C语言的语法风格与代码书写规范指南

这篇文章主要介绍了C语言的语法风格与代码书写规范指南,文中主张了一些诸如变量和结构体的命名规范等细节方面的问题,需要的朋友可以参考下

C代码：

```
#include <stdio.h>
int main(void)
{
 printf("That is Right Style\n");
 return 0;
}
```

在一个标准的C语言程序中，最特殊的莫过于main函数了，而说到底它就是一个函数而已，仅仅因为它地位特殊拥有第一执行权力，换句话说，难道因为一个人是省长它就不是人类了？所以函数该有的它都应该有，那么函数还有什么呢？

函数大体上分为内联函数(C99)(内联函数并非C++专属，C语言亦有，具体见前方链接)和非内联的普通函数，它们之间有一个很明显的特点(一般情况下)，那就是不写原型直接在main函数上方定义，即使不加'inline'关键字，也能被编译器默认为内联函数，但之后带来的某些并发问题就不是编译器考虑的了。

普通函数正确的形式应该为声明与定义分离，声明就是一个函数原型，函数原型应该有一个函数名字，一个参数列表，一个返回值类型和一个分号。定义就是函数的内在，花括号内的就是函数的定义:

```
//...
int function(int arg_1, float arg_2);
//...
int main(int argc, char* argv[])
{
 int output = function(11, 22.0);
 printf("%d\n",output);
 return 0;
}

int function(int arg_1, float arg_2)
{
 int return_value = arg_1;
 float temp_float = arg_2;
 return return_value;
}
```

依上所述，当非必要时，在自己编写函数的时候请注意在开头(main函数之前)写上你的函数的原型，并且在末尾(main函数之后)写上你的函数定义，这是一个很好的习惯以及规范。所谓代码整洁之道，就是如此。

函数的另一种分类是，有返回值和无返回值，返回值的类型可以是内建(build-in)的也可以是自己定义的(struct, union之类)，无返回值则是void。

为什么我们十分谴责void main()这种写法？因为这完全是中国式教育延伸出来的谭式写法，main函数的返回值看似无用，实际上是由操作系统接收，在Windows操作系统下也许无甚"大碍"(实际上有),当你使用Linux的过程中你会清晰的发现一个C语言程序的main返回值关系到一个系统是否能正常，高效的运行，这里稍微提一句，0在Linux程序管道通信间代表着无错可行的意思。所以请扔掉void main这种写法。
为什么我们对 main()这种省略返回值的写法置有微词？能发明这种写法的人，必定是了解了，在C语言中，如果一个函数不显式声明自己的返回值，那么会被缺省认为是int，但这一步是由编译器掌控，然而C语言设计之初便是让我们对一切尽可能的掌握，而一切不确定因子我们都不应该让它存在。其次有一个原则，能自己做的就不要让编译器做。

为什么我们对参数放空置有不满(int main())?在C语言中，一个函数的参数列表有三种合法形态:

```
 int function();
 int function(void);
 int function(int arg_n);
 int function(int arg_n, ...);
```

第一种代表拥有未知个参数,第二种代表没有参数，第三种代表有一个参数，第四种代表拥有未知个参数，并且第一个参数类型为int,未知参数在C语言中有一个解决方案就是，可变长的参数列表，具体参考C标准库，在此我们解释的依据就是，我们要将一切都掌控在自己的手中，我们不在括号内填写参数，代表着我们

认为一开始的意思是它为空，正因此我们就应该明确说明它为void,而不该让它成为一个未知参数长度的函数，如此在你不小心传入参数的时候，编译器也无法发现错误。

int main(int argc, char* argv[]) 和 int main(void)才是我们该写的C语言标准形式

对于缩进，除了编译器提供的符号缩进之外，我们可以自己给自己一个规范(请少用或者不用Tab)，比如每一块代码相教上一个代码块有4格的缩进。

对于学习C语言，请使用.c文件以及C语言编译器练习以及编写C程序，请不要再使用C++的文件编写C语言程序，并且自圆其说为了效率而使用C++的特性在C语言中，我们是祖国的下一代，是祖国的未来，请不要让自己毁在当下，珍爱编程，远离清华大学出版社。

