工程开发基础---编码规范

声明：

> 本课程仅仅对程序员在开发过程中基础编程行为的进行规范，一是用来提高项目开发的可交流性，二是用来提高程序程序设计的质量。对于算法上并不做规范要求。仅作内部交流使用。

目录

正文

第01章 课程目标

软件质量一直都是被程序员挂在嘴边而没有实际行动的一件重要的事情，不论是刚迈入程序员道路的新手，还是正在程序员的道路上行走的老手。

本课程从高质量软件的基础出发，对高质量软件编程进行了介绍。之后对高质量软件编程设计进行了编码规范。

通过编程规范进行程序的设计，一方面可以提高编程的效率和高质量的设计，另一方面可以形成良好的编码风格。不仅方便别人阅读自己的代码，还便于交流，提高编程水平。最后对高质量编程的进阶之路进行了简单的介绍。

> 什么是好的设计？
>
> 1. 好的设计应当恰如其分地反映、解决一个具体的问题，没有不必要的特性；
> 2. 好的设计应该尽量降低对象之间、模块之间的耦合性。

要想提高编程质量，仅仅通过看书是有限的，要将书上的知识与实际的编码风格结合，逐渐过渡到成熟的编码风格。

第02章 课程内容

2.1 内容简介

本课程第一章对高质量软件编程进行了介绍，解答了为什么要进行高质量编程。第二三章对高质量软件编程之路进行了详细的介绍。通过不断的练习，形成自己的高质量编码风格。最后对课程进行了总结。接下来进入高质量软件编程之路的学习吧！！！

2.2 高质量软件编程之道

2.2.1 软件质量的定义

> 软件质量：
>
> 1. 一个系统、组件或过程符合特定需求的程度；
> 2. 一个系统、组件或过程符合客户或用户的要求或期望的程度。

> 如何理解软件质量的定义？
>
> 通俗的讲，就是软件能够按照既定的规则运行，并且不会出现错误。

当然，什么东西都有评价方法，我们可以通过考察软件的质量属性来评价软件的质量。软件开发人员可以对正确性、健壮性、可靠性、效率、易用性、可读性（可理解性）、可扩展性、可复用性、兼容性、可移植性等质量属性应进行详细的了解和认识，并给出提高软件质量的方法。

2.2.2 高质量软件的属性

软件的质量属性有很多，我们从实用性出发，在这里主要介绍10种重要的软件质量属性。

从功能的角度出发，将10种软件质量属性划分为：

> - 功能性质量属性：正确性、健壮性、可靠性。
> - 非功能性质量属性：性能、易用性、清晰性、安全性、可扩展性、兼容性、可移植性。

- 功能性

  正确性：软件质量属性中最重要的部分，指软件按照需求正确执行任务的能力。从软件的"需求开发"到"系统设计"到"实现"，都对软件的正确性有很高的要求。

健壮性：在软件出现异常的情况下，软件依然正常的运行的能力。健壮性是软件在需求范围之外的行为。

> 健壮性的两层含义：1、容错能力。2、恢复能力。
>
> - 容错：指发生异常情况时系统不出错误的能力。(针对不同的软件项目需求，容错性有很大的差别。)
> - 恢复：指软件发生错误后能够重新运行，并且恢复到没有发生错误前的状态。(例如：使用电脑时，有时会出现死机，但重启后，系统依然能够恢复到之前的状态)

可靠性：是一个与时序相关的属性，指在一定环境，在一定时间段内，程序不出现故障的概率。用"平均无故障时间"来衡量。

> 故障和错误的区别：
>
> - 故障：在测试时的环境和条件不足以使代码或硬件中的错误暴露。不可预料，后果严重。
> - 错误：例如语法、语义等可预见的错误，在调试的时候就可以改正。

- 非功能性

性能：指软件的"时间-空间"效率，可以通过优化数据结构、算法复杂度进行提前的预估。

> 目标："既要马儿跑的快，又要马儿吃得少"。

易用性：指用户使用软件的容易程度。(最好是使用按键对应的功能)，测试最好是从用户角度出发，进行用户测评。

清晰性：指软件成果易读、易理解，让用户在使用时明白用的是什么东西。和易用性相照应。只有研发人员思路清晰，才能写出易读，可理解的程序，简洁。

安全性：指信息安全。

> 不存在绝对安全的软件或系统，但如果盗用软件或系统的代价大于开发软件的成本，就认为软件或系统是安全的。

可扩展性：指软件适应"变化"的能力。 (软件或系统设计阶段重点考虑的质量属性)

兼容性：指两个或两个以上的软件相互交换信息的能力。

> 兼容性规则：弱者设法与强者兼容，强者应避免被兼容。

可移植性：指软件不经修改或稍加修改就可以运行于不同软硬件环境的能力。

2.2.3 为什么要讲高质量软件编程

程序员最大的对手是自己，随意的按照自己的规则去写程序。当自己写的东西里有缺陷，而自己花了很长时间还没有找到缺陷，这时就懊恼自己的编程的坏习惯。只有在编程初期逐渐养成好的习惯，才能在出现错误时，很快的定位错误的位置。

