2022《面向对象程序设计训练》大作业

积极的阴性结果

Negative is a positive result

1.背景

模拟被测者、现场检测信息采集人员、实验室测试结果录入人员联动，完成核酸检测信息系统的 DEMO。

2.基本功能要求

2.1 用户管理。本系统用户包括：被试者，现场检测信息采集人员（以下简称“采集员”），实验室测试结果录入人员（以下简称“录入员”）和管理员。被试者自行注册，用户信息包括：身份证号（18 位）、姓名、密码，其中身份证号在所有用户中不可重复。采集员、录入员因也要接受核酸检测，故也是被试者，只能由管理员在已注册被试者中指定其为采集员、录入员或同时具有两者身份。管理员**不唯一**，也要接受核酸检测，其管理员身份可由系统第一次运行前手工修改用户文件指定。

2.2 用户信息自动读取和保存。在软件开始运行时，自动读取特定文本文件保存的全部用户信息；在软件退出运行时，自动保存全部用户信息到特定文本文件；身份证号、姓名均明文存储，密码自行决定采用明文、或 MD5 密文存储。（MD5 类代码已提供）

2.3 现场检测信息采集。采集员登录系统后，可输入试管编号（生产日期+时间+6 位数字组成的流水号），并为该试管添加 1 个或多个被试信息（已注册被试者的身份证，采样日期时间）。

2.4 实验室测试结果录入。录入员登录系统后，可输入试管编号（生产日期+时间+6 位数字组成的流水号），并为该试管添加 1 个测试结果信息（结果录入日期时间、 阳性/阴性）。

2.5 采样和结果信息自动读取和保存。在软件开始运行时，自动读取特定文本文件保存的采样和结果信息；在软件退出运行时，自动保存全部采样和结果信息到特定文本文件；均为明文存储。

2.6 结果查询。被试者登录系统后，自动列出最近一次的检测结果（采样时间、结果录入时间、阳性/阴性）。

2.7 角色切换。当用户登录时，如同时具有被试者/采集员/录入员/管理员，则提示用户选择期中 1 个角色，并切换到 2.3~2.6 功能。

3.设计与实现要求

3.1 除程序主函数（广义的主函数）、用于运算符重载的友元函数、必要的 lambda 表达式外，不允许出现任何一个非类成员函数。

3.2 任何不改变对象状态（不改写自身对象数据成员值）的成员函数均需显示标注 const。

3.3 全部类分为三大类：界面类（开发环境提供的、与图形/非图形交互界面相关的类， MVC 模式中的 V）、业务流程/控制器类（用于和界面实现耦合，MVC 中的 C）、可重用类（不是仅为本软件独特需求设计，脱离大作业特定要求的、尽可能便利的、在其他应用中被重用的类，MVC 中的 M）。**（此条为强烈建议，未实现界面类和可重用类的解耦将严重影响成绩）**

3.4 仅有界面类可以用开发环境自动生成代码框架。**（图形界面不加分，**

非图形界面不减分）

3.5 仅在业务流程/控制器类可以使用开发环境提供的数据类型和函数。

3.6 可重用类只允许使用 C++11 支持的标准语法、标准算法库、标准模板库。

3.7 不可在自己编写的代码（业务流程/控制器类、可重用类）中调用操作系统 API。

3.8 任何第三方库（非 C++标准提供、非操作系统提供、非开发环境提供）的使用，只能处于源代码级别，不可依赖 lib/so/dylib 文件等（静态库也不可以）和 DLL 文件。全部第三方库/代码，来自网络示例代码及改编代码，均需标注来源和版权信息。

4.代码与发布要求

4.1 通过开发环境自动生成的界面类代码，可做少量注释。

4.2 全部自行编写的代码，均需遵循编码规范要求，见附录。

5.分数构成、比例与考察重点

5.1 基本功能分 10%。以答辩现场测试记录为依据。只考虑功能是否实现、是否鲁棒，不考虑背后的实现机制。

5.2 类设计实现分 60%。以 code review 为依据。MVC 模式运用 15%；全部流程/控制器类的合理性 10%；全部可重用类的合理性（类与成员名的可理解性、属性与行为的从属关系、类间关系、知识点运用覆盖率等）10%、正确性（语法、逻辑、潜在错误等）10%、可重用性（是否从普适性角度进行了抽象和封装，是否可仅仅依

