
step further

专题

起步：

认知与体验（硬件、软件、程序与C语言）

进阶：

判断与推理（流程控制方法、语句）

抽象与封装（模块设计方法、函数）

表达与转换（基本操作、数据类型）

提高：

构造与访问（数组、**指针**、结构）

归纳与推广（程序设计的本质）

指针及其运用

- 指针的基本概念
 - 概述
 - 指针类型的构造
 - 指针变量的定义与初始化
 - 指针的基本操作
- 用指针操纵数组
- 用指针在函数间传递数据
 - 指针类型参数
 - const的作用
 - 指针类型返回值
- 用指针访问动态变量
 - 通用指针与void类型
 - 动态变量的创建、访问和撤销
 - 内存泄漏与悬浮指针
- 多级指针
- 用指针操纵函数
- C++的引用类型

关于课件中内存单元示意图的说明

1个字节 (1byte, 8bit) 0x00003000

4个字节 (4byte, 32bit)

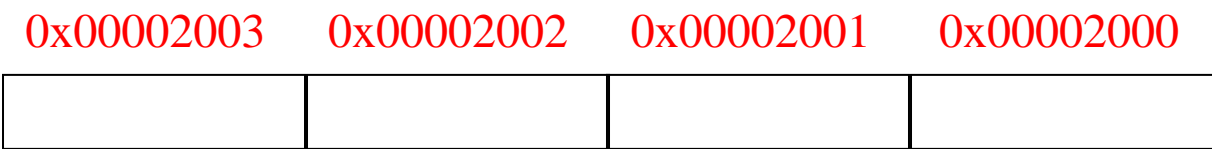
0x00002000

0x00002001

0x00002002

0x00002003

有时画成：



有时压缩成：

(只标记第一个单元的地址)



每一个内存单元都有一个地址（一般用十六进制数描述地址，开头的00可以省略）
（其中存储的数据一般用十进制或十六进制描述）

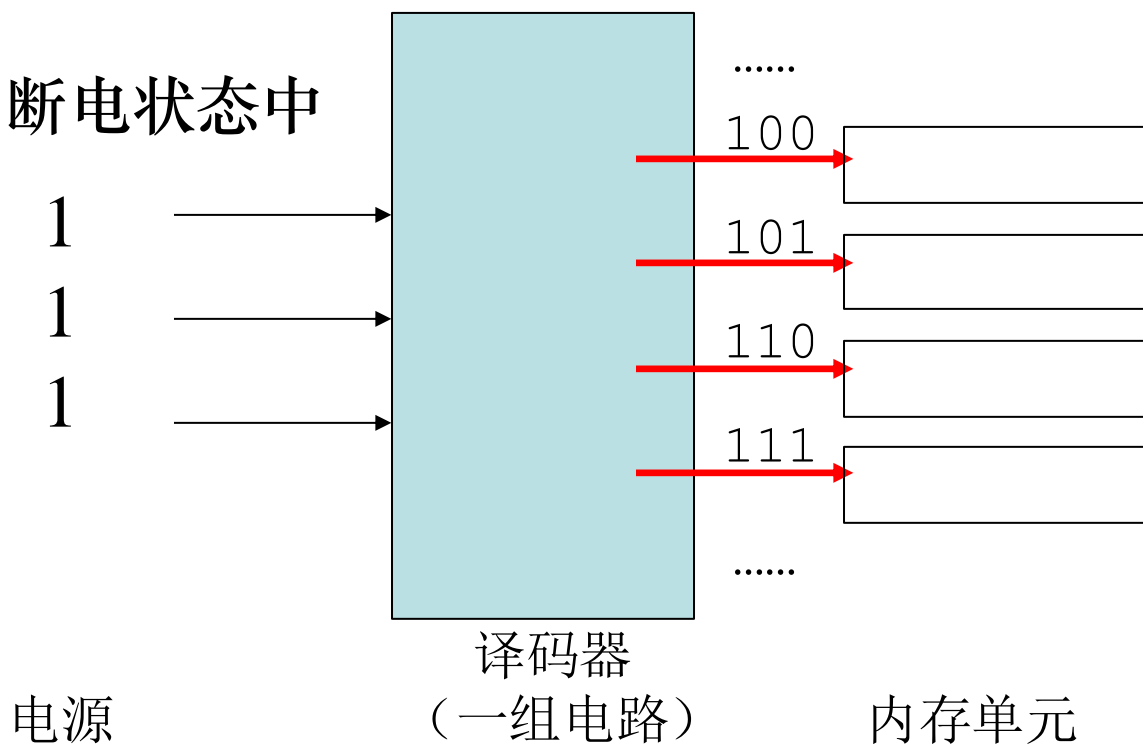
概述

□ 地址是什么？

- 计算机内存可看作由一系列内存单元组成，每个内存单元（容量为1个字节）由一个地址进行标识。

- 地址藏在哪儿？

- 藏在若干根地址线的通电、断电状态中



概述

□ 为什么不能用 int 来描述地址？

- 不是所有内存单元的地址都能在任一程序中使用，一般地，一个程序只能使用执行环境在编译和执行期间分配给该程序的内存单元的地址

能取的值有限

- 不能进行某些操作：

➤ `int address = 9;` // 9是有效的地址吗？

➤ `address *= 10;` // 乘以10之后，还是有效的地址吗？

能参与的操作有限

□ 地址是特殊的整数

- 值集、操作集与 int 不同

概述

□ 为什么在程序中要描述地址？不是有变量名吗？

- 变量的地址与变量名之间存在映射关系，一般情况下，高级语言程序通过这种映射关系用变量名访问数据。
- 如果在程序中知道**其他函数**中变量的地址（其他函数中的变量名不能使用），则可以省略数据传输环节，从而提高数据的访问效率。
- 在**没有变量名**的情况下，这种访问方式更具意义。

指针 (pointer)

- C语言用**指针类型**描述地址，通过对指针类型数据的相关操作，可以实现与地址有关的数据访问功能。
 - 指针类型数据通常占用1个**字**空间（与int型数据占用相等大小的空间）
- 指针
 - “指针”的含义未被严格界定，可能是指“地址”，也可能是指“指针类型”或“指针类型变量”

指针类型的构造

- 指针类型由一个表示变量类型（在这里又叫基类型）的关键字和一个*构造而成。
- 可以给构造好的指针类型取一个别名，作为指针类型标识符。
- 比如，

```
typedef int *Pointer;
```

```
//Pointer是类型标识符
```

```
//其值集为本程序中 int 型变量的地址
```


指针变量的定义

- 可以用构造好的指针类型来定义指针变量。

```
typedef int *Pointer;  
Pointer p;
```

- 也可以在构造指针类型的同时直接定义指针变量。

```
int *p;  
// 指针变量的变量名不包括*
```

- 多个定义有时可以合并写（只共用基类型），但不主张合并写。

- 比如，

```
int *p1, *p2; // p2前的*不能省略
```

```
int m = 3, n = 5, *px;      或      int *px, m = 3, n = 5;
```

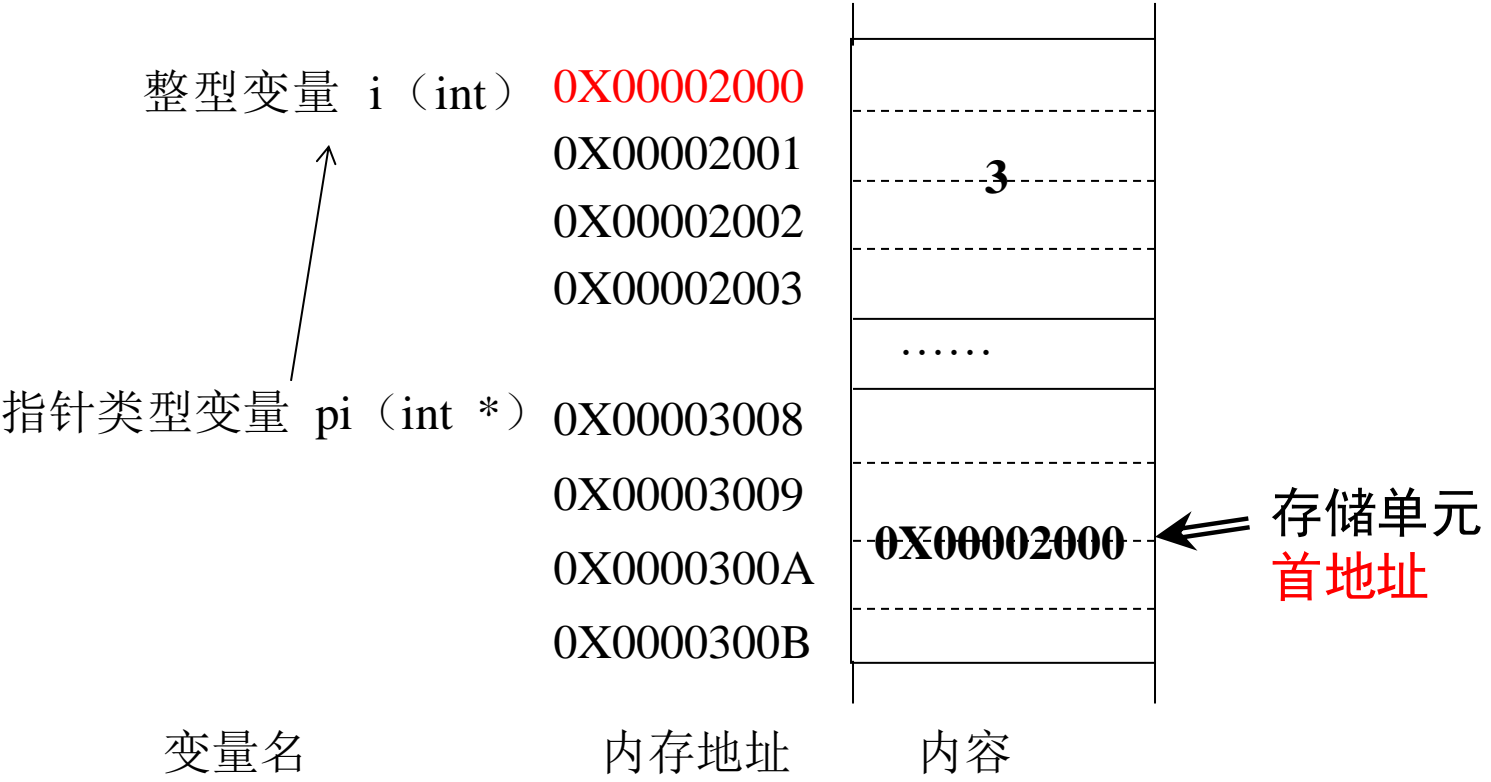
指针变量的初始化

□ 定义变量的时候明确指针变量存储哪个变量的地址，即指向哪个变量

```
int i = 3;  
int *pi = &i; //pi指向i
```

基类型

取地址操作符 (Address Operator)	
操作数:	各种类型的内存变量 数组元素 ...
不能是:	非左值表达式 字面常量 寄存器变量



-
- 用来初始化指针变量的变量要预先定义，且类型与指针变量的基类型要一致。

```
int i = 3;  
int *pi = &i;
```

```
double f = 3.2;  
double *pf = &f;      //不可以 double *pf = &i;
```

