指针

2020年7月16日 13:50

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);

取地址用%p，不用%d，地址和整数可能不同，这和编译器，架构等有关。

```c
#include <stdio.h>

int main(void)
{
    int i = 0;
    printf("0x%x\n", &i);
    printf("%p\n", &i);

    return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
    int i = 0;
    int p;
    p = &i;
    printf("0x%x\n", &i);
    printf("%p\n", &i);

    return 0;
}
```

类型转换的问题

```
1 warning generated.
0xbff62d6c
0xbff62d6c
[Finished in 0.1s]
```

```
Users/wengkai/cc/add.c:7:4: warning:
        p = &i;
          ^ ~~
Users/wengkai/cc/add.c:8:19: warning
        printf("0x%x\n", &i);
```

&不能取的地址

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

只能对一个变量取地址，不是变量不能取地址

```c
#include <stdio.h>

int main(void)
{
    int i = 0;
    int p;
    p = (int)&i;
    printf("0x%x\n", p);
    printf("%p\n", &i);

    return 0;
}
```

强制类型转换

```
0xbfff5d6c
0xbfff5d6c
[Finished in 0.1s]
```

```c
#include <stdio.h>

int main(void)
{
    int i = 0;
    int p;
    p = (int)&i;
    printf("0x%x\n", p);
    printf("%p\n", &i);
    printf("%lu\n", sizeof(int));
    printf("%lu\n", sizeof(&i));

    return 0;
}
```

64位架构

```
0x5dd1ad28
0x7fff5dd1ad28
4
8
[Finished in 0.1s]
```

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

自顶向下存放，是放在堆栈中的！

```c
#include <stdio.h>

int main(void)
{
    int i = 0;
    int p;
    p = (int)&i;
    p = (int)&(i+p);

    printf("0x%x\n", p);
    printf("%p\n", &i);
    printf("%lu\n", sizeof(int));
    printf("%lu\n", sizeof(&i));

    return 0;
}
```

```
/Users/wengkai/cc/add.c:8:11: error: ca
        p = (int)&(i+p);
                 ^ ~~~
1 error generated.
[Finished in 0.1s with exit code 1]
```

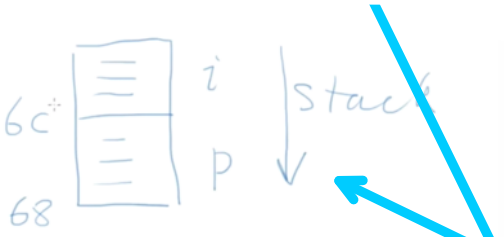
32位架构

```c
#include <stdio.h>

int main(void)
{
    int i = 0;
    int p;
    p = (int)&i;
    printf("0x%x\n", p);
    printf("%p\n", &i);
    printf("%lu\n", sizeof(int));
    printf("%lu\n", sizeof(&i));

    return 0;
}
```

```
0xbffb6d6c
0xbffb6d6c
4
4
```

```c
#include <stdio.h>

int main(void)
{
    int i = 0;
    int p;

    printf("%p\n", &i);
    printf("%p\n", &p);

    return 0;
}
```

```c
#include <stdio.h>

int main(void)
{
    int a[10];

    printf("%p\n", &a);
    printf("%p\n", a);
    printf("%p\n", &a[0]);
    printf("%p\n", &a[1]);

    return 0;
}
```