我们学习高质量软件编程的目的是：

> 在写程序的时候一次性的写出高质量程序，而不是在程序出错的时候花费大量的时间去找错误。

提高软件的质量最本质上是消除缺陷，消除缺陷有以下三种方式：

> 1. 将高质量编程内建于开发过程中，而不是在出错后花费大量的时间找错误，**最佳方式**。
> 2. 在开发完成后立马**进行软件测试**，从而尽快的消除软件开发过程中的一些缺陷。
> 3. 软件交付后，出现问题，及时补救。

2.2.4 软件的调试与测试

- 软件的调试

  > 软件调试：寻找错误的根源的过程。

  调试的基本方法是“粗分细找”。有些隐藏很深的Bug，应该运用归纳、推理等方法，确认Bug的范围，最后再用调试工具仔细地跟踪此范围的源代码。

  调试时注意事项：

  1. 寻找到错误不要急于修改，应先思考修改该错误会不会引发其他的错误。
  2. 思考是不是还存在类似的错误，一并解决。
  3. 修改完错误后要马上回归测试，避免其他错误产生。

- 软件的测试

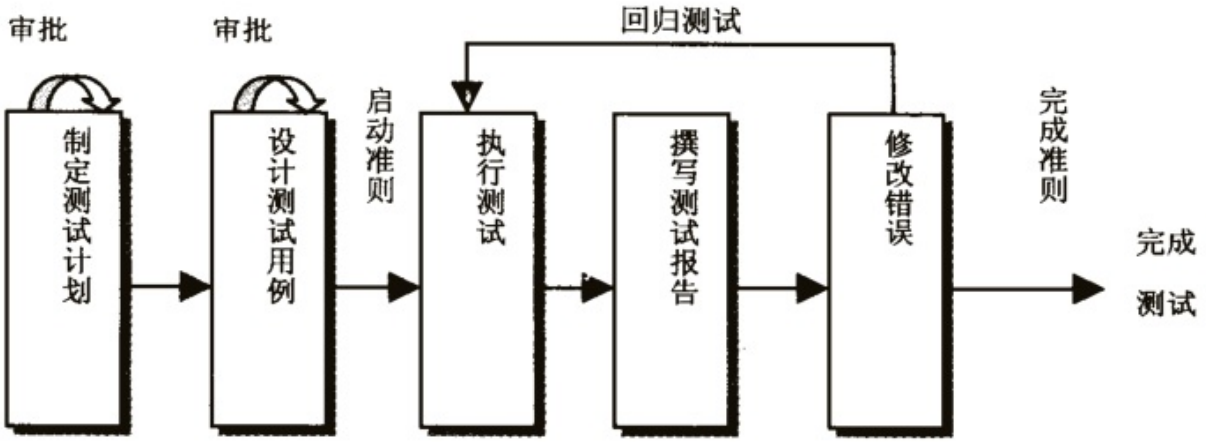
  > 软件测试是通过测试用例来找出软件的缺陷。

  在软件开发过程中，编程与测试是紧密相关的，但是现在软件的测试一直是被程序员或企业所忽略的，一是软件的测试会增加软件的开发成本，二是没有系统的软件测试人员和相关知识。

  由于本文档只是对C/C++进行编码规范，就不花费大量的篇幅去介绍软件测试的方法和流程，仅仅给出简单的方法和流程。但并不代表软件的测试不重要，相反软件的测试是软件投产最重要的步骤，希望读过此文档之后能够重视起来，并且深入学习相关的测试方法。

  ```
  软件测试的常规分类：
  测试阶段：单元测试、集成测试、系统测试、验收测试。
  测试方式：白盒测试、黑盒测试。
  测试内容：功能测试、健壮性测试、性能测试、用户界面测试、安全性测试、压力测试、可靠性测试.....
  ```

  测试流程：



2.3 开始高质量软件编程之路

2.3.1 排版

- **1. 文件结构**

每个C/C++程序通常分为两类文件，一类文件用于保存程序的声明，称为头文件；另一类文件用于保存程序的实现，称为源文件。

C/C++程序的头文件以.h为后缀，C程序的源文件以.c 为后缀，C++程序的源文件以.cpp为后缀。

C/C++编译器在读取到一条函数调用语句时首先应当知道该函数的原型和定义，函数原型一般都放在头文件中，函数定义放在源文件中，当编译器在源文件中读取到#include指令包含另一个头文件的时候，编译预处理器用头文件的内容取代#include指令，最后头文件的所有内容会被合并到某一个或几个源文件中。

- **版权声明**

> 为什么要在头文件和源文件前加入版权和版本信息？
> 如果你在企业工作，请记住：你已经不是学生了，你工作期间编写的一切代码都是属于企业的，所以要给每个程序打上企业的“名号”，即版权和版本声明。

