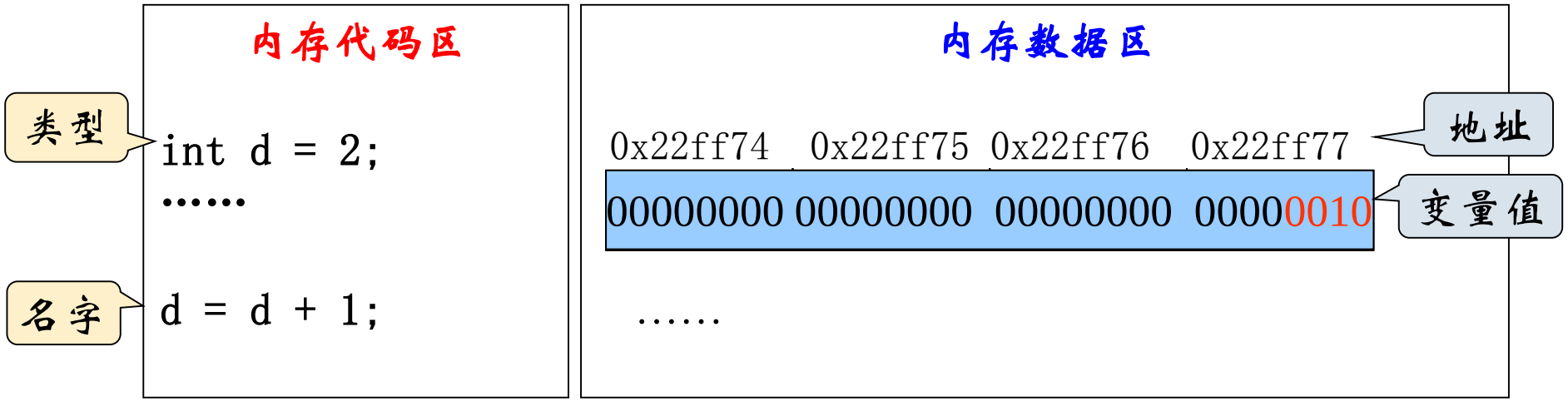
内容回顾：变量的属性

- 程序执行到变量定义处，系统会为变量分配一定大小的空间，用以存储变量的值。
  - 存储空间：存放数据的二进制表示；由地址来标识，一般由系统自动管理。

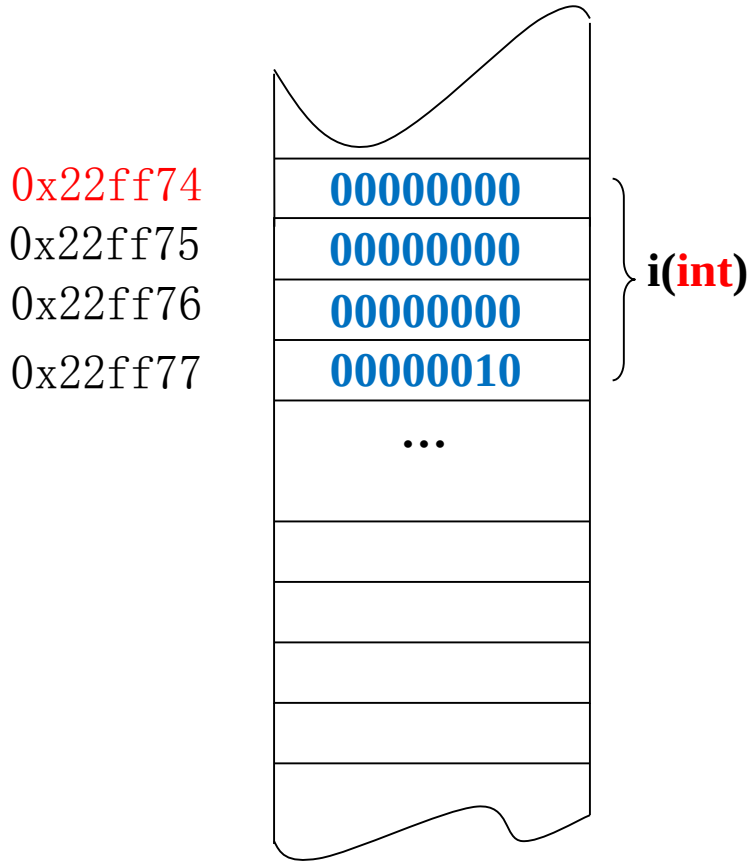


用户定义变量类型和名字

系统决定地址，存储其二进制值

地 址

- 特殊的整数，通常用十六进制整数表示
  - 计算机的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
- 通过首地址和存储单元里数据的数据类型，可定位内存中的数据
  - 首地址决定访问的起点，数据类型决定数据占用的内存单元数



访问数据的方式

- 通过变量名：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使用变量名访问数据。
- 通过变量地址：如果在程序中已知变量的地址，则可以使用这个地址访问数据，省略系统的映射环节，从而提高数据的访问效率。

如何通过地址访问变量呢？

6 复杂数据的描述—构造数据类型
6.2 指针及其应用

郭延文

2019级计算机科学与技术系

指针及其应用

- 指针的基本概念
  - 指针类型的构造
  - 指针变量的定义与初始化
  - 指针数组
  - 多级指针变量
  - 通用指针与void类型
  - 指针类型相关的基本操作
- 用指针操纵数组
- 用指针在函数间传递数据
- 用指针访问动态变量
- 用指针操纵函数**

访问数据的方式

- 通过变量名：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使用变量名访问数据。
- 通过变量地址：如果在程序中已知变量的地址，则可以使用这个地址访问数据，省略系统的映射环节，从而提高数据的访问效率。

如何描述地址？

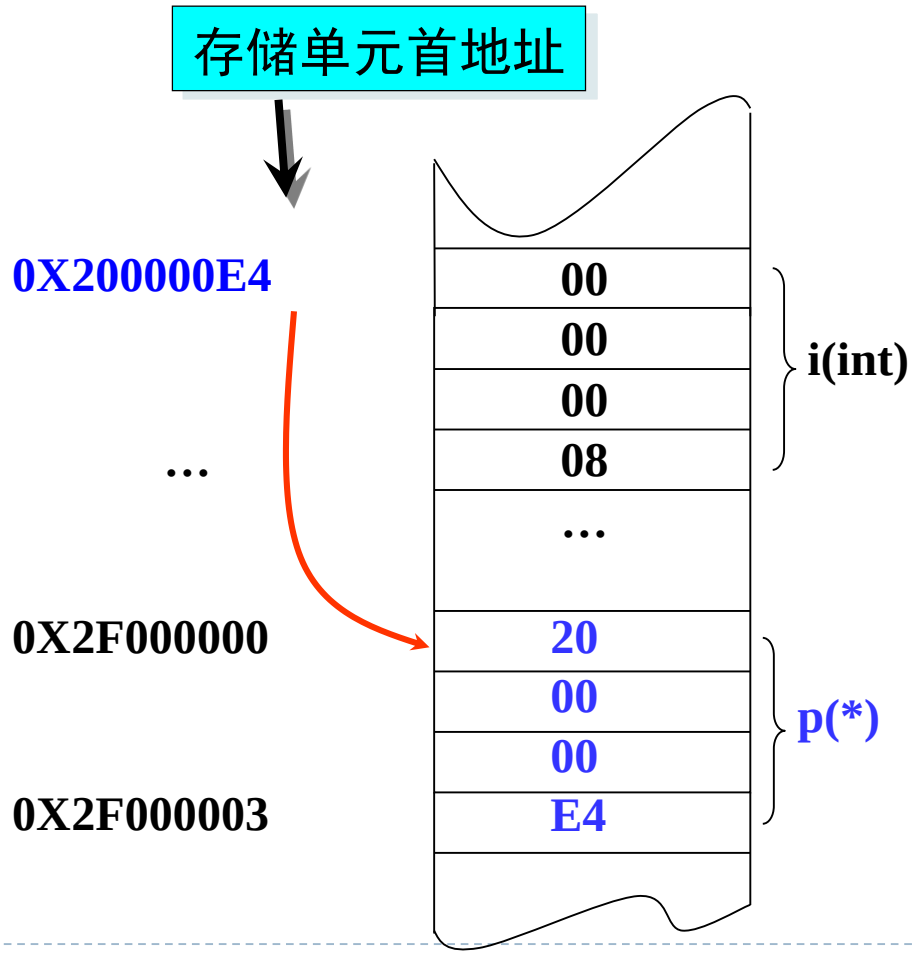
- C语言用指针类型描述地址，通过指针类型数据的相关操作可以实现与地址有关的程序功能。
- 即允许程序使用一个指针类型变量（简称指针变量或指针，pointer）表示一个变量的地址。

什么是指针(pointer)

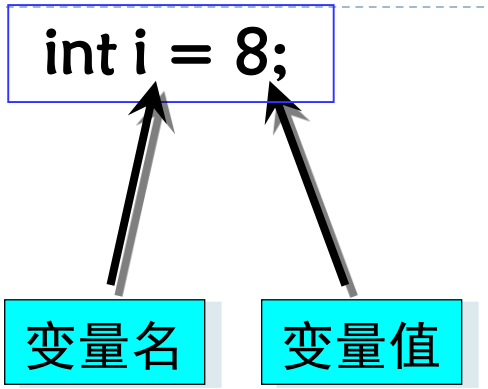
int i = 8;

变量名　　变量值

```
int  *p;
p = &i;
// p就是一个指针，指向i
```



什么是指针(pointer)?




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

// p 取值： 南京市栖霞区仙林大道163号

什么是指针(pointer)？

指针类型的构造

- 指针类型由一个变量类型（在这里又叫基类型）关键词和一个*构造而成：
- 比如，

```
typedef int * Pointer;
// Pointer是由int型变量的地址所构成的指针类型标识符
//其值集为int型数据的地址
```

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 p1, p2;  //定义了两个同类型的指针变量p1、p2
```

- 可以在构造指针类型的同时直接定义指针变量：

```
int *p;
// int和*构造了一个指针类型，
// 同时定义了一个指针变量p
```

指针变量初始化

- 指针变量所存储的变量的内存单元地址，可以通过初始化来指定：

```
int i = 0;
int *pi = &i;
//取出变量i的地址，用来初始化指针变量pi，即pi指向i
```

- 用来初始化指针变量的变量要预先定义，且类型与指针基类型要一致，初始化后称指针指向该变量，未初始化的指针变量不指向某个变量：

```
int i = 0;
float f = 3.2;
float *pf = &i;        //应改为float *pf = &f;
```

指针变量初始化

- 也可以用另一个指针变量或指针常量来初始化一个指针变量

```
int i = 0;
int *pi = &i;
int *pj = pi;  //pj和pi值一样，即pj也指向变量i
```

  - 又比如，

```
int a[10];
int *pa = a;   //pa指向数组a (a即&a[0])
```

- 还可以用NULL来初始化一个指针变量，表示该指针变量暂不指向任何变量：

```
int *pv = NULL;//NULL是一个空地址；或int *pv = 0;
```

- 不可以将一个非0整数赋给一个指针变量，因为一个事先确定的地址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：

```
int *pn = (int*)2000;
//编译器不一定会报错，但执行时可能会引起系统故障
```

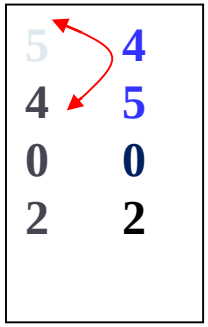
指针类型相关的基本操作

- 取地址操作
- 取值操作
- 加/减一个整数
- 关系/逻辑操作
- 赋值操作
- 下标操作

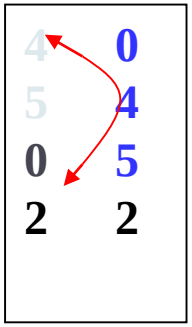
- 这些操作只有在一定约束条件下才有意义。不能参与的操作是没有意义的操作，不一定会出语法错误，但可能会造成危险的结果
  - 比如：将两个指针类型数据相加或乘/除法运算，结果不一定是一个有效的内存单元地址，即使是一个有效地址，也未必是该程序可访问的内存空间的地址

直接插入法排序

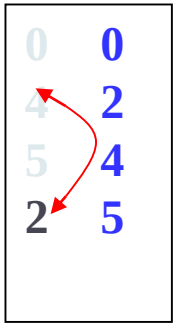
- 在已排好序的i个数中寻找合适的位置，将下标为i的元素插入，插入点后的数往后移
- N个数考察N-1趟，每趟内比较的次数随趟数递增，一旦发现需要进行插入操作，则需要进行部分元素的搬移操作。



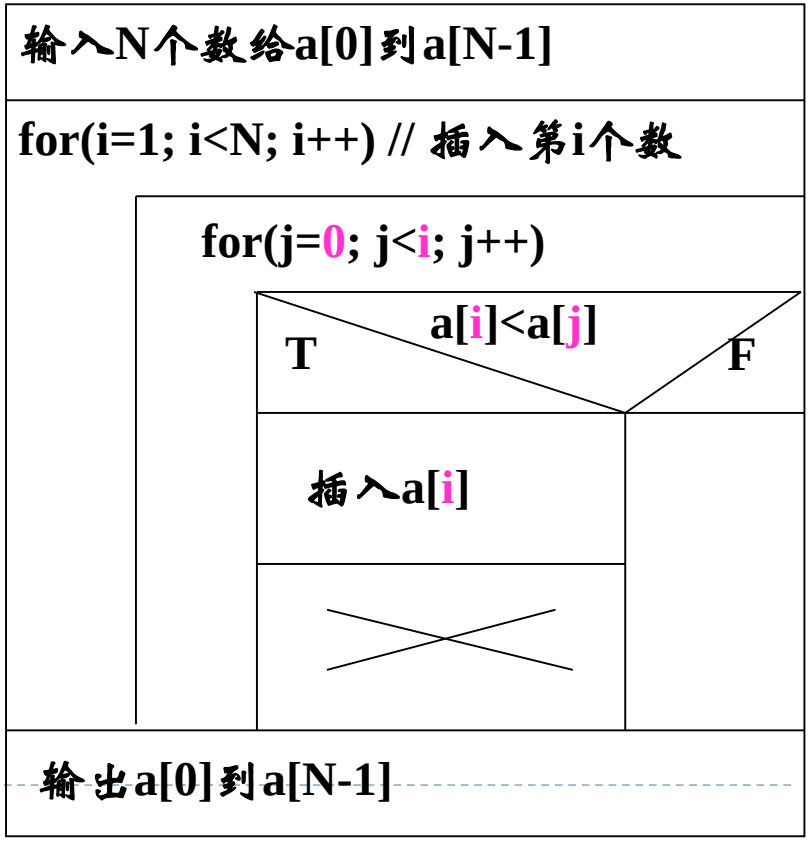
插入4



插入0



插入2



内容回顾

