指针

“指针”，是真正的能够**存储地址**的变量

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**

由于指针变量的特殊性，我们若在函数中修改了指针变量，**那么也会修改它所指向的值**

&，它实质上是一个 运算符 ，它能够**获得变量的地址**，常和指针的操作一起进行

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

int main(void){
    int i = 6 ;
    printf("&i = %p\n",&i);
    f(&i);  // 将变量i的地址传入
    g(i);
    return 0 ;
}

void f (int *p){
    printf(" p = %p\n" , p); //显示该地址是多少，也就是地址大小
    printf(" p = %d\n" , *p); // 显示地址上变量对于的值是多少
    *p = 26 ; //就在这里，把p变量的地址指向的那个变量（就是i）改为26
    //这里将传入的p指向的位置上面的值改成了26，实际上就是把传入的变量的值做了修改
}

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

第一个实例:

```c
#include <stdio.h>

int main ()
{
   int  var = 20;   /* 实际变量的声明 */
   int  *ip;        /* 指针变量的声明 */

   ip = &var;  /* 在指针变量中存储 var 的地址 */

   printf("var 变量的地址: %p\n", &var  );

   /* 在指针变量中存储的地址 */
   printf("ip 变量存储的地址: %p\n", ip );

   /* 使用指针访问值 */
   printf("*ip 变量的值: %d\n", *ip );

   return 0;
}
```

上述代码编译会得到结果:
var 变量的地址: 0x7ffeeef168d8
ip 变量存储的地址: 0x7ffeeef168d8
*ip 变量的值: 20

指针上的+1指的是增加一个sizeof()的单位
两个指针是可以相减的,指针相减,结果是是 `(地址差)/sizeof()` ,表示的是**二者中间有多少"这种类型的东西"**

指针可以使用```*p++`` , 意义是“取出p所指的那个数据，然后再利用指针++,把p移到下一个位置去”

> *的优先级没有++高
> 比如我们就可以把遍历数组的代码写为

```c
int main(void){
    char ac[] = {0,1,2,3,4,5,6,7,-1} ; //最后一个-1表示这是数组的结尾
    char *p = &ac[0] ;

    while(*p != -1 ) {
        printf("%d \n" , *p++);
    }
    return 0 ;
}
```

无论指向的是什么类型，所有的**指针的大小**都是一样的，因为它们本质上都是地址
但是指针存在类型的差别，**不同类型的指针是不能相互赋值的**
`void*`：表示这是一个指针，但不确定它指向的是什么

malloc

来自 `#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))`

申请失败时会返回一个0，或者NULL

free()

free是和malloc配套的函数，把申请来的空间重新归还给系统
只能还申请来的空间的**首地址**，也就是地址改变之后(比如p++,p--)是不可以归还的
必须归还最开始的，申请来的那个地址

为了配合，**建议在初始化指针的时候都给它一个0地址，如** `void *p = 0;`
如此一来，若我们在运行过程中没有malloc这个指针，最终归还的时候也是free(p)也是就free(NULL)，不会报错

> free(NULL)总是可以的

数组

函数的参数表中的数组，实质上就是个指针，指向该数组初始地址/基地址的一个指针

> 这里牵扯另外一个东西，在数组中变量是按顺序存储的，因此只要知道基地址，我们就知道数组中任意一个元素的位置
> 如基地址（下标为0）是1000，存的数据占4个位置，那么第二个（下标为1）的数据则在1004

因此在函数中我们**不能直接用sizeof**得到正确的数组长度

> 函数参数表中的数组实际上是指针

数组变量是特殊的指针,这使得它有如下性质
1.数组变量本身表达地址，所以我们取数组的地址时无需使用&

```
int a[10] ;
int *p = a ;
```

2.但是数组的单元表达的是变量，我们需要用&来取它。数组a的地址，等于数组单元a[0]的地址

> 可以想象为数组是一系列连续的指针地址构成的,其中第一位(下标为0的)那一位代表整个数组的开始
> 3.*运算符可以对指针做，也可以对数组做
> 4.数组变量是const的指针，所以不能被**赋值**

当我们取出地址时， `&a,a,a[0]` 是相同的

字符串

在C语言中，字符串就是字符数组 char[]

在C语言中，字符串指以0（整数0）结尾的一串字符
`0`和`'\0'` 是一样的，但是和 `'0'` 是不一样的
0标志着字符串的结束，但是它不是字符串的一部分，计算字符串长度的时候也不包含这个0
字符串以数组的形式存在，也以数组或指针的形式访问（更多的是以指针的形式）
在 `string.h` 中有很多处理字符串的函数

这是一个字符数组: `char word[] = {'H','e','l','l','o','!'}`
而这是一个字符串: `char word[] = {'H','e','l','l','o','!','\0'}`
二者的区别在于，我们在初始化该数组的过程中，用一个 **\0** 结尾

C语言的字符串是以字符数组的形态存在的，不能用运算符对字符串做运算，通过数组的方式可以遍历字符串

我们有多种方式表达字符串

```
char *str = "Hello" ;
char word[] = "Hello" ;
char line[10] = "Hello" ;
```

这里面 "Hello"被称为**字符串常量**,"Hello"会被编译器变成一个字符数组放在某处，这个数组长度是6，结尾还有表示结束的0(Hello五位,0一位,共六位)

两个相邻的字符常量会自动连接

当我们编译过程中有两个相同的字符串（比如s1 s2 两个字符串都是Hello world），它们会指向同一个地方
如果想要制作一个能修改的字符串，那么在**一开始**就需要用**数组**定义