□ 也可以用另一个指针变量或指针常量（基类型要一致）来初始化一个指针变量。

■ 比如，

```
int i = 0;
int *pi = &i;
int *pj = pi;    //pj也指向变量i
```

■ 又比如，

```
int a[10];
int *pa = a;    //pa指向数组a
```

□ 还可以用0来初始化一个指针变量，表示该指针变量暂时不指向任何变量。

■ 比如，

```
int *pv = 0;    // 这里的 0 是一个空地址，通常用 NULL 代替
int *pv = NULL; // NULL 是系统定义的符号常量（一般在头文件stdio.h中）
```

□ 不可以将一个非0整数赋给一个指针变量，因为程序员指定的常数不一定是系统分配给该程序的内存单元地址，不一定能在该程序中访问其对应的空间。

■ 比如，

```
int *pn = 0x2000;
```

// 编译器不一定会报错，但执行时可能会引起系统故障

指针变量的赋值

□ 语句执行的时候（不是定义变量的时候）给指针变量一个地址

■ 注意事项参见指针变量的初始化

```
int i;  
int a[3];  
int *pi, *pa, *pj, *pv;  
pi = &i;  
pa = a;  
pj = pi;  
pv = NULL;
```

} 将首地址赋给指针变量

□ 指针变量必须先初始化或赋值，然后才能进行其他操作，否则其所存储的地址是不确定的，对它的操作会引起不确定的错误。

```
int *p;  
p++; ?
```

← p没有初始化或赋值就使用，警告

指针的基本操作

□ 可参与的操作很有限

- 取值操作

- 加/减一个整数

- 减法操作

- 关系/逻辑操作

- 下标操作

} 通常用于访问数组元素

□ 而且这些操作只在一定的约束条件下才有意义。不能参与的操作不一定会出语法错误，但可能会造成难以预料的结果（例如，将两个地址相加或相乘/除，结果不一定是有效的内存地址，或者即使地址有效，但其对应的内存单元未必能被该程序访问）。

取值操作

用指针变量通过取值操作访问数据时，地址决定访问的起点，基类型决定访问的终点（通过指定起点和终点即可访问数据）

□ 指针操作符（Indirection Operator）

```
int i = 3;
int *pi = &i;
```

这里的 * 不是指针操作符

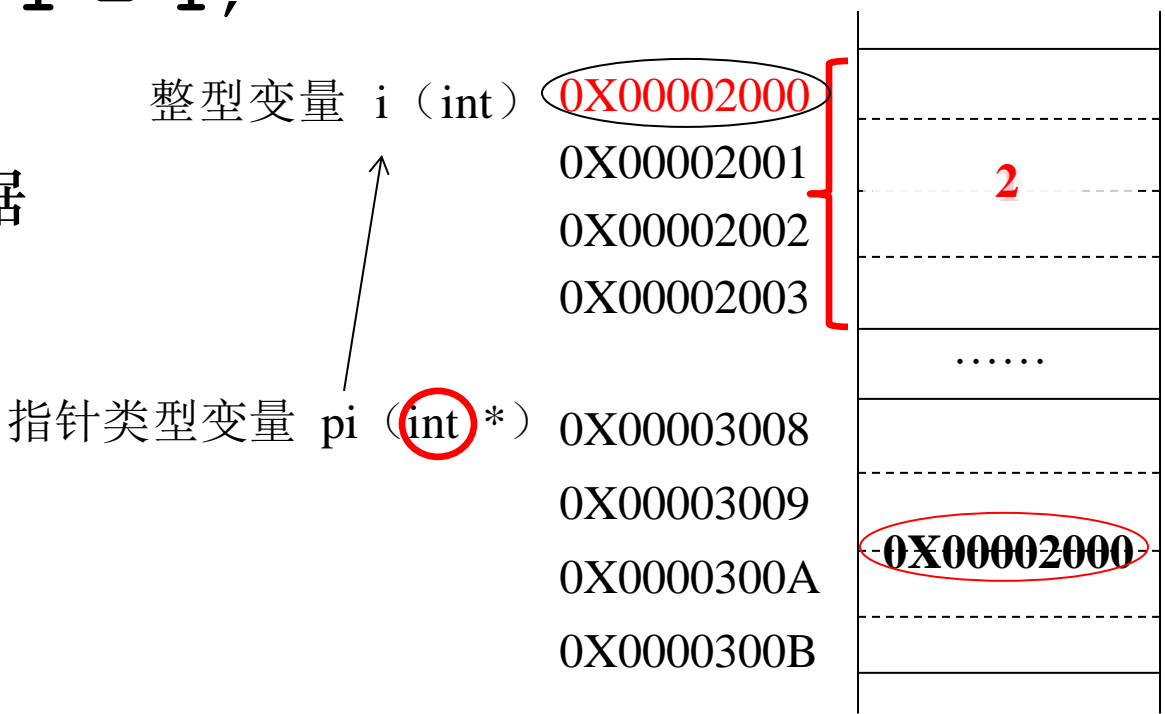
```
printf("%d ", *pi); //取值，相当于 printf("%d ", i);
*pi = 1;           //取值，相当于 i = 1;
```

