分享一下自己总结的华为公司的编程风格

原创

花huajh 于 2018-01-09 23:38:57 发布 5076 收藏 3
分类专栏： 经验

随着计算机技术的发展，软件的规模增大了，软件的复杂性也增强了。为了提高程序的可阅读性，要建立良好的编程风格。
程序设计风格指一个人编制程序时所表现出来的特点，习惯逻辑思路等。 在程序设计中要使程序结构合理、清晰，形成良好的编程习惯，对程序的要求不仅是可以在机器上执行，给出正确的结果，而且要便于程序的调试和维护，这就要求编写的程序不仅自己看得懂，而且也要让别人能看懂。

ㄏx协入天舌稔底评讧飓桂

1. 掘隳

1.1 稔底坝覇重觚缯迟飓桂缜匾，缩进的空格数为4丰ザ

诺昔: 对于由开发工具自动生成的代码可以有不一致。

1.2 盾寿狒竑皊稔底坝釆闺ヤ叡釕诺昔釆吪忙颁劲窀衙ザ；

1.3 徥甜ヤ删斮筏诓叫弗苫胐辉閇皊衰迄引蜮诓叫，则要进行适应的划分，长表达式要在低优先级操作符处划分新行，操作符放在新行之首。

1.4 苫刃敶蜮迍稔弗皊又敶辉閇，则要进行适当的划分。

1.5 专兇设抎夠丰石诓叫匽圯ㄏ衙弗，即一行只写一条语句。

1.6 ifヤforヤdoヤwhileヤcaseヤswitchヤdefault筏诓叫皋卦ㄏ衙，且ifヤforヤdoヤwhile筏诓叫皊扭衙诓叫鄘刘旦诀夠赤鄗覇劲拳叽{}ザ

1.7 寿齖台侅觚窀桂锴，不使用TAB锴ザ

1.8 刃敶蜮迍稔皊弄姑ヤ给柲皊宠乏反徥甜ヤ删斮筏诓叫弗皊仩砍鄗覇重觚缯迟飓桂，case 诓叫丑皊惋冻夂琇诓叫乏覇遝仔诓叫缯迟覇汃ザ

1.9 稔底坝皊刘畎第(如C/C++ 诓訓皊夭拳叽‘{’咒‘}’)应各独占一行并且位于同一列，同时与引用它们的语句左对齐。在函数体的开始、类的定义、结构的定义、枚举的定义以及if、for、do、while、switch、case诓叫弗皊稔底鄗覇重觚妈丐皊缯迟旂引ザ

1.10 圯个丰仫丐皊兹锴孝ヤ叡釕ヤ帾釕迡衙寿筏攓佢眭，它们之间的操作符之前、之后或者前后要加空格；进行非对等操作时，如果是关系密切的立即操作符(如->)，后不应加空格。

诺昔: 采用这种松散方式编写代码的目的是使代码更加清晰。

男五娶寇桂戾亭甥皊渋晶惚昵盾寿皊, 所以, 在已经非常清晰的语句中没有必要再留空格, 如果语句已足够清晰则括号内侧(卹娈拳叽吨靨咒叹拳叽匆靨)专霆覇刻寇桂, 多重括号间不必加空格, 因为在C/C++诓訃弗拳叽跁绕昵酋渋晶皊性忝二ザ

圮閂诓叫弗, 如果需要加的空格非常多, 那么应该保持整体清晰, 而在局部不加空格。给操作符留空格时不要连续留两个以上空格。

2. 泮釐

2.1 厂舲惋冻丑, 源程序有效注释量必须在20 # 仫丐ザ

诺昔: 注释的原则是有助于对程序的阅读理解, 在该加的地方都加了, 注释不宜太多也不能太少, 注释语言必须准确、易懂、简洁。

2.2 诺昔惚竟任(如头文件.h 竟任サ.inc 竟任サ.def 竟任サ缜诗诺昔竟任.cfg 筏)头部应进行注释, 注释必须列出: 版权说明、版本号、生成日期、作者、内容、功能、与其它文件的关系、修改日志等, 头文件的注释中还应有函数功能简要说明。

2.3 準竟任夺鄙庚迡衔泮釐, 列出: 版权说明、版本号、生成日期、作者、模块目的/讣胞サ乁覇刃敶友兼讣胞サ俊政即忝筏ザ

2.4 刃敶夺鄙庚迡衔泮釐, 列出: 函数的目的/ 讣胞サ辙八又敶サ辙刀又敶サ迎囤偷サ谆觉兹叙(函数、表)等

祀侑: 下面这段函数的注释比较标准, 当然, 并不局限于此格式, 但上述信息建议要包含在内。

```
Function:       // 刃敶吓案
Description:    // 刃敶讣胞サ惚胞筏皊揕迶
Calls:          // 袱杕刃敶谆觉皊刃敶渋嚞
Called By:      // 谆觉杕刃敶皊刃敶渋嚞
Table Accessed: // 袱诅间皊袞(此项仅对于牵扯到数据库操作的程序)
Table Updated:  // 袱俊政皊袞(此项仅对于牵扯到数据库操作的程序)
Input:          // 辙八又敶诺昔, 包括每个参数的作
                // 觉サ娈偷诺昔妀又敶闺兹叙ザ
Output:         // 寿辙刀又敶皊诺昔ザ
Return:         // 刃敶迎囤偷皊诺昔
```

2.5 迪匼仩砍迪泮釐, 修改代码同时修改相应的注释, 以保证注释与代码的一致性。不再有用的注释要删除。

2.6 泮釐皊同宿覇渋楠サ昔二, 含义准确, 防止注释二义性。

2.7 遭巯圮泮釐弗侅觉缯匼, 特别是非常用缩写

2.8 泮釐庚且兼揕迶皊仳砍盾迠, 对代码的注释应放在其上方或右方(对单条语句的注释)相邻位置, 不可放在下面, 如放于上方则需与其上面的代码用空行隔开。

2.9 寿五宸胐胐牯琋吱乏皊叡釕サ帾釕, 如果其命名不是充分自注释的, 在声明时都必须加以注释, 说明其物理含义。变量、常量、宏的注释应放在其上方相邻位置或右方。

2.10 敶绊サ给柲サ籽サ梓メ筏) , 如果其命名不是充分自注释的, 必须加以注释。对数据结构的注释应放在其上方相邻位置, 不可放在下面; 对结构中的每个域的注释放在此域的右方。

2.11 兮屆叡釕覇胐辉诬绌皊泮釐, 包括对其功能、取值范围、哪些函数或过程存取它以及存取时注意事项等的说明。

2.12 泮釐且宸揕迶问宿迡衙吒栽皊缯掘ザ

2.13 酨泮釐且兼丐靨皊仳砍觡窀衙隍弄ザ

2.14 刃敶皊夺鄘庚迡衙泮釐= 初刀刃敶皊劝胞サ直皊サ辙八辙刀又敶サ迚囤偷サ谆觡兹叙 + 衰サ刃敶 - 筏

2.15 寿叡釕皊宠乏咒刘攵诓叫(条件分支、循环语句等)必须编写注释。

2.16 寿五switch诓叫丑皊case诓叫, 如果因为特殊情况需要处理完一个case吪迡八丑厂丰case攵琋, 必须在该case诓叫攵琋宅サ丑厂丰case诓叫匆劲丐昔砸皊泮釐ザ

