可行的，即算法中描述的操作都是可以通过已经实现的基本运算来实现的。

输入

一个算法有零个或者多个输入，这些输入取自与某个特定对象的集合。

输出

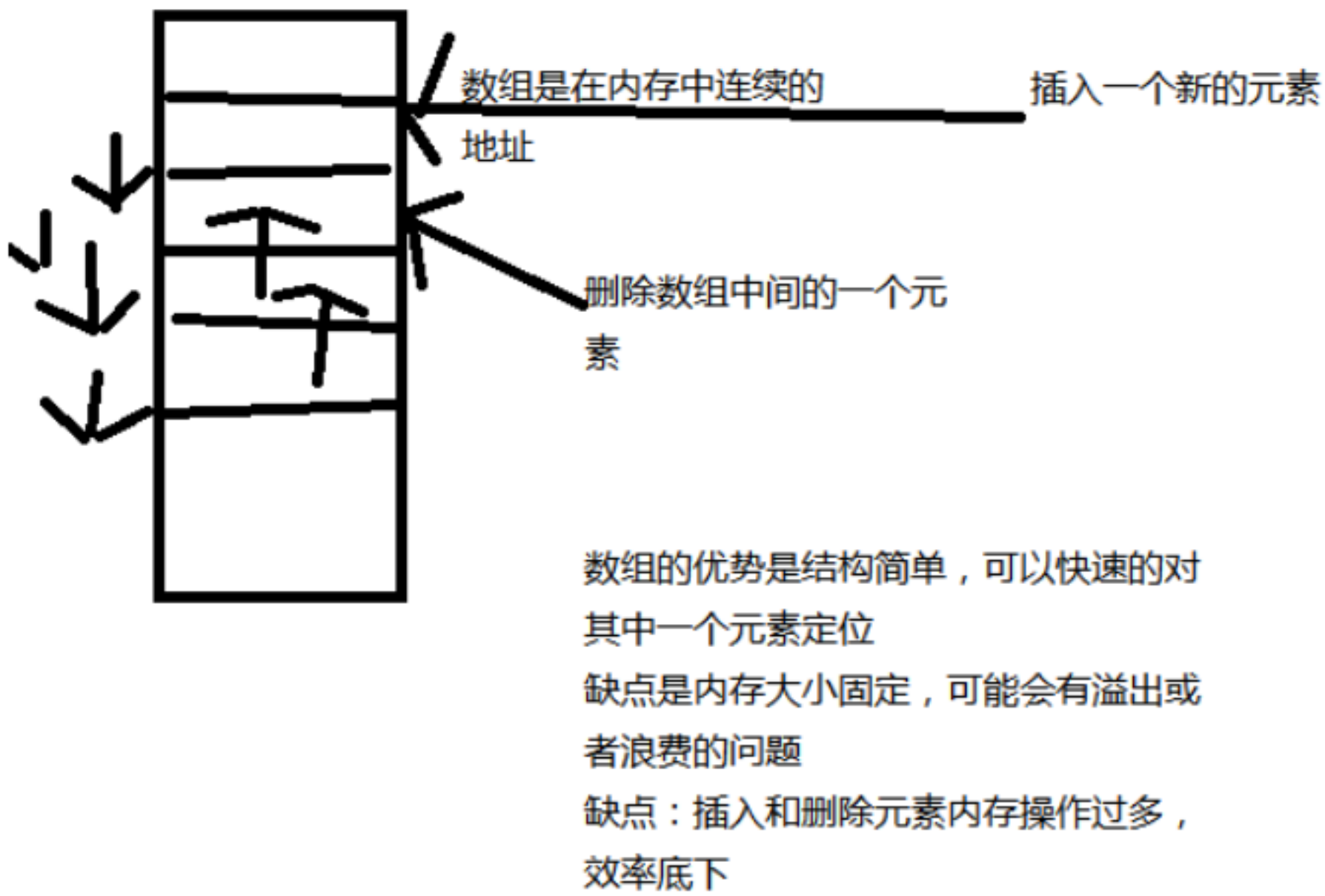
一个算法有一个或多个输出，这些输出是和输入有某些特定关系的量。

17.3 链表

17.3.1 单向链表定义

对于数组，逻辑关系上相邻的连个元素的物理位置也是相邻的，这种结构的优点是可以随机存储任意位置的元素，但缺点是如果从数组中间删除或插入元素时候，需要大量移动元素，效率不高。

