**竞赛题目及解释**

**1、头文件应当自包含，且没有**循环依赖

1）自包含就是任意一个头文件均可独立编译。如果一个文件包含某个头文件，还要包含另外一个头文件才能工作的话，给这个头文件的用户增添不必要的负担。

比如，如果a.h不是自包含的，需要包含b.h才能编译，会带来的危害：每个使用a.h头文件的.c文件，为了让引入的a.h的内容编译通过，都要包含额外的头文件b.h。额外的头文件b.h必须在a.h之前进行包含，这在包含顺序上产生了依赖。

另外，不能为了让a.h自包含，而在a.h中包含不必要的头文件。a.h要刚刚可以自包含，不能在a.h中多包含任何满足自包含之外的其他头文件。

2）头文件循环依赖，指 a.h 包含 b.h，b.h 包含 c.h，c.h 包含 a.h， 导致任何一个头文件修改，都导致所有包含了a.h/b.h/c.h的代码全部重新编译一遍。而如果是单向依赖，如a.h包含b.h，b.h包含c.h，而c.h不包含任何头文件，则修改a.h不会导致包含了b.h/c.h的源代码重新编译。

**2、模块内部函数参数的合法性检查，由调用者负责**

对于模块外部传入的参数，必须进行合法性检查，保护程序免遭非法输入数据的破坏。模块内部函数调用，缺省由调用者负责保证参数的合法性，如果都由被调用者来检查参数合法性，可能会出现同一个参数，被检查多次，产生冗余代码，很不简洁。由调用者保证入参的合法性，这种契约式编程能让代码逻辑更简洁，可读性更好。  
示例：

int SomeProc(...)

{

int data;

bool dataOK = GetData(&data); // 获取数据

if (!dataOK) { // 检查上一步结果，其实也就保证了数据全法性

return -1;

}

DealWithData(data); // 调用数据处理函数

...

}

void DealWithData(int data)

{

if (data < MIN || data > MAX) { // Bad: 调用者已经保证了数据合法性

return;

}

...

}

**3、对函数的错误返回码要全面处理**

一个函数（标准库中的函数/第三方库函数/用户定义的函数）能够提供一些指示错误发生的方法。这可以通过使用错误标记、特殊的返回数据或者其他手段，不管什么时候函数提供了这样的机制，调用程序应该在函数返回时立刻检查错误指示。

示例：

char fileHead[128];

ReadFileHead(fileName, fileHead, sizeof(fileHead)); // Bad: 未检查返回值

DealWithFileHead(fileHead, sizeof(fileHead)); // fileHead 可能无效

正确写法：

char fileHead[128];

ret = ReadFileHead(fileName, fileHead, sizeof(fileHead));

if (ret != OK) { // Good: 确保 fileHead 被有效写入

return ERROR;

}

DealWithFileHead(fileHead, sizeof(fileHead)); // 处理文件头

注意，当函数返回值被大量的显式(void)忽略掉时，应当考虑函数返回值的设计是否合理、有必要。如果所有调用者都不关注函数返回值时，请将函数设计成void型。

**4、除了常见的、通用缩写外，不使用单词缩写，不得使用汉语拼音**

简短的命名总是方便阅读的，但前提是容易理解。在命名中，使用常见、通用的缩写是允许并被推荐的。某个系统的专用缩写，或局部范围内形成共识的缩写，也是可以的。

一些常见可以缩写的例子：

| **单词** | **惯用缩写** | **单词** | **惯用缩写** |
| --- | --- | --- | --- |
| argument | arg | buffer | buf |
| clock | clk | command | cmd |
| compare | cmp | configuration | cfg |
| device | dev | error | err |
| hexadecimal | hex | increment | inc |
| initialize | init | maximum | max |
| message | msg | minimum | min |
| parameter | para | previous | prev |
| register | reg | semaphore | sem |
| statistic | stat | synchronize | sync |
| temp | tmp |  |  |

**5、函数命名无法表达的信息，必须加函数头注释辅助说明**

函数头注释统一放在函数声明或定义上方。选择并统一使用如下风格之一：  
**使用'//'写函数头**

// 单行函数头

int Func1(void);

// 多行函数头

// 第二行

int Func2(void);

**使用'/\*` '\*/' 写函数头**

/\* 单行函数头 \*/

int Func1(void);

/\*

\* 单行或多行函数头

\* 第二行

\*/

int Func2(void);

函数尽量通过函数名自注释，**按需**写函数头注释。不要写无用、信息冗余的函数头；不要写空有格式的函数头。函数头注释内容**可选**，但不限于：功能说明、返回值，性能约束、用法、内存约定、算法实现、可重入的要求等等。模块对外头文件中的函数接口声明，其函数头注释，应当将重要、有用的信息表达清楚。

例：

/\*

\* 返回实际写入的字节数，-1表示写入失败

\* 注意，内存 buf 由调用者负责释放

\*/

int WriteString(char \*buf, int len);

坏的例子：

/\*

\* 函数名：WriteString

\* 功能：写入字符串

\* 参数：

\* 返回值：

\*/

int WriteString(char \*buf, int len);

上面例子中的问题：

* 参数、返回值，空有格式没内容
* 函数名信息冗余
* 关键的 buf 由谁释放没有说清楚

**6、变量或内存块，按需初始化；禁止冗余清零**

初始化并不一定是赋零值或清零。变量定义时按需初始化。未赋值而直接使用，静态检查工具可以及时检查出来。冗余初始化不仅不简洁； 如果初始化成无效值或错误值，工具则无法起到保护作用。

常见的冗余初始化：

void \*p = NULL; // Bad: 冗余的赋NULL，将会被后面直接覆盖

...

p = malloc(MEM\_SIZE);

...

针对大数组的冗余清零，甚至可能会影响到性能。

char buf[VERY\_BIG\_SIZE] = {0};

memset(buf, 0, sizeof(buf)); // Bad: 冗余清零

因此，应该写简洁的代码，对变量或内存块，按需初始化。

C99不再限制局部变量定义必须在语句之前，可以按需定义，即在靠近变量使用的地方定义变量。这种简洁的做法，不仅将变量作用域限制更小，而且能解决定义变量时不知该怎么初始化的问题。如果编译环境支持，建议按需定义。