

版权声明: 本文转载自 [CSDN · BLOG · 陈皓专栏](#), 所有权利归原作者所有。制作本 PDF 之目的仅限于方便学习。

编程修养

概述

什么是好的程序员? 是不是懂得很多技术细节? 还是懂底层编程? 还是编程速度比较快? 我觉得都不是。对于一些技术细节来说和底层的技术, 只要看帮助, 查资料就能找到, 对于速度快, 只要编得多也就熟能生巧了。

我认为好的程序员应该有几方面的素质:

- 1、有专研精神, 勤学善问、举一反三。
- 2、积极向上的态度, 有创造性思维。
- 3、与人积极交流沟通的能力, 有团队精神。
- 4、谦虚谨慎, 戒骄戒躁。
- 5、写出的代码质量高。包括: 代码的稳定、易读、规范、易维护、专业。

这些都是程序员的修养, 这里我想谈谈“编程修养”, 也就是上述中的第 5 点。我觉得, 如果我要了解一个作者, 我会看他所写的小说, 如果我要了解一个画家, 我会看他所画的图画, 如果我要了解一个工人, 我会看他所做出来的产品, 同样, 如果我要了解一个程序员, 我想首先我最想看的就是他的程序代码, 程序代码可以看出一个程序员的素质和修养, 程序就像一个作品, 有素质有修养的程序员的作品必然是一图精美的图画, 一首美妙的歌曲, 一本赏心悦目的小说。

我看过许多程序, 没有注释, 没有缩进, 胡乱命名的变量名, 等等, 等等, 我把这种人统称为没有修养的程序, 这种程序员, 是在做创造性的工作吗? 不, 完全就是在搞破坏, 他们与其说是在编程, 还不如说是在对源程序进行“加密”, 这种程序员, 见一个就应该开除一个, 因为他编的程序所创造的价值, 远远小于需要上面进行维护的价值。

程序员应该有程序员的修养, 那怕再累, 再没时间, 也要对自己的程序负责。我宁可要那种动作慢, 技术一般, 但有好的写程序风格的程序员, 也不要那种技术强、动作快的“搞破坏”的程序员。有句话叫“字如其人”, 我想从程序上也能看出一个程序员的优劣。因为, 程序是程序员的作品, 作品的好坏直接关系到程序员的声誉和素质。而“修养”好的程序员一定能做出好的程序和软件。

有个成语叫“独具匠心”, 意思是做什么都要做得很专业, 很用心, 如果你要做一个“匠”, 也就是造诣高深的人, 那么, 从一件很简单的作品上就能看出你有没有“匠”的特性, 我觉得做一个程序员不难, 但要做一个“程序匠”就不简单了。编程序很简单, 但编出有质量的程序就难了。

我在这里不讨论过深的技术,我只想在一些容易让人忽略的东西上说一说,虽然这些东西可能很细微,但如果你不注意这些细微之处的话,那么他将会极大的影响你的整个软件质量,以及整个软件程的实施,所谓“千里之堤,毁于蚁穴”。

“细微之处见真功”,真正能体现一个程序的功底恰恰在这些细微之处。

这就是程序员的——编程修养。我总结了在用 C/C++语言(主要是 C 语言)进行程序写作上的三十二个“修养”,通过这些,你可以写出质量高的程序,同时也会让看你程序的人啧啧称道,那些看过你程序的人一定会说:“这个人的编程修养不错”。

-
- 01、版权和版本
 - 02、缩进、空格、换行、空行、对齐
 - 03、程序注释
 - 04、函数的[in][out]参数
 - 05、对系统调用的返回进行判断
 - 06、if 语句对出错的处理
 - 07、头文件中的#ifndef
 - 08、在堆上分配内存
 - 09、变量的初始化
 - 10、h 和 c 文件的使用
 - 11、出错信息的处理
 - 12、常用函数和循环语句中的被计算量
 - 13、函数名和变量名的命名
 - 14、函数的传值和传指针
 - 15、修改别人程序的修养
 - 16、把相同或近乎相同的代码形成函数和宏
 - 17、表达式中的括号
 - 18、函数参数中的 const
 - 19、函数的参数个数
 - 20、函数的返回类型,不要省略
 - 21、goto 语句的使用
 - 22、宏的使用
 - 23、static 的使用
 - 24、函数中的代码尺寸
 - 25、typedef 的使用
 - 26、为常量声明宏
 - 27、不要为宏定义加分号
 - 28、||和&&的语句执行顺序
 - 29、尽量用 for 而不是 while 做循环
 - 30、请 sizeof 类型而不是变量
 - 31、不要忽略 Warning

32、书写 Debug 版和 Release 版的程序

1、版权和版本

好的程序员会给自己的每个函数，每个文件，都注上版权和版本。

对于 C/C++ 的文件，文件头应该有类似这样的注释：

```
/******  
*  
* 文件名: network.c  
*  
* 文件描述: 网络通讯函数集  
*  
* 创建人: Hao Chen, 2003 年 2 月 3 日  
*  
* 版本号: 1.0  
*  
* 修改记录:  
*  
*****/
```

而对于函数来说，应该也有类似于这样的注释：

```
/*=====  
*  
* 函 数 名: XXX  
*  
* 参 数:  
*  
*      type name [IN] : descripts  
*  
* 功能描述:  
*  
*      .....  
*  
* 返 回 值: 成功 TRUE, 失败 FALSE  
*  
* 抛出异常:  
*  
* 作 者: ChenHao 2003/4/2  
*  
=====*/
```

这样的描述可以让人对一个函数, 一个文件有一个总体的认识, 对代码的易读性和易维护性有很大的好处。这是好的作品产生的开始。

2、缩进、空格、换行、空行、对齐

i) 缩进应该是每个程序都会做的, 只要学程序过程就应该知道这个, 但是我仍然看过不缩进的程序, 或是乱缩进的程序, 如果你的公司还有写程序不缩进的程序员, 请毫不犹豫的开除他吧, 并以破坏源码罪起诉他, 还要他赔偿读过他程序的人的精神损失费。缩进, 这是不成文规矩, 我再重提一下吧, 一个缩进一般是一个 TAB 键或是 4 个空格。(最好用 4 个空格) <--由网友 vector_3d 提醒改正

ii) 空格。空格能给程序带来什么损失吗? 没有, 有效的利用空格可以让你的程序读进来更加赏心悦目。而不一堆表达式挤在一起。看看下面的代码:

```
ha=(ha*128+*key++)%tabPtr->size;
```

```
ha = ( ha * 128 + *key++ ) % tabPtr->size;
```