4个字节，相邻单元

```
0xbff81d6c
0xbff81d68
[Finished in 0.2s]
```

相邻地址差距永远是4

```
0xbff8dd44
0xbff8dd44
0xbff8dd44
0xbff8dd48
```

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;
```

p 是一个指针，它指向 int。现在把 i 的地址交给p

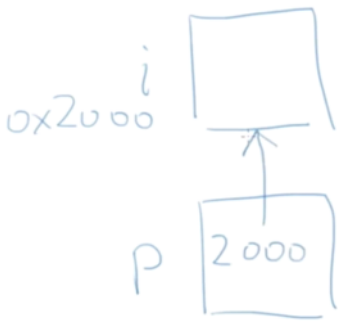
即 p 的值是 i 的地址。p，i 都是变量！

p 都是指针，指向int，p 的意思一样。

q 只是一个int类型的变量。

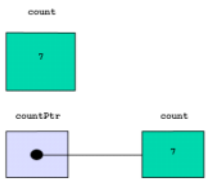
把*加给了p，*p 是一个int！！

不是把*加给了int，p不能说是int*这种类型！



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



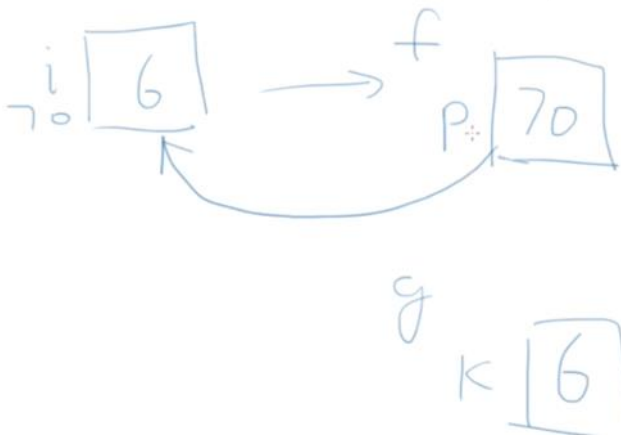
作为参数的指针

- 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


```c
#include <stdio.h>

void f(int *p);

int main(void)
{
    int i = 6;
    printf("&i=%p\n", &i);
    f(&i);

    return 0;
}

void f(int *p)
{
    printf(" p=%p\n", p);
}
```

不知道 i，但是有地址了。

```
&i=0xbff17d70
 p=0xbff17d70
[Finished in 0.7s]
```




```c
void f(int *p);
void g(int k);

int main(void)
{
    int i = 6;
    printf("&i=%p\n", &i);
    f(&i);
    g(i);

    return 0;
}

void f(int *p)
{
    printf(" p=%p\n", p);
}

void g(int k)
{
    printf("k=%d\n", k);
}
```

g 得到的只是i 的值

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;

f 函数调用后 i 改了，发生的是值传递！
传的是地址，就可以修改 i 的值

```c
void f(int *p);
void g(int k);

int main(void)
{
    int i = 6;
    printf("&i=%p\n", &i);
    f(&i);
    g(i);

    return 0;
}

void f(int *p)
{
    printf(" p=%p\n", p);
    printf("*p=%d\n", *p);
}

void g(int k)
{
    printf("k=%d\n", k);
}
```

```
&i=0xbff51d70
 p=0xbff51d70
*p=6
k=6
[Finished in 0.2s]
```

```c
int main(void)
{
    int i = 6;
    printf("&i=%p\n", &i);
    f(&i);
    g(i);

    return 0;
}

void f(int *p)
{
    printf(" p=%p\n", p);
    printf("*p=%d\n", *p);
    *p = 26;
}

