C专家编程

-原书集锦解读，从小工到专家

rock.zhou@nokia-sbell.com

课程介绍

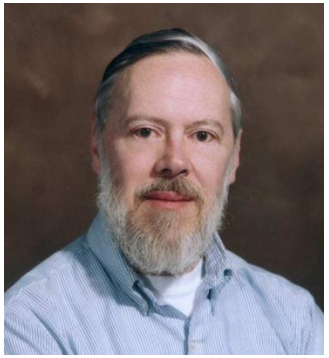
《C专家编程》一书展示了最优秀的C程序员所使用的编码技巧，对C的历史，语言特性，声明，数组，指针，链接等问题进行了细致的讲解，可以帮助有一定经验的C程序员迅速获取一些专家观点和技巧。

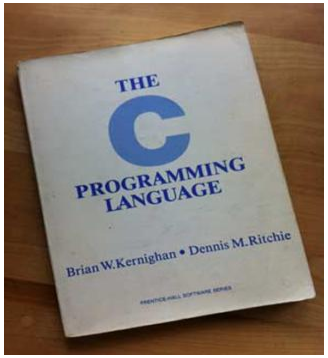
即使你已经读过《C陷阱和缺陷》，还是应该好好读读这本书。

本课程以原书为基础，实例探讨那些需要多年实践才能得到的知识和经验。

如果你已经读过《C专家编程》，那么应该好好读读《UNIX环境高级编程》

第1章-- C: 穿越时空的迷雾





http://en.cppreference.com/w/c/language/history
–C语言的标准化演进历史

https://www.bell-labs.com/usr/dmr/www/chist.html
– Dennis Ritchie自述C语言的发展历史

设计哲学：小即是美。事物发展都有个过程，由简入繁，不能一开始就想得太复杂。
– Multics工程就是因此而失败。

C语言的许多特性是为了方便编译器设计者而建立的
– 比如数组下标从0而不是1开始 。
– 比如register关键字(极少使用)，这个设计可以说是一个失误，如果让编译器在使用各个变量时自动处理寄存器的分配工作，显然比一经声明就把这类变量在生命周期内始终保留在寄存器里要好，使用register关键字，简化了编译器，却把包袱丢给了程序员。

C本身不具备的一些功能通过标准库和C预处理器来实现。
– 预处理器宏只应该适量使用(会有一些副作用)，宏名应该大写这样便很容易与函数调用区分开来。
– 预处理器pragma指定编译器完成一些特殊的特性 (**举例**)

容易混淆的 const (**举例**)
– const关键字原先如果命名为readonly就好了☺
const int *p;是指p指向的内容不能改变，即：*p = 30这样是错误，但p本身是可以改变的。
const最有用之处就是用它来限定函数的形参，这样该函数将不能修改实参指针所指的数据。

第2章-- 这不是Bug，而是语言特性

C诡异离奇，缺陷重重，却获得了巨大的成功。

-- Dennis Ritchie

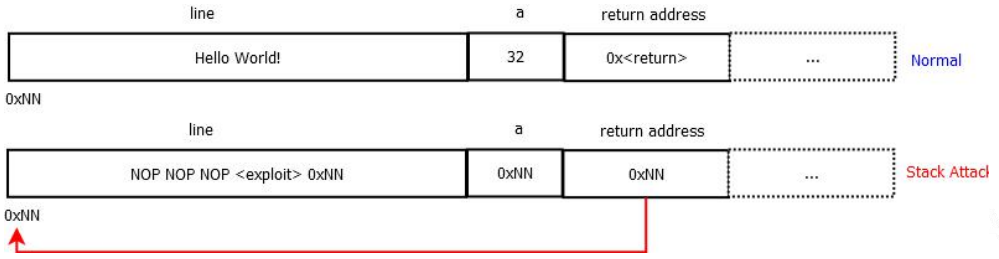
多做之过(不该存在的特性)：
- 把fall through作为switch的缺省行为是个失误，在压倒多数的情况下，你不希望这个缺省的行为而不得不加上一条额外的break语句来改变它(举例)
- 相邻的字符串自动合并成一个字符串(举例)
- 函数在缺省情况下全局可见，大多数情况下应该缺省采用有限可见性