有空格和没有空格的感觉不一样吧。一般来说, 语句中要在各个操作符间加空格, 函数调用时, 要以各个参数间加空格。如下面这种加空格的和不加的:

```
if ((hProc=OpenProcess(PROCESS_ALL_ACCESS,FALSE,pid))==NULL){
}
if ( ( hProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid) ) == NULL ){
}
```

iii) 换行。不要把语句都写在一行上, 这样很不好。如:

```
for(i=0;i<len;i++) if((a[i]<'0' || a[i]>'9') && (a[i]<'a' || a[i]>'z')) break;
```

我拷, 这种即无空格, 又无换行的程序在写什么啊? 加上空格和换行吧。

```
for ( i=0; i<len; i++) {
    if ( ( a[i] < '0' || a[i] > '9' ) &&
        ( a[i] < 'a' || a[i] > 'z' ) ) {
        break;
    }
}
```

好多了吧? 有时候, 函数参数多的时候, 最好也换行, 如:

```
CreateProcess(
    NULL,
    cmdbuf,
    NULL,
    NULL,
    bInhH,
    dwCrtFlags,
    envbuf,
```

```

        NULL,
        &siStartInfo,
        &prInfo
    );

```

条件语句也应该在必要时换行:

```

if ( ch >= '0' || ch <= '9' ||
     ch >= 'a' || ch <= 'z' ||
     ch >= 'A' || ch <= 'Z' )

```

iv) 空行。不要不加空行，空行可以区分不同的程序块，程序块间，最好加上空行。如:

```

    HANDLE hProcess;
    PROCESS_T procInfo;
    /* open the process handle */
    if((hProcess = OpenProcess(PROCESS_ALL_ACCESS, FALSE, pid)) == NULL)
    {
        return LSE_MISC_SYS;
    }
    memset(&procInfo, 0, sizeof(procInfo));
    procInfo.idProc = pid;
    procInfo.hdProc = hProcess;
    procInfo.misc |= MSCAVA_PROC;
    return(0);

```

v) 对齐。用 TAB 键对齐你的一些变量的声明或注释，一样会让你的程序好看一些。如:

```

typedef struct _pt_man_t {
    int    numProc;    /* Number of processes          */
    int    maxProc;    /* Max Number of processes      */
    int    numEvt;     /* Number of events             */
    int    maxEvt;     /* Max Number of events         */
    HANDLE* pHndEvt;   /* Array of events              */
    DWORD  timeout;    /* Time out interval            */
    HANDLE hPipe;      /* Namedpipe                    */
    TCHAR  usr[MAXUSR];/* User name of the process      */
    int    numMsg;     /* Number of Message            */
    int    Msg[MAXMSG];/* Space for intro process communicate */
} PT_MAN_T;

```

怎么样？感觉不错吧。

这里主要讲述了如果写出让让人赏心悦目的代码，好看的代码会让人的心情愉快，读起代码也就不累，工整、整洁的程序代码，通常更让人欢迎，也更让人称道。现在的硬盘空间这么大，不要让你的代码挤在一起，这样它们会抱怨你虐待它们的。好了，用“缩进、空格、换行、空行、对齐”装饰你的代码吧，让他们从没有秩序的土匪中变成一排排整齐有秩序的正规部

队吧。

3、程序注释

养成写程序注释的习惯，这是每个程序员所必须要做的工作。我看过那种几千行，却居然没有一行注释的程序。这就如同在公路上驾车却没有路标一样。用不了多久，连自己都不知道自己的意图了，还要花上几倍的时间才看明白，这种浪费别人和自己的时间的人，是最为可耻的人。

是的，你也许会说，你会写注释，真的吗？注释的书写也能看出一个程序员的功底。一般来说你需要至少写这些地方的注释：文件的注释、函数的注释、变量的注释、算法的注释、功能块的程序注释。主要就是记录你这段程序是干什么的？你的意图是什么？你这个变量是用来做什么的？等等。

不要以为注释好写，有一些算法是很难说或写出来的，只能意会，我承认有这种情况的时候，但你也要写出来，正好可以训练一下自己的表达能力。而表达能力正是那种闷头搞技术的技术人员最缺的，你有再高的技术，如果你表达能力不行，你的技术将不能得到充分的发挥。因为，这是一个团队的时代。

好了，说几个注释的技术细节：

i) 对于行注释（“//”）比块注释（“/* */”）要好的说法，我并不是很同意。因为一些老版本的 C 编译器并不支持行注释，所以为了你的程序的移植性，请你还是尽量使用块注释。

ii) 你也许会为块注释的不能嵌套而不爽，那么你可以用预编译来完成这个功能。使用“#if 0”和“#endif”括起来的代码，将不被编译，而且还可以嵌套。

4、函数的[in][out]参数

我经常看到这样的程序：

```
FuncName(char* str)
{
    int len = strlen(str);
    .....
}
char*
GetUserName(struct user* pUser)
{
    return pUser->name;
}
```

不！请不要这样做。

你应该先判断一下传进来的那个指针是不是为空。如果传进来的指针为空的话,那么,你的一个大的系统就会因为这一个小函数而崩溃。一种更好的技术是使用断言(assert),这里我就不多说这些技术细节了。当然,如果是在C++中,引用要比指针好得多,但你也需要对各个参数进行检查。

写有参数的函数时,首要工作,就是要对传进来的所有参数进行合法性检查。而对于传出的参数也应该进行检查,这个动作当然应该在函数的外部,也就是说,调用完一个函数后,应该对其传出的值进行检查。

当然,检查会浪费一点时间,但为了整个系统不至于出现“非法操作”或是“Core Dump”的系统级的错误,多花这点时间还是很值得的。

5、对系统调用的返回进行判断

继续上一条,对于一些系统调用,比如打开文件,我经常看到,许多程序员对fopen返回的指针不做任何判断,就直接使用了。然后发现文件的内容怎么也读出不,或是怎么也写不进去。还是判断一下吧:

```
fp = fopen("log.txt", "a");
if ( fp == NULL ){
    printf("Error: open file error\n");
    return FALSE;
}
```

其它还有许多啦,比如:socket返回的socket号,malloc返回的内存。请对这些系统调用返回的东西进行判断。

6、if 语句对出错的处理

我看见你说了,这有什么好说的。还是先看一段程序代码吧。

```
if ( ch >= '0' && ch <= '9' ){
    /* 正常处理代码 */
}else{
    /* 输出错误信息 */
    printf("error ..... \n");
    return ( FALSE );
}
```

这种结构很不好,特别是如果“正常处理代码”很长时,对于这种情况,最好不要用else。

先判断错误, 如:

```
if ( ch < '0' || ch > '9' ){
    /* 输出错误信息 */
    printf("error .....\\n");
    return ( FALSE );
}

/* 正常处理代码 */
.....
```