之所以如此叙述，并不是因为情绪，而是当真如此，下方代码:

```
/*file: test.c*/
#include <stdio.h>
#define SIZES 5
int main(void)
{
 int* c_pointer = malloc(SIZES * sizeof(int));
 /*发生了一些事情*/
 free(c_pointer);
 return 0;
}
```

这是一段标准的C语言程序，但是它能在C++个编译器下编译运行吗？换句话说当你将文件扩展名由.c改为.cpp之后，它能编译通过吗？答案是不能。

为什么？答案是C++并不支持void*隐式转换为其他类型的指针，但是C语言允许。还有许许多多C于C++不相同的地方，兴许有人说C++是C的超集，但我并不这么认为，一门语言的出现便有它的意义所在，关键在于我们如何发挥它的最大优势，而不是通过混淆概念来增强实用性。

程序式子的写法

一个人活在世界上，时时刻刻都注意着自己的言行举止，而写程序也是如此，对于一个规范的能让别人读懂的程序而言，我们应该尽可能减少阻碍因子，例如:

```
int main(void)
{int complex_int=100;
int i,j,k,x;
for(int temp=0;temp<complex_int;++temp){k=temp;
x=k+complex_int;}
printf(complex_int="%d is k=%d x=%d\n",complex_int,k,x);
return 0;}
```

对于上述的代码，我总是在班级里的同学手下出现，但这段代码除了让别人困惑以外，自己在调试的时候也是十分不方便，每每遇到问题了，即便IDE提示了在某处错误，你也找不到问题所在，经常有人来问我哪里错了，大部分情况都是少了分号，括号，或者作用域超过，原因在哪？

要是一开始将代码写清楚了，这种情况简直是凤毛麟角，想遇上都难。对于一个代码而言，我们应该注意让其变得清晰。

等号两边使用空格：

```
int complex_int = 100;
```

使用多个变量的声明定义，或者函数声明定义，函数使用时，注意用空格分开变量：

```
int i, j, k, x;//但是十分不建议这么声明难以理解意义的变量
printf("complex_int = %d is k = %d x = %d\n", complex_int, k, x);
void present(int arg_1, double arg_2);
```

对于一个清晰的程序而言，我们要让每一个步骤清晰且有意义，这就要求我们在编写程序的时候尽量能让代码看起来结构化，或者整体化。尽量让每个程序式子为一行，如果有特别的需要让多个式子写在同一行，可以使用,操作符进行组合，但是会让程序更难理解，日后调试的时候也更难发现错误。

```
/*Style 1*/
for(int temp = 0;temp < complex_int;++temp)
{
 k = temp;
 x = k + complex_int;
}
/*Style 2*/
for(int temp = 0;temp < complex_int;++temp){
 k = temp;
 x = k + complex_int;
}
```

对于上方的代码，是C语言代码花括号的两种风格，最好能选择其中一种作为自己的编程风格，这样能让你的程序看起来更加清晰，混合使用的利弊并不好说，关键还是看个人风格。

对于作用域而言，在C语言中有一个经常被使用的特例，当一个条件语句，或者循环只有一条语句的时候，我们常常省略了花括号{}，而是仅仅使用一个分号作为结尾，这在很多情况下让代码不再啰嗦：

```
if(pointo_int == NULL)
 fprintf(stderr, "The pointer is NULL!\n");
else
{
 printf("%d\n",*pointo_int);
 pointo_int = pointo_int->next;
}
```