3. 栘诌第啪吓

3.1 栘诌第皊啪吓覇渋晶サ昔二, 有明确含义, 同时使用完整的单词或大家基本可以理解的缩写, 避免使人产生误解。

诺昔: 较短的单词可通过去掉“光韹”彤或缯罝; 较长的单词可取单词的头几个字母形成缩写; 一些单词有大家公认的缩写。

祀侑: 如下单词的缩写能够被大家基本认可。

temp叵缯罝乀 **tmp;** 为皎

flag叵缯罝乀 **flg;** 栘忝

statistic叵缯罝乀 **stat ;** 绥讧

increment叵缯罝乀 **inc;** 墳釕

message叵缯罝乀 **msg;** 涎惠

3.2 啪吓弗苫侅觡牾殊纬宠或缯罝, 则要有注释说明。

诺昔: 应该在源文件的开始之处, 对文件中所使用的缩写或约定, 特别是特殊的缩写, 进行必要的注释说明。

3.3 鼻巷牾胐皊啪吓飓桂, 要自始至终保持一致, 不可来回变化。

诺昔: 个人的命名风格, 在符合所在项目组或产品组的命名规则的前提下, 才可使用。(即命名规则中没有规定到的地方才可有个人命名风格)。

3.4 寿五叡釕咆吓, 禁止取单个字符(如i、j、k...), 建议除了要有具体含义外, 还能表明其变量类型、数据类型等, 但i、j、k 佢屇鄘徭甜叡釕昵兇设皊ザ

诺昔: 变量, 尤其是局部变量, 如果用单个字符表示, 很容易敲错(如i叵或j), 而编译时又检查不出来, 有可能为了这个小小的错误而花费大量的查错时间。

祀侑: 下面所示的局部变量名的定义方法可以借鉴。

int liv_Width

兼叡釕吓觫釐妈丑:

l 屇鄘叡釕(Local) (其它: g 兮屇叡釕(Global)...)

i 敶捻籽垮(Interger)

v 叡釕(Variable) (其它: c 帾釕(Const)...)

Width 叡釕吱乏

迟裁叵仫陡走屇鄘叡釕且兮屇叡釕瓠吓ザ

3.5 咆吓觇茉忙颁且宸侅觭皊绥飓桂俣指厂臺, 并在同一项目中统一, 比如采用UNIX皊兮尕罝劦丑刘绅皊飓桂蝝夭尕罝滥掘皊旂引, 不要使用大小写与下划线混排的方式, 用作特殊栕诌妈栕诌或呵叡釕蝝兮屇叡釕皊m_ 咒g_ , 兼吪劦丐夭尕罝滥掘皊旂引昵兇设皊

4. 叵谁怵

4.1 泮愕达箝第皊伞兎纭= 幼觭拳叽昔砵袞迄引皊摛佢颀底= 遭殈侅觭點讪伞兎纭ザ

4.2 遭殈侅觭专昙琇觫皊敶孝, 用有意义的标识来替代。涉及物理状态或者含有物理意义的常量, 不应直接使用数字, 必须用有意义的枚举或宏来代替。

5. 叡釕サ给枰

5.1 叁掏沧忙覇皊天具叡釕ザ

诺昔: 公共变量是增大模块间耦合的原因之一, 故应减少没必要的公共变量以降低模块间的耦合度。

5.2 <u>介绌宠乏幼昔砵天具叡釕皊吱乏サ佢觭サ娈偷茉固发天具叡釕闺皊兹叙ザ</u>

诺昔: 在对变量声明的同时, 应对其含义、作用及取值范围进行注释说明, 同时若有必要还应说明与其它变量的关系。

5.3 昔砵天具叡釕且摛佢歪天具叡釕皊刃敶氨迍毯皊兹叙, 如访问、修改及创建等。

诺昔: 明确过程操作变量的关系后, 将有利于程序的进一步优化、单元测试、系统联调以及代码维护等。这种关系的说明可在注释或文档中描述。

祀侑: 在源文件中, 可按如下注释形式说明。

RELATION　System_Init　Input_Rec　Print_Rec　Stat_Score

Student　Create　Modify　Access　Access

Score　Create　Modify　Access　Access, Modify

泮: RELATION乀摛佢兹叙; System_InitサInput_RecサPrint_RecサStat_Score乀囡丰专吒皊刃敶;
StudentサScore乀个丰兮屉叡釕; Create袞祀剋开, Modify袞祀倰政, Access袞祀诅间ザ

兼弗, 函数Input_RecサStat_Score鄗叵倰政叡釕Score, 故此变量将引起函数间较大的耦合, 并可能增加代码测试、维护的难度。

5.4 彙吗天具叡釕伦逶敶捻皊, 要十分小心, 防止赋与不合理的值或越界等现象发生。

诺昔: 对公共变量赋值时, 若有必要应进行合法性检查, 以提高代码的可靠性、稳定性。

5.5 陂走屉鄘叡釕且天具叡釕吒吓ザ

诺昔: 若使用了较好的命名规则, 那么此问题可自动消除。

5.6 丫褚侅觭杏绕刨姑匝皊叡釕佢乀叹偷ザ

诺昔: 特别是在C/C++弗弛觭杏绕蜮偷皊按钎, 经常会引起系统崩溃。

6. 刃敶サ迍稔

¹6-1 : 对所调用函数的错误返回码要仔细、全面地处理

¹6-2 : 明确函数功能, 精确(而不是近似)地实现函数设计

¹6-3 : 编写可重入函数时, 应注意局部变量的使用(如编写C/C++ 诓訓皊叵乩八刃敶皊, 应使用auto 卹铫眇怛屉鄘叡釕或寅忐噮叡釕)

诺昔: 编写C/C++诓訓皊叵乩八刃敶皊, 不应使用static屉鄘叡釕, 否则必须经过特殊处理, 才能使函数具有可重入性。

¹6-4 : 编写可重入函数时, 若使用全局变量, 则应通过关中断、信号量(即P、Ⅴ摛佢)等手段对其加以保护

诺昔: 若对所使用的全局变量不加以保护, 则此函数就不具有可重入性, 即当多个进程调用此函数时, 很有可能使有关全局变量变为不可知状态。

祀侑: 假设Exam呢int垮兮屉叡釕, 函数Squre_Exam迍囤Exam幹旂偷ザ烬乎妈丑刃敶专兽胐叵乩八怵ザ

```
unsigned int example( int para )
{
   unsigned int temp;
   Exam = para; // (**)
   temp = Square_Exam( );
   return temp;
}
```