这样的结构, 不是很清楚吗? 突出了错误的条件, 让别人在使用你的函数的时候, 第一眼就能看到不合法的条件, 于是就会更下意识的避免。

7、头文件中的#ifndef

千万不要忽略了头文件中的#ifndef, 这是一个很关键的东西。比如你有两个 C 文件, 这两个 C 文件都 include 了同一个头文件。而编译时, 这两个 C 文件要一同编译成一个可运行文件, 于是问题来了, 大量的声明冲突。

还是把头文件的内容都放在#ifndef 和#endif 中吧。不管你的头文件会不会被多个文件引用, 你都要加上这个。一般格式是这样的:

```
#ifndef <标识>
#define <标识>

.....

.....

#endif
```

<标识>在理论上来说可以是自由命名的, 但每个头文件的这个“标识”都应该是唯一的。标识的命名规则一般是头文件名全大写, 前后加下划线, 并把文件名中的“.”也变成下划线, 如: stdio.h

```
#ifndef _STDIO_H_
#define _STDIO_H_

.....

#endif
```

(BTW: 预编译有多很有用的功能。你会用预编译吗?)

8、在堆上分配内存

可能许多人对内存分配上的“栈 stack”和“堆 heap”还不是很明白。包括一些科班出身的人也不明白这两个概念。我不想过多的说这两个东西。简单的来讲, stack 上分配的内存系统自动释放, heap 上分配的内存, 系统不释放, 哪怕程序退出, 那一块内存还是在那里。stack 一般是静态分配内存, heap 上一般是动态分配内存。

由 malloc 系统函数分配的内存就是从堆上分配内存。从堆上分配的内存一定要自己释放。用 free 释放, 不然就是术语——“内存泄露”(或是“内存漏洞”)——Memory Leak。于是, 系统的可分配内存会随 malloc 越来越少, 直到系统崩溃。还是来看看“栈内存”和“堆内存”的差别吧。

栈内存分配

```
char*
AllocStrFromStack()
{
    char pstr[100];
    return pstr;
}
```

堆内存分配

```
char*
AllocStrFromHeap(int len)
{
    char *pstr;

    if ( len <= 0 ) return NULL;
    return ( char* ) malloc( len );
}
```

对于第一个函数, 那块 pstr 的内存存在函数返回时就被系统释放了。于是所返回的 char* 什么也没有。而对于第二个函数, 是从堆上分配内存, 所以哪怕是程序退出时, 也不释放, 所以第二个函数的返回的内存没有问题, 可以被使用。但一定要调用 free 释放, 不然就是 Memory Leak!

在堆上分配内存很容易造成内存泄漏, 这是 C/C++ 的最大的“克星”, 如果你的程序要稳定, 那么就不要再出现 Memory Leak。所以, 我还是要在这一千叮咛万嘱咐, 在使用 malloc 系统函数(包括 calloc, realloc)时千万要小心。

记得有一个 UNIX 上的服务应用程序, 大约有几百的 C 文件编译而成, 运行测试良好, 等使用时, 每隔三个月系统就是 **down** 一次, 搞得许多人焦头烂额, 查不出问题所在。只好, 每隔两个月人工手动重启系统一次。出现这种问题就是 **Memory Leak** 在做怪了, 在 C/C++ 中这种问题总是会发生, 所以你一定要小心。一个 Rational 的检测工作——**Purify**, 可以帮你测试你的程序有没有内存泄漏。

我保证, 做过许多 C/C++ 的工程, 都会对 **malloc** 或是 **new** 有些感冒。当你什么时候在使用 **malloc** 和 **new** 时, 有一种轻度的紧张和惶恐的感觉时, 你就具备了这方面的修养了。

对于 **malloc** 和 **free** 的操作有以下规则:

- 1) 配对使用, 有一个 **malloc**, 就应该有一个 **free**。(C++ 中对应为 **new** 和 **delete**)
- 2) 尽量在同一层上使用, 不要像上面那种, **malloc** 在函数中, 而 **free** 在函数外。最好在同一调用层上使用这两个函数。
- 3) **malloc** 分配的内存一定要初始化。**free** 后的指针一定要设置为 **NULL**。

注: 虽然现在的操作系统(如: UNIX 和 Win2k/NT)都有进程内存跟踪机制, 也就是如果你有没有释放的内存, 操作系统会帮你释放。但操作系统依然不会释放你程序中所有产生了 **Memory Leak** 的内存, 所以, 最好还是你自己来做这个工作。(有的时候不知不觉就出现 **Memory Leak** 了, 而且在几百万行的代码中找无异于海底捞针, Rational 有一个工具叫 **Purify**, 可能很好的帮你检查程序中的 **Memory Leak**)

9、变量的初始化

接上一条, 变量一定要被初始化再使用。C/C++ 编译器在这个方面不会像 **JAVA** 一样帮你初始化, 这一切都需要你自己来, 如果你使用了没有初始化的变量, 结果未知。好的程序员从来都会在使用变量前初始化变量的。如:

- 1) 对 **malloc** 分配的内存进行 **memset** 清零操作。(可以使用 **calloc** 分配一块全零的内存)
- 2) 对一些栈上分配的 **struct** 或数组进行初始化。(最好也是清零)

不过话又说回来了, 初始化也会造成系统运行时间有一定的开销, 所以, 也不要对所有的变量做初始化, 这个也没有意义。好的程序员知道哪些变量需要初始化, 哪些则不需要。如: 以下这种情况, 则不需要。

```
char *pstr; /* 一个字符串 */
pstr = (char*) malloc( 50 );
if ( pstr == NULL ) exit(0);
strcpy( pstr, "Hello Wrold" );
```

但如果是下面一种情况, 最好进行内存初始化。(指针是一个危险的东西, 一定要初始化)

```
char **pstr; /* 一个字符串数组 */
pstr = (char**) malloc( 50 );
if ( pstr == NULL ) exit(0);

/* 让数组中的指针都指向 NULL */
memset( pstr, 0, 50*sizeof(char*) );
```

而对于全局变量, 和静态变量, 一定要声明时就初始化。因为你不知道它第一次会在哪里被使用。所以使用前初始这些变量是比较不现实的, 一定要在声明时就初始化它们。如:

```
Links *plnk = NULL; /* 对于全局变量 plnk 初始化为 NULL */
```

10、h 和 c 文件的使用

H 文件和 C 文件怎么用呢? 一般来说, H 文件中是 **declare** (声明), C 文件中是 **define** (定义)。因为 C 文件要编译成库文件 (Windows 下是 .obj/.lib, UNIX 下是 .o/.a), 如果别人要使用你的函数, 那么就要引用你的 H 文件, 所以, H 文件中一般是变量、宏定义、枚举、结构和函数接口的声明, 就像一个接口说明文件一样。而 C 文件则是实现细节。

H 文件和 C 文件最大的用处就是声明和实现分开。这个特性应该是公认的了, 但我仍然看到有些人喜欢把函数写在 H 文件中, 这种习惯很不好。(如果是 C++ 话, 对于其模板函数, 在 VC 中只有把实现和声明都写在一个文件中, 因为 VC 不支持 **export** 关键字)。而且, 如果在 H 文件中写上函数的实现, 你还得在 **makefile** 中把头文件的依赖关系也加上, 这个就会让你的 **makefile** 很不规范。

最后, 有一个最需要注意的地方就是: 带初始化的全局变量不要放在 H 文件中!