误做之过(不适当的特性):
- C语言中符号"重载":
  static 在函数内部，表示该变量的值在各个调用间一直保持延续性; 在函数这一级，表示该函数只对本文件可见。
  extern用于函数定义表示全局可见（属于冗余）; 用于变量，表示它在其他地方定义。

- C语言中运算符优先级存在的问题:
  .优先级高于*, *p.f表示*(p.f)； 函数()高于*； ==和!=高于位运算符(val & mask != 0)表示val & (mask != 0)；
  ==和!=高于赋值符，c = getchar() != EOF表示c = (getchar() != EOF)；算数运算符高于移位运算符 msb<<4 + lsb表示msb<<(4+lsb)；逗号最低。
  有些专家建议在C语言中记牢两个优先级就够了：乘除先于加减，在涉及其他的操作符时一律加括号☺
- 计算的次序x = f() + g() * h()未定义，是想让编译器充分利用自身架构的特点，或者充分利用存储于寄存器的值。

- C标准库中的不安全性
  gets(char *s)，不检查缓冲区的空间，攻击者向针对堆栈缓冲区溢出漏洞的应用程序发送包含一段恶意代码的数据，还在数据中包含这段恶意代码的初始内存地址，应用程序将攻击者的数据写入缓冲区中，并且不管缓冲区大小持续写入数据，直到恶意代码的地址重写函数的返回地址, 示例如下:

```
void trouble()
{
    int a = 32;
    char line[128];
    gets(line);
}
```



fgets(char *s, int n, FILE *stream)可以对读入的字符数设置一个上限, 对缓冲大小进行限制的方式，更为安全。

第2章-- 这不是Bug，而是语言特性 (续)

少做之过(未能提供的特性):

在C语言中很容易写出一些能够轻松通过编译，但是在运行时产生一堆垃圾代码的例子

(举例)

这个例子中buffer是函数的局部变量，当控制流离开声明局部变量的范围时，局部变量自动失效，这就意味着如果试图返回一个指向局部变量的指针，当函数结束时，由于该变量已经被销毁，谁也不知道这个指针所指向的地址的内容是什么，这取决于堆栈中先前的局部变量位于何处，原先的局部变量地址的内容可能立即覆盖，也可能稍后才被覆盖。

实际上类似的潜在问题编译器并不会直接报error，但是可以通过静态检查工具或者编译器的warning体现出来，因此静态检查不是可有可无的东西，因为由它来寻找潜在的Bug是一笔很划算的投资。

第3章-- 分析C语言的声明

你明白下面的声明(取自telnet程序)的确切意思么?
char * const *(*next)();

理解C语言声明的优先级规则
A 声明从它的名字开始读取，然后按照优先级顺序依次读取。
B 优先级从高到底依次是:
B.1 声明中被括号括起来的那部分
B.2 后缀操作符:
括号()表示一个函数，
方括号[]表示这是一个数组。
B.3 前缀操作符:
*表示指向...的指针
C 如果const和(或)volatile关键字与类型说明符(如int, long等)相邻，它作用于类型说明符；其他情况下const和(或)volatile关键字作用于它左边紧邻的指针*号。

char * const *(*next)();
<用优先级规则解决这个声明>
A 首先看变量名next, 注意到它直接被括号所括住
B.1 所以先把括号里的东西作为一个整体，得出”next 是一个指向…的指针”
B.2 然后考虑括号外边的东西，规则告诉我们优先级较高的是右边的函数括号，所以得出”next 是一个函数指针，指向一个返回…的函数”
B.3 处理前缀操作符*，它表示指向...的指针
C 最后，把char * const解释为指向字符的常量指针