数组字符串：这个字符串在这，作为本地变量会被自动回收
指针字符串：不知道这个字符串在哪，需要处理参数，可以动态分配空间

如果要构造一个字符串-->数组
如果要处理一个字符串-->指针

赋值

```
char *t = "title" ;
char *s ;
s = t ;
```

实际上并没有产生新的字符串，只是让指针s指向了t所指的字符串。对s的任何操作就是对t做的,**因为二者指向同一块地址**

输入输出

%s代表输入输出的是字符串

```
char string[8];
scanf("%s",string);
printf("%s",string);
```

在百分号和s中间，可以增加一个数字，表示我们希望最多可以读入多少字符，以此提高安全性。此时就不一定是以空格tab回车来区分了，读完了，这个scanf就结束了

char[][X]

代表的是一个字符串数组，每一个字符串的大小至多为X(有X-1字符)
可以认为是把一个个“数组型的字符串”放置到了数组里面，比如
a[0] 的值是 world\0
a[1] 的值是 Hello\0

string.h

在string.h中，有许多帮助处理字符串的函数

常用函数

strlen

size_t strlen(const char *s) ;
返回s的字符串长度，不包括结尾的0

strcpy

char * strcpy(char *restrict dst , const char *restrict src) ;
可以把 `src` 的字符串拷贝到 `dst`
在c99下，restrict表名src和dst不重叠
最终返回的是 `dst`
另外，参数中，注意第一个参数是目的地，第二个是源
目的地需要有足够空间

```
char *dst = (char*) malloc(strlen(src)+1) ;
// +1是因为考虑末尾的0
strcpy(dst,src) ;
```

strcat

char * strcat(char *restric s1 , const char *restrict s2) ;
把s2拷贝到s1的后面，接触成一个长的字符串
返回s1
s1需要有足够的空间

> 尽可能不要使用strcpy和strcat因为有安全问题，可以使用下面的安全版本

```
char * strncpy(char *restrict dst , const char *restrict src , sizr_t n) ;
chat * strcat(char *restrict s1 , const char *restrict s2 , size_t n) ;
int strncmp(const char *s1 , const char *s2 , size_t n );   //判断前n个字符是否是xxx
```

size_t n 参数表示了最多可以运输多少字符

字符串搜索函数

char * strchr(const char *s , int c);
char * strrchr(const char *s , int c);

返回NULL表示没有找到
(前者是从左往右，后者是从右往左)

因为返回的是对应位置的指针，所以也可以尝试打印接下来的部分：

```
char s[] = "hello" ;
char *p = strchr(s,'l');
printf("%s \n" , p );
```

最后结果会是"llo"
因此我们可以利用这个特性寻找第n个字符，比如：

```
char s[] = "hello" ;
char *p =s strchr(s,'l') ;
p = strchur(p+1,'l');
printf("%s \n" , p );
```

在字符串中寻找字符串

char * strstr(const char *s1 , const char *s2);
char * strcasestr(const char *s1 , const char *s2); //相较前者，忽略大小写

结构体

声明结构的形式

A:

```
struct point{
    int x ;
    int y ;
};

struct point p1,p2 ;
```

此时p1和p2都是point,里面有x和y的值

B:

```
struct {
    int x ;
    int y ;
} p1,p2 ;
```

p1,p2都是无名结构，里面有x和y
一般用于暂时使用

C:

```
struct point{
    int x ;
    int y ;
}p1,p2 ;
```

这里p1和p2都是point，里面有x和y的值

对于第一种和第三种形式，都声明了结构point
第二种形式没有声明point,只是定义了p1,p2两个变量

C语言提供了一个叫 `typedef` 的功能来声明一个已有的数据类型的新名字
比如 `typedef int Length;`
这样可以使得**Lenghth**成为**int类型**的别名

同样的,我们可以利用 `typeof` 操作,来为自己创建的结构来定义一个名字,这样就不用每次都打上 `struct xxx` 来使用了

```
typedef struct ADATEexpale {
    int month ;
    int day ;
    int year ;
} Date ;
```

或者：

```
#  define SIZE 100
<p class="mume-header " id="define-size-100"></p>


typedef struct{
    int a[SIZE] ;
    int length ;
}SqList ;

// 在这之后就可以直接用SqList来表示该结构体了，声明变量无需struct
SqList l1 ;
```

整个结构可以作为参数的值传入函数；同理，我们也可以返回一个结构

当我们想要利用函数来返回一个结构体的时候，有两种选择：

一，新建一个结构体，接受值。在函数返回时返回该结构体，并在需要它的地方直接使用p1 = p2

二，利用指针。这是更推荐的方式，因为它消耗的时间和空间都更小(见后文)

结构体与指针相关,以及->运算符

和数组不同，结构变量的名字并不是结构变量的地址。

因此表示某种结构变量的地址，需要&运算符

```
struct date{
    int month ;
    int day ;
    int year ;
}myday ;

struct date *p = &myday ; //这里表示p指针取得myday对应的地址

(*p).month = 12 ;
//p是指针，(*p)表示的则是这个结构体，因此可以直接利用 (*p).month来访问结构成员


p->month = 12 ;
//p是指针，->代表p指针所代表的那个结构体中的对应的成员
//->month即该指针对应的结构体中的month成员
```

结构中的结构的成员的访问是和单层结构相同的

也是使用．来进行逐级访问

示例

若有变量定义

```
struct rectangle r , *rp ;
rp = &r;
```

那么以下的四种形式表达的是一样的

```
r.pt1.x // 经典
rp->pt1.x // 通过指针得到结构体，然后用dot继续访问
(r.pt1).x
(rp->rt1).x
```