靠头文件阅读进行方便的重用）15%。

5.3 代码规范分 30%。以 code review 为依据，每有 1 处违反代码规范要求，扣 1%，扣完为止。

5.4 特别说明：以知识与技能讲解、发现问题、实践锻炼为目标的小作业和示例代码均已融入大作业，故小作业不在单独计分，大作业得分为本课程最终得分。

6. 作业提交与答辩

6.1 作业提交截止日期为 2022 年 7 月 22 日 23 点 59 分。以网络学堂计时为准，请充分考虑网络拥堵、本机时间与网络学堂时间不一致等一切可能出现的负面因素，尽早完成并提交大作业。

6.2 提交的内容包括：**全部源代码**；用于测试的**用户信息文本文件、采样与结果信息文本文件**；已在本机编译好的可执行文件（如有）、**开发环境版本的说明文件**。以上全部文件，放入“学号”**文件夹**中，对此**文件夹**压缩，**提交压缩包**。

6.3 答辩计划于 2022 年 7 月 24 日开展，初步预计线上线下结合，分组开展。具体答辩分组、时间、顺序在综合考虑大家实际情况和疫情防控政策的基础上，后续发布。

附录:

面向程序设计与训练课程编码规范

V1.0 版本 edited by 范静涛 @ 16/07/2019

1.前言

本编码规范针对 C++语言。制定本规范的目的:

- 适用于课下训练、大作业，督促学生养成良好的编码习惯
- 提高代码的健壮性，使代码更安全、可靠
- 提高代码的可读性，使代码易于查看和维护

本文档分别对 C++程序的格式、注释、标识符命名、语句使用、函数、类运用、程序组织、公共变量等方面做出了要求。规范分为两个级别——规则和建议。规则级的规范要求学生必须要遵守，建议级的规范学生应尽量遵守。

2.编码规范正文

2.1 格式

2.1.1 空行的使用

级别：建议

描述:

- 在头文件和实现文件中，各主要部分之间要用空行隔开。

所谓文件的主要部分，包括：序言性注释、防止被重复包含部分（只在头文件中）、#include 部分、#define 部分、类型声明和定义部分、实现部分等等。

- 在一个函数中，完成不同功能的部分，要用空行隔开。

理由：段落分明，提高代码的可读性。

2.1.2 哪里应该使用空格

级别：规则

描述:

- 在使用赋值运算符、关系运算符、逻辑运算符、位运算符、算术运算符等二元操作符时，在其两边各加一个空格。

  例： nCount = 2; 而不是 nCount=2;
- 三目运算符的“？”和“:”前后均各加一个空格。
- 函数的各参数间、数组初始化列表的各个初始值间，要用“,”和后续一个空格隔开。

  例：void GetDate(int x, int y);

  而不是 void GetDate(int x,int y)或 void GetDate(int x ,int y)
- 控制语句(if , for , while , switch)和之后的“(”之间加一个空格。

- 控制语句(if , for , while , switch)之后的“)”与“{”之间加一个空格（同行的情况下）。
- 控制语句 do 和之后“{”之间加一个空格（同行的情况下）。
- case 的常数表达式之后、default 之后的“：”前面，要有一个空格。

理由：提高代码的可读性。

2.1.3 哪里不应该使用空格

级别：规则

描述：

- 不要在引用操作符前后使用空格，引用操作符指“ .”和“->”，以及“[]”。
- 不要在“: : ”前后使用空格。
- 不要在一元操作符和其操作对象之间使用空格，一元操作符包括“++”、“--”“！”、“&”“*”等。
- “; ”前不能有空格。

理由：提高代码的可读性。

举例:

```
// 不要象下面这样写代码:
m_pFont -> Font;
//应该写成这样
m_pFont->Font;
```

2.1.4 缩进

级别：规则

描述：对程序语句要按其逻辑进行水平缩进，以 4 个空格为一个缩进单位，使同一逻辑层次上的代码在列上对齐。

理由：提高代码的可读性。

2.1.5 长语句的书写格式

级别：规则

描述：较长的语句（长度大于 80 字符，包含缩进）要分成多行书写。长表达式要在低优先级操作符处分新行，操作符放在新行之首，划分出的新行要进行适当的缩进，缩进长度以 4 个空格为单位。

理由：提高代码的可读性。

举例：

```
// 下面是一个处理的较为合理的例子
nCount = Fun1(n1, n2, n3)
        + (nNumber1 * GetDate(n4, n5, n6)) * nNumber1;
```

