2.1.1 LINUX 下 C 代码规范

在深入分析标准 C 的各个语法要点之前，必须先来澄清 LINUX 环境下 C 代码的风格和规范，良好的编码习惯和一开始对代码可维护性的关注将会使工作更加轻松和高效，就像学习语言时的音标，学习乐器时的站姿或坐姿，学习烹饪时菜肴的外观等，这些看似可有可无的基本技术要领实际上会贯穿整个职业生涯，乃至直接影响最后达到的高度。

编码风格不能看成是个人的东西，因为虽然每个人的喜好并不一致，但是如果我们全凭自己的意愿来编写代码，那么看的人就会比较痛苦，将不同风格的代码杂糅在一起形成一个大项目软件，看起来也不那么舒服，因此我们需要一种被大众接受的统一规范的编码风格，来约束程序员在编写代码时的格式要求，提高程序的可读性和维护性。在 LINUX 环境下，C 代码遵循一定的书写习惯，这些习惯可能跟别的环境不同，原则上它们没有对错之分，只是习惯不同而已。而且这些我们将要介绍的不成文的约定俗成的代码风格，是写好 LINUX 下 C 程序的前提。其中的语句和语法在下面的章节逐一介绍，在此间碰到不熟悉的语句语法只需悄然掠过即可，不必纠缠于它们，我们将学习重心放在代码编写风格上。

1、 标识符的命名

标识符指的是程序中的函数名和变量名，C 语言中对标识符命名的限制有：

A 只能包含数字、字母以及下划线，不能包含其他任何特殊字符。

B 只能以字母或者下划线开头。

C 不能跟系统已有的关键字重名，也不能跟本命名空间（namespace）中具有相同作用域的其他标识符重名。

事实上对于一个大型程序而言，标识符的命名是个大问题，原因之一是我们经常会使用相同的单词来命名具有某个确切含义的变量，比如我们通常用 count 或者 counter 来定义计数器，如果在不同的文件当中存在这样的重名的变量，那编译的时候就会有问题。解决这种问题的方法之一可能是用更多的信息来命名标识符，比如除了写出其作用之外，再加上该变量的长度、类型等信息，即饱受抨击的所谓的匈牙利命名法，这种包含了过多信息的标识符虽然在一定程度上能让冲突减少，但显然理解起来很啰嗦。

以上问题在 C++中尤为明显，因为 C++就是为解决大型软件开发而产生的，期间可能会整合各种不同公司的软件包和类库，标识符名字冲突的矛盾便凸显了出来，因此 C++有专门的机制---namespace 来解决此问题。C 语言也有自己的 namespace，即所谓的命名空间，但是相对而言要简单得多，实际上在一个 C 程序中，所有的标识符处在 4 个命名空间的其中一个，这四个命名空间是：

1、所有的结构体、联合体、枚举列表的标签名。

2、每一个结构体、联合体内部的成员列表。

3、goto 语句的标签名。

4、其他。

举一个例子来综合说明上面的情况：

```
vincent@ubuntu:~/ch02/2.1$ cat namespace.c -n
     1  struct apple // 1，结构体的标签
     2  {
     3      int apple; // 2，结构体的内部成员
     4  };
```

```
 5
 6  int main(void)
 7  {
 8      struct apple sweet_fruit;
 9      sweet_fruit.apple = 100;
10
11      double apple = 3.14; // 3，普通变量
12
13  apple: // 4，goto 的标签
14      if(apple == 0)
15          goto apple;
16
17      return 0;
18  }
```

上面的例子说明了一个问题：在相同的命名空间里标识符不能冲突，但是在不同的命名空间中名字可以相同。就像一个教室不能有两个张三，否则叫起来会混乱，但是不同的教室可以有两个人都叫张三。就像上面代码所示，有四个标识符都叫 apple，但分别在不同的命名空间中，因此不冲突。

我们在命名标识符的时候，除了不能跟关键字、相同命名空间里的其他标识符重复，只能包含数字、字母以及下划线且不能以数字开头这几点语法上的“硬性指标”之外，还需要符合 LINUX 下 C 编程的规范，比如一般而言，在 LINUX 中标识符中如果包含有多个单词，则不同的单词用小写字母开头且之间用下划线隔开，比如 apple_tree，而不是 AppleTree，当然后者也没有任何不妥，两者都是为了使标识符更具有可读性，只不过 LINUX 的习惯是前者罢了。另外，命名标识符要符合以下两个原则：

1、 max-info：即尽可能地包含更多的信息。
2、 min-length：即尽可能地让长度最短。

显然上述两点是矛盾的互相制衡的，一个标识符所包含的信息越多其长度势必越长，因此我们需要在两者之间找一个权衡，避免出现在一个程序中所有的变量甚至是函数都是单个字母的标识符，那样看起来会很郁闷的，因为你根本不知道那些字母究竟代表什么含义。在可能的情形下，我们要使用一目了然、顾名思义的标识符命名，当然那些用来控制循环次数的变量简单地命名为 i，j，k 就可以了。

2、缩进

缩进从语法角度上讲是无关紧要的，但是对于程序的可读性和可维护性而言，无疑是重中之重，没有人会看一篇写得跟面条一样的程序的，将一个逻辑块代码用缩进的方式让人一目了然非常重要，不管是对于大型的软件，还是小测试程序，逻辑清晰格式悦目的代码永远受欢迎。那什么时候需要缩进呢？很简单，每一个代码块都需要缩进，比如函数体，循环结构（包括 while 循环，do…while 循环和 for 循环），分支语句等，比如：

```
vincent@ubuntu:~/ch02/2.1$ cat indentation.c -n
     1  int main(void)
     2  {
```

```
 3      int i;
 4      int j;
 5      int k = 1;
 6
 7      while(k <= 100)
 8      {
 9          i = k;
10          j = k;
11          k++;
12      }
13
14      return 0;
15  }
```