void g(int k)
{
    printf("k=%d\n", k);
}
```

```
&i=0xbffbcd70
 p=0xbffbcd70
*p=6
k=26
[Finished in 0.2s]
```

* 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

*p是对应地址单元的值
a[0]是数组a下标为0的地方的值

地址值 &i 被传进了函数 f(int *p)，这里，仍然是值的传递，因为传进来的是地址，所以，通过这个地址，在函数内部，可以以这种方式，访问到外面的 i 变量。

因为 p 的值，就是 i 的地址。 *p 就代表了 i 。

当我们做 *p = 26 这个运算的时候，我们实际做的事情，是对 i 做的。

指针的运算符&*

- 互相反作用
  - *&yptr -> * (&yptr) -> * (yptr的地址)-> 得到那个地址上的变量 -> yptr
  - &*yptr -> &(*yptr) -> &(y) -> 得到y的地址，也就是yptr -> yptr

传入地址

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

32位架构中，地址和整数一样大。编译器以为你传入了地址。
编译不会报错，运行一定会出错。

scanf把读进来的数字写到了不该写的地方进去，eg.写到了6的地方去，这个地方很小，不能写！，有很重要的东西!

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;
}
```

```
3  void swap(int *pa, int *pb);
4
5  int main(void)
6  {
7      int a = 5;
8      int b = 6;
9      swap(&a, &b);
10     printf("a=%d,b=%d\n", a,b);
11
12     return 0;
13 }
14
15 void swap(int *pa, int *pb)
16 {
17     int t = *pa;
18     *pa = *pb;
19     *pb = t;
20 }
21
```

```
a=6,b=5
[Finished in 0.2s]
```

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

```
3  void minmax(int a[], int len, int *max, int *min);
4
5  int main(void)
6  {
7      int a[] = {1,2,3,4,5,6,7,8,9,12,13,14,16,17,21,23,55,};
8      int min,max;
9      minmax(a, sizeof(a)/sizeof(a[0]), &min, &max);
10     printf("min=%d,max=%d\n", min, max);
11
12     return 0;
13 }
14
15 void minmax(int a[], int len, int *min, int *max)
16 {
17     int i;
18     *min = *max=a[0];
19     for ( i=1; i<len; i++ ) {
20         if ( a[i] < *min ) {
21             *min = a[i];
22         }
23         if ( a[i] > *max ) {
24             *max = a[i];
25         }
26     }
27 }
```

```
min=1,max=55
[Finished in 0.2s]
```

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

```
#include <stdio.h>

/**
    @return 如果除法成功，返回1；否则返回0
*/
int divide(int a, int b, int *result);

int main(void)
{
    int a=5;
    int b=2;
    int c;
    if ( divide(a,b,&c) ) {
        printf("%d/%d=%d\n", a, b, c);
    }
    return 0;
}

int divide(int a, int b, int *result)
{
    int ret = 1;
    if ( b == 0 ) ret = 0;
    else {
        *result = a/b;
    }
    return ret;
}
```

除数为0，返回0，c不改变。反之c改变

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

```
int i = 6;
int *p;
int k;
k = 12;
*p = 12;
printf("&i=%p\n", &i);
f(&i);
g(i);
```

```
int main(void)
{
    int i = 6;
    int *p=0;
    int k;
    k = 12;
    *p = 12;

    printf("&i=%p\n", &i);
    f(&i);
    g(i);

    return 0;
}

void f(int *p)
{
    printf(" p=%p\n", p);
    printf("*p=%d\n", *p);
    *p = 26;
}
```

```
bash: line 1: 11443 Segmentation fault
```

所有本地变量都不会有默认初始值，没有赋值，本地变量里面什么都没有，它可能是乱七八糟的东西。如果这个时候把它当做地址，它可能指向了乱七八糟的地方。
往乱七八糟的地方写了东西？？程序可能崩溃！
任何一个地址变量没得到实际变量地址之前不能通过*访问变量数据！！

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

是一个指针！！！看上去像数组

```
void minmax(int a[], int len, int *max, int *min);

int main(void)
{
    int a[] = {1,2,3,4,5,6,7,8,9,12,13,14,16,17,21,23,55,};
    int min,max;
    printf("main sizeof(a)=%lu\n", sizeof(a));
    minmax(a, sizeof(a)/sizeof(a[0]), &min, &max);
    printf("min=%d,max=%d\n", min, max);

    return 0;
}

void minmax(int a[], int len, int *min, int *max)
{
    int i;
    printf("minmax sizeof(a)=%lu\n", sizeof(a));
    *min = *max=a[0];
    for ( i=1; i<len; i++ ) {
        if ( a[i] < *min ) {
            *min = a[i];
        }
        if ( a[i] > *max ) {
```

```
1 warning generated.
main sizeof(a)=68
minmax sizeof(a)=4
min=1,max=55
```