2.1.6 清晰划分控制语句的语句块

级别：规则

描述：

- 控制语句(if , for , while , do...while，switch)的语句部分一定要用 ‘{’和‘}’括起来(即使只有一条语句)。
- ‘{’与控制语句同行或者，‘{’和单独占一行，与控制语句的首字母应处在同一列上。
- ‘}’单独占一行，但 do…while 结构中，while 前的‘}’不能单独占一行，必须和 while 同行。

理由：这样做，能够划分出清晰的语句块，使语句的归属明确，使代码更加容易阅读和修改。

举例:

```
//不要象下面这样写代码:
if (x == 0)
return;
else
while (x > min)
x--;
// 应该这样写
if (x == 0)
{
    return;
}
else
{
    while (x > min)
    {
        x--;
    }
}
```

2.1.7 一行只写一条语句或标号

级别：规则

规则描述：一行只写一条程序语句 或 标号（仅针对 case）。

理由：提高代码的可读性。

举例:

```
// 不要这样写
x = x0; y = y0;
while (IsOk(x)) {x++;}
// 应该这样写代码
x = x0;
y = y0;
while (IsOk(x))
{
    x++;
}
```

2.1.8 一次只声明、定义一个变量

级别：规则

描述：一次（一条声明、定义语句）只声明、定义一个变量。

理由：提高代码的可读性，方便加入后置注释。

举例:

```
// 应该这样写
int width;
int length;
```

```
// 不要这样写
int width, length;
```

2.1.9 在表达式中使用括号

级别：建议

描述：对于一个表达式，在一个二元、三元操作符操作的操作数的两边，应该放置“(”和“)”，直到最高运算逻辑。

理由：避免出现不明确的运算、赋值顺序，提高代码的可读性。

举例:

```
// 下面这行代码:
result = fact / 100 * number + rem;
//最好写成这样
result = ((fact / 100) * number) + rem;
```

2.1.10 将操作符“*”(Dereferencing)、“&”和类型写在一起

级别：规则

描述：在定义指针变量时，将操作符“*”、“&”和类型写在一起。

理由：统一格式，提高代码的可读性。

举例:

```
// 不要象下面这样写代码:
char    *s;
//而应该写成这样
char*    s;
```

2.2 注释

这一部分对程序注释提出了要求。

程序中的注释是程序与日后的程序读者之间通信的重要手段。良好的注释能够帮助读者理解程序，为后续阶段进行测试和维护提供明确的指导。

下面是关于注释的基本原则:

(1)注释内容要清晰明了，含义准确，防止出现二义性。

(2)边写代码边注释，修改代码的同时修改相应的注释，保证代码与注释的一致性。

2.2.1 对函数进行注释

级别：规则

描述:

●在函数的声明之前，要给出精练的注释（不必牵扯太多的内部细节），让使用者能够快速获得足够的信息使用函数。格式不做具体要求。

●在函数的定义之前，要给出足够的注释。注释格式要求如下:

```
/*********************************************************************
【函数名称】          （必需）
```

【函数功能】　　（必需）
【参数】　　（必需。标明各参数是输入参数还是输出参数。）
【返回值】　　（必需。解释返回值的意义。）
【开发者及日期】　　（必需）
【更改记录】　　（若有修改，则必需注明）
**********/
理由：提高代码的可读性。

2.2.2 对类进行注释

规范级别：规则
描述：在类的声明之前，要给出足够而精练的注释。注释格式要求如下：
/**********
【类名】　　（必需）
【功能】　　（必需）
【接口说明】　　（必需）
【开发者及日期】　　（必需）
【更改记录】　　（若修改过则必需注明）
**********/
理由：提高代码的可读性。

2.2.3 对文件进行注释

级别：规则
描述：
在头文件、实现文件的首部，一定要有文件注释，用来介绍文件内容。注释格式要求如下：
/**********
【文件名】　　（必需）
【功能模块和目的】　　（必需）
【开发者及日期】　　（必需）
【更改记录】　　（若修改过则必需注明）
**********/
理由：提高代码的可读性。

2.2.4 对每个空循环体要给出确认性注释

级别：建议
描述：建议对每个空循环体给出确认性注释。
理由：提示自己和别人，这是空循环体，并不是忘了。
举例：

```
while (g_bOpen == 1)
{
    //空循环
}
```

