**函数的定义**

数据类型 函数名 ([形式参数说明表])

数据类型可以是基本数据类型也可以是构造类型

函数名与前面说的变量名没啥大区别，由字母、数字、下划线组成，不能以数字开头

形式参数说明表简单来说有两种形式，一种是固定参数，像我们常见到的那些都是，还有一种是不固定参数，非常典型的就是printf，函数原型int printf(const char \*fomat, …);

int main()

{

printf(“Hello!\n”);

return 0;

}

用echo $?来查看上一条命令的执行情况，如果返回0表示上一条命令执行成功，非0就表示执行失败

该main函数的返回值为0，如果把return 0;注释掉，那就返回7，这个7是最后一条执行的printf语句的返回值，printf的返回值是它打印的有效字符个数，它把最后一条执行语句的返回值作为当前进程的返回值状态，一个进程的返回状态是给它的父进程看的，谁创建了这个进程，谁就是它的父进程，在这里是shell，但换个环境就不一定是shell了

main函数里面看起来没有参数，可以写成int main(void)，但它仅仅表示不需要传参，但其实它是有参数的

int main(int argc, char \*argv[])

{

printf(“argc = %d\n”, argc);

//printf(“Hello!\n”);

return 0;

}

argc是一个计数器，来帮你计算从终端上到底传了多少个参数过来，



./main就是一个参数



./main abc 123 hello就是4个参数



同理，这里有7个参数



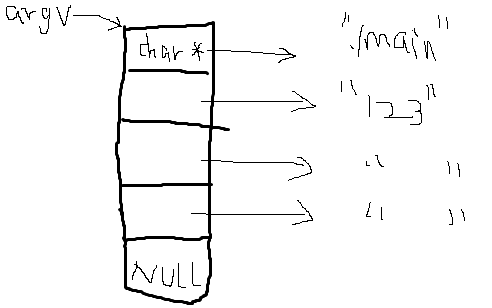
etc下的以a开头的conf文件有3个

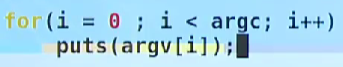


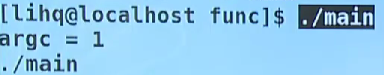
当前的shell有解析通配符的功能，所以main接收的参数不是2个而是4个

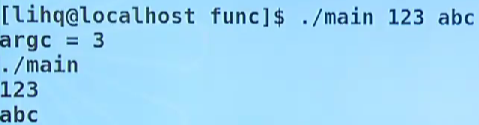
argv是一个列表，用来保存你刚才在main函数中所传递过来的所有参数，看到它后面的[]你就知道它是一个数组，argv是数组名，char \*是数组当中每个元素的类型，是字符指针，所以argv的本质是字符指针数组的首地址

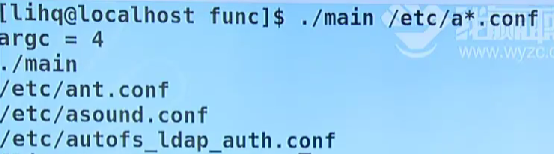
至于它里面有多少个参数，取决于你的argc是多少



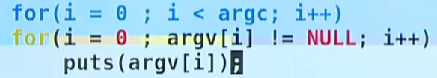








有个细节，前面也说过，argv最后会有一个空指针来表示当前存储的结束，所以理论上来讲，我在输出所有argv的时候，其实可以不依赖于argc，像下面这样写，效果一样

