



编者按:本文是《学习 MISRA - C》系列连载讲座之六,共六讲。

第一讲:“安全第一”的 C 语言编程规范,简述 MISRA - C 的概况。

第二讲:“跨越数据类型的重重陷阱”,介绍规范的数据定义和操作方式,重点在隐式数据类型转换中的问题。

第三讲:“指针、结构体、联合体的安全规范”,解析如何安全而高效地应用指针、结构体和联合体。

第四讲:“防范表达式的失控”,剖析 MISRA - C 中关于表达式、函数声明和定义等的不良使用习惯,最大限度地减小各类潜在错误。

第五讲:“准确的程序流控制”,表述 C 语言中控制表达式和程序流控制的规范做法。

第六讲:“构建安全的编译环境”,讲解与编译器相关的规范编写方式,避免来自编译器的隐患。

构建安全的编译环境

清华大学 陈萌萌 邵贝贝

预处理是编译环境处理 C 程序的第一个环节,但往往最先被程序员忽略。这份看似只是由编译环境做的简单工作,其实也是机关重重。通过介绍 MISRA - C 与预处理相关的规则,希望读者能够更准确地认识编译器的预处理过程,避免出错。无论是自定义函数还是由编译环境提供的标准库函数,如果使用不当,都会存在安全隐患。能不能保证函数被正确的定义、声明和调用,关系到整个程序的成败。这里介绍 MISRA - C 中涉及函数部分有代表性的规则,并试图分析制定这些规则的出发点,以帮助读者构建更为安全的编译环境。

1 函数的定义和声明

1.1 在哪里定义

读者也许会有这样的经历:在一个头文件中定义了一个变量,又让这个头文件被多个源文件引用。这时编译器会“报怨”说重复定义了同一个函数。出错的原因与其说是粗心,不如说是一个习惯的问题。

规则 8.5:头文件中不允许包含对象或函数的定义。

当源文件包含某一头文件时,预处理器会将头文件的内容在包含指令处展开。显然,在头文件中函数的定义会在其他源文件中一模一样的出现,导致函数被重复定义。解决这一问题的关键是明确一个概念:所有可执行的代码或者对象和函数的定义都应在 C 的源文件中,头文件中

只能存在其声明。具体的做法是:为全局变量的声明增加 extern 修饰符,并在相应的 C 源文件中定义对象或函数。例如,在 Goble.h 文件中仅声明变量 GCounter。

```
/*在 Goble.h 中 */  
extern uint32_t GCounter;
```

.....

而在 C 文件中定义变量 GCounter:

```
/*在 GobleVariables.C 中 */  
uint23_t GCounter;
```

.....

这样,就可以在所有需要用到全局变量的地方直接引用"Goble.h"头文件了。不过也有一些程序员喜欢采取以下的做法:

```
/*在 Goble.h 中 */  
#ifdef GLOBLES  
#define EXT  
#else  
#define EXT extern  
#endif  
EXT uint32_t GCounter;
```

.....

```
/*在 GobleVariables.C 中 */  
#define GLOBLE  
#include "Goble.h"  
.....
```



```

/ *在其他 C 文件中 */
#include "Goble.h"
#include "MyLib.h"
.....

```

这样做的好处是只需要维护“Goble.h”就可以维护所有全局变量。其他的文件中,直接包含“Goble.h”就可以使用这些全局变量了。上述的两种做法是等价的,读者可选择任意一种方式。

1.2 用好编译器的检查功能

在 MISRA - C 制定关于函数编程规则的背后,有一条很重要的思想:要充分利用编译器的类型检查功能来提高函数的可靠性。这里的类型检查包括在函数定义和调用时对函数参数和返回值的类型检查。

规则 8.1:函数必须声明原型,在函数定义和调用时原型必须可见。

首先明确一下什么是原型声明。在科尼汉和里查(K & R)的著名著作《C 程序设计语言(第二版)》的前言中提到:“C 不是一种强类型语言,但随着它的发展,其类型检查机制已得到了加强……在这个方向上,新的函数声明方式是迈出的另外一步。”

这里“新的函数声明方式”指的就是原型声明。原型声明是标准 C 语言中出现的概念,它可以提供更多关于函数参数的信息。在原型声明中,函数的参数要在声明时指定参数名和类型;而非原型声明,参数的类型可以缺省,被忽略的参数声明默认为 int 型。