数组的优缺点

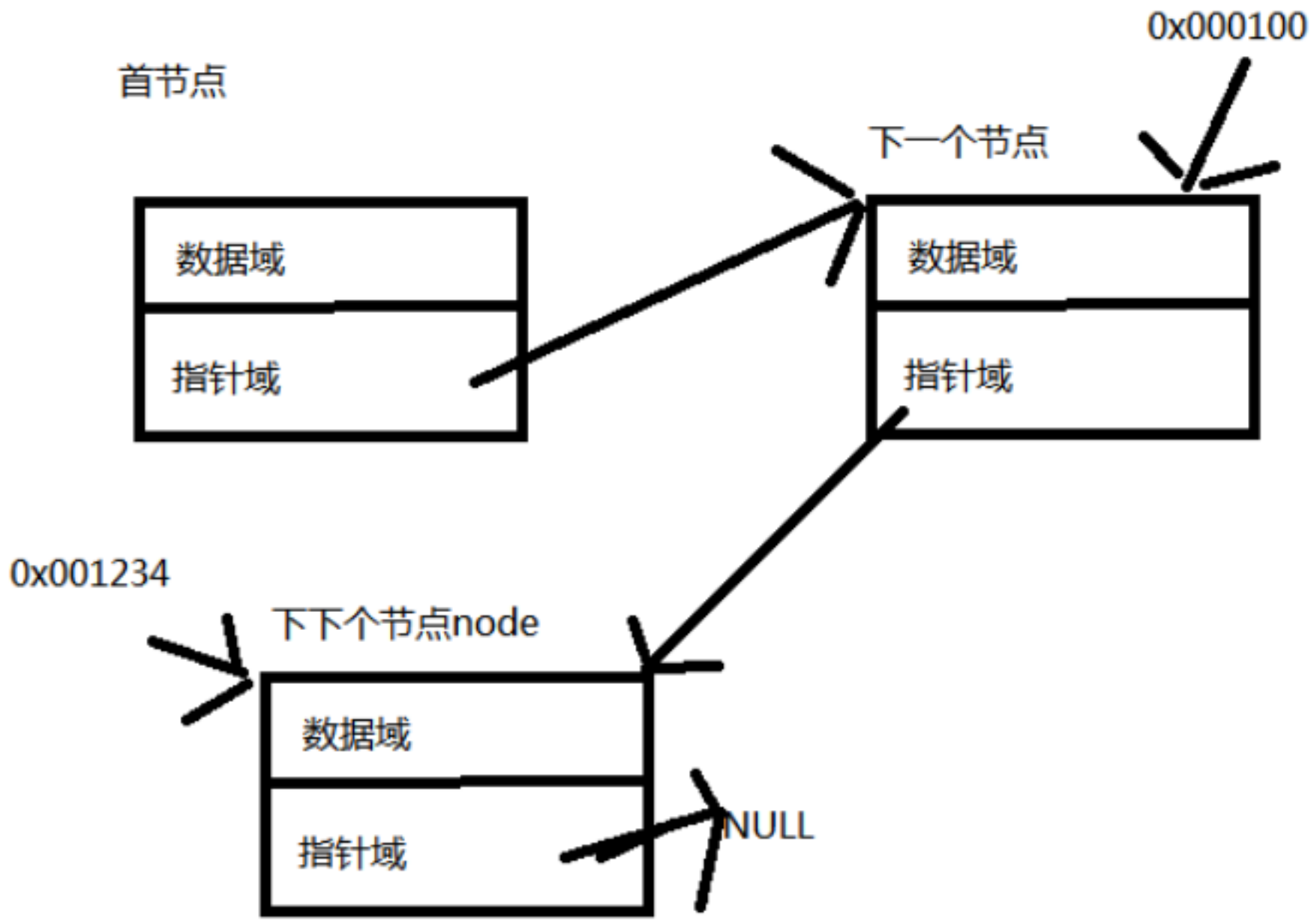


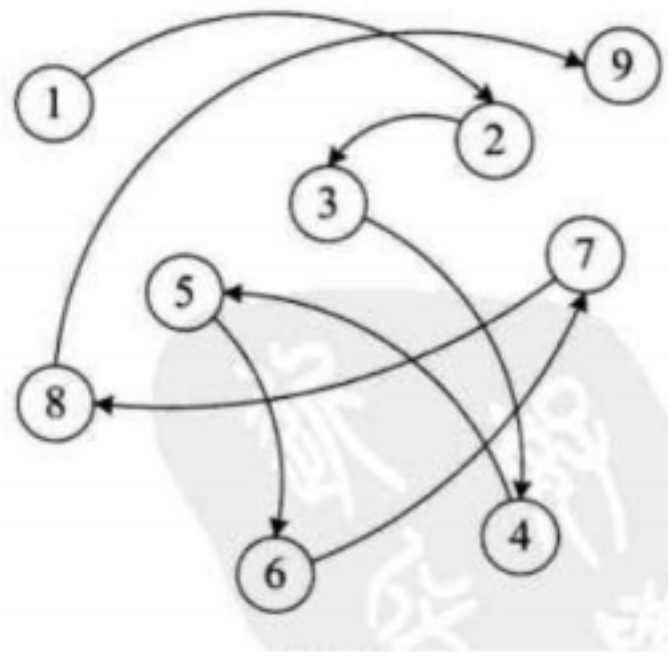
链式存储结构的特点，元素的存储单元可以是连续的，也可以是不连续的，因此为了表示每个元素 a，与其接后的元素 a+1 之间的关系，对于元素 a，除了存储其本身信息外，还需要存储一个指示其接后元素的位置。这两部分数据成为结点 (node) 。

