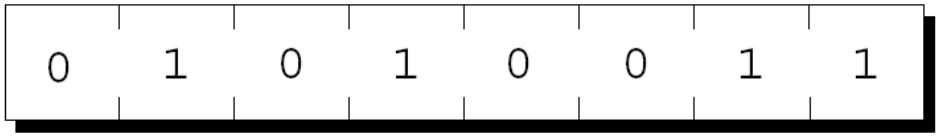
第 11 章

指针

指针变量

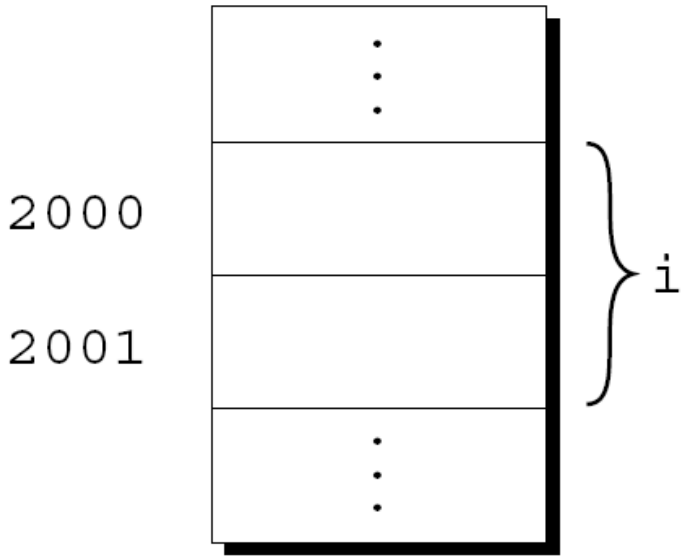
- 现代计算机将内存分割为字节（***bytes***）：



- 每个字节都有唯一的地址（***address*** ）。

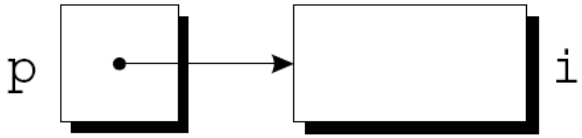
指针变量

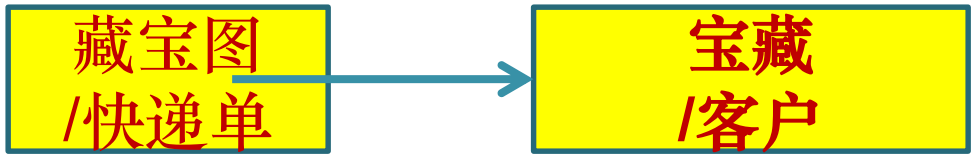
- 变量占有一个或多个字节内存。
- 将第一个字节的地址称为变量的地址。
- 在下图中，变量 i 的地址是 2000：



指针变量

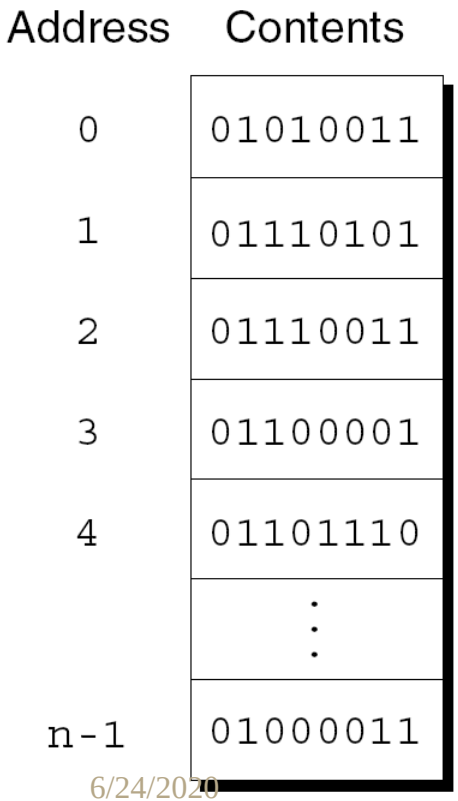
- 存储内存地址
- 指针变量p存储变量i的地址，通常说“p指向i”。
- 采用图形方式可以表达为：