加以概括，这个声明表示”next 是一个指针，它指向一个函数，该函数形参列表为空且返回一个指针，返回的指针类型是一个指向字符串的常量指针”(举例)

合法的声明(举例):
●函数的返回值允许是一个函数指针，如int(*fun())();
●函数的返回值允许是一个数组指针，如int(*fun())[ ];
●数组里面允许有函数指针，如int(*fun[ ])();
●数组里面允许有其他数组，如int[ ][ ];

第3章-- 分析C语言的声明(续)

一般情况下typedef用于简洁地表示指向其他东西的指针。典型的例子是signal()原型的声明:
void (*signal(int sig, void(*func)(int)))(int);
➔ 看一下signal.h (举例)
typedef void(*ptr_to_func)(int);
ptr_to_func signal(int, ptr_to_func);

typedef和宏文本替换之间存在一个关键性的区别：typedef看成是一种彻底的“封装”类型——在它声明后不能再往里面增加别的东西，它和宏的区别体现在两个方面(本质区别在于宏定义是由预处理器处理的，而typedef是由编译器处理的)：

首先，可以用其他类型说明符对宏类型名进行扩展，但对typedef所定义的类型名称不能这样做。
#define peach int
unsigned peach i; /* 没问题 */
typedef int banana;
unsigned banana i; /* 错误！ 非法 */

其次连续几个变量声明中，用typedef定义的类型能够保证声明中所有的变量均为同一种类型，而用#define无法保证
#define int_ptr int *
Int_ptr chalk, cheese;
经过宏扩展变成了int * chalk, cheese; //类型不同
typedef char * char_ptr;
Char_ptr Bentley, Rolls_Royce; //类型相同

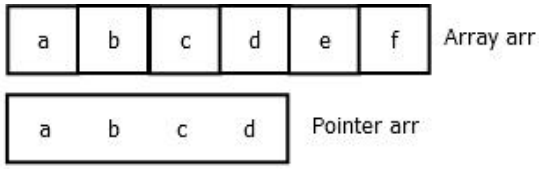
第04章-- 令人震惊的事实：数组和指针并不相同

C编程新手最常听到的说法之一就是”数组和指针是相同的”，不幸的是，这是一种非常危险的说法，并不完全正确。

1. 将数组声明为指针

比如在文件1中定义了数组，在文件2中声明它为指针，运行时直接崩溃。

```
char arr[] = “abcdef”;  //test.c
extern char* arr; // main.c
printf("%s",arr);
```



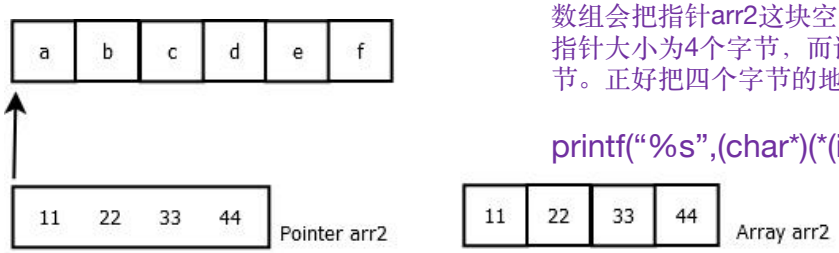
这里把数组arr当成指针arr来用，指针大小是四个字节，里面存储的是一个四字节的地址，所以这里截取了前面四个字节的空间作为指针的空间，把里面存的abcd当成了一个地址0Xabcd，函数会对该地址进行访问，而该地址是未知的，属于非法访问，所以程序崩溃了。

printf("%s",&arr);可以得到正确的值abcdef

2. 将指针声明为数组

我们在文件1中定义了一个字符串指针，在文件2中，我们把这个指针当成了数组来用，运行时不会崩溃(但是结果不正确)。

```
char* arr2 = "abcdef"; //test.c
extern char arr2[ ]; // main.c
printf("%s",arr2);
```