4? 指针的大小！

编译正确！！运行正确！！

函数体内不用sizeof！！

```
sizeof on array function parameter will return size of 'int *' instead of 'int []'
```

```
void minmax(int *a, int len, int *max, int *min);

int main(void)
{
    int a[] = {1,2,3,4,5,6,7,8,9,12,13,14,16,17,21,23
    int min,max
    printf("mai sizeof(a)=%lu\n", sizeof(a));
    printf("mai a=%p\n",a);
    minmax(a, sizeof(a)/sizeof(a[0]), &min, &max);
    printf("a[0]=%d\n", a[0]);
    printf("min=%d,max=%d\n", min, max);

    return 0;
}

void minmax(int *a, int len, int *min, int *max)
{
    int i;
    printf("minmax sizeof(a)=%lu\n", sizeof(a));
    printf("minmax a=%p\n",a);
    a[0]=1000;
    *min = *max=a[0];
    for ( i=1; i<len; i++ ) {
```

```
    int a[] = {1,2,3,4,5,6,7,8,9,12,13,14,16,17,21,23,55,};
    int min,max;
    printf("main sizeof(a)=%lu\n", sizeof(a));
    printf("main a=%p\n",a);
    minmax(a, sizeof(a)/sizeof(a[0]), &min, &max);
    printf("min=%d,max=%d\n", min, max);

    return 0;
}

void minmax(int a[], int len, int *min, int *max)
{
    int i;
    printf("minmax sizeof(a)=%lu\n", sizeof(a));
    printf("minmax a=%p\n",a);
    *min = *max=a[0];
    for ( i=1; i<len; i++ ) {
        if ( a[i] < *min ) {
            *min = a[i];
```

```
main a=0xbff67d10
minmax sizeof(a)=4
minmax a=0xbff67d10
```

数组传的是一摸一样的数组

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);

```
7     int a[] = {1,2,3,4,5,6,7,8,9,12,13,14,16,17,21,23,55,};
8     int min,max;
9     printf("main sizeof(a)=%lu\n", sizeof(a));
10    printf("main a=%p\n",a);
11    minmax(a, sizeof(a)/sizeof(a[0]), &min, &max);
12    printf("a[0]=%d\n", a[0]);
13    printf("min=%d,max=%d\n", min, max);
14
15    return 0;
16 }
17
18 void minmax(int a[], int len, int *min, int *max)
19 {
20    int i;
21    printf("minmax sizeof(a)=%lu\n", sizeof(a));
22    printf("minmax a=%p\n",a);
23    a[0]=1000;
24    *min = *max=a[0];
25    for ( i=1; i<len; i++ ) {
```

```
minmax sizeof(a)=4
minmax a=0xbffbcd10
a[0]=1000
min=2,max=1000
[Finished in 0.1s]
```

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=....

```
7     int a[] = {1,2,3,4,5,6,7,8,9,12,13,14,16,17,21,2
8     int min,max;
9     printf("main sizeof(a)=%lu\n", sizeof(a));
10    printf("main a=%p\n",a);
11    minmax(a, sizeof(a)/sizeof(a[0]), &min, &max);
12    printf("a[0]=%d\n", a[0]);
13    printf("min=%d,max=%d\n", min, max);
14    int *p = &min;
15    printf("*p=%d\n", *p);
16    printf("p[0]=%d\n", p[0]);
17
18    return 0;
19 }
20
21 void minmax(int *a, int len, int *min, int *max)
22 {
23    int i;
24    printf("minmax sizeof(a)=%lu\n", sizeof(a));
25    printf("minmax a=%p\n",a);
```

```
a[0]=1000
min=2,max=1000
*p=2
p[0]=2
```

