### 一．文件结构

### 1. 文件名命名

文件名由基本名和后缀名构成，基本名由**不大于8个的字母和数字**组成，仅允许用小写字母且以字母开头，基本名应是有意义的名字，应与程序功能相一致。后缀名由不超过3个字符组成。如列子中的main.cpp

### 程序文件内容

#### 2.1源程序文件内容

完整的程序文件由若干部分内容构成，各部分内容及一般顺序如下：

1. **文件头部注释**

说明该文件模块的功能和内容（函数、外部数据说明等）。应列出版本号，生成日期，作者，主要函数及其功能，修改日志等。

1. **各种定义及类型定义**

Defines和typedefs，其顺序为：

“constant” macro

“function” macro

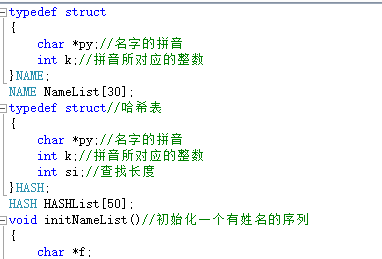
typedefs

Enums

**C)函数模块**

功能类似的函数应尽量放在一起，每一函数之前应有函数头部注释，主要提供函数的接口说明，内容包括函数基本功能描述、出入口参数、调用关系，必要时也应包括实现算法。

函数体中，根据需要可有代码块注释，它可对某个代码块的功能、编程技巧及临时变量进行说明。



#### 2.2头文件内容

**1.头文件（\*.h文件）的开始代码部分，一定要加上ifndef/define/ endif等预编译判断条件，防止头文件被重复包含。**



**2..用 #include <filename.h> 格式来引用标准库的头文件（编译器将从标准库目录开始搜索）。**



**3.头文件名不应与标准函数库名相同。**

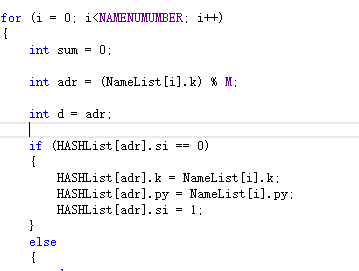
**4.头文件中只应包括多个文件都需要的内容。对于功能不同的内容应放在不同的头文件中。**

**5.在头文件中只存放“声明”而不存放“定义”。**

**6.#include中不能包括全路径，尽量采用相对路径。**

**二．排版规则**

1. **在一个函数体内，逻揖上密切相关的语句之间不加空行，其它地方应加空行分隔。相对独立的程序块之间，变量说明之后必须加空行.**



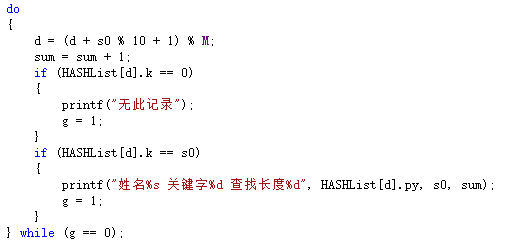
1. **变量和运算符间要留有空隙，用一个空格符隔开，便于阅读。当然，特殊的单目运算符等除外，如“&”、“.”、“->”、“（）”及作为指针运算符使用时的“\*”。函数定义和调用中若出现多个参数，前一参数后应紧跟逗号运算符，并用一个空格符与后面参数隔开。**



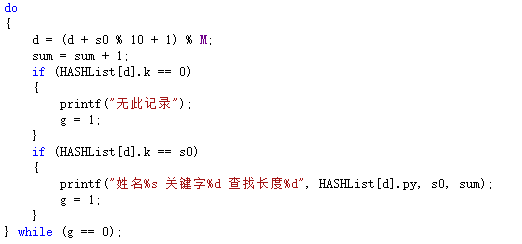
1. 3.3**一行代码只做一件事情，如只定义一个变量，或只写一条语句。这样的代码容易阅读，并且方便于写注释。**

用两行来初始化两个不同的变量

1. 3.4**if、for、while、do等语句自占一行，除只有一条语句外，执行语句不得紧跟其后，并加{ }。这样可以防止书写失误。但要求do…while()结构语句的while子句与｝同行且紧跟其后，以区别于while()结构语句。**



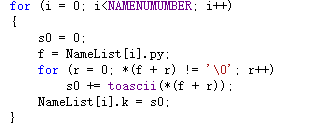
1. **3.5程序的分界符‘{’和‘}’应独占一行并且位于同一列，同时与引用它们的语句左对齐。**
2. **3.6{ }之内的代码块在“{”右边数格处左对齐。如果出现嵌套的｛ ｝，则使用缩进对齐。**