在这段代码中if语句下方的代码并没有使用{}运算符进行指明，但是根据语法，该语句的确是属于if语句的作用范围内，如果我们此时写上了{}反而会令代码看起来过于啰嗦。但是有的时候，这条特性并不是那么的有趣，当使用嵌套功能的时候，还是建议使用{}进行显式的范围规定，而不是使用默认的作用域：

```
for(int i = 0;i< 10;++i)
 for(int k = 0;k < 10;++k)
  while(flag != 1)
   set_value(arr[i][k]);
```

这段代码，看起来十分简洁，但是确实是一个很大的隐患，当我们要调试这段代码的时候，总是需要修改它的构造，而这就带来了潜在的隐患。所以建议在使用嵌套的时候，无论什么情况，都能使用{}进行包装。

综上所述，在开始编写一个标准C语言程序的时候，请先把下面这些东西写上：

```
#include <stdio.h>

int main(void)
{
 return 0;
}
```

C代码规范

命名

只要提到代码规范，就不得不说的一个问题。
在一些小的演示程序中，也许费尽心思去构思一个 命名 是一件十分傻的行为，但是只要程序上升到你需要严正设计，思考，复查的层次，你就需要好好考虑 命名 这个问题。
函数命名：

C语言中，我们可以让下划线或者词汇帮助我们表达函数功能：

前缀：

1. set 可以表示设置一个参数为某值
2. get 可以表示获取某一个参数的值
3. is 可以表示询问是否是这种情况

后缀：

1. max/min 可以表示某种操作的最大(小)次数
2. cnt 可以表示当前的操作次数
3. key 某种关键值

```
  size_t get_counts();
  size_t retry_max();
  int is_empty();
```

需要注意的只是，不要让命名过于赘述其义，只简单保留动作以及目的即可，详细功能可以通过文档来进行进一步的解释。
结构体命名：

由于结构体的 标签，不会污染命名，即标签不在命名搜索范围之内，所以可以放心使用：

有人习惯使用 typedef, 而有人喜欢使用 struct tag obj，后者比较多，但是前者也不失为一种好方法，仁者见仁智者见智。

```
  /*方法1*/
  struct inetaddr_4{
```

```
  int port;
  char * name;
};
struct inetaddr_4 *addr_info;
/*方法2*/
typedef struct _addr{
  int port;
  char * name;
}inetaddr_4;
inetaddr_4 *addr_info_2;
```

两者同处一个文件内亦不会发生编译错误。

变量命名

1. 所有字符都使用小写
2. 含义多的可以用 _ 进行辅助
3. 以 = 为标准进行对齐
4. 类型， 变量名左对齐。

等号左右两端，最少有一个空格。

```
int main(void)
{
 int   counts = 0;
 inetaddr_4 *addr = NULL;

 return 0;
}
```

为了防止指针声明定义时候出错，将 * 紧贴着变量名总不会出错。

```
inetaddr_4 *addr, object, *addr_2;
```

其中 addr 和 addr_2 是指针，而 object 则是一个栈上的完整对象，并不是指针。

全局变量能少用就少用，必须要用的情况下，可以考虑添加前缀 g_

```
int g_counts;
```

#define 命名

1. 所有字符都是用大写，并用 _ 进行分割。
2. 如果多于一个语句，使用 do{...}while(0) 进行包裹，防止；错误。

```
#define SWAP(x, y)  \
do{      \
 x = x + y;  \
 y = x - y;   \
 x = x - y;   \
}while(0)
```

当然这个交换宏实际上有一点缺陷，在大后方会提出。此处是代码规范，就不重复强调。

enum 命名

1. 所有字符都是用大写，并用 _ 进行分割
2. 与 define 相比，enum适用于同一类型的常量声明，而不是单一独立的常量。往往出现都是成组。

格式化代码

花括号 **{}**

1. 混合使用符合节俭思想，但会稍微有一点结构紊乱。
2. 单一使用能更好让代码结构清晰。
3. 所谓混合，单一指的是是否一直使用 {} 进行代码包裹。
4. 有人认为 当单一语句的时候不必要添加 {}，有的人则习惯添加
5. 当作用域超过一个屏幕的时候，可以适当的使用注释来指明 {} 作用域

```
while(1){
   if(tmp == NULL){
    break;
   }
```

```
  else if(fanny == 1){
   ... 大概超过了一个屏幕的代码
  } /*else if fanny*/
 }/*end while*/
```