p[0]：把p指向的地方当做一个数组，其实它不是数组，是一个变量，但是可以当成一个int min[1].
对普通变量不能这样，指针可以这样。

```
7     int a[] = {1,2,3,4,5,6,7,8,9,12,13,14,16,17,21,
8     int min,max;
9     printf("main sizeof(a)=%lu\n", sizeof(a));
10    printf("main a=%p\n",a);
11    minmax(a, sizeof(a)/sizeof(a[0]), &min, &max);
12    printf("a[0]=%d\n", a[0]);
13    printf("min=%d,max=%d\n", min, max);
14    int *p = &min;
15    printf("*p=%d\n", *p);
16    printf("p[0]=%d\n", p[0]);
17    printf("*a=%d\n", *a);
18    return 0;
19 }
20
21 void minmax(int *a, int len, int *min, int *max)
22 {
23    int i;
24    printf("minmax sizeof(a)=%lu\n", sizeof(a));
25    printf("minmax a=%p\n",a);
```

```
min=2,max=1000
*p=2
p[0]=2
*a=1000
```

```
int b[] --> int * const b;
int *q = a;
```

b是常数，不能被改变！！初始化创建后就不能被改变！！

指针与const

C99 ONLY!

指针是const

所指是const

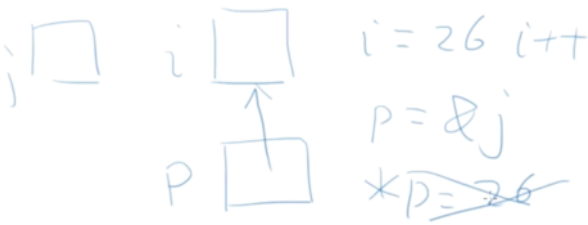
- 表示不能通过这个指针去修改那个变量（并不能使得那个变量成为const）
- const int *p = &i;

- 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

q指向i这个事实不能被改变！！
q不能被指向其他
q指向的int i不是const！！

- *p = 26; // ERROR! (*p) 是 const
- i = 26; //OK
- p = &j; //OK

i ,p都可以修改，不可以通过p修改 i



这些是啥意思？

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

在*前面，指的东西不能被修改。1,2,一样 这P不改
3：指针不能被修改 P不改

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

给一个指针，我保证在函数内部不会改变指针所指的值

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

1+1=2?

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);

- 给一个指针加1表示要让指针指向下一个变量

  int a[10];

  int *p = a;

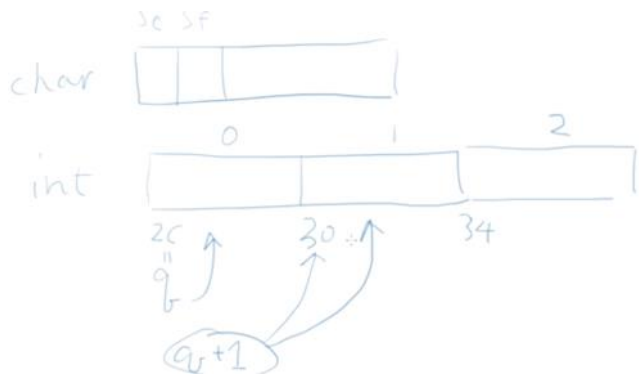
  *(p+1) —> a[1]
- 如果指针不是指向一片连续分配的空间，如数组，则这种运算没有意义

*(p+n) <-> ac[n]

```
3  int main(void)
4  {
5      char ac[] = {0,1,2,3,4,5,6,7,8,9,};
6      char *p = ac;
7      printf("p  =%p\n", p);
8      printf("p+1=%p\n", p+1);
9      int ai[] = {0,1,2,3,4,5,6,7,8,9,};
10     int *q = ai;
11     printf("q  =%p\n", q);
12     printf("q+1=%p\n", q+1);
13     return 0;
14 }
```

```
p  =0xbffbad2b
p+1=0xbffbad2c
q  =0xbffbad2c
q+1=0xbffbad30
```