指针变量

- 地址：内存编号
  - 设内存有*n*个字节，内存地址取值范围可想像成0到*n*-1之间的整数（编号）
  - 便于处理，定长
    - 如，学号



声明指针变量

- 格式：

```
type * pt_name;//指向类型为type的对象
eg. int *pi; char *pc;
```

- 可以和其它同类型变量一起声明

```
int i, j, a[10], b[20], *p, *q;
```

- 只能指向声明类型的对象：

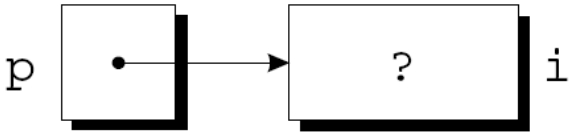
```
int *p;      /* only to integers */
double *q;   /* only to doubles  */
char *r;     /* only to characters */
```

- 指针声明只是分配空间，并未指向任何对象，
  - 拿一个快递单
- 使用前需（　　）
- 填写/指向，值从何而来？

指针指向——取址运算符&

- 让指针指向程序中的对象
  - 将对象地址填入指针变量

```
int i, *p;
```



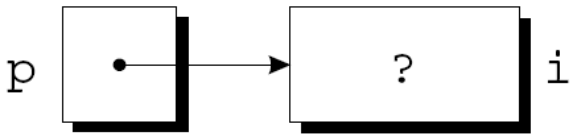
让p指向i

p = &i; // &：取址运算符

初始化指针变量

- 取变量地址进行赋值：

```
int i, *p;
…
p = &i;
```



- 声明的同时初始化：

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

- 甚至合并变量和指针变量的声明：

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

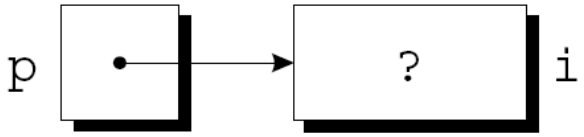
使用指针——间接寻址运算符*

- 指针前加*，使用指针
  - *p表示p指向的对象
  - p = &i后，*p是i的别名（*alias*）
- p等价于&i，*p等价于i
  - *p 拥有和 i相同的值
  - 改变 *p 的值，同时也会改变i的值
- 先找到p，再访问i——间接寻址

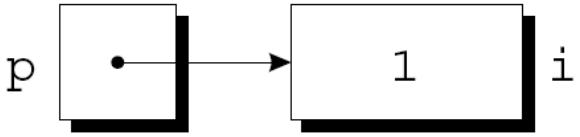
间接寻址运算符

- *p和i的等价关系示意图

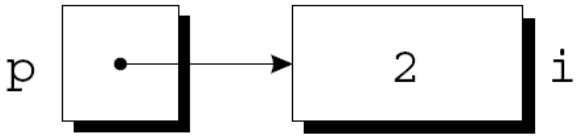
```
p = &i;
```




```
i = 1;
```




```
printf("%d\n", i);     /* prints 1 */
printf("%d\n", *p);    /* prints 1 */
*p = 2;
```




```
printf("%d\n", i);     /* prints 2 */
printf("%d\n", *p);    /* prints 2 */
```