```
void InsSort(int ndata[ ], int N)
{
        int temp;
        for (int i = 1; i < N; i++)
              for (int j = 0; j < i; j++)
                    if(ndata[i] < ndata[j])
                    {
                          temp = ndata[i];
                          for (int k = i-1; k >= j; k--)    // 先挪动从j开始的元素
                                ndata[k+1] = ndata[k];
                          ndata[j] = temp; // 插入原来的sdata[i]到相应位置
                          break;
                    }
}
```

快速法排序*

1. 是对起泡法的一种改进
2. 基本思想：(1)通过一趟排序将待排数据分隔成独立的两部分，其中一部分数据均比另一部分数据小，然后(2)对这两部分数据分别进行快速排序。整个排序过程可以递归进行，以达到整个序列有序
3. 即，设两个下标变量first和last，它们的初始值分别为0和N-1，然后寻找分割点split_point，再将分割点±1分别作为右半部分的first和左半部分的last，对两部分重复上述步骤，每部分直至first==last为止。

内容回顾

```
int Split(int sdata[ ], int first, int last);

void QuickSort(int sdata[ ], int first, int last)   // 采用递归！
{
        if(first < last)
        {
                int split_point;
                split_point = Split(sdata, first, last);
                                //寻找分割点
                QuickSort(sdata, first, split_point-1);
                                //对分割点以左的部分进行排序
                QuickSort(sdata, split_point+1, last);
                                //对分割点以右的部分进行排序
        }
}
```

内容回顾

```
int Split(int sdata[ ], int first, int last)
{
        int split_point, pivot;
        pivot = sdata[first];
        split_point = first;
        for (int unknown = first+1; unknown <= last; unknown++)
            if(sdata[unknown] < pivot)
            {
                split_point++;
                int t = sdata[split_point];
                sdata[split_point] = sdata[unknown];
                sdata[unknown] = t;
             }
        sdata[first] = sdata[split_point];
        sdata[split_point] = pivot;
        return split_point;
}
```

排序算法总结

内容回顾

要求大家至少熟练掌握一种排序方法！

排序算法	冒泡排序	选择排序	直接插入排序	快速法排
特点	比较相邻两个数，小的调到前头， 像气泡不断的往上冒...	从i个数中选择最大者，与第i个数交换位置 ...	在已排好序的i个数中寻找合适的位置，将下标为i的元素插入，插入点后的数往后移	通过一趟排序将待排数据分隔成独立的两部分，其中一部分数据均比另一部分数据小，然后对这两部分数据分别进行快速排序
算法结构	双层循环 for(i=0;i<N-1;i++)	双层循环 (for(i=N;i>1;i++))	三层循环 for(i=1; i<N; i++)	递归＋循环

什么是指针(pointer)

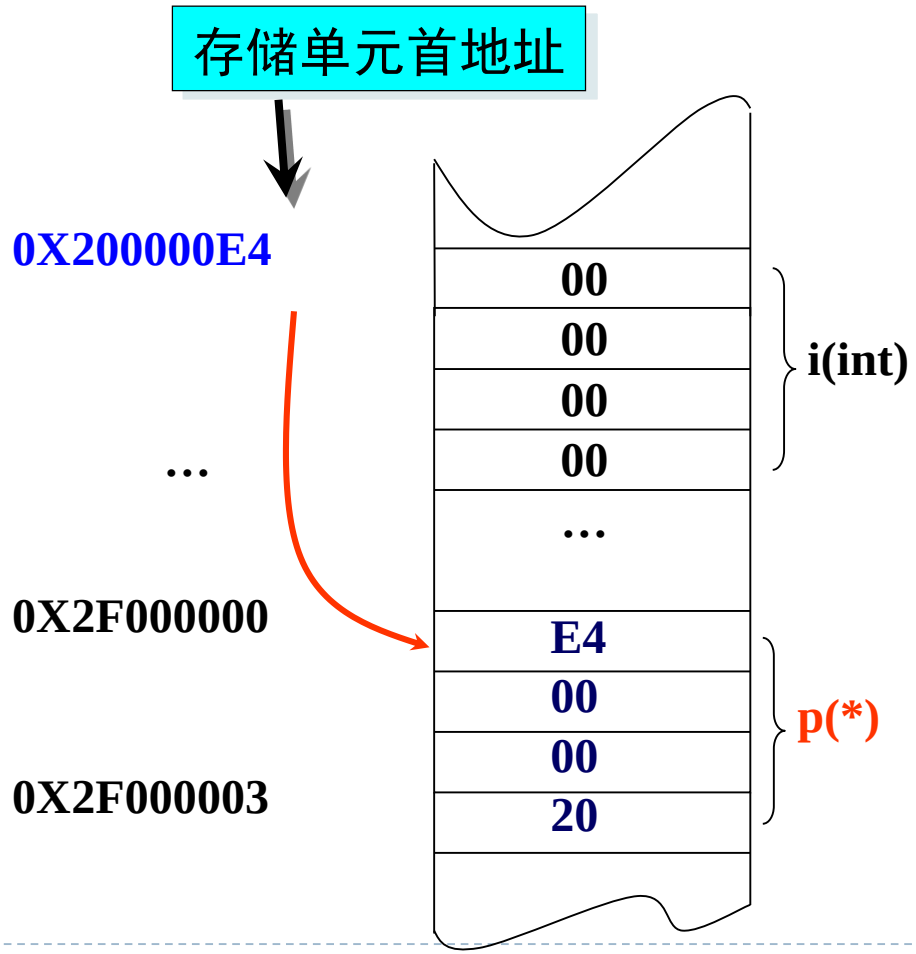
int i = 8;

变量名　　变量值

int *p;

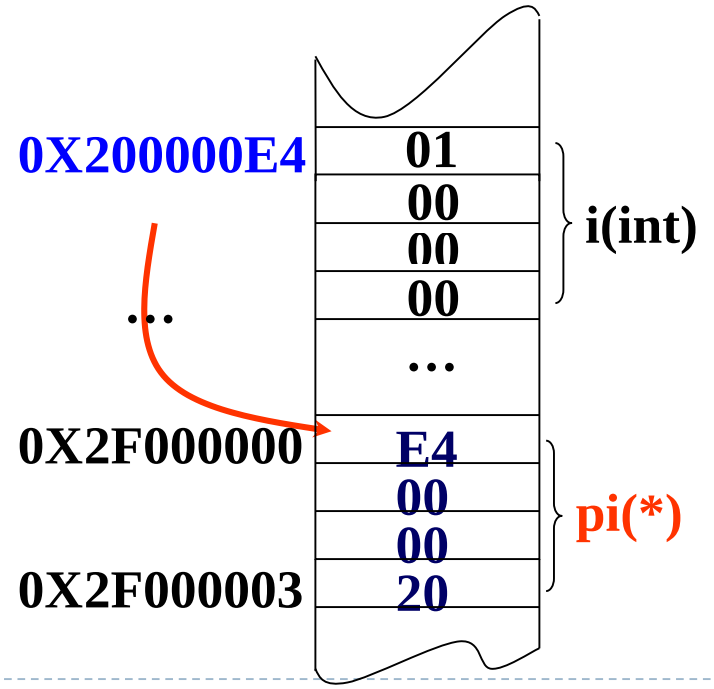
p = &i;

// p就是一个指针，指向i



举 例

```
int *pi=NULL;    //这里的 * 不是指针操作符
int i = 0;
pi = &i;         //pi指向i
*pi = 1;     //对pi进行取值操作并对其赋值，相当于i = 1;
*(&i) = 2;       //相当于i = 2;
pi = &(*pi);   //pi仍然指向i
```



&:取地址操作 – 关于指针的两个操作之一

- 用地址操作符&获取操作数的地址； & 是一个单目操作符，优先级较高（2级），结合性为自右向左：

```
int i = 0;
int *pi;
pi = &i;
```

- 其操作数应为各种类型的变量、数组元素等，不能是表达式、字面常量、寄存器变量。

*:取值操作 – 关于指针的两个操作之二

- 用指针操作符*获得指针变量指向的内存数据

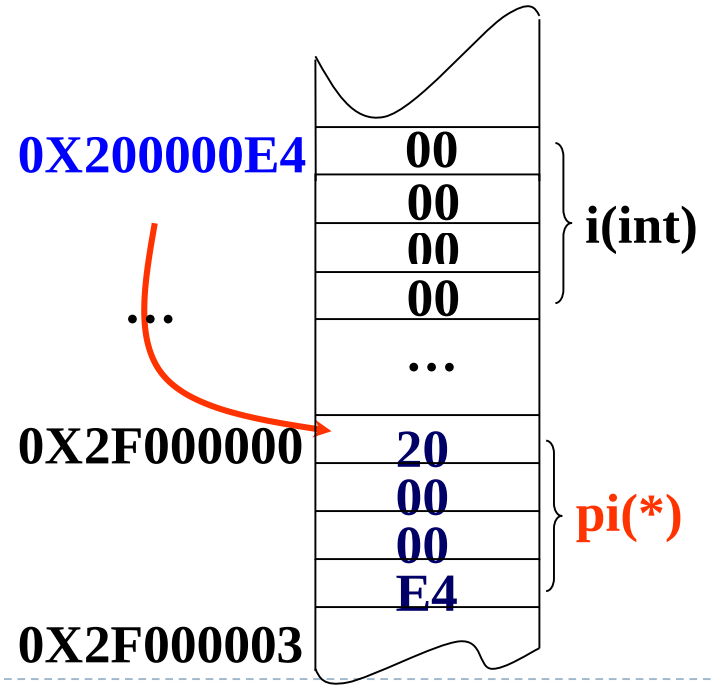
```
int i = 0,   *pi=NULL;
pi = &i;     // *pi 等于i
```

- *是一个单目操作符，优先级较高（2级），结合性为自右向左
- 其操作数应为指针变量或地址类型的数据。
- 注意：构造指针类型或定义指针变量时的“*”（int *p;） 不是指针操作符

- 取值操作*与取地址操作&是一对逆操作

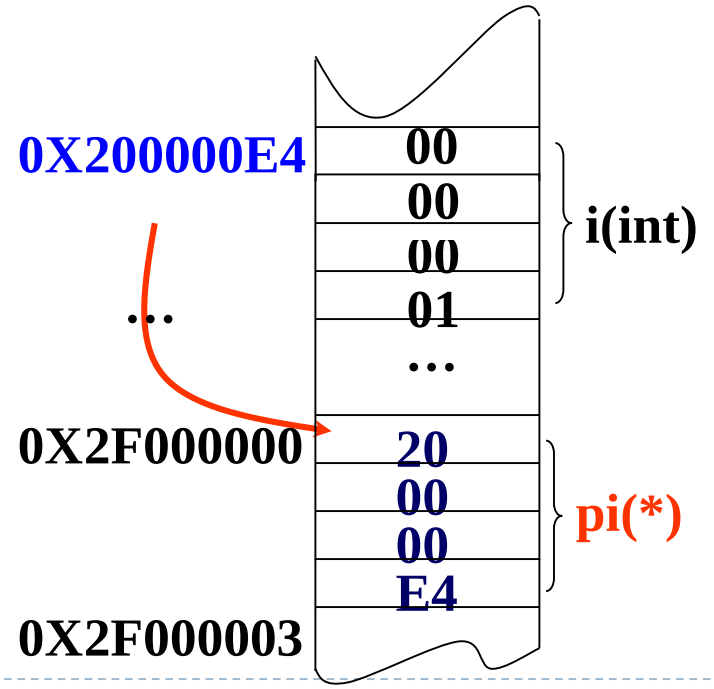
举 例 (&, *)