sizeof(char)=1,sizeof(int)=4



指针计算

```c
int main(void)
{
    char ac[] = {0,1,2,3,4,5,6,7,8,9,};
    char *p = ac;
    printf("p  =%p\n", p);
    printf("p+1=%p\n", p+1);
    printf("*(p+1)=%d\n", *(p+1));
    int ai[] = {0,1,2,3,4,5,6,7,8,9,};
    int *q = ai;
    printf("q  =%p\n", q);
    printf("q+1=%p\n", q+1);
    printf("*(q+1)=%d\n", *(q+1));
    return 0;
}
```

```
*(p+1)=1
q  =0xbffc4d2c
q+1=0xbffc4d30
*(q+1)=1
```

- 这些算术运算可以对指针做：
  - 给指针加、减一个整数(+, +=, -, -=)
  - 递增递减(++/—)
  - 两个指针相减

```c
char ac[] = {0,1,2,3,4,5,6,7,8,9,};
char *p = ac;
char *p1 = &ac[5];
printf("p  =%p\n", p);
printf("p+1=%p\n", p+1);
printf("p1-p=%d\n", p1-p);
int ai[] = {0,1,2,3,4,5,6,7,8,9,};
int *q = ai;
int *q1 = &ai[6];
printf("q  =%p\n", q);
printf("q+1=%p\n", q+1);
printf("q1-q=%d\n", q1-q);
return 0;
}
```

当做两个指针相减的时候，它给你的是两个地址的差除以sizeof(它的类型)，也就是，在这两个地址之间，有几个这样类型的东西在，或者说，能放几个这样类型的东西

```
p1-p=5
q  =0xbffcd28
q+1=0xbffcd2c
q1-q=6
```

*p++

- 取出p所指的那个数据来，完事之后顺便把p移到下一个位置去
- *的优先级虽然高，但是没有++高
- 常用于数组类的连续空间操作
- 在某些CPU上，这可以直接被翻译成一条汇编指令

```c
char ac[] = {0,1,2,3,4,5,6,7,8,9,-1};
char *p = &ac[0];
int i;
for ( i=0; i<sizeof(ac)/sizeof(ac[0]); i++ ) {
    printf("%d\n", ac[i]);
}
// for ( p=ac; *p!=-1 ; ) {
while ( *p != -1 ) {
    printf("%d\n", *p++);
}
int ai[] = {0,1,2,3,4,5,6,7,8,9,};
int *q = ai;
return 0;
```

数组遍历方式

```c
char ac[] = {0,1,2,3,4,5,6,7,8,9,-1};
char *p = &ac[0];
int i;
for ( i=0; i<sizeof(ac)/sizeof(ac[0]); i++ ) {
    printf("%d\n", ac[i]);
}
for ( p=ac; *p!=-1 ;p++ ) {
    printf("%d\n", *p);
}
int ai[] = {0,1,2,3,4,5,6,7,8,9,};
int *q = ai;
return 0;
```

指针比较

- <, <=, ==, >, >=, != 都可以对指针做
- 比较它们在内存中的地址
- 数组中的单元的地址肯定是线性递增的

指针的类型

- 无论指向什么类型，所有的指针的大小都是一样的，因为都是地址
- 但是指向不同类型的指针是不能直接互相赋值的
- 这是为了避免用错指针

```c
char ac[] = {0,1,2,3,4,5,
char *p = &ac[0];
int i;
for ( i=0; i<sizeof(ac)/s
    printf("%d\n", ac[i])
}
// for ( p=ac; *p!=-1 ; )
while ( *p != -1 ) {
    printf("%d\n", *p++);
}
int ai[] = {0,1,2,3,4,5,6
int *q = ai;
q = p;
return 0;
}
```