数组会把指针arr2这块空间里的内容作为数组的元素。上面我们说到指针大小为4个字节，而该数组是字符数组，每个元素大小为一个字节。正好把四个字节的地址作为四个数组元素。

printf(“%s”,(char*)(*(int*)arr2)); 可以得到正确的值abcdef

第09章,第10章-- 再论数组和指针

何时数组和指针相同？在实际应用中，数组和指针可以互换的情形要比两者不可互换更为常见。

C语言标准作了如下说明：

1. 表达式中的数组名被编译器当作一个指向该数组第一个元素的指针
2. 下标总是与指针的偏移量相同
3. 在函数参数的声明中，数组名被编译器当作指向该数组第一个元素的指针

规则1和规则2可以合在一起理解，就是对数组下标的引用总是可以写成一个指向数组的起始地址的指针加上偏移量。对数组的引用比如a[i]在编译时总是被编译器改写成*(a+i)的形式，那么对于编译器来说等同于*(i+a)，也就是说你写成i[a]也是合法的，只是很少有人这么使用而已 (举例)

规则3这是出于效率原因的考虑，如果要copy整个数组，无论在时间上还是内存空间上的开销都可能是非常大的。

数组可以作为参数传递给函数，那么反过来可以从函数返回一个数组吗？严格来讲无法直接从函数返回一个数组，但是可以让函数返回一个指向任何数据结构的指针，当然也可以是一个指向数组的指针(举例)

第5章-- 对链接的思考

静态库被链接后库就直接嵌入可执行文件中了，这样就带来了两个弊端：

1.首先就是系统空间被浪费了。这是显而易见的，想象一下，如果多个程序链接了同一个库，则每一个生成的可执行文件就都会有一个库的副本，必然会浪费系统空间。
2.再者，一旦发现了库中有bug或者是需要升级，必须把链接该库的程序找出来，然后全部需要重新编译。

动态库的出现正是为了弥补静态库的弊端。因为动态库是在程序运行时被链接的，所以磁盘上只要保留一份副本，因此节约了磁盘空间。如果发现了bug或要升级也很简单，只要用新的库把原来的替换掉就行了。

建议只使用动态链接。

//

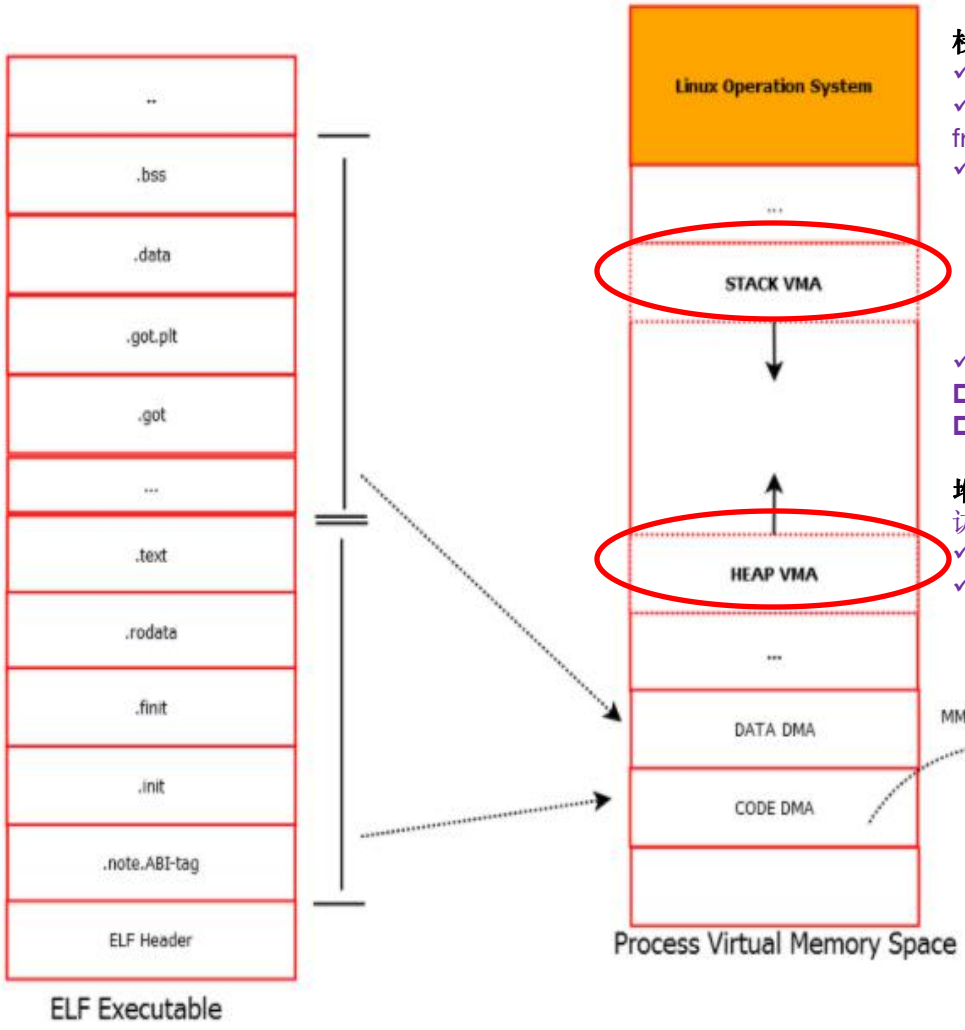
Linux环境下的动态链接对象都是以.so为扩展名的共享对象(Shared Object).