■ 指针操作符的操作数应该是地址类型的数据

```
*(&i) = 2; //相当于 i = 2;
```

■ 取值操作与取地址操作是一对逆操作。

```
pi = &(*pi); //相当于 pi = &i;
```



指向一维数组元素的指针变量

□ 由于一维数组名表示第一个元素的地址，所以可将一维数组名赋给一个一级指针变量，此时称该指针变量指向这个数组的第一个元素。

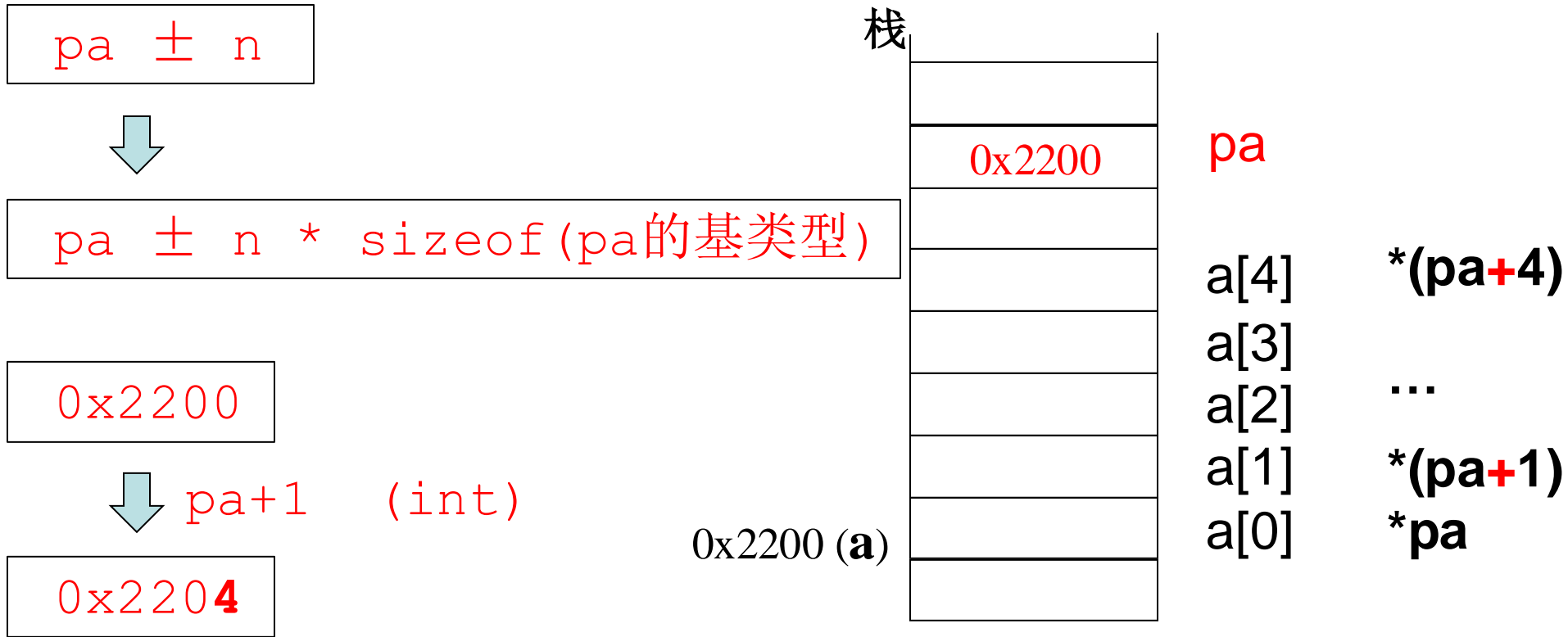
■ 比如，

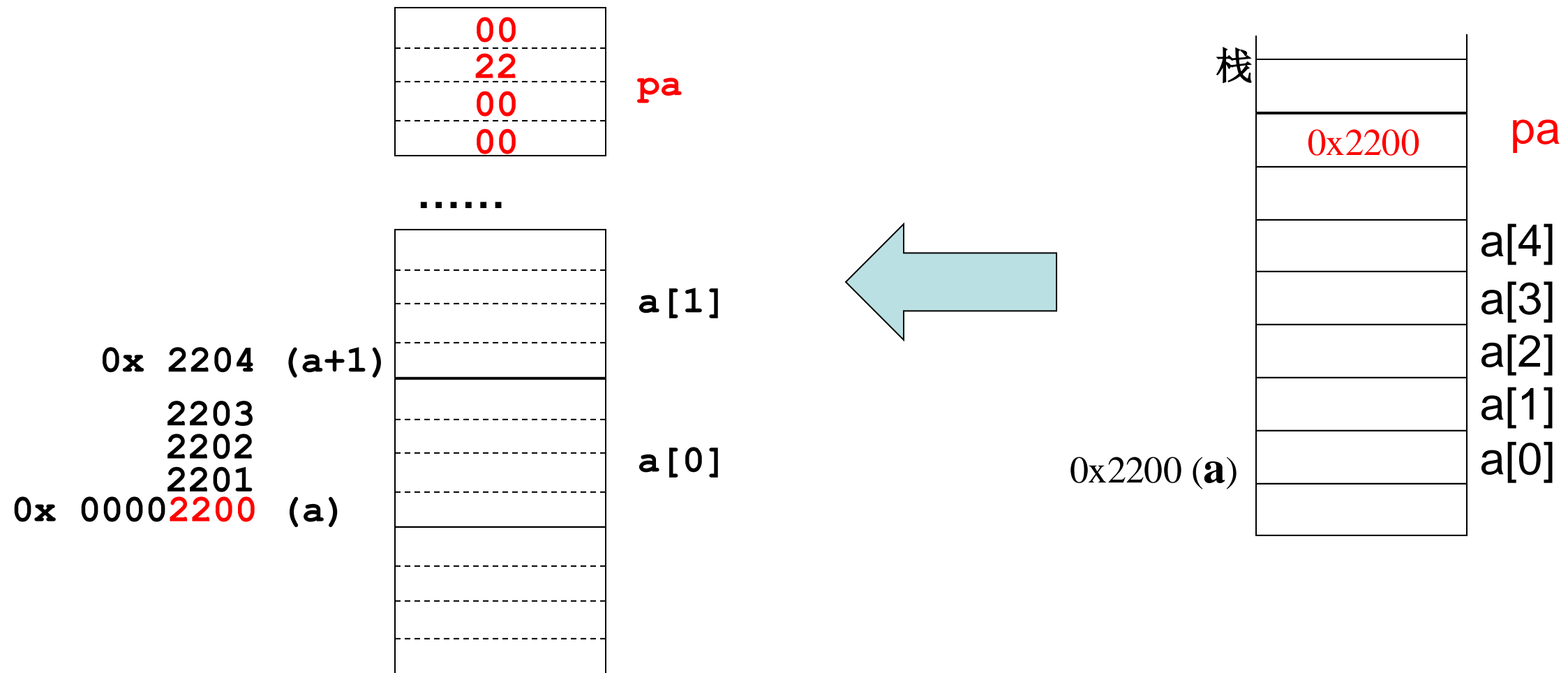
```
int a[5];
```

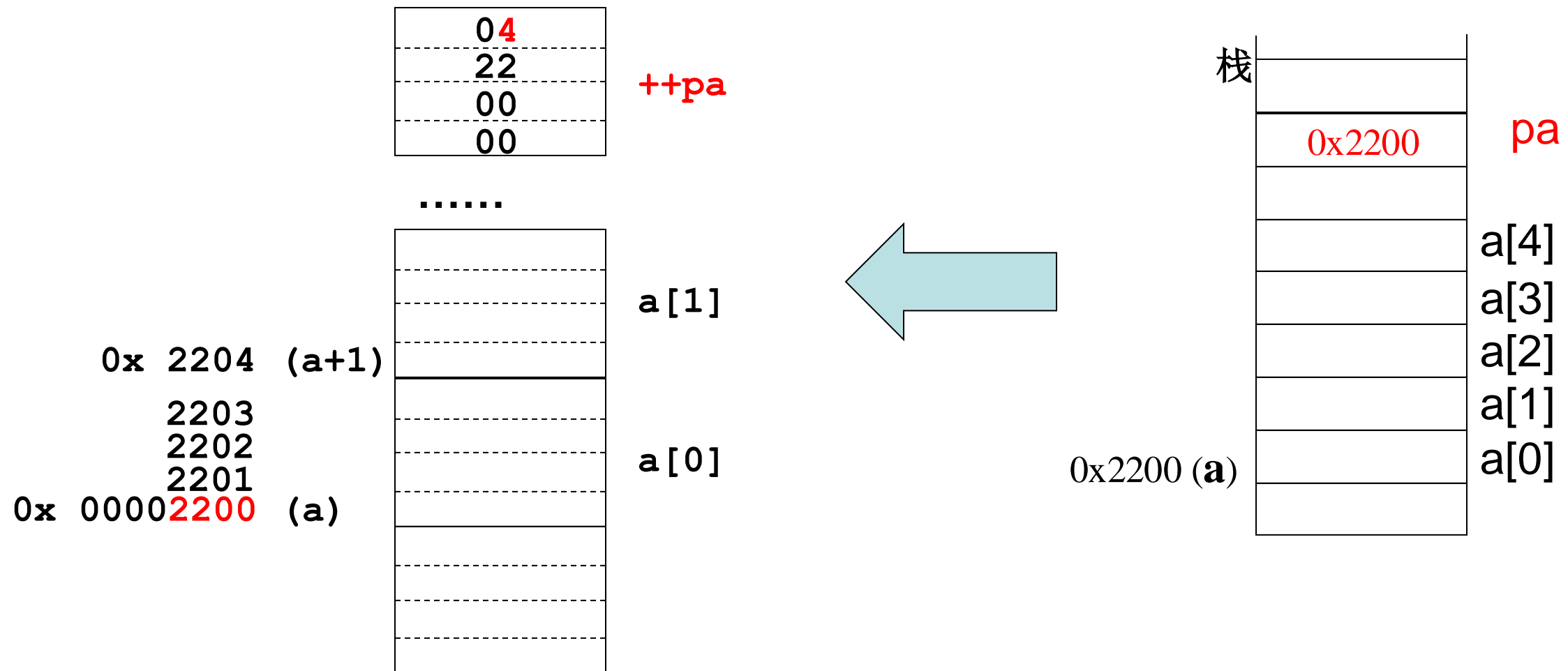
```
int *pa = a;    //相当于int *pa = &a[0];
```

■ 当一个指针变量指向一个数组的元素时，该指针变量可以存储数组的任何一个元素的地址，即`pa`先存储`a[0]`的地址，不妨设为`0x00002000` (简作`2000`)，然后，`pa`的值可以变化为`2004`，`2008`，`200C`，`2010`，于是可以通过`pa`来访问数组`a`的各个元素。

- 加/减一个整数：地址加/减一个整数，成为另一个地址。
 - 加/减一个整数 n 之后的结果并非在原来地址值的基础上加/减 n ，而是 n 的若干倍，具体倍数由基类型决定，即加/减 $n * \text{sizeof}(\text{基类型})$ 。
 - 操作前指针变量指向某数组的一个元素，操作后的结果指向该数组的另一个元素（不能越界）。







❑ 减法操作：两个基类型相同的地址可以相减

- 相减结果为两个地址之间可存储基类型数据的个数，通常用来计算一个数组两个元素之间的距离

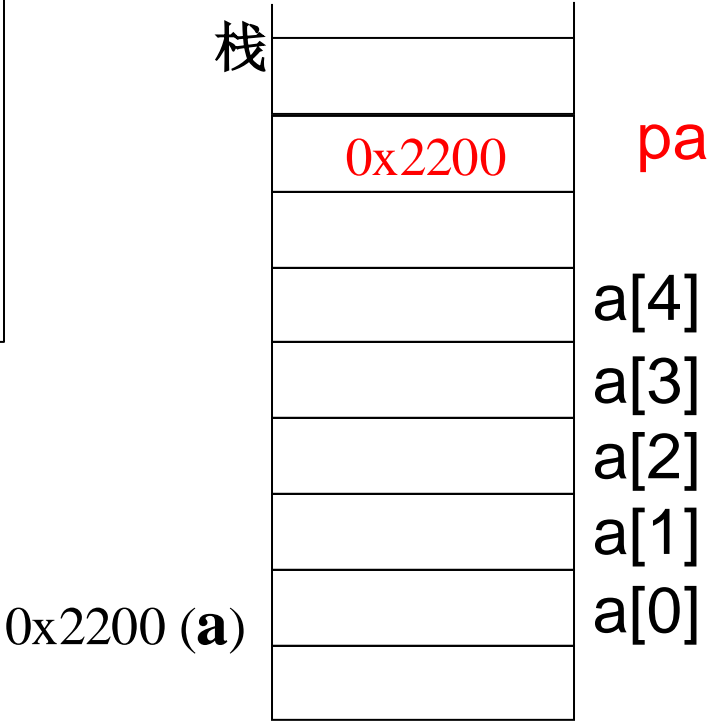
```
int s = 0;
for(int *pa=a; pa < a+n; ++pa)
    s += *pa;
int offset = pa - a;
if(offset != n) .....
```

int * + int → int *

int * - int * → -int

int * - int → int *

int * - int * → int



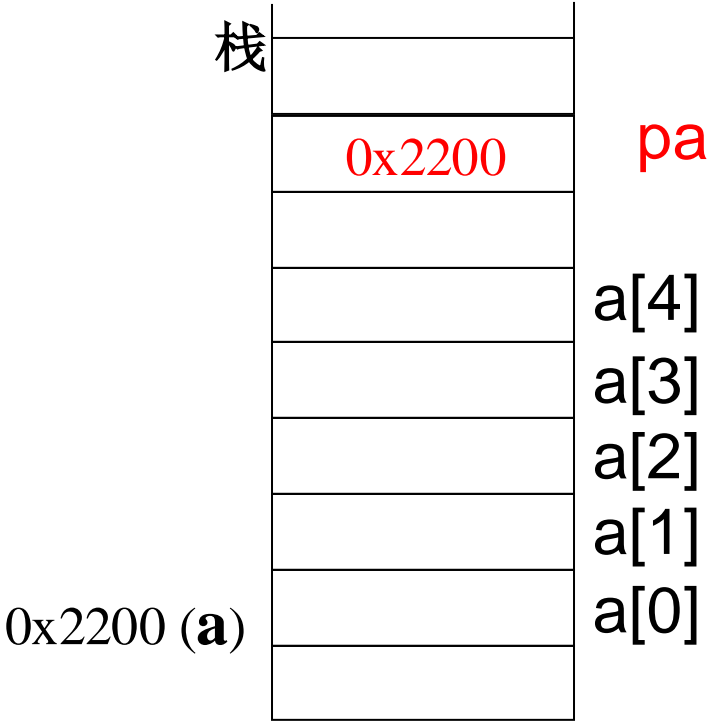
□ 关系/逻辑操作：两个基类型相同的地址可以比较

- 比较结果为逻辑值，通常用于判断一个数组两个元素的内存位置前后关系

```
if (pa <= a) .....
```

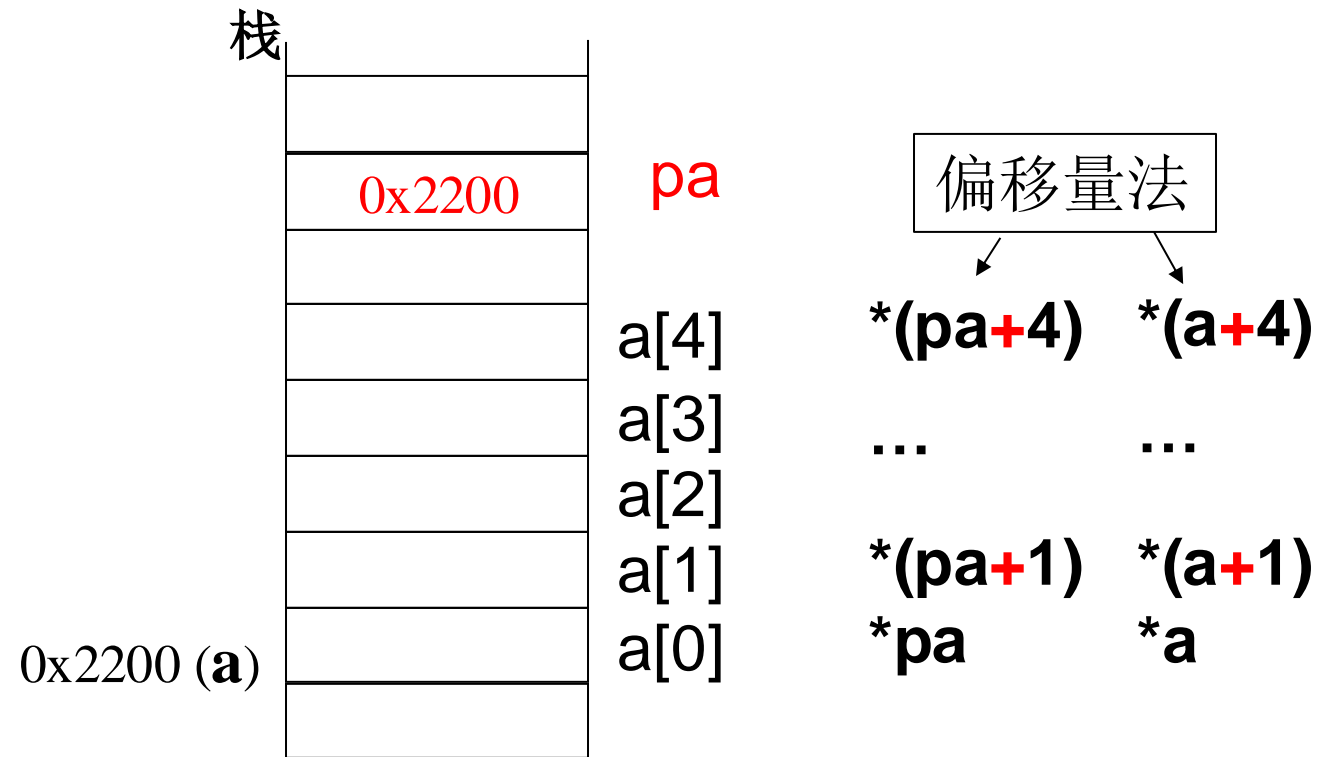
- 也可以判断一个地址是否为NULL，
以明确该地址是否为0（不是某个实际内存单元的地址）