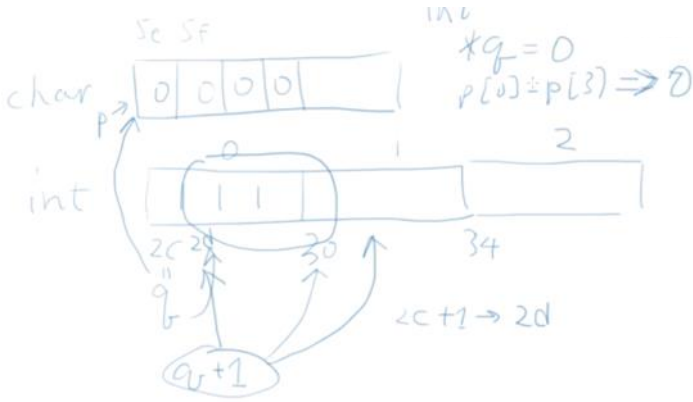
int类型指针，四个字节，char一个字节，把char的四个单元全变成0???

类型不匹配!

```
/Users/wengkai/cc/9.5.c:17:4: war
        q = p;
          ^ ~
1 warning generated.
[Finished in 0.1s]
```

0地址

- 当然你的内存中有0地址，但是0地址通常是个不能随便碰的地址
- 所以你的指针不应该具有0值
- 因此可以用0地址来表示特殊的事情：
  - 返回的指针是无效的
  - 指针没有被真正初始化（先初始化为0）
- NULL是一个预定定义的符号，表示0地址
  - 有的编译器不愿意你用0来表示0地址



用指针来做什么

- 需要传入较大的数据时用作参数
- 传入数组后对数组做操作
- 函数返回不止一个结果
  - 需要用函数来修改不止一个变量
- 动态申请的内存...

指针的类型转换

- void* 表示不知道指向什么东西的指针
  - 计算时与char*相同（但不相通）
- 指针也可以转换类型
  - int *p = &i; void*q = (void*)p;
- 这并没有改变p所指的变量的类型，而是让后人用不同的眼光通过p看它所指的变量
  - 我不再当你是int啦，我认为你就是个void!

9.2-2 动态内存分配

输入数据

- 如果输入数据时，先告诉你个数，然后再输入，要记录每个数据
- C99可以用变量做数组定义的大小，C99之前呢？
- int *a = (int*)malloc(n*sizeof(int));

malloc

#include <stdlib.h>

void* malloc(size_t size);

- 向malloc申请的空间的大小是以字节为单位的
- 返回的结果是void*，需要类型转换为自己需要的类型
- (int*)malloc(n*sizeof(int))

计算机不管什么是char，什么是int等等，计算机只知道内存是一大块空间，要几个字节数

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

int main(void)
{
    int number;
    int* a;
    int i;
    printf("输入数量: ");
    scanf("%d", &number);
    // int a[number];
    a = (int*)malloc(number*sizeof(int));
    for (i=0; i<number; i++ ) {
        scanf("%d", &a[i]);
    }
    for ( i=number-1; i>=0; i-- ) {
        printf("%d ", a[i]);
    }
    free(a);

    return 0;
}
```

malloc返回的结果是是void *，而a是int*，所以我们还要类型转换一下。(在前面加上(int*))

使用完malloc之后，就可以把a当做数组使用！！

没空间了？

- 如果申请失败则返回0，或者叫做NULL
- 你的系统能给你多大的空间？

这样的代码同时做了2件事情：

1.把malloc的结果赋给了p这个变量；

2.要让p得到的这个值，拿来做while的条件。

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

int main(void)
{
    void *p;
    int cnt = 0;
    while ( (p=malloc(100*1024*1024)) ) {
        cnt++;
    }
    printf("分配了%d00MB的空间\n", cnt);

    return 0;
}
```

每次申请100兆的空间，然后把申请到的空间，交给p，(p=malloc(100*1024*1024))中，对p做了一个赋值。赋值也是个表达式，有个结果，结果就是p得到的malloc的那个结果。