2.2.5 对多个 case 语句共用一个出口的情况给出确认性注释

级别：建议

描述：建议对多个 case 语句共用一个出口的情况给出确认性注释。
理由：提示自己和别人，这几个 case 语句确实是共用一个出口，并不是遗漏了。
举例：

```
switch (nNumber)
{
    case 1:
        nCount++;
        break;
    case 2:
    case 3:
        nCount--;
        break;          // 当 nNumber 等于 2 或 3 时，进行同样的处理
    default:
        break;
 }
```

2.2.6 其它应该考虑进行注释的地方

级别：建议
描述：除上面说到的，对于以下情况，也应该考虑进行注释：

- 变量的声明、定义。通过注释，解释变量的意义、存取关系等；
  例如：

```
int m_iNumber;  //记录图形个数。被 SetDate( )、GetDate( )使用。
```

- 数据结构的声明。通过注释，解释数据结构的意义、用途等；
  例如：

```
//定义结构体，存储元件的端点。用于将新旧的端点对应。
typedef struct
{
    short int nBNN;
    short int nENN;
    short int nBNO;
    short int nENO;
} Element;
```

- 分支。通过注释，解释不同分支的意义；
  例如：

```
if (m_iShortRadio == 0)          //三相的情况
{
    strvC.Format("%-10.6f", vC);
    straC.Format("%-10.6f", aC);
}
else if (m_iShortRadio == 1)     //两相的情况
{
    strvC = _T("");
    straC = _T("");
}
```

- 调用函数。通过注释，解释调用该函数所要完成的功能；
  例如：
  ```
  SetDate(m_nNumber );  //设置当前的图形个数。
  ```
- 赋值。通过注释，说明赋值的意义；
  例如：
  ```
  m_bDraw = 1;  //将当前设置为绘图状态
  ```
- 程序块的结束处。通过注释，标识程序块的结束。
  例如：
  ```
  if (name == White)
  {
      ...
      if (age = = 20)
      {
          ...
      }//年龄判断、处理结束
      ...
  }//姓名判断、处理结束
  ```
- 其它有必要加以注释的地方

理由：提高代码的可读性。

2.2.7 行末注释尽量对齐

级别：建议

描述：同一个函数或模块中的行末注释应尽量对齐。

理由：提高代码的可读性。

举例:

```
nCount = 0;                  //计数器，表示正在处理第几个数据块
BOOL bNeedSave;              //是否保存从服务器返回的数据
DWORD BytesWritten;          //写入的数据长度
```

2.2.8 注释量

级别：规则

描述：注释行的数量不得少于程序行数量的 1/3。

2.3 命名

对标识符和文件的命名要求。

2.3.1 标识符命名要求

级别：规则

描述：在程序中声明、定义的变量、常量、宏、类型、函数，在对其命名时应该遵守统一的命名规范。具体要求如下：

- 变量。变量名=作用域前缀+类型前缀+物理意义。物理意义部分应当由至少一个英文描

述单词组成，各英文描述单词的首字母分别大写，其他字母一律小写。对于不同作用域的变量，其命名要求如表 2-1 所示；对于不同数据类型变量，其命名要求如表 2-2 所示：

表 2-1 作用域前缀

变量种类	作用域前缀要求	示例
全局变量（在整个程序中可以使用）	g_	g_iNumber 全局整型变量
全局指针变量	g_p	g_pNumber
对象级变量（类内数据成员）、文件作用域变量（文件中静态变量。只在某个.c 文件中可以使用。但如整个程序只有一个.c 文件，应当认为是全局变量）	m_	m_cClassCode 文件作用域整型变量
对象级指针变量、文件作用域指针变量	m_p	m_pNumber
局部变量	无	fPrice 局部单精度浮点型变量
静态局部变量	s_	s_Number

表 2-2 类型前缀

数据类型	类型前缀	示例
char	c（优先级第 3）	m_cClassCode 文件作用域整型变量
int	i（优先级第 3）	g_iNumber 全局整型变量
short int	n（优先级第 3）	m_nCount 文件作用域短整型变量
long int	l（优先级第 3）	lCount 局部长整型变量
long long int	ll（优先级第 3）	llBigCount 局部长长整型变量
用 unsigned 修饰	u（优先级第 2）但当仅为 unsigned int 时，用 u 替换 i	g_ulCount 全局无符号长整型变量
float	f（优先级第 3）	fPrice 局部浮点型变量
double	r（优先级第 3）	rPrice 局部浮点型变量
指针	p（优先级第 1）	g_pulPrice 全局指向无符号长整型的指针变量