例如有一个处理错误信息的结构:

```
char* errmsg[] = {
    /* 0 */    "No error",
    /* 1 */    "Open file error",
    /* 2 */    "Failed in sending/receiving a message",
    /* 3 */    "Bad arguments",
    /* 4 */    "Memeroy is not enough",
    /* 5 */    "Service is down; try later",
    /* 6 */    "Unknow information",
    /* 7 */    "A socket operation has failed",
    /* 8 */    "Permission denied",
    /* 9 */    "Bad configuration file format",
    /* 10 */   "Communication time out",
    .....
}
```

```
.....  
};
```

请不要把这个东西放在头文件中, 因为如果你的这个头文件被 5 个函数库 (.lib 或是 .a) 所用到, 于是他就被链接在这 5 个 .lib 或 .a 中, 而如果你的一个程序用到了这 5 个函数库中的函数, 并且这些函数都用到了这个出错信息数组。那么这份信息将有 5 个副本存在于你的执行文件中。如果你的这个 `errmsg` 很大的话, 而且你用到的函数库更多的话, 你的执行文件也会变得很大。

正确的写法应该把它写到 C 文件中, 然后在各个需要用到 `errmsg` 的 C 文件头上加上 `extern char* errmsg[];` 的外部声明, 让编译器在链接时才去管他, 这样一来, 就只会会有一个 `errmsg` 存在于执行文件中, 而且, 这样做很利于封装。

我曾遇到过的最疯狂的事, 就是在我的目标文件中, 这个 `errmsg` 一共有 112 个副本, 执行文件有 8M 左右。当我把 `errmsg` 放到 C 文件中, 并为一千多个 C 文件加上了 `extern` 的声明后, 所有的函数库文件尺寸都下降了 20% 左右, 而我的执行文件只有 5M 了。一下子少了 3M 啊。

[备注]

有朋友对我说, 这个只是一个特例, 因为, 如果 `errmsg` 在执行文件中存在多个副本时, 可以加快程序运行速度, 理由是 `errmsg` 的多个复本会让系统的内存换页降低, 达到效率提升。像我们这里所说的 `errmsg` 只有一份, 当某函数要用 `errmsg` 时, 如果内存隔得比较远, 会产生换页, 反而效率不高。

这个说法不无道理, 但是一般而言, 对于一个比较大的系统, `errmsg` 是比较大的, 所以产生副本导致执行文件尺寸变大, 不仅增加了系统装载时间, 也会让一个程序在内存中占更多的页面。而对于 `errmsg` 这样数据, 一般来说, 在系统运行时不会经常用到, 所以还是产生的内存换页也就不算频繁。权衡之下, 还是只有一份 `errmsg` 的效率。即便是像 `logmsg` 这样频繁使用的的数据, 操作系统的内存调度算法会让这样的频繁使用的页面常驻于内存, 所以也就不会出现内存换页问题了。

11、出错信息的处理

你会处理出错信息吗? 哦, 它并不是简单的输出。看下面的示例:

```
if ( p == NULL ){  
    printf ( "ERR: The pointer is NULL\n" );  
}
```

告别学生时代的编程吧。这种编程很不利于维护和管理, 出错信息或是提示信息, 应该统一处理, 而不是像上面这样, 写成一个“硬编码”。第 10 条对这方面的处理做了一部分说明。如果要管理错误信息, 那就要有以下的处理:

```
/* 声明出错代码 */
#define ERR_NO_ERROR 0 /* No error */
#define ERR_OPEN_FILE 1 /* Open file error */
#define ERR_SEND_MESG 2 /* sending a message error */
#define ERR_BAD_ARGS 3 /* Bad arguments */
#define ERR_MEM_NONE 4 /* Memeroy is not enough */
#define ERR_SERV_DOWN 5 /* Service down try later */
#define ERR_UNKNOW_INFO 6 /* Unknow information */
#define ERR_SOCKET_ERR 7 /* Socket operation failed */
#define ERR_PERMISSION 8 /* Permission denied */
#define ERR_BAD_FORMAT 9 /* Bad configuration file */
#define ERR_TIME_OUT 10 /* Communication time out */
```

```
/* 声明出错信息 */
```

```
char* errmsg[] = {
    /* 0 */ "No error",
    /* 1 */ "Open file error",
    /* 2 */ "Failed in sending/receiving a message",
    /* 3 */ "Bad arguments",
    /* 4 */ "Memeroy is not enough",
    /* 5 */ "Service is down; try later",
    /* 6 */ "Unknow information",
    /* 7 */ "A socket operation has failed",
    /* 8 */ "Permission denied",
    /* 9 */ "Bad configuration file format",
    /* 10 */ "Communication time out",
};
```

```
/* 声明错误代码全局变量 */
```

```
long errno = 0;
```

```
/* 打印出错信息函数 */
```

```
void perror( char* info)
{
    if ( info ){
        printf("%s: %s\n", info, errmsg[errno] );
        return;
    }

    printf("Error: %s\n", errmsg[errno] );
}
```

这个基本上是 ANSI 的错误处理实现细节了, 于是当你程序中有错误时你就可以这样处理:

```
bool CheckPermission( char* userName )
```

```
{
    if ( strcpy(userName, "root") != 0 ){
        errno = ERR_PERMISSION_DENIED;
        return (FALSE);
    }

    ...
}

main()
{
    ...
    if (! CheckPermission( username )){
        perror("main()");
    }
    ...
}
```

一个即有共性，也有个性的错误信息处理，这样做有利同种错误出一样的信息，统一用户界面，而不会因为文件打开失败，A 程序员出一个信息，B 程序员又出一个信息。而且这样做，非常容易维护。代码也易读。

当然，物极必反，也没有必要把所有的输出都放到 `errmsg` 中，抽取比较重要的出错信息或是提示信息是其关键，但即使这样，这也包括了大多数的信息。

12、常用函数和循环语句中的被计算量

看一下下面这个例子：

```
for( i=0; i<1000; i++){
    GetLocalHostName( hostname );
    ...
}
```