```
int *pi=NULL;  //这里的 * 不是指针操作符
 int i = 0;
pi = &i;        //pi指向i
```



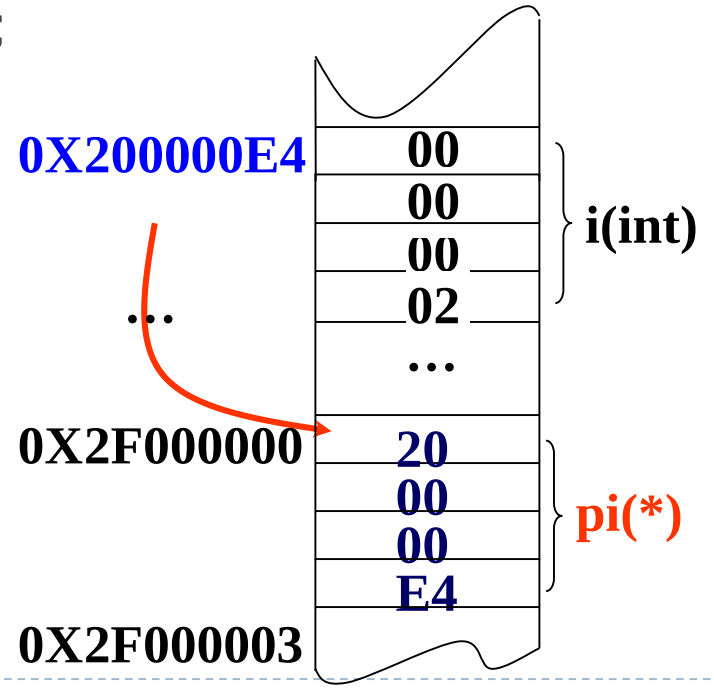
举 例 (&, *)

```
int *pi=NULL;  //这里的 * 不是指针操作符
 int i = 0;
pi = &i;          //pi指向i
*pi = 1;   //对pi进行取值操作并对其赋值，相当于i = 1;
```



举 例 (&, *)

```
int *pi=NULL;  //这里的 * 不是指针操作符
int i = 0;
pi = &i;          //pi指向i
*pi = 1;   //对pi进行取值操作并对其赋值，相当于i = 1;
*(&i) = 2;     //相当于i = 2;
pi = &(*pi);  //pi仍然指向i
```



指针类型数据的赋值操作

- 指针变量除初始化外，还可以通过赋值操作指定一个地址，使之指向某一个变量

```
int i = 0;
int *pi = &i;  //初始化，pi指向i
……
int x = 3;
pi = &x;       //赋值，pi指向x
*pi = 2;       //改变x的值
```

- 指针变量赋值操作的右边操作数也应为地址类型的数据(还可以为0)。右操作数中的变量要预先定义，其类型要与左边指针变量的基类型一致，如果是指针变量，其基类型要与左边指针变量的基类型一致(必要时可使用强制类型转换)。

- 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没有初始化就使用，警告

```
int i;
int b[3];
int *pi,*pb,*pj,*pv;
pi = &i;
pb = b;
pj = pi;
pv = 0;
```

将首地址赋给指针变量

思 考

- 判断赋值运算的对错：

```
int i, *p, *p1;
double x, *q;
.....
p = &i;          // p指向i
q = &x;          // q指向x
p = &x;          // ×类型不一致
q = &i;          // ×类型不一致
p1 = p;          // p1指向p所指向的变量
p1 = q;          // ×类型不一致
p = 0;           // 使得p不指向任何变量
p = 2000;        // ×2000为int型
p = (int *)2000; // 不建议使用
```

指针类型数据的关系/逻辑操作

- 两个指针类型的数据可以进行比较，以判断在内存的位置前后关系，前提是它们表示同一组数据的地址
- 比如，
  - 两个指针变量都指向某数组的元素，用关系操作比较这两个地址在数组中的前后位置关系
  - 也可以判断一个地址是否为0，以明确该地址是否为某个实际内存单元的地址（一般指针初值赋0）

指针类型数据的加/减一个整数操作

- 一个指针类型的数据加/减一个整数，可以使其成为另一个变量的地址，前提是操作结果仍然是一个有效的内存单元地址。
  - 例如：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（常用于数组元素的操作，不能超出数组范围）
  - 注意，加/减一个整数i后的结果并非在原来地址值的基础上加/减i，而是i的倍数，具体倍数由基类型决定，即加/减i*sizeof(基类型)：

```
int i = 0;
int *pi = &i;
  pi++;          // 大家上机实验!
//设int型数据占4个字节空间，则pi的地址值实际增加了4，而不是1
```

两个相同类型的指针类型数据的相减操作

- 两个相同类型的指针类型数据的相减操作，结果为两个地址之间可存储基类型数据的个数。通常用来计算某数组两个元素之间的偏移量。
  - 比如，

```
int *pi, *pj;
……
int offset = pj - pi;     // 大家上机实验!
//offset为pj与pi之间可存储int型数据的个数
```

指针的输出

```
int i = 1;

int *p = &i;

printf(“%x”, p);        //输出p的值(i的地址，16进制)

printf("%d", *p);       //输出i的值
```

指针数组

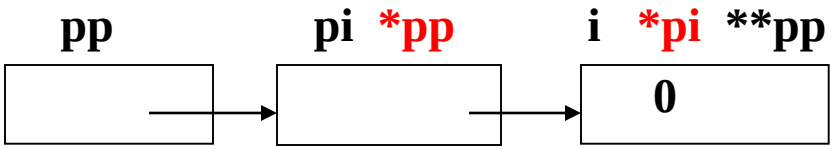
- 如果数组的每一个元素都是一个指针类型的数据，则该数组叫做指针数组。
  - 比如，

```
int i = 0, j = 1, k = 2;
int *ap[3] = {&i, &j, &k};
//ap这个指针数组的长度是3，各个元素的类型是int *
```

- 指针数组一般用于多个字符串的处理

多级指针变量

- C语言的指针变量还可以存储另一个指针变量的地址。
  - 比如，




```
int i = 0;
int *pi = &i;
int **pp = &pi;  //指针变量pp存储的是指针变量pi的地址
```

- 这时，称指针变量pp是一个二级指针变量(指向指针的指针)，它指向的变量是一个一级指针变量pi。

- 如果再定义一个指针变量ppp存储pp的地址，则ppp是一个三级指针变量。
- 多级指针变量通常用于指针类型数据的传址调用，或者用于多维数组和多个字符串的处理。

void类型

- void：空类型的关键词
- 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

void类型：用作通用指针类型**

- 还可以作为指针类型的基类型，形成通用指针类型（void *）
- 通用指针类型变量不指向具体的数据，不能用来访问数据，但是可以与其他任何指针类型数据进行相互赋值和比较操作，并且不需要显式的强制类型转换
- 通用指针类型经常作为函数的形参和返回值的类型，以提高函数的通用性

指针的应用

- 用指针操纵数组
- 用指针在函数间传递数据
  - 传址调用
  - const
  - 指针型函数**　　// **：目前了解即可
- 用指针访问动态变量
  - 动态变量的创建、访问与撤销
  - 内存泄露与悬浮指针**
- 用指针操纵函数**

用指针操纵数组-1

- 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指针变量，此时称该指针变量指向这个数组的元素：

```
int a[10];
int *pa = a;        //相当于int *pa = &a[0];
```

- 基本原理：当一个指针变量指向一个数组元素时，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2000，十六进制)，然后，pa的值可以变化为2004，2008，200C，2010，2014，2018，201C，2020，2024，于是可以通过pa来操纵数组a的各个元素。

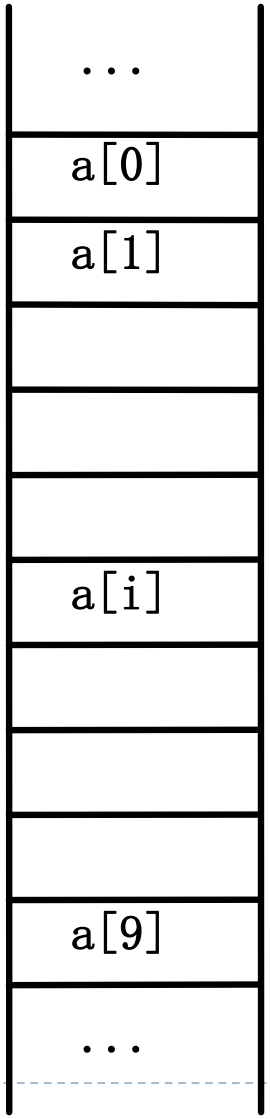
- 操纵方法有三种：
  - 下标法：通过下标操作指定元素；
  - 指针移动法：通过加/减一个整数操作指定元素地址；
  - 偏移量法：通过取值操作和加/减一个整数操作指定元素。
- 通过指针变量操纵数组时，要注意防止下标越界。

例：用指针变量操纵数组

```
#include <stdio.h>
#define N 10

int main( )
{   int i;
    int a[N] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int *pa;
    for(pa = a, i = 0; i < N; i++)
            printf("%d ", pa[i]);          //下标法，用a[i]也行
    for( ; pa < (a + N); pa++)             //指针移动法，用a++不行，因为a是常量
            printf("%d ", *pa);
    for(pa = a, i = 0; i < N; i++)    //这里没有“pa = a, i = 0”，则会出现越界!
            printf("%d ", *(pa + i));     //偏移量法，用*(a + i)也行
    return 0;
}
```

下标指示　　偏移量



pa[0]　　*pa, *a

pa[1]　　*(pa+1), *(a+1)

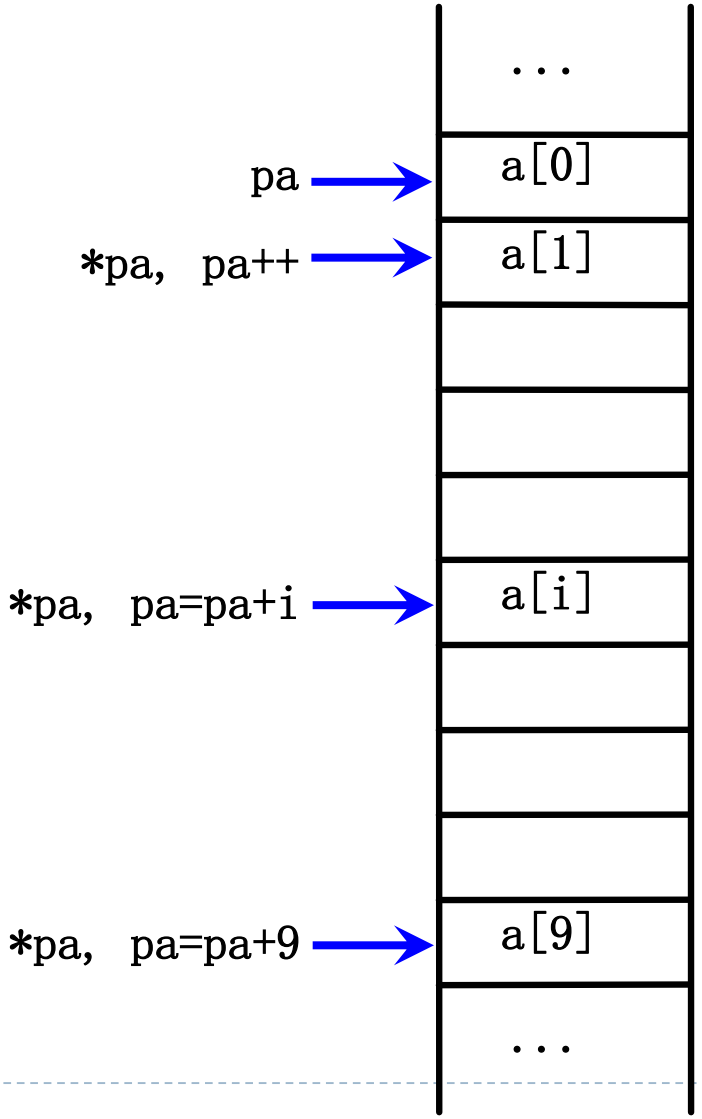
pa[i]　　*(pa+i), *(a+i)

pa[9]　　*(pa+9), *(a+9)

指针本身的值

并未改变！

指针移动



指针本身的

值改变了！