- 常量

  常量的名字要全部大写，包括至少一个英文单词。常量指：

const 修饰的量。如 const int NUMBER = 100;

枚举量。如 enum Number{ ONE，TWO，THREE };

- 宏

  所有用宏形式定义的名字，包括宏常量和宏函数，名字要全部大写。

- 自定义类型类型

  自定义类型名应以大写字母打头。C++中自定义类型包括：class、struct、enum、union、typedef 声明的类型、namespace。

  例如：typedef struct Student;

  class CMsgDialog;函数

  函数名应以大写字母打头，由动词性英文单词或动宾型英文短语构成。

  例如：void GetCount();

- 下面还有一些在命名时应该遵守的基本规范：
  - 名中含多于一个单词时，每个单词的第一个字母大写。

    例如：m_LastCount 中要大写 L 和 C;
  - 不要使用以下划线“_”打头的标识符。

    例如：_bFind 是不允许出现的变量;
  - 不要使用仅用大小写字母区分的名称。

    例如：m_Find 和 M_FIND;
  - 尽量使用有意义的名字。应做到见其名知其意。

  例如：m_uErrorCode 表示错误的代码;

理由：减少命名冲突；提高代码的可读性。

2.3.2 标识符长度要求

级别：规则

描述：在程序中声明、定义的变量、常量、宏、类型、函数，它们的名字长度要在 4 至 25 个字符之内（下限不包括前缀，上限包括名字中所有的字符）。对于某些已经被普遍认同的简单命名，可不受本规则的限制。如 for 循环的循环记数变量，可使用 i 、j 、x、y 等简单字符命名。如名字过长，可使用缩写，缩写时应当尽可能保留影响发音的辅音字母，例如 Index 可缩写为 Idx，Button 可缩写为 Btn，Solution 可缩写为 Sln。

理由：名字长度应该在一个恰当的范围内，名字太长不够简洁，名字太短又不能清晰表达含义。

2.3.3 文件命名要求

级别：建议

描述：代码文件的名字要与文件中声明、定义的重要重要函数名字或整体功能描述基本保持一致，使功能与类文件名建立联系。如 math.h 包括的都是和数学运算相关的函数声明。

举例:

将类 CMsgDialog 的头文件和实现文件命名为 msgdialog.h 和 msgdialog.cpp 就是一种比较简单、恰当的方法。

理由：使应用程序容易理解。

2.4 语句

对具体程序语句的使用要求。

2.4.1 一条程序语句中只包含一个赋值操作符

级别：规则

描述：在一条程序语句中，只应包含一个赋值操作符。赋值操作符包括：=, +=, -=, *=, /=, %=, >>=, <<=, &=, |=,^=, ++, --。

理由：避免产生不明确的赋值顺序。

举例:

```
// 不要这样写
b = c = 5;
a = (b * c) + d++;
// 应该这样写
c = 5;
b = c;
a = (b * c) + d;
d++;
```

2.4.2 不要在控制语句的条件表达式中使用赋值操作符

级别：建议

描述：不要在控制语句 if, while, for 和 switch 的条件表达式中使用赋值操作符。赋值操作符包括：=, +=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=,++,--。

理由：一个类似于 if (x = y)这样的写法是不明确、不清晰的，代码的作者也许是想写成这样：if (x == y)。

举例:

```
//不要象下面这样写代码:
if (x -= dx)
{
        ...
}
//应该这样写:
x -= dx;
if (x)
{
      ...
}
```

2.4.3 赋值表达式中的规定

级别：建议

描述： 在一个赋值表达式中：

- 一个左值，在表达式中应该仅被赋值一次。
- 对于多重赋值表达式，一个左值在表达式中仅应出现一次，不要重复出现。

理由：避免产生不明确的赋值顺序。
举例:

```
//不要像下面这样写代码:
i = t[i++]; //一个左值，在表达式中应该仅被赋值一次
a = b = c + a;      //对于多重赋值表达式，一个左值在表达式中仅应出现一次，不能重复出现。
i = t[i] = 15;      //对于多重赋值表达式，一个左值在表达式中仅应出现一次，不能重复出现。
```

