2025《面向对象程序设计训练》大作业

构建人工神经网络

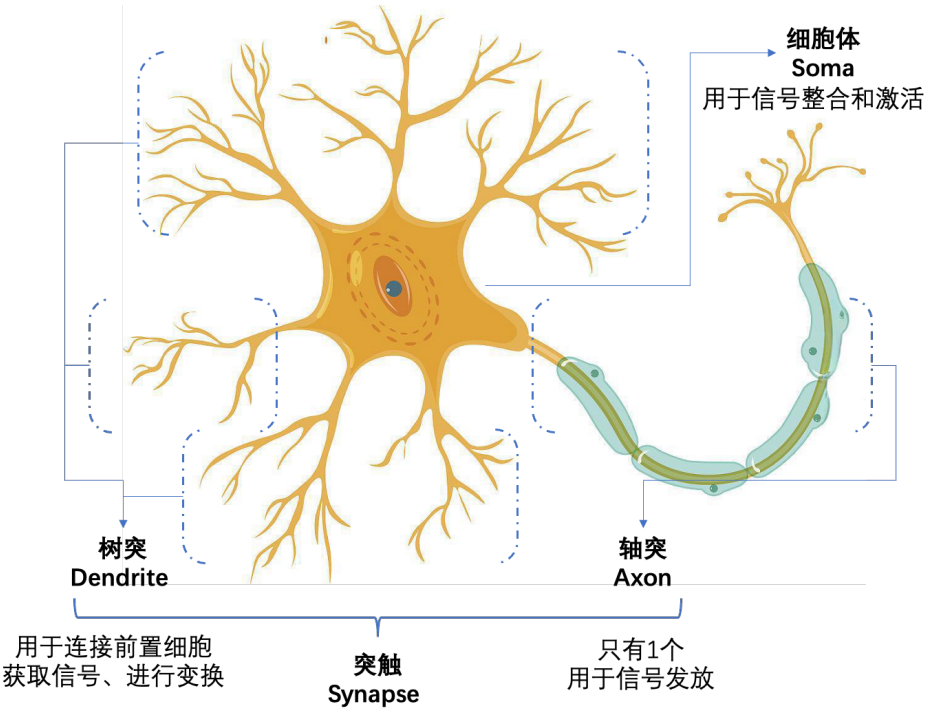
Construct Artificial Neural Network

1.背景

人工神经网络 （Artificial Neural Network，ANN）是 人工智能（Artificial Intelligence，AI）领域的关键技术之一，属于机器学习的分支方法。其核心是通过模拟人脑神经元结构处理数据，实现智能决策和预测。本次大作业将以面向对象程序设计思想为引导，通过 ANN 示意性 DEMO 的编写，强化课程知识和技能的运用能力。在本次大作业中按如下形式，简要模拟生物神经网络定义 ANN 架构：

1.1 神经元(Neuro)

神经元负责对神经信号(数学意义为数值)进行传导和处理，本节描述了神经元的主要结构和功能组成，参考下图进行分述。



1.1.1 突触(Synapse)

突触用于实现神经元细胞间的神经信号线性缩放和单向传导，设$Input$为输入信号，$Weight$为 Synapse 对信号的缩放权重，$Signal$为输出信号，突触功能的数学描述为$Signal = Input \times Weight$。突触可分为树突(Dendrite)和轴突(Axon)。树突的$Weight$可为任何实数值，而轴突的$Weight$恒为 1。每个神经元可以有不少于 1 个树突用于接受其他神经元的信号作为输入信号。每个神经元只有 1 个轴突用于向其他多个神经元输出信号。虽然只有 1 个轴突，但轴突尾部的树状末梢结构可以连接多个后继神经元的树突，也就是说对多个后继神经元的信号发放是相同信号。

1.1.2 细胞体（Soma）

细胞体用于实现信息处理的整合函数(Sum Function)和激活函数（Activation Function）功能。整合函数将所有树突传递给细胞体的信号与细胞体自身具备的偏置(Bias)进行累加，功能的数学描述$SumFunction = bias + \sum_{i=1}^{DendriteCount} Signal_i$。激活函数将累加函数的结果作为输入，进行非线性变换，功能可描述为$ActivationFunction = f(SumFunction)$，典型激活函数形式如下：