缩进时，最好用 8 个空格缩进，避免用 4 个空格甚至是 2 个空格（或许还有外星人用诸如 3 个空格！），8 个空格对于阅读大型软件而言非常有用，它使你的眼睛不会容易疲劳，缩进的空格数过少，代码就感觉是揉在一起的，看久了就会降低我们的生活品质。也许有人会抱怨说 8 个空格太多了，如果缩进超过 4 层可能几乎都要写在行末了，但是要反过来想一想，缩进超过 4 层的代码，是否嵌套太深了，是否需要修改一下代码的逻辑是指不至于阅读起来那么困难。

还有，如果你厌烦缩进时要敲很多的空格，你当然可以用制表符<Tab>来代替，但是代价是你的代码最好不要在多款不同的编辑器中来回编辑，因为不同的编辑器的制表符所代表的空格数可能不一致，这时切换编辑器就有可能会使代码的格式紊乱。你有几种办法来避免这种尴尬的局面，一是统一用宽度恒定的空格缩进；二是不要在不同的编辑器中倒腾你的程序；三是在切换你的编辑器的时候设置编辑器的选项，使它们的制表符所代表的空格数相等。

3、空格和空行

空格和空行也是提高程序可读性的很重要的一方面，谁都不愿意阅读挤在一起的代码，因此适当的空格和空行能让程序看起来逻辑更加清晰。

什么时候需要用空格和空行将程序中的不同标识符和不同代码段隔开，并没有一个严格的统一的说法，但是有一些一般做法我们可以借鉴：

在赋值、比较、逻辑操作等运算式中，用空格将操作数隔开，在标点符号后面也用空格隔开（符合英文的书写规范），比如：

a = b 要比 a=b 好。

if((a < 2) && (b > 100)) 要比 if((a<2)&&(b>100)) 好。

另外，不同的逻辑块代码之间最好用空行适当地隔开，不能挤到一起。比如：

```
vincent@ubuntu:~/ch02/2.1$ cat empty_line.c -n
 1  #include <stdio.h>
 2  #include <stdlib.h>
 3
 4  int main(void)
 5  {
 6      int count = 100;
```

```
 7       int bytes_read, bytes_write;
 8
 9       while(count > 0)
10       {
11           printf("%d\n", count--);
12       }
13
14       return 0;
15   }
```

可以看到：第 3、8、13 行是空行，他们介于头文件与主函数、变量定义和循环体、函数体与返回值之间，由于这些逻辑块确实没有太大的逻辑联系，因此最好使用空行隔开，使得整个程序一目了然。总之，空格也好空行也罢，目的只有一个，那就是让你的代码看起来舒服，不要挤在一起就行了。

4、括号

函数体和循环结构、分支结构等代码块需要用到花括号将其代码括起来，对于函数而言，在 LINUX 编码风格中，左右花括号分别占用一行，在其他的代码块中，左花括号可以放在上一行的最右边，右花括号单独占一行（推荐像函数一样左右括号各自单独占一行，使得代码的风格更具一致性）。比如：

```
vincent@ubuntu:~/ch02/2.1$ cat bracket.c -n
 1  int main(void)
 2  {
 3      if(1)
 4      {
 5          // some statements
 6      }
 7
 8      while(1)
 9      {
10          // some statements
11      }
12
13      return 0;
14  }
```

5、注释

在我们编写程序的时候，如果遇到比较复杂的情形，我们可以在代码中用自然语言添加一些内容，来辅助我们自己和将来要阅读该程序的人员更好地理解程序。首先介绍 C 语言中写注释的几种方法：

第一，用形如/* … … */的方式书写注释，被这一对正斜杠和星号包含的内容将被视为注释语句，这是 C 语言注释语句的传统编写方式，它有两个特点，一是不能嵌套，二是可以注释多行。比如：

```
/*
        this is one line of comment
        this is another comment
*/
```

第二，用 C++风格的注释// some comment，来注释掉一行，比如：

```
//  comment1
//  comment2
```

在程序中，如果有许多地方需要输出调试信息，而且需要经常打开和关闭调试，如果不想把代码写得很难看，建议可以用条件编译语句来注释，例如：

```
vincent@ubuntu:~/ch02/2.1$ cat debug.c -n
 1  #include <stdio.h>
 2
 3  int main(void)
 4  {
 5      int a;
 6      a = 100;
 7      printf("a = %d\n", a);
 8
 9  #ifdef DEBUG
10      printf("this is a debuging message.\n");
11  #endif
12
13      return 0;
14  }
```

这样，第 10 行调试信息可以很方便地在编译的时候，通过一个-D 选项来打开和关闭。

打开调试信息：

```
vincent@ubuntu:~/ch02/2.1$ gcc debug.c -o debug -DDEBUG
```

关闭调试信息：

```
vincent@ubuntu:~/ch02/2.1$ gcc debug.c -o debug
```

注释的作用相当于商品的《使用说明书》，比如你买了一台微波炉，里面会有一册使用说明书，具体告诉你它的功能及如何使用它，但是它不会告诉你微波炉的详细制作原理及工艺，你也并不关心此类信息，同样的道理，你需要在一段比较复杂的需要说明的代码中添加说明其功能的注释，这些注释讲清楚该段程序能够做什么（what）即可，不需要阐述其原理，即不需要说明它是如何做到的（how）。