2.4.4 禁用 Goto 语句

级别：规则
描述：程序中不要使用 goto 语句。
理由：这条规则的目的是为了确保程序的结构化，因为滥用 goto 语句会使程序流程无规则，可读性差。Goto 语句只在一种情况下有使用价值，就是当要从多重循环深处跳转到循环之外时，效率很高，但对于一般要求的软件，没有必要费劲心思追求多么高的效率，而且效率主要是取决于算法，而不在于个别的语句技巧。

2.4.5 避免对浮点数值类型做精确比较

级别：规则
描述：不要对浮点类型的数据做等于、不等于这些精确的比较判断，要用范围比较代替精确比较。
理由：由于存在舍入的问题，计算机内部不能精确的表示所有的十进制浮点数，用等于、不等于这种精确的比较方法就可能得出与预期相反的结果。所以应该用大于、小于等范围比较的方法代替精确比较的方法。
举例:

```
//不要象下面这样写代码:
float number;
...
if (number = = 0)                    //精确比较
```

2.4.6 对 switch 语句中每个分支结尾的要求

级别：规则
描述：switch 语句中的每一个 case 分支，都要以 break 作为分支的结尾（几个连续的空 case 语句允许共用一个）。
理由：使代码更容易理解；减少代码发生错误的可能性。

2.4.7 switch 语句中的 default 分支

级别：规则
描述：在 switch 语句块中，一定要有 default 分支来处理其它情况。仅在 switch 中所有 case 已经包含了被判定表达式全部取值范围时候，可以不受本规则限制。
理由：用来处理 switch 语句中默认、特殊的情况。

2.4.8 对指针的初始化

级别：规则

描述：在定义指针变量的同时，对其进行初始化。如果定义时还不能为指针变量赋予有效值，则使其指向 NULL。

理由：减少使用未初始化指针变量的几率。

举例:

```
// 不要这样写代码
int* y ;
y = &x ;
// 应该这样写
int* y = &x;
```

2.4.9 释放内存后的指针变量

级别：规则

描述：当指针变量所指的内存被释放后，应该赋予指针一个合理的值。除非该指针变量本身将要消失这种情况下不必赋值，否则应赋予 NULL。

理由：保证指针变量在其生命周期的全过程都指向一个合理的值。

2.4.10 使用正规格式的布尔表达式

规范级别：建议

规则描述：对于 if, while, for 等控制语句的条件表达式，建议使用正规的布尔格式。

理由：使代码更容易理解。

举例:

```
//不要象下面这样写代码:
while(1)
{
    ...
}
if(test)
{
    ...
}
for(i = 1; function_call(i); i++)
{
    ...
}

//最好这样写:
AlwaysTrue = true;
while(AlwaysTrue == true)
{
    ...
}
if(test == true)
{
```

```
        ...
    }
    for(i = 1; function_call(i) == true; i++)
    {
        ...
    }
```

2.4.11 new 和 delete

规范级别：规则

规则描述:局部的 new 和 delete 要成对出现；new 要与 delete 对应，new[]要与 delete[]对应。

理由：防止内存泄露。

2.5 函数

对函数的要求。

2.5.1 明确函数功能

级别：规则

描述：函数体代码长度不得超过 100 行（不包括注释）。

理由：明确函数功能（一个函数仅完成一件事情），精确（而不是近似）地实现函数设计。

2.5.2 将重复使用的代码编写成函数

级别：建议

描述：将重复使用的简单操作编写成函数。

理由：对于重复使用的功能，虽然很简单，也应以函数的形式来处理，这样可以简化代码，使代码更易于维护。

2.5.3 函数声明和定义的格式要求

级别：规则

描述：在声明和定义函数时，在函数参数列表中为各参数指定类型和名称。

理由：提高代码的可读性，改善可移植性。

举例:

```
// 不要象下面这样写代码:
f(int, char*);              //函数声明
.....
f(int a, char* b)        //函数定义
{
    ...
}
// 应该这样写:
f(int a, char* b);      //函数声明
```

```
……
f(int a, char* b)       //函数定义
{
        …
}
```

2.5.4 为函数指定返回值

级别：规则

描述：要为每一个函数指定它的返回值。如果函数没有返回值，则要定义返回类型为 void。

理由：提高代码的可读性；改善代码的可移植性。

2.5.5 在函数调用语句中不要使用赋值操作符

级别：建议

描述：函数调用语句中，在函数的参数列表中不要使用赋值操作符。赋值操作符包括=, +=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=,++,--。