```
…
int *pa;
for(pa = a, i = 0; i < N; i++)
        printf("%d ", pa[i]);
for( ; pa < (a + N); pa++)
        printf("%d ", *pa);
for(pa = a, i = 0; i < N; i++)
        printf("%d ", *(pa + i));
…
```

- 指向数组元素的指针变量 vs. 数组名
  - 数组名是常量
    - 所以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，这个变量的初始值与数组名表示的地址相同
    - 经过操作，其值可能会是其他元素的地址

思 考

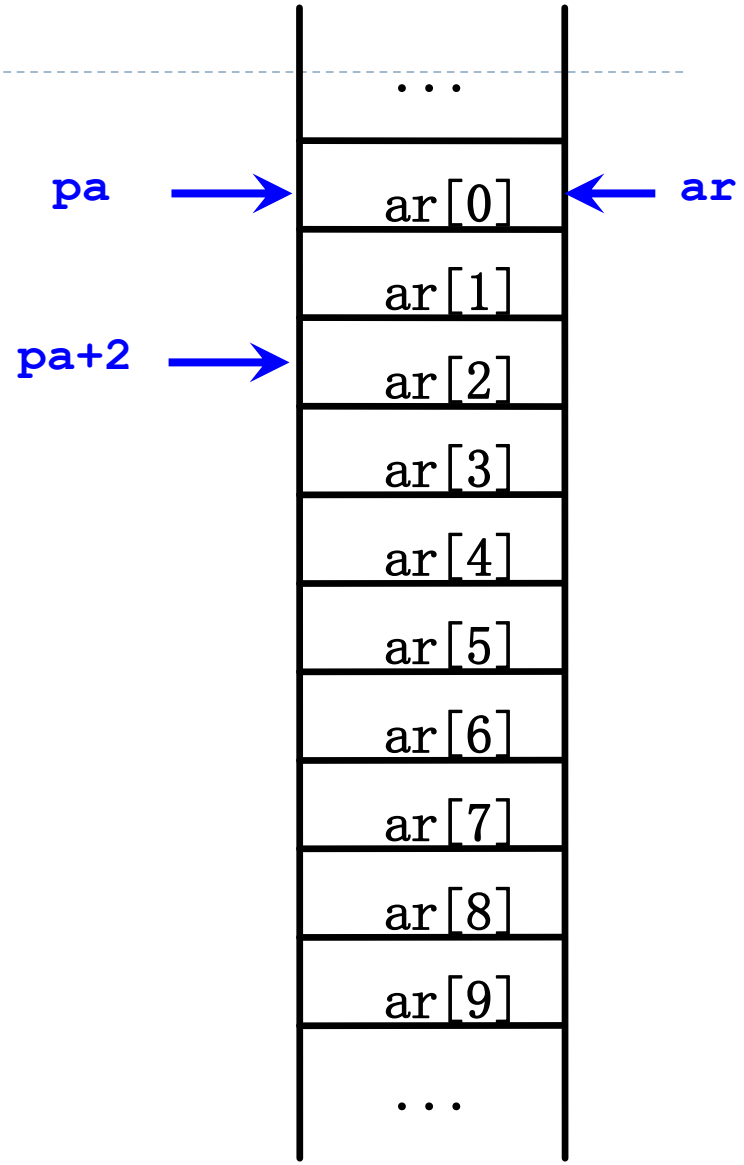
pa　　0X20000000

pa+2　　0X20000002 ?

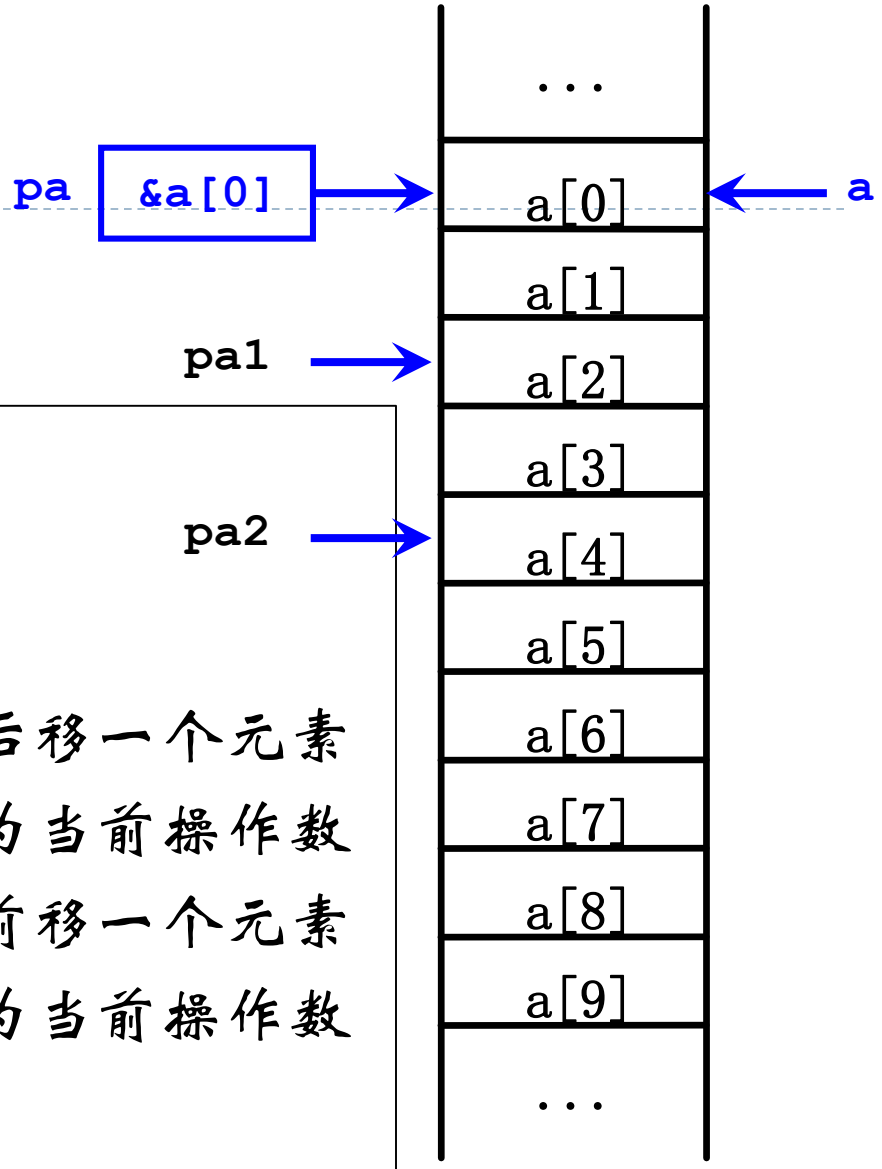
0X20000002 (char ar[10])
　　//+ sizeof(char)*2

0X20000008 (int ar[10])
　　//+ sizeof(int)*2

+ sizeof()*2



搞清以下相关操作！



pa+n　　pa之后第n个元素的地址

pa-n　　pa之前第n个元素的地址

pa++　　pa作为当前操作数，再后移一个元素

++pa　　pa后移一个元素，再作为当前操作数

pa--　　pa作为当前操作数，再前移一个元素

--pa　　pa前移一个元素，再作为当前操作数

pa2-pa1

结果为2，表示pa2和pa1之间相隔2个元素

```
if (a--)
{
    int  i = a ++;
}
```

搞清以下相关操作！

pa++
pa = pa + 1
pa += 1
++pa
} 使pa指向下一个元素

*pa++
*(pa++)
} 先取*pa值，后使pa自增

*(++pa)　先使pa自增，再取*pa

(*pa)++　使pa所指向的元素值自增

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

执行结果：1 1 2 3 4

被指向的元素需和指针的基类型一致

- int 和 *　　int *pi;
- float 和 *　　float *pf;
- double 和 *　　double *pd;
- char 和 *　　char *pc;

- int [10] 和　　int (*q)[10];

用指针操纵数组-2

- 指向整个数组的指针变量，即数组的指针（注意与指针数组的区别）。
  - 一个指针类型的基类型可以是int、float这样的基本类型，也可以是数组这样的派生类型。比如，

```
typedef int A[10];
A *q;
```

  - 或者合并写成，

```
int (*q)[10];
```

  - 该指针变量q可以存储一个类型为A的数组的地址，比如，

```
int a[10];
q = &a;  //q相当于一个二级指针变量
```

举 例:

- int a[3] = {1, 2, 3};
- int (*p)[3];
- p = &a;
- (*p)[i] 与 a[i]等价, i = {1, 2, 3}

int *p[10] V.S. int (*p)[10]

- int *p[10]中p是一个数组。可以理解为

  int* p[10]，先定义一个一维数组，再看括号外，数组中每个变量都是int型指针。

- int (*p)[10]中p是一个指针。它的类型是：指向int x[10]这样的一维数组的指针。

用指针操纵二维数组-1

- 对于二维数组，可以通过不同级别的指针变量来操纵。
  注意：二维数组名表示第一行的地址。

  - 比如，

```
int b[5][10];
int *p;
p = &b[0][0];  //或“p = b[0];”
// 一级指针变量可以存储二维数组b某一元素的地址
```

  - 然后可以通过指针移动法指定某个元素地址，再取值指定任一元素，比如 p++，*p;

用指针操纵二维数组-2

- 或，

```
int (*q)[10];
q = &b[0];          //或“q = b;”
//二级指针变量可以存储二维数组b某一行的地址
```

- 然后可通过下标法指定元素，比如q[i][j]（相当于b[i][j]）；
- 也可以通过指针移动法指定某一行的地址，再取值指定某一行的首元素，比如q++，**q;
- 还可以通过偏移量法指定任一元素，比如
  - *(*(q+i)+j)（相当于*(*(b+i)+j)）
  - *(&q[0][0]+10*i+j)（相当于*(&b[0][0]+10*i+j)）
  - *(q[i]+j)（相当于*(b[i]+j)）
  - (*(q+i))[j]（相当于(*(b+i))[j]）

用指针代表的地址来理解！！！

用指针操纵二维数组-3

- 或

```
  int b[5][10];
int (*r)[5][10];
r = &b;
//三级指针变量可以存储整个二维数组b的地址
```

三者的区别

```
int b[5][10];

int *p;

int (*q)[10];

int (*r)[5][10];

p = &b[0][0]; // q = b[0];

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

r = &b;
```

重点

用指针在函数间传递数据

- 传址调用
  - 指针变量可以用作函数的形参，以提高函数间大量数据（比如数组）的传递效率。

例：指针变量作为函数形参

```
#include <stdio.h>
#define N 10
int Fun(int *pa, int n);
int main( )
{   int a[N] = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
    int sum = Fun(a, N);     // int *pa = a
    printf("%d \n", sum);
    return 0;
}
 int Fun(int *pa, int n)
{   int s = 0;
    for(int i = 0; i < n; i++)
            s += *(pa + i);
    return s;
}
```

不用指针变量做参数如何写？

```
Int Fun(int a[], int n)
{
    …
    a[i]
    …
}
```

- 可见，当函数的形参定义成指针变量，实参是数组名时，调用时不是将实参的副本通过赋值的形式一一传递给形参，而是通过指针变量（地址）直接访问实参，以提高数据的“传递”效率。这种函数调用方式通常叫做传址调用。

- 实际上，C语言中，写成数组定义形式的形参也是按指针类型数据处理的，即只给形参分配1个字的内存空间，以存储实参第一个元素的地址，只不过接下来不是通过指针变量操纵数组，而是通过基于数组名的下标法或偏移量法访问数组元素。

- 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”效率外，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：

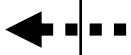
```
*pa= 5; // 会改编实参数组的值
```

- 被调函数的return语句只能返回一个值，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：利用函数的副作用（☹）实现两个数据的交换

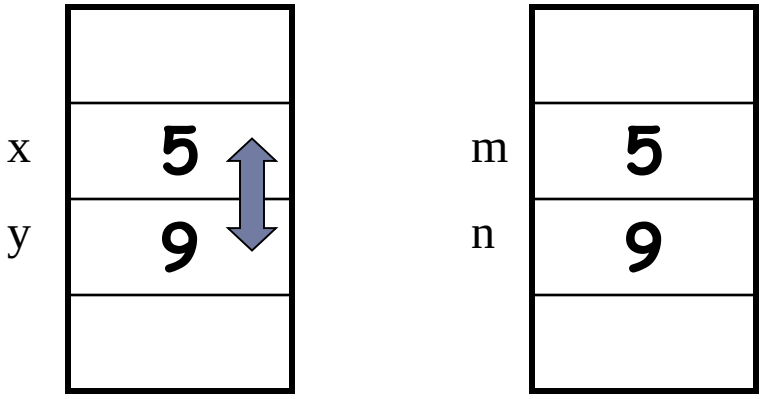
```
#include <stdio.h>
void Swap(int *pm, int *pn);
int main( )
{
    int m = 5, n = 9;
    Swap(&m, &n);              // int *pm = &m，int *pn = &n，传址调用
    printf("%d  %d", m, n);
    return 0;
}
void Swap(int *pm, int *pn)
{
    int temp = *pm;
    *pm = *pn;
    *pn = temp;
}
```

	形参	实参	特点	举例

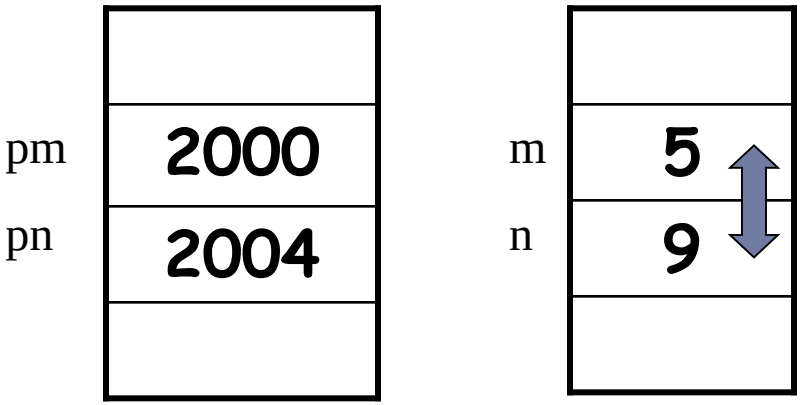
传值调用	变量 	常变量值
		量量值
		值值
		表达式值
			形参的改变不影响实参	void Swap1(int x, int y)
				{
				int temp = x;
				x = y;
				y = temp;
				}
				int x=m
				int y=n
				int main( )
				{
				int m = 5, n = 9;
				Swap1(m, n);
				printf("%d %d", m, n);
				return 0;
				}

传址调用	指针 	地址值	改变形参所指向的变量值来影响实参	void Swap (int *pm, int *pn)
				{
				int temp = *pm;
				*pm = *pn;
				*pn = temp;
				}
				int *pm=&m
				int *pn=&n
				int main( )
				{
				int m = 5, n = 9;
				Swap(&m, &n);
				printf("%d %d", m, n);
				return 0;
				}

Swap1



Swap

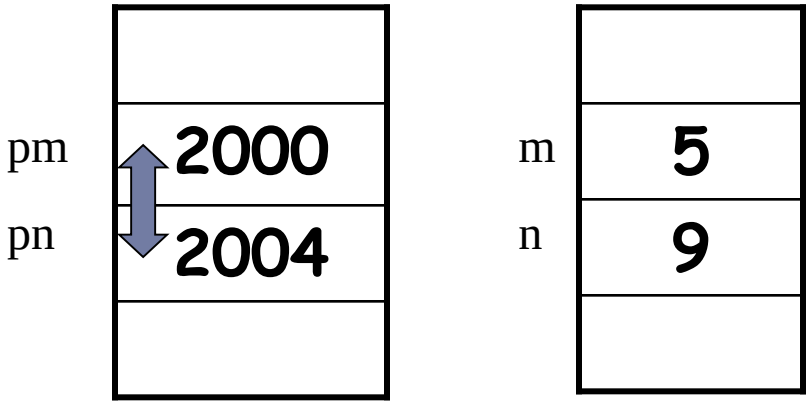



