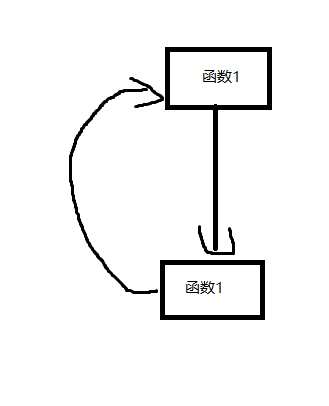
# C语言

**阶乘**

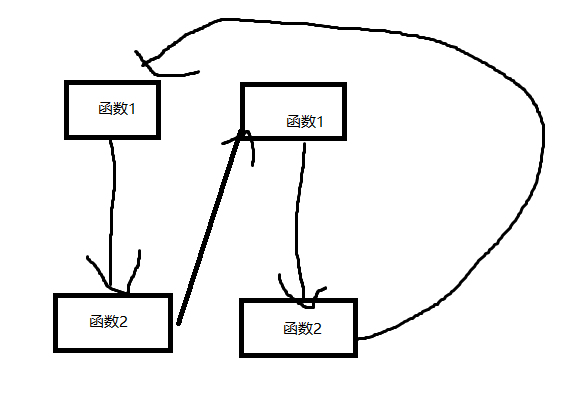
首先我们要知道阶乘是个什么概念

阶乘其实就是一个特殊的函数嵌套



这个图说明了一个简单的递归问题，也就是函数自己调用自己

还有一种如下图的间接调用



这种是一个函数调用另外一个函数然后那个函数又调用原来调用它的函数

这叫间接调用

递归存在一个**压栈弹栈**的一个问题首先我们先看一个程序

#include <stdio.h>

#include <stdlib.h>

void c(void)

{

printf("d\n");

printf("e\n");

}

void b(void)

{

printf("c\n");

c();

printf("f\n");

}

void a(void)

{

printf("b\n");

b();

printf("g\n");

}

int main()

{

printf("a\n");

a();

printf("h\n");

exit(0);

}

这个程序的输出结果是：

a

b

c

d

e

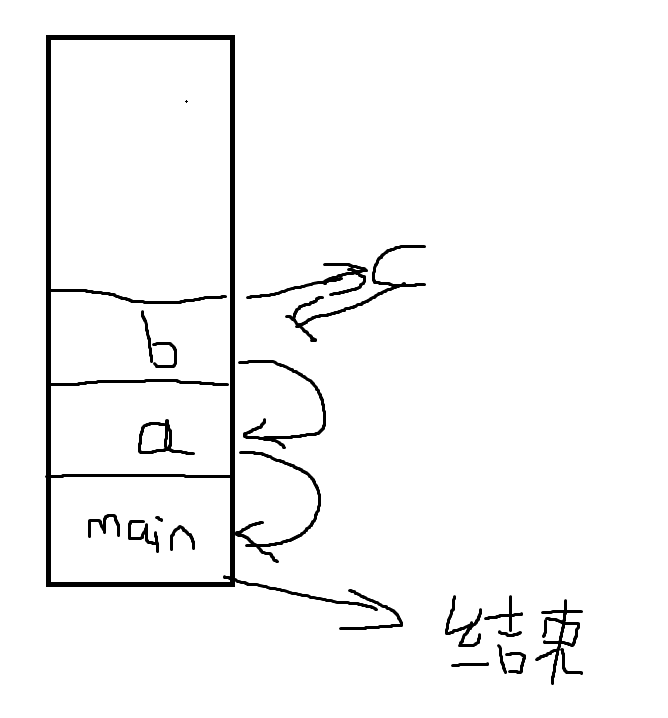
f

g

h

这里有一个压栈弹栈的问题

首先我们main函数打印了一个“a”，然后进入函数a，这时main函数的现场被压入栈，函数a打印“b”后调用函数b，a函数的现场被压入栈，b函数打印c调用c函数，b的现场被压入栈，c把“d”和“f”打印完后返回弹栈，栈是先进后出，所以现在栈的情况是这样



所以c退出后首先回到b的现场，b打印出来f返回a·····直到返回到main函数结束

递归就是利用了这一点

比如说我们利用递归算出斐波那契数列，

#include <stdio.h>

#include <stdlib.h>

int fi(int n)

{

if(n < 1)

return -1;

if(n == 1 || n ==2)

return 1;

return fi(n-1) + fi(n-2);

