第九章 高质量编程

- 1 代码风格
- 2 Windows程序命名规则
- 3 函数处理规则
- 4 文件结构
- 5 程序的版式
- 6 表达式
- 7 基本语句
- 8 内存管理
- 9 其它编程经验
- 10 Java编程规则
- 11 C++编程规则

代码风格

- 两个例子

例（1）

If（1= = j）

If（ j= =1） 优秀

例（2）

If（i＞ MAX_NUM） 优秀

If（i＞ 5000）

Windows程序命名规则

- 匈牙利命名规则对照表

关键字首	数据类型
c	char
by	BYTE(无符号字节)
n	short
i	int
x,y	int 分别用作 x 和 y 座标
cx,cy	int 分别用作 x 和 y 长度,c 代表"计数器"
b 或 f	BOOL(int),f 代表"标志"
w	WORD(无符号短整型)
l	LONG(有符号长整型)
dw	DWORD(无符号长整型)
fn	function(函数)
s	string(字符串)
sz	以数组值为 0 结尾的字串
h	句柄
p	指针

Windows程序命名规则

- 常用命名规则
  - 对于一般标示符，适当的使用简写形式，以最短的组合词表达所需要表达的意义。
  - 程序中不要仅靠大小写来区分相似的标识符。
  - 程序中尽量不要出现与标示符完全相同的局部变量和全局变量。
  - 变量的名字应当使用“名词”或者“形容词+名词”。
  - 全局函数的名字应当使用“动词”或者“动词+名词”。
  - 类的成员函数应当只使用“动词”，被省略掉的名词就是对象本身
  - 用正确的反义词组命名具有互斥意义的变量或者相反动作的函数。

Windows程序命名规则

- 常用命名规则（续）
  - 类名和函数名用大写字母开头的单词组合而成；变量和参数用小写字母开头的单词组合而成；常量全用大写的字母，用下划线分割单词。
  - 静态变量加前缀s_。如果必须定义使用全局变量，则在全局变量前加g_。
  - 类的数据成员加前缀m_，可以避免数据成员与成员函数的参数同名。
  - 为了防止某一软件库中的一些标识符和其他软件库中的标识符冲突，可以为各种标识符加上反映软件性质的前缀。
  - 使用i、j、k、l、m作为循环计数变量。

函数处理规则

- 函数参数、返回值和接口部分的规则
  - 参数的书写要完整，不要只写参数的类型而不写参数名。函数没有参数时用void填充。
  - 参数命名要恰当，顺序要合理。
  - 如果参数是指针且仅做输入用，应该在类型前面加const，以防止该指针在函数体内被意外修改。
  - 如果输入参数以值传递的方式传递对象，宜改用“const &”方式来传递，这样可以省去临时对象的构造和析构过程，从而提高效率。
  - 避免参数太多，参数的个数尽量控制在5个以内。
  - 不要省略返回值的类型。
  - 函数名字与返回值类型在语义上不可冲突。

函数处理规则

- 函数参数、返回值和接口部分的规则（续）
  - 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，错误标志用return语句返回。
  - 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，错误标志用return语句返回。
  - 有时函数原本不需要返回值，但是为了增加灵活性，如支持链式表达，可以附加返回值
  - 如果函数的返回值是一个对象，要注意”引用传递”、”值传递”的不同使用。
  - 在函数体的”出口处”，对return语句的正确性和效率进行检查。

文件结构

定义文件的结构

定义文件的结构有三部分内容:

- 定义文件开头处的版权和版本声明。
- 对一些头文件的引用。
- 程序的实现体（包括数据和代码）

文件结构

- 版权和版本的声明
  - 版权信息。
  - 文件名称，标识符，摘要。
  - 当前版本号，作者/修改者，完成日期。
  - 版本历史信息。

文件结构

- 头文件的结构
  - 为了防止头文件被重复引用，应当用ifndef/define/endif结构产生预处理块。
  - 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）。
  - 用 #include “filename.h” 格式来引用非标准库的头文件（编译器将从用户的工作目录开始搜索）。
  - 头文件中只存放“声明”而不存放“定义”
  - 不提倡使用全局变量，尽量不要在头文件中出现象extern int value 这类声明。