Sigmod(S 型函数)　　：　　$f(x) = {}^{1}/_{(1+e^{-x})}$

Tanh(双曲线正切函数)：　　$f(x) = (e^{x} - e^{-x})/(e^{x} + e^{-x})$

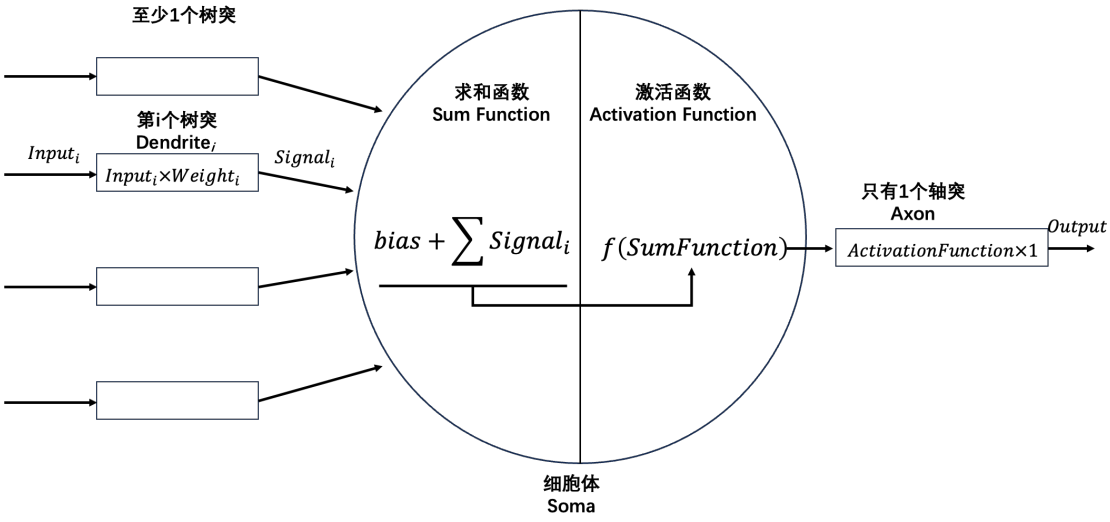
ReLU(线性修正单元)　：　　$f(x) = \max(0, x)$

特别注意，细胞体可以只有整合函数而无激活函数。此时，可以认为激活函数是一个线性函数$f(x) = x$。

1.1.3 神经元数据处理流程

前驱神经元的输出，进过本神经元的树突、细胞体、轴突进行变换和输出。

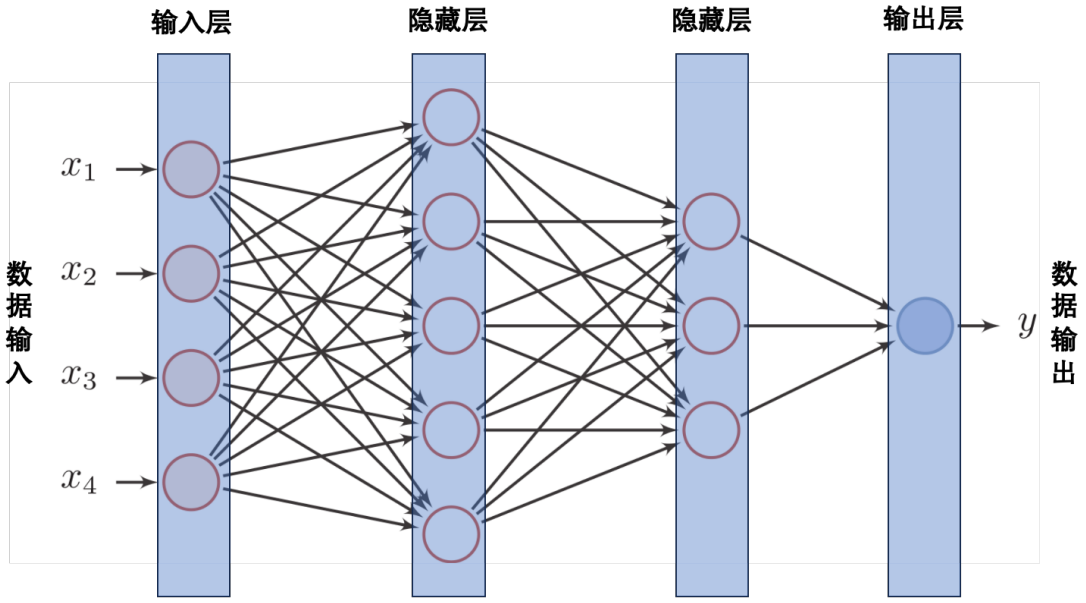
顺序	环节	输入	输出
1	第i个树突	$Input_i$ (前驱神经元轴突的 Output)	$Signal_i = Input_i \times Weight_i$
2	整合函数	$Signal_{1...n}$	$SumFunction = bias + \sum Signal_i$
3	激活函数	$SumFunction$结果	$ActivationFunction = f(SumFunction)$
4	轴突	$ActivationFunction$结果	$Output = ActivationFunction \times 1$



