

9.1-1 取地址运算

运算符 &

- `scanf("%d", &i);`里的&
- 获得变量的地址，它的操作数必须是变量
 - `int i; printf("%x",&i);`
- 地址的大小是否与int相同取决于编译器
 - `int i; printf("%p",&i);`

&不能取的地址

- &不能对没有地址的东西取地址
 - $\&(a+b)?$
 - $\&(a++)?$
 - $\&(++a)?$

试试这些&

- 变量的地址
- 相邻的变量的地址
- &的结果的sizeof
- 数组的地址
- 数组单元的地址
- 相邻的数组单元的地址

9.1-2 指针

scanf

- 如果能够将取得的变量的地址传递给一个函数，能否通过这个地址在那个函数内访问这个变量？
- `scanf("%d", &i);`
- `scanf()`的原型应该是怎样的？ 我们需要一个参数能保存别的变量的地址， 如何表达能够保存地址的变量？

指针

- 就是保存地址的变量

```
int i;
```

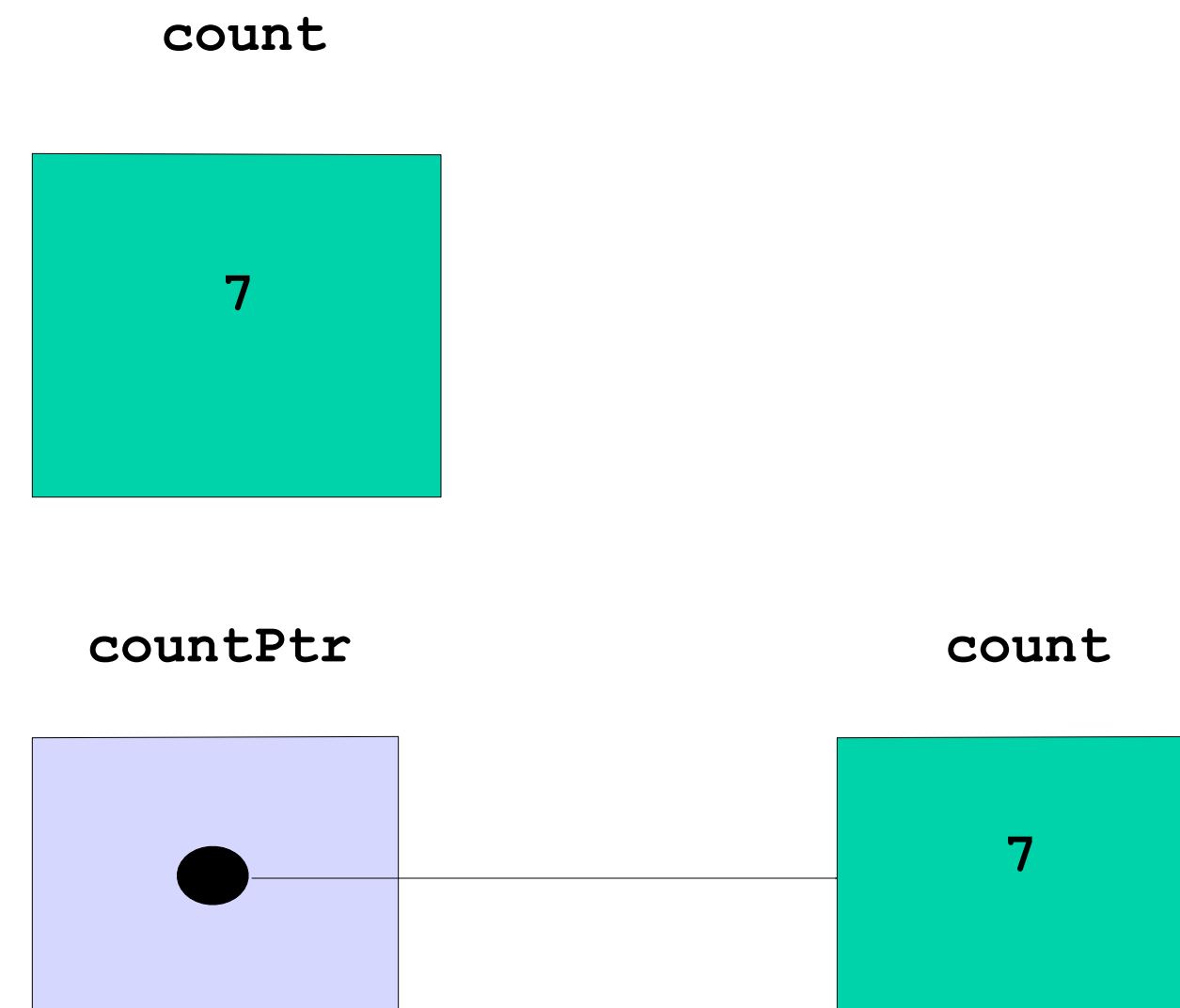
```
int* p = &i;
```

```
int* p,q;
```

```
int *p,q;
```