请看下面的声明:

```

int f(int i, long j) { .....}      (原型声明)
int f(i, j) int i; { .....}      (非原型声明)

```

要求程序使用原型声明函数,主要是希望可以利用编译器检查函数调用时数据类型的一致性。如果调用函数时,没有进行原型声明,则编译器不会检查出函数形式参数与调用参数不一致。请看下面这段程序:

```

double square(x)
double x;
{
.....
}
调用时:
long func(i)
long i;
{ return square(i);
}

```

函数 square() 的形式参数类型是 double 型,但实际调用时,调用参数是 long 类型。因为没有进行原型声明,编译器不需要对此给出警告,结果在没有出错信息的情况

下,函数返回了一个不正确的值。

现在将这段程序改写为原型声明:

```

double square(double x)
{ ...
}
调用时:
long func(i)
long i;
{ return square(i);
}

```

这里,编译器会检查出函数 square 的实际调用参数和形式参数类型不符,并且会将实际参数转换成相应的形式参数的数据类型。这样,参数 i 就在程序员不知情的情况下被编译器自动转换为 double 类型,函数返回正确值。

了解了原型调用的一些机制,下面的问题就是如何操作才能保证每个函数调用都使用原型调用,也就是说,要使原型声明对于函数定义和调用都“可见”。简单的方法是:每一个外部函数都在头文件中有一个唯一的原型声明;需要调用此外部函数的源文件要包含这一头文件,保证调用时由原型控制(原型对于“调用”可见);同时,函数定义所在的源文件也包含这一头文件,以便编译器可以检查原型声明和其定义相匹配(原型对于“定义”可见)。

使用原型调用除了可以帮助检查参数的一致性,还可以使“编译器产生更为有效的函数调用序列”。

为了配合编译器对函数参数的检查,程序员应牢记规则 16.1。

规则 16.1:不允许定义参数数量不确定的函数。

标准库函数 printf() 深受许多程序员的喜爱,因为 printf() 允许不确定的参数数量,用起来很方便。但是,参数数量的不确定很可能造成编译器无法检查函数调用时的参数一致性。对于像 printf() 这种使用广泛的标准库函数,编译器提供了一些合适的调用机制。但程序员必须明确,编译器无法保证对用户自行定义的数量不确定的函数进行数据类型检查。因此, MISRA-C 不允许用户冒险去定义新的参数数量不确定的函数。

2 函数的调用和标准库函数

程序员应该清楚,嵌入式应用开发中系统的资源往往十分有限,在程序开发上会有特殊的限制。比如,在 RAM 空间的使用上往往会捉襟见肘。像早期的 PC 机程序员一样,对 RAM 空间的使用可以用“吝惜”来形容,往往可以使程序少占用几个字节的 RAM 而大做文章。

一个典型的例子就是递归函数的调用。递归函数的代码紧凑,且容易理解,很受 C 程序员的推崇。但递归函数的一个缺点就是:占用 RAM(这里主要是指栈空间)

资源太多。对于嵌入式系统来说这是尤为严重的问题。一旦递归调用的层数过多,就会出现栈空间不足的情况。唯一可以避免该情况发生的方法就是能够预先估计出最大的递归调用层数,从而算出最大栈空间。遗憾的是,很多情况下程序员根本没法做出估计,这时系统中的递归函数成为一个巨大的隐患。MISRA - C 从系统安全角度考虑,选择了最为安全的做法,不准使用递归调用。

规则 16.2: 函数不得调用本身,无论是直接的调用,还是间接的调用。

一般来说,标准库函数是很好用的。它的定义和使用都很清晰,尤其是像 `printf()` 这样的函数,对于程序员的调试工作帮助很大。但某些库函数的使用也可能会造成问题。要尽量安全地使用库函数,需要注意三个方面的问题。

要保证库函数头文件中的宏、标识符和函数的定义不受干扰。

规则 20.1: 不得定义、重新定义或是取消定义标准函数库中的标识符、宏和函数。

要按照正确的方法使用库函数。库函数对参数的类型、数值都有很明确的要求,只有传递给库函数正确的参数,才能保证结果的正确性。

规则 20.3: 必须检查传递给库函数的数值的有效性。

避免使用可能有问题的库函数或者其结果。比如很多库函数都会通过一个叫做 `errno` 的变量为非零值来表示执行失败。但是,由于没有强制库函数在执行成功后将 `errno` 清零,一个非零的 `errno` 有可能是因为当前库函数执行失败了,也有可能是因为之前某个库函数没有正确执行。因此,完全依赖 `errno` 来判断库函数的执行成功与否是不可靠的。