程序的版式

- 空行规则
  - 在每个类声明、每个函数定义结束之后都要加空行。
  - 在一个函数体内，逻辑上密切相关的语句之间不加空行，其它地方应加空行分隔。

程序的版式

- 代码行
  - 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。
  - if、for、while、do等语句自占一行，执行语句不得紧跟其后。不论执行语句有多少都要加{}。这样可以防止书写失误。
  - 尽可能在定义变量的同时初始化该变量（就近原则）
    - 如果变量的引用处和其定义处相隔比较远，变量的初始化很容易被忘记。如果引用了未被初始化的变量，可能会导致程序错误。本建议可以减少隐患。

程序的版式

- 对齐
  - 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。
  - {}之内的代码块在‘{’右边数格处左对齐。
- 长行拆分
  - 代码行最大长度宜控制在70至80个字符以内。代码行不要过长，否则眼睛看不过来，也不便于打印。
  - 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。
  - 应当将修饰符 * 和 & 紧靠变量名。

```
int   *x, y;  // 此处y不会被误解为指针
```

程序的版式

- 注释
  - 注释是对代码的“提示”，而不是文档。程序中的注释不可喧宾夺主。注释的花样要少。
  - 如果代码本来就是清楚的，则不必加注释。否则多此一举，令人厌烦。例如 i++; //i 加 1，多余的注释
  - 边写代码边注释，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。
  - 注释应当准确、易懂，防止注释有二义性。错误的注释不但无益反而有害。
  - 尽量避免在注释中使用缩写，特别是不常用缩写。
  - 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。
  - 当代码比较长，特别是有多重嵌套时，应当在一些段落的结束处加注释，便于阅读。

程序的版式

➢ 类的版式

类的版式主要有两种方式：

- 将private类型的数据写在前面，而将public类型的函数写在后面。采用这种版式的程序员主张类的设计“以数据为中心”，重点关注类的内部结构。
- 将public类型的函数写在前面，而将private类型的数据写在后面。采用这种版式的程序员主张类的设计“以行为为中心”，重点关注的是类应该提供什么样的接口。
  - 建议读者采用“以行为为中心”的书写方式，即首先考虑类应该提供什么样的函数。
  - 这样做不仅让自己在设计类时思路清晰，而且方便别人阅读。因为用户最关心的是接口。

表达式

共性规则

- 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。
- 不要编写太复杂的复合表达式。
  - 例如：i = a >= b && c < d && c + f <= g + h;
- 不要有多用途的复合表达式。
  - 例如：d = (a = b + c) + r;
- 不要把程序中的复合表达式与“真正的数学表达式”混淆。
  - 例如：if (a < b < c) // a < b < c是数学表达式而不是程序表达式。并不表示if ((a<b) && (b<c))而是成了令人费解的if ((a<b)<c)。

基本语句

- if 语句
  - **不可将布尔变量直接与TRUE、FALSE或者1、0进行比较。**
  - 假设布尔变量名字为flag，

  它与零值比较的标准if语句如下：
    - if (flag)　　// 表示flag为真
    - if (!flag)　　// 表示flag为假
  - 其它的用法都属于不良风格，例如：
    - if (flag == TRUE)
    - if (flag == 1)
    - if (flag == FALSE)
    - if (flag == 0)

基本语句

- if 语句
  - **应当将整型变量用“==”或“！=”直接与0比较。**
  - 假设整型变量的名字为value，

  它与零值比较的标准if语句如下：
    - if (value == 0)
    - if (value != 0)
  - 不可模仿布尔变量的风格而写成
    - if (value)　// 会让人误解 value是布尔变量
    - if (!value)

基本语句

- if 语句
  - **不可将浮点变量用“==”或“！=”与任何数字比较。**
  - 无论是float还是double类型的变量，都有精度限制。
  - 所以一定要避免将浮点变量用“==”或“！=”与数字比较，应该设法转化成“>=”或“<=”形式。
  - 假设浮点变量的名字为x，应当将
    - if (x == 0.0)　　// 隐含错误的比较

    转化为
    - if ((x>=-EPSINON) && (x<=EPSINON))

    其中EPSINON是允许的误差（即精度）。

基本语句

- if 语句
  - **应当将指针变量用“==”或“！=”与NULL比较。**
  - 有时候我们可能会看到 if (NULL == p) 这样古怪的格式。不是程序写错了，是程序员为了防止将 if (p == NULL) 误写成 if (p = NULL)，而有意把p和NULL颠倒。编译器认为 if (p = NULL) 是合法的，但是会指出 if (NULL = p)是错误的，因为NULL不能被赋值。
  - 程序中有时会遇到if/else/return的组合，应该将如下不良风格的程序
    - if (condition) return x; return y;