```
int *pm=&m
int *pn=&n
```

```
void Swap2(int *pm, int *pn)
{
  int *temp = pm;
  pm = pn;
  pn = temp;
}
```

```
int main( )
{
  int m = 5, n = 9;
  Swap2(&m, &n);
  printf("%d %d", m, n);
  return 0;
}
```

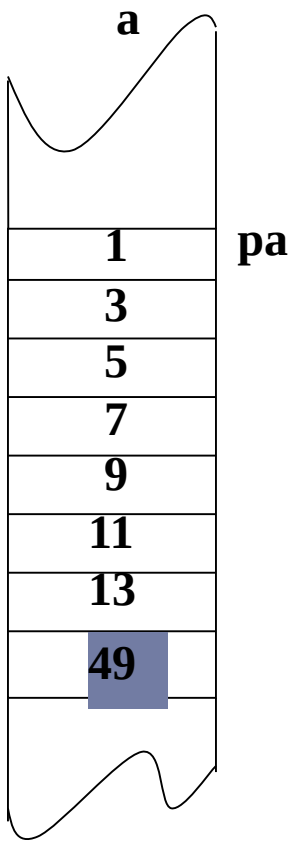
Swap2



例：指针变量作为函数形参的双重作用

```
#include <stdio.h>
#define N 8
void Fun(int * pa, int n);
int main( )
{   int a[N] = {1, 3, 5, 7, 9, 11, 13};//注意a[N-1]初始化为0
    Fun(a, N);       // int *pa = a
    printf("%d \n", a[N-1]);
    return 0;
}
void Fun(int * pa, int n)
{   for(int i = 0; i < n-1; i++)
            *(pa + n - 1) += *(pa + i);
}
```

int *pa=a



//既利用传址调用提高了数据正向“传递”的效率

//又利用函数的副作用实现了数据的反向“传递”(修改了数组最后一个元素的值)

思考：以下程序是否有问题

写出下面程序的运行结果：

```
#include <stdio.h>
fun(int *a,int *b)
{
   int *w;
   *a=*a+*a;
   *w=*a;
   *a=*b;
   *b=*w;
}
main()
{
  int x=9,y=5,*px=&x,*py=&y;
  fun(px,py);
  printf( “%d, %d\n” ,x,y);
}
```

程序的编译错误：

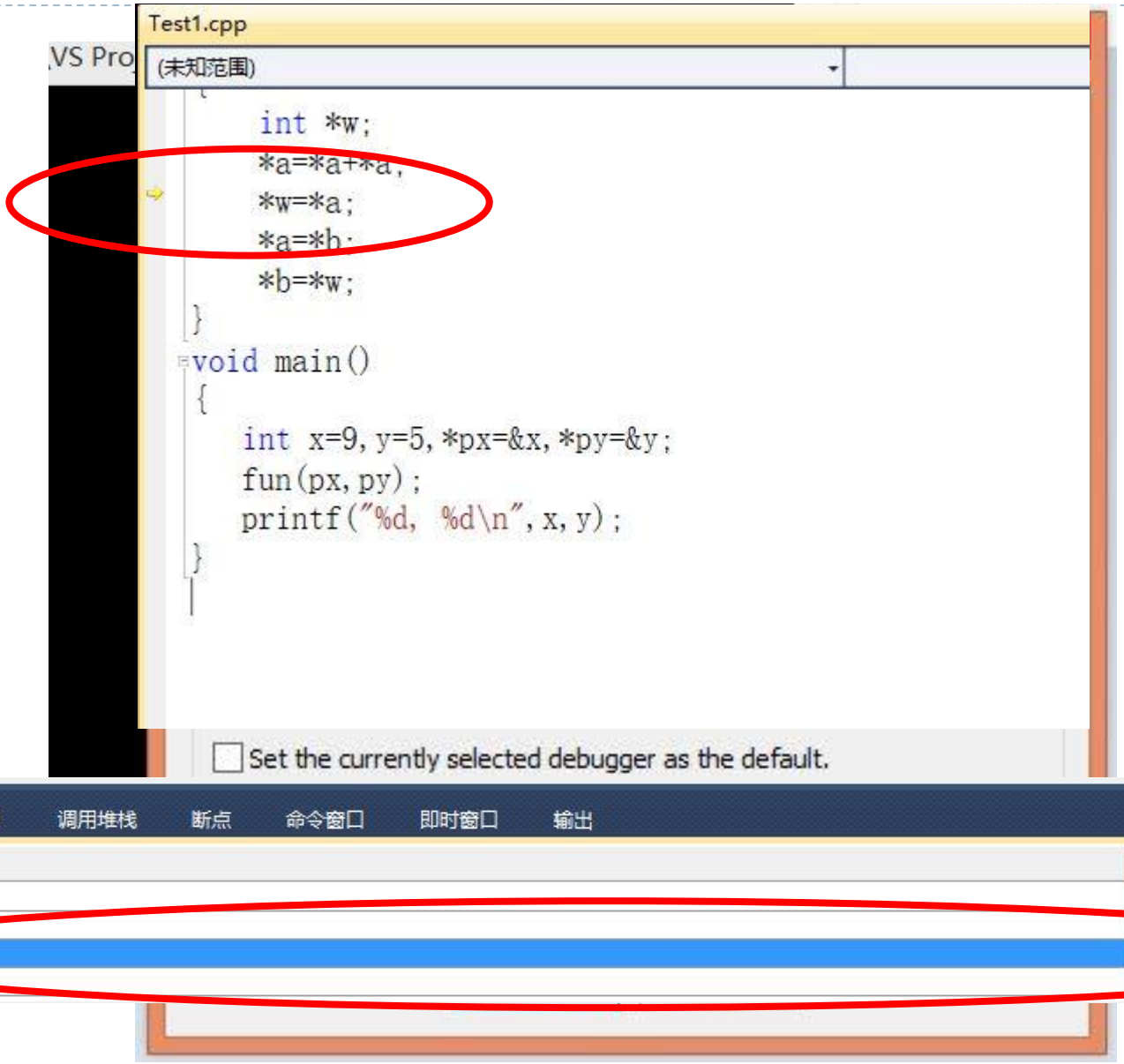
fun(), main()返回值呢？？

程序的运行错误：

Runtime…failure: … w is being used without being initilized

如何改？？

调试小贴士


Test1.cpp
(未知范围)
int *w;
*a=*a+*a;
*w=*a;
*a=*b;
*b=*w;
}
void main()
{
int x=9, y=5, *px=&x, *py=&y;
fun(px, py);
printf("%d, %d\n", x, y);
}
Set the currently selected debugger as the default.
监视 1
调用堆栈
断点
命令窗口
即时窗口
输出
名称
值
a
0x0113f8b4
w
0xcccccccc

回顾：一个完整的例子（C++）

- 例0 计算一组圆（直径为n以内的正整数）的周长之和（计量单位为米）。

```
#include <iostream>
using namespace std;
const double PI = 3.14;
int main( )
{  int n, d = 1;              //d初
   double sum = 0;            //sur
   char ch = 'm';
   cout << "Input n: " ;
   cin >> n;
   ……
   return 0;
}
```

```
while(d <= n)
{
        sum = sum + PI * d;
        d = d + 1;
}

cout << "The sum is: " << sum;
cout << ch;
```

const：用来定义常量 （C++）

```
const double pi = 3.1415926;
//用const定义常量时，必须初始化
```

- 这种常量的定义形式往往比

#define PI 3.14159 （C语言）

更好，因为系统会对其进行类型检查，而不是只进行文本替换。

const：指向常量的指针

- 指针变量作为形参一般可以起到1. 提高数据正向“传递”的效率与2. 函数的副作用（给*p赋值）两个效果。
- 如何克服函数的副作用？用关键词const将形参设置成指向常量的指针类型。
  - 比如，

```
void F(const int *p, int num)   // 或const int p[]
{
    ......
    *p = 1;// 不允许该操作，同样地，“p[i] = 1;”也是不允许的
    ......
}  //该程序通过const限制程序员在被调函数中修改实参的值
```

const的不同位置含义不同

- const double PI = 3.1415926;

  //定义const浮点类型的数据

- void F(const int *p, int num)

  // p指向的整形不能被改变值

int n = 0;	
const int m = 0;	// m为整型常量，其值不可以改变
const int *p1;	// p1为指向常量的指针变量，*p1的值不可以改变
p1 = &m;	// p1的值可以改变
p1 = &n;	// 允许，只不过不能通过p1修改n的值
int * const p2 = &n;	// p2是指向变量的指针常量，其值不可以改变
	// *p2的值可以改变
int * const q2 = &m;	// 不允许,以防止通过q2修改m的值
const int * const p3 = &m;	// p3为指向常量的指针常量,其值与*p3的值都不可 // 改变
const int * const q3 = &n;	// 允许，只不过不能通过q3修改n的值
n++;	// 允许
m++;	// 不允许
p1++;	// 允许
p2++;	// 不允许
p3++;	// 不允许

int n = 0;	
const int m = 0;	// m为整型常量，其值不可以改变
const int *p1;	// p1为指向常量的指针变量，*p1的值不可以改变
p1 = &m;	// p1的值可以改变
p1 = &n;	// 允许，只不过不能通过p1修改n的值
int * const p2 = &n;	// p2是指向变量的指针常量，其值不可以改变
	// *p2的值可以改变
p2 = &m;	// 不允许
const int * const p3 = &m;	// p3为指向常量的指针常量，其值与*p3的值都不可 // 改变
p3 = &n;	// 不允许
int *q;	
q = &n;	// 允许
q = &m;	// 不允许，以防止通过q修改m的值

函数间有多种通讯方式

- 传值方式(把实参值的复制给形参)
- 利用函数返回值传递数据
- 通过全局变量传递数据
  (根据“切实”需要定义全局变量)
- 通过指针/引用类型参数传递数据
  (函数副作用问题，可以通过指向常量的指针来避免)

指针型函数

- C语言的return语句只能返回一个值（不能返回数组类型的数据），但可以返回数组或数组元素的地址。当函数的返回值是一个地址时，称该函数为指针型函数

```
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

例：指针型函数示例

```
#include <stdio.h>
#define N 5
int *Max(const int ac[ ], int num);
int main( )
{
   int a[ ] = {1, 2, 5, 4, 3};
   *Max(a, N) = 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;
}
```

指针型函数一般不能返回局部变量的地址

- 比如调用以下程序可能带来意想不到的问题：

```
int *F( )
{
    int m, n;
    scanf("%d%d", &m, &n);
    if(m > n)
        return &m;
    else
        return &n;
}
```

- 避免主调函数在调用多个函数时，指针型函数返回的局部变量内存空间在主调函数使用前被其他函数使用，从而使程序发生错误。

```
int main( )
{   int *p = F( );          //p指向i
    int *q = G( );          //q指向j
    int x = *p + *q;        //x为2，因为i的内存空间又被j占用
    printf( "%d \n", x);    //输出2
    return 0;
}
```

```
int *F( )
{       int i = 0;
        return &i;
}
 int *G( )
{       int j = 1;
        return &j;
}
```

思 考：

- 对输入的100个数进行排序：

```
int a[100];
for (int i=0; i<100; i++) cin >> a[i];
sort(a,100);
```

- 对输入时指定的若干个数进行排序，下面的做法可行吗？

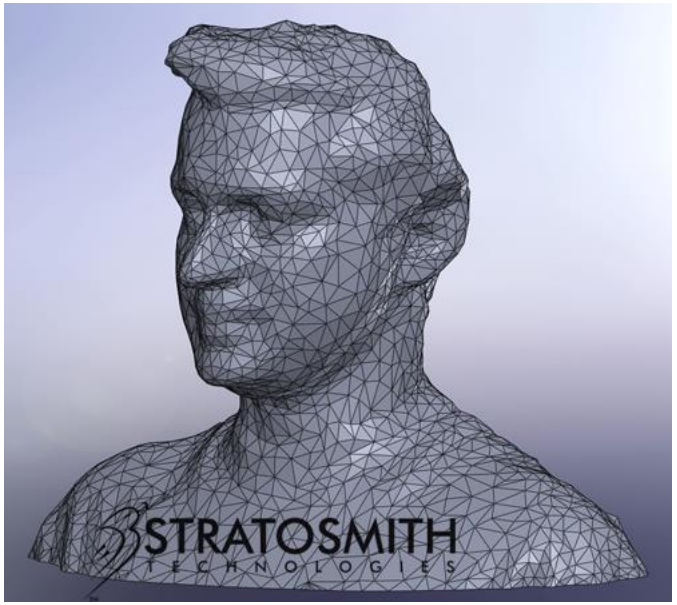
```
int n;
cin >> n; //待排序的数的个数
int a[n]; //?
for (int i=0; i<n; i++) cin >> a[i];
sort(a,n);
```