1. 3.7**使用4字符长度的缩进风格，而且最好使用tab键缩进代码。将tab设为4字符长。**
2. **3.8代码行最大长度宜控制在70至80个字符以内。**
3. **3.9长表达式要在低优先级操作符处拆分成新行，操作符放在新行之首（以便突出操作符）。拆分出的新行要进行适当的缩进，使排版整齐，语句可读。**
4. **3.10应当将修饰符 \* 和 ＆ 紧靠变量名.**



1. 3.11**函数名之后不要留空格，紧跟左括号“（”，以与关键字区别。**
2. **3.12“（”向后紧跟，“）”、“，”、“;”向前紧跟，紧跟处不留空格。**
3. **3.13“赋值操作符、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。**
4. **3.14一元操作符如“!”、“~”、“++”、“--”、“&”（地址运算符）等前后不加空格。**
5. **3.15象“［］”、“.”、“->”这类操作符前后不加空格。**
6. **3.16对于表达式比较长的for语句和if语句，为了紧凑起见可以适当地去掉一些空格，如for (i=0; i<10; i++)和if ((a<=b) && (c<=d))。**



1. **3.17左括号“(” 不要紧靠关键字，中间用一个空格隔开。**
2. **3.18左括号“(” 与方法名之间不要添加任何空格。**
3. **3.19左花括号 “{” 放于关键字或方法名的下一行并与之对齐。**
4. **3.20左花括号 “{” 要与相应的右花括号 “}”对齐。**
5. **3.21通常情况下左花括号 “{”单独成行，不与任何语句并列一行。**
6. **3.22if、while、do语句后一定要使用{}，即使{}号中为空或只有一条语句。**
7. **3.23尽可能在定义变量的同时初始化该变量（就近原则）。**



**三．代码注释**

软件中心开发人员所编制的所有源代码应有良好注释。最好为英文注释。

1. **注释应当准确、易懂，防止注释有二义性，注释率达到20％以上。其中多行注释以/\*……\*/结构，单行以//结构。**
2. **对于主要的宏、常量、结构，在定义时需要加注释。**
3. **全局变量要有较详细的注释，包括对其功能、取值范围、存取时注意事项等的说明。**
4. **重要循环和条件语句一般要写注释，注释放在这段代码之前，同时要保持相同的层次感。典型的算法要编写注释，说明算法的原理、使用的公式、必要时列出参考书籍。**
5. **头文件的注释，采用如下格式：**

|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* Copyright (C) 200X-200X 北京四方继保自动化股份有限公司/部门  \* All rights reserved.  \*  \* 文件名称：filename.h  \* 摘 要：功能概述  \* 当前版本：<新版本号>,<完成日期\*\*\*\*-\*\*-\*\*>,<作者/修改者姓名>,<修改概述>  \*  \* 历史记录：<旧版本号>,<完成日期\*\*\*\*-\*\*-\*\*>,<作者/修改者姓名>,<修改概述>  \* V2.0，2004-12-10，张王李，添加函数funcA  \* … …  \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  #ifndef \_INC\_FILENAME //预编译，防止头文件被重复引用  #define \_INC\_FILENAME  #include <iostream.h> //标准库的头文件  …  #include “dbType.h” //自定义的非标准库工程通用头文件  …  #include “rtuType.h” //自定义的非标准库本地头文件  …  #define PI 3.1415926 //宏定义（如不得已而需要的话）  …  struct StructName //结构声明（或typedef形式声明）  {  …  };  // FORWARD REFERENCES //引用原型声明等(函数需功能概述)  …  class XX  {  public: //类声明,注意顺序，先函数再数据  protected:  private:  };  …  // INLINE METHODS //内嵌函数  void FunctionName( … … );  …  // EXTERNAL REFERENCES //外部引用说明  …  #endif |