歪刃敶苦袱匈丰迟毡谆觳皑诣, 其结果可能是未知的, 因为当(**)语句刚执行完后, 另外一个使用本函数的进程可能正好被激活, 那么当新激活的进程执行到此函数时, 将使Exam燊且召丆丰专吒皑para偭, 所以当控制重新回到“temp = Square_Exam( )”吔, 计算出的temp徨叵胞专昵颊惹弗皑给枢ザ歪刃敶庚妈丑政迟ザ

```
unsigned int example( int para )
{
   unsigned int temp;
   [粤诽徒叽釕摭佢]       // 苫粤诽专制“徒叽釕”, 说明另外的进程正处于
   Exam = para;           // 统Exam燊偭幼讧箝兼幹旂迍毡弗(即正在使用此
   temp = Square_Exam( );  // 徒叽), 本进程必须等待其释放信号后, 才可继
   [釐敕徒叽釕摭佢]       // 绳扭衙ザ苫粤诽制徒叽, 则可继续执行, 但其
                     // 安迟毡忙颁筏律杌迟毡釐敕徒叽釕吔, 才能再使
                     // 觳杌徒叽ザ
   return temp;
}
```

¹6-5 : 在同一项目组应明确规定对接口函数参数的合法性检查应由函数的调用者负责还是由接口函数本身负责, 缺省是由函数调用者负责

诺昔: 对于模块间接口函数的参数的合法性检查这一问题, 往往有两个极端现象, 即: 要么是调用者和被调用者对参数均不作合法性检查, 结果就遗漏了合法性检查这一必要的处理过程, 造成问题隐患; 要么就是调用者和被调用者均对参数进行合法性检查, 这种情况虽不会造成问题, 但产生了冗余代码, 降低了效率。

½6-1 : 防止将函数的参数作为工作变量

诺昔: 将函数的参数作为工作变量, 有可能错误地改变参数内容, 所以很危险。对必须改变的参数, 最好先用局部变量代之, 最后再将该局部变量的内容赋给该参数。

祀侑: 下函数的实现不太好。

```
void sum_data( unsigned int num, int *data, int *sum )
{
   unsigned int count;
   *sum = 0;

   for (count = 0; count < num; count++)
   {
      *sum  += data[count]; // sum或二巫佢叟釕, 不太好。
   }
```

```
}
```

苦政乀妈丑, 则更好些。

```
void sum_data( unsigned int num, int *data, int *sum )
{
    unsigned int count ;
    int sum_temp;
    sum_temp = 0;

    for (count = 0; count < num; count ++)
    {
        sum_temp  += data[count];
    }

    *sum = sum_temp;
}
```

½6-2 : 函数的规模尽量限制在200 衙仏问

诺昔: 不包括注释和空格行。

½6-3 : 一个函数仅完成一件功能

½6-4 : 为简单功能编写函数

诺昔: 虽然为仅用一两行就可完成的功能去编函数好象没有必要, 但用函数可使功能明确化, 增加程序可读性, 亦可方便维护、测试。

祀侑: 如下语句的功能不很明显。

```
value = ( a > b ) ? a : b ;
```

政乀妈丑尷徨渋晶二ザ

```
int max (int a, int b)
{
    return ((a > b) ? a : b);
}
value = max (a, b);
```

戛政乀妈丑ザ

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

value = MAX (a, b);

½6-5 : 不要设计多用途面面俱到的函数

诺昔: 多功能集于一身的函数, 很可能使函数的理解、测试、维护等变得困难。

½6-6 : 函数的功能应该是可以预测的, 也就是只要输入数据相同就应产生同样的输出

诺昔: 带有内部“忐傮嚘”皊刃敶皊邡胳叵胳频浑皊, 因为它的输出可能取决于内部存储器(如某标记)的状态。这样的函数既不易于理解又不利于测试和维护。在C/C++诓訒弗, 函数的static屆鄘叡釕昵刃敶皊问鄘忐傮嚘, 有可能使函数的功能不可预测, 然而, 当某函数的返回值为指针类型时, 则必须是STATIC皊屆鄘叡釕皊垄坟佢乀迍囤偷, 若为AUTO粀, 则返回为错针。

祀侑: 如下函数, 其返回值(即功能)是不可预测的。

```
unsigned int integer_sum( unsigned int base )
{
   unsigned int index;
   static unsigned int sum = 0; // 泮愕, 是static粀垮皊ザ
                                // 苦政乀auto粀垮, 则函数即变为可预测。
   for (index = 1; index <= base; index++)
   {
      sum += index;
   }
   return sum;
}
```

½6-7 : 尽量不要编写依赖于其他函数内部实现的函数

诺昔: 此条为函数独立性的基本要求。由于目前大部分高级语言都是结构化的, 所以通过具体语言的语法要求与编译器功能, 基本就可以防止这种情况发生。但在汇编语言中, 由于其灵活性, 很可能使函数出现这种情况。

½6-8 : 避免设计多参数函数, 不使用的参数从接口中去掉

诺昔: 目的减少函数间接口的复杂度。

½6-9 : 非调度函数应减少或防止控制参数, 尽量只使用数据参数

诺昔: 本建议目的是防止函数间的控制耦合。调度函数是指根据输入的消息类型或控制命令, 来启动相应的功能实体(即函数或过程), 而本身并不完成具体功能。控制参数是指改变函数功能行为的参数, 即函数要根据此参数来决定具体怎样工作。非调度函数的控制参数增加了函数间的控制耦合, 很可能使函数间的耦合度增大, 并使函数的功能不唯一。

½6-10 : 检查函数所有参数输入的有效性

½6-11 : 检查函数所有非参数输入的有效性, 如数据文件、公共变量等

诺昔: 函数的输入主要有两种: 一种是参数输入; 另一种是全局变量、数据文件的输入, 即非参数输入。函数在使用输入之前, 应进行必要的检查。

½6-12 : 函数名应准确描述函数的功能

½6-13 : 使用动宾词组为执行某操作的函数命名。如果是OOP 旂泛, 可以只有动词(名词是对象本身)

祀侑: 参照如下方式命名函数。

```
void print_record( unsigned int rec_ind ) ;

int  input_record( void ) ;

unsigned char get_current_color( void ) ;
```

开讵6-14 : 避免使用无意义或含义不清的动词为函数命名

诺昔: 避免用含义不清的动词如process艹handle筏乀刃敶咆吓, 因为这些动词并没有说明要具体做什么。

开讵6-15 : 函数的返回值要清楚、明了, 让使用者不容易忽视错误情况

诺昔: 函数的每种出错返回值的意义要清晰、明了、准确, 防止使用者误用、理解错误或忽视错误返回码。

½6-16 : 除非必要, 最好不要把与函数返回值类型不同的变量, 以编译系统默认的转换方式或强制的转换方式作为返回值返回

½6-17 : 让函数在调用点显得易懂、容易理解

½6-18 : 在调用函数填写参数时, 应尽量减少没有必要的默认数据类型转换或强制数据类型转换

诺昔: 因为数据类型转换或多或少存在危险。

½6-19 : 避免函数中不必要语句, 防止程序中的垃圾代码

诺昔: 程序中的垃圾代码不仅占用额外的空间, 而且还常常影响程序的功能与性能, 很可能给程序的测试、维护等造成不必要的麻烦。

½6-20 : 防止把没有关联的语句放到一个函数中

诺昔: 防止函数或过程内出现随机内聚。随机内聚是指将没有关联或关联很弱的语句放到同一个函数或过程中。随机内聚给函数或过程的维护、测试及以后的升级等造成了不便, 同时也使函数或过程的功能不明确。使用随机内聚函数, 常常容易出现在一种应用场合需要改进此函数, 而另一种应用场合又不允许这种改进, 从而陷入困境。