}

int main()

{

int a;

int n;

scanf("%d",&n);

a = fi(n);

if(a == -1)

{

printf("你他妈在逗我！\n");

}

else

{

printf("%dfi = %d\n",n,a);

}

exit(0);

}

这个程序就是计算斐波那契数列，运用递归自己调用自己，然后不断地把自己压入栈，然后压入到最后一个的时候一个一个地弹出，也就是说，首先我们调用压栈的时候是在把参数逐个递减，真正的计算部分是在弹栈的时候完成的，这就是递归的原理

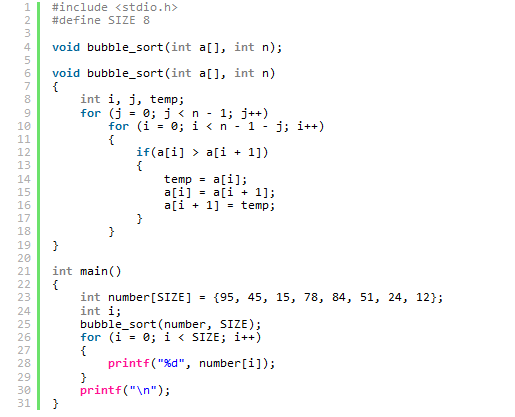
**尾递归**：尾递归其实也就是在函数最后使用的递归，也就是说在递归返回的时候没有任何操作也就是说是放在return之前

**冒泡排序**

所谓冒泡排序就是先把数组内第一个元素与第二个元素相比较，如果第一个元素大的话两个元素的互换，接着就是第二个元素和第三个元素比较，以此类推比到

最后一个元素为止，然后第二次比较的时候就不比最后一个元素了，就比数组大小减一个数，以此类推就可以排出大小了

举例



**选择排序的方法**：

其实选择排序跟冒泡排序是差不多的

选择排序是把最小的放在前面，从第一个数开始依次跟其它数比较如果没有比它更小的那么它就不动，如果有就互换位置依次类推就可以了

数组的大小尽量用#define定义，在判断一个数组的大小的时候尽量这么写：sizeof(数组)/sizeof(数组[0])

float类型无法精确的确定一个值，所以如果if中有 float == 0 这种的那就是不对的

定义char型变量的时候有无符号是未定义的

未定义返回值的函数默认的返回值是**int**型

变量要先定义后使用，定义必须在函数或者函数的前面

如果一个函数不是void型，但是没有定义返回值，那么这个函数返回的是最后一条语句的返回值（printf（）也是有返回值的）

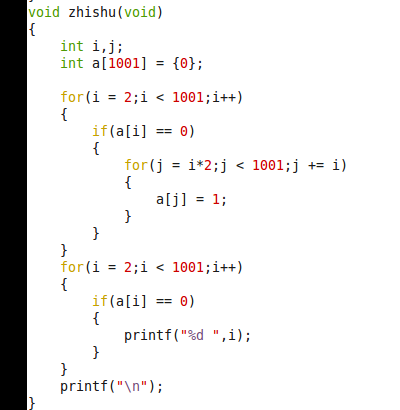
写程序的时候要防止内存泄露：不要忘了分配的内存要关闭

sscanf是一个很好用的函数，利用它可以从字符串中取出整数、浮点数和字符串等等

**删除法求质数:**

质数是除了1和它本身以外没有任何因数的数，2是最小的质数

我们可以先定义一个数组，使它的下标和正常的数字对应，然后把不是质数的空间写为非0，然后再检测打印就可以了



char str[512]={0};  
　　sscanf("123456","%s",str);  
　　printf("str=%s",str);