指针变量

- 变量的值是内存的地址
- 普通变量的值是实际的值
- 指针变量的值是具有实际值的变量的地址



作为参数的指针

- `void f(int *p);`
- 在被调用的时候得到了某个变量的地址：
 - `int i=0; f(&i);`
- 在函数里面可以通过这个指针访问外面的这个i

访问那个地址上的变量*

- *是一个单目运算符，用来访问指针的值所表示的地址上的变量
- 可以做右值也可以做左值
 - `int k = *p;`
 - `*p = k+1;`

* 左值之所以叫左值

- 是因为出现在赋值号左边的不是变量，而是值，是表达式计算的结果：
 - $a[0] = 2;$
 - $*p = 3;$
- 是特殊的值，所以叫做左值

指针的运算符&*

- 互相反作用
 - $*\&y\text{ptr} \rightarrow *(\&y\text{ptr}) \rightarrow *(y\text{ptr的地址}) \rightarrow \text{得到那个地址上的变量} \rightarrow y\text{ptr}$
 - $\&*y\text{ptr} \rightarrow \&(*y\text{ptr}) \rightarrow \&(y) \rightarrow \text{得到}y\text{的地址, 也就是}y\text{ptr} \rightarrow y\text{ptr}$

传入地址

- 为什么
 - `int i; scanf("%d", i);`
- 编译没有报错?

指针应用场景一

- 交换两个变量的值

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

指针应用场景二

- 函数返回多个值，某些值就只能通过指针返回
- 传入的参数实际上是需要保存带回的结果的变量

指针应用场景二b

- 函数返回运算的状态，结果通过指针返回
- 常用的套路是让函数返回特殊的不属于有效范围内的值来表示出错：
 - -1或0（在文件操作会看到大量的例子）
- 但是当任何数值都是有效的可能结果时，就得分开返回了
- 后续的语言（C++,Java）采用了异常机制来解决这个问题

指针最常见的错误

- 定义了指针变量，还没有指向任何变量，就开始使用指针

9.1-3 指针与数组

传入函数的数组成了什么？

```
int isPrime(int x, int knownPrimes[], int numberOfKnownPrimes)
{
    int ret = 1;
    int i;
    for ( i=0; i<numberOfKnownPrimes; i++ ) {
        if ( x % knownPrimes[i] ==0 ) {
            ret = 0;
            break;
        }
    }
    return ret;
}
```

- 函数参数表中的数组实际上是指针
 - `sizeof(a) == sizeof(int*)`
 - 但是可以用数组的运算符[]进行运算

数组参数

- 以下四种函数原型是等价的：
 - `int sum(int *ar, int n);`
 - `int sum(int *, int);`
 - `int sum(int ar[], int n);`
 - `int sum(int [], int);`

数组变量是特殊的指针

- 数组变量本身表达地址，所以
 - `int a[10]; int *p=a; // 无需用&取地址`
 - 但是数组的单元表达的是变量，需要用&取地址
 - `a == &a[0]`
- []运算符可以对数组做，也可以对指针做：
 - `p[0] <==> a[0]`
- *运算符可以对指针做，也可以对数组做：
 - `*a = 25;`
- 数组变量是const的指针，所以不能被赋值
 - `int a[] <==> int * const a=....`

* 9.1-4 指针与const

C99 ONLY!

指针与const

指针 -- 可以是const

0xaffefado

值 -- 可以是const

54



```
graph LR; A[0xaffefado] --> B[54];
```

The diagram illustrates a pointer variable pointing to a value. A green box on the left contains the hexadecimal address '0xaffefado'. A curved arrow originates from the right side of this box and points to a green box on the right containing the integer value '54'. Above the left box is the text '指针 -- 可以是const' (Pointer -- can be const), and above the right box is the text '值 -- 可以是const' (Value -- can be const).

指针是const

- 表示一旦得到了某个变量的地址，不能再指向其他变量
 - `int * const q = &i; // q 是 const`
 - `*q = 26; // OK`
 - `q++; // ERROR`

所指是const

- 表示不能通过这个指针去修改那个变量（并不能使得那个变量成为const）
 - `const int *p = &i;`
 - `*p = 26; // ERROR! (*p) 是 const`
 - `i = 26; //OK`
 - `p = &j; //OK`

这些是啥意思？

```
int i;  
const int* p1 = &i;  
int const* p2 = &i;  
int *const p3 = &i;
```

判断哪个被const了的标志是const在*的前面还是后面

转换

- 总是可以把一个非const的值转换成const的

```
void f(const int* x);  
int a = 15;  
f(&a); // ok  
const int b = a;
```

```
f(&b); // ok
```

```
b = a + 1; // Error!
```

- 当要传递的参数的类型比地址大的时候，这是常用的手段：既能用比较少的字节数传递值给参数，又能避免函数对外面的变量的修改

const数组

- `const int a[] = {1,2,3,4,5,6};`
- 数组变量已经是const的指针了，这里的const表明数组的每个单元都是const int
- 所以必须通过初始化进行赋值

保护数组值

- 因为把数组传入函数时传递的是地址，所以那个函数内部可以修改数组的值
- 为了保护数组不被函数破坏，可以设置参数为const
- `int sum(const int a[], int length);`