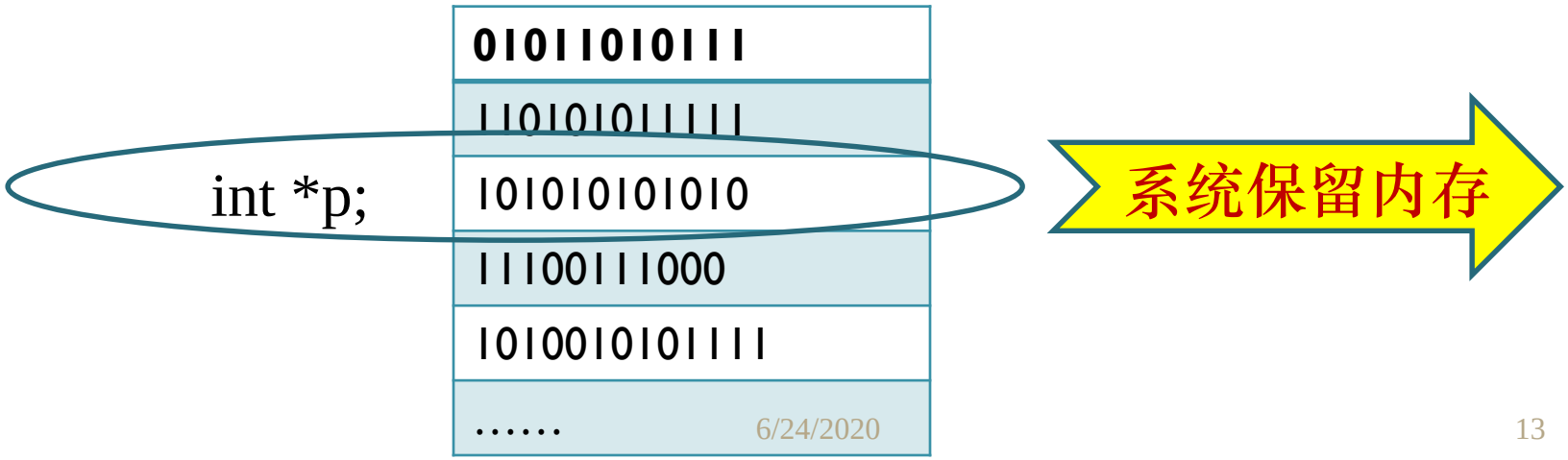
间接寻址运算符

- 使用未初始化指针变量，未定义行为

```
int *p;
printf("%d", *p);   /* WRONG */
```

- 对*p直接赋值更加危险：

```
int *p;
*p = 1;    /* WRONG */
```



指针赋值

- 类型相同才能赋值

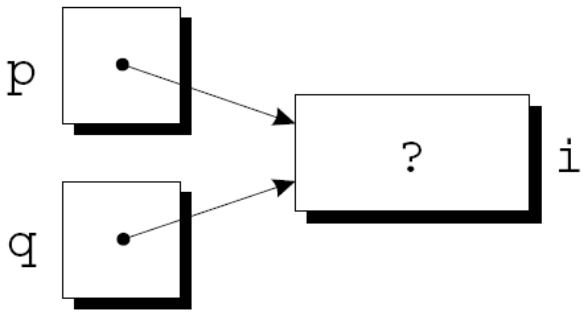
```
int i, j, *p, *q;
```

- 取址赋值：

```
p = &i;
```

- 指针赋值：

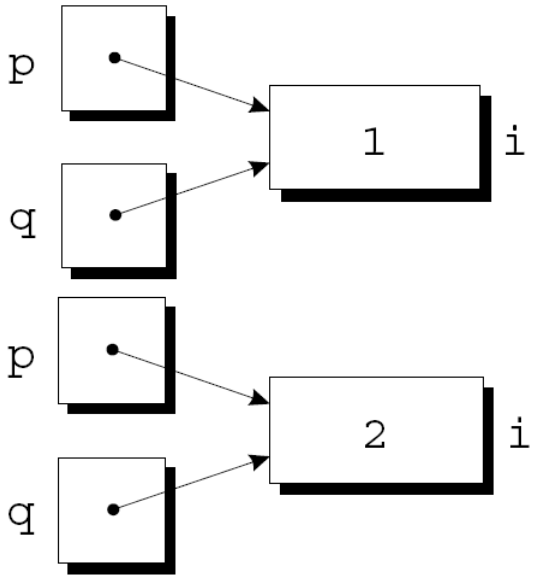
```
q = p; //指针q与p指向相同的位置：
```



指针赋值

- 可通过p或q改变i的值：

```
*p = 1;
```




```
*q = 2;
```


- 任意数量的指针变量都可以指向同一个对象

指针赋值

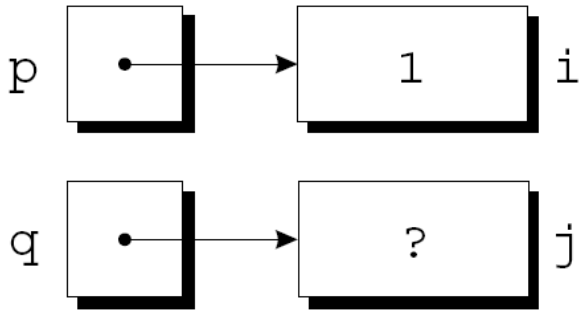
- 注意不要混淆概念：

  q = p 和*q = *p;

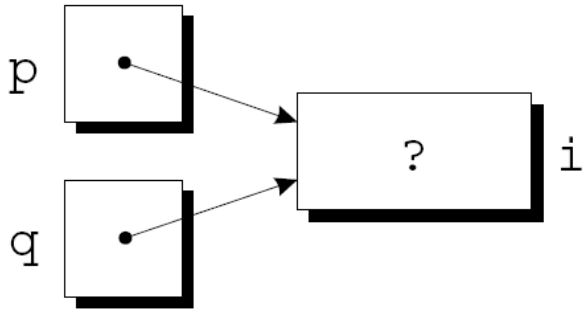
- q = p：指针赋值语句
- *q = *p：指针指向内存赋值
- 下页中的图示给出了第二条语句的实际效果