圮缜秸眭, 经常遇到在不同函数中使用相同的代码, 许多开发人员都愿把这些代码提出来, 并构成一个新函数。若这些代码关联较大并且是完成一个功能的, 那么这种构造是合理的, 否则这种构造将产生随机内聚的函数。

祀侑: 如下函数就是一种随机内聚。

```
void Init_Var( void )

{

   Rect.length = 0;

   Rect.width = 0; /* 刨姑匝矯彤皊閂且寇 */

   Point.x = 10;
```

```
    Point.y = 10;   /* 刨姑匝“炴”皊垔栓 */
}
```

矯彬皊閂ヤ寇且炴皊垔栓堀杕沧胐企佛兹叙, 故以上函数是随机内聚。

庚妈丑刘乀个丰刃敶:

```
void lnit_Rect( void )
{
    Rect.length = 0;
    Rect.width = 0; /* 刨姑匝矯彬皊閂且寇 */
}

void lnit_Point( void )
{
    Point.x = 10;
    Point.y = 10;   /* 刨姑匝“炴”皊垔栓 */
}
```

½6-21 : 如果多段代码重复做同一件事情, 那么在函数的划分上可能存在问题

诺昔: 若此段代码各语句之间有实质性关联并且是完成同一件功能的, 那么可考虑把此段代码构造成一个新的函数。

½6-22 : 功能不明确较小的函数, 特别是仅有一个上级函数调用它时, 应考虑把它合并到上级函数中, 而不必单独存在

诺昔: 模块中函数划分的过多, 一般会使函数间的接口变得复杂。所以过小的函数, 特别是扇入很低的或功能不明确的函数, 不值得单独存在。

½6-23 : 设计高扇入、合理扇出(小于7)的函数

诺昔: 扇出是指一个函数直接调用(控制)其它函数的数目, 而扇入是指有多少上级函数调用它。

才刀迍夭, 表明函数过分复杂, 需要控制和协调过多的下级函数; 而扇出过小, 如总是1, 表明函数的调用层次可能过多, 这样不利程序阅读和函数结构的分析, 并且程序运行时会对系统资源如堆栈空间等造成压力。函数较合理的扇出(调度函数除外)通常是3-5ザ才刀齐夭, 一般是由于缺乏中间层次, 可适当增加中间层次的函数。扇出太小, 可把下级函数进一步分解多个函数, 或合并到上级函数中。当然分解或合并函数时, 不能改变要实现的功能, 也不能违背函数间的独立性。

才八趔夭, 表明使用此函数的上级函数越多, 这样的函数使用效率高, 但不能违背函数间的独立性而单纯地追求高扇入。公共模块中的函数及底层函数应该有较高的扇入。

辉羝妃皊钑任给枸造帾昵顼屈刃敶皊才刀辉龘, 中层函数的扇出较少, 而底层函数则扇入到公共模块中。

½6-24 : 减少函数本身或函数间的递归调用

诺昔: 递归调用特别是函数间的递归调用(如A->B->C->A), 影响程序的可理解性; 递归调用一般都占用较多的系统资源(如栈空间); 递归调用对程序的测试有一定影响。故除非为某些算法或功能的实现方便, 应减少没必要的递归调用。

½6-25 : 仔细分析模块的功能及性能需求, 并进一步细分, 同时若有必要画出有关数据流图, 据此来进行模块的函数划分与组织

诺昔: 函数的划分与组织是模块的实现过程中很关键的步骤, 如何划分出合理的函数结构, 关系到模块的最终效率和可维护性、可测性等。根据模块的功能图或/友敶捻浇圄昦尊刀刃敶给柲昵帾觅旂泛承ア ザ

½6-26 : 改进模块中函数的结构, 降低函数间的耦合度, 并提高函数的独立性以及代码可读性、效率和可维护性

伞匝刃敶给柲皊, 要遵守以下原则:

(1)不能影响模块功能的实现。

(2)仔细考查模块或函数出错处理及模块的性能要求并进行完善。

(3)通过分解或合并函数来改进软件结构。

(4)考查函数的规模, 过大的要进行分解。

(5)降低函数间接口的复杂度。

(6)不同层次的函数调用要有较合理的扇入、扇出。

(7)函数功能应可预测。

(8)提高函数内聚。(单一功能的函数内聚最高)

诺昔: 对初步划分后的函数结构应进行改进、优化, 使之更为合理。

½6-27 : 在多任务操作系统的环境下编程, 要注意函数可重入性的构造

诺昔: 可重入性是指函数可以被多个任务进程调用。在多任务操作系统中, 函数是否具有可重入性是非常重要的, 因为这是多个进程可以共用此函数的必要条件。另外, 编译器是否提供可重入函数库, 与它所服务的操作系统有关, 只有操作系统是多任务时, 编译器才有可能提供可重入函数库。如DOS丑BC咒MSC筏尷专兽复叵虱八刃敶庙, 因为DOS昵孿帾舜孿企劲攓佢叙绥ザ

½6-28 : 避免使用BOOL 又敶

诺昔: 原因有二, 其一是BOOL又敶偷旦愕乏, TURE/FALSE皊吱乏昵皑帾椒糐皊, 在调用时很难知道该参数到底传达的是什么意思; 其二是BOOL又敶偷专刞五扯克ザ连胐NULL乞昵ア丰旦愕乏皊孿诓ザ

½6-29 : 寿五揖侕二迍囤偷皊刃敶, 在引用时最好使用其返回值

½6-30 : 当一个过程(函数)中对较长变量(一般是结构的成员)有较多引用时, 可以用一个意义相当的宏代替

デ三ト =====[叵浑悒]=====

¹7-1 : 在同一项目组或产品组内, 要有一套统一的为集成测试与系统联调准备的调测开关及相应打印函数, 并且要有详细的说明

诺昔: 本规则是针对项目组或产品组的。

¹7-2 : 在同一项目组或产品组内, 调测打印出的信息串的格式要有统一的形式。信息串中至少要有所在模块名(或源文件名)及行号

诺昔: 统一的调测信息格式便于集成测试。

¹7-3 : 编程的同时要为单元测试选择恰当的测试点, 并仔细构造测试代码、测试用例, 同时给出明确的注释说明。测试代码部分应作为(模块中的)一个子模块, 以方便测试代码在模块中的安装与拆卸(通过调测开关)

诺昔: 为单元测试而准备。

¹7-4 : 在进行集成测试/ 紉绥聚谇豕剪, 要构造好测试环境、测试项目及测试用例, 同时仔细分析并优化测试用例, 以提高测试效率

诺昔: 好的测试用例应尽可能模拟出程序所遇到的边界值、各种复杂环境及一些极端情况等。

¹7-5 : 使用断言来发现软件问题, 提高代码可测性

诺昔: 断言是对某种假设条件进行检查(可理解为若条件成立则无动作, 否则应报告), 它可以快速发现并定位软件问题, 同时对系统错误进行自动报警。断言可以对在系统中隐藏很深, 用其它手段极难发现的问题进行定位, 从而缩短软件问题定位时间, 提高系统的可测性。实际应用时, 可根据具体情况灵活地设计断言。