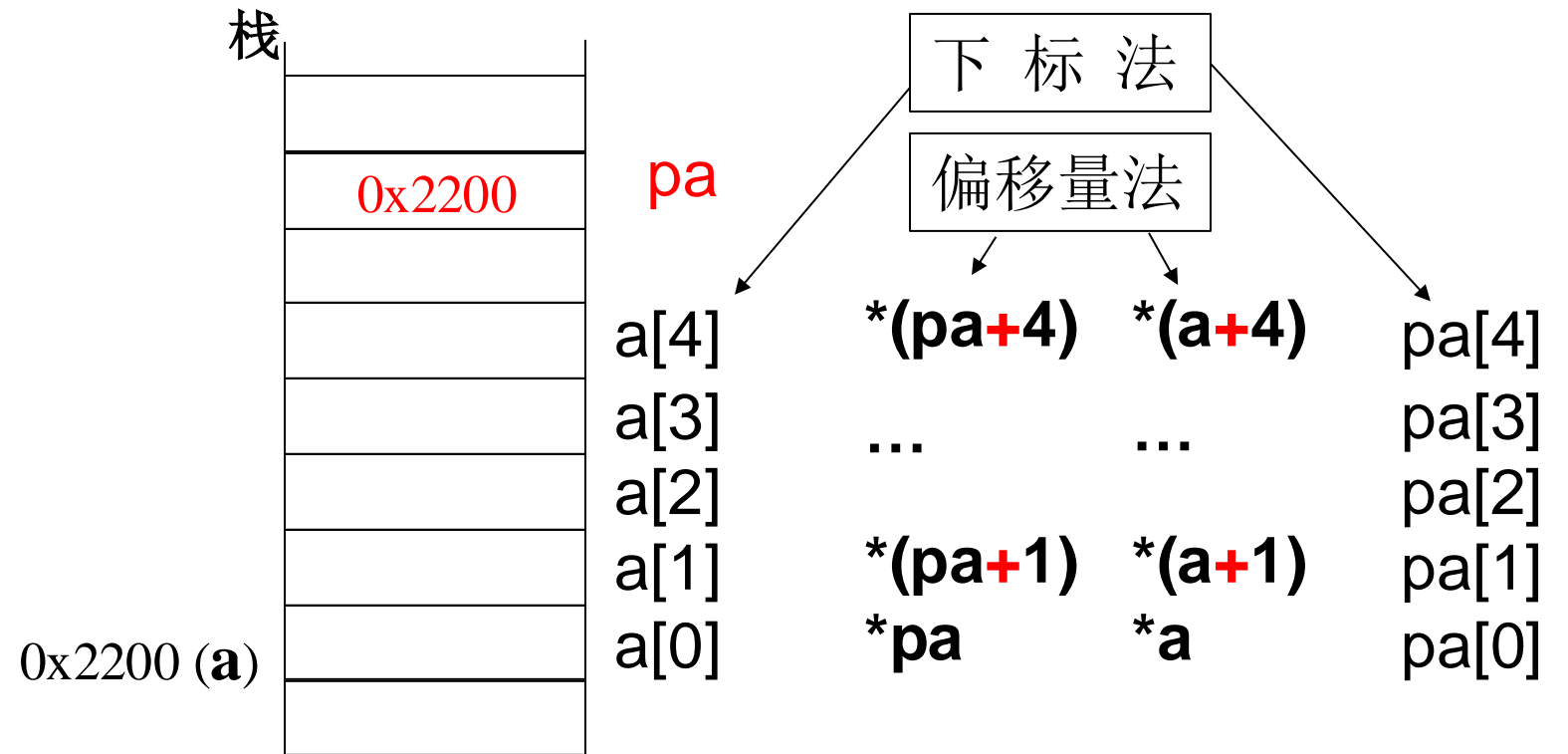
```
if (pa == NULL) .....
```

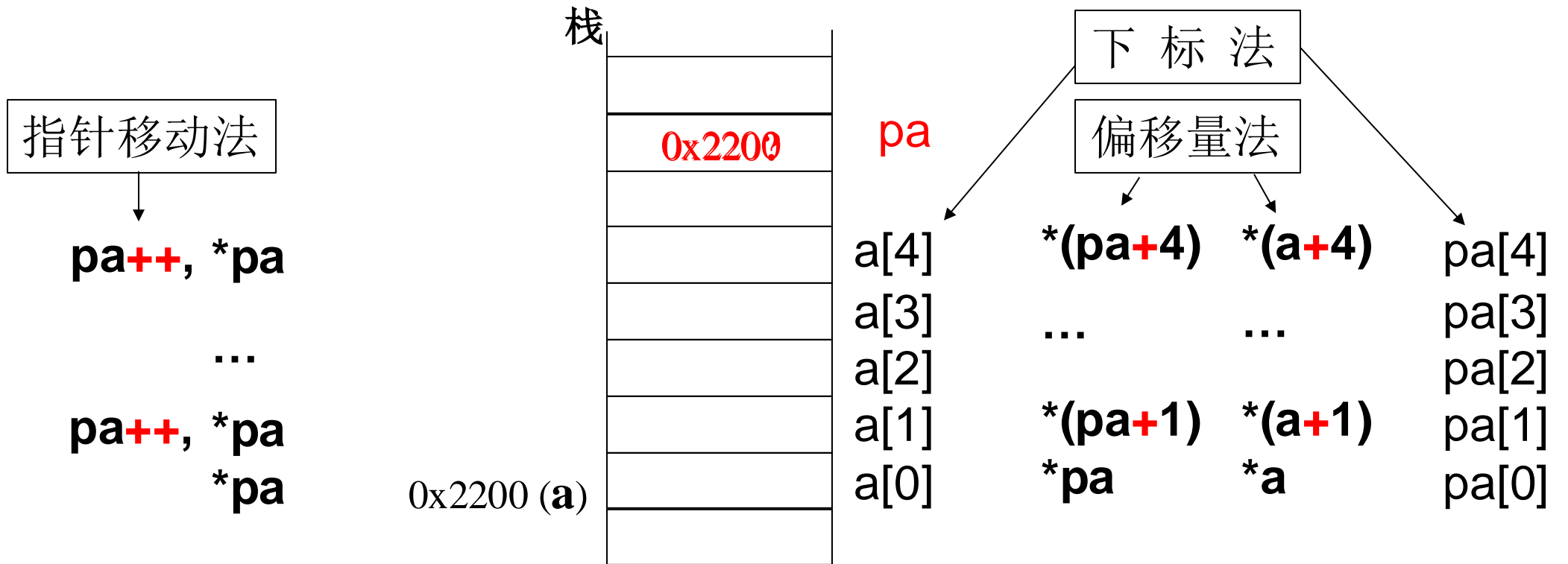


□ 指向数组元素的指针变量 vs. 数组名

- 数组名可以代表第一个元素的地址，是地址常量
 - 数组名的值不能被修改
- 指向数组元素的指针变量
 - 用来存放一个数组各个元素的地址
 - 一般情况下，这个变量的初始值为数组名代表的地址
 - 经过操作，其值可能会是其他元素的地址







```
int a[5] = {8,9,3,2,0};  
int s = 0;  
for(int i = 0; i < n; ++i)  
    s += a[i]; // s += *(a+i);
```

```
int a[5] = {8,9,3,2,0};  
int *pa = a;  
int s = 0;  
for(int i = 0; i < n; ++i)  
    s += pa[i]; // s += *(pa+i);
```

```
int a[5] = {8,9,3,2,0};  
int *pa = a;  
int s = 0;  
for(int *pa=a; pa < a+n; ++pa)  
    s += *pa;
```

```
int a[5] = {8,9,3,2,0};  
int s = 0;  
for(; a < a+n; ++a)  
    s += *a;
```

□ 通过指针变量操纵数组时，更加要注意防止越界问题

□ 判断下列对指针类型数据的初始化和操作是否恰当

```
    int m = 3;
    float f;
1.  int *pi;
2.  ++pi;
3.  pi = &f;
4.  pi = &a;
5.  float *pf = &f;
    int a[5];
    char ch;
6.  char *pch = &ch;
7.  int *pa = &a;
8.  int *pv = NULL;
9.  int *pn = pv;
10. pn = 100;
11. pn = (int *)4096;
12. printf("%d %f\n", *4096, *(m+f));
```

□ 判断下列对指针类型数据的初始化和操作是否恰当

```
    int m = 3;
    float f;
1.  int *pi;
2.  ++pi;
3.  pi = &f;
4.  pi = &a;
5.  float *pf = &f;
    int a[5];
    char ch;
6.  char *pch = &ch;
7.  int *pa = &a;
8.  int *pv = NULL;
9.  int *pn = pv;
10. pn = 100;
11. pn = (int *)4096;
12. printf("%d %f\n", *4096, *(m+f));
```

指针及其运用

□ 指针的基本概念

- 概述
- 指针类型的构造
- 指针变量的定义与初始化
- 指针的基本操作

以上用简单的例子，示意指针变量初始化、赋值或操作中的注意事项。其实，

□ 用指针操纵数组

□ 用指针在函数间传递数据

- 指针类型参数
- const的作用
- 指针类型返回值

指针变量的初始化通常是实参传给形参

□ 用指针访问动态变量

- 通用指针与void类型
- 动态变量的创建、访问和撤销
- 内存泄漏与悬浮指针

指针变量的赋值通常操作于动态变量

□ 多级指针

□ 用指针操纵函数

□ C++的引用类型

用指针在函数间传递数据

重点

- 实际参数是地址
- 形式参数是指针变量
- 被调函数通过指针变量和取值操作直接访问主调函数中实参那里的数据
- 这种函数调用方式通常叫做**传址调用**
 - 以便提高函数间大量数据（比如数组）的传递效率
 - 或改变主调函数中的数据（利用函数的副作用）
- 实际上，C语言中，写成数组定义形式的形参也是按指针变量看待的，即只给形参分配存放一个字的内存空间，以存储实参传来的地址

□ 传值调用

对比

- 实参的副本单向传给形参
- 被调函数访问的不是主调函数的数据区

传址调用（指针型参数）

例6.1 `int Sum(int *x, int n)`

```
int Sum(int x[], int n)
```

```
{ int s = 0;
```

```
  for(int i = 0; i < n; ++i)
```

```
    s += x[i];
```

```
    s += *(x+i);
```

```
  return s;
```

```
}
```

```
int main()
```

```
{ int a[5] = {8,9,3,2,0};
```