一个结点中存储的数据元素被成为　　数据域 。存储接后存储位置的域叫做　　指针域 。 n 个结点(ai(1<=i<=n)　的存储映像链接成一个　链表 。

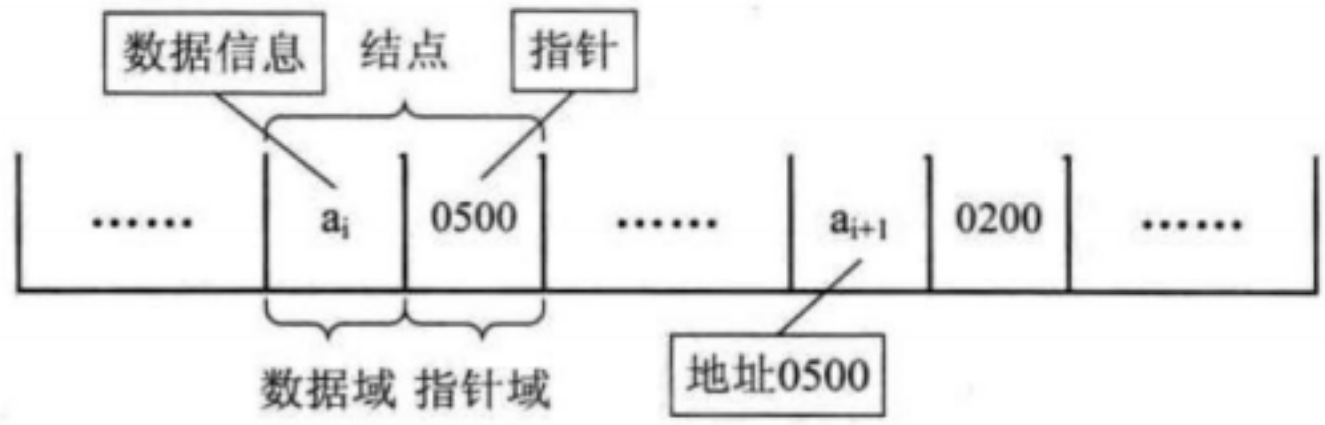
整个链表必须从　头结点 开始进行，头结点的指针指向下一个结点的位置，最后一个结点的指针指向　NULL 。

在链表中，通过指向接后结点位置的指针实现将链表中每个结点“ 链 ”到一起。链表中第一个结点称之为　头结点 。





n 个结点（a_i 的存储映像）链结成一个链表，即为线性表（a_1，a_2，…，a_n）的链式存储结构，因为此链表的每个结点中只包含一个指针域，所以叫做单链表。单链表正是通过每个结点的指针域将线性表的数据元素按其逻辑次序链接在一起，如图 3-6-2 所示。



17.3.2　单向链表数据结构定义

```
struct    list
{
    int   data;   //  链表数据域
    struct    list    *next;    //  链表指针域
};
```