¹7-6 : 用断言来检查程序正常运行时不应发生但在调测时有可能发生的非法情况

¹7-7 : 不能用断言来检查最终产品肯定会出现且必须处理的错误情况

诺昔: 断言是用来处理不应该发生的错误情况的, 对于可能会发生的且必须处理的情况要写防错程序, 而不是断言。如某模块收到其它模块或链路上的消息后, 要对消息的合理性进行检查, 此过程为正常的错误检查, 不能用断言来实现。

¹7-8 : 对较复杂的断言加上明确的注释

诺昔: 为复杂的断言加注释, 可澄清断言含义并减少不必要的误用。

¹7-9 : 用断言确认函数的参数

¹7-10 : 用断言保证没有定义的特性或功能不被使用

¹7-11 : 用断言对程序开发环境(OS/Compiler/Hardware)的假设进行检查

诺昔: 程序运行时所需的软硬件环境及配置要求, 不能用断言来检查, 而必须由一段专门代码处理。用断言仅可对程序开发环境中的假设及所配置的某版本软硬件是否具有某种功能的假设进行检查。如某网卡是否在系统运行环境中配置了, 应由程序中正式代码来检查; 而此网卡是否具有某设想的功能, 则可由断言来检查。

寿缜诗喂揖偭岭劢胞夂牿怭偍评叵箣斱訓棆梆, 原因是软件最终产品(即运行代码或机器码)与编译器已没有任何直接关系, 即软件运行过程中(注意不是编译过程中)不会也不应该对编译器的功能提出任何需求。

¹7-12 : 正式软件产品中应把断言及其它调测代码去掉(即把有关的调测开关关掉)

诺昔: 加快软件运行速度。

¹7-13 : 在软件系统中设置与取消有关测试手段, 不能对软件实现的功能等产生影响

诺昔: 即有测试代码的软件和关掉测试代码的软件, 在功能行为上应一致。

¹7-14 : 用调测开关来切换软件的DEBUG 牕咒步引牕, 而不要同时存在正式版本和DEBUG 牕朳岭专吒準竟任, 以减少维护的难度

¹7-15 : 软件的DEBUG 牕朳咒受衙牕朳庚诫绥丆绺捇, 不允许分家, 并且要时刻注意保证两个版本在实现功能上的一致性

½7-1 : 在编写代码之前, 应预先设计好程序调试与测试的方法和手段, 并设计好各种调测开关及相应测试代码如打印函数等

诺昔: 程序的调试与测试是软件生存周期中很重要的一个阶段, 如何对软件进行较全面、高率的测试并尽可能地找出软件中的错误就成为很关键的问题。因此在编写源代码之前, 除了要有一套比较完善的测试计划外, 还应设计出一系列代码测试手段, 为单元测试、集成测试及系统联调提供方便。

½7-2 : 调测开关应分为不同级别和类型

诺昔: 调测开关的设置及分类应从以下几方面考虑: 针对模块或系统某部分代码的调测; 针对模块或系统某功能的调测; 出于某种其它目的, 如对性能、容量等的测试。这样做便于软件功能的调测, 并且便于模块的单元测试、系统联调等。

½7-3 : 编写防错程序, 然后在处理错误之后可用断言宣布发生错误

デ共ト =====[稓底教生]=====

¹8-1 : 编程时要经常注意代码的效率

8-2：在保证软件系统的正确性、可读性、稳定性及可测试性的前提下，提高代码效率

½8-3 : 对模块中函数的划分及组织方式进行分析、优化, 改进模块中函数的组织结构, 提高程序效率

½8-4 : 编程时, 要随时留心代码效率; 优化代码时, 要考虑周全

½8-5 : 不应花过多的时间拼命地提高调用不很频繁的函数代码效率

½8-6 : 要仔细地构造或直接用汇编编写调用频繁或性能要求极高的函数

½8-7 : 在保证程序质量的前提下, 通过压缩代码量、去掉不必要代码以及减少不必要的局部和全局变量, 来提高空间效率

½8-8 : 在多重循环中, 应将最忙的循环放在最内层

½8-9 : 尽量减少循环嵌套层次

½8-10 : 避免循环体内含判断语句, 应将循环语句置于判断语句的代码块之中

8-11：尽量用乘法或其他的方法代替除法，特别是浮点运算中的除法

½8-12 : 不要一味追求紧凑的代码

9 贮釘俣诃

¹9-1 : 在软件设计过程中构筑软件质量

¹9-2 : 代码质量保证优先原则

(1)正确性, 指程序要实现设计要求的功能。

(2)稳定性、安全性, 指程序稳定、可靠、安全。

(3)可测试性, 指程序要具有良好的可测试性。

(4)规范/叵谁怵, 指程序书写风格、命名规则等要符合规范。

(5)全局效率, 指软件系统的整体效率。

(6)局部效率, 指某个模块/孜榱坝/刃敶皊杁躲教生ザ

(7)个人表达方式/丰什游俅怵, 指个人编程习惯。

¹9-3 : 只引用属于自己的存贮空间

诺昔: 若模块封装的较好, 那么一般不会发生非法引用他人的空间。

¹9-4 : 防止引用已经释放的内存空间

诺昔: 在实际编程过程中, 稍不留心就会出现在一个模块中释放了某个内存块(如C诓訓挍钎), 而另一模块在随后的某个时刻又使用了它。要防止这种情况发生。

¹9-5 : 过程/ 刃敶弗刘畬皊问忐, 在过程/ 刃敶逆刀釆歹覇釐孜

¹9-6 : 过程/ 刃敶弗粵诽皊(为打开文件而使用的)文件句柄, 在过程/ 刃敶逆刀釆歹覇兹闼

诺昔: 分配的内存不释放以及文件句柄不关闭, 是较常见的错误, 而且稍不注意就有可能发生。这类错误往往会引起很严重后果, 且难以定位。

¹9-7 : 防止内存操作越界

诺昔: 内存操作主要是指对数组、指针、内存地址等的操作。内存操作越界是软件系统主要错误之一, 后果往往非常严重, 所以当我们进行这些操作时一定要仔细小心。

¹9-8 : 认真处理程序所能遇到的各种出错情况

¹9-9 : 系统运行之初, 要初始化有关变量及运行环境, 防止未经初始化的变量被引用

¹9-10 : 系统运行之初, 要对加载到系统中的数据进行一致性检查

诺昔: 使用不一致的数据, 容易使系统进入混乱状态和不可知状态。

¹9-11 : 严禁随意更改其它模块或系统的有关设置和配置

诺昔: 编程时, 不能随心所欲地更改不属于自己模块的有关设置如常量、数组的大小等。

¹9-12 : 不能随意改变与其它模块的接口

¹9-13 : 充分了解系统的接口之后, 再使用系统提供的功能

¹9-14 : 编程时, 要防止差1 锟诵

诺昔: 此类错误一般是由于把“<=”诵訇或“<”或“>=”诵訇或“>”筏邇或皊, 由此引起的后果, 很多情况下是很严重的, 所以编程时, 一定要在这些地方小心。当编完程序后, 应对这些操作符进行彻底检查。