1.2 层（Layer）

层是一种神经元的组织结构，接收前一层神经元的输出或系统的输入数据，通过层内每个神经元的计算，向后一层进行传递。层内神经元无突触连接，相邻层间神经元有突触连接。信息总是从前一层传

向后一层，称为前向传播（Forward-Propagation）。因为层内神经元并无计算依赖关系，故层内神经元计算先后顺序不影响层的输出结果。同时，层也是神经元批量化管理工具，如添加层的行为是批量添加神经元等。层的存在，还因为同层神经元大多数情况下要共同完成一个任务（如卷积、池化、非线性、全连接等），神经元彼此之间具有类似功能（如全连接层的神经元都没有激活函数，卷积层的神经元数量与前层连接的关系趋同）。



1.3 网络（Network）

网络是由若干层神经元构成的功能整体。首层神经元均只有 1 个树突，用于接受外界信息输入；末层神经元轴突用于向外界输出结果。将网络功能看做一个黑盒，信息从入到出的过程，称之为推理（Inference），而推理的过程是依靠层的依次前向传播实现的。

特别强调：面向对象！面向对象!!面向对象!!!

无论写什么，先考虑 OOP 思想、原则！

先问是不是，再问为什么

先想该不该，再想好不好，最后想怎么做

优雅的设计 > 功能的实现

修修补补的架构不可能产生优雅的实现

优雅的架构，怎么写也不会太差

2.软件基本功能要求

2.1 人工神经网络模型导入。根据给定文件名，将以特定格式编码存储的人工神经网络模型文件，导入到内存，转化为可编辑、可执行推理的类对象。人工神经网络模型文件格式有很多种，要求至少支持 ANN 格式（示例模型文件为 Simple.ANN；ANN 文件格式说明见附录 1）。说明：要求“至少支持 ANN 格式”的目的（支持导入多种格式的功能，不加分），是促进类架构设计优化（不同开发环境下的神经网络模型描述有多种格式，如 ONNX 等），从可能支持多种模型格式的角度，需要考虑导入器(Importer)类的继承体系，而非单独写一个本次大作业专用的 ANN 导入器。

2.2 人工神经网络模型导出。将内存中的人工神经网络模型对象，根据给定文件名，以特定格式编码写入人工神经网络模型文件。人工神经网络模型文件格式有很多种，要求至少支持 ANN 格式（示例模型文件为 Simple.ANN；ANN 文件格式说明见附录 1）。说明：要求“至少支持 ANN 格式”的目的（导出多种格式的功能，不加分），

是促进类架构设计优化（不同开发环境下的神经网络模型描述有多种格式，如 ONNX 等），从可能支持多种模型格式角度出发，则需要考虑导出器(Exporter)类的继承体系，而非单独写一个本次大作业专用的 ANN 导出器。

2.3 人工神经网络模型修改。

2.3.1 列出所有 Layer 对象，显示其序号和内部神经元数量、序号。显示格式自拟。

2.3.2 删除指定 Layer 对象。同时删除此 Layer 内部神经元及其关联的层间突触连接；

2.3.3 添加一个 Layer 对象。添加后内部没有 Neuron 对象。

2.3.4 列出指定 Layer 对象中的所有 Neuron 对象，显示其序号和偏置。显示格式自拟。

2.3.5 修改指定 Neuron 对象包含的偏置。

2.3.6 列出指定 Neuron 对象的突触连接关系信息。显示格式自拟。

2.3.7 删除指定 Neuron 对象。关联的 Synapse 也应一同删除，原连接的 Neuron 对象也应更新。

2.3.8 连接指定的 Neuron 对象。必须是相邻 Layer 间、未有 Synapse 连接的 Neuron 对象才可进行连接。

2.4 显示统计信息：人工神经网络模型中 Layer 总数，Neuron 总数和总 Synapse 总数。显示格式自拟。

2.5 人工神经网络模型验证与推理。

2.5.1 验证 Network 合理性 IsValid。数据可从首层输入、经各个层

到达末层输出，且每个神经元均参与了数据传递过程，且无环状连接，则为合理，否则为不合理。

2.5.2 执行推理。在 Network 具有可行性的前提下，给定与首层神经元数量规模相同的数据输入，通过逐层前向传播，得到与末层神经元数量规模相同的数据输出。

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 不可使用非类成员的全局变量（文件作用域或全局作用域）。