17.3.3　单向链表的实现

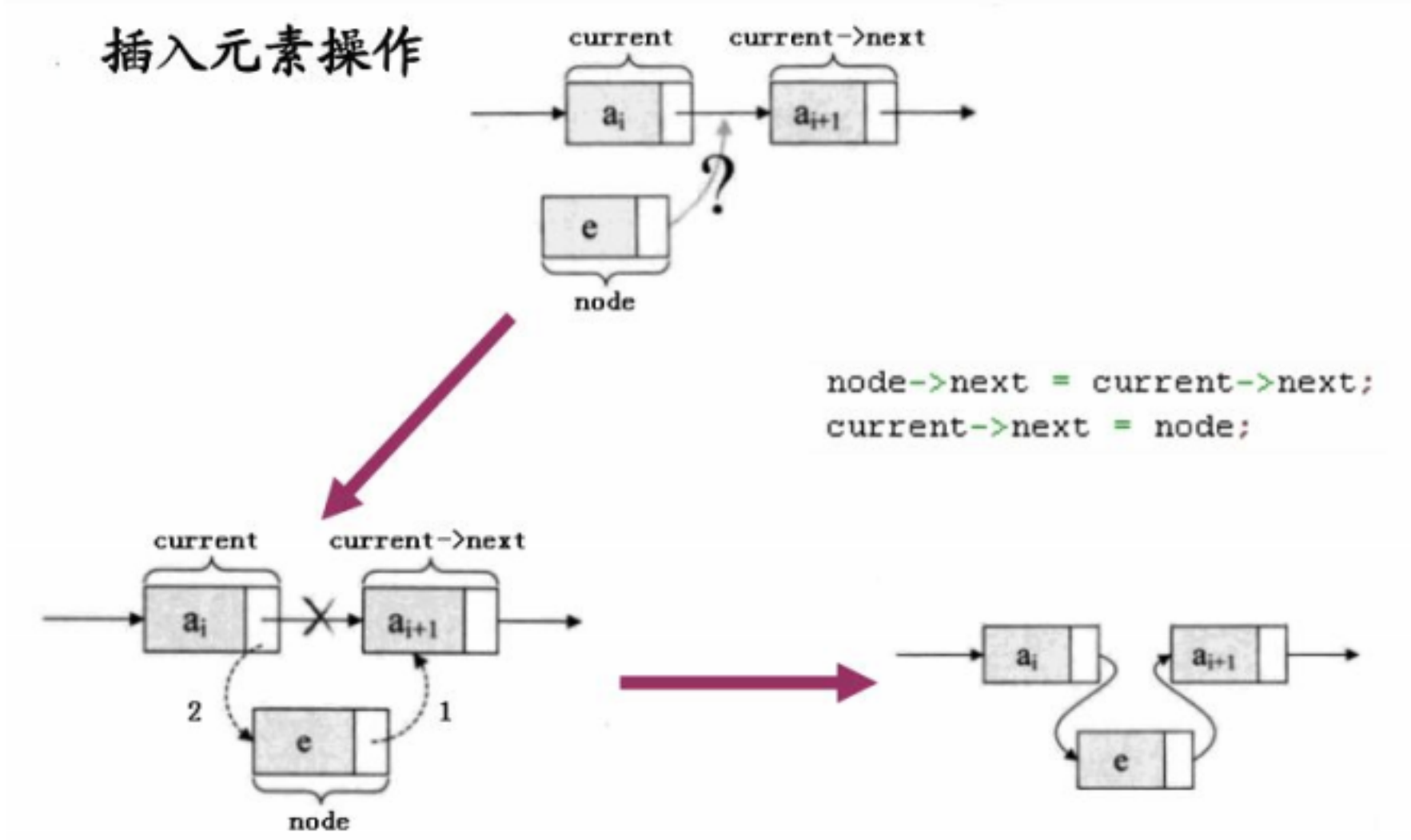
```
struct    list    *create_list()       //  建立一个节点
void  traverse(    struct    list    * ls ) //  循环遍历链表
struct    list    *insert_list(       struct    list    * ls ,   int    n,   int    data ) //  在指定位置插入元素
int   delete_list(       struct    list    * ls ,   int    n) //  删除指定位置元素
int   count_list(     struct    list    * ls ) //  返回链表元素个数
void  clear_list(     struct    list    * ls ) //  清空链表，只保留首节点
int   empty_list(     struct    list    * ls ) //  返回链表是否为空
struct    list    *locale_list(       struct    list    * ls ,   int    n) //  返回链表指定位置的节点
struct    list    *elem_locale(    struct    list    * ls ,   int    data ) //  返回数据域等于 data 的节点
int   elem_pos( struct    list    * ls ,   int    data ) //  返回数据域等于  data 的
```