¹9-15 : 要时刻注意易混淆的操作符。当编完程序后, 应从头至尾检查一遍这些操作符, 以防止拼写错误

¹9-16 : 有可能的话, if 诓叫员釕劦丐else 刘攵, 对没有else 刘攵皊诓叫覇尕忉寿律; switch 诓叫忙颁胐default 刘攵

¹9-17 : Unix 丑, 多线程的中的子线程退出必需采用主动退出方式, 即子线程应return 刀叩ザ

¹9-18 : 不要滥用goto 诓叫ザ

诺昔: goto诓叫传砺丕毡底皊给柳悒, 所以除非确实需要, 最好不使用goto诓叫ザ

½9-1 : 不使用与硬件或操作系统关系很大的语句, 而使用建议的标准语句, 以提高软件的可移植性和可重用性

½9-2 : 除非为了满足特殊需求, 避免使用嵌入式汇编

诺昔: 程序中嵌入式汇编, 一般都对可移植性有较大的影响。

½9-3 : 精心地构造、划分子模块, 并按“ 撖叩” 鄘刱爻“ 同栾” 鄘刱后琎垄绊绍孜棳坝, 以提高“ 同栾” 鄘刱龄叵豪椓怵咒叵虮觚怵

½9-4 : 精心构造算法, 并对其性能、效率进行测试

½9-5 : 对较关键的算法最好使用其它算法来确认

½9-6 : 时刻注意表达式是否会上溢、下溢

½9-7 : 使用变量时要注意其边界值的情况

½9-8 : 留心程序机器码大小(如指令空间大小、数据空间大小、堆栈空间大小等)是否超出系统有关限制

½9-9 : 为用户提供良好的接口界面, 使用户能较充分地了解系统内部运行状态及有关系统出错情况

½9-10 : 系统应具有一定的容错能力, 对一些错误事件(如用户误操作等)能进行自动补救

½9-11 : 对一些具有危险性的操作代码(如写硬盘、删数据等)要仔细考虑, 防止对数据、硬件等的安全构成危害, 以提高系统的安全性

½9-12 : 使用第三方提供的软件开发工具包或控件时, 要注意以下几点:

(1)充分了解应用接口、使用环境及使用时注意事项。

(2)不能过分相信其正确性。

(3)除非必要, 不要使用不熟悉的第三方工具包与控件。

½9-13 : 资源文件(多语言版本支持), 如果资源是对语言敏感的, 应让该资源与源代码文件脱离, 具体方法有下面几种: 使用单独的资源文件、DLL 斋任或兼安攀狒皑揕迶斋任(如数据库格式)

デ升ト =====[仩砍缜辗サ缜诗サ宦柫]=====

¹10-1 : 打开编译器的所有告警开关对程序进行编译

¹10-2 : 在产品软件(项目组)中, 要统一编译开关选项

¹10-3 : 通过代码走读及审查方式对代码进行检查

诺昔: 代码走读主要是对程序的编程风格如注释、命名等以及编程时易出错的内容进行检查, 可由开发人员自己或开发人员交叉的方式进行; 代码审查主要是对程序实现的功能及程序的稳定性、安全性、可靠性等进行检查及评审, 可通过自审、交叉审核或指定部门抽查等方式进行。

¹10-4 : 测试部测试产品之前, 应对代码进行抽查及评审

½10-1 : 编写代码时要注意随时保存, 并定期备份, 防止由于断电、硬盘损坏等原因造成代码丢失

½10-2 : 同产品软件(项目组)内, 最好使用相同的编辑器, 并使用相同的设置选项

诺昔: 同一项目组最好采用相同的智能语言编辑器, 如Muiti Editor, Visual Editor筏, 并设计、使用一套缩进宏及注释宏等, 将缩进等问题交由编辑器处理。

½10-3 : 要小心地使用编辑器提供的块拷贝功能编程

诺昔: 当某段代码与另一段代码的处理功能相似时, 许多开发人员都用编辑器提供的块拷贝功能来完成这段代码的编写。由于程序功能相近, 故所使用的变量、采用的表达式等在功能及命名上可能都很相近, 所以使用块拷贝时要注意, 除了修改相应的程序外, 一定要把使用的每个变量仔细查看一遍, 以改成正确的。不应指望编译器能查出所有这种错误, 比如当使用的是全局变量时, 就有可能使某种错误隐藏下来。

½10-4 : 合理地设计软件系统目录, 方便开发人员使用

诺昔: 方便、合理的软件系统目录, 可提高工作效率。目录构造的原则是方便有关源程序的存储、查询、编译、链接等工作, 同时目录中还应具有工作目录----庖胍皊缜诗サ锄掫筏巫佢庚圮歪直彝弗迟衙, 工具目录----胍兹竟任缜辗喂サ竟任柫抄筏巫兽叵忐孜圮歪直彝弗ザ

½10-5 : 某些语句经编译后产生告警, 但如果你认为它是正确的, 那么应通过某种手段去掉告警信息

诺昔: 在Borland C/C++弗, 可用“#pragma warn”杜兹掏蛩扙弃绍亡呐譬ザ

祀侑:

#pragma warn -rvl // 兹闵呐譬

int examples_fun(void)