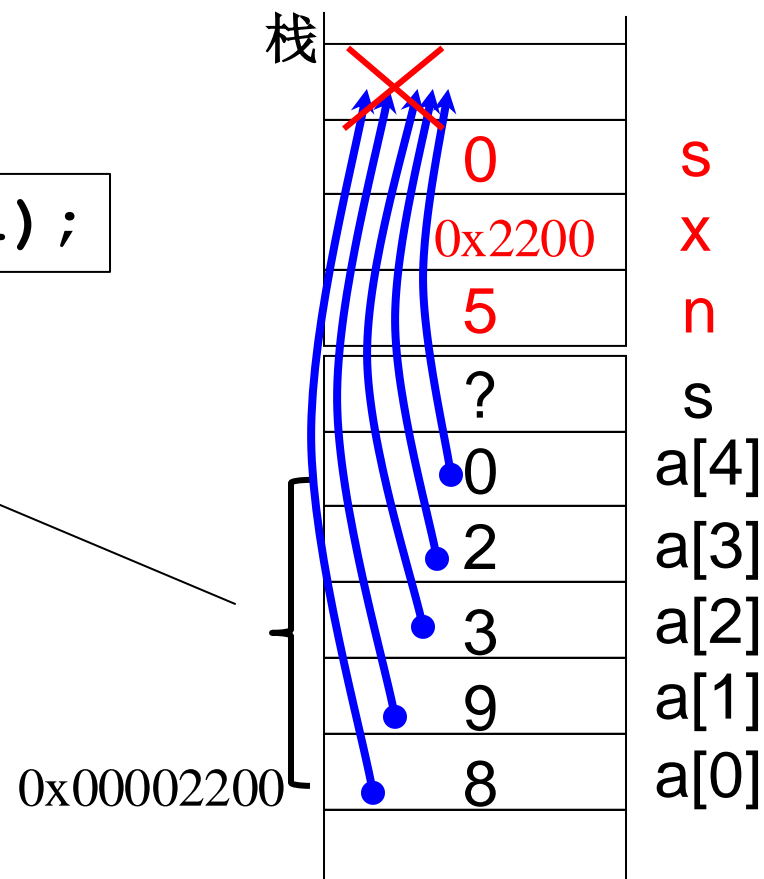
```
  int s = Sum(a, 5);
```

```
  printf("%d", s);
```

```
  return 0;
```

```
}
```

```
int *x = a;
```

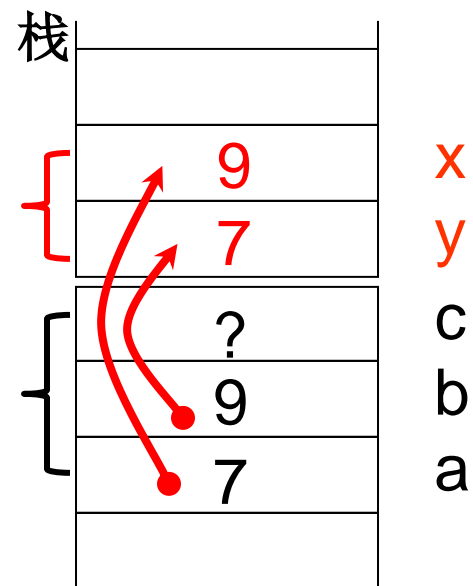


对比：传值调用

□ 实参的副本传给形参

```
int Sum(int x, int y)
{
    return x + y;
}
```

```
int main()
{
    int a, b, c;
    scanf("%d%d", &a, &b);
    c = Sum(a, b);
    printf("%d", c);
    return 0;
}
```



函数的副作用

- 指针类型数据用作函数的形参，除了可以提高函数间大量数据的正向传递效率，还存在一种**函数的副作用**，即在被调函数中可以通过指针变量修改实参的值。
- 被调函数的return语句一般只能返回一个值，如果需要返回多个值，则可以利用这种函数的副作用实现函数间数据的反向传递。

例6.2 两个变量值的交换函数

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

int main( )
{
    int a = 5, b = 9;
    MySwap(a, b);
    printf("%d, %d", a, b);
    return 0;
}
```

`int x = a`
`int y = b`

x	5	↕	a	5
y	9		b	9

```
void MySwap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main( )
{
    int a = 5, b = 9;
    MySwap(&a, &b);
    printf("%d, %d", a, b);
    return 0;
}
```

`int *x = &a`
`int *y = &b`

x	0x...	↕	*x	a	5	↕
y	0x...		*y	b	9	

□ 传值调用还是传址调用?

```
void MySwap(int *x, int *y)
{
    int *temp = x;
    x = y;
    y = temp;
}

int main( )
{
    int a = 5, b = 9;
    MySwap(&a, &b);
    printf("%d, %d", a, b);
    return 0;
}
```

`int *x = &a`
`int *y = &b`

x	0x...	a	5
y	0x...	b	9

```
void MySwap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main( )
{
    int a = 5, b = 9;
    MySwap(&a, &b);
    printf("%d, %d", a, b);
    return 0;
}
```

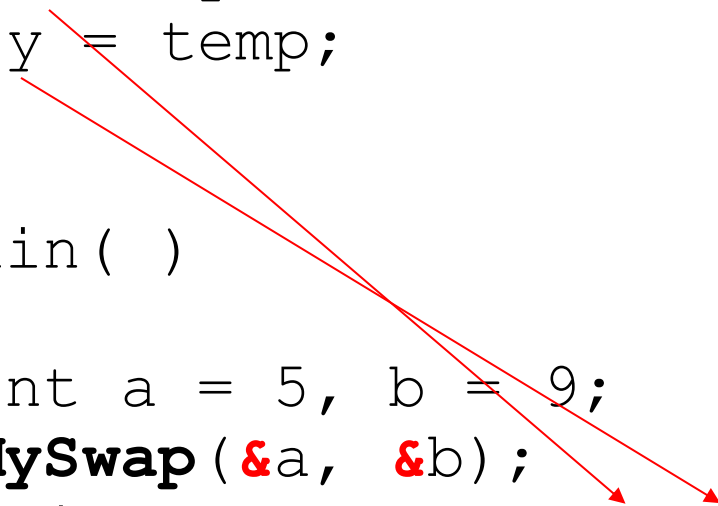
`int *x = &a`
`int *y = &b`

x	0x...	*x	a	5
y	0x...	*y	b	9

函数间的通讯方式Ⅲ

```
void MySwap(int *x, int *y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main( )
{
    int a = 5, b = 9;
    MySwap(&a, &b);
    printf("%d, %d", a, b);
    return 0;
}
```



函数的副作用

传址调用的双重作用

例6.3 void Fun(int *pa, int n)

```
{  
    for(int i = 0; i < n-1; ++i)  
        pa[n - 1] += pa[i];  
} //又利用函数的副作用实现了数据的反向传递(修改了最后一个元素的值)
```

```
int main( )  
{  
    int *pa = a
```

```
    int a[6] = {1, 3, 5, 7, 9}; //注意a[5]初始化为0  
    Fun(a, 6);  
    printf("%d \n", a[5]);  
    return 0;
```

} //既利用传址调用提高了数据正向传递的效率

栈		i pa n a[5] a[4] a[3] a[2] a[1] a[0]
	5	
	0x2200	
	6	
	25	
	9	
	7	
	5	
	3	
0x2200	1	

const

```
void Fun(const int *pa, int n)
{
    for(int i = 0; i < n-1; ++i)
        pa[n - 1] += pa[i]; // *pa = sum;
} //禁止了函数的副作用，就不能修改数组元素的值了
```

```
int sum = 0;
for(int i = 0; i < n-1; ++i)
    sum += pa[i];
printf("%d \n", sum);
```

```
int main( )
{
    int a[6] = {1, 3, 5, 7, 9}; //注意a[5]初始化为0
    Fun(a, 6);
    printf("%d \n", a[5]);
    return 0;
} //仅利用传址调用提高数据正向传递的效率
```

□ **const**还可以用来定义一个常量。

```
const double PI = 3.1415926;
```

```
#define PI 3.14159
```

- 用**const**定义常量时，必须初始化
- **const**的不同位置含义不同

```
int i = 0, j = 0;  
const int M = 0;  
const int *p1;  
int * const P2 = &i;  
const int * const P3 = &M;
```

p1 = &M

p1 = &i //只不过不能通过 p1 修改 i

~~*p1 = 1;~~

~~P2 = &M;~~

~~P2 = &j;~~

*P2 = 1;

~~int * const P2 = &M~~ //防止通过 P2 修改 M

```
int *q = &i;  
q = &j;  
q = &M;
```

~~P3 = &i;~~

~~*P3 = 1;~~

const int * const P3 = &i //只不过不能通过P3修改i的值

在主调函数中进行取值操作

□ 指针类型返回值

□ 例6.4

```
int *Max(const int ac[ ], int num)
{
    int max_index = 0;
    for(int i = 1; i < num; ++i)
        if(ac[i] > ac[max_index])
            max_index = i;
    return (int *)&ac[max_index];
}
```

将const int 型地址转换成 int 型地址

```
int main( )
{
    int a[ ] = {1, 2, 5, 4, 3};
    int *p = Max(a, N);
    *p = 0; //用0替换数组a中的最大数
    for(int i = 0; i < N; ++i)
        printf("%d \t", a[i]);
    return 0;
}
```

□ 注意，函数一般不能返回局部变量的地址。

■ 例如，

```
int *F( )
{
    int m, n;
    scanf("%d%d", &m, &n);
    if(m > n)
        return &m;
    else
        return &n;
} //调用者获得地址时，m、n空间已经释放，地址无意义
```

指针及其运用

□ 指针的基本概念

- 概述
- 指针类型的构造
- 指针变量的定义与初始化
- 指针的基本操作

以上用简单的例子，示意指针变量初始化、赋值或操作中的注意事项。其实，

□ 用指针操纵数组

□ 用指针在函数间传递数据

- 指针类型参数
- const的作用
- 指针类型返回值

指针变量的初始化通常是实参传给形参

□ 用指针访问动态变量

- 通用指针与void类型
- 动态变量的创建、访问和撤销
- 内存泄漏与悬浮指针

指针变量的赋值通常操作于动态变量

□ 多级指针

□ 用指针操纵函数

□ C++的引用类型

通用指针与void类型

- void: 空类型的关键字
- 空类型的值集为空，在计算机中不占空间，一般不能参与基本操作
- 通常用来描述不返回数据的函数的返回值，以及不需要参数的函数的形参
- 还可以作为指针类型的基类型，形成通用指针类型（void *）。

```
void *malloc(unsigned int size);
```

```
void free(void *p);
```

-
- 通用指针类型变量不指向具体的数据，不能用来访问数据，但**任何指针类型都可以隐式转换为通用指针类型**，而且在将通用指针类型转换回原来的指针类型时不会丢失信息。
 - 通用指针类型经常作为函数的**形参和返回值的类型**，以提高所定义的函数（比如创建动态变量的库函数）的通用性。

内存分配方式

□ 系统为程序中的数据分配内存空间一般有两种方式

■ 静态：程序开始执行前在静态数据区分配空间

- 先由编译器对其定义行进行特殊处理（规划所需内存，分析、处理其初始化值），程序执行时，由执行环境在静态数据区为之分配空间，并写入初始化值，若未初始化则写入初值0，此后程序可获取或修改其值，直到整个程序执行结束才收回其空间。

■ 动态：程序执行时在栈区或堆区分配空间

- 在栈区：定义行有对应的目标代码，通过代码的执行在栈区获得空间，若已初始化则获得初始化值，若未初始化则其初值是内存里原有的值，此后程序可以访问其内存空间，获取或修改其值，一旦复合语句执行结束即收回其内存空间
- 在堆区：由程序员编写的相关代码（调用malloc库函数）申请内存空间，通过代码的执行在堆区获得空间，由于没有初始化，其初值为内存里原有的值，此后程序可以访问其内存空间，进行赋值操作，以及获取或修改其值，最后通过程序员编写的相关代码（调用free库函数）释放其内存空间。如果程序员忘记编写释放其内存空间的代码，则要等整个程序执行结束时才收回其内存空间

内存分配方式-1