理由：避免产生不明确的赋值顺序。

举例:

```
// 不要象下面这样写代码:
void fun1(int a);
void fun2(int b)
{
        fun1(++b);     //注意这里!
}
```

2.6 程序组织

对程序组织的要求。

2.6.1 一个头文件中只声明一个函数、一类函数或一个类

级别：规则

描述：在一个头文件中，只应该包含对一个函数的声明或一类函数的声明，使用类时则只包含一个类的声明。当头文件中包含一类函数时，这些函数功能必须可以抽象为一个共同的单词或短语。头文件是指以.h 为后缀的文件。

理由：提高代码的可读性和文件级别重用的可能性。

2.6.2 一个源文件中只实现一个函数、一类函数或一个类

级别：规则

描述:在一个源文件中，只应该包含对一个函数的定义或一类函数的定义，使用类时则只包含一个类的定义。当源文件中包含一类函数时，这些函数功能必须可以抽象为一个共同的单词或短语。源文件指以.c 为后缀的代码文件。

理由：提高代码的可读性和文件级别重用的可能性。

2.6.3 头文件中只包含声明，不应包含定义

级别：规则

描述：在头文件中只包含声明，不要包含全局变量和函数的定义。但宏和 const 要分情况讨论，不一定受本规则限制。Inline 可以不受本规则限制。

理由：在头文件中只应该包含各种声明，而不应该包含具体的实现。

2.6.4 源文件中不要有函数的声明

级别：规则

描述：在源文件中只应该包含对全局变量、文件作用域变量、和函数的定义，不应该包含任何声明。声明应该统一放到头文件中去。但宏和 const 要分情况讨论，不一定受本规则限制。

理由：内外有别，限制细节知悉范围，提高代码的可读性和可靠性。

2.6.5 可被包含的文件

级别：规则

描述：只允许头文件被包含到其它的代码文件中去。

理由：改善程序代码的组织结构。

2.6.6 避免头文件的重复包含

级别：规则

描述：头文件的格式应该类似于：

```
#ifndef <IDENT>
#define <IDENT>
...
#endif
```

或者

```
#if !defined (<IDENT>)
#define <IDENT>
...
#endif
```

上面的<IDENT>是一个标识字符串，要求该标识字符串必须唯一。建议使用该文件的大写文件名。

理由：避免对同一头文件的重复包含。

举例:

```
// 对于文件 audit.h，它的文件结构应该为：
#ifndef AUDIT_H       //第一行
#define AUDIT_H       //第二行
...
#endif                //最后一行
```

2.7 公共变量

对公共变量（全局变量）的要求。

2.7.1 严格限制公共变量的使用

级别：建议

描述：在程序中要尽可能少的使用公共变量。在决定使用一个公共变量时，要仔细考虑，权衡得失。

理由：公共变量会增大模块间的耦合，甚至扩大错误传播范围。

2.7.2 明确公共变量的定义

级别：规则

描述：当你真的决定使用公共变量时，要仔细定义并明确公共变量的含义、作用、取值范围、与其它变量间的关系。明确公共变量与操作此公共变量的函数之间的关系，如访问、修改和创建等。

2.7.3 防止公共变量与局部变量重名

级别：规则

描述：防止公共变量与局部变量重名。

2.8 类

对类的要求。

2.8.1 关于默认构造函数

规范级别：规则

规则描述：为每一个类显示定义默认构造函数。

理由：确保类的编写者考虑在类对象初始化时，可能出现的各种情况。

举例:

```
class CMyClass
{
    CMyClass();
    ...
};
```

2.8.2 关于拷贝构造函数

规范级别：规则

规则描述：当类中包含指针类型的数据成员时，必须显示的定义拷贝构造函数。建议为每个类都显示定义拷贝构造函数。

理由：确保类的编写者考虑类对象在被拷贝时可能出现的各种情况。

举例:

```
class CMyClass
{
    ...
    CMyClass(CMyClass& object);
    ...
```

```
};
```

2.8.3 为类重载“=”操作符

规范级别：规则

规则描述：当类中包含指针类型的数据成员时，必须显示重载“=”操作符。建议为每个类都显示重载“=”操作符。

理由：确保类的编写者考虑将一个该类对象赋值给另一个该类的对象时，可能出现的各种情况。

举例:

```
// 应该这样写代码
class CMyClass
{
    ...
    operator = (const CMyClass& object);
     ...
 };
```