{

    // 稓底, 但无return诓叫ザ

}

#pragma warn +rvl // 扙弃呐譬

缜诗刃敶examples_fun眩杕庚亭甥“刃敶庚胍迪囤偷”呐譬, 但由于关掉了此告警信息显示, 所以编译时将不会产生此告警提示。

½10-6 : 使用代码检查工具(如C 诓訕舀PC-Lint)对源程序检查

½10-7 : 使用软件工具(如 LogiSCOPE)进行代码审查

デ升ア卜 =====[仩砍浑诛サ绡抪]=====

¹11-1 : 单元测试要求至少达到语句覆盖

¹11-2 : 单元测试开始要跟踪每一条语句, 并观察数据流及变量的变化

¹11-3 : 清理、整理或优化后的代码要经过审查及测试

¹11-4 : 代码版本升级要经过严格测试

¹11-5 : 使用工具软件对代码版本进行维护

¹11-6 : 正式版本上软件的任何修改都应有详细的文档记录

½11-1 : 发现错误立即修改, 并且要记录下来

½11-2 : 关键的代码在汇编级跟踪

½11-3 : 仔细设计并分析测试用例, 使测试用例覆盖尽可能多的情况, 以提高测试用例的效率

½11-4 : 尽可能模拟出程序的各种出错情况, 对出错处理代码进行充分的测试

½11-5 : 仔细测试代码处理数据、变量的边界情况

½11-6 : 保留测试信息, 以便分析、总结经验及进行更充分的测试

½11-7 : 不应通过“ 诛” 杜觫泼间颞, 应寻找问题的根本原因

½11-8 : 对自动消失的错误进行分析, 搞清楚错误是如何消失的

½11-9 : 修改错误不仅要治表, 更要治本

½11-10 : 测试时应设法使很少发生的事件经常发生

½11-11 : 明确模块或函数处理哪些事件, 并使它们经常发生

½11-12 : 坠指圮缜砹阼殷尴寿仩砹迟衙征庇皑嚣光浑诛, 不要等以后的测试工作来发现

½11-13: 去除代码运行的随机性(如去掉无用的数据、代码及尽可能防止并注意函数中的“同鄘寅忐嘏”筏), 让函数运行的结果可预测, 并使出现的错误可再现

ァ升互卜 =====[宕]=====

¹12-1 : 用宏定义表达式时, 要使用完备的括号

¹12-2: 刽宕展宠乏皑匈杠衰迄引孜圮夭拳叽弗

¹12-3: 侬觚宕眩, 不允许参数发生变化

互ｘ腠讵夭孑 C++编码规范

1目的

讠彨或夭孑绥厂皑C++编码风格，以保障公司项目代码的易维护性和编码安全性，特制定本规范“腾讯集团”是指腾讯控股有限公司、其附属公司、及为会计而综合入账的公司，包括但不限于腾讯控股有限公司、深圳市腾讯计算机系统有限公司、腾讯科技（深圳）有限公司、腾讯科技（北京）有限公司、深圳市世纪凯旋科技有限公司、时代朝阳科技（深圳）有限公司、腾讯数码（深圳）有限公司、深圳市财付通科技有限公司。

3总体原则

展胍侬觚C和C++作为开发语言的软件产品都须遵照本规范的内容进行编码。4程序的版式

4.1规则：程序块要采用缩进风格编写，缩进的空格数为4个。说明：

男弄受巫兽枭勃甥或皑仩砹叵脆专厂臺= 侃妈枢弄受巫兽叵仫畬累= 刮庚诫绥厂畬累缯迟讠4个空格

4.2规则：缩进或者对齐只能使用空格键，不可使用TAB键。

侬觚TAB键需要设置

4.3规则：相对独立的程序块之间、变量说明之后必须加空行。说明：

仫丑惋冻庚诫昵觚窀衙刂弃ｘ

1）函数之间应该用空行分开；

2）变量声明应尽可能靠近第一次使用处，避免一次性声明一组没有马上使用的变量；

3）用空行将代码按照逻辑片断划分；

4）每个类声明之后应该加入空格同其他代码分开。

4.3规则：相对独立的程序块之间、变量说明之后必须加空行。说明：

仫丑惋冻庚诫昵觚窀衙刂弃ｘ

1）函数之间应该用空行分开；

2）变量声明应尽可能靠近第一次使用处，避免一次性声明一组没有马上使用的变量；

3）用空行将代码按照逻辑片断划分；

4）每个类声明之后应该加入空格同其他代码分开示例：

4.4规则：较长的语句（>80字符）要分成多行书写。说明：

仫丑惋冻庚刘匆衔昙匼 x

1）长表达式要在低优先级操作符处划分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，使排版整齐，语句可读。

2）若函数或过程中的参数较长，则要进行适当的划分。

3）循环、判断等语句中若有较长的表达式或语句，则要进行适应的划分，长表达式要在低优先级操作符处划分新行，操作符放在新行之首

4.5规则：不允许把多个短语句写在一行中

厂衔仩砍台偠厂任云惋= 妈台宠乏厂丰殿釕= 蛷台匼厂杠诳叫ザ迟栽皊仩砍宿昙阅谁= 幼业旂俅五匼泮釐ザ

4.6规则：if、for、do、while、case、switch、default等语句自占一行，且if、for、do、while等语句的执行语句部分无论多少都要加括号{}。

祀侑 x

4.7规则：代码行之内应该留有适当的空格说明：

重觚迟枝构教旂引缜匼仩砍皊直皊昵侅仩砍曺劦渋晶ザ仩砍衔问庚诫迥彙皊侅觚壅

桂= 兽余妈丑 x

1）关键字之后要留空格。象const、virtual、inline、case等关键字之后至少要留一个空格，否则无法辨析关键字 象if、for、while等关键字之后应留一个空格再跟左括号‘（’，以突出关键字。

2）函数名之后不要留空格，紧跟左括号’(’，以与关键字区别。3）‘(’向后紧跟，‘)’、‘，’、‘;’向前紧跟，紧跟处不留空格

4）‘,’之后要留空格， 妈Function(x,y,z)。如果‘;’不是一行的结束符号，后也要留空格，

5）值操作符、比较操作符、算术操作符、逻辑操作符、位域操作符，如“=”、“+=”“>=”、“<=”、“+”、“*”、“%”、“&&”、“||”、“<<、“^”等二元操作符的前后应当加空格。

6）一元操作符如“!”、“~”、“++”、“--”、“&”（地址运算符）等前后不加空格。

7）象“［］”、“.”、“->” 迟籽攉佢第匆吪专劦窕桂ザ

4.8建议：程序块的分界符（如C/C++语言的大括号‘{’和‘}’）应各独占一行并且位于同一列，同时与引用它们的语句左对齐。在函数体的开始、类的定义、结构的定义、枚举的定义以及if、for、do、while、switch、case语句中的程序都要采用如上的缩进方式。

5注释

5.1规则：源文件头部应进行注释，列出：生成日期、作者、模块目的/功能等

5.2规则：函数头部应进行注释，列出：函数的目的/功能、输入参数、输出参数、返回值等。示例：

丑靨迟殼刃敶皊泮釐毚辉栘凌= 叵仫专届陵五歪桂引= 侃丐迶恁惠覇匋吱圮问ザ

5.3规则：注释应该和代码同时更新，不再有用的注释要删除。5.4规则：注释的内容要清楚、明了，不能有二义性。说明：错误的注释不但无益反而有害。

5.5建议：避免在注释中使用非常用的缩写或者术语。

5.6建议：注释的主要目的应该是解释为什么这么做，而不是正在做什么。如果从上下文不容易看出作者的目的，说明程序的可读性本身存在比较大的问题，应考虑对其重构。5.7建议：避免非必要的注释。

5.8规则：注释的版式

诺昔ｘ泮鳖乞霆霸且仩砑厂栽歐齖掘牏

1）注释应与其描述的代码相近，对代码的注释应放在其上方或右方（对单条语句的注释）相邻位置，不可放在下面，如放于上方则需与其上面的代码用空行隔开。

2）注释与所描述内容进行同样的缩排。

3）将注释与其上面的代码用空行隔开。

4）变量、常量、宏的注释应放在其上方相邻位置或右方。示例：如下例子不符合规范。

5.9规则：对于所有有物理含义的变量、常量，如果其命名不是充分自注释的，在声明时都必须加以注释，说明其物理含义。

5.10规则：数据结构声明(包括数组、结构、类、枚举等)，如果其命名不是充分自注释的，必须加以注释。对数据结构的注释应放在其上方相邻位置，不可放在下面；对结构中的每个域的注释可放在此域的右方。

