- **指针应用场景一**
  如在函数内交换两个数在函数外就会失效，因为函数只传入值，但学完指针后就可以交换了

```
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
  或，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，通常-1或0（在文件操作会看见大量的例子），但是当任何数值都是有效的可能结果时，就得分开返回了

```
int divide(int,int b,int *);

int main(void){
    int a = 2;
    int b = 3;
    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;
}
```

后续的语言采用了异常机制来解决这个问题

- **指针最常见错误**

定义来指针变量，还没有指向任何变量，就开始使用指针

```
//错误实例
int *p;
*p = 12;
```

创建来一个指针但没有做初始化指向任何变量，这时候p可能指向很奇怪的地方，在做写入12后，二那个地方是不能写的地方，程序就崩溃了

- **传入函数的数组成了什么**
  对于函数的参数的sizeof返回的是int*的sizeof，而不是数组的sizeof
  形式参数的数组实际上是指针，可以改写成int *a
  数组变量是特殊的指针
  两个数组不能互相赋值，是因为int a[]事实上等同于 int * const a，是常量指针，地址是唯一固定的，不能赋值修改
- **指针与const**
  指针包括两部分，指针本身的地址，以及指针所指向的变量，因此，指针可以是const，其所指向的变量也可以是const

```
int *const q = &i; //此情况下，q是const
*q = 26; //ok
q++; //ERROR

const int *p = &i;//(*p)是const
/*不能通过p进行修改*/
/*int const *p = &i */
*p = 26;//ERROR (*p)是const
i = 26;//ok
p = &j;//ok
```

判断的标志是const在*前面就表示*p不能被修改，const在*后面就表示p不能被修改

- **转换**
  总是可以把一个非const值转换为const

```
void f(const int*a);
int a = 12;
f(&a);//ok
const int b =13;
f(&b);//ok
```

```
b = a+1; //error
```

当要传递的参数类型被地址大的时候，这是常用的手段，既能用较小的字节数传递参数，又能避免函数对外面的变量进行修改

```
const int a[] = {1,23,45,5,};
```

数组变量本身就是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就进行赋值

- **保护数组数值**
  将数组传入函数中，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);
```

- **指针运算**

```
int ai[] = {1,2,}
int *p = ai;
printf("q+1=%p\n",q+1);
```

指针加1，不是在指针地址上加1，而是在地址值上加一个sizeof(指针所指的类型)

```
*(p+1) --> ai[1]
```

如果在地址指针上加1是没有任何意义的
这些算数运算可以对指针做
给指针加减一个整数；递增递减（++，--）；两个指针相减，不是给出地址差，而是给出有几个这种sizeof类型的东西

```
*p++
```

取出p所指的那个数据来，完事后，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

- **0地址**
  内存中有0地址，但0地址通常是不能直接碰的地址，所以指针不应该具有0值，
  因此，可以用0地址表示特殊的事情
  返回的指针无效，
  指针没有被真正初始化 （先赋为0值） NULL是一个预定定义的符号，表示0地址
  因为有的编译器不愿意你用0来表示0地址
- **指针类型**
  指针指向对象的类型不匹配，不能互相赋值，会导致强制导致赋值，产生warning，初学不要使用
  void *表示不知道指向什么类型的指针，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所指向的变量的类型，而是让人用不同的眼光通过p看它所指向的变量，不再当其是int，而是当其是void

- **动态内存分配**
  c99可以用变量做数组定义的大小，之前？用动态内存分配

```
int *a = (int*)mailloc(n*sizeof(int));
```

```
// man malloc
#include<stdio.h>
#include<stdlib.h>
int main(void){
    int number;
    int *a;
    int i;
    printf("please insert your number:");
    scanf("%d",&number);
    //int a[number]; c99中可以直接这样
    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要的不是数组有多少个单元，而是这个数组占据多少个空间，以字节为单位，mallco返回的是void*，而a是int*,因此需要进行类型转换,其后便可以当成普通数组使用,最后使用完成后，需要free(a)把空间还掉
没空间了？如果申请失败则返回0，或者NULL，系统能给多大空间呢?

```
int main(void){
    void *p;
    int cnt =0;
    while(p=malloc(100*1024*1024)){
        cnt++;
    }
    printf("分配了%d00m的空间\n", cnt);

    return  0;
}
```

- **free()**
  free()是和malloc()配套的函数，把申请得来的空间还给系统，只能还申请来的空间的首地址
  free(NULL)编译没问题，因为0不可能给，因此什么也不会做。良好的习惯，写指针先初始化为NULL，因为这样的话，即使后面malloc失败，fre也不会导致错误
- **常见问题：**
  申请来没free，长时间运行内存逐渐下降，对于小程序没问题，内存垃圾会在程序关闭后自动清理，但在大程序中不行；对于老手来说，找不到合适的时机，合适的地方来做free，或者free过后又再次free

```
char *p;
while (1) {
    p = malloc(1);
    *p = 0;
```

```
}
```

这段程序的结果是最终程序会因为向0地址写入而退出，因为，程序一直分配内存，肯定会引起内存耗尽。而malloc在分配内存失败时并不会终止程序，而是返回NULL指针。而第5行代码试图向NULL指针位置写入数据，这会引起程序终止（通常操作系统会因为“段错误”而终止程序）。