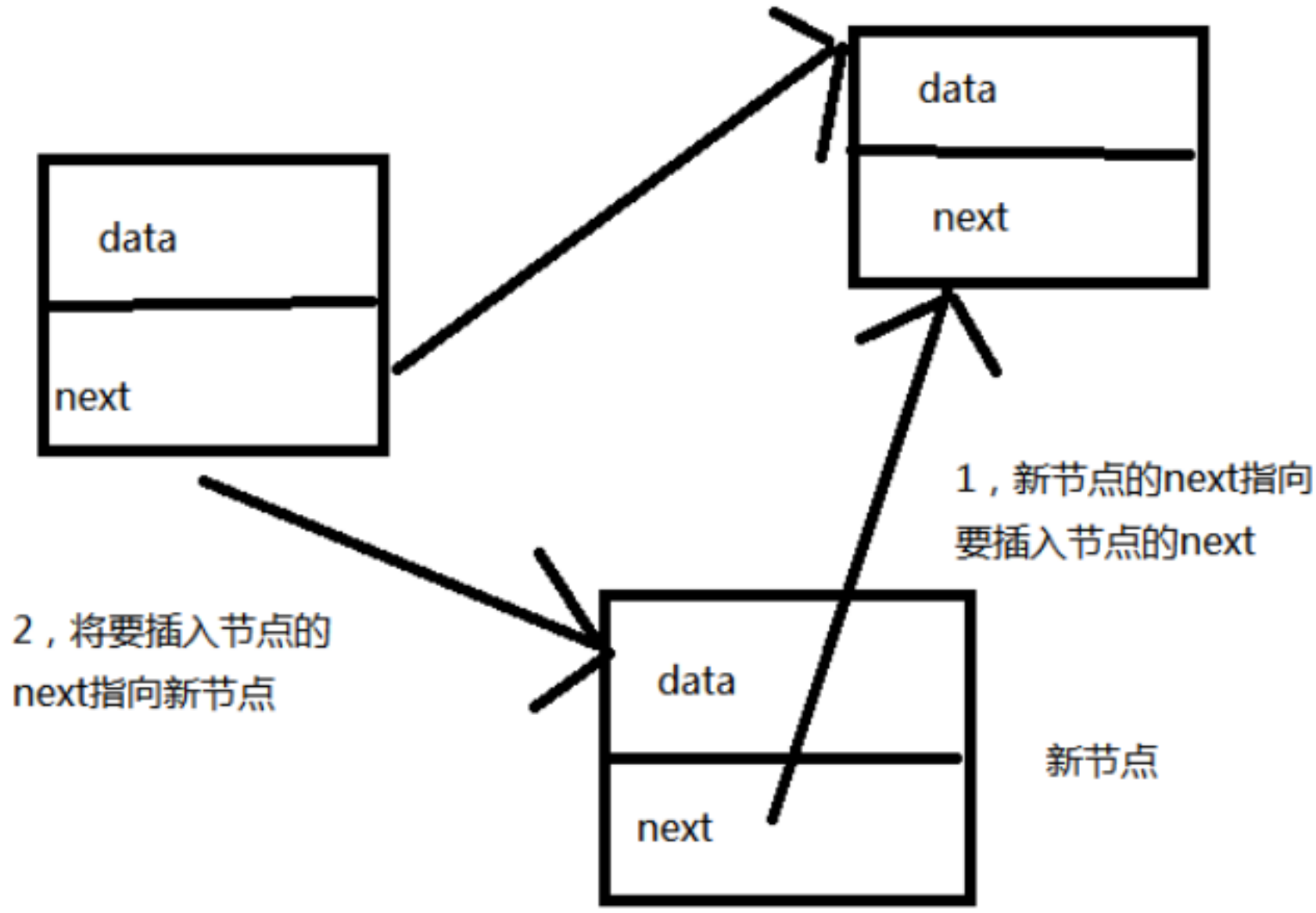
节点位置

struct list *last_list(struct list * ls) // 得到链表最后一个节点

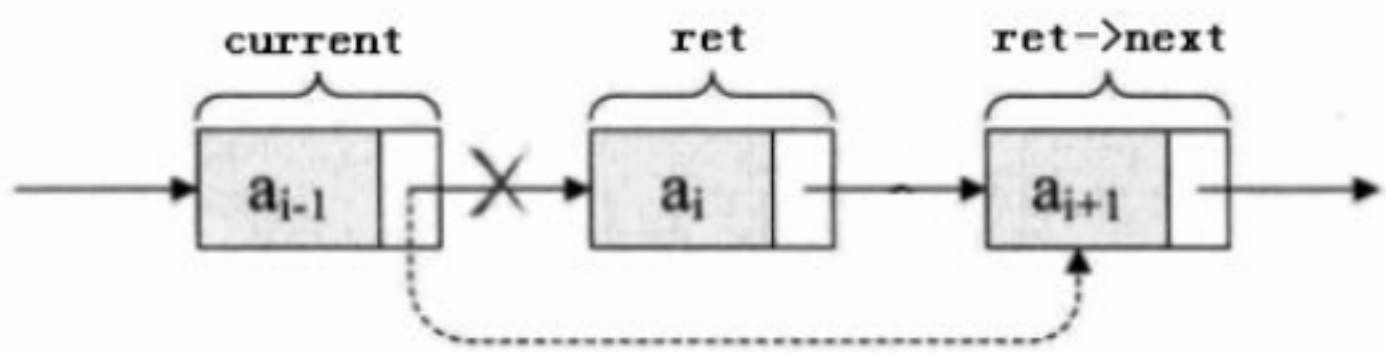
void merge_list(struct list * st1 , struct list * ls2) // 合并两个链表 , 结果放入 st1 中