规则 20.5: 不得使用错误指示符 `errno`。

可能带来问题的库函数还有很多,MISRA - C 为此做了一份总结。

规则 20.4: 不得使用动态堆空间分配。

规则 20.6: 不得使用库函数 `<stddef.h>` 中的宏 `offsetof`

规则 20.7: 不得使用 `longjmp` 函数中的宏 `setjmp`

规则 20.8: 不得使用信号处理函数 `<signal.h>`

规则 20.9: 不得用输入/输出库函数 `<stdio.h>` 来产生代码

规则 20.10: 不得使用标准库 `<stdlib.h>` 中的库函数 `atof`、`atoi` 和 `atol`

规则 20.11: 不得使用标准库 `<stdlib.h>` 中的库函数 `abort`、`exit` 和 `system`

规则 20.12: 不得使用标准库 `<time.h>` 中的时间处理函数

3 预处理——看似简单的第一步

预处理是编译器处理程序的第一步。预处理会在编译器编译程序代码前做一些准备工作,最为常见的工作是处理文件包和宏定义(分别对应 `#include` 和 `#define` 两个预处理指令)。

预处理器并不对源代码做编译,只是进行一些转换工作,例如将文件或宏展开等。很多程序员认为这种类似复制、粘贴的活没什么了不起,也就放松了对预处理工作的检查。其实,很多程序的失败就从这看似简单的第一步开始。

宏定义是最常见的预处理指令之一。MISRA - C 关于宏定义有一些很有代表性的规则。

3.1 小括号——一个也不能少

当程序中多处出现同一个数值的时候,程序员就会想起使用宏。宏定义最大的好处就是使一些常量集中起来,修改其值只需要修改一次,大大提高了程序的可维护性。

编译器对宏的处理原则比较简单,宏定义只对程序文本起作用。一个经典的例子是:

```
#define abs(x) (x >= 0) x : - x
```

显然,程序员希望求变量 `x` 的绝对值。但是,下面的调用会是什么结果呢?

```
abs(a - b);
```

展开后为 `(a - b >= 0) a - b : - a - b`。显然这里的 `- a - b` 并不是程序员想要的结果,原因是变元 `x` 两边没有加小括号。那么在 `x` 加上括号以后呢?

```
#define abs(x) ((x) >= 0) (x) : - (x)
```

如果此时是

```
abs(a) + 1;
```

展开后是 `(a >= 0) a : - a + 1`,也不是我们想要的结果。看来还要把整个宏定义加上括号,这样才能得到安全可靠的宏定义:

```
#define abs(x) (((x) >= 0) (x) : - (x))
```

为了防止宏展开后因缺少括号而存在的优先级错误问题,MISRA - C 有如下规定。

规则 19.10: 在函数式宏定义中,任何一个参数都应加上小括号,除非是在 `#` 或 `##` 运算符中。

3.2 宏定义不是函数

规则 19.7(推荐): 应优先考虑使用函数而非函数式宏定义。

利用类似函数式的宏定义来取代函数调用,是一个常用的技巧。这样做有很多好处,主要是能够提高程序的运行速度。MISRA - C 从代码安全的角度制定这一规则,主要有两点考虑。一是宏定义不能像函数调用那样提供参数类型检查,错误的变元类型无法得到纠正,运行的结果



就可能不正确。二是宏定义中的变元可能会多次求值,当变元表达式带有副作用时,就会出现这个问题。例如:

```
# define SQUARE(x) ((x) *(x))
```

当有如下语句时:

```
a = 3;
b = SQUARE(a + +);
```

程序员肯定希望得到 $b = 9$ 和 $a = 4$ 的结果,可实际上的结果却是 $b = 12$ 和 $a = 5$,这是为什么呢?