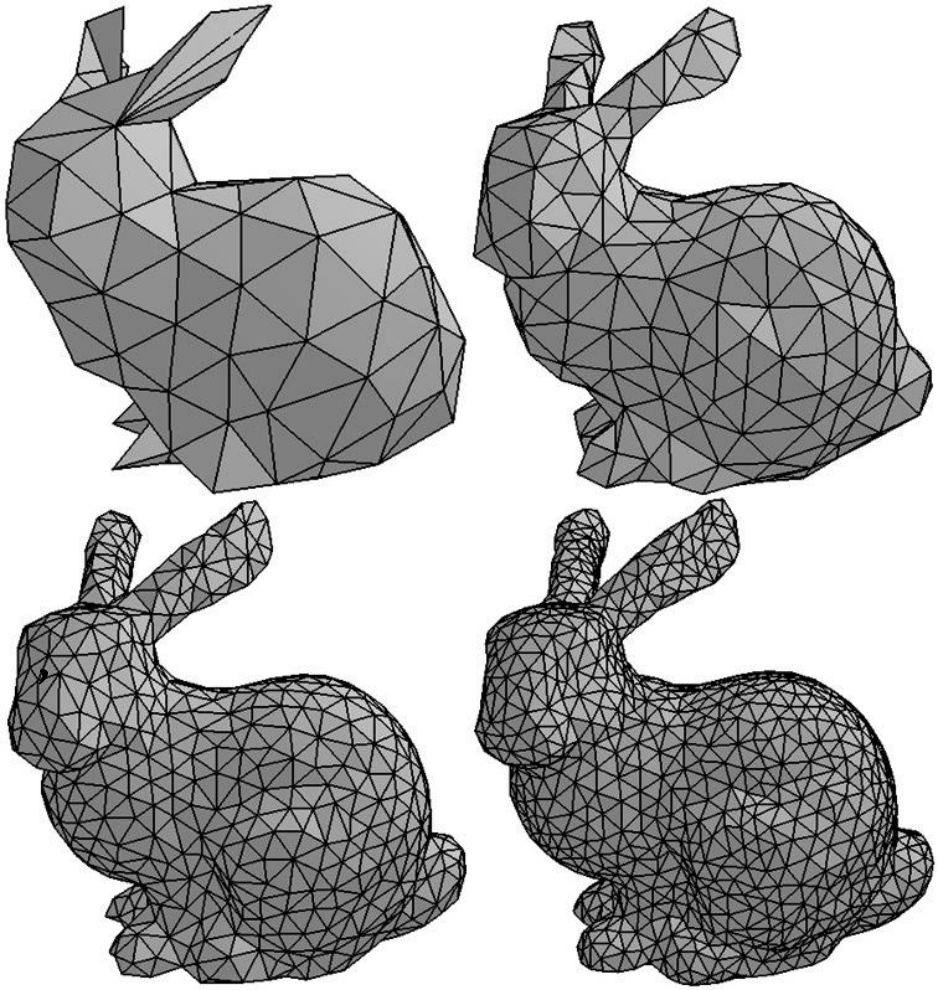
思 考：
编写一个程序，读入图像，并进行图像处理

```
#define M 1920
#define N 1080
bool read_image(CIpl *image) //从硬盘读入图像
// 不同图像的分辨率不同
{
    int pixel_r[M][N];
    //？同样对于G（绿色），B（蓝色）通道定义？
    …


}
```

思 考：
同样对于几何模型的处理(不同模型分辨率不一)





数组的长度通过硬盘读入
（或通过程序运行计算得到），
而非预先设定好的固定值！

如何解决这些问题？

- （一维、二维）数组长度过长、占用过多内存空间
  - 尝试下开辟一个超大的数组？☹

- 数组长度是变量，其值通过用户输入、文件读取或程序计算得到

借助指针定义动态变量\数组

动态变量的定义（C形式）

- 指在程序运行中，由程序根据需要所创建的变量。例如：
  - int *p1;
  - p1 = (int *)malloc(sizeof(int)); //#include <cstdlib> C

- 再例如：
  - int *p2;
  - p2 = (int *)malloc(sizeof(int)*n);

// 创建了一个由n（n可以是变量）个int型元素所构成的动态
// 数组变量，“相当”于int a[n];

- malloc库函数的原型是

  void *malloc(unsigned int size);

  - 该函数在stdlib.h中声明，其功能是在程序的堆区分配size个单元，并返回该内存空间的首地址，其类型为：void *。调用时一般需要用操作符sizeof计算需要分配的单元个数作为实参，并对返回值进行强制类型转换，以便存储某具体类型的数据。
  - 比如：

```
(int *)malloc(sizeof(int));
                        // 创建一个int型动态变量，可存储1个int型数据
(double *)malloc(sizeof(double) * n);
                        // 创建一个double型动态数组，含n个元素
(int *)malloc(sizeof(int) * n * 10);
                        // 创建一个int型n行10列二维动态数组
```

动态变量的撤销（C形式）

- 在C/C++中，动态变量需要由程序显式地撤消（使之消亡）。例如：
  - free(p1); // 撤消p1指向的int型动态变量
- 再例如：
  - free(p2); // 撤消p2指向的动态数组
- 用malloc创建的动态变量则需要用free撤销。

动态变量的定义（C++形式）

- 指在程序运行中，由程序根据需要所创建的变量。例如：
  - int *p1;
  - p1 = new int; //C++; 创建了一个int型动态变量，p1指向之 。

- 再例如：
  - int *p2;
  - p2 = new int[n];

// 创建了一个由n（n可以是变量）个int型元素所构成的动
// 数组变量，“相当”于int a[n]

- 对于一个动态的n维数组，除了第一维的大小外，其它维的大小必须是常量或常量表达式。例如：

```
int (*q)[20]; //q为一个指向由20个int型元素所构成的数组的指针
// 或者 typedef int A[20];
// A *q;
int n;
.....
q = new int[n][20]; //创建一个n行、20列的二维动态数组，
                    //返回第一行的地址。 等价于：q = new A[n];
... q[i][j] ... //访问q指向的二维数组的第i行、第j列的元素
```

- 如何创建一个m行、n列的动态数组？
  - 用一维数组实现：int *p=new int[m*n];
  - 第i行、第j列元素：*(p+i*n+j)

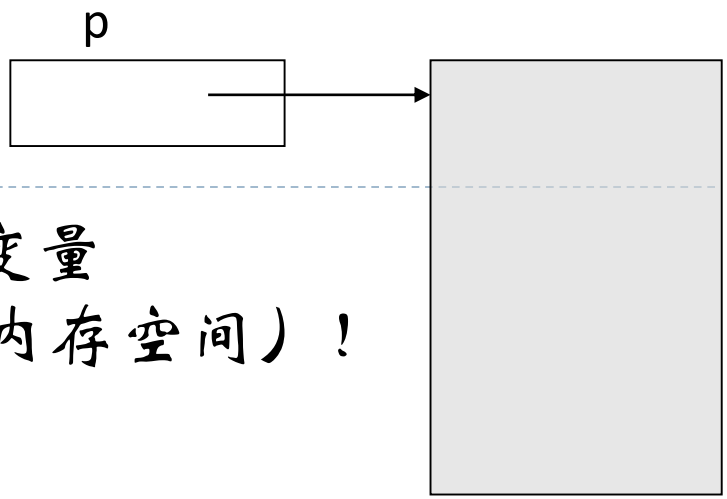
通过指针访问动态变量

- 动态变量没有名字，对动态变量的访问需要通过指向动态变量的指针变量来进行（间接访问）。例如：

```
int *p,*q;
p = new int;
...*p... //访问上面创建的int型动态变量
q = new int[n];
...*(q+3)... //或...q[3]...，访问上面创建的
                //动态数组中的第4个元素
```

动态变量的撤销（C++形式）

- 在C/C++中，动态变量需要由程序显式地撤消（使之消亡）。例如：
  - `delete p1;  // 撤消p1指向的int型动态变量`
- 再例如：
  - `delete []p2; // 撤消q2指向的动态数组`
- 用new创建的动态变量需要用delete来撤销



- 用delete和free只能撤消动态变量
（撤销的是动态变量所在的堆区内存空间）！
  - int x,*p;
  - p = &x;
  - delete p; //Error
- 用delete和free撤消动态数组时，其中的指针变量必须指向数组的第一个元素！
  - int *p=new int[n];
  - p++;
  - delete []p; //Error

  或
  - int *p=(int *)malloc(sizeof(int)*n);
  - p++;
  - free(p); //Error

动态变量的生存期

- 动态变量具有动态生存期：
  - 动态生存期是指从new操作或函数调用malloc后开始，到delete操作或函数调用free时结束的时间段。
  - 一个动态变量创建后，只要不对它进行delete操作或函数调用free，它就一直存在，直到程序执行结束。
- 动态变量的空间是在堆区中分配。

栈区和堆区

- 栈区（stack）

  由编译器自动分配释放，存放函数的参数值，局部变量的值等，内存的分配是连续的，类似于平时我们所说的栈，如果还不清楚，那么就把它想成数组，它的内存分配是连续分配的，即:所分配的内存是在一块连续的内存区域内。当我们声明变量时，那么编译器会自动接着当前栈区的结尾来分配内存

- 堆区（heap）

  一般由程序员分配释放，若程序员不释放，程序结束时可能由操作系统回收。类似于链表，在内存中的分布不是连续的，它们是不同区域的内存块通过指针链接起来的。一旦某一节点从链中断开，我们要人为的把所断开的节点从内存中释放

注意：“内存泄漏”问题

- 内存泄漏（memory leak）：
  - 没有撤消动态变量，而把指向它的指针变量指向了别处或指向它的指针变量的生存期结束了，这时，这个动态变量存在但不可访问（这个动态变量已成为一个“孤儿”），从而浪费空间。

    例如：

```
int x,*p;
p = new int[10]; //动态数组
p = &x; //之后，上面的动态数组就访问不到了！
```

注意：“悬浮指针”问题

- 悬浮指针（dangling pointer）：
  - 用delete或free撤消动态变量后，C++编译程序一般不会把指向它的指针变量的值赋为0，这时该指针指向一个无效空间。例如：

```
int *p;
p = new int;
.......
delete p; //撤销了p所指向的动态变量
.......
*p = NULL; // ?
```

建议后面上机用C++风格的动态变量！

```
int n, *q=NULL;  //初始化一般放构造函数中
n= …
q = new int[n];  //创建
 ....
q[i] // i=0,...,n-1
 ...

delete []q;          //撤销
q=NULL;          //一般放析构函数中
```

例：动态变量的应用——动态数组

- 对输入的若干个数进行排序，如果输入时先输入数的个数，然后再输入各个数，则可用下面的动态数组来表示这些数：

```
int n;
int *p=NULL;
cin >> n;
p = new int[n];
for (int i=0; i<n; i++)
    cin >> p[i];
sort(p,n);
……
delete []p;
p = NULL;
```

例：对输入的若干个数进行排序，在输入时，先输入各个数，最后输入一个结束标记（如：-1）：

```
const int INCREMENT=10;
int max_len=20,count=0,n,*p=new int[max_len];
cin >> n;
while (n != -1)
{
    if (count >= max_len)
    {       max_len += INCREMENT;
            int *q=new int[max_len];
            for (int i=0; i<count; i++)
                  q[i] = p[i];
            delete []p;
            p = q;
    }
    p[count] = n;
    count++;
    cin >> n;
}
sort(p,count);
……
delete []p;
p =NULL;
```

- 上面的实现方法虽然可行，但是，
  - 当数组空间不够时，它需要重新申请空间、进行数据转移以及释放原有的空间，这样做比较麻烦并且效率有时不高。
  - 当需要在数组中增加或删除元素时，它还将会面临数组元素的大量移动问题。
- 链表可以避免数组的上述问题！

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的变量（简称函数指针，注意与指针型函数的区别），然后通过函数指针来调用函数。
  - 比如，

```
typedef int (*PFUNC)(int); // 构造了一个函数指针类型
PFUNC pf;          // 定义了一个函数指针
```

  - 也可以在构造函数指针类型的同时直接定义函数指针：

```
int (*pf)(int);
//第一个int为函数的返回值类型，第二个int为参数的类型。
```

- 对于一个函数，比如int F(int m) { ... }，可以用取地址操作符&（或直接用函数名）来获得其内存地址。
  - 比如，

    pf = &F;
  - 或

    pf = F;