```
void MyMax(int, int, int);
int max = 0;
int main( )
{
    int m1, m2, m3;
    scanf("%d%d%d", &m1, &m2, &m3);
    while(m1 != m2 && m2 != m3)
    {
        MyMax(m1, m2, m3);
        printf("%d \n", max);
        scanf("%d%d%d", &m1, &m2, &m3);
    }
    return 0;
}
```

//全局变量

静态内存分配方式

```
void MyMax(int n1, int n2, int n3)
{
    static int count = 0;    //静态变量
    if(n1 >= n2)
        max = n1;
    else
        max = n2;
    if(max < n3)
        max = n3;
    ++count;
}
```

内存分配方式-2

```
void MyMax(int, int, int);
int max = 0;
int main( )
{
    int m1, m2, m3; //自动变量
    scanf("%d%d%d", &m1, &m2, &m3);
    while(m1 != m2 && m2 != m3)
    {
        MyMax(m1, m2, m3);
        printf("%d \n", max);
        scanf("%d%d%d", &m1, &m2, &m3);
    }
    return 0;
}
```

动态内存分配方式

```
void MyMax(int n1, int n2, int n3) //形参
{
    static int count = 0;
    if(n1 >= n2)
        max = n1;
    else
        max = n2;
    if(max < n3)
        max = n3;
    ++count;
}
```


内存分配方式-2

```
int Sum(int *, int)
```

```
int main( )
```

```
{
```

```
    int a[5] = {8, 9, 3, 2, 0}; //自动变量
```

```
    int s = Sum(a, 5);
```

```
    printf("%d", s);
```

```
    return 0;
```

```
}
```

动态内存分配方式

```
int Sum(int *x, int n) //形参
```

```
{    int s = 0;
```

```
    for(int i = 0; i < n; ++i)
```

```
        s += *(x+i);
```

```
    return s;
```

```
}
```

内存分配方式-2'

例6.5 `#include <stdlib.h>`

```
int Sum(int *, int)
```

```
int main( )  
{
```

```
    int n; //自动变量  
    scanf("%d", &n);
```

```
    int *pda; //自动变量
```

```
    pda = (int *)malloc(n * sizeof(int)); //创建了动态变量(数组)
```

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

```
    int s = Sum(pda, n);  
    printf("%d", s);  
    return 0;
```

```
}
```

动态内存分配方式

```
int Sum(int *x, int n) //形参  
{  
    int s = 0;  
    for(int i = 0; i < n; ++i)  
        s += *(x+i);  
    return s;  
}
```

动态变量的创建与撤销

□ malloc库函数的原型是

void *malloc(unsigned int size); // 在堆区分配size个单元

■ `int *pdi = (int *)malloc(sizeof(int));`

■ `double *pdf = (double *)malloc(sizeof(double));`

通用指针类型: **void ***

□ free库函数的原型是

`void free(void *p);`

■ `free(pdi);`

■ `free(pdf);`

动态变量的创建与撤销

```
#include <stdlib.h>
```

```
int Sum(int *, int)
```

```
int main( )
```

```
{
```

```
    int n;
```

```
    scanf("%d", &n);
```

```
    int *pda; int (*pdaa)[2] = (int (*)[2])malloc(n * sizeof(int) * 2);
```

```
    pda = (int *)malloc(n * sizeof(int));
```

```
        int (*pdaa)[2] = (int (*)[2])malloc(2 * n * sizeof(int));
```

```
    for(int i = 0; i < n; ++i)
```

```
        scanf("%d", &pda[i]);
```

```
    int s = Sum(pda, n);
```

```
    free(pda);
```

```
    printf("%d", s);
```

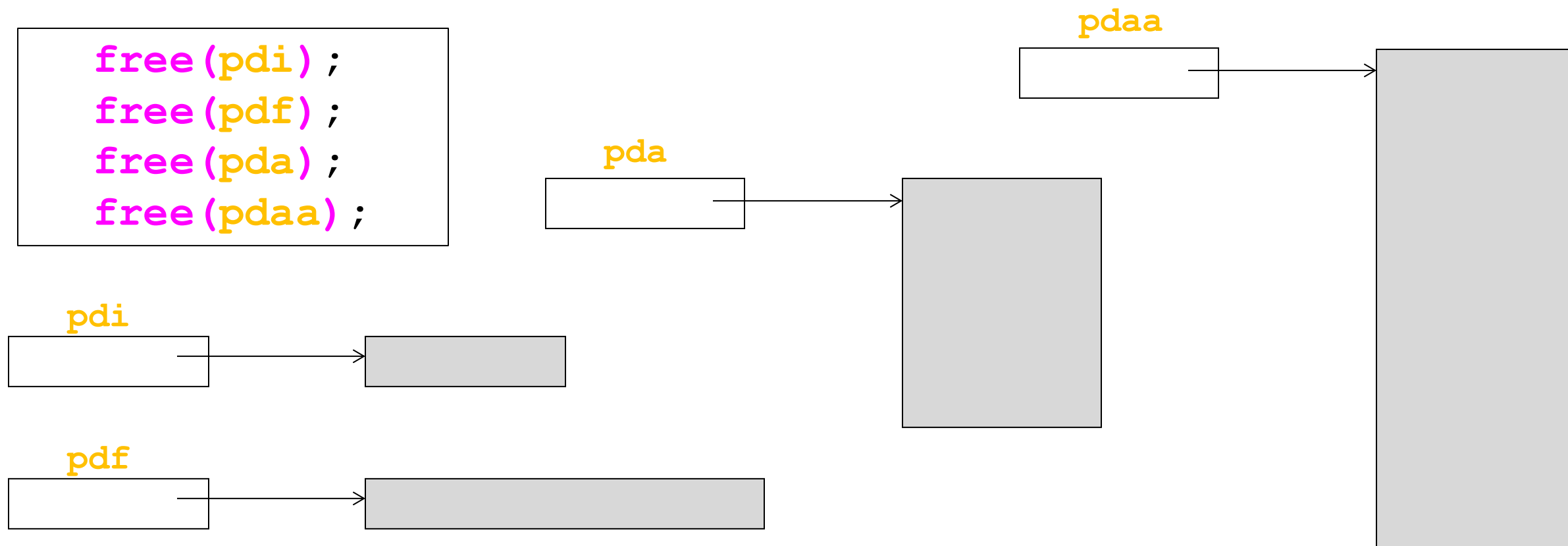
```
    return 0;
```

假定二维数组有 n 行

```
free(pdaa);
```

动态变量的撤销

- 撤销的是动态变量所在的堆区内存空间，即图中的阴影区域，而不是指向它的指针变量，定义的指针变量pd、pdf、pda、pdaa都在栈区



C vs. C++

```
#include <stdlib.h>
```

```
int *pd = (int *)malloc(sizeof(int));  
double *pda = (double *)malloc(n * sizeof(double));  
int (*pdaa)[10] = (int (*)[10])malloc(n * sizeof(int) * 10);
```

```
free(pd);  
free(pda);  
free(pdaa);
```

```
#include <iostream>
```

```
int *pd = new int;  
double *pda = new double[n];  
int (*pdaa)[10] = new int[n][10];
```

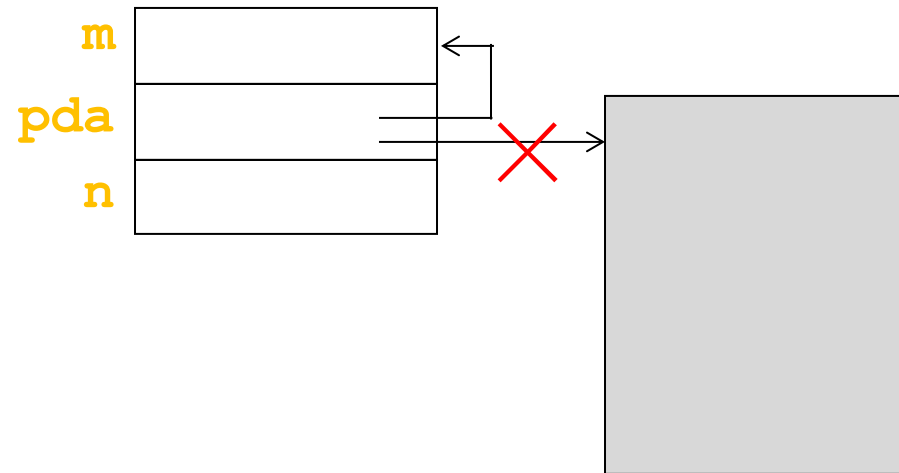
```
delete pd;  
delete []pda;  
delete []pdaa;
```

内存泄露

```
int n;  
scanf("%d", &n);  
int *pda = (int *)malloc(n * sizeof(int));  
.....
```

```
free(pda);
```

```
int m;  
pda = &m; // pda指向的堆区泄露了
```



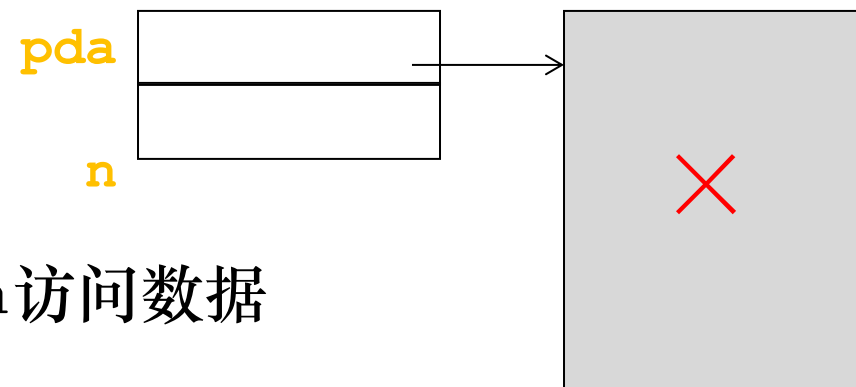
悬浮指针

```
int n;  
scanf("%d", &n);  
int *pda = (int *)malloc(n * sizeof(int));  
.....
```

```
free(pda);
```

```
pda = NULL; //清理 pda 里的地址，以免乱指
```

```
*pda = 3; // pda变为“悬浮指针”，不能通过pda访问数据
```



□ 实际应用

1) 对输入的10个整数进行排序，可以用数组来实现：

```
const int N = 10;
```

```
int i, a[N];
```

```
for(i = 0; i < N; ++i)
```

```
    scanf("%d", &a[i]); //cin >> a[i];
```

```
Sort(a, N);
```

```
...
```

2) 对输入的**若干个**整数进行排序（先**输入整数的个数n**，后输入n个整数），可以用数组来实现（新C标准）：

```
int n, i;  
scanf("%d", &n); //cin >> n;  
int a[n];
```

有些新标准，
允许数组长度为变量，
此法适用于支持这类新标准的编译器。