如果考虑到宏展开只是做文本的展开,那么上面的预处理结果应该是:

```
a = 3;
b = (a + +) * (a + +);
```

很明显,这里 $a++$ 运行了两次,运行后 $a = 5$ 。至于 b ,其结果应该是 $b = 3 \times 4 = 12$ 。

现在,读者应该可以看出来类似函数的宏展开并不完全和函数一样。考虑到系统可靠性是我们所关注的,上面的工作还不如直接用函数来完成。多数情况下,函数的运行速度应该让位于其结果的正确性。

关于宏定义还有很多有趣的问题可以讨论,这里就不一一赘述了。

结 语

至此,关于 MISRA - C:2004 的学习暂告一段落。MISRA - C:2004 有 141 条规则,在 6 期的《学习园地》栏目中,列举和解释了其中有代表性的规则,大约二分之一,且尽量使每篇文章都能涵盖 MISRA - C 规范的一个重要方向,以便使读者了解到 MISRA - C 的概貌和主导思想。由于 MISRA - C:2004 一书中关于每条规则的解释很少,很多例子是在我们理解的基础上加的,可能存在着错误或偏差,欢迎大家和我们共同讨论。

通过这 6 期介绍,希望大家能够意识到:C 是一门并不容易掌握的语言。作为嵌入式工程师,多数人只是某应用领域的专家,对于 C 语言编程,有个从“业余”到“专业”的过程,学习和参考 MISRA - C 可以帮助他们编写出更安全、更“专业”的代码。尤其对于一些安全性要求很高的

小系统,MISRA - C 是非常合适的安全编程规范。这里小系统指开发团队的规模小,甚至全部工作都是由一个人完成的系统。

尽管手册式的编程规范很难引起读者的兴趣,MISRA - C:2004 还是值得仔细品味的。经验对于程序员来说是一笔财富。MISRA - C 的编写专家们大都来自于汽车工业及相关软件公司,他们有着丰富的汽车安全方面的知识和软件开发经验,MISRA - C:2004 很大程度上是他们对如何提高 C 软件可靠性的经验总结。对于大多数程序员来说,仔细研读这份经验总结可以少走很多弯路,同时提高自身的编程素养。

目前,“嵌入式”还不是一个学科或专业,也许永远也不会是一个独立的学科。然而,各行各业都需要嵌入式系统,都有义务推动“嵌入式学科”(如果能这样表述的话)的发展。例如,便携类应用就推动了嵌入式系统低功耗技术的发展。MISRA - C 从汽车工业软件可靠性角度,对 C 语言的使用做出种种限制,使之成为汽车工业的行业标准,并被其他对可靠性要求高的行业采纳,这是汽车行业对嵌入式领域的贡献。其他行业的嵌入式工程师也应该有责任、有能力在借鉴其行业相关技术、规约的基础上,从不同角度推动嵌入式技术的全面发展。ME

参考文献

- [1] MISRA - C:2004, Guidelines for the use of the C language in critical systems. The Motor Industry Software Reliability Association, 2004.
- [2] Kernighan. Brian W, Ritchie. Dennis M. C 程序设计语言. 徐宝文译. 第 2 版. 北京:机械工业出版社, 2001.
- [3] Harbison III. Samuel P, Steele Jr. Guy L. C 语言参考手册. 邱仲潘, 等译. 第 5 版. 北京:北京机械工业出版社, 2003.
- [4] Les Hatton. The MISRA C Compliance Suite - The next step, Oakwood Computing. <http://www.misra-c2.com>.
- [5] ISO/IEC 9899:1999. International Organization of Standardization, 1999.

(收稿日期:2006-01-11)

ARM 和 TSMC 签署 65 纳米和 45 纳米的物理 IP 长期协议

2006 年 4 月 21 日,ARM 公司和台湾积体电路制造股份有限公司(TSMC)签署协议把公司的长期合作关系扩展至开发一套全新的 ARM Advantage 产品,该产品是 Artisan 物理 IP 系列产品的一部分,用来支持 TSMC 的 65 纳米和 45 纳米工艺。通过该协议,用户可以从 ARM Access Library Program 获得针对 TSMC 先进技术的 ARM Advantage 产品。

ARM Advantage IP 提供高速、低功耗的性能表现,满足消费电子、通讯和网络市场众多应用的需求。Advantage 标准单元包括功耗管理工具包,实现动态和漏泄功耗节省技术,例如时钟门控、多电压分离和电能门控等。它还提供五个 Advantage 存储编译器,提供相似的高级功耗节省特性。该产品套件在扩展的电压范围内对时钟和功耗作了特殊设置,使得设计师可以完成多电压设计的精确模拟。此外,ARM 还将发布 TSMC Nexsys I/O 产品,从而提供完整的物理 IP。