gcc创建动态链接库和使用
创建：gcc -fPIC -shared -o libfruit.so fruit.c
使用：gcc -o test test.c –lfruit

ldd程序可以打印出共享对象的依赖关系即 shared library dependencies

//

第6章-- 运行时数据结构
第7章-- 对内存的思考



运行时数据结构有好几种：数据，堆栈，堆等

static变量保存在数据段，而不是堆栈中

栈有三个主要的用途：
✓堆栈为函数内部声明的局部变量提供存储空间
✓进行函数调用时，堆栈存储与此有关的一些维护信息，称为stack frame.
✓堆栈也可以被用作暂时存储区，比如计算表达式，存储中间结果

注意：编译器的设计者会尽可能地把函数调用过程的内容放到寄存器中，这样可以提高速度，如果在某些场合你需要使用C和汇编的相互调用那么要留心编译或者汇编手册中和函数调用相关的那部分内容

✓使用setjmp和longjmp可以进行跳转，它和goto又有不同(举例)：
- goto语句不能跳出当前的函数
- longjmp只能跳回到曾经到过的地方

堆用于动态分配的存储，在C语言中通过malloc函数获得，并通过指针访问。堆经常会出现的两类问题：
✓释放或改写正在使用的内存
✓未释放不再使用的内存(内存泄露)

段错误的几个直接原因 (举例)
✓解引用一个包含非法值的指针
✓解引用一个空指针
✓未经授权访问
✓用完了堆栈空间
✓...

第5章-- 对链接的思考 第6章-- 运行时数据结构 第7章-- 对内存的思考(续)

关于链接，装载和库相关的知识，大家可以参考以下视频课程
http://edu.51cto.com/course/10863.html

本课程从最简单的Hello World程序谈起，介绍了程序的整体编译过程，然后详细分析了Linux环境下目标文件和可执行文件的内容，接下来探讨了静态链接的过程，动态链接的过程，程序的启动和装载过程，以及运行库等等，值得大家学习和思考

建议阅读资料

- C Traps and Pitfalls - by Andrew Koenig
- Pointers on C - by Kenneth A.Reek
- Secure Programming with Static Analysis - by Brian Chess
- Advanced Programming in the UNIX Environment - by Richard Stevens