表示将123456中字符串放到str中

C语言中不能用二进制赋值，只能用十六进制 八进制 和十进制赋值

八进制的换算方法和十六进制的差不多，三个一组（十六进制是四个一组） 4 2 1 4 2 1换算

八进制 ：1 0 1 0 1 0 =52

C语言中：Bxxx 二进制

0xxx 八进制

0Xxx 十六进制

什么都不加 十进制

转义字符表

|  |  |  |
| --- | --- | --- |
| 转义字符 | 意义 | ASCII码值（十进制） |
| \a | 响铃(BEL) | 007 |
| \b | 退格(BS) ，将当前位置移到前一列 | 008 |
| \f | 换页(FF)，将当前位置移到下页开头 | 012 |
| \n | 换行(LF) ，将当前位置移到下一行开头 | 010 |
| \r | 回车(CR) ，将当前位置移到本行开头 | 013 |
| \t | 水平制表(HT) （跳到下一个TAB位置） | 009 |
| \v | 垂直制表(VT) | 011 |
| \\ | 代表一个反斜线字符''\' | 092 |
| \' | 代表一个单引号（撇号）字符 | 039 |
| \" | 代表一个双引号字符 | 034 |
| \? | 代表一个问号 | 063 |
| \0 | 空字符(NULL) | 000 |
| \ddd | 1到3位八进制数所代表的任意字符 | 三位八进制 |
| \xhh | 1到2位十六进制所代表的任意字符 | 二位十六进制 |

转义字符容易错的地方：’\018’这样的写法是错误的，0开头代表是八进制，但是八进制里面没有数字‘8’，而且如果它是字符串的话应该是用””来引上，所以这样是没有意义的

“a” 和‘a’是不一样的，一个是字符串常量，一个是字符常量

宏定义的时候尽量在宏体上加一个括号

Typeof（参数）；获取当前参数是什么类型

register（建议型）：只能定义局部变量，不能定义全局变量，大小只能定义32位的数据类型，寄存器没有地址，不能打印出地址查看或使用

static：初始化为0值或空值，其变量具有继承性

继承性：在前面定义了一个静态变量X 那么之后的代码再次定义静态变量x的时候X不再被再次定义，直接就用的上面的X

auto变量是一个随机值

局部变量和全局变量：如果局部变量和全局变量都定义了一个“i”，那么作用范围是小括号屏蔽大括号的，也就是说在局部变量所在的括号里，用的是局部变量的“i”而不是全局变量的（内部屏蔽外部）

|| 和 &&是判断用的

| 和 &是计算用的

强制类型转换不能改变一个变量的值

左移运算相当于这个数的十进制数乘2

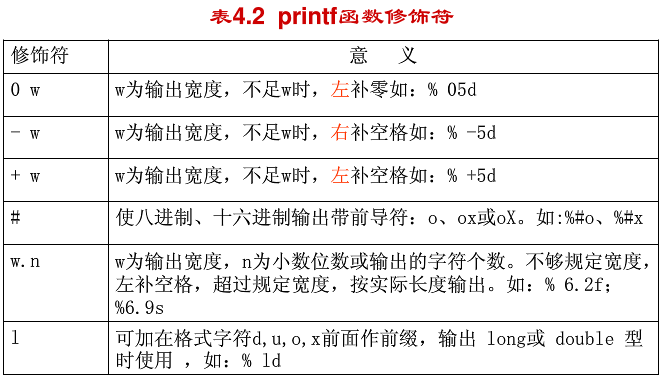
右移运算相当于这个数的十进制数除2

Printf输出“%”后的字母类型表：

|  |  |
| --- | --- |
| **格式字符** | **意义** |
| d | 以十进制形式输出带符号整数(正数不输出符号) |
| o | 以八进制形式输出无符号整数(不输出前缀0) |
| x,X | 以十六进制形式输出无符号整数(不输出前缀Ox) |
| u | 以十进制形式输出无符号整数 |
| f | 以小数形式输出单、双精度实数 |
| e,E | 以指数形式输出单、双精度实数 |
| g,G | 以%f或%e中较短的输出宽度输出单、双精度实数 |
| c | 输出单个字符 |
| s | 输出字符串 |
| P | 输出该变量的地址 |
|  |  |

这些字母前面还可以添加一些修饰符（修饰符可省略），修饰符的取值范围如下表：

数组越界但不报错现象是由于数组下表是指针，而指针偏移并不出错



.n需要注意一下，.n也可以作用在字符串中，表示显示的有效数字，如果.n小于字符串的长度，那么n后的字符将被忽略打印（但是不是改变字符串 只是忽略而已）

***“-”  项目的左对齐的；也就是说，会把项目打印在字段的左侧开始处 示例：“%-20s”***

***“+”  有符号的值若为正，则显示带加号的符号；若为负，则带减号的符号 （带代数符号打印）示例：“%+6.2f”***

***“#”  使用转换说明的可选形式。若为%x和%X格式，则以0x或0X开始。对于所有的浮点形式，#保证了即使不跟任何数字，也打印一个小数点字母。 示例：“%#o”，“%#8.0f”，“%+#10.3E”***

***“0”  对于所有的数字格式，用前导零而不是用空格填充字段宽度。如果出现-标志或者指定了精度（对于整数）则忽略该标志 示例：“%101d”，“%08.3f”***

在C中没有定义的常量后面加“L”或者“LL”的时候就可以让其变成有意义的值

GCC编译器提供的宏：

\_\_FUNCTION\_\_ 打印当前所在的函数名

\_\_LINE\_\_打印当前语句所在的行号

用scanf输入字符串的时候字符之间不能用空格或者tab分开，否则系统认为是输入结束，如果输入两个参数，中间什么都不加的话，输入的时候空格 tab 回车都是可以的