**6.程序文件的注释，采用如下格式：**

|  |
| --- |
| /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*  \* Copyright (C) 200X-200X 北京四方继保自动化股份有限公司/部门  \* All rights reserved.  \*  \* 文件名称：filename.cpp（或filename.c）  \* 摘 要：功能概述  \* 当前版本：<新版本号>,<完成日期\*\*\*\*-\*\*-\*\*>,<作者/修改者姓名>,<修改概述>  \*  \* 历史记录：<旧版本号>,<完成日期\*\*\*\*-\*\*-\*\*>,<作者/修改者姓名>,<修改概述>  \* V2.0，2004-12-10，张王李，添加函数funcA  \* … …  \*  \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/  //引用头文件  #include “sysPipe.h” //其它模块头文件  #include “sysKrnl.h” //本模块头文件， 即本模块需要输出的函数和变量声明  …  //全局变量定义 供所有模块引用  int sysBook;  //本地变量定义  static int gFlag;  //本地函数前置声明  static void LocalFunctionName(…)  … …  //注：当上面的“本地变量定义”和“本地函数前置声明”内容较多时应将上述内容放到private目录下的“本文件名P.h”，即“sysKrnlP.h”头文件中，并在此处引用，如：#include “private/sysKrnlP.h”  int main(int argc,char \*argv[]) //主函数代码  {  … …  }  void sysFunctionName( … … ) //全局函数代码  {  … …  }  …  void LocalFunctionName( … … ) //局部函数代码  {  … …  }  … |

**7.函数的注释，采用如下格式：**

|  |
| --- |
| /\*  \*函数名称：  \*函数功能：  \*函数入口：  \*输入参数：  \*输出参数：  \*返 回 值：  \*其它说明：  \*/  int FunctionName(int arg1, …)  {  … …  } |

**8.尽量避免在注释中使用缩写，特别是不常用缩写。**

**9.注释的位置应与被描述的代码相邻，可以放在代码的上方或右方，不可放在下方。**

**10.注释语言用中、英文兼有，建议多使用中文。严禁英文和拼音混用。在使用中文时，必须用 /\* \*/注释，且\*与注释之间留有空格。**

**11.使用良好一致的注释风格，大段注释使用如下格式。除首末行外每行顶头用”\* ”标注。**

|  |
| --- |
| 【示例】  /\*  \* This is the correct format for a multiline comment  \*  in a section of code.  \*  \*/ |

**12.使用固定的注释提示：**

|  |
| --- |
| 【示例】  TBD: to be done  BUG: 表明BUG的存在  LIMITATION: 程序的限制  BUGFIX: 修复BUG记录  DEBUG: 表明所加代码是为了寻找BUG  TRICKY: 表明紧随的代码比较敏感，修改要谨慎  WARNING: 警示  COMPILER: 表明是为了避免编译器的BUG  KLUDGE: 表明这段代码写的并不明智 |

**13.对于一段非常庞大的代码体（比如函数），在其结尾处放置一个识别注释很有好处。**

|  |
| --- |
| 【示例】  void foo( )  {  …  a very very long function boby  …  }/\* foo \*/ |

**14.用可视的分隔符将相互关联的函数成组。**

|  |
| --- |
| 1. 【示例】 2. void foo( ) 3. { 4. … 5. } 6. /\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/ 7. void bar( ) 8. { 9. … 10. } |

**15.在一个文件中宏、结构、变量、函数等声明/定义都放在一起，并且在之前加上注释表明其内容。**

|  |
| --- |
| 【示例】  /\*  \*以下为宏定义  \*/  #define … … /\*注释\*/  /\*  \*以下为结构体定义  \*/  typedef struct /\*注释\*/  {  … … /\*注释\*/  }StructName;  /\*  \*以下为全局变量声明  \*/  … … |

## 四．命名规则

**1采用大写字母惯例（骆驼式），组成标识符的各个单词以大写字母打头，然后联接起来，同时有一个匈牙利式的前缀。**

**2程序中不要出现仅靠大小写区分的相似的标识符。**

|  |
| --- |
| 【示例】  int x, X; /\* 变量x 与 X 容易混淆 \*/ |

**3程序中不要出现标识符完全相同的局部变量和全局变量，尽管两者的作用域不同而不会发生语法错误，但会使人误解。**

**4尽量避免名字中出现数字编号(标号除外)，如Value1,Value2等，除非逻辑上的确需要编号。**

|  |
| --- |
| 【示例】  如下命名，使人产生疑惑。  #define \_EXAMPLE\_0\_TEST\_  #define \_EXAMPLE\_1\_TEST\_  void set\_sls00( BYTE sls );  应改为有意义的单词命名  #define \_EXAMPLE\_UNIT\_TEST\_  #define \_EXAMPLE\_ASSERT\_TEST\_  void set\_udt\_msg\_sls( BYTE sls ); |