2.8.4 关于析构函数

规范级别：规则

规则描述：为每一个类显示的定义析构函数。

理由：确保类的编写者考虑类对象在析构时，可能出现的各种情况。

举例:

```
class CMyClass
{
    ...
    ~CMyClass ();
    ...
};
```

2.8.5 虚拟析构函数

该规则参考自《Effective C++》中的条款 14。

规范级别：规则

规则描述：基类的析构函数一定要为虚拟函数（virtual Destructor）。

理由：保证类对象内存被释放之前，基类和派生类的析构函数都被调用。

2.8.6 不要重新定义继承来的非虚函数

规范级别：规则

规则描述：在派生类中不要对基类中的非虚函数重新进行定义。如果确实需要在派生类中对该函数进行不同的定义，那么应该在基类中将该函数声明为虚函数；

理由不要忘了，当通过一个指向对象的指针调用成员函数时，最终调用哪个函数取决于指针本身的类型，而不是指针当前所指向的对象。

2.8.7 如果重载了操作符"new"，也应该重载操作符 "delete"

该规则参考自《Effective C++》中的条款 10。

规范级别：规则

规则描述：如果你为一个类重载了操作符 new，那你也应该为这个类重载操作符 delete。

理由：操作符 new 和操作符 delete 需要一起合作。

2.8.9 类数据成员的访问控制

规范级别：规则

规则描述：类对外的接口应该是完全功能化的，类中可以定义 Public 的成员函数，但不应该有 Public 的数据成员。

理由：要想改变对象的当前状态，应该通过它的成员函数来实现，而不应该通过直接设置它的数据成员这种方法。一个类的数据成员应该声明为 private 的，最起码也应该是 protected 的。

2.6.10 限制类继承的层数

规范级别：建议

规则描述：当继承的层数超过 5 层时，问题就很严重了，需要有特别的理由和解释。

理由：

●很深的继承通常意味着未做通盘的考虑；

●会显著降低效率；

●可以尝试用类的组合代替过多的继承；

●与此类似，同层类的个数也不能太多，否则应该考虑是否要增加一个父类，以便做某种程度上的新的抽象，从而减少同层类的个数。

2.6.11 慎用/最好不用多继承

规范级别：建议

规则描述：C++提供多继承的机制。多继承在描述某些事物时可能是非常有利的，甚至是必须的，但我们在使用多继承的时，一定要慎重，在决定使用多继承时，确实要有非常充分的理由。

理由：多继承会显著增加代码的复杂性，还会带来潜在的混淆。比如在很多 C++书籍中提到的菱形继承问题

2.6.12 考虑类的复用

规范级别：建议

规则描述：类设计的同时，考虑类的可复用性。

2.9 其它

下面这几条要求，不适合合并到上面任何一类，所以单独作为一部分。

2.9.1 用常量代替无参数的宏

级别：规则

描述：使用 const 来定义常量，代替通过宏来定义常量的方法。
理由：在不损失效率的同时，使用 const 常量比宏更加安全。
举例:

```
//宏定义的方法
#define string "Hello world!"
#define value   3
//常量定义的方法可以代替宏，且要更好
const char* string = "Hello world!";
const int value = 3;
```

2.9.2 用内联代替有参数的宏

级别：规则
描述：使用 inline 关键字声明函数为内联函数，代替有参数的宏。
理由：保证效率和安全，同时提高代码的可读性。

2.9.3 尽量使用 C++风格的类型转换

该规则参考自《More Effective C++》中的条款 2。
规范级别：建议
规则描述：用 C++提供的类型转换操作符（static_cast，const_cast， dynamic_cast 和 reinterpret_cast）代替 C 风格的类型转换符。
理由：C 风格的类型转换符有两个缺点：
1 允许你在任何类型之间进行转换，即使在这些类型之间存在着巨大的不同。
2 在程序语句中难以识别。

2.9.4 将不再使用的代码删掉

级别：规则
描述：将程序中不再用到的、注释掉的代码及时清除掉。
理由：理由不用做太多的解释了吧？没有用的东西就应该清理掉。如果觉得这些代码你可能以后会用到，可以备份到其它地方，而不要留在正式的版本里。

3 并不会结束

以上就是我们目前要求 C++程序遵守的规范的全部内容。欢迎大家讨论、补充和修订。