scanf函数是有返回值的，以便于检测输入时候的正确和错误，输入几个值如果成功了就返回几个，如果返回的数小于这个值，那就代表其中有一个或几个输入失败了，成功输入几个就返回几个

scanf输入时可以这么写scanf(“%\*d%d”,&i);其中“\*”代表抑制符表示输入的这个值只是放在程序中，程序“吃掉”了这个数并不是赋值给i，接下来输入的数才会给i赋值

字符输入输出：getchar，putchar

字符串输入输出：gets（这个函数很危险）,puts

因为gets这个函数不检查缓冲区是否溢出，所以很危险

If-goto可以实现循环，但是比较危险，goto只能在一个函数内用而且是无条件的跳转，最主要的是它不能保存现场容易造成程序结构的破坏，但不是不能用

Ctrl+c可杀掉死循环

Continue只是跳出一次循环，换句话说就是如果continue接下来还有语句的话这次就不执行了然后不跳循环而是进行下一次循环的判断

用scanf“%s”是无法获取带有分隔符的字符串的，比如说hello world它只能获得hello

在字符串头文件中string.h

strlen是计算字符串长度的，但是只以“\0”位单位，也就是说只识别这一个字符串“\0”之前的长度而且“\0”不算在个数之内

strncpy比strcpy多了一个参数，那个参数的意思是前n个字节复制到所指的数组中

strncat一样，意思是指定前几位和目标对接，函数的意思就是后面的字符串接到前面的字符串上

strncmp也是，两个字符串相互比较，比的是ascii码

一个二维数组的定义可以这么理解：比如定义一个二维数组：int a[2][3]可以理解为（int [3]） a[2],这代表的意思就是元素为3个的数组被定义成了两个

指针

无论什么类型的指针所占空间大小都是一样的（在同一个编译环境中）

指针变量指向的类型必须要和指针变量的类型要相同，比如说一个char型指针指向int型变量，int型是占4个字节，但是char型是1个字节，也就是说当char型指针读取int型的时候只读取了一个字节

当我们定义指针的时候如果现在指针没有一个确定的地址要给它，那么我们需要给它一个NULL来预防野指针的产生

野指针指向一个已删除的对象或未申请访问受限内存区域的[指针](http://baike.baidu.com/view/159417.htm)。也就是说指向了一个未知不可控的空间甚至是没有指向任何地址

NULL绝对不会给任何东西分配空间

指针变量一旦定义就必须要定义下来

Void \*是一个百搭的数据类型，任何类型值都能赋值给它，这个类型通常都在传递的参数不确定是什么类型的时候用这个定义void \*同样赋值给谁都是合法的

定义的数组名是常量而定义并把数组赋值给指针的是变量，这是数组名和指针变量的区别：一个常量，一个变量

也就是说，数组是不会变的，但是如果用指针比如说p++自增的话那么p就会指向别的空间而数组名则不会，它只会指向这个数组所在的地址

定义一个数组也可以不需要数组名字比如：

int \*p = (int [3]){1,2,3};这样也是可以的

**指针与二维数组：**

如果说一个二维数组要给一个指针变量赋值的话直接把数组名赋值是不合法的，因为一个数组名+1是加一行，指针变量+1是加一列，所以这么赋值是不合法的

我们可以在数组名之前加上一个“\*”或者“数组名【】【】”，这样就可以了，因为数组名其实也是一个地址指针，以a[2][3]为例，\*(a + i) 是加i列，\*(\*(a + i) + j) 是第i行第j列，所以这样就把地址给了指针变量了a[ i ][ j ]跟 \*(\*(a + i) + j) 的效果是一模一样的