  - 这样，函数指针pf指向内存的代码区（而不是数据区），接下来可以通过函数指针调用函数：

    (*pf)(10);
  - 或

    pf(10); // 实参为10，调用函数F

例：根据要求(输入的值除8的余数)，执行在函数表中定义的某个函数（8个函数中的一个）

```
#include <stdio.h>
#include <cmath>
typedef double (*PF)(double); // 构造函数指针类型
PF func_list[8] = {sin, cos, tan, asin, acos, atan, log, log10}; // 定义长度为8的函数指针的数组
int main( )
{   int index;
    double x;
    do
    {     printf("请输入要计算的函数(0:sin 1:cos 2:tan 3:asin 4:acos 5:atan 6:log 7:log10):\n");
          scanf("%d", &index);
    } while(index < 0 );
    printf("请输入参数：");
    scanf("%lf", &x);
    printf("结果为： %d \n", (*func_list[index%8])(x));
    return 0;
}
```

将一个函数作为另一个函数的形参
例：根据要求计算不同函数的积分

- 还可以把一个函数作为参数传给被调用函数，被调用函数（实参）对应的形参定义为一个函数指针类型，调用时的实参为一个函数的地址。
  - 比如，

```
double Integrate(double (*pfun)(double x), double x1, double x2)
{   double s = 0;
    int i = 1, n;       //i为步长，n为等份的个数，n越大，计算结果精度越高
    printf("please input the precision: ");
    scanf("%d", &n);
    while(i <= n)
    {       s += (*pfun)(x1 + (x2 - x1) / n * i);
            i++;
    }
    s *= (x2 - x1) / n;
    return s;
}
```

例：根据要求计算不同函数的积分

- 函数Integrate可以计算任意一个一元可积函数（由函数指针pfun操纵）在一个区间[x1, x2]上的定积分，该函数的调用形式：

```
double My_func(double x)
{       double f = x;
        return f;
}
```

- Integrate(My_func, 1, 10); //C语言中，一个函数总是占用一段
  // 连续的内存区，而函数名就是该函数所占内存区的首地址
  //计算函数My_func在区间[1, 10]上的定积分
- Integrate(sin, 0, 1);
  //计算函数sin在区间[0, 1]上的定积分
- Integrate(cos, 1, 2);
  //计算函数cos在区间[1, 2]上的定积分

内容回顾

- 指针及其应用-1
  - 指针的基本概念
    - 指针类型的构造
    - 指针变量的定义与初始化
    - 指针数组
    - 多级指针变量
    - 通用指针与void类型
    - 指针类型相关的基本操作

```
int *
float *
double *
char *
```

```
int * pi = &i;
float * pf = &f;
double * px = &x;
char * pch = &ch;
```

```
int * ap[3] = {&i, &j, &k};
```

```
int ** pp = &pi;
```

```
void *
```

```
void * pv = 0;
```

```
* (与&互逆)
=
>  <  >=  <=  ==  !=
+n  -n  ++  --
-
[ ]
```

内容回顾

- 指针及其应用-2
  - 用指针操纵数组
  - 用指针在函数间传递数据
    - 传址调用
    - const
    - 指针型函数**
  - 用指针访问动态变量
    - 动态变量的创建、访问与撤销
    - 内存泄露与悬浮指针**
  - 用指针操纵函数**

```
int * ap[3] = {&i, &j, &k};
//注意ap与q的区别
```

```
int * p = a;
int (*q)[10] = &a;
int (*r)[5][10] = &b;
```

```
Fun(a, …);
void Fun(int *pa)
{ *pa…pa++…}
```

```
Fun(&n);
void Fun(int *pn)
{ *pn = …}
```

```
void Fun(const int *pa)
{ *pa…pa++…}
  *pa = …   ×
```

```
int * Fun(…)
{…return &…}
```

内容回顾

- 指针及其应用-2
  - 用指针操纵数组
  - 用指针在函数间传递数据
    - 传址调用
    - const
    - 指针型函数**
  - 用指针访问动态变量
    - 动态变量的创建、访问与撤销
    - 内存泄露与悬浮指针**
  - 用指针操纵函数**

```
int n; …
int q = new int [n];
delete []q;
q = NULL
```

q自身消亡或指向了别处，但q指向的堆区未撤销；
q自身未消亡、也未指向别处，而q指向的堆区已撤销。

```
int (*pfun1)() = &Fun1;
double (*pfun2)(int) = &Fun2;
```

```
int * Fun(…)
 {…return &…}
//注意Fun与pfun1的区别
```

内容小结

- 指针：
  - 一种派生数据类型，
  - 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。
- 要求：
  - 掌握指针的定义、初始化和操作方法
  - 掌握指针类型的特征及其典型应用场合
    - **可以在函数间高效地传递数据**
    - **用指针操纵数组**
    - **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。**
    - 此外，本章还介绍了用指针操纵函数的方法，以及指针数组、多级指针、void类型、const类型等概念
  - 继续保持良好的编程习惯

Q & A

什么是指针(pointer)

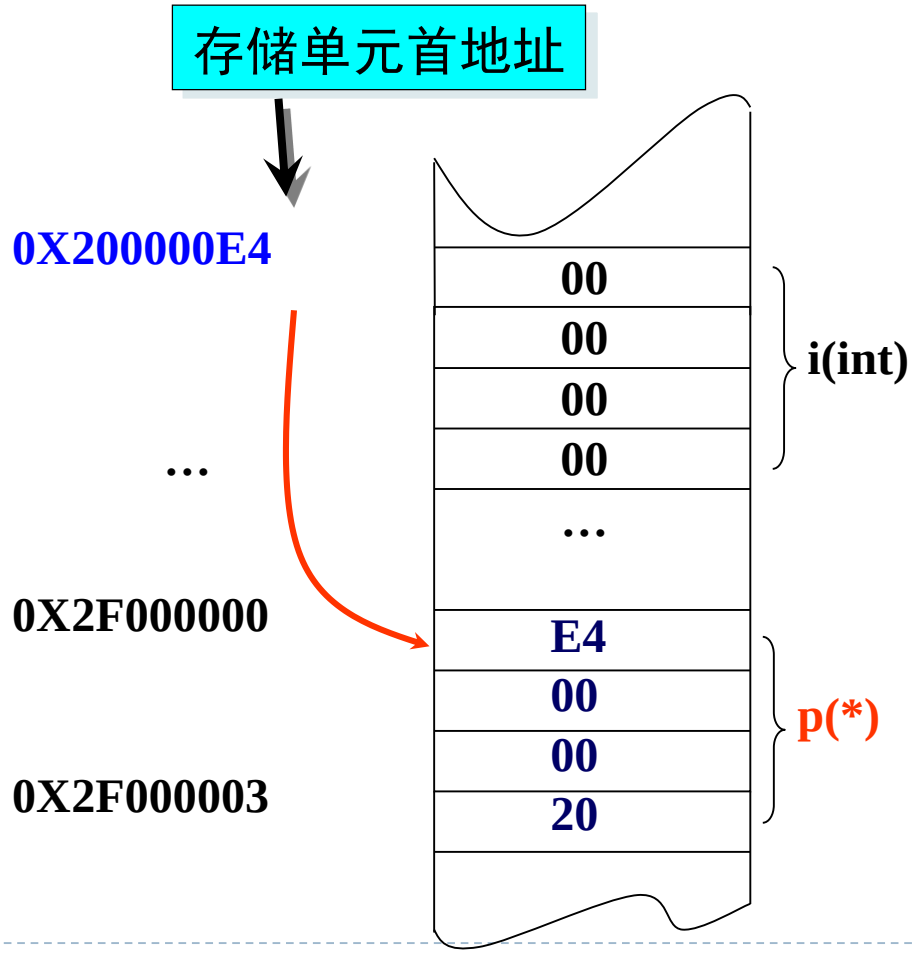
int i = 8;

变量名　　变量值

int *p;

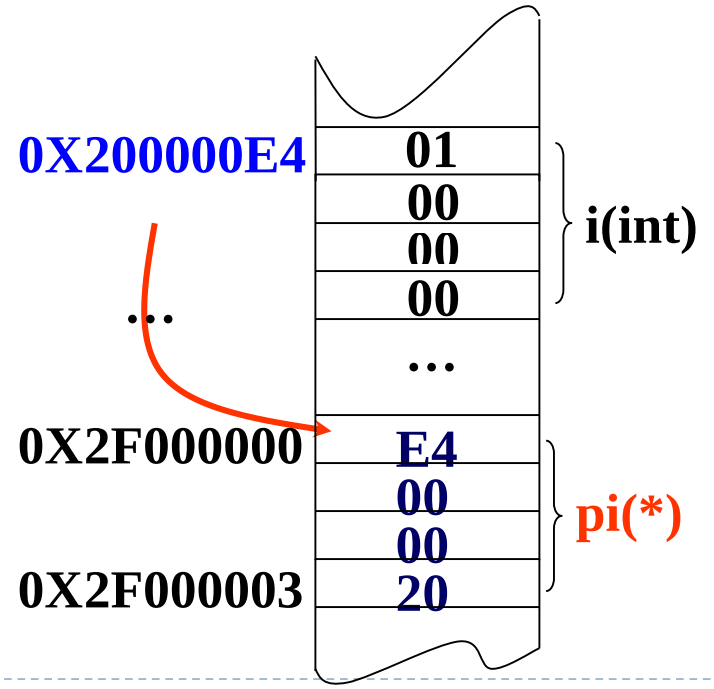
p = &i;

// p就是一个指针，指向i



举 例

```
int *pi=NULL;    //这里的 * 不是指针操作符
int i = 0;
pi = &i;         //pi指向i
*pi = 1;     //对pi进行取值操作并对其赋值，相当于i = 1;
*(&i) = 2;       //相当于i = 2;
pi = &(*pi);   //pi仍然指向i
```



指针数组

- 如果数组的每一个元素都是一个指针类型的数据，则该数组叫做指针数组。
  - 比如，

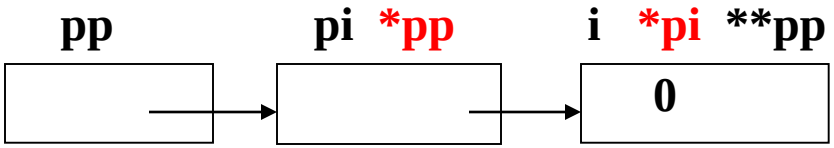
```
int i = 0, j = 1, k = 2;
int *ap[3] = {&i, &j, &k};
//ap这个指针数组的长度是3，各个元素的类型是int *
```

- 指针数组一般用于多个字符串的处理

多级指针变量

- C语言的指针变量还可以存储另一个指针变量的地址。
  - 比如，

```
int i = 0;
int *pi = &i;
int **pp = &pi;  //指针变量pp存储的是指针变量pi的地址
```



- 这时，称指针变量pp是一个二级指针变量(指向指针的指针)，它指向的变量是一个一级指针变量pi。

- 如果再定义一个指针变量ppp存储pp的地址，则ppp是一个三级指针变量。
- 多级指针变量通常用于指针类型数据的传址调用，或者用于多维数组和多个字符串的处理。

例：用指针变量操纵数组

```
#include <stdio.h>
#define N 10

int main( )
{   int i;
    int a[N] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9};
    int *pa;
    for(pa = a, i = 0; i < N; i++)
            printf("%d ", pa[i]);          //下标法，用a[i]也行
    for( ; pa < (a + N); pa++)             //指针移动法，用a++不行，因为a是常量
            printf("%d ", *pa);
    for(pa = a, i = 0; i < N; i++)    //这里没有“pa = a, i = 0”，则会出现越界!
            printf("%d ", *(pa + i));     //偏移量法，用*(a + i)也行
    return 0;
}
```

int *p[10] V.S. int (*p)[10]

- int *p[10]中p是一个数组。可以理解为

  int* p[10]，先定义一个一维数组，再看括号外，数组中每个变量都是int型指针。

- int (*p)[10]中p是一个指针。它的类型是：指向

  int [10]这样的一维数组的指针。

用指针操纵二维数组-1

- 对于二维数组，可以通过不同级别的指针变量来操纵。
  注意：二维数组名表示第一行的地址。

  - 比如，

```
int b[5][10];
int *p;
p = &b[0][0];  //或“p = b[0];”
// 一级指针变量可以存储二维数组b某一元素的地址
```

  - 然后可以通过指针移动法指定某个元素地址，再取值指定任一元素，比如 p++，*p;

用指针操纵二维数组-2

- 或，

```
int (*q)[10];
q = &b[0];          //或“q = b;”
//二级指针变量可以存储二维数组b某一行的地址
```

- 然后可通过下标法指定元素，比如q[i][j]（相当于b[i][j]）；
- 也可以通过指针移动法指定某一行的地址，再取值指定某一行的首元素，比如q++，**q;
- 还可以通过偏移量法指定任一元素，比如
  - *(*(q+i)+j)（相当于*(*(b+i)+j)）
  - *(&q[0][0]+10*i+j)（相当于*(&b[0][0]+10*i+j)）
  - *(q[i]+j)（相当于*(b[i]+j)）
  - (*(q+i))[j]（相当于(*(b+i))[j]）

用指针代表的地址来理解！！！

内容回顾

	形参	实参	特点	举例

传值调用	变量 ←	常变表
		量量达
		值值式
		值
			形参的改变不影响实参	void Swap1(int x, int y)
				{
				int temp = x;
				x = y;
				y = temp;
				}
				int x=m
				int y=n
				int main( )
				{
				int m = 5, n = 9;
				Swap1(m, n);
				printf("%d %d", m, n);
				return 0;
				}