指针赋值

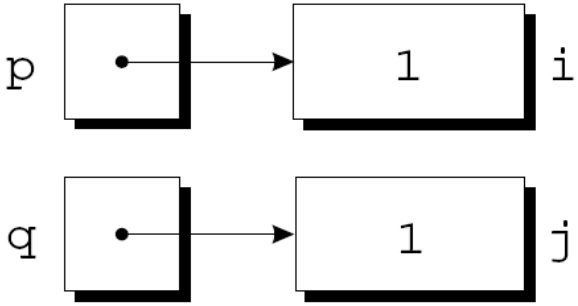
```
p = &i;
q = &j;
i = 1;
```



q = p




```
*q = *p;
```



指针作为参数

- 9.3节decompose函数，试图通过形式参数修改实际参数

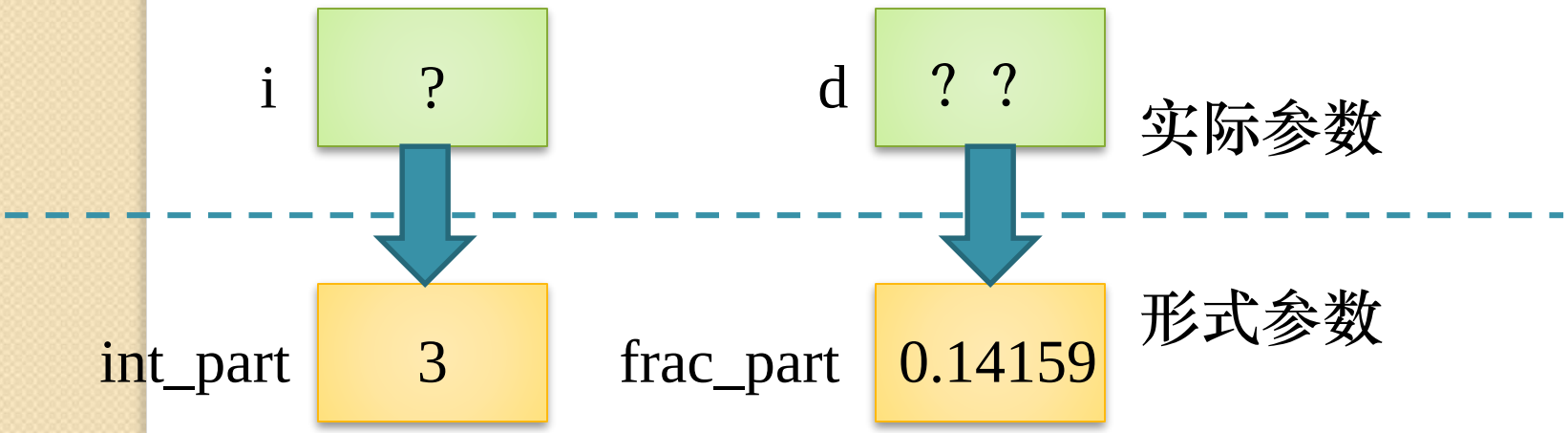
```
void decompose(double x, long
int_part, double frac_part)
{
   int_part = (long) x;
   frac_part = x - int_part;
}
```

- 调用函数：

```
decompose(3.14159, i, d);
```

指针作为参数

- 普通变量参数—》值传递，单向性



形参是实参副本，形参改变不影响实参

指针作为参数

- 期望decompose函数同时得到两个结果（整数、小数部分）怎么办？
- 返回值？

```
?? decompose(double x, long
int_part, double frac_part)
{
    int_part = (long) x;
    frac_part = x - int_part;
    return int_part;
    return frac_part;
```

函数只能返回一个数据（基本类型、聚合类型）
but why?

