C语言 编码规范

Programming
Your Future

东软IT人才实训中心
胡本仁　Mail: Hubr@neusoft.com

目的与目标

- 规范部门内软件开发和设计风格，保证所有开发人员写出风格一致的代码
- 增强代码的健壮性、可读性和可维护性
- 通过人为以及自动的方式对最终软件应用质量标准
- 减少程序的潜在错误

时间：4 学时
教学方法：讲授PPT
+案例分析

课程概述

- 表达式和控制流程
- 初始化
- 代码格式
- 函数
- 宏
- 兼容性
- 类型使用
- 类型转换
- 命名原则
- 内存分配和释放
- 类、头文件
- 性能
- 注释

表达式和控制流程

- 规则：在移位操作中，右操作数必须小于左操作数的位数；
- 原理：此规则应用于32位操作系统，移位位数超出32位后，数值不会是0；
- Example：

```
void foo()
 {
        unsigned int uVal = 1u;
        uVal <<= 100; /* Violation */
 }
```

- Repair：

```
void foo()
{
        unsigned int uVal = 1u;
        uVal <<= 2; /* OK */
}
```

表达式和控制流程

- 规则：禁止在逻辑运算符[&&/||]的右操作数中出现++/--运算符和函数调用操作；
- 原理：右操作数有可能没有执行

- Example

```
int foo1(int);
void foo2()
{
  int ishigh=1;
  int x,i;
  if (ishigh && ( x == i++)) {}      /* Violation */
  if (ishigh || ( x == foo1(x))) {}  /* Violation */
}
```

表达式和控制流程

- 规则：不要将两个int类型变量的除法结果赋值给float类型变量
- 原理：避免丢掉小数部分
- Example

```
void func()
{
   int a = 3;
   int b = 4;
   double d;
   d = a / b;  // Violation
}
```

Repair

```
void func()
{
   int a = 3;
   int b = 4;
   double d;
   d = ((double)a) / b; // OK
}
```

表达式和控制流程

- 规则：不要在if/while/switch条件表达式中使用++/--运算符
- 原理：提高可读性，避免错误
- Example

```
int Foo (int iVar)
{
    if (iVar-- && iVar<10) {        // Violation
        iVar += 10;
    }
    else{
        iVar -= 10;
    }
    return iVar;
}
```

表达式和控制流程

- 禁止在bool表达式中出现赋值语句
- 禁止对于带符号的整型数进行位操作
- 不要对浮点型变量进行等于/不等于的判断
- 清晰的表示变量与0值的比较
- 在for语句的表达式中，只出现影响循环控制的变量，而不是其他无关变量
- 不要比较指针类型变量
- 不要在sizeof的参数中进行自增、自减[++/--]、赋值操作
- 在条件表达式中，对每个逻辑判断都使用（），即使没有逻辑上的需求
- 不要在if语句中的条件表达式中进行赋值操作

循环语句效率

- 如果计数器从0开始计数，建议For语句的循环控制变量的取值采用“前闭后开区间”写法，以防止出现“差1”错误
- 在多重嵌套的循环中，建议将最长的循环放在最内层，最短的循环放在最外层，以减少CPU跨切循环层的次数，从而优化程序性能
- 如果循环体内存在逻辑判断，并且循环次数很大，建议将逻辑判断移到循环体的外面

If语句

- If语句独占一行，执行语句不得紧跟其后。
- 无论有多少条执行语句，都必须用大括号将执行语句括起来。
- 使用else if语句的时候，最后的else节一定要记述，避免else与if的匹配错误。
- if语句中记述多个条件时，在逻辑演算符“&&”, “||”的两侧的条件是用括弧”(”、”)”括起来。但是，条件是单项的时候可以不必用括弧括起。
- 需要用浮点数比较的时候，不要通过等号“==”和不等号“!=”来进行比较。

switch语句

- switch语句独占一行。
- 每个case独占一行，执行语句不得紧跟其后。
- 若某个case不需要break要给出确认性注释。
- 一定要提供default。
- 真假两种处理采用if语句，不采用switch语句。
- case语句的排列顺序是、从以下这样的排列顺序中选择有效的。
  - (1). 按频度多少的顺序，将比较频繁的处理放在前面，这样可以提高执行的效率；
  - (2). 从通常的case到例外的case；
  - (3). 数值顺序或字母顺序。