3.3 不可将任何受访问规则限制的数据成员作为公有成员。

3.4 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。

3.5 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，可以认为是没有界面的整个软件功能集合，MVC 中的 C）、**可重用类（包括但不限于支撑本软件底层功能实现的类，特别重要之处在于：可重用类不是仅为本软件独特需求设计，而是可以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）**

举例：Neuron、Layer、Network、ActivationFunction 等都可脱离大作业具体功能要求，在人工智能领域独立大量运用，其类体

系和封装设计不能仅仅考虑做大作业“刚好够用”。

3.6 界面类必须以类的形式存在，不可直接写为 main 函数，或其他不属于类非静态成员的函数。

3.7 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）**

3.8 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。比如，用 QT 做图形化界面，控制器类可接收来自界面的 QT 独有的 QString 类型输入，而 Model3D 的 Name 属性则不可使用 QString 类型。

3.9 业务流程/控制器类，特别限定：在不被修改代码的前提下，任何情况都只能产生 1 个对象实例（设计时需使用单例模式）。

3.10 可重用类只允许使用 C++11 及之后标准支持的标准语法、标准模板库。不依赖于具体编辑器特性，可以跨开发环境、编译环境、操作系统编译运行。

3.11 不可在自己编写的代码（一定包含业务流程/控制器类、可重用类，可能包含界面类）中直接调用操作系统 API。

3.12 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，且需要根据编码规范调整。

3.13 大作业是 OOP 的大作业不是 C++的大作业，需要综合运用课

上知识和技能，侧重于以优雅的设计，实现多个、可被更多场合、更应用便捷使用的类。一切以“我为人人”为出发点，而不是为了实现大作业功能“刚好够用”。

4.代码注释要求

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 2：编码规范 V1.3）。

5.分数构成、比例与考察重点

5.1 基本功能分 10%。只考虑功能是否实现、是否鲁棒，不考虑背后的实现机制。以编译运行、ANN 文件导入导出、测试用例的手工编辑效果评价。

5.2 类设计实现分 60%。以 code review 为依据。MVC 模式运用 15%；流程/控制器类的合理性 10%；可重用类的合理性（这些类是否应该存在、类与成员名的可理解性、属性与行为是否符合课上讲的提取准则、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，是否对超出完成大作业最小类集合的、可以更好支撑其他应用的类进行了抽象和封装，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：大作业得分为本课程最终得分的 80%；因参加比赛获奖的加分合并后不超过满分。

6. 作业提交

6.1 作业提交截止时间**初定为 2025 年 8 月 1 日 23 点 59 分**（在与教务部门确认教师提交成绩的 DDL 后，可能会推迟作业提交时间，注意：是可能，不是一定。今年选课 240 人整，作业评阅压力骤增，希望有充足时间认真对待每位同学辛勤付出的成果）。**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。**（每年都有各种高估自己 coding、debug 耗时、高估网络带宽、高估其他各种因素，导致无法按时提交大作业，而第二年重新上课的同学，请一定力求靠谱）**

4.3 提交 1 个压缩包。压缩包内最顶层为**以学号命名的文件夹**，里面存有：（1）所有 CPP 和 HPP 文件，不需要编译后的可执行文件，不需要工程文件，不需要编译过程中产生的一系列中间文件；（2）开发环境版本的说明 txt 文件：操作系统版本(如 windows11 专业版-64bit)、编辑器版本（如 vsCode 11.3）、编译器版本（如 GCC14.1 x86-64）、C++标准版本（如 C++23）。请保证提交的所有文件都是必要的，不要放入整个 debug 文件夹。

附录 1:

ANN 文件格式说明

1、 概述

ANN（或 .ANN）是本次大作业定义的一种人工神经网络架构和参数描述文件格式。

2、 文件结构

ANN 文件为多行 ASCII 码文件，仅包含英文字符。每行表示下列之一的内容:

- 行首为 1 个字符'#'，表示注释行，后接空格，空格后为注释内容，持续到换行符。
- 行首为 1 个字符'N'，表示神经元。后接空格，空格后为 1 个浮点数和 1 个整数（两者之后各有 1 个空格），之后为换行符。浮点数表示此神经元的 bias；整数表示此神经元使用的激活函数序号，0 为无激活函数，1~3 分别为 Sigmod、SoftMax 和 ReLU。
- 行首为 1 个字符'L'，表示层。后接空格，空格后为 2 个整数（每个整数之后有 1 个空格），之后是换行符。整数表示此层中神经元最小和最大索引号（神经元在文件中出现的先后顺序，最小为 0）。
- 行首为 1 个字符'S'，表示突触。后接空格，空格后为 2 个整数和 1 个浮点数（每个数之后有 1 个空格），之后是换行符。整数表示突出所连接的起点神经元和终点神经元索引号（神经元在文件中出现的先后顺序，最小为 0，-1 表示无起点或无终点）。

- 行首为 1 个字符'G'，表示网络名称。后接空格，空格后为一个不再包含空格的连续字符串，之后为换行符。一个文件中仅仅包含一个 G 开头的行。

OBJ 文件同时满足下列规则：

- 所有 N 开头的行，一定连续出现，且均在所有以 L 和 S 开头的行之前。
- 所有 L 开头的行，一定连续出现。
- 所有 S 开头的行，一定连续出现。

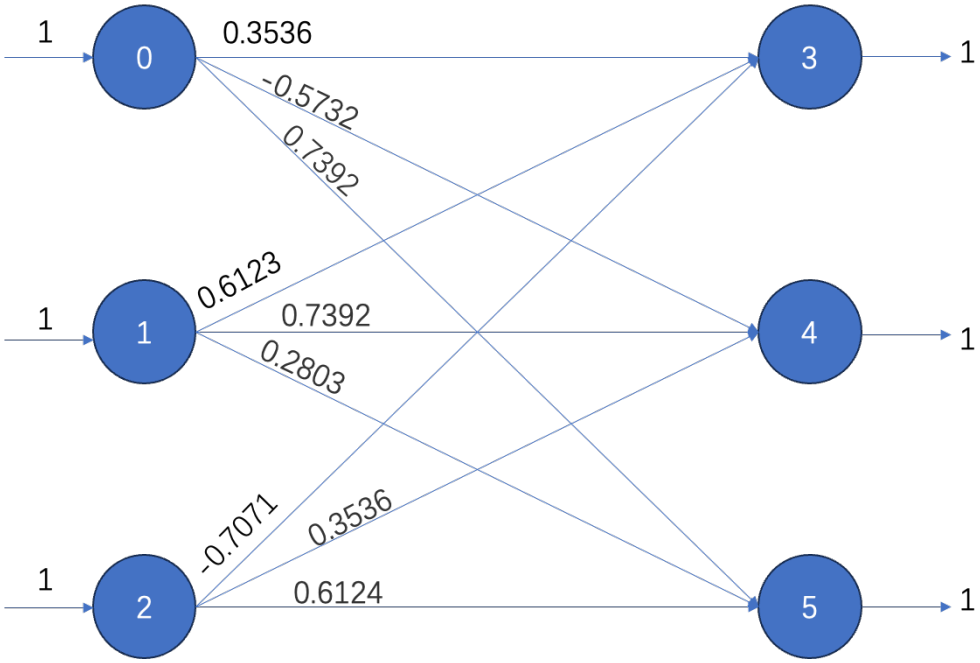
3、 示例

大作业压缩包中的 simple.ANN 文件，用文本编辑器打开后，为以下内容：

```
# simple.ANN
G RotationNetwork
# Six Neurons: zero bias, without activation function
N 0.0 0
N 0.0 0
N 0.0 0
N 0.0 0
N 0.0 0
N 0.0 0
# Layer 0: Neuron 0 to 2
L 0 2
# Layer 1: Neuron 3 to 5
L 3 5
# Neuron 0 to 2: has one Dendrite
S -1 0 1.0
S -1 1 1.0
S -1 2 1.0
# Neuron 3 to 5: has one Axon
S 3 -1 1.0
S 4 -1 1.0
S 5 -1 1.0
# Dendrites from Neuron 0 to Neuron 3~5
S 0 3 0.3536
S 0 4 -0.5732
S 0 5 0.7392
# Dendrites from Neuron 1 to Neuron 3~5
S 1 3 0.6123
S 1 4 0.7392
```

```
S 1 5 0.2803
# Dendrites from Neuron 2 to Neuron 3~5
S 2 3 -0.7071
S 2 4 0.3536
S 2 5 0.6124
```