指针作为参数

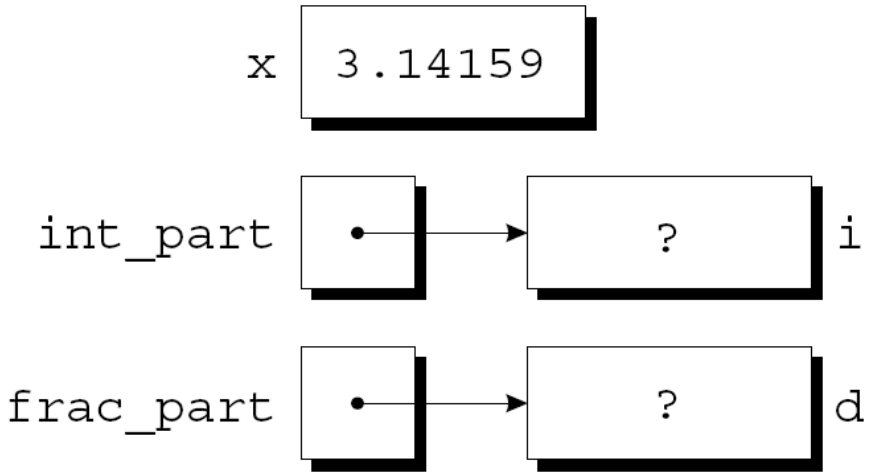
- 指针参数：传地址，类似数组
- Decompose函数新定义：

```
void decompose(double x,
long *int_part, double
*frac_part)
{
   *int_part = (long) x;
   *frac_part = x - *int_part;
}
```

通过地址函数访问实参原件

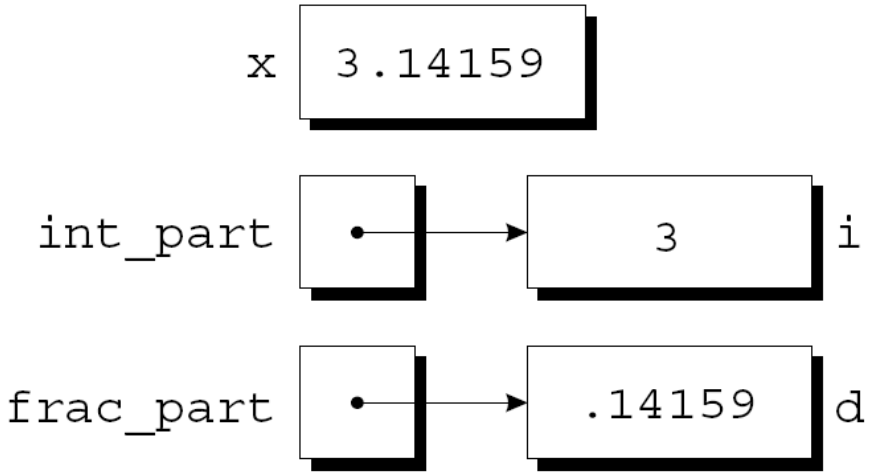
指针作为参数

- 调用decompose函数：
  - `decompose(3.14159, &i, &d);`
- 调用结果：
  - `int_part = &i`，指向i（别名）
  - `frac_part = &d`，指向 d：



指针作为参数

```
*int_part = (long) x;
*frac_part = x - *int_part;
```



函数与外界（主调函数）通信

- 普通参数传递+返回值：单通
  - 外—》内：单向参数传递
  - 内—》外：单向返回单个值（对象）
- 外部变量：双通
  - 读/写：双向，共享（混乱）
- 以指针（地址）或数组为参数：双通
  - 读/写：双向访问实参原件

函数与外界通信示例

```
int m=3,n;
int f1(int a, int b){……}
int f2(int *p, int *q){……}
void f3(void){n=2*m;}
int main(void){
    int x,y,z;
    scanf（"%d%d%d",&x,&y,&m）;
    z = f1(x,y);
    f3();
    f2(&x,&y);
}
```

由内向外，return

由外向内，实形参值传递

外部变量，双向

地址参数，双向

指针作为参数

- scanf(“%d”, a)读取键盘输入，通过参数传递回主调函数（由内向外）
- 所以参数必须是地址

```
int i;
…
scanf("%d", &i);
```

- 或指针：

```
int i, *p;
…
p = &i;
scanf("%d", p);
scanf("%d", &p);/**WRONG**/
```

指针作为参数

- 以指针为函数参数，如果参数传递错误，可能导致灾难性的后果。
- 例如函数调用时，省略了&：

```
decompose(3.14159, i, d);
```

- 将试图修改以i、d值为地址的内存空间。类似，scanf漏掉&

程序：找出数组中的最大和最小元素

- 函数max_min找出数组中最大和最小元素。
  - 结果：两个，返回不可行
  - 外部变量不推荐使用
- 原型：

```
void max_min(int a[], int n,
int *max, int *min);
```