for语句

- for语句独占一行，执行语句不得紧跟其后。
- 无论有多少条执行语句，都必须用大括号将执行语句括起来。
- 空的循环体要给出确认性注释。
- 若无特殊情况，循环计数要从0开始，不要从1开始。结束条件要用“<”，不要用“<=”。
- Loop记数器在Loop内不强制进行变更（即Loop记数器的数值只在增量表达式中修改）。
- 尽量避免在while/for循环体中申请或释放内存，以防止产生内存碎片。
- 在For循环中，禁止使用逗号演算子“,”。

while语句

- while语句独占一行，执行语句不得紧跟其后。
- 无论有多少条执行语句，都必须用大括号将执行语句括起来。
- 空的循环体要给出确认性注释。

return语句

- Return语句的返回值的类型和已声明的类型是同样的。
- void型的函数时、Return语句不应该有返回值。
- 建议一个函数一条返回语句。

- 禁止使用goto语句。

初始化

- 规则：在enum类型中，将成员全部初始化，或者都不初始化，或者全部初始化
- 原理：避免代码错误
- Example

```
enum TEST { /* Violation */
    X=5,
    Y,
    Z=9,
};
enum TEST2 { /* Violation */
    X2,
    Y2=5,
    Z2,
};
```

- Repair

```
enum TEST3 {        /* OK */
    X3,
    Y3,
    Z3,
};
enum TEST4 {        /* OK */
    X4=2,
    Y4,
    Z4,
};
enum TEST5 {        /* OK */
    X5=2,
    Y5=4,
    Z5=8,  };
```

初始化

- 规则：初始化所有变量，并且在声明时初始化而不是使用时初始化
- 原理：避免初始化时创建临时对象，可以提高代码效率
- Example

```
int main()
{
 int a;  // Violation
 int b = 0; // OK
}
```

Repair

```
int main()
{
 int a = 0; // OK
 int b = 0; // OK
}
```

初始化

- 规则：初始化所有指针变量
- 原理：避免参照未初始化的指针变量
- Example

```
void foo() {
 int *i;                    // Violation
}
```

- Repair

```
void foo() {
 int *i = 0;                // OK
}
```

代码格式

- 规则：注意控制参数的数量，一般来说不要超过7个，当参数过多时，应该考虑将参数定义为一个结构体，并且将结构体指针作为参数
- 原理：提高可读性和可维护性
- 规则：函数的大小不要过长，一般定为350行以内（除去注释，空行，变量定义，调试开关等）
- 原理：提高可读性和可维护性，遵循函数设计原则
- 规则：在switch中提供default分支
- 原理：提高容错性和可维护性，不必担心是否会遗漏相关的switch语句

代码格式

- 对嵌套语句使用4个字符的缩进
- 用空行将代码按逻辑片断划分
- 如果函数的参数过长，要从第二个参数开始换行显示，每一个参数和第一个参数对齐
- 函数没有参数，使用（ void ），而不使用（）
- 避免switch语句中只有一条case分支
- 存储类型必须紧跟类型，而不是变量
- 在声明和定义处，将*/&操作符紧跟类型
- 不要在使用./->操作符的左右存在空格
- {}总是独立一行

代码格式

- 在代码中，使用符号值代替数值
- 在switch中，每一条case分支必须使用break语句结束
- 每行只有一条语句
- 赋值操作符前后留有一个空格
- 位操作符前后留有一个空格
- 逻辑判断操作符前后留有一个空格
- 在[ ]操作符前后不能出现空格
- 逗号后面加一个空格
- 分号后面加一个空格
- 不能在操作符(++/--)与操作数之间出现空格

函数

- 规则：使用const定义不需要被改变的引用方式的函数参数，使用const定义不需要被改变的指针类型函数参数
- 原理：避免无意的修改调用者数据
  - Example

```
struct Foo {
   int x;
   int y;
};
int Bar(Foo &f) // Violation
{
   return f.x;
}
int FooBar(Foo &f)        // OK
{
   return f.x++;
}
```

  - Repair

```
struct Foo
{
   int x;
   int y;
};
int Bar(const Foo &f)   // OK
{
   return f.x;
}
```

函数

- 规则：将数组作为函数参数传递的时候，不要指定数组大小
- 原理：避免数据丢失
- Example

```
void foo2(int ii[30])                          // Violation
{
}
void foo3(char a,int ii[30][30][30]) // Violation
{
}
```