**数组指针：**

数组指针顾名思义就是一个指向数组的指针，也就是说这个指针是指向数组的

具体定义是这么定义的 **int (\*p)[3]** 它一般用于是二维数组，比如上面说到的，一个指针变量要指向一个二维数组就要加上一个“\*”或者要写上这个数组的第一个元素的地址，那么这个数组指针就不是那么的麻烦比如一个二维数组是**a[ 2 ][ 3 ]** 那么我们就可以直接赋值 ： **int (\*p)[3] = a;** 这样就可以了，这样的功效和a一样的，比如说p+1是跳到一行，在这里我对这个数组指针的理解是，定义一个一次可以偏移n个地址长度的指针变量，比如说上面的\*p 它是定义了3，所以它一次可以偏移3个地址长度，所以p + 1和a + 1是一个效果，如果p定义了一个2，那么p+1将不再指向第二行而是指向a[ 1 ][ 2 ],所以这个就是数组指针

**字符数组与字符指针**

字符数组和字符指针的区别就在于，一个在strcpy是合法的一个是不合法的，可以这么理解，一个是常量，占用一段空间，所以可以把一串不大于这段空间的字符串覆盖到那个空间上，而指针却只是一个地址，如果我们用sizeof一下它的话不管是什么样的指针都是相同的值，因为他们就是一个地址的大小，没有足够的空间，所以是不合法的

**const与指针的关系**

**#define**定义的东西是不可变的，但是没有报错功能，因为它是预处理

用const定义的变量可以理解为常量，因为它是不可被赋值不可变的

所以用const定义的东西可以理解为可以报错的define，所以这样const就没有define可以定义的范围广

但是我们可以使用一种方法让它改变，也就是说定义一个指针指向这个const的地址然后再去“\*”运算一下给它赋值，但是这样做是很危险的，有些编译器甚至会编译不过去，所以不建议这么用，因为毕竟定义的是一个常量，如果想改变值的话就定义一个变量好了

**常量指针和指针常量的区别：**

首先我们先看一下定义

const int \*p 常量指针

int const \*p 常量指针

int \*const p 指针常量

首先我们应该区分一下

*最简单的方法就是看const和“\*”的位置，如果const在“\*”前面，那就是常量指针，如果const在“\*”后面那就是指针常量*

*接下来是更深入的理解：首先我们看一下常量指针，const定义在“\*”前面也就是说const定义的是\*p本身，\*p本身是这个指针指向的地址内的数值，所以常量指针就是指针指向可以随意改变，但是指针指向的该空间的数值是不可以被改变的，如果指针地址变了，那么原来的那个地址的值就可以改变了，但是现在指向的不能改变，也就是说指向哪里，哪里就不能被改变*

*指针常量就是和常量指针相反，我们看一下，const这回定义的是p 而不是\*p所以这个指针的指向不能被改变，但是指向那个空间的值可以改变*

**指针数组与数组指针：**

我们数组指针在上面有提到过，

我们看一下这两个的定义：

数组指针：【存储类型】 数据类型 （\*指针名） 【下标】

比如说 int （\*p） [ 2 ]

指针数组：【存储类型】 数据类型 \*数组名 【下标】

比如说 int \*a [2]

这两个实际上在定义上的区别其实就是一个括号的问题

我们来看一下它的含义问题，首先从字面上来看，指针数组是一个数组，数组指针是一个指针

我们再来深入地理解一下，首先，指针数组其实就是一个数组里面装的全都是指针，而数组指针上面也讲了，这个就是定义了一个长度为多少的指针

指针数组定义的是一个数组，数组指针定义的是一个指针，这是他们两个的本质的区别

**多级指针**

当我们定义一个二级指针是这么定义的 int \*\*p，这怎么使用呢，首先定义了一个一级指针，然后给了它一个初值，然后我们再定义二级指针，将一级指针给这个二级指针，从而实现间接访问的

比如：

int i = 10;

int \*p = &i;

int \*\*q = p;

我们来分析一下，

q为多少

\*q为多少

\*\*q为多少

P为多少

\*p为多少

q就是它本身的地址

\*q是指针p的地址

\*\*q也就是说\*（\*q），对\*q取值，\*q是p的地址，所以说这是对p取值也就是i的值

p为i的地址值，也就是等于\*p

\*p就不多说了，i的值

多级指针和这个二级指针是一样的

当我们定义一个结构体指针的时候，我们可以拿这个指针当做if的判断语句

比如说我定义了一个结构体

Struct a {

Int a;

}

Struct a \*b;