`GetLocalHostName` 的意思是取得当前计算机名，在循环体中，它会被调用 1000 次啊。这是多么的没有效率的事啊。应该把这个函数拿到循环体外，这样只调用一次，效率得到了很大的提高。虽然，我们的编译器会进行优化，会把循环体内的不变的东西拿到循环外面，但是，你相信所有编译器会知道哪些是不变的吗？我觉得编译器不可靠。最好还是自己动手吧。同样，对于常用函数中的不变量，如：

```
GetLocalHostName(char* name)
{
    char funcName[] = "GetLocalHostName";

    sys_log( "%s begin.....", funcName );
```

```
...
sys_log( "%s end.....", funcName );
}
```

如果这是一个经常调用的函数，每次调用时都要对 `funcName` 进行分配内存，这个开销很大啊。把这个变量声明成 `static` 吧，当函数再次被调用时，就会省去了分配内存的开销，执行效率也很好。

13、函数名和变量名的命名

我看到许多程序对变量名和函数名的取名很草率，特别是变量名，什么 `a,b,c,aa,bb,cc`，还有什么 `flag1,flag2,cnt1,cnt2`，这同样是一种没有“修养”的行为。即便加上好的注释。好的变量名或是函数名，我认为应该有以下的规则：

- 1) 直观并且可以拼读，可望文知意，不必“解码”。
- 2) 名字的长度应该即要最短的长度，也要能最大限度的表达其含义。
- 3) 不要全部大写，也不要全部小写，应该大小写都有，如：`GetLocalHostName` 或是 `UserAccount`。
- 4) 可以简写，但简写得要让人明白，如：`ErrorCode` -> `ErrCode`, `ServerListener` -> `ServLisner`, `UserAccount` -> `UsrAcct` 等。
- 5) 为了避免全局函数和变量名字冲突，可以加上一些前缀，一般以模块简称做为前缀。
- 6) 全局变量统一加一个前缀或是后缀，让人一看到这个变量就知道是全局的。
- 7) 用匈牙利命名法命名函数参数，局部变量。但还是要坚持“望文生意”的原则。
- 8) 与标准库（如：`STL`）或开发库（如：`MFC`）的命名风格保持一致。

14、函数的传值和传指针

向函数传参数时，一般而言，传入非 `const` 的指针时，就表示，在函数中要修改这个指针把指内存中的数据。如果是传值，那么无论在函数内部怎么修改这个值，也影响不到传过来的值，因为传值是只内存拷贝。

什么？你说这个特性你明白了，好吧，让我们看看下面的这个例程：

```
void
GetVersion(char* pStr)
{
    pStr = malloc(10);
    strcpy ( pStr, "2.0" );
}
main()
{
```

```
char* ver = NULL;
GetVersion ( ver );
...
...
free ( ver );
}
```

我保证, 类似这样的问题是一个新手最容易犯的错误。程序中妄图通过函数 `GetVersion` 给指针 `ver` 分配空间, 但这种方法根本没有什么作用, 原因就是——这是传值, 不是传指针。你或许会和我争论, 我分明传的时指针啊? 再仔细看看, 其实, 你传的是指针其实是在传值。

15、修改别人程序的修养

当你维护别人的程序时, 请不要非常主观臆断的把已有的程序删除或是修改。我经常看到有的程序员直接在别人的程序上修改表达式或是语句。修改别人的程序时, 请不要删除别人的程序, 如果你觉得别人的程序有所不妥, 请注释掉, 然后添加自己的处理程序, 毕竟, 你不可能 100% 的知道别人的意图, 所以为了可以恢复, 请不要依赖于 CVS 或是 SourceSafe 这种版本控制软件, 还是要在源码上给别人看到你修改程序的意图和步骤。这是程序维护时, 一个有修养的程序员所应该做的。

如下所示, 这就是一种比较好的修改方法:

```
/*
 * ----- commented by haoel 2003/04/12 -----
 *
 * char* p = ( char* ) malloc( 10 );
 * memset( p, 0, 10 );
 */

/* ----- Added by haoel 2003/04/12 ----- */
char* p = ( char* ) calloc( 10, sizeof char );
/* ----- */
...
```

当然, 这种方法是在软件维护时使用的, 这样的方法, 可以让再维护的人很容易知道以前的代码更改的动作和意图, 而且这也是对原作者的一种尊敬。

以“注释 — 添加”方式修改别人的程序, 要好于直接删除别人的程序。

16、把相同或近乎相同的代码形成函数和宏

有人说, 最好的程序员, 就是最喜欢“偷懒”的程序, 其中不无道理。

如果你有一些程序的代码片段很相似,或直接就是一样的,请把他们放在一个函数中。而如这段代码不多,而且会被经常使用,你还想避免函数调用的开销,那么就把它写成宏吧。

千万不要让同一份代码或是功能相似的代码在多个地方存在,不然如果功能一变,你就要修改好几处地方,这种会给维护带来巨大的麻烦,所以,做到“一改百改”,还是要形成函数或是宏。

17、表达式中的括号

如果一个比较复杂的表达式中,你并不是很清楚各个操作符的优先级,即使是你很清楚优先级,也请加上括号,不然,别人或是自己下一次读程序时,一不小心就看走眼理解错了,为了避免这种“误解”,还有让自己的程序更为清晰,还是加上括号吧。

比如,对一个结构的成员取地址:

```
GetUserAge( &(amp; UserInfo->age) );
```

虽然, `&UserInfo->age` 中, `->` 操作符的优先级最高,但加上一个括号,会让人一眼就看明白你的代码是什么意思。

再比如,一个很长的条件判断:

```
if ( ( ch[0] >= '0' || ch[0] <= '9' ) &&  
      ( ch[1] >= 'a' || ch[1] <= 'z' ) &&  
      ( ch[2] >= 'A' || ch[2] <= 'Z' ) )
```

括号,再加上空格和换行,你的代码是不是很容易读懂了?