- Repair

```
void foo1(int ii[])                            //OK
{
}
void foo4(char a,int ii[][30][30])             //OK
{
}
```

函数

- 规则：如果不需要修改内建类型参数的值，将变量值作为参数进行函数参数传递，而不是地址
- 原理：提高代码性能和效率，防止误修改
- Example

```
int Foo(int i, int &j)  // Violation
{
    return i + j;
}
int Bar(int i, int &j)  // Violation
{
    j += i;
    return j;
}
```

Repair

```
int Foo(int i, int j)  // OK
{
    return i + j;
}
```

函数

- 规则：使用ctype.h中的函数进行字符判断
- 原理：提高代码效率，使代码简洁
- Example

```
#include <ctype.h>
void test(char c) {
   if( 'a' <= c && c <= 'z') {  // Violation
   }

   while( 'A' <= c && c <= 'Z') { // Violation
   }
}
```

Repair

```
#include <ctype.h>
void test(char c) {
   if(islower(c)) // OK
    {
    }
   while(isupper(c)) //
OK
    {
    }
}
```

函数

- 显式地声明函数的返回值类型，如未指定，将依赖编译器
- 声明和定义处的参数标识符应该保持一致
- 禁止函数返回局部变量的指针或引用，避免程序不可预知的行为
- 对于非void返回类型的函数，return语句必须提供表达式；对于void返回类型的函数，return语句不要提供表达式；避免不确定的行为，编译器出warning
- 不要写出永远无法执行到的代码
- 不要声明永远不会使用的局部变量和参数
- 对所有的指针参数要进行非空检查后才能使用
- 在使用scanf/fscanf函数的时候，需要指定输入字符串的长度

宏

- FALSE要被定义成0，TRUE要被定义成1
- 宏定义中的乘号/除号前后必须使用括号，提高可读性，保证操作符的顺序
- 使用typedef代替define的类型别名定义，便于改变函数返回值，易于代码review
- 禁止在宏定义中使用关键字和基本类型，避免重新定义语法

兼容性

- 规则：任何通过数组下标索引的操作都要做溢出检查
- 原理：避免数组下标越界
- Example

```
void foo()
{
    int j;
    int k[10];
    int m;
    m = k[j]; /* Violation */
}
```

Repair

```
void moo()
{
    int j;
    int k[10];
    int m;

    if ((j < 10) && (j >= 0)) /* OK */
    {
            m = k[j];
    }
}
```

兼容性

- 不要使用移位操作代替数学操作
- 检查函数的返回值，提高容错性
- 在代码中的路径只使用/，而不使用\，保证在头文件的路径中只使用标准字符
- 不要把NULL用在指针以外的场合
- 解引用指针前，需要判断指针有效性
- 传入库函数中的变量需要进行有效性判断（C库函数）

类型使用

- 规则：使用[ ]操作数组元素，而不要使用*
- 原理：避免隐含操作和意图
- Example

```
void foo()
{
    int array[2];
    *(array+1) = 0; //Violation
    *array = 0; //Violation
}
```

Repair

```
void foo()
{
    int array[2];
    array[1] = 0; //OK
}
```

类型使用

- 禁止在结构体中包含成员函数
- 确保使用前，定义预处理标识符
- 不要重新定义保留的关键字和标准库中的名字，

  例如：#define __LINE__ 12

  #define break 1
- 不要定义超过两级的指针

类型转换

- 规则：不能将函数指针转换成其他原始类型指针
- 原理：意图不明
- Example

```
void Foo(char *ptrC)
{
    *ptrC = 0;
    return;
}
```

```
void function( )
{
    void *ptrV = 0;
    void (*funPtr) (char*) = 0;
    funPtr = &Foo;
    ptrV = (void*)funPtr;          //
Violation
    return;
}
```

类型转换

- 规则：不要将const类型转换成非const类型
- 原理：避免破坏数据的完整性
- Example
- Example:

```
void main( )
{
    const int a = 10;
    int b;
    b = (int) a;       // Violation
}
```