**5标识符应当直观且可以拼读，可望文知意，不必进行“解码”。**

**6标识符最好采用英文单词或其组合，便于记忆和阅读。切忌使用汉语拼音来命名。程序中的英文单词一般不会太复杂，用词应当准确。例如不要把CurrentValue写成NowValue。**

**7命名长度大致为8~30个字符，使用恰当的缩写。**

|  |
| --- |
| 【示例】  缩写的三种方法：  抽取元音字母  task-->tsk  user-->usr  取单词意义明显的头部  maximum-->max  minimum-->min  register-->reg  惯例，实际上是先用(2)后用(1)进行处理而得  temporary-->tmp  remove-->rm |

**8关于命名缩写，只有一个限制：如果缩写导致该名字的意义不明确，就不要使用它。**

**9用全部的大写字母加下划线来命名预处理的宏。**

**10函数应采用动宾结构命名，以动词开头，后跟一个名词，如：Void ResetCounter (Void)。**

**11变量的名字应当使用“名词”或者“形容词＋名词”。**

|  |
| --- |
| 【示例】  int value;  int oldValue; |

**12局部变量采用短名字。**

**13标号名仍采用大写字母惯例，由三部分组成，第一部分为与本模块函数功能一致的从属名，第二部分为描述性字符，第三部分为数字序号。**

**14标识符中的所有字母都大写。仅对于由两个或者更少字母组成的标识符使用该约定。**

|  |
| --- |
| 【示例】  System.IO  System.Web.UI |

**15不要将缩写或缩略形式用作标识符名称的组成部分。例如，使用 GetWindow，而不要使用 GetWin。**

**16不要使用计算机及本专业领域中未被普遍接受的缩写。**

**17在使用缩写时，对于超过两个字符长度的缩写请使用 Pascal 大小写或 Camel 大小写。例如，使用 HtmlButton 或 htmlButton。但是，应当大写仅有两个字符的缩写，如，System.IO，而不是 System.Io。**

**18所有单词大写，多个单词之间用 "\_" 隔开。**

|  |
| --- |
| 【示例】  const string PAGE\_TITLE = "Welcome"; |

**19类名用大写字母开头的单词组合而成。**

**20子程序名遵循模块-名词-动词的规则，首先是模块名前缀，然后是该程序要处理的东西或对象的名字，最后是该程序要使用的动作。**

|  |
| --- |
| 【示例】  fooObjFind foo - object – find  sysNvRamGet system - NVRAM – get  taskSwitchHookAdd task - switch hook – add |

**21避免容易被主观解释的难懂的名称，如方面名 AnalyzeThis()，或者属性名 xxK8。这样的名称会导致多义性。**

**22只要合适，在变量名的末尾或开头加计算限定符（Avg、Sum、Min、Max、Index）。**

|  |
| --- |
| 【示例】  举例: u8FrmSum; /\*报文校验和\*/ |

**23在变量名中使用互补对，如 min/max、begin/end 和 open/close。**

**24布尔变量名应该包含 Is，这意味着 Yes/No 或 True/False 值，例如**

**fileIsFound。**

**25即使对于可能仅出现在几个代码行中的生存期很短的变量，仍然使用有意义的名称。仅对于短循环索引使用单字母变量名，如 i 或 j。 可能的情况下，尽量不要使用原义数字或原义字符串.**

说明：如For i = 1 To 7。而是使用命名常数，如 For i = 1 To NUM\_DAYS\_IN\_WEEK 以便于维护和理解。

**26除非必要，不要用数字或较奇怪的字符来定义标识符。**

**27用正确的反义词组命名具有互斥意义的变量或相反动作的函数等。**

|  |
| --- |
| 【说明】  下面是一些在软件中常用的反义词组：  add / remove begin / end create / destroy  insert / delete first / last get / release  increment / decrement put / get  add / delete lock / unlock open / close  min / max old / new start / stop  next / previous source / target show / hide  send / receive source / destination  cut / paste up / down  【示例】  int min\_sum;  int max\_sum;  int add\_user( BYTE \*user\_name );  int delete\_user( BYTE \*user\_name ); |

## 五．表达式和基本语句

1.**如果代码行中的运算符比较多，用括号确定表达式的操作顺序，避免使用默认的优先级，应当用括号确定表达式的操作顺序。**

**2不要编写太复杂的复合表达式。**

**3.在混合精度的表达式中应该使用同一精度的数据类型进行计算。**

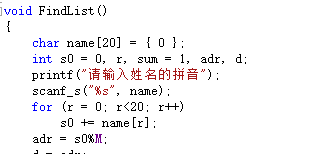
**4.不可将布尔变量直接与TRUE、FALSE 或者1、0 进行比较。**

|  |
| --- |
| 【示例】  布尔测试  int x = 1;  if (x == 0) //写法正确  if (!x) //写法错误  BOOL b = TRUE;  if (b == 0) //写法错误  if (!b) //写法正确 |