这个\*b我们可以当做if的判断语句，如果\*b只是被定义了而没有赋值（也就是说指向NULL）那就为假，如果赋值了那就为真

**一维数组与函数形参的关系**

首先我们知道一维数组其实就是一个常量的指针，所以我们传递参数的时候需要注意一下，我们如何把一个数组全部传递给一个函数，我们不能写a[ 0 ]因为这样只能传递第一个数，我们如果要传递过来需要写a就可以了，但是这里的函数的形参的接收类型需要注意一下，我们不能用跟定义一个数组的类型是一样的，因为传递过来的“数组名”是一个指针，所以我们需要用指针变量来接收

**二维数组与函数形参的关系**

当我们要传递二维数组的时候我们可以用两种方式传递

1. 将二维数组看成一个大的一维数组看待，那么我们的形参需要定义成一个指针变量，比如： int \*p,然后我们传递的数组需要把这个数组的初始位置传递过来，也就是说&a[ 0 ][ 0 ]、\*a····然后我们再传一个M\*N（纵坐标乘横坐标）
2. 我们将它看成二维数组的时候需要像以前说的那样定义一个二维数组指针，比如说int (\*p)[N]（N是已知的列数），，然后我们再把它的行和列传过来，也就是这么写（a,M,N）这样就可以了

**函数与指针的关系**

函数和指针最容易分不清的有以下三个东西

**函数指针：**

就是一个指针，指向的是一个函数可以如下定义

int (\*p)(需要指向的函数有多少个形参这就多少个形参)

赋值的时候直接赋值就可以了，然后用的时候和函数名一样的用法无需加“\*”

如：一个函数名是a，函数指针是p

p = a;a无需加上“&”

**指针函数：**

这是一个函数，只不过它的返回值是指针而已，比如说我需要筛选一个数组，我们需要把这个数组的起始位置返回出来让我们做下一步操作的时候就需要用到数组指针，如：

一维数组：

int \*a(int \*p)

{

`````

return p;

}

二维数组：

int \*a(int (\*p)[N])

{

`````

return \*p;

}

所以我们也可以看出来，返回int型的就叫整型函数，返回char型的就叫字符型函数

**函数指针数组：**

这是一个数组，里面装的全是函数，

我们可以这么定义

int (\*a[N])( 需要指向的函数有多少个形参这就多少个形参)

所以我们看出来，这个数组只能装同一个类型的函数

所以我们运行同类型的函数的时候可以像输出数组一样利用for循环来运行函数

int add(int a,int b)

{

return a > b ? a : b;

}

int sub(int a,int b)

{

return a - b;

}

int (\*b[2])(int,int);

b[0] = add;

b[1] = sub;

for(l = 0;l < 2;l++)

{

res = b[l](4,5);

printf("%d\n",res);

}

还有就是混合的定义：

举个例子：

指向指针函数的函数指针数组的定义：

因为是指向指针函数，所以我们的返回值是指针，所以把之前定义的函数指针数组最开始的int改成int \*就可以了

int \*(\*a[N])(int)

结构体

结构体的类型描述是不占任何空间的

struct 名字

{

}**；**这样是类型的描述

结构体定义在函数外

所以struct 名字其实是一个数据类型跟int是差不多的，它也可以定义结构体变量

只有在定义了一个结构体的时候才会占用内存空间，比如说：

struct a{

int a;

};

struct a b = {1};

这时候b是会占用内存空间的

指针指向结构体时调用成员用“->”,正常的结构体调用成员用“.”

还有就是结构体数组

其实是跟正常的数组是一样的：

struct a arr[ 2 ] = {{ 1 } , { 2 }};

因为这个数组中有两个成员，所以定义一个结构体数组就相当于定义了很多个结构体

其中数组中第一个大括号中是第一个结构体的，然后一个接一个的是接下来的结构体

这时候我们运行结构体其实就可以用for循环了

比如上面的数组打印

for( i = 1;I < sizeof(a)/sizeof(\*a);i++)

printf(“%d %d \n”,arr[ 0 ].a,arr[ 1 ].a);

结构体也可以嵌套

比如说有两个结构体

struct a{

int a;

};

sturct b{

struct a c;

int b;

};

struct b d；

这样结构体b中嵌套了一个结构体a，调用时可以这么调用 d.c.a

如果指针p指向d那么

\*p = &d;

p -> c.a这样就可以了，如果c也被一个指针指向了我们可以这么写

p -> q ->a

当然，a结构体也可以写在b的里面

sturct b{

struct a{

int a;