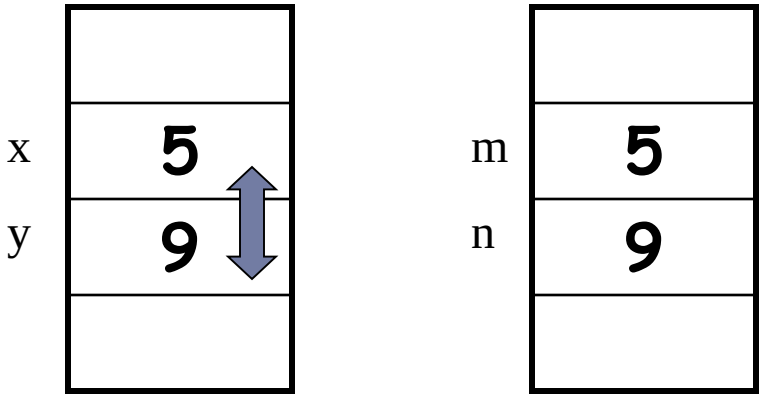
传址调用	指针 ←	地址值	改变形参所指向的变量值来影响实参	void Swap (int *pm, int *pn)
				{
				int temp = *pm;
				*pm = *pn;
				*pn = temp;
				}
				int *pm=&m
				int *pn=&n
				int main( )
				{
				int m = 5, n = 9;
				Swap(&m, &n);
				printf("%d %d", m, n);
				return 0;
				}

内容回顾

int *pm=&m

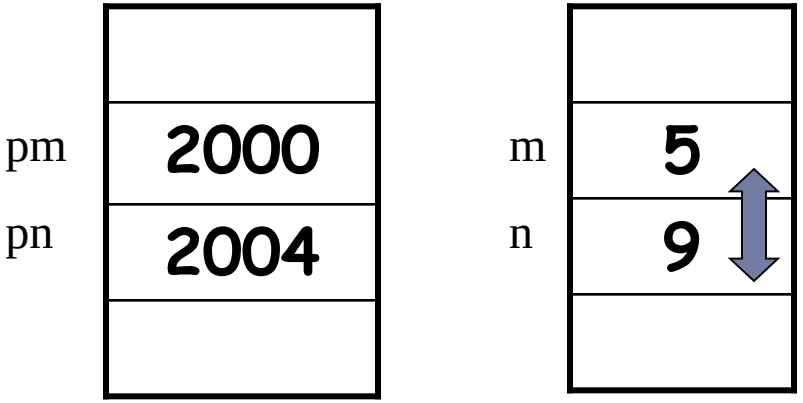
int *pn=&n

Swap1

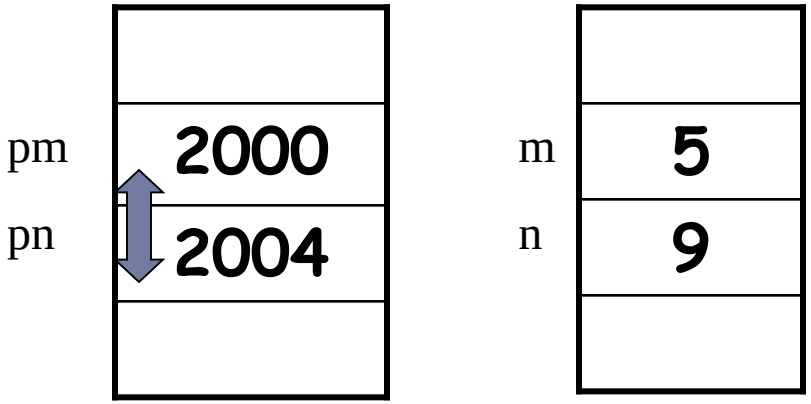



```
void Swap2(int *pm, int *pn)
{
  int *temp = pm;
  pm = pn;
  pn = temp;
}
```

```
int main( )
{
  int m = 5, n = 9;
  Swap2(&m, &n);
  printf("%d %d", m, n);
  return 0;
}
```



Swap

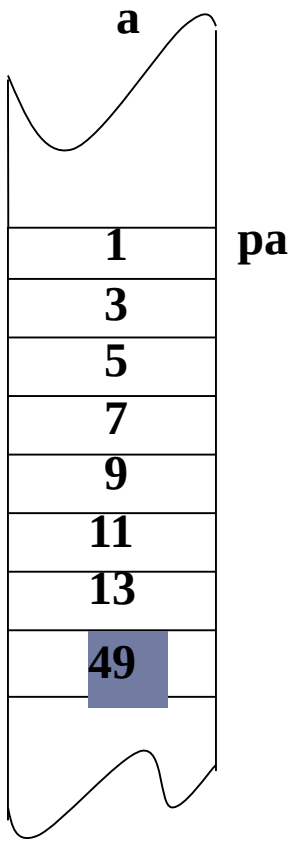


Swap2

例：指针变量作为函数形参的双重作用

```
#include <stdio.h>
#define N 8
void Fun(int * pa, int n);
int main( )
{   int a[N] = {1, 3, 5, 7, 9, 11, 13};//注意a[N-1]初始化为0
    Fun(a, N);       // int *pa = a
    printf("%d \n", a[N-1]);
    return 0;
}
void Fun(int * pa, int n)
{   for(int i = 0; i < n-1; i++)
            *(pa + n - 1) += *(pa + i);
}
```

int *pa=a



//既利用传址调用提高了数据正向“传递”的效率

//又利用函数的副作用实现了数据的反向“传递”(修改了数组最后一个元素的值)

const的不同位置含义不同

- const double pi = 3.1415926;

  //定义const浮点类型的数据

- void F(const int *p, int num)

  // p指向的整形不能被改变值

函数间有多种通讯方式

- 传值方式(把实参的复制给形参)
- 利用函数返回值传递数据
- 通过全局变量传递数据
  (根据“切实”需要定义全局变量)
- 通过指针/引用类型参数传递数据
  (函数副作用问题，可以通过指向常量的指针来避免)

内存泄露**

- 对于动态变量，如果没有用free库函数显式地撤销，那么当指向它的指针变量的生存期结束后（比如其所属函数执行完毕但整个程序尚未执行完毕时），或者指向它的指针变量指向了别处，则该动态变量仍然存在，但却无法访问，从而造成内存空间的浪费，这一现象称作“内存泄露” 。
  - 比如，

```
double *pda;
int m, n = 5;
pda = (double *)malloc(sizeof(double) * n);  //pda指向动态数组
pda = &m;  //     pda指向m，上面的动态数组造成内存泄露
```

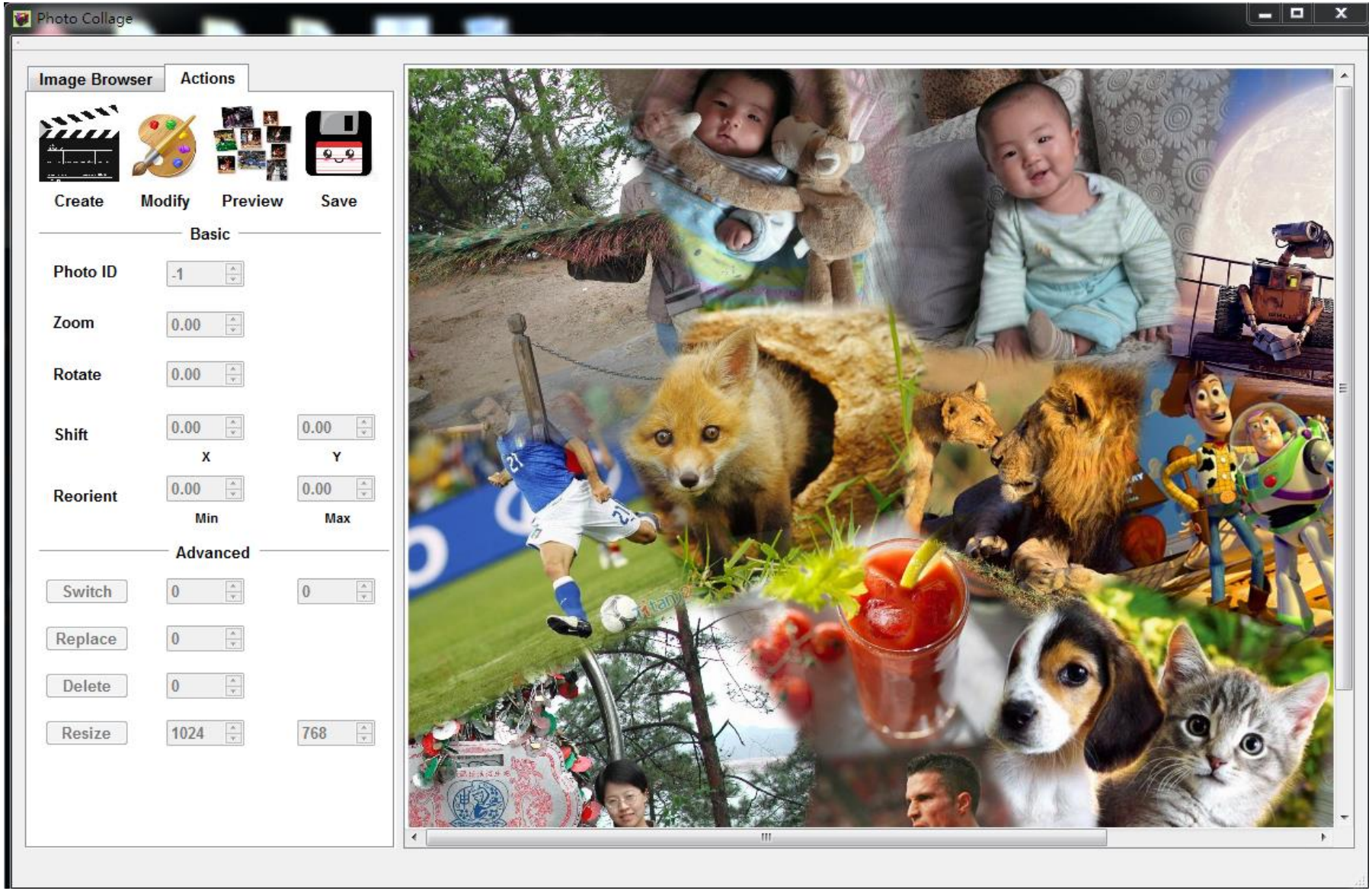
悬浮指针**

- 动态变量在用free库函数撤销后，指向它的指针变量则指向一个无效空间，这时该指针变量变为“悬浮指针”（dangling pointer）。即前图中阴影区域的内存空间释放后，指针变量p则变为“悬浮指针”
  - 比如，

```
double *pda;
int m, n = 5;
pda = (double *)malloc(sizeof(double) * n);  // pda指向动态数组
free(pda);
// pda变为“悬浮指针”，不能通过pda访问数据，比如不可*pda = 0
```

Our PhotoCollage TOOL

http://cs.nju.edu.cn/ywguo/PhotoCollage/Index.html



Zongqiao Yu, Lin Lu, **Yanwen Guo**, Rongfei Fan, Mingming Liu, Wenping Wang, Content-aware Photo Collage Using Circle Packing, IEEE Trans. on Visualization and Computer Graphics, 2014, 20(2): 182-195.

Microsoft AutoCollage

Microsoft Research
AutoCollage

- Home
- About
- Download
- Press

Media
- Demo Video
- Image Gallery
- Meet the Team

Support
- Help
- Forum
- Flickr community

Contributors
- Innovation Development
- Vision Technology
- Constraint Reasoning

Research Topics
- Automatic Collage
- Face Detection
- Image Segmentation
- Image Blending

Microsoft Research AutoCollage 2008

What is AutoCollage?
Photo collages celebrate important events and themes in our lives. Pick a folder, press a button, and in a few minutes AutoCollage presents you with a unique memento to print or email to your family and friends.

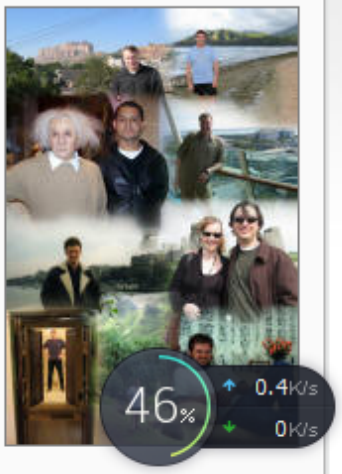
AutoCollage Touch 2009 Support
Some customers acquiring new computers with Windows 7 will have a welcome surprise: they will receive a pre-installed and pre-registered copy of Microsoft Research AutoCollage Touch 2009.

v1.1 Available
Cambridge Innovation Development is proud to make available AutoCollage version 1.1, which includes several enhancements based on customer's feature requests, including selection of the top ranked image, removal of images, rich Windows Live Photo Gallery integration and arbitrary output sizes! Download it now and start creating your own collages in just a few clicks.

Share your collages
Have you created a collage you really like? Why not share it with the world!

Maximise your photo collection
AutoCollage allows you to create beautiful collages of your favorite pictures in a few clicks of a mouse. See the program in action, or meet the team behind it.

Sample images.



Please address feedback to us.

Our New Collage Method








Microsoft AutoCollage | Circle Packing | Our Semantic-absent Version | Our Approach

Lingjie Liu, Hongjie Zhang, Guanmei Jing, **Yanwen Guo**, Zhonggui Chen, and Wenping Wang,Correlation-preserving Photo Collage, IEEE Transactions on Visualization & Computer Graphics, In Peer Review.