void reverse(struct list * ls) // 链表逆置

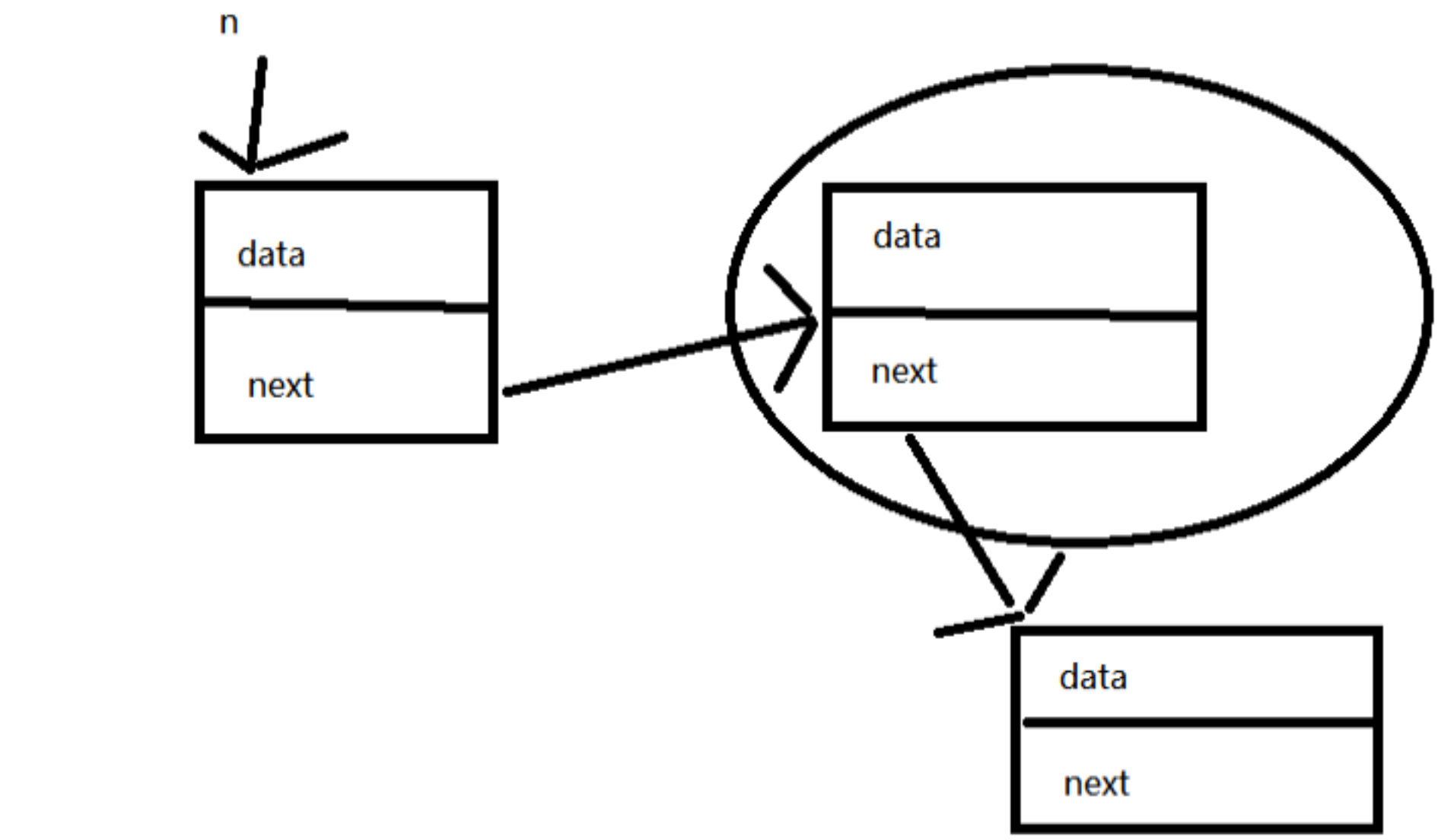


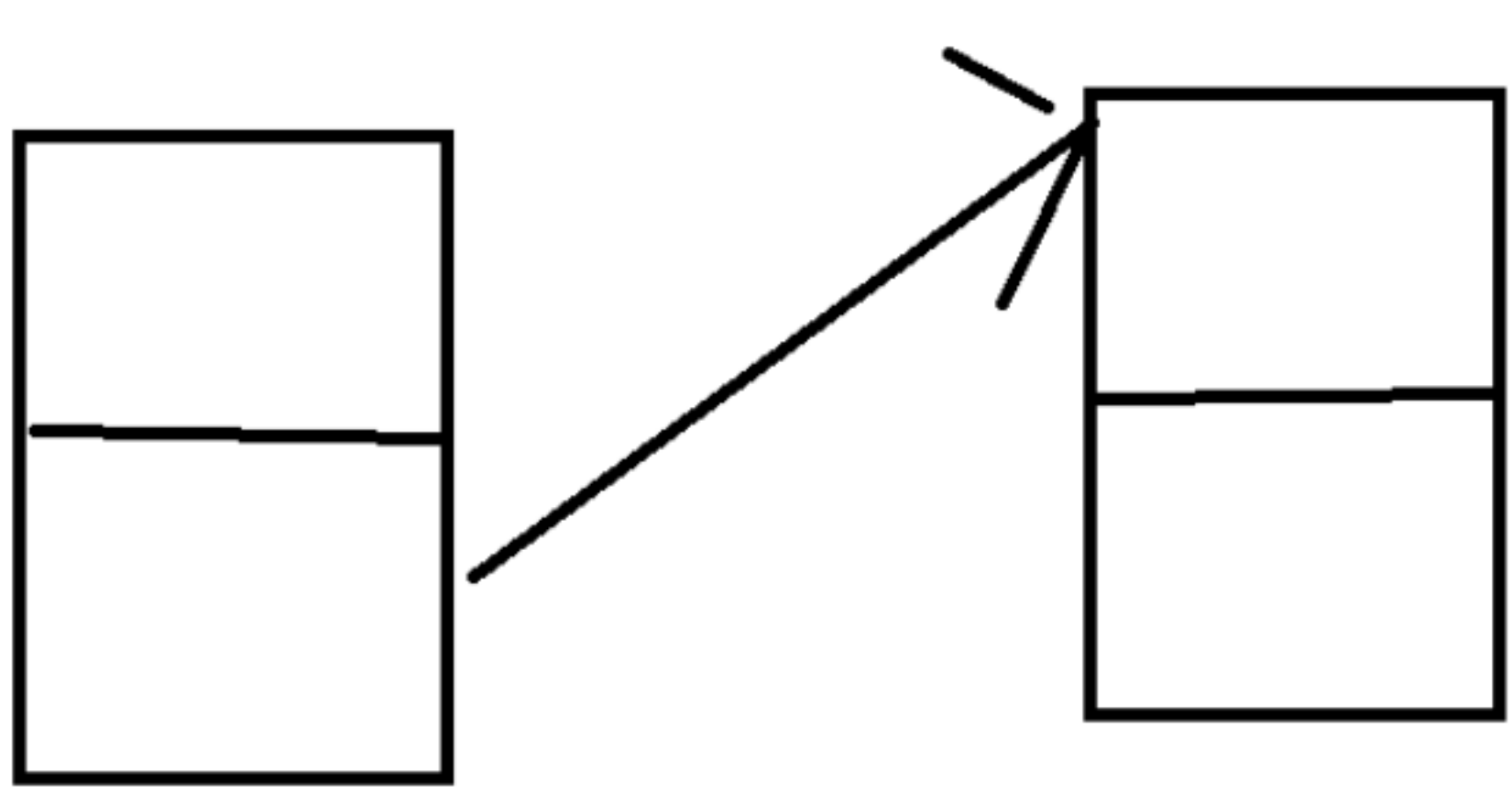


删除元素操作

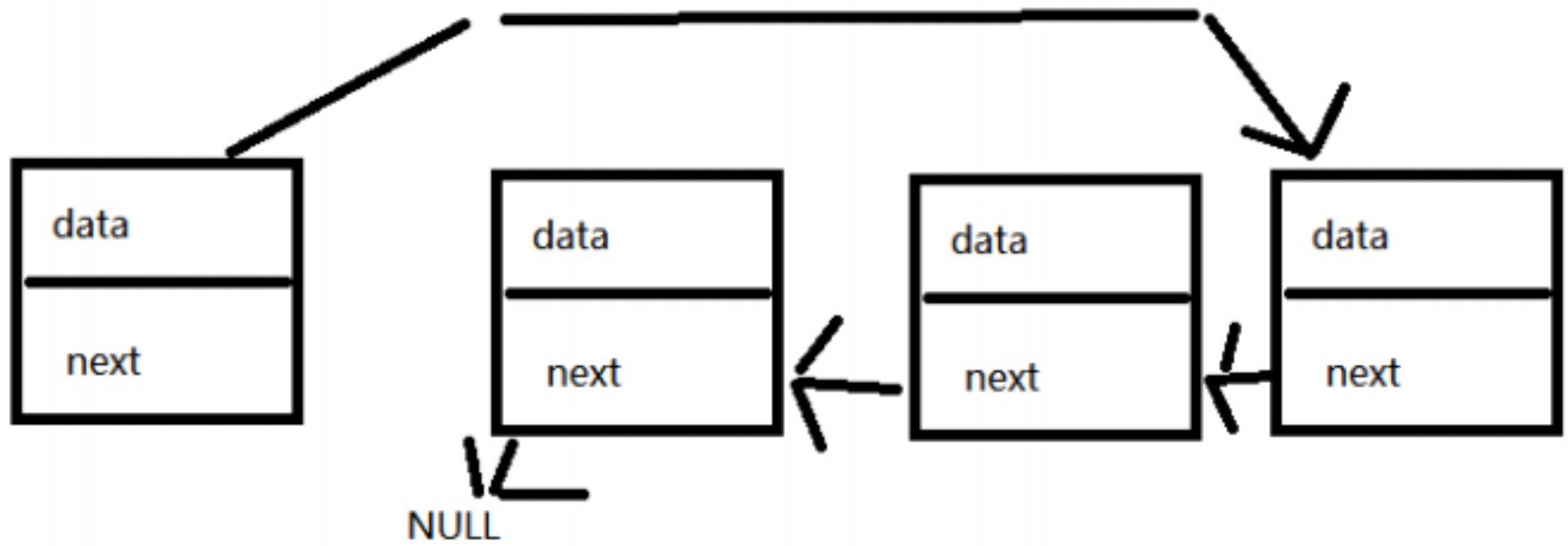



```
current->next = ret->next;
```

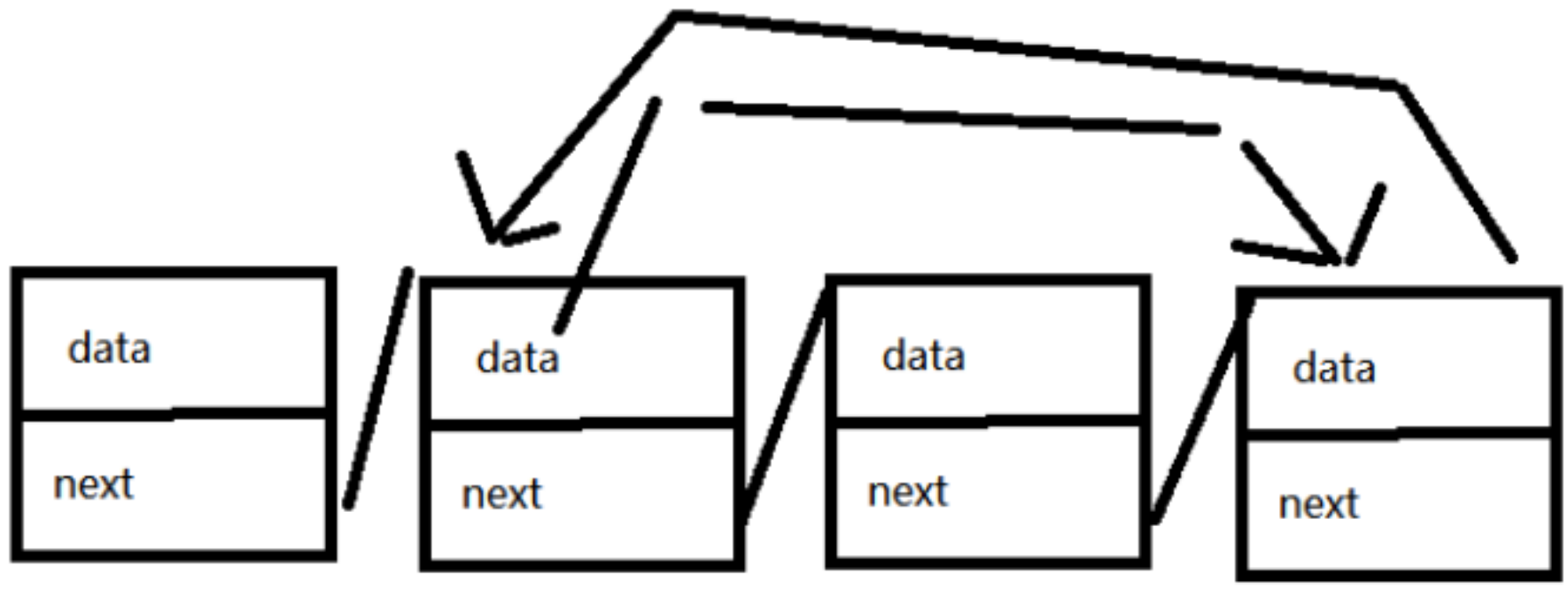




逆置链表 -方式 1，移动指针域



逆置链表 -方式 2，移动数据域



逆置操作

1. 判断首节点的 next 是否为 NULL ；
2. 判断首节点 next 的 next 是否为空，如果为空证明链表除首节点之外只有一个节点，所以不需要逆置；
3. 定义一个指针 last，指向首节点的 next 域，因为逆置之后，该域为链表尾节点；
4. 定义三个指针，分别代表前一个节点，当前节点，下一个节点；