    改写为if (condition) {return x;} else {return y;}

    或者改写成更加简练的 return (condition ? x : y);

基本语句

- 语句的循环控制变量
  - 不可在for 循环体内修改循环变量，防止for 循环失去控制
  - 建议for语句的循环控制变量的取值采用“半开半闭区间”写法。

```
for (int x=0; x<N; x++)
    {
     …
    }
for (int x=0; x<=N-1; x++)
    {
     …
    }
```

内存管理

- 内存分配方式
  - 从静态存储区域分配。
    - 内存在程序编译的时候就已经分配好，这块内存在程序的整个运行期间都存在。例如全局变量，static变量。
  - 在栈上创建。
    - 在执行函数时，函数内局部变量的存储单元都可以在栈上创建，函数执行结束时这些存储单元自动被释放。
    - 栈内存分配运算内置于处理器的指令集中，效率很高，但是分配的内存容量有限。
  - 从堆上分配，亦称动态内存分配。
    - 程序在运行的时候用malloc或new申请任意多少的内存，程序员自己负责在何时用free或delete释放内存。
    - 动态内存的生存期由我们决定，使用非常灵活，但问题也最多。

内存管理

- 常见的内存错误及其对策
  - 内存分配未成功，却使用了它。
    - 编程新手常犯这种错误，因为他们没有意识到内存分配会不成功。
    - 常用解决办法是，在使用内存之前检查指针是否为NULL。
    - 如果指针p是函数的参数，那么在函数的入口处用assert(p!=NULL)进行检查。
    - 如果是用malloc或new来申请内存，应该用if(p==NULL)或if(p!=NULL)进行防错处理。

内存管理

- 常见的内存错误及其对策
  - 内存分配虽然成功，但是尚未初始化就引用它。
    - 犯这种错误主要有两个起因：一是没有初始化的观念；二是误以为内存的缺省初值全为零，导致引用初值错误（例如数组）。
    - 内存的缺省初值究竟是什么并没有统一的标准，尽管有些时候为零值，我们宁可信其无不可信其有。所以无论用何种方式创建数组，都别忘了赋初值，即便是赋零值也不可省略，不要嫌麻烦。

内存管理

- 常见的内存错误及其对策
  - 内存分配成功并且已经初始化，但操作越过了内存的边界。
    - 例如在使用数组时经常发生下标“多1”或者“少1”的操作。特别是在for循环语句中，循环次数很容易搞错，导致数组操作越界。
  - 忘记了释放内存，造成内存泄露。
    - 含有这种错误的函数每被调用一次就丢失一块内存。刚开始时系统的内存充足，你看不到错误。终有一次程序突然死掉，系统出现提示：内存耗尽。
    - 动态内存的申请与释放必须配对，程序中malloc与free的使用次数一定要相同，否则肯定有错误。

内存管理

- 常见的内存错误及其对策
  - 释放了内存却继续使用它。

有三种情况：

- 程序中的对象调用关系过于复杂，实在难以搞清楚某个对象究竟是否已经释放了内存，此时应该重新设计数据结构，从根本上解决对象管理的混乱局面。
- 函数的return语句写错了，注意不要返回指向“栈内存”的“指针”或者“引用”，因为该内存在函数体结束时被自动销毁。
- 使用free或delete释放了内存后，没有将指针设置为NULL。导致产生“野指针”。

内存管理

- 常见的内存错误及其对策
  - 释放了内存却继续使用它。

对策:

- 用malloc或new申请内存之后，应该立即检查指针值是否为NULL。防止使用指针值为NULL的内存。
- 不要忘记为数组和动态内存赋初值。防止将未被初始化的内存作为右值使用。
- 动态内存的申请与释放必须配对，防止内存泄漏。
- 用free或delete释放了内存之后，立即将指针设置为NULL，防止产生“野指针”。

内存管理

- 常见的内存错误及其对策
  - free和delete对指针操作的正确理解。
    - free和delete只是把指针所指的内存给释放掉，但并没有把指针本身干掉。
    - 指针p被free以后其地址仍然不变（非NULL），只是该地址对应的内存是垃圾，p成了“野指针”。如果此时不把p设置为NULL，会让人误以为p是个合法的指针。
    - 如果程序比较长，我们有时记不住p所指的内存是否已经被释放，在继续使用p之前，通常会用语句if (p != NULL)进行防错处理。很遗憾，此时if语句起不到防错作用，因为即便p不是NULL指针，它也不指向合法的内存块。
    - 指针消亡了并不表示它所指的内存会被自动释放。
    - 内存被释放了并不表示指针会消亡或者成了NULL指针。