如果p得到的地址不是0，那就意味着它得到了一个有效的地址，那么，我们循环要继续，要让cnt去加加。
如果它得到的地址是0，那么while就要退出来。

```
9.7(20083,0xa039f1a8) malloc: *** mach_vm_map(size=104857600) failed (error code=3)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
分配了3300MB的空间
[Finished in 0.1s]
```


free()

- 把申请得来的空间还给“系统”
- 申请过的空间，最终都应该要还
  - 混出来的，迟早都是要还的
- 只能还申请来的空间的首地址
- free(0)?

```
4  int main(void)
5  {
6      void *p;
7      int cnt = 0;
8      p=malloc(100*1024*1024);
9      p++;
10     free(p);
11
12     return 0;
13 }
```

```
4  int main(void)
5  {
6      int i;
7      void *p;
8      int cnt = 0;
9      // p=malloc(100*1024*1024);
10     // p++;
11     p = &i;
12     free(p);
13
14     return 0;
15 }
```

```
4  int main(void)
5  {
6      int i;
7      void *p =0;
8      int cnt = 0;
9      // p=malloc(100*1024*
10     // p++;
11     free(p);
12
13     return 0;
14 }
```

```
9.7(20110,0x7fff72673310) malloc: *** err
*** set a breakpoint in malloc_error_brea
bash: line 1: 20110 Abort trap: 6
[Finished in 0.6s with exit code 134]
```

```
9.7(20117,0x7fff72673310) malloc: *** e
*** set a breakpoint in malloc_error_br
bash: line 1: 20117 Abort trap: 6
[Finished in 0.6s with exit code 134]
```

```
12     free(NULL);
13
14     return 0;
15 }
```

```
[Finished in 0.1s]
```

```
[Finished in 0.1s]
```

0不可能是malloc得到的地址，free(null)时啥都不干
指针初始化为0，之后如果没有去malloc，或者malloc失败，之后free这个p没有问题
好习惯！！初始化为0!

常见问题

- 申请了没free—>长时间运行内存逐渐下降
  - 新手：忘了
  - 老手：找不到合适的free的时机
- free过了再free
- 地址变过了，直接去free

如果程序是个小东西，不会有什么伤害（内存垃圾）。因为程序运行结束后，OS保证程序中的东西会被释放。

大程序？？一直不释放？？内存不够了！！
比如服务器等等

9.2-3 函数间传递指针

```
int* init(int a[], int length);
int* print(int a[], int length);

int main()
{
    const int MAX_SIZE = 1000;
    int size;
    do {
        printf("输入数量(0,1000): ");
        scanf("%d", &size);
    } while (size>0 && size<MAX_SIZE);
    int* a = (int*)malloc(size*sizeof(
    print(init(a
    fr
    return 0;
}
```

```
int* init(int a[], int length)
{
    int i;
    for ( i=0; i<length; i++ ) {
        a[i] = i;
    }
}

int* print(int a[], int length)
```

除非函数的作用就是分配空间，否则不要在函数中malloc然后传出去用

好的模式

- 如果程序中要用到动态分配的内存，并且会在函数之间传递，不要让函数申请内存后返回给调用者
- 因为十有八九调用者会忘了free，或找不到合适的时机来free
- 好的模式是让调用者自己申请，传地址进函数，函数再返回这个地址出来

函数返回指针?


```
{
    int i;
    for ( i=0; i<length; i++ ) {
        printf("%d\t", a[i]);
    }
    printf("\n");
    return a;
}
```

- 返回指针没问题，关键是谁的地址？
  - 本地变量（包括参数）？函数离开后这些变量就不存在了，指针所指的是不能用的内存
  - 传入的指针？没问题
  - 动态申请的内存？没问题
  - 全局变量—>以后再解释

函数返回数组?

- 如果一个函数的返回类型是数组，那么它实际返回的也是数组的地址
- 如果这个数组是这个函数的本地变量，那么回到调用函数那里，这个数组就不存在了
- 所以只能返回
  - 传入的参数：实际就是在调用者那里
  - 全局变量或动态分配的内存

和返回指针是一样的!