如果是代码量少的情况下，但嵌套比较多，也可以使用这个方式进行注释。

括号 **()**

有人建议除了函数调用以外，在条件语句等类似情况下使用 () 要在关键字后空一格，再接上 ()语句，对于这一点，我个人习惯是不空格，但总有这种说法。

```
if (space == NULL) {
 /**TODO**/
}
while(1){
 /**我习惯于如此写**/
}
strcpy(str1, str2); /**第一种写法是为了和函数调用写法进行区分**/
return 0;
switch
```

一定要放一个 default 在最后，即使它永远不会用到。
每个 case 如果需要使用新变量，可以用 {} 包裹起来，并在里面完成所有操作。

```
switch(...)
{
 case 1:
  /**TODO**/
  break;

 case 2:
 {
  int new_vari;
  /**创建新变量则用 {} 包裹起来**/
 }
 break;

 default:
  call_error();
}
```

goto

虽然许多人，许多书都提醒不再使用 goto 关键字，而是使用 setjmp 和 longjmp来取代它，但是这还是那句话，仁者见仁智者见智，如果 goto 能够让代码清晰，那何乐而不为呢，这个观点也是最近才体会到的（并非我一己之言）。
具体使用可以查询官方文档。
语句

1. 应该让完整的语句在每一行中，只出现一次。
2. 对于变量声明定义亦是如此
3. 原因是这样能让文档更有针对性

头文件保护

对于头文件而言，在一个程序中有可能被多次包含(#include)，如果缺少头文件保护，则会发生编译错误
不要将 _ 作为宏的开头或者结尾。

```
#ifndef VECTOR_H_INCLUDE
 #define VECTOR_H_INCLUDE
  /**TODO**/
 #endif
```

宏

C语言的宏有诸多弊端，所以尽量使用 inline 函数来代替宏。在大后方会有解释
但是，请不要因此抛弃了宏，比如在 C11 中有一个新兴的宏。
变量

第一时刻初始化所有所声明的变量，因为这么做总没有坏处，而且能减少出错的可能。
函数

函数应该尽可能的短小，一个ANSI屏幕的为最佳。

如果某个循环带着空语句，使用 {} 进行挂载，以免出现意外。

```
while(*is_end++ != '\0')
{
 ;
}
```

虽然是空的循环体，但是写出来以免造成误循环。

尽量不要让函数返回值直接作为条件语句的判断，这样会极大降低可读性

```
if(is_eof(file) == 0)
 好过
if(!is_eof(file))
```

不要为了方便或者一点点的所谓速度提升(也许根本没有)，而放弃可读性，使用嵌入式的赋值语句

```
int add = 10;
int num = 11;
int thr = 20;
add = add + thr;
num = add + 20;
```

不要写成

```
num = (add = add + thr) + 20;
```

浮点数

1. 万万记住不要再使用浮点数比较彼此是否相等或不等。
2. 如果把浮点数用在离散性的数据上，比如循环计数器，那就...

其他

使用 #if 而不是 #ifdef

可以使用 define() 来代替 #ifdef的功能

```
#if !define(USERS_DEFINE)
 #define USERS_DEFINE ...
#endif
```

对于某些大段需要消除的代码，我们不能使用注释 /**/，因为注释不能内嵌着注释(//除外)，我们可以使用黑魔法：

```
#if NOT_DECLARATION
 /**想要注释的代码**/
#endif
```

不要使用纯数字

意味着，不在使用毫无标记的数字，因为可能你过了几个月再看源代码的时候，你根本不知道这个数字代表着什么。

而应该使用#define 给它一个名字，来说明这个数字的意义。

Tags: C语言 语法 规范