18、函数参数中的 const

对于一些函数中的指针参数,如果在函数中只读,请将其用 `const` 修饰,这样,别人一读到你的函数接口时,就会知道你的意图是这个参数是[in],如果没有 `const` 时,参数表示[in/out],注意函数接口中的 `const` 使用,利于程序的维护和避免犯一些错误。

虽然, `const` 修饰的指针,如: `const char* p`,在 C 中一点用也没有,因为不管你的声明是不是 `const`,指针的内容照样能改,因为编译器会强制转换,但是加上这样一个说明,有利于程序的阅读和编译。因为在 C 中,修改一个 `const` 指针所指向的内存时,会报一个 `Warning`。这会吸引起程序员的注意。

C++中对 `const` 定义的就严格了,所以 C++中要多多的使用 `const`, `const` 的成员函数, `const`

的变量，这样会让你的代码和你的程序更加完整和易读。（关于 C++ 的 `const` 我就不多说了）

19、函数的参数个数（多了请用结构）

函数的参数个数最好不要太多，一般来说 6 个左右就可以了，众多的函数参数会让读代码的人一眼看上去就很头昏，而且也不利于维护。如果参数众多，还请使用结构来传递参数。这样做有利于数据的封装和程序的简洁性。

也利于使用函数的人，因为如果你的函数个数很多，比如 12 个，调用者很容易搞错参数的顺序和个数，而使用结构 `struct` 来传递参数，就可以不管参数的顺序。

而且，函数很容易被修改，如果需要给函数增加参数，不需要更改函数接口，只需更改结构体和函数内部处理，而对于调用函数的程序来说，这个动作是透明的。

20、函数的返回类型，不要省略

我看到很多程序写函数时，在函数的返回类型方面不太注意。如果一个函数没有返回值，也请在函数前面加上 `void` 的修饰。而有的程序员偷懒，在返回 `int` 的函数则什么都不修饰（因为如果不修饰，则默认返回 `int`），这种习惯很不好，还是为了原代码的易读性，加上 `int` 吧。

所以函数的返回值类型，请不要省略。

另外，对于 `void` 的函数，我们往往会忘了 `return`，由于某些 C/C++ 的编译器比较敏感，会报一些警告，所以即使是 `void` 的函数，我们在内部最好也要加上 `return` 的语句，这有助于代码的编译。

21、goto 语句的使用

N 年前，软件开发的一代宗师——迪杰斯特拉(Dijkstra)说过：“`goto statment is harmful !!`”，并建议取消 `goto` 语句。因为 `goto` 语句不利于程序代码的维护性。

这里我也强烈建议不要使用 `goto` 语句，除非下面的这种情况：

```
#define FREE(p) if(p) { \
    free(p); \
    p = NULL; \
}
```

```
    }

main()
{
    char *fname=NULL, *lname=NULL, *mname=NULL;
    fname = ( char* ) calloc ( 20, sizeof(char) );
    if ( fname == NULL ){
        goto ErrHandle;
    }
    lname = ( char* ) calloc ( 20, sizeof(char) );
    if ( lname == NULL ){
        goto ErrHandle;
    }
    mname = ( char* ) calloc ( 20, sizeof(char) );
    if ( mname == NULL ){
        goto ErrHandle;
    }

    .....

ErrHandle:
    FREE(fname);
    FREE(lname);
    FREE(mname);
    ReportError(ERR_NO_MEMOEY);
}
```

也只有在这种情况下，goto 语句会让你的程序更易读，更容易维护。（在用嵌 C 来对数据库设置游标操作时，或是对数据库建立链接时，也会遇到这种结构）

22、宏的使用

很多程序员不知道 C 中的“宏”到底是什么意思？特别是当宏有参数的时候，经常把宏和函数混淆。我想在这里我还是先讲讲“宏”，宏只是一种定义，他定义了一个语句块，当程序编译时，编译器首先要执行一个“替换”源程序的动作，把宏引用的地方替换成宏定义的语句块，就像文本文件替换一样。这个动作术语叫“宏的展开”

使用宏是比较“危险”的，因为你不知道宏展开后会是什么一个样子。例如下面这个宏：

```
#define MAX(a, b)    a>b?a:b
```

当我们这样使用宏时，没有什么问题： MAX(num1, num2); 因为宏展开后变成 num1>num2?num1:num2;。 但是，如果是这样调用的，MAX(17+32, 25+21); 呢，编译时

出现错误, 原因是, 宏展开后变成: $17+32>25+21?17+32:25+21$, 哇, 这是什么啊?

所以, 宏在使用时, 参数一定要加上括号, 上述的那个例子改成如下所示就能解决问题了。

```
#define MAX( (a), (b) )    (a)>(b)?(a):(b)
```

即使是这样, 也不这个宏也还是有 Bug, 因为如果我这样调用 `MAX(i++, j++)`, 经过这个宏以后, `i` 和 `j` 都被累加了两次, 这绝不是我们想要的。

所以, 在宏的使用上还是要谨慎考虑, 因为宏展开是的结果是很难让人预料的。而且虽然, 宏的执行很快 (因为没有函数调用的开销), 但宏会让源代码膨胀, 使目标文件尺寸变大, (如: 一个 50 行的宏, 程序中有 1000 个地方用到, 宏展开后会很不得了), 相反不能让程序执行得更快 (因为执行文件变大, 运行时系统换页频繁)。

因此, 在决定是用函数, 还是用宏时得要小心。

23、static 的使用

`static` 关键字, 表示了“静态”, 一般来说, 他会被经常用于变量和函数。一个 `static` 的变量, 其实就是全局变量, 只不过他是有作用域的全局变量。比如一个函数中的 `static` 变量:

```
char*
getConsumerName()
{
    static int cnt = 0;

    ....
    cnt++;
    ....
}
```

`cnt` 变量的值会跟随着函数的调用次而递增, 函数退出后, `cnt` 的值还存在, 只是 `cnt` 只能在函数中才能被访问。而 `cnt` 的内存也只会是在函数第一次被调用时才会被分配和初始化, 以后每次进入函数, 都不为 `static` 分配了, 而直接使用上一次的值。

对于一些被经常调用的函数内的常量, 最好也声明成 `static` (参见第 12 条)

但 `static` 的最多的用处却不在这里, 其最大的作用的控制访问, 在 C 中如果一个函数或是一个全局变量被声明为 `static`, 那么, 这个函数和这个全局变量, 将只能在这个 C 文件中被访问, 如果别的 C 文件中调用这个 C 文件中的函数, 或是使用其中的全局 (用 `extern` 关键字), 将会发生链接时错误。这个特性可以用于数据和程序保密。

24、函数中的代码尺寸

一个函数完成一个具体的功能, 一般来说, 一个函数中的代码最好不要超过 600 行左右, 越少越好, 最好的函数一般在 100 行以内, 300 行左右的函数就差不多了。有证据表明, 一个函数中的代码如果超过 500 行, 就会有和别的函数相同或是相近的代码, 也就是说, 就可以再写另一个函数。

另外, 函数一般是完成一个特定的功能, 千万忌讳在一个函数中做许多件不同的事。函数的功能越单一越好, 一方面有利于函数的易读性, 另一方面更有利于代码的维护和重用, 功能越单一表示这个函数就越可能给更多的程序提供服务, 也就是说共性就越多。

虽然函数的调用会有一定的开销, 但比起软件后期维护来说, 增加一些运行时的开销而换来更好的可维护性和代码重用性, 是很值得的一件事。

25、typedef 的使用

typedef 是一个给类型起别名的关键字。不要小看了它, 它对于你代码的维护会有很好的作用。比如 C 中没有 bool, 于是在一个软件中, 一些程序员使用 int, 一些程序员使用 short, 会比较混乱, 最好就是用一个 typedef 来定义, 如:

```
typedef char bool;
```