}c;

int b;

};

这样也是可以的

结构体的空间大小不是按定义的数据类型的数目累加的

它是按地址对齐来分配的空间，比如：

struct a {

int a;

char ch;

float f;

};

这样的结构体的大小如果只是按照数据类型累加的话应该是9，可是我们sizeof一下却发现却是12个字节，这就说明结构体的大小并不是累加的，是按照地址对齐来排序的



从这张图可以看出来我们的地址对齐是怎么回事了地址对齐的意思就是地址号除以这个类型所占的字节大小，整除就可以放，不能整除就需要到下面的地址空间去所以这样我们就看出来了，int是4字节，0能整除，所以int的那个变量放到了0～3地址空间去，char型是1个字节4可以整除1所以定义了char型变量放到4中，剩下一个float，4个字节，5,6,7都不能整除4,到了8才可以，所以float类型的变量存到了8～11中 这样正好是占了12个字节

如果我们在float后面再定义一个char型理论上是13个字节，可是实际上却输出了16个字节，这是因为后面那个char型首先存入到了13的地址中，但是后面的14 15地址都是一个不能被这个类型整除的数，所以系统就直接把这两个地址加入到结构体当中了，所以就成为了16个字节

**补充，结构体的字节对齐也可以这么理解，找到里面字节数最大的数据类型 ，比如说上面的结构体最大就是4个字节，也就是4个字节对齐，也就是说每个成员都是要占用4个字节没用到的自动填充，所以三个成员就是12个字节，4个就是16个字节**

**结构体在函数中的传参**

我们在用函数传递结构体时需要注意一下

当我们正常用结构体传递的时候

struct a{

int a;

int b;

};

struct a b;

hanshu(a);

当我们这么传递参数的时候其实是相当于传递了b.a和b.b，这样很浪费内存，我们可以用指针传递

\*p = &b;

hanshu(struct a \*p)

hanshu(p)就可以了

结构体也可以不用名字，但是定义这种结构体之后必须要把变量都定义好

比如：

struct {

int a;

int b;

}a = { , , },b{ , , },\*p;

当我们用两个指针指向两个相同类型的结构体的时候 \*p 和\*q

当我们需要其中的一个结构体给另外的一个结构体赋值的时候我们直接就 \*p = \*q;就可以了

因为\*p和\*q都是相当于取这个指针当前指向的内容，这两个指针都是指向一个结构体，所以取值就是相当于把结构体中全部的值都取出来，又因为是两个相同类型的结构体，所以可以赋值。

共用体，共用体的定义是跟结构体是差不多的只不过就是把struct换成了union而已

但是它们的作用还是有所不同的，结构体定义的时候是要把所有的变量都要分配空间的，而共用体是所有成员都共用同一个空间，也就是说用的时候只能用一个成员，而结构体可以用多个

共用体的产生是因为以前硬件条件不足，空间不够，所以用共用体

共用体的存储特点是以里面的数据类型所占空间最大的字节数分配的，比如：

union a {

int a;

float b;

char c;

double d;

}b；

这时候共用体b所占的字节数就是double类型的大小 8个字节

但是也别以为不是很占空间，如果里面嵌套了一个结构体那么所占的空间就大了

当我们给其中一个赋值的时候需要注意

b. d = 1.23456;

我们这时候输出b. d时候是正常输出，但是这时候如果输出a就会出现一个不可预料的结果

因为现在这个8个字节的空间存的是1.23456这个值，而a只是占一个字节，输出只是输出了1.23456中的前一个字节，所以结果是不可预料的，所以用共用体的时候这个空间现在是哪个值就用哪个值，用别的成员是没有任何意义的

其它的共用体特性是跟结构体是差不多的，比如指针，函数传参，数组都是很像的

传参不要以为共用体不是很占资源，上面也说了，如果嵌套一个结构体就大了，所以传参也用指针

**枚举类型**

枚举类型的类型名字是enum，它的定义形式跟定义结构体是差不多的，有一个区别是枚举类型中定义的变量都是整型，也就是说不需要写数据类型，比如说：

enum a{

a,

b,

c，

d，

e

};