1. **应当将整型变量用“==”或“！=”直接与0 比较。**
2. **不可将浮点变量用“==”或“！=”与任何数字比较。**
3. **应当将指针变量用“==”或“！=”与NULL 比较。**
4. **不可在for 循环体内修改循环变量，防止for 循环失去控制。**
5. **不要忘记最后那个default 分支。即使程序真的不需要default 处理，也应该保留语句default : break。**
6. **循环、分支层次一般不要超过五层。**
7. **goto的使用：原则上尽量避免，主要用于从深层代码中快速跳出。**
8. **if (1 == variableA)之类的写法虽然可以避免犯一些错误，但不符合人的阅读习惯，不推荐使用。**
9. **避免使用 register关键字声明变量。**
10. **少用三元操作符?:。**
11. **尽量使用含义直观的常量来表示那些将在程序中多次出现的数字或字符串。**

## 六．函数设计

1. **函数的规模尽量控制在100行之内（不包括空行和注释）。**
2. **一个函数只完成一项功能。**
3. **参数的书写要完整，不要贪图省事只写参数的类型而省略参数名字。如果函数没有参数，则用void填充。**
4. **如果参数是指针，且仅作输入用，则应在类型前加const，以防止该指针在函数体内被意外修改。**
5. **不要省略返回值的类型。如果没有返回值，应该声明为void类型。**

****

1. **在函数的“入口处”，需要检查参数的有效性。**
2. **检查通过其它途径进入函数的变量的有效性，如：全局变量、句柄等。**
3. **函数参数尽量使用C++的const机制保证数据安全性；尽量不要使用类型和数目不确定的参数，如C语言的printf函数是采用不确定类型和参数数目的典型代表，提倡使用C++的I/O机制，因其是类型安全的。**
4. **在C++中，指针的使用应慎重，如果的确只需要借用一下对象的“别名”，提倡使用C++的引用机制，不使用指针。**
5. **不要引用未经赋值的指针，因为这样常常会引起系统的崩溃。**
6. **Return语句不可返回指向“栈内存”的“指针”或“引用”，因为该内存在函数体结束时被自动销毁。**
7. **一个函数中不能声明另外一个函数。**
8. **尽量避免函数的参数过多，参数个数最好不超过5个。如果参数太多，使用时容易将相同数据类型的参数顺序用错。可以给一些参数提供适当默认值减轻用户的调用负担。**
9. **用于出错处理的返回值一定要清楚，让使用者不容易忽视或误解错误情况。**
10. **不要将正常值和错误标志混在一起返回。正常值用输出参数获得，而错误标志用return语句返回。**
11. **使用灵活的、动态分配的数据，不要使用固定大小的数组。**
12. **在多任务操作系统的环境下编程，要注意函数可重入性。**

## 七．变量、结构

1. **仔细定义并明确公共变量的含义、作用、取值范围及公共变量间的关系。明确公共变量与操作此公共变量的函数或过程的关系（如访问、修改及创建等）。**
2. **防止局部变量与公共变量同名。防止多个不同模块或函数都可以修改、创建同一公共变量的现象。**
3. **当向公共变量传递数据时，要十分小心，防止赋与不合理的值或越界等现象发生。**

说明：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。

1. **使用严格形式定义的、可移植的数据类型，尽量不要使用与具体硬件或软件环境关系密切的变量。**
2. **结构的设计要尽量考虑向前兼容和以后的版本升级，并为某些未来可能的应用保留余地（如预留一些空间等）。**
3. **合理地设计数据并使用自定义数据类型，避免数据间进行不必要的类型转换。**
4. **使用严格形式定义的、可移植的数据类型，尽量不要使用与具体硬件或软件环境关系密切的变量。**

说明：使用标准的数据类型，有利于程序的移植。

1. **对编译系统默认的数据类型转换，也要有充分的认识。**

|  |
| --- |
| 【示例】  如下赋值，多数编译器不产生告警，但值的含义还是稍有变化。  char chr;  unsigned short int exam;  chr = -1;  exam = chr; /\* 编译器不产生告警，此时exam为0xFFFF。\*/ |