第 1 行，为注释行，注释内容为文件名；
第 2 行，为网络名称 RotationNetwork；
第 3 行为第 4~9 行的注释；
第 4~9 行，为神经元信息，每行 1 个，索引号 0~5。每个神经元 bias 均为 0，激活函数为 f(x)=x；
第 10 行，为第 11 行的注释；
第 11 行，为层信息，第 0 层包含了神经元 0~2；
第 12 行，为第 13 行的注释；
第 13 行，为层信息，第 1 层包含了神经元 3~5；
第 14 行，为第 15 到 17 行的注释；
第 15~17 行，为输入神经元的树突信息，均无前驱神经元、Weight 为 1；
第 17 行，为第 19~21 行的注释；
第 19~21 行，为输出神经元的轴突信息，均无后继神经元、Weight 为 1；
第 22 行，为第 23~25 行的注释；
第 23~25 行，为从第 0 号神经元分别连接至第 3~5 号神经元的突触信息；
第 26 行，为第 27~29 行的注释；
第 27~29 行，为从第 1 号神经元分别连接至第 3~5 号神经元的突触信息；
第 30 行，为第 31~33 行的注释；
第 31~33 行，为从第 2 号神经元分别连接至第 3~5 号神经元的突触信息；



附录 2:

面向对象程序设计与训练课程编码规范

V1.3 版本 by 范静涛 @ 31/07/2024

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 hpp 文件和 cpp 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 hpp 文件中）、#include 部分、#define 部分、类型声明和定义性声明部分、实现部分等。
- 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： `Count = 2;` 而不是 `Count=2;`
- 三目运算符的“？”和“：”前后均各加一个空格。
- 函数的各参数（包括形参和实参）间、初始化列表的各个初始值间、枚举类型定义的枚举值间、模版参数间，要用“，”和后续一个空格隔开，考虑可读性的对齐可使用多个空格。

  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“：”前后，要有一个空格。

理由：提高代码的可读性。

2.1.3 哪里可以使用空格

级别：建议

描述：考虑可读性的对齐可使用多个空格

举例：

```
int a1 =  5; //等号后 2 个空格，目的是对齐
int b  = 10; //等号前 2 个空格，目的是对齐
```

理由：提高代码的可读性。

2.1.4 哪里不应该使用空格

级别：规则

描述：

- 不要在引用性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;
```

2.1.5 缩进

级别：规则

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

举例：

```
for (……) {
    if (……) {
        int a =  5; //等号后 2 个空格，是
        int b = 10;
    }
}
```

理由：提高代码的可读性。

2.1.6 长语句的书写格式

级别：建议

前置约束：应极力避免一个语句中优先级层次较多，或控制嵌套深度较大。如优先级层次较多，或控制嵌套深度较大导致语句过程，应拆分成多个语句。

描述：满足前置条件的较长的语句（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。一般来说，一个长语句，起始于 x 列，需拆分为 n 行才能满足 80 字符要求，那么第 1 行起点位于 x 列，第 2~n 行起点均位于 x+4 列。

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;
```

2.1.7 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，与控制语句的首字母应处在同一列上；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--;
    }
}
```

2.1.8 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++;
}
```

2.1.9 一次只声明、定义一个变量/常量

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/常量。
理由：提高代码的可读性，方便加入后置注释。
举例：

```
// 不要这样写
int width, length;
// 应该这样写
int width;
int length;
```

2.1.10 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;
```

2.1.11 将 “*”(声明/定义指针类型)、“&”(定义引用)和类型写在一起

级别：规则
描述：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：

```
// 不要像下面这样写代码：
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 (Number)
{
   case 1:
       Count++;
       break;
   case 2:
   case 3:
      Count--;
      break;       // 当 Number 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 (ShortRadio == 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(Number );  //设置当前的图形个数。
```

- 赋值。通过注释，说明赋值的意义；

  例如：