而且如果不赋值的话那么就是a=0，b=1，c=2，依次排下去，也是不占用内存的

我们在程序中直接调用就可以了，如果c赋值是1的话，a，b的值是不变的，然后从c赋值的地方开始重新依次排下去

枚举类型也可以用作define的一部分，因为如果我们预处理调试的时候define看到的是数字，而这个枚举类型看到的还是字母，调试会方便一点

void \*和函数指针进行传递参数是属于未定义行为，除了函数指针之外viod \*类型可以赋值给任何的一个指针

如果我们调用一个不是我们自己写的函数是系统C库封装好的函数但是我们没有include这个函数所在的头文件，系统就会默认认为它的返回值为整型

动态内存分配

动态内存分配是在堆上开辟了一段内存空间来方便我们使用，这种方法的思路就是使用的时候才给分配一段空间，用完再把这段内存释放

这样就可以让数组长度可变，节省内存空间

void \*malloc(size\_t size);

void free(void \*ptr);

void \*calloc(size\_t nmemb, size\_t size);

void \*realloc(void \*ptr, size\_t size);

malloc函数返回一个指针，输入的是需要分配的空间的字节的大小，可以使用sizeof函数来计算大小，最后函数返回的地址就是已经开辟好的空间的首地址

然后在使用完之后别忘了用free函数进行释放空间

calloc在内存的[动态存储](http://baike.baidu.com/view/589359.htm)区中分配n个长度为size的连续空间，函数返回一个指向分配起始地址的[指针](http://baike.baidu.com/view/159417.htm)；如果分配不成功，返回NULL。

realloc将原来的分配空间重新开辟空间，ptr代表重新开辟的空间的首地址，size代表的是大小

**注意，上面的这些函数都是在stdlib.h头文件中的，如果不包含就会被认为返回值是int型从而报错**

大致的使用方法就是这样，但是在我们使用的时候很有可能出现下面的错误出现内存分配不足

比如说我们需要用到一个子函数来开辟空间

void func(int \*p, int num)

{

p = malloc(num);

}

int main()

{

int \*p = NULL;

int num = 10;

func(p, num);

free(p);

exit(0);

}

如果按照这么写的话那么就会出现内存分配错误，分配的空间并没有给到main函数的p上

因为我们看，在func函数中分配的空间的地址是先给到形参中传递的指针p，此时指针p的地址为NULL这样一给形参p赋值，但是main函数的p却没有被赋值，p的地址却还是NULL，但是为什么没报错呢，因为free函数如果释放的是NULL，那么free函数是什么都不干的，但是并不认为是一个错误

我们有两种方法改进

1、

void func(int \*\*p, int num)

{

\*p = malloc(num);

}

int main()

{

int \*p = NULL;

int num = 10;

func(&p, num);

free(p);

exit(0);

}

这样就是把指针p本身的地址传过去了，形参用一个二级指针接着，这样返回的地址值直接操作了p的地址

void \*func(int \*p, int num)

{

p = malloc(num);

return p;

}

int main()

{

int \*p = NULL;

int num = 10;

p = func(p, num);

free(p);

exit(0);

}

这样就是把这个指针返回了

说一下free函数，free函数并不是把要释放的那段空间中的数据给清除掉，也不是把指向它的指针清掉，它只是把指向这个地方的指针权限给除去掉了，也就是说这个指针现在不能访问那段空间了，如果不给这个指针一个新的地址直接用的话那就成为了一个名副其实的野指针，如想再用需要给它重新赋值

**typedef的一些注意的**

typedef 是将一个数据类型进行定义成另外的一个名字的比如说

typedef int INT

这样INT就和int是一个效果了

然而typedef跟#define是不一样的，如果我们

#define int \*A

和typedef int \*A

我们A p，q这两个的效果是完全不同的，define定义的是一个\*p和一个整型变量q，typedef就定义了两个指针变量，这是因为define只是将这个int \*替换成了一个A而已，而typedef是将A变成了一个int \*的数据类型，这个是；两者的本质差别

我们也可以用typedef定义结构体，在以后的程序中经常会用到typedef定义的结构体，这样我们定义的结构体也可以不用写名字了

比如

typedef struct{

int a;

}P , \*Q;

这样就可以定义这个结构体以及结构体指针了

也可以用typedef定义函数或者是函数指针

Typrdef int P(int) ，这样就定义了一个返回值以及形参都是int的函数

我们定义的时候P a; 我们可以这么看，其实就是把P和a调换一下

# Makefile

如果makefile 和Makefile同时存在的时候执行make是执行的makefile文件

RM跟rm –f的意义是一样的

$^代表依赖文件 $@代表目标文件

%.o所有的.o文件 %.c所有的.c文件