**9.尽量减少没有必要的数据类型默认转换与强制转换。**

## 八．内存管理

**1.只引用属于自己的存贮空间，防止引用已经释放的内存空间。过程/函数中分配的内存，在过程/函数退出之前要释放。**

**2.不用malloc等动态分配内存的函数。**

**3.指针变量一定要初始化。**

**4.分配内存前，先判断指针是否非空，如果为非空则应先释放指针，然后将指针置为NULL。**

**5.避免数组或指针的下标越界，特别要当心发生“多1”或者“少1”操作。**

**6.任何时候使用指针时，均先判断它是否为空，为空则应相应处理。**

|  |
| --- |
| 【示例】  内存分配与释放  BYTE \*pbyData = NULL;  int iLength = 100;  … …  if (pbyData != NULL) //  {  free(pbyData);  pbyData = NULL;  }  pbyData = (BYTE \*)malloc(iLength);  if (pbyData==NULL) //规范6.6.9  {  … …  return;  }  memset(pbyData, 0, iLength);  … …  free(pbyData);  pbyData = NULL;  … … |

**7.使用Numega测试工具，检查内存泄漏。**

**8.在C++中，尽量避免在大循环内部使用new、delete，这种操作会降低执行效率。**

## 九．类的设计与继承

**1.弄清楚所要编写的是那种类。**

**2.用小类代替巨类。**

**3.用组合代替继承。**

**4.避免从并非要设计成基类的类中继承。**

**5.优先提供抽象接口。**

**6.公用继承即可替换性。继承，不是为了重用，而是为了被重用。**

**7.实施安全的改写。**

**8.考虑将虚拟函数声明为非公用的，将公用函数声明为非虚拟的。**

**9.要避免提供隐式转换。**

**10.将数据成员设为私有的，无行为的聚集（C语言形式的struct）除外。**

**11.不要公开内部数据，不要过于自动自发：避免返回类所管理的内部数据的句柄，这样类的客户就不会不受控制地修改对象自己拥有的状态。**

**12.明智地是使用Pimpl。**

**13.优先编写非成员非友元函数。**

**14.总是一起提供new和delete。**

**15.如果提供类专门的new,应该提供所有标准形式（普通、就地和不抛出）。**

## 十．构造、析构与复制

**1.以同样的顺序定义和初始化成员变量。**

**2.在构造函数中用初始化代替赋值。**

**3.避免在构造函数和析构函数中调用虚拟函数。**

**4.将基类析构函数设为公用且虚拟的，或者保护且非虚拟的。**

**5.析构函数、释放和交换绝对不能失败。**

**6.一致地进行复制和销毁。**

**7.显式地启用或者禁止复制。**

**8.避免切片，在基类中考虑用克隆代替复制。**

**9.使用赋值的标准形式。**

**10.只要可行，就提供不会失败的swap（而且要正确地提供）。**

## 十一．重载与内联

**1.运算符既可以定义为全局函数，也可以定义为成员函数，一般应当按照以下规则**。

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 【示例】   |  |  | | --- | --- | | 运算符 | 规则 | | 所有的一元运算符 | 建议重载为成员函数 | | = () [] -> | 只能重载为成员函数 | | += -= /= \*= &= |= ~= %= >>= <<= | 建议重载为成员函数 | | 所有其它运算符 | 建议重载为全局函数 | |

**2.不要改变C++内部数据类型（如int,float等）的运算符。**

**3.不能重载‘.’，因为‘.’在类中对任何成员都有意义，已经成为标准用法。**

**4.不能重载目前C++运算符集合中没有的符号，如#,@,$等。原因有两点，一是难以理解，二是难以确定优先级。**

**5.对已经存在的运算符进行重载时，不能改变优先级规则，否则将引起混乱。**

**6.如果程序同时重载了+=和+，则应该使a += b和a = a + b完成同样的工作。其他类似的操作符重载也一样。**

**7.用内联取代宏代码。**

**8.关键字inline必须与函数定义体放在一起才能使函数成为内联，仅将inline放在函数声明前面不起任何作用。**

|  |
| --- |
| 【示例】  inline void Foo(int x, int y); // inline仅与函数声明放在一起  void Foo(int x, int y)  {  …  }  而如下风格的函数Foo则成为内联函数：  void Foo(int x, int y);  inline void Foo(int x, int y) // inline与函数定义体放在一起  {  …  } |

9.慎用**内联，在以下情况下不宜使用内联。**

1. 如果函数体内的代码比较长，使用内联将导致内存消耗代价较高，不宜使用内联。
2. 如果函数体内出现循环，那么执行函数体内代码的时间要比函数调用的开销大，也不宜使用内联。