```
for(i = 0; i < n; ++i)  
    scanf("%d", &a[i]); //cin >> a[i];
```

```
Sort(a, n);
```

```
...
```

2) 对输入的**若干个**整数进行排序（先**输入整数的个数n**，后输入n个整数），可以用动态数组来实现：

老标准和有些新标准，
不允许数组长度为变量，
此法适用于支持这类标准的编译器。

```
int n, i;
int *pda;

scanf("%d", &n); //cin >> n;

pda = (int *)malloc(n * sizeof(int));    // pda = new int[n];

for(i = 0; i < n; ++i)
    scanf("%d", &pda[i]); //cin >> pda[i];

Sort(pda, n);

...

free(pda); //delete []pda;

pda = NULL;
```

3) 对输入的**若干个**正整数进行排序（先输入各个正整数，**最后输入一个结束标志 -1**）

? //用不断调整动态数组的大小来实现

```
Sort(pda, count);
```

```
... //输出排序后的数组
```

```
free(pda); //delete []pda;
```

用不断调整动态数组的大小来实现

```
const int INC = 5;
int max_len = 10, count = 0, m;
int *pda = (int *)malloc(max_len * sizeof(int));
        //int *pda = new int[max_len];

scanf("%d", &m);
//cin >> m;
while(m != -1)
{
    .....

    pda[count] = m;
    ++count;
    scanf("%d", &m); //cin >> m;
}

if(count >= max_len)
{
    max_len += INC; //扩容
    int *q = (int *)malloc(max_len * sizeof(int));
        //int *q = new int[max_len];
    for(int i = 0; i < count; ++i)
        q[i] = pda[i];
    free(pda); //delete []pda;
    pda = q;
    q = NULL;
}
```

指针及其运用

- 指针的基本概念
 - 概述
 - 指针类型的构造
 - 指针变量的定义与初始化
 - 指针的基本操作
- 用指针操纵数组
- 用指针在函数间传递数据
 - 指针类型参数
 - const的作用
 - 指针类型返回值
- 用指针访问动态变量
 - 通用指针与void类型
 - 动态变量的创建、访问和撤销
 - 内存泄漏与悬浮指针
- 多级指针
- 用指针操纵函数
- C++的引用类型

二级指针 与 数组的指针

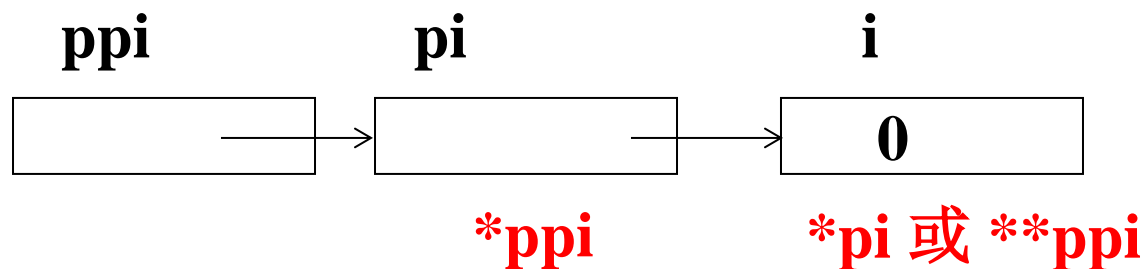
□ C语言的指针变量可以存储另一个指针变量的地址。

■ 比如,

```
int i = 0;
```

```
int *pi = &i;
```

```
int **ppi = &pi; // 指针变量ppi存储的是指针变量pi的地址
```



□ C语言的指针变量的基类型可以是int、float这样的基本类型，也可以是数组这样的构造类型。

```
int a[5];
```

```
int (*q)[5] = &a; // q相当于一个二级指针
```



```

int Sum(int x[ ][5], int row)
{
    int s = 0;
    for(int i = 0; i < row; ++i)
        for(int j = 0; j < 5; ++j)
            s += x[i][j];
    return s;
}

```

```

int Sum(int (*q)[5], int row)
{
    int s = 0;
    for(int i = 0; i < row; ++i)
        for(int j = 0; j < 5; ++j)
            s += q[i][j];
    return s;
}

```

```

int main( )
{
    int mtrx[10][5] = {.....};
    printf("%d", Sum(mtrx, 10));
    return 0;
}

```

&mtrx[0]

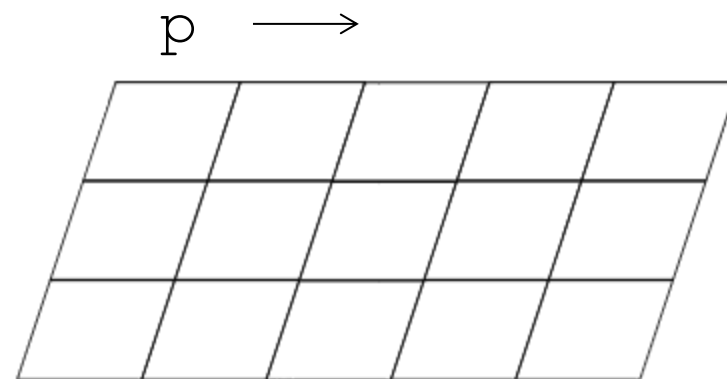


把二维数组看成特殊的一维数组

```
int Sum(int *p, int n)
{
    int s = 0;
    for(int i = 0; i < n; ++i)
        s += p[i];
    return s;
}
```

```
int main( )
{
    int mtrx[10][5] = {.....};
    printf("%d", Sum(&mtrx[0][0], 10*5));
    return 0;
}
```

mtrx[0]



二维数组的指针*

```
int b[3][5];
```

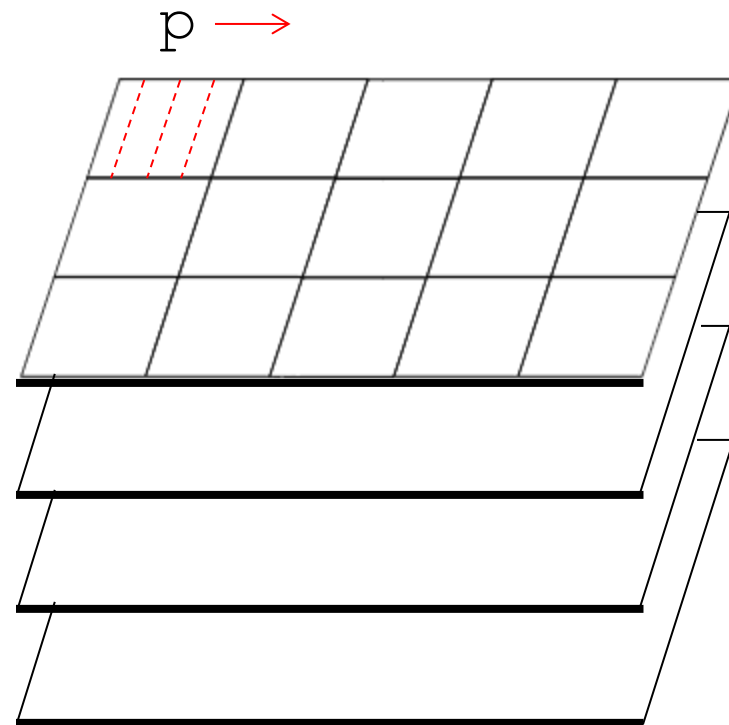
```
int *p;
```

```
p = &b[0][0];
```

*p 即 b[0][0]

p[0] 即 b[0][0]

某个元素 p[j]



二维数组的指针*

```
int b[3][5];
```

```
int *p;
```

```
p = &b[0][0]; //或 “p = b[0];”
```

*p 即 b[0][0]

p[0] 即 b[0][0]

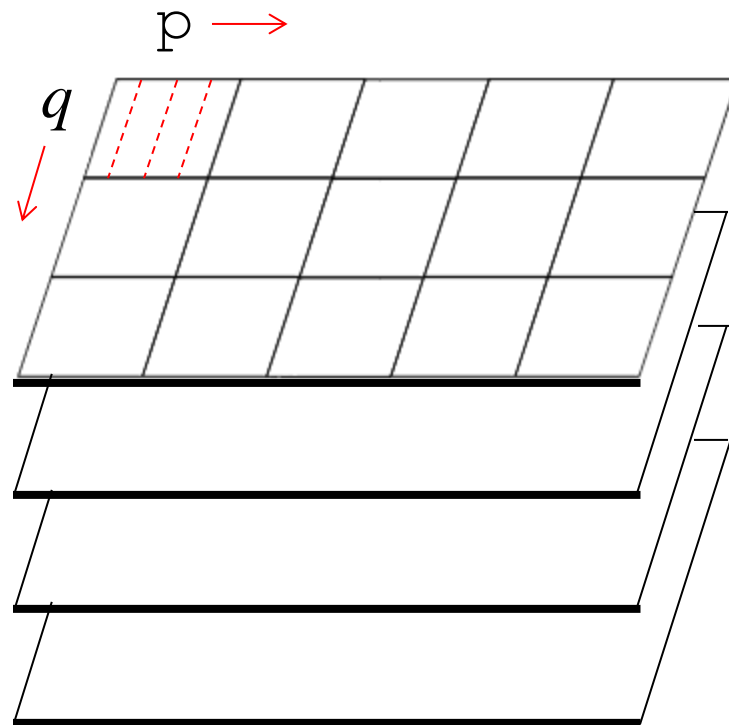
某个元素 p[j]

```
int (*q)[5];
```

```
q = &b[0];
```

*q 即 b[0], **q 即 b[0][0]

某个元素 q[i][j]



二维数组的指针*

```
int b[3][5];
```

```
int *p;
```

```
p = &b[0][0]; //或 “p = b[0];”
```

***p** 即 b[0][0]

p[0] 即 b[0][0]

某个元素 p[j]

```
int (*q)[5];
```

```
q = &b[0]; //或 “q = b;”
```

q** 即 b[0], *q** 即 b[0][0]

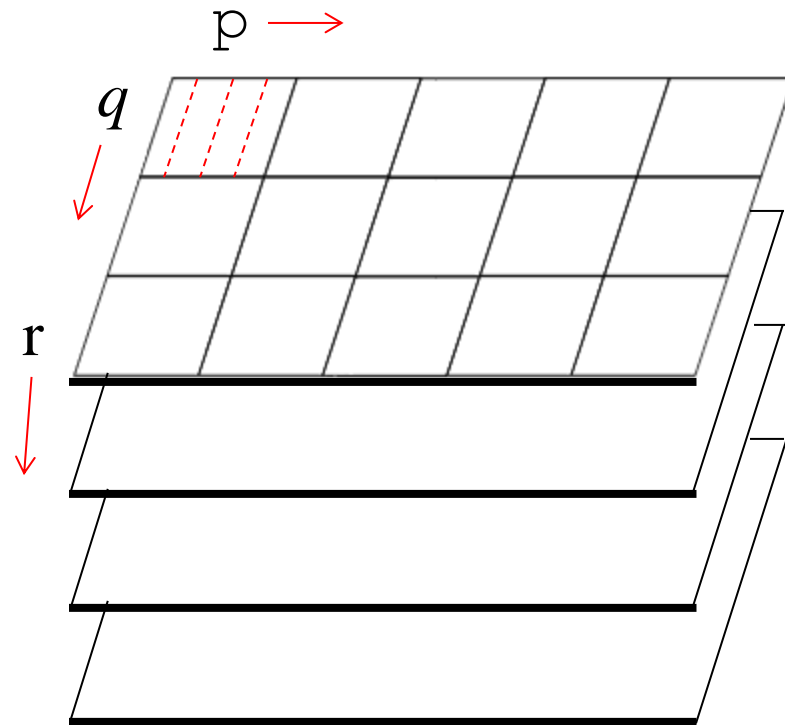
某个元素 q[i][j]