内存管理

- 常见的内存错误及其对策
  - 杜绝“野指针”
    - “野指针”不是NULL指针，是指向“垃圾”内存的指针。人们一般不会错用NULL指针，因为用if语句很容易判断。但是“野指针”是很危险的，if语句对它不起作用
  - “野指针”的成因主要有两种：
    - 指针变量没有被初始化。任何指针变量刚被创建时不会自动成为NULL指针，它的缺省值是随机的，它会乱指一气。所以，指针变量在创建的同时应当被初始化，要么将指针设置为NULL，要么让它指向合法的内存。
    - 指针p被free或者delete之后，没有置为NULL，让人误以为p是个合法的指针。

其它编程经验

const更大的魅力是它可以修饰函数的参数、返回值，甚至函数的定义体。被const修饰的东西都受到强制保护，可以预防意外的变动，能提高程序的健壮性。

- 用const修饰函数的参数
  - 如果参数作输出用，不论它是什么数据类型，也不论它采用“指针传递”还是“引用传递”，都不能加const修饰，否则该参数将失去输出功能。 const只能修饰输入参数。
  - 如果输入参数采用“指针传递”，那么加const修饰可以防止意外地改动该指针，起到保护作用。

其它编程经验

- 用const修饰函数的参数（续）
  - 对于非内部数据类型的参数而言，象void Func(A a) 这样声明的函数注定效率比较低。
  - 因为函数体内将产生A类型的临时对象用于复制参数a，而临时对象的构造、复制、析构过程都将消耗时间。
  - 为了提高效率，可以将函数声明改为void Func(A &a)，因为“引用传递”仅借用一下参数的别名而已，不需要产生临时对象。
  - 但是函数void Func(A &a) 存在一个缺点：“引用传递”有可能改变参数a，这是我们不期望的。
  - 解决这个问题很容易，加const修饰即可，因此函数最终成为void Func(const A &a)。

Java编程规则

- 类名首字母应该大写。字段、方法以及对象（句柄）的首字母应小写。
- 对于所有标识符，其中包含的所有单词都应紧靠在一起，而且大写中间单词的首字母。
- Java包全都是小写字母，即便中间的单词亦是如此。
- 对于域名扩展名称，全部都应小写
- 对于自己创建的每一个类，都考虑置入一个main()，其中包含了用于测试那个类的代码。
- 应将方法设计成简要的、功能性单元，用它描述和实现一个不连续的类接口部分
- 使类尽可能短小精悍，而且只解决一个特定的问题。
- 让一切东西都尽可能地“私有”——private。
- 谨惕“巨大对象综合症”。

Java编程规则

- 若不得已进行一些不太雅观的编程，至少应该把那些代码置于一个类的内部。
- 任何时候只要发现类与类之间结合得非常紧密，就需要考虑是否采用内部类，从而改善编码及维护工作
- 尽可能细致地加上注释，并用javadoc注释文档语法生成自己的程序文档。
- 避免使用“魔术数字”，应创建一个常数，并为其使用具有说服力的描述性名称，并在整个程序中都采用常数标识符。这样可使程序更易理解以及更易维护。
- 涉及构建器和异常的时候，通常希望重新丢弃在构建器中捕捉的任何异常。
- 在构建器内部，只进行那些将对象设为正确状态所需的工作。

Java编程规则

- 在一个特定的作用域内，若一个对象必须清除，请采用下述方法：初始化对象；若成功，则立即进入一个含有finally从句的try块，开始清除工作。
- 对象不应只是简单地容纳一些数据；它们的行为也应得到良好的定义。
- 在现成类的基础上创建新类时，请首先选择“新建”或“创作”。
- 用继续及方法覆盖来表示行为间的差异，而用字段表示状态间的区别。
- 为避免编程时碰到麻烦，请保证在自己类路径指到的任何地方，每个名字都仅对应一个类。
- 用合理的设计方案消除“伪功能”。

Java编程规则

- 警惕“分析瘫痪”。请记住，无论如何都要提前了解整个项目的状况，再去考察其中的细节。
- 警惕“过早优化”。
- 请记住，注释、细致的解释以及一些示例往往具有不可估量的价值。
- 更换一下思维角度用完全新鲜的眼光考察你的工作。

C++编程规则

- 课后阅读：

《***C++编程规范 101条规则、准则与最佳实践***》