一般来说, 一个 C 的工程中一定要做一些这方面的工作, 因为你会涉及到跨平台, 不同的平台会有不同的字长, 所以利用预编译和 typedef 可以让你最有效的维护你的代码, 如下所示:

```
#ifdef SOLARIS2_5
    typedef boolean_t    BOOL_T;
#else
    typedef int          BOOL_T;
#endif

typedef short           INT16_T;
typedef unsigned short  UINT16_T;
typedef int             INT32_T;
typedef unsigned int    UINT32_T;

#ifdef WIN32
    typedef _int64       INT64_T;
#else
    typedef long long    INT64_T;
```

```
#endif
```

```
typedef float          FLOAT32_T;  
typedef char*          STRING_T;  
typedef unsigned char  BYTE_T;  
typedef time_t          TIME_T;  
typedef INT32_T        PID_T;
```

使用 `typedef` 的其它规范是, 在结构和函数指针时, 也最好用 `typedef`, 这也有利于程序的易读和可维护性。如:

```
typedef struct _hostinfo {  
    HOSTID_T    host;  
    INT32_T     hostId;  
    STRING_T    hostType;  
    STRING_T    hostModel;  
    FLOAT32_T   cpuFactor;  
    INT32_T     numCPUs;  
    INT32_T     nDisks;  
    INT32_T     memory;  
    INT32_T     swap;  
} HostInfo;  
  
typedef INT32_T (*RsrcReqHandler)(  
    void *info,  
    JobArray *jobs,  
    AllocInfo *allocInfo,  
    AllocList *allocList);
```

C++中这样也是很让人易读的:

```
typedef CArray<HostInfo, HostInfo&> HostInfoArray;
```

于是, 当我们用其定义变量时, 会显得十分易读。如:

```
HostInfo* phinfo;  
RsrcReqHandler* pRsrcHand;
```

这种方式的易读性, 在函数的参数中十分明显。

关键是在程序种使用 `typedef` 后, 几乎所有的程序中的类型声明都显得那么简洁和清晰, 而且易于维护, 这才是 `typedef` 的关键。

26、为常量声明宏

最好不要在程序中出现数字式的“硬编码”，如：

```
int user[120];
```

为这个 120 声明一个宏吧。为所有出现在程序中的这样的常量都声明一个宏吧。比如 TimeOut 的时间，最大的用户数量，还有其它，只要是常量就应该声明成宏。如果，突然在程序中出现下面一段代码，

```
for ( i=0; i<120; i++){  
    ....  
}
```

120 是什么？为什么会是 120？这种“硬编码”不仅让程序很读，而且也让程序很不好维护，如果要改变这个数字，得同时对所有程序中这个 120 都要做修改，这对修改程序的人来说是一个很大的痛苦。所以还是把常量声明成宏，这样，一改百改，而且也很利于程序阅读。

```
#define MAX_USR_CNT 120  
  
for ( i=0; i<MAX_USER_CNT; i++){  
    ....  
}
```

这样就很容易了解这段程序的意图了。

有的程序员喜欢为这种变量声明全局变量，其实，全局变量应该尽量的少用，全局变量不利于封装，也不利于维护，而且对程序执行空间有一定的开销，一不小心就造成系统换页，造成程序执行速度效率等问题。所以声明成宏，即可以免去全局变量的开销，也会有速度上的优势。

27、不要为宏定义加分号

有许多程序员不知道在宏定义时是否要加分号，有时，他们以为宏是一条语句，应该要加分号，这就错了。当你知道了宏的原理，你会赞同我为么不要为宏定义加分号的。看一个例子：

```
#define MAXNUM 1024;
```

这是一个有分号的宏，如果我们这样使用：

```
half = MAXNUM/2;
```

```
if ( num < MAXNUM )
```

等等, 都会造成程序的编译错误, 因为, 当宏展开后, 他会是这个样子的:

```
half = 1024;/2;
```

```
if ( num < 1024; )
```

是的, 分号也被展进去了, 所以造成了程序的错误。请相信我, 有时候, 一个分号会让你的程序出现成百个错误。所以还是不要为宏加最后一个分号, 哪怕是这样:

```
#define LINE "=====
```

```
#define PRINT_LINE printf(LINE)
```

```
#define PRINT_NLINE(n) while ( n-- >0 ) { PRINT_LINE; }
```

都不要在最后加上分号, 当我们在程序中使用时, 为之加上分号,

```
main()
{
    char *p = LINE;
    PRINT_LINE;
}
```

这一点非常符合习惯, 而且, 如果忘加了分号, 编译器给出的错误提示, 也会让我们很容易看懂的。

28、||和&&的语句执行顺序

条件语句中的这两个“与”和“或”操作符一定要小心, 它们的表现可能和你想像的不一样, 这里条件语句中的有些行为需要和说一下:

```
express1 || express2
```

先执行表达式 `express1` 如果为“真”, `express2` 将不被执行, `express2` 仅在 `express1` 为“假”时才被执行。因为第一个表达式为真了, 整个表达式都为真, 所以没有必要再去执行第二个表达式了。

```
express1 && express2
```

先执行表达式 `express1` 如果为“假”, `express2` 将不被执行, `express2` 仅在 `express1` 为“真”

时才被执行。因为第一个表达式为假了，整个表达式都为假了，所以没有必要再去执行第二个表达式了。

于是，他并不是你所想像的所有的表达式都会去执行，这点一定要明白，不然你的程序会出现一些莫名的运行时错误。

例如，下面的程序：

```
if ( sum > 100 &&
    ( ( fp=fopen( filename,"a" ) ) != NULL ) ) {

    fprintf(fp, "Warring: it beyond one hundred\n");
    .....
}

fprintf( fp, " sum is %id \n", sum );
fclose( fp );
```

本来的意图是，如果 `sum > 100`，向文件中写一条出错信息，为了方便，把两个条件判断写在一起，于是，如果 `sum <= 100` 时，打开文件的操作将不会做，最后，`fprintf` 和 `fclose` 就会发现未知的结果。

再比如，如果我想判断一个字符是不是有内容，我得判断这个字符串指针是不为空 (NULL) 并且其内容不能为空 (Empty)，一个是空指针，一个是空内容。我也许会这样写：

```
if ( ( p != NULL ) && ( strlen(p) != 0 ) )
```

于是，如果 `p` 为 NULL，那么 `strlen(p)` 就不会被执行，于是，`strlen` 也就不会因为一个空指针而“非法操作”或是一个“Core Dump”了。