Repair:

```
void main( )
{
    const int a = 10;
    int b;
}
```

类型转换

- 不要将指针类型转换成非指针类型
- 不要将有符号的char类型转换成无符号int类型

命名原则

- 公开的接口函数和全局名字空间的函数选择SmallTalk风格的命名原则（一种首字母大写，字间直接相连而无分隔符的书写风格）
- 变量的命名采用匈牙利命名法（类型 + 标识符）
- 避免使用只靠字母大小写才能区分的名称
- 命名时避免以下划线开头，名称中避免出现两个下划线相连
- 如果某个单词没有标准的缩写，要使用全名
- 名字中不要出现形状容易混淆的字母和数字
- 布尔型的名字要使用正值谓词从句
- 所有宏定义的标识符必须大写
- 要防止局部变量与全局变量同名
- 以‘is’开头命名的函数返回值必须是bool类型
- 实现行为的过程或函数要使用动词或动词短语

命名原则

- short的类型为s
- unsigned short的类型为us
- long的类型为l
- unsigned long的类型为ul
- signed char类型变量名使用‘c’前缀
- unsigned char类型变量名以‘uc’前缀
- int类型变量名使用‘n’前缀
- unsigned int类型变量名使用‘ui’前缀
- bool类型变量名使用‘b’前缀
- double类型变量名使用‘d’前缀
- float类型变量名使用‘f’前缀

命名原则

- wide char的类型为wc
- struct的类型为st
- union的类型为un
- enum的类型为en
- 指针类型变量名使用‘p’作为前缀
- 结构体名格式为tagXXXX_t
- 结构体别名格式为XXXX_t
- 联合名格式为tagXXXX_u
- 联合别名格式为XXXX_u
- 枚举名格式为tagXXXXEnum
- 枚举别名格式为XXXXEnum

命名原则

- 数组的命名方式为"a" + 基本类型前缀 ＋ 标识符
- 指针的命名方式为"p" + 基本类型前缀 ＋ 标识符
- 指针数组的命名方式为 "a" + "p" + 基本类型前缀 ＋ 标识符
- 数组指针的命名方式为 "p" + "a" + 基本类型前缀 ＋ 标识符
- 函数指针命名方式为 "p" + "f" ＋ 标识符
- 函数指针数组的命名方式为 “a” + “p” + “f” ＋ 标识符

内存分配和释放

- 规则：使用new/delete替换malloc/calloc/realloc/free
- 原理：malloc/free是标准库函数，new/delete是操作符，对于非内部数据类型，malloc/free无法满足动态对象的要求
- Example

```
#include <malloc.h>
int main(){
    char *pc;
    pc = (char *)malloc(100);  // Violation
    free(pc);                  // Violation
    return 0;
}
```

Repair:

```
int main(){
    char *pc;
    pc = new char[100];
    delete [] pc;
    return 0;
}
```

内存分配和释放

- 规则：使用相同的形式调用new/delete
- 原理：避免内存泄露
- Example

```
class A {
public:
    A() { }
};
void foo() {
    A *a = new A[100];
    delete a;  // Violation
}
```

Repair:

```
class A {
public:
    A() { }
};
void foo() {
    A *a = new A[100];
    delete a[];  // OK
}
```

内存分配和释放

- 内存指针释放后，指针应该置空，保证指针使用的安全性
- 检查new操作的返回值，避免申请失败，操作空指针

头文件

- 规则：为头文件加上重复包含的保护
- 原理：提高可维护性，利于代码优化
- Example

  file foo.hh:

  ```
  #ifndef FOO_HH
  #define FOO_HH
  int i;
  #endif
  ```

头文件

- 不要多处声明extern类型的对象，只在头文件中使用extern关键字，而不要在源文件中使用
- 只引用需要的头文件
- < >只用在引用系统或语言本身提供的头文件，其他情况一律用” ”

性能

- 使用 +=， -=，〉>>=, 等复合运算符，而不使用 A = A+1…等这样的操作，以减少临时变量的创建
- 使用前增量代替后增量，避免创建临时对象

注释

- 在每个函数前添写注释
- 在每个文件前添写注释
- 每个空的循环体都需要给出确认性注释
- 若某个case不需要break需要加注释
- 尽量用代码自说明，而不是使用注释
- 注释不能超出被注释代码所包含的内容
- 不可以在注释中约束使用者的行为
- 将不再使用的代码删除而不是注释掉
- Bug修改时注释要明确。

- 程序的一致性比你个人的习惯更重要。

参考资料

- 《高质量程序设计指南-C/C++语言》
- 《代码大全》

- 详见 1、C编码规范.xls
  2、C++编码规范.xls

Beyond Technology