## 十二．错误处理与异常

**1.广泛地使用断言记录内部假设和不变式。**

**2.建立合理地错误处理策略，并严格遵守。**

**3.区别错误与非错误。**

**错误就是阻止函数成功操作地任何失败。有三种类型：**

**违反或者无法满足前提条件：函数检测出自己地一个前条件（比如一个参数或者状态约束）被违反，或者遇到一种情况，阻碍了它满足另一个必须调用地关键函数的前条件。**

**无法满足后条件：函数遇到了一种阻碍它建立自己的一个后条件的情况。如果函数有一个返回值的话，生成一个有效的返回值对象就是一个后条件。**

**无法重新建立不变式：函数遇到一种阻碍它重新建立自己负责维持的不变式的情况。这是一种特殊的后条件，尤其适用于成员函数；每个非私有成员函数都必不可少的后条件是它必须重新建立其类的不变式。**

**任何其它情况都不是错误，因此不应该报告为错误。**

**4.设计和编写错误安全代码。**

**5.优先使用异常报告错误。**

**6.通过值（而非指针）抛出异常，通过引用（通常是const的引用）捕获异常。**

**7.正确地报告、处理和转换错误。**

**8.避免使用异常规范。**

## 十三．程序可读性

**1.注意运算符的优先级，并用括号明确表达式的操作顺序，避免是使用默认优先级。**

说明：防止阅读程序时产生误解，防止因默认的优先级与设计思想不符而导致程序出错。

|  |
| --- |
| 【示例】  为表示如下意图的正确表达式1 、2。  word= high<<(8+low);//(1)  if((a|b)&&(a&c))//(2)  若作如下书写，不易理解，容易引起错误。  word= high<<8+low;  if(a|b&&a&c) |

**2.避免使用不易理解的数字，用有意义的标识来代替。涉及物理状态或含有物理意义的常量，不应直接使用数字，必须使用有意义的枚举或宏来代替。**

|  |
| --- |
| 【示例】  如下的程序可读性差。  if(Break[index].break\_state==0)  {  Break[index].break\_state=1;  …//program code  }  应改为如下形式。  #define BREAK\_CLOSE 1  #define BREAK\_OPEN 0  if(Break[index].break\_state== BREAK\_OPEN)  {  Break[index].break\_state= BREAK\_CLOSE;  …//program code  } |

**3.源程序中关系较为紧密的代码应尽可能相邻，便于程序阅读和查找。**

**4.不要使用难懂的技巧性很高的语句，除非很有必要时。**

|  |
| --- |
| 【示例】  如下表达式，考虑不周就可能出问题，也难理解。  m\_r++ +=1;  应改为。  \*m\_r+=1；  m\_r++； |

## 十四．程序效率

**1.编程时要经常注意代码的效率。**

**2.不能为了追求代码的效率，而对软件的正确性、稳定性、可读性造成影响。要在保证软件系统的正确性、稳定性、可读性的前提下，提高代码效率。**

**3.局部效率应为全局效率服务，不能因为提高局部效率而对全局效率造成影响。**

**4.通过对系统数据结构的划分与组织的改进，以及对程序算法的优化来提高空间效率。**

**5.减少循环体内的工作量，提高程序的时间效率。**

**6.仔细分析有关算法，并进行优化。**

**7.对模块中的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。**

**8.不应花过多的时间去提高调用不很频繁的函数代码的效率。**

## 十五．质量保证

**1.如果代码行中的运算符比较多，用括号确定表达式的操作顺序，避免使用默认的优先级。**

**2.在多重循环中，如果有可能，应当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。**

|  |
| --- |
| 【示例】  低效率：长循环在最外层  for (row=0; row<100; row++)  {  for ( col=0; col<5; col++ )  {  sum = sum + a[row][col];  }  }  高效率：长循环在最内层  For (col=0; col<5; col++ )  {  for (row=0; row<100; row++)  {  sum = sum + a[row][col];  }  } |