记住一点，条件语句中，并非所有的语句都会执行，当你的条件语句非常多时，这点要尤其注意。

29、尽量用 for 而不是 while 做循环

基本上来说，`for` 可以完成 `while` 的功能，我是建议尽量使用 `for` 语句，而不要使用 `while` 语句，特别是当循环体很大时，`for` 的优点一下就体现出来了。

因为在 `for` 中，循环的初始、结束条件、循环的推进，都在一起，一眼看上去就知道这是一个什么样的循环。刚出学校的程序一般对于链接喜欢这样来：

```
p = pHead;
```

```
while ( p ){  
    ...  
    ...  
    p = p->next;  
}
```

当 while 的语句块变大后, 你的程序将很难读, 用 for 就好得多:

```
for ( p=pHead; p; p=p->next ){  
    ..  
}
```

一眼就知道这个循环的开始条件, 结束条件, 和循环的推进。大约就能明白这个循环要做个什么事? 而且, 程序维护进来很容易, 不必像 while 一样, 在一个编辑器中上上下下的捣腾。

30、请 sizeof 类型而不是变量

许多程序员在使用 sizeof 中, 喜欢 sizeof 变量名, 例如:

```
int score[100];  
char filename[20];  
struct UserInfo usr[100];
```

在 sizeof 这三个的变量名时, 都会返回正确的结果, 于是许多程序员就开始 sizeof 变量名。这个习惯很虽然没有什么不好, 但我还是建议 sizeof 类型。

我看到过这个的程序:

```
pScore = (int*) malloc( SUBJECT_CNT );  
memset( pScore, 0, sizeof(pScore) );  
...
```

此时, sizeof(pScore)返回的就是 4 (指针的长度), 不会是整个数组, 于是, memset 就不能对这块内存进行初始化。为了程序的易读和易维护, 我强烈建议使用类型而不是变量, 如:

```
对于 score:      sizeof(int) * 100  /* 100 个 int */  
对于 filename:   sizeof(char) * 20  /* 20 个 char */  
对于 usr:        sizeof(struct UserInfo) * 100  /* 100 个 UserInfo */
```

这样的代码是不是很容易读? 一眼看上去就知道什么意思了。

另外一点, sizeof 一般用于分配内存, 这个特性特别在多维数组时, 就能体现出其优点了。如, 给一个字符串数组分配内存,

```
/*
 * 分配一个有 20 个字符串,
 * 每个字符串长 100 的内存
 */
char** p;
/*
 * 错误的分配方法
 */
p = (char**)calloc( 20*100, sizeof(char) );

/*
 * 正确的分配方法
 */
p = (char**) calloc ( 20, sizeof(char*) );
for ( i=0; i<20; i++){
    /*p = (char*) calloc ( 100, sizeof(char) );*/
    p[i] = (char*) calloc ( 100, sizeof(char) );
}
```

(注: 上述语句被注释掉的是原来的, 是错误的, 由 dasherest 朋友指正, 谢谢)

为了代码的易读, 省去了一些判断, 请注意这两种分配的方法, 有本质上的差别。

31、不要忽略 Warning

对于一些编译时的警告信息, 请不要忽视它们。虽然, 这些 **Warning** 不会妨碍目标代码的生成, 但这并不意味着你的程序就是好的。毕竟, 并不是编译成功的程序才是正确的, 编译成功只是万里长征的第一步, 后面还有大风大浪在等着你。从编译程序开始, 不但要改正每个 **error**, 还要修正每个 **warning**。这是一个有修养的程序员该做的事。

一般来说, 一面的一些警告信息是常见的:

1) 声明了未使用的变量。(虽然编译器不会编译这种变量, 但还是把它从源程序中注释或是删除吧)

2) 使用了隐晦声明的函数。(也许这个函数在别的 C 文件中, 编译时会出现这种警告, 你应该这使用之前使用 **extern** 关键字声明这个函数)

3) 没有转换一个指针。(例如 **malloc** 返回的指针是 **void** 的, 你没有把之转成你实际类型而报警, 还是手动的在之前明显的转换一下吧)

4) 类型向下转换。(例如: **float f = 2.0;** 这种语句是会报警告的, 编译会告诉你正试图把一个 **double** 转成 **float**, 你正在阉割一个变量, 你真的要这样做吗? 还是在 **2.0** 后面加个 **f** 吧, 不然, **2.0** 就是一个 **double**, 而不是 **float** 了)

不管怎么说, 编译器的 Warning 不要小视, 最好不要忽略, 一个程序都做得出来, 何况几个小小的 Warning 呢?

32、书写 Debug 版和 Release 版的程序

程序在开发过程中必然有许多程序员加的调试信息。我见过许多项目组, 当程序开发结束时, 发动群众删除程序中的调试信息, 何必呢? 为什么不像 VC++ 那样建立两个版本的目标代码? 一个是 debug 版本的, 一个是 Release 版的。那些调试信息是那么的宝贵, 在日后的维护过程中也是很宝贵的东西, 怎么能说删除就删除呢?

利用预编译技术吧, 如下所示声明调试函数:

```
#ifdef DEBUG
    void TRACE(char* fmt, ...)
    {
        .....
    }
#else
    #define TRACE(char* fmt, ...)
#endif
```

于是, 让所有的程序都用 TRACE 输出调试信息, 只需要在在编译时加上一个参数 “-DDEBUG”, 如:

```
cc -DDEBUG -o target target.c
```

于是, 预编译器发现 DEBUG 变量被定义了, 就会使用 TRACE 函数。而如果要发布给用户了, 那么只需要把取消 “-DDEBUG” 的参数, 于是所有用到 TRACE 宏, 这个宏什么都没有, 所以源程序中的所有 TRACE 语言全部被替换成了空。一举两得, 一箭双雕, 何乐而不为呢?

顺便提一下, 两个很有用的系统宏, 一个是 “__FILE__”, 一个是 “__LINE__”, 分别表示, 所在的源文件和行号, 当你调试信息或是输出错误时, 可以使用这两个宏, 让你一眼就能看出你的错误, 出现在哪个文件的第几行中。这对于用 C/C++ 做的大工程非常的管用。

总结

综上所述 32 条, 都是为了三大目的——

- 1、程序代码的易读性。
- 2、程序代码的可维护性,

3、程序代码的稳定可靠性。

有修养的程序员，就应该要学会写出这样的代码！这是任何一个想做编程高手所必需面对的细小的问题，编程高手不仅技术要强，基础要好，而且最重要的是要有“修养”！

好的软件产品绝不仅仅是技术，而更多的是整个软件的易维护和可靠性。

软件的维护有大量的工作量花在代码的维护上，软件的 Upgrade，也有大量的工作花在代码的组织上，所以好的代码，清晰的，易读的代码，将给大大减少软件的维护和升级成本。