```
IsDraw = true;  //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：

```
int   Count = 0;            //计数器，表示正在处理第几个数据块
bool  IsNeedSave;           //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。如使用 ID 等约定俗成的缩写，可以全大写。对于作用域和类型的变量（数据成员），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

作用域	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
类的私有/受保护数据成员、文件作用域变量（文件中变量。只在某个.cpp 文件中可以使用。但如整个程序只有一个.cpp 文件，应当认为是全局变量）	m_	m_cClassCode 私有/受保护/文件作用域整型变量
类的私有/受保护指针型数据成员、文件作用域指针变量	m_p	m_pNumber
非静态局部变量、形参	无	Price 局部单精度浮点型变量
静态局部变量	s_	s_Number
类的公有成员(包括静态)	无	

表 2-2 类型前缀

例外：公有数据成员、局部变量、形参，无类型前缀

数据类型	类型前缀	示例

char	c（优先级第 3）	m_cClassCode 文件作用域字符型变量
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 私有作用域短整型变量
long int	l（优先级第 3）	m_lCount 受保护长整型变量
long long int	ll（优先级第 3）	g_llBigCount 全局长长整型变量
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）	m_fPrice 私有单精度浮点型成员
double	r（优先级第 3）	m_rPrice 受保护双精度浮点型成员
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
非简单类型（如结构体、类、枚举、模版）	无	
引用（不是类型）	无	

- 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/constexpr 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、enum class、union、typedef、using 声明的类型、namespace。

  例如：typedef struct Student;

  class MsgDialog;
- 函数

  函数名应以大写字母打头，由至少一个动词性英文单词或动宾型英文短语构成。各英文单词的首字母分别大写，其他字母一律小写。如使用 ID 等约定俗成的缩写，可以全大写。

  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：_Find 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；
- bool 类型数据/函数名标识符，必须以 Is 或 Has 等开头，用于表示取值为 true 时的指向性。
- 不可使用下划线或双下划线开头

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（包括 stl 迭代器），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.hpp 包括的都是和数学运算相关的函数声明。

举例：

将类 MsgDialog 的头文件和实现文件命名为 MsgdDialog.hpp 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 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/常量的同时，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 nullptr。

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 nullptr。

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(i); i++) {
    ...
}

//最好这样写：
bool IsAlwaysTrue = true;
while(IsAlwaysTrue == true) {
    ...
}
if(test == true) {
    ...
}
for(i = 1; Function(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：除构造函数外，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（模板和 inline 函数例外，必须在头文件中给出声明和定义）。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.hpp 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.cpp 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(模板和 inline 函数例外，必须在头文件中给出声明和定义)。但宏和 const 要分情况讨论，不一定受本规则限制。

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/类的声明

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.hpp，它的文件结构应该为：
#ifndef AUDIT_HPP    //第一行
#define AUDIT_HPP    //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:除非该类不应该存在默认构造函数,否则必须为每一个类显式定义默认构造函数。using 继承基类构造函数是可以的。

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 MyClass {
    MyClass();
    ...
};
```

2.8.2 关于拷贝构造函数

规范级别：规则

规则描述：除非该类不应该存在拷贝构造函数，否则必须为每一个类显式的定义拷贝构造函数

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 MyClass
{
    ...
    MyClass(const MyClass& object);
    ...
};
```

2.8.3 为类重载“=”操作符

规范级别：规则

规则描述：除非该类不应该存在拷贝构造函数，否则必须为每一个类显式的重载“=”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 MyClass
{
    ...
    MyClass& operator = (const MyClass& 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 MyClass
{
    ...
    ~MyClass ();
    ...
};
```

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，除非该数据成员没有任何读写规则（public 常引用性成员认为没有读规则，是可以的）。
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。

2.8.10 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。

2.8.11 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

2.8.12 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。

2.8.13 类的声明和实现必须分离

规范级别：规则
规则描述：

- 类的声明在 hpp 中，实现在 cpp 中。
- 内嵌类的声明在外围类声明所属 hpp 中，实现在外围类实现所属 cpp 中。
- 类模版的声明和实现均在 hpp 中，但声明和实现仍需分离，声明在前，实现在后，不可使用定义性声明。
- 类中 inline 函数的声明和实现均在 hpp 中，但声明和实现仍需分离，声明在前，实现在后，不可使用定义性声明。

理由：内外有别，大多数场景下，不需要后续开发者/使用者解除实现部分。

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。