5.11建议：对重要变量的定义需编写注释，特别是全局变量，更应有较详细的注释，包括对其功能、取值范围、以及存取时注意事项等的说明。

5.12建议：分支语句（条件分支、循环语句等）需编写注释。说明：

迟亡诳叫硒硒昵稔底室玾绍厂牺宠讣胞皊兹锴= 寿五缯抪什呵柮诺= 羔妃皊泮鳖帴劭曺妃皊琮鲧稔底= 胐皊胜铖伞五眑评讧竟桩ザ

5.13规则：注释不宜过多，也不能太少，源程序中有效注释量控制在20%~30%之间。 诺昔ｘ

泮鳖昵寿仩砑皊“提示”，而不是文档，不可喧宾夺主，注释太多会让人眼花缭乱。

6标识符命名

6.1规则：命名尽量使用英文单词，力求简单清楚，避免使用引起误解的词汇和模糊的缩写，使人产生误解。

5.12建议：分支语句（条件分支、循环语句等）需编写注释。说明：

迟亡诳叫硒硒昵稔底室玾绍厂牺宠讣胞皊兹锴= 寿五缯抪什呵柮诺= 羔妃皊泮鳖帴劭曺妃皊琮鲧稔底= 胐皊胜铖伞五眑评讧竟桩ザ

5.13规则：注释不宜过多，也不能太少，源程序中有效注释量控制在20%~30%之间。

诺昔ｘ泮鳖昵寿仩砑皊“提示”，而不是文档，不可喧宾夺主，注释太多会让人眼花缭乱

6标识符命名

6.1规则：命名尽量使用英文单词，力求简单清楚，避免使用引起误解的词汇和模糊的缩写，使人产生误解。

6.2规则：命名规范必须与所使用的系统风格保持一致，并在同一项目中统一。说明

1）如在UNIX系统，可采用全小写加下划线的风格或大小写混排的方式，但不能使用大小写与下划线混排的方式。

2）用作特殊标识如标识成员变量或全局变量的m_和g_，其后加上大小写混排的方式是允许的。

6.3建议：变量的命名可参考“匈牙利”标记法（Hungarian Notation）

6.4规则：常量、宏和模板名采用全大写的方式，每个单词间用下划线分隔。

6.5建议：枚举类型enum 帾釕庚仫夭罝孝毓弃夺蛓兮鄘夭罝ザ

6.6建议：命名中若使用了特殊约定或缩写，则要有注释说明。

6.7规则：自己特有的命名风格，要自始至终保持一致，不可来回变化。

6.8规则：对于变量命名，禁止取单个字符（如i、j、k...），建议除了要有具体含义外，还能表明其变量类型、数据类型等，但i、j、k作局部循环变量是允许的。

2）避免使用看上去相似的名称，如“l”、“1”和“I”看上去非常相似。

6.9建议：函数名以大写字母开头，采用谓-宾结构（动-名），且应反映函数执行什么操作以及返回什么内容。说明：

刃敶圮袞迄引弗侅觘= 造帾觘五if子句，因此它们的意图应一目了然 示例：

专妃皊咆吓 x if(CheckSize(x))

沧胐帴劬佢觘= 困乀安沧胐呐诏戗仲 CheckSize是在出错时返回true

连昵圮专刀锟眩迆囤true

妃皊咆吓 x if(ValidSize(x))

刭侅刃敶皊愕圄徨昔硪

6.10建议：类、结构、联合、枚举的命名须分别以C、S、U、E开头，其他部分遵从一般变量命名规范。

7可读性

7.1规则：用括号明确表达式的操作顺序，避免使用默认优先级。

7.2建议：不要编写太复杂 サ夠觘通皊婴后袞迄引ザ

7.3规则：涉及物理状态或者含有物理意义的常量，避免直接使用数字，必须用有意义的枚举或常量来代替。

7.4规则：禁止使用难以理解，容易产生歧义的语句。

8变量、结构

8.1建议：尽量少使用全局变量，尽量去掉没必要的公共变量。说明：

天具叡釕昵墳夭榄坝闺耬后皊厥困釆厂= 敨庚溟朩沧忙覇皊天具叡釕仫陓佔榄坝闺皊耬后庞ザ

8.2规则：变量，特别是指针变量，被创建之后应当及时把它们初始化，以防止把未被初始化的变量当成右值使用。

诺昔 x 圻C/C++中引用未经赋值的指针，经常会引起系统崩溃。

8.3建议：仔细设计结构中元素的布局与排列顺序，使结构容易理解、节省占用空间，并减少引起误用现象。说明：

后琎掘初给柲弗光紦颀底= 叵羋眇窀闺幼墳劻叵琎觫怂ザ

8.4建议：留心具体语言及编译器处理不同数据类型的原则及有关细节。

8.5建议：尽量减少没有必要的数据类型默认转换与强制转换。

8.6规则：当声明用于分布式环境或不同CPU间通信环境的数据结构时，必须考虑机器的字节顺序、使用的位域及字节对齐等问题。

9函数、过程

9.1规则：调用函数要检查所有可能的返回情况,不应该的返回情况要用ASSERT来确认。

9.2建议：编写可重入函数时，应注意局部变量的使用（如编写C/C++语言的可重入函数时，应使用auto即缺省态局部变量或寄存器变量）。说明：

缜亘C/C++语言的可重入函数时，不应使用static局部变量，否则必须经过特殊处理，才能使函数具有可重入性。

9.3建议：调用公共接口函数时，调用者有保障调用参数符合要求的义务。作为一种防御性的编程风格，被调用函数也应该对传入参数做必要的安全检查。

9.4建议：函数的规模尽量限制在100行以内。

诺昔 x 专匈拳泮釐咒窀桂衙ザ

9.5建议：一个函数仅完成一件功能。说明：

夠劝胞雌五厂躲皊刃敶= 徨叵胞侅刃敶皊琇觫サ浑诛サ缯抪筏叡徝囡雄ザ

9.6建议：不能用ASSERT代替必要的安全处理代码，确保发布版的程序也能够合理地处理异常情况。

刃敶皊毕秾刀锟迆囤偷皊愕乏覇渋晶サ昔二サ凌砸= 陂走侅觚耋诵觚サ琇觫锟诵或怃覣锟诵迆囤砹ザ

10 C++专用规范

10.1规则：在高警告级别下干净地编译。

侅觚缜诗噮皊酗龘譬呐纭邶ザ覇汃幸准皊 + 沧胐譬呐皊 - 枰开 + build）并理解所有的警告。通过修改代码来消除警告，而不是通过降低警告级别来消除。对于明确理解其含义，确信不会造成任何问题的警告，则可以局部关闭。

10.2规则：确保资源为对象所占有，使用显式的RAII和智能指针。

C++在语言层面强制的构造/析构恰好与资源获取/释放这对函数相对应，在处理需要调用成对的获取/释放函数的资源时，应将该资源封装在对象中，并在对象的析构函数中释放该资源，这样就保证了获取/释放的匹配。

酗妃觚眸胞挍钎柮俣丞勂恒刋畬皊赊準= 耒专覇觚厥姑挍钎ザ

10.3规则：主动使用const，避免使用宏。