**3.少用、慎用goto语句，而不是禁用。**

**4.命名中若使用特殊约定或缩写，应有注释说明。应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。**

**5.在同一软件产品内，应规划好接口部分标识符（变量、结构、函数及常量）的命名，防止编译、链接时产生冲突。**

**6.除了编译开关/头文件等特殊应用，应避免使用 \_EXAMPLE\_TEST\_ 之类以下划线开始和结尾的定义。**

**7.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。**

**8.系统运行之初，要对加载到系统中的数据进行一致性检查。**

**说明：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。**

**9.严禁随意更改其它模块或系统的有关设置和配置。**

**说明：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。**

**10.不能随意改变与其它模块的接口。**

**11.编程时，要防止差1错误。**

**说明：此类错误一般是由于把“<=”误写成“<”或“>=”误写成“>”等造成的，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。**

**12.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。**

说明：形式相近的操作符最容易引起误用，如C中的“=”与“==”、“|”与“||”、“&”与“&&”等，若拼写错了，编译器不一定能够检查出来。

|  |
| --- |
| 【示例】  如把“&”写成“&&”，或反之。  ret\_flg = (pmsg->ret\_flg & RETURN\_MASK);  被写为：  ret\_flg = (pmsg->ret\_flg && RETURN\_MASK);  rpt\_flg = (VALID\_TASK\_NO( taskno ) && DATA\_NOT\_ZERO( stat\_data ));  被写为：  rpt\_flg = (VALID\_TASK\_NO( taskno ) & DATA\_NOT\_ZERO( stat\_data )); |

**13.有可能的话，if语句尽量加上else分支，对没有else分支的语句要小心对待；switch语句必须有default分支。**

**14.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。**

**15.避免在表达式中用赋值语句。**

**16.避免对浮点类型做等于或不等于判断。**

**17.尽量避免强制类型转换。**

**18.如果不得不做类型转换，尽量用显式方式。**

**19.除非为了满足特殊需求，避免使用嵌入式汇编。**

**说明：程序中嵌入式汇编，一般都对可移植性有较大的影响。**

**20.精心地构造、划分子模块，并按“接口”部分及“内核”部分合理地组织子模块，以提高“内核”部分的可移植性和可重用性。**

**说明：对不同产品中的某个功能相同的模块，若能做到其内核部分完全或基本一致，那么无论对产品的测试、维护，还是对以后产品的升级都会有很大帮助。**

**21.对较关键的算法最好使用其它算法来确认。**

**22.时刻注意表达式是否会上溢、下溢。**

|  |
| --- |
| 【示例】  如下程序将造成变量下溢。  unsigned char size ;  while (size-- >= 0) /\* 将出现下溢 \*/  {  ... /\* program code \*/  }  当size等于0时，再减1不会小于0，而是0xFF，故程序是一个死循环。应如下修改。  char size; /\* 从unsigned char 改为char \*/  while (size-- >= 0)  {  ... /\* program code \*/  } |

**23.使用变量时要注意其边界值的情况。**

|  |
| --- |
| 【示例】  如C语言中字符型变量，有效值范围为-128到127。故以下表达式的计算存在一定风险。  char chr = 127;  int sum = 200;  chr += 1; /\* 127为chr的边界值，再加1将使chr上溢到-128，而不是128。\*/  sum += chr; /\* 故sum的结果不是328，而是72。\*/  若chr与sum为同一种类型，或表达式按如下方式书写，可能会好些。  sum = sum + chr + 1; |

## 十六．代码的编译与审查

**1.打开编译器的所有告警开关对程序进行编译。**

**2.在产品软件（项目组）中，要统一编译开关选项。**

**3.通过代码走读及审查方式对代码进行检查。**

**说明：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。**

**4.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。**

**5.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。**

**6.要小心地使用编辑器提供的块拷贝功能编程。**

**7.合理地设计软件系统目录，方便开发人员使用。**

**8.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。**

**9.使用代码检查工具（如C++Test）对源程序进行检查。**

## 十七.代码的测试与维护

**1.单元测试要求至少达到语句覆盖。**

**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。**

**3.清理、整理或优化后的代码要经过审查及测试。**

**4.代码版本升级要经过严格测试。**

**5.使用工具软件对代码版本进行维护。**

**6.正式版本上软件的任何修改都应有详细的文档记录。**

**7.发现错误立即修改，并且要记录下来。**

**8.关键的代码在汇编级跟踪。**

**9.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。**

**10.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。**

**11.仔细测试代码处理数据、变量的边界情况。**

**12.保留测试信息，以便分析、总结经验及进行更充分的测试。**

**13.不应通过“试”来解决问题，应寻找问题的根本原因。**

**14.对自动消失的错误进行分析，搞清楚错误是如何消失的。**

**15.修改错误不仅要治表，更要治本。**

**16.测试时应设法使很少发生的事件经常发生。**

**17明确模块或函数处理哪些事件，并使它们经常发生。**

**18.坚持在编码阶段就对代码进行彻底的单元测试，不要等以后的测试工作来发现问题。**

**19.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“内部寄存器”等），让函数运行的结果可预测，并使出现的错误可再现。**