- 调用示例：

```
int big,small;
max_min(b, N, &big, &small);
```

- 将最大元素，通过max存储到big。
- 将最小元素，通过min存储small。

程序：找出数组中的最大和最小元素

- 首先读入10个数到一个数组中，
- 然后将该数组传递给max_min函数，最后现实出结果：

```
Enter 10 numbers: 34 82 49
102 7 94 23 11 50 31
Largest: 102
Smallest: 7
```

```
#define N 10
void max_min(int a[], int n, int
  *max, int *min);
int main(void)
{
   int b[N], i, big, small;
   printf("Enter %d numbers: ", N);
   for (i = 0; i < N; i++)
     scanf("%d", &b[i]);//构造数列

   max_min(b, N, &big, &small);

   printf("Largest: %d\nSmallest:
  %d\n", big, small);
   return 0;
}
```

```
void max_min(int a[], int n,
  int *max, int *min)
{
   int i;

   *max = *min = a[0];
   for (i = 1; i < n; i++)
   {
      if (a[i] > *max)
         *max = a[i];
      else if (a[i] < *min)
         *min = a[i];
   }
}
```

用 const 保护参数

- x：宝藏
  - f(x)：传宝藏副本（图片）
  - f(&x)：传藏宝图（地址），通常意味着x的值将被修改
- 指针参数更高效：
  - 指针4byte，无论实际数据（如前x）多大
  - 传递占大量内存空间参数时，值传递会造成时空浪费
- eg,网上一段视频很有趣，分享给好友：
  - a.下载下来传给好友
  - b.直接传链接给好友
- 所以有时f仅需检查（或使用，不修改）x值，也使用指针，但需保护x

温故而知新——指针

- 先声明，再指向，后使用
  - `type *p, i;`
  - `p = &i;`
  - `*p = 10;`
- 本质：地址
  - p，&i，&a[i]，a实质一样，都是地址
- *p：p指向的对象
  - *p为i别名
  - *：间接寻址运算符
    - 先访问p，再通过p访问对象（i）

温故而知新——指针

- 指针做参数
  - 声明：int max(int *a, int *b)
  - 调用：max（&i,&j)
  - 形参指针指向实参，*a、*b就是i和j，函数直接修改i,j
  - 依然单向值传递

```
int max(int *a, int *b)
{a=b=NULL;}
max(&i, &j);//i,j不受影响
```

- 返回类型为指针

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

调用：p=max(&i,&j);

  - 返回有效地址：由函数外传入的地址
  - 返回指针类型的形参

用 const 保护参数

- 使用const关键字确保函数不会修改指针参数所指向的对象。
- const 形式参数：

```
void f(const int *p)
{
   *p = 0;    /*** WRONG ***/
}
```

- 宝藏冻结，可观不可写

指针作为返回值

- 函数返回指针：

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

- max函数调用：

```
int *p, i, j;
…
p = max(&i, &j);
```

- 调用结束后，p指向i或j

返回指针：返回内存地址

指针作为返回值

- 返回指针：返回内存地址，须有效地址
  - 函数返回后仍有效，不能函数局部变量地址
  - 函数外部可见，内部可访问—传入的指针类型的参数
- 返回某个形参，要求形参类型为地址（指针）
  - 形参就是实参地址，返回实参地址
  - 上例返回实参i或j的地址
- 不要返回指向自动局部变量的指针

```
int *f(void)
{
   int i;
   …
   return &i;
}
```

```
int *max(int a, int b){
   if (a > b) return &a;
   el
}
```

eg, 分享有趣链接，结果链接失效 虚情假意

- what happen? ?
- 函数也可返回指向外部变量或static局部变量的指针

指针作为返回值

- 指针可指向数组元素
- &a[i]指向数组a中元素i的地址（指针）
  - *p=a，p+i<—>&a[i]
- 数组作函数参数时，可返回指向数组元素的指针，有时会有特殊用途
  - eg，数组a有n个元素，函数返回指向中间元素的指针：

```
int *find_middle(int a[], int n)
{
   return &a[n/2];
}
```

程序练习——函数实现两数交换

- 值传递方式
  - 函数处理传入数据的副本
  - 副本之间交换不影响原来两数
- 指针或地址传递
  - 函数通过地址找到原来两数
  - 交换原件

```
#include<stdio.h>
void swap(int *, int *);
int main(void)
{
    int a=5,b=9;
    swap(&a,&b);
    printf("a=%d\tb=%d\n",a,b);
    return 0;
}
voi
{
}
```

```
void swap(int *p, int *q)
{
    int *t;
    t = p;
    p = q;
    q = t;
}//p, q指向交换
```

*p,*q实质就是a，b