5. 前节点指向链表首节点；
6. 当前节点指向链表首节点的 next 域；
7. 下一个节点为 NULL ；
8. 循环条件判断当前节点是否为 NULL ，如果为 NULL 退出循环；
   a) 下一个节点指向当前节点的下一个节点；
   b) 当前节点的下一个节点指向前一个节点；

c) 前一个节点指向当前节点；

d) 当前节点指向下一个节点；

9. 循环完成；
10. 设置 last 节点的 next 为 NULL ；
11. 设置链表首节点的 next 为前一个节点。

17.4 查找

17.4.1 顺序查找

顺序查找的过程为：从表的最后一个记录开始，逐个进行记录的关键字和给定值比较，如果某个记录的关键字与给定值相等，则查找成功，反之则表明表中没有所查找记录，查找失败。

17.4.2 二分查找

在一个已经排序的顺序表中查找，可以使用二分查找来实现。

二分查找的过程是：先确定待查记录所在的范围（区间），然后逐步缩小查找范围，直到找到或者找不到该记录为止。

假设指针 low 和 high 分别指示待查找的范围下届和上届，指针 mid 指示区间的中间值，即 mid=(low + high) / 2 。

17.5 排序

17.5.1 冒泡排序

冒泡排序首先将一个记录的关键字和第二个记录的关键字进行比较，如果为逆序（elem[1] > elem[2] ），则两个记录交换之，然后比较第二个记录和第三个记录的关键字，以此类推，直到第 n-1 个记录和第 n 个记录的关键字进行过比较为止。

上述过程称作第一次冒泡排序，其结果是将关键字最大的记录被安排到最后一个记录的位置上。然后进行第二次冒泡排序，对前 n-1 个记录进行同样操作，其结果是使关键字第二大记录被安置到第 n-1 位置上。直到将所有记录都完成冒泡排序为止。

17.5.2 选择排序

选择排序是每一次在 n −I + 1(i=1,2, ⋯n) 个记录中选取关键字，最小的记录作为有序序列

中第 i 个记录。

通过 n-i 次关键字间的比较，从 n-i+1 个记录中选取出关键字最小的记录，并 和第 i（1<=i<=n ）个记录交换之。