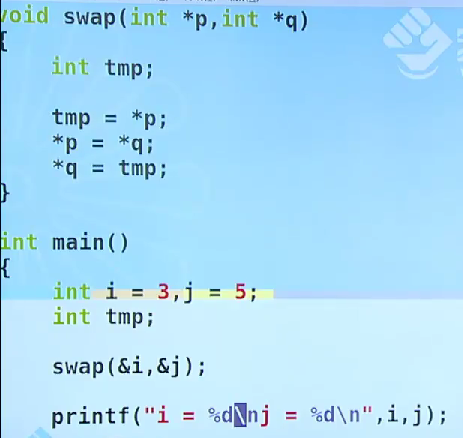


如果主调函数的定义在前，被调函数的定义在后，一定要在主调函数定义前加上被调函数的声明

**函数的传参**

值传递

地址传递



**函数的调用**

嵌套调用

a调b，b调c…

模块化编程的思想，函数内部实现的功能越简单，复用性越强

递归调用

一个函数直接或间接地嵌套自身，这样的方法就叫做递归，是嵌套调用的特例

递归的三个条件：

1.一层一层调用或递进的条件

2.某一个已知条件用作递归的退出条件

3.还需要判断当前条件是否成立

即使递归再复杂，只要掌握了上述3个条件，那就没啥问题

压栈出栈

递归特点：

1.面试常考，因为它非常能够检验出来一个程序员的编程功底有多少

2.实际生产环境中，其实在使用它时非常小心，有的时候，我们希望来制定一个递归的上限次数，也就是当递归达到某一个次数，我还没得到我要的一个结果，那么我宁可退出，得不到结果，也不能让栈破裂，让程序出错

**学会去模仿着写，慢慢地，这些东西都会变成你的**

**函数与数组**

函数与一维数组

数组名作为参数进行传递时会退化成一个指针，在接收位置也应该用一个指针来接收，当然你也可以写成用数组来接收的形式，只不过它依然是个指针，比如，一个接收整型数组的函数，定义时，可以写成void print\_arr(int \*p)，也可以写成void print\_arr(int p[])，在这里p[]依然是一个指针而不是数组，指针指向你传过来的数组首地址

回顾，int main(int argc, char \*argv[])，很多参考书上不写char \*argv[]，而写成char \*\*argv，不要觉得它是个二级指针，它不是的，它的本质更接近于一个char \*数组的起始位置，写成char \*argv[]更有助于理解其本质，在形参的位置上一个[]就等价于\*

函数与二维数组

跟一维的差不多，一个接收整型二维数组的函数，定义时，可以写成void print\_arr(int (\*p)[N])，也可以写成void print\_arr(int p[][N])，N是一个宏哈，代表数组的列数，在形参位置的p[][N]看起来像个数组，但实际上依然是一个指针而不是数组，在这里是数组指针，还是那句话，一个[]等价于一个\*，但第二个[N]不能省略，因为这个指针指向的到底是一个多长的小一维数组，长度要给它，这个长度不能省略

函数与字符数组

**函数与指针**

指针函数

定义为，返回值 \* 函数名（形参），如int \* fun(int)

还是一个函数，但其返回值是一个指针

函数指针

定义为，类型 （\*指针名）（形参），如int (\*p)(int)

是一个指针，指向函数

int add(int a, int b)

{

}

int main()

{

int ret;

int (\*p)(int, int) //这是一个函数指针，它的类型应该和它指向的函数的类型是匹配的

p = add;

ret = p(a, b);

}

上面p = add;可以，p = &add;也可以，因为你的add函数名本身就是一个地址，所以地址符可以省略，同样的原本通过指针变量引用要加\*，也就是ret = \*p(a, b);，这个\*也可以省略，变成ret = p(a, b);

一个指向函数的指针，指针指向某个函数的入口地址，之后通过指针来传参数，就如同通过函数名来传参数，是一样的

函数指针数组

定义为，类型 （\*数组名[下标]）（形参），如int (\*arr[N])(int)

小测试，写出一个指向指针函数的函数指针数组

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

前面的数组，指针，函数都理解到位了，写出来不难

当需要传参的时候，我传整型数过去，你用整型指针来接收，我传数组名过去，你用对应类型的指针来接收，我如果传一个函数名过去，你用什么来接收呢，实际上，函数名不就是一个入口地址嘛，依然是要用指向函数的指针来接收，很多函数都这么封装的

比如说qsort快速排序，函数原型

void qsort(void \*base, size\_t nmemb, size\_t, size, int(\*compar)(const void \*, const void \*));

最后一个参数显然是一个指向函数的指针，它所指向的函数必须是参数是两个const void \*，返回值是int

再比如说pthread\_create，函数原型

Int pthread\_create(pthread\_t \*thread, const pthread\_attr\_t \*attr, void \*(\*start\_routine) (void \*), void \*attr)

第三个参数依然是一个指向函数的指针，所指向的函数参数是void \*，返回值void \*也是一个指针，所以可以把它称作指向指针函数的函数指针，当我想调pthread\_create这个函数时，第三个参数我必须给它一个指针函数的入口地址，所以它用一个指向指针函数的函数指针来接收，所以为什么会出现指向各种各样函数的指针，就是为了传参的要求，像这些函数叫回调函数

**数组、指针、函数，这三座C语言中的大山，拿下**