```
int (*r)[3][5];
```

```
r = &b;
```

r** 即 b, *r** 即 b[0], *****r** 即 r[0][0][0]

某个元素 r[i][j][k]



指针及其运用

- 指针的基本概念
 - 概述
 - 指针类型的构造
 - 指针变量的定义与初始化
 - 指针的基本操作
- 用指针操纵数组
- 用指针在函数间传递数据
 - 指针类型参数
 - const的作用
 - 指针类型返回值
- 用指针访问动态变量
 - 通用指针与void类型
 - 动态变量的创建、访问和撤销
 - 内存泄漏与悬浮指针
- 多级指针
- 用指针操纵函数
- C++的引用类型

函数指针：用指针操纵函数*

□ C程序运行期间，程序中每个函数的目标代码也占据一定的内存空间。C语言允许将该内存空间的首地址赋给函数指针类型变量（简称函数指针，注意与指针类型返回值的区别），然后通过函数指针来调用函数。

- 先构造一个函数指针类型
- 定义一个函数指针
或
- 在构造类型的同时定义函数指针
- 接着让函数指针 `pf` 指向内存的代码区
 - 用取地址操作符`&`获得函数的内存地址
 - 或
 - 直接用函数名获得函数的内存地址
- 然后通过函数指针调用函数`F`

```
typedef int (*PFUNC) (int);  
PFUNC pf;
```

或

```
int (*pf) (int);
```

```
pf = &F;
```

或

```
pf = F;
```

```
(*pf) (10);
```

或

```
pf(10); //实参为10
```

```
int F(int m)  
{  
    ...  
}
```

函数指针

□ 可以用来调用函数

```
double (*pfu)(double) = MyFoo;
```

```
pfu(10.7); //实参为10.7
```

```
double (*pfu)(double) = &MyFoo;
```

```
(*pfu)(10.7); //实参为10.7
```

```
double MyFoo(double x)
{
    double f = 2 * x - 7;
    return f;
}
```

```
typedef double (*PFU)(double);
```

```
PFU pfu;
```

```
PFU func_list[5] = {sin, cos, tan, log, log10};
```


□ 例6.6 积分函数Integrate (可计算任意一个一元可积函数在一个区间 $[x1, x2]$ 上的定积分)

■ 该函数的调用形式:

```
Integrate(sin, 0, 3.14);
```

//计算函数sin在区间 $[0, 3.14]$ 上的定积分

```
Integrate(cos, 1, 2);
```

//计算函数cos在区间 $[1, 2]$ 上的定积分

```
Integrate(MyFoo, 1, 10);
```

//计算函数MyFoo在区间 $[1, 10]$ 上的定积分

■ 该函数的原型:

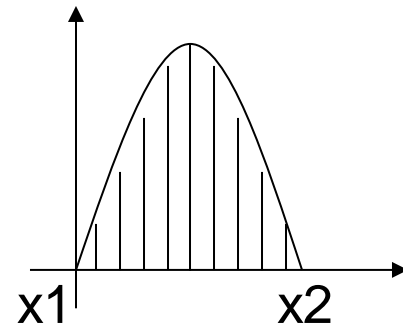
```
double Integrate(double (*pfu)(double x), double x1, double x2)
```

用函数指针pfu操纵某个一元可积函数

`Integrate(sin, 0, 1);`

```
double Integrate(double (*pfu)(double x), double x1, double x2)
{
    double s = 0;
    int n = 10;
    printf("please input the precision: ");
    scanf("%d", &n);

    for(int i=1; i <= n; ++i) // i为步长, n为等份的个数, n越大, 计算结果精度越高
        s += sin(x1 + (x2 - x1) / n * i); // 窄条的高度(即矩形的上底边≈下底边)
    s *= (x2 - x1) / n; // 乘以窄条的宽度 (即矩形的高) 得到面积
    return s;
}
```



回调函数 (Callback Functions)

```
#define N 4

void BubbleSort(int sdata[ ], int count, bool (*)(int, int));

bool Less(int m, int n){ return m < n;} //回调函数
bool More(int m, int n){ return m > n;} //回调函数

int main( )
{
    int a[N];
    for(int i = 0; i < N; ++i)
        scanf("%d", &a[i]);
    BubbleSort(a, N, Less);
        //按升序排
    for(int i = 0; i < N; ++i)
        printf("%d \t", a[i]);
    return 0;
}
```

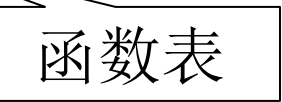
```
void BubbleSort(int sdata[ ], int count,
                bool (*Ordrr)(int, int))
{
    for(int i = 0; i < count-1; ++i)
        for(int j = 0; j < count-1-i; ++j)
            if(Ordrr(sdata[j+1], sdata[j]))
                // if(sdata[j+1] < sdata[j])
                {
                    int temp = sdata[j];
                    sdata[j] = sdata[j+1];
                    sdata[j+1] = temp;
                }
}
```

函数指针数组

例6.7 根据输入的要求，执行在函数表中定义的某个函数。

```
...
#include <cmath>
typedef double (*PF) (double) ;
PF func_list[8] = {sin, cos, tan, asin, acos, atan, log, log10};
int main( )
{
    int index;
    double x;
    printf("请输入序号(0:sin, 1:cos, 2:tan, ..., 6:log, 7:log10):\n");

    scanf("%d", &index);
    printf("请输入参数: ");
    scanf("%lf", &x);
    printf("结果为: %f \n", (*func_list[index])(x));
    return 0;
}
```



指针及其运用

- 指针的基本概念
 - 概述
 - 指针类型的构造
 - 指针变量的定义与初始化
 - 指针的基本操作
- 用指针操纵数组
- 用指针在函数间传递数据
 - 指针类型参数
 - const的作用
 - 指针类型返回值
- 用指针访问动态变量
 - 通用指针与void类型
 - 动态变量的创建、访问和撤销
 - 内存泄漏与悬浮指针
- 多级指针
- 用指针操纵函数
- C++的引用类型

C++的引用类型

□ 什么是引用

注意：Typedef 是给类型名取一个别名

- 引用类型用于给一个变量取一个别名。

➤ 例如：

```
int x = 0;
```

```
int &y = x; //y为引用类型的变量
```

```
cout << x << ' , ' << y << endl; //结果为：0,0
```

```
y = 1;
```

```
cout << x << ' , ' << y << endl; //结果为：1,1
```

```
x = 2;
```

```
cout << x << ' , ' << y << endl; //结果为：2,2
```

在语法上，对引用类型变量的访问与非引用类型相同；
但在语义上，对引用类型变量的访问实际访问的是另一个变量（被引用的变量），
通常用来定义形式参数。

引用类型的变量定义与初始化

□ &

```
int x;
```

```
int &y = x;
```

□ 注意：

- 定义引用变量时**必须要有初始化**，并且引用变量的基类型和被引用变量的类型相同
- 引用类型的变量定义之后，它不能再引用其他变量。（可以编译执行，但含义不是引用其他变量）

引用与指针的对比

```
int a=1, c=3;
int &b = a;    //b引用a
cout << a << ', ' << b << ', ' << c << endl;
b = c;      //不是b引用c, 而是将c的值赋给b
cout << a << ', ' << b << ', ' << c << endl;
```

1, 1, 3

b a
c

3
3

3, 3, 3

```
a=1, c=3;
int *p = &a;    //p指向a
cout << a << ', ' << *p << ', ' << c << endl;
p = &c;      //p指向c
cout << a << ', ' << *p << ', ' << c << endl;
```

1, 1, 3

p a
p c

1
3

1, 3, 3

理解引用

```
int a=1, c=3;
```

```
int * const p = &a; // *p 即 a 的引用
```

```
cout << a << ', ' << *p << ', ' << c << endl;
```

```
*p = c;
```

```
cout << a << ', ' << *p << ', ' << c << endl;
```

1, 1, 3

3, 3, 3

p a	3
c	3

```
int a=1, c=3;
```

```
int &b = a;
```

```
cout << a << ', ' << b << ', ' << c << endl;
```

```
b = c;
```

```
cout << a << ', ' << b << ', ' << c << endl;
```

1, 1, 3

3, 3, 3

b a	3
c	3

引用类型参数

C++ 有这样的库函数 swap!

□ 引用传递：把实参的地址传给相应的形式参数

```
void Swap(int &x, int &y)
{
    int t = x;
    x = y;
    y = t;
}
```

```
int main()
{
    int a=5, b=9;
    cout << a << ' , ' << b << endl; //结果为: 5, 9
    Swap(a, b);
    cout << a << ' , ' << b << endl; //结果为: 9, 5
    return 0;
}
```

```
void MySwap(int *pm, int *pn)
{
    int temp = *pm;
    *pm = *pn;
    *pn = temp;
}
```

```
MySwap(&m, &n);
```

常量引用-防止函数副作用问题

```
void F(const int &x)
{
    .....
    x = 1;    //Error
    .....
}
```

```
int main()
{
    int a;
    F(a);
    return 0;
}
```

引用与指针的异同点

- 引用类型与指针类型都可以实现**通过一个变量访问另一个变量**，但**访问的语法形式**不同：引用采用变量名直接访问，指针则采用取值操作间接访问。引用类型的访问过程对使用者而言是**透明**的，因此更安全。
- 在作为函数参数类型时，引用类型参数的实参是一个变量的名字，而指针类型参数的实参是一个变量的地址。引用更**方便**。除了在定义时指定的被引用变量外，引用类型变量不能再引用其他变量；而指针变量定义后可以指向其他同类型的变量。因此，引用类型比指针类型要**安全**。

函数间有多种通讯方式

- 传值方式(把实参的副本复制给形参)
- 利用函数返回值传递数据
- 通过全局变量传递数据
(函数副作用问题，尽量不用)
- 通过指针类型参数传递数据
(函数副作用问题，可以通过指向常量的指针来避免)
- 通过引用类型参数传递数据 (C++)
(函数副作用问题，可以通过指向常量的引用来避免)

小结

- 指针：
 - 一种构造数据类型，
 - 地址是一种特殊的整数，将地址专门用指针类型来描述，可以限制该类型数据的操作集，从而得以保护数据。
 - 引用是基于指针封装的一种构造数据类型。

小结

□ 要求:

- 掌握指针/引用的基本概念及定义、初始化和操作方法
- 掌握指针/引用类型的特征及其典型运用场合之一
 - 可以在函数间高效地传递数据(如数组, 使用过程中要注意避免数组越界等问题)
 - ✓ 函数副作用如何利用或避免
- 掌握指针的典型用法之二
 - 指针可以用来操作动态数据
 - ✓ 动态变量和动态数组需要在程序中创建与撤销, 使用过程中要注意避免内存泄露和悬浮指针等问题。
- 一个程序代码量 ≈ 40 行
- 继续保持良好的编程习惯