版权、版本信息的主要内容有：

1. 版权信息；
2. 文件名称，标识符，摘要；
3. 当前版本号，作者/修改者，完成日期；
4. 版本历史信息。

格式参考入下：

```
/************************************************
* Copyright (c) 2017,北京中科浩电科技有限公司研发部
```

```
* All rights reserved.
*
*
* 文件名称：  filename.h
* 文件标识：  见配置管理计划书
* 摘      要：  简要描述本文件的内容、功能
*
*
* 当前版本：  1.1
* 作      者：  输入作者（或修改者）名字
* 完成日期：  ***年**月**日
* 修改说明：
*
* 取代版本：1.0
*  原作者 ：  输入原作者（或修改者）名字
* 完成日期：  ***年**月**日
* 原版本说明：
*
**************************************************/
```

- **头文件的结构与作用**
  - **头文件的作用：**

1. 通过头文件来调用库函数。用户只要按照头文件中的接口声明来调用库函数，不必关心接口的实现；
2. 头文件能加强类型安全检查，如果出现函数实现或使用时的方式与头文件中的声明不一致，编译器会指出错误；
3. 提高程序的可读性。

  - **头文件的结构：**

[规则2-3-1-1] 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。

[规则2-3-1-2] 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）。

[规则2-3-1-3] 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）。

结构如下：

> 1. 头文件的注释(包括文件说明、功能描述、版权声明等)。
> 2. 内部包含"卫哨开始"(#ifdef 头文件名大写、#define 头文件名大写)
> 3. #include其他头文件 (如果需要)
> 4. 外部变量、全局函数声明 (如果需要)
> 5. 常量、宏定义 (如果需要)
> 6. 类型前置声明、定义 (如果需要)
> 7. 全局函数原型和内联函数的定义 (如果需要)

8. 内部包含“卫哨结束” #endif
9. 文件版本及修订说明

格式参考如下：

```
#ifndef GRAPHICS_H // 防止 graphics.h 被重复引用
#define GRAPHICS_H
#include <math.h>  // 引用标准库的头文件
…
#include “myheader.h” // 引用非标准库的头文件
…
void Function1(…); // 全局函数声明
…
class Box // 类结构声明
{
    …
};
#endif
```

- **源文件的结构**

  C/C++源文件用来保存函数的实现和类的实现，其结构如下：

  1. 源文件注释(包括文件说明、功能描述、版权声明等等)
  2. 预处理指令
  3. 常量 、宏定义
  4. 外部变量声明、全局变量定义及初始化
  5. 成员函数和全局函数的定义
  6. 文件修改记录

上述头文件、源文件的结构可根据实际情况灵活安排。

- **2. 程序文件目录结构**

一个正在开发的程序工程不仅包含源代码文件，还包含许多资源文件、数据文件、库文件及配置文件等等，为了方便于开发与维护，我们需要有个一目了然的目录结构。例如，我们创建一个C/C++程序工程TestProj，其程序目录结构可以参照图2-1上所示。

图2-1 程序目录结构参照图：

图2-1说明：

> 1. **Include**存放程序的头文件(.h),还可以添加子文件。
> 2. **Source**存放程序的源文件(.c .cpp)，还可以添加子文件。。
> 3. **Shared**存放一些共享的文件。
> 4. **Resource**存放程序所用的各种资源文件，还可以添加子文件。
> 5. **Debug**存放程序发行版本生成的中间文件。
> 6. **Bin**存放程序员自己创建的lib文件或dll文件。

- **3. 程序版式**

版式虽然不会影响程序的功能，但是会影响程序的可读性和清晰性。可以使维护人员明了程序的内容。

  - **空行**

> > 空行起着分隔程序段落的作用。使用得当使得程序的布局更加清晰。

[规则2-3-1-4] 在每个类声明之后、每个函数定义结束之后都要加空行。

[规则2-3-1-5] 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。

实例：

```
// 空行
void Function1(…)
{
    …
}
// 空行
void Function2(…)
{
    …
}
```

```
// 空行
while (condition)
{
    statement1;
    // 空行
    if (condition)
    {
        statement2;
    }
    // 空行
    statement4;
}
```

- **代码行**

[规则2-3-1-6] 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。

规则[2-3-1-7] 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。

[规则2-3-1-8]尽可能在定义变量的同时对变量进行初始化。(就近原则)

实例：

```
int width = 10;  // 宽度
int height = 10; // 高度
int depth = 10;  // 深度
//空行
x = a + b;
y = c + d;
z = e + f;
//空行
if (width < height)
{
    dosomething();
}
//空行
for (initialization; condition; update)
{
    dosomething();
}
// 空行
other();
```

- **代码行内空格**

> 说明：采用这种松散方式编写代码的目的是使代码更加清晰

[规则2-3-1-9] 关键字之后要留空格。比如 const、virtual、inline、case 等关键字之后至少要留一个空格，否则无法辨析关键字;if、for、while 等关键字之后应留一个空格再跟左括号'（'，以突出关键字。

[规则2-3-1-10] 函数名之后不要留空格，紧跟左括号'（'，与关键字区别。

[规则2-3-1-11] 左括号'（'向后紧跟，右括号'）'、逗号'，'、分号';'向前紧跟，紧跟处不留空格。

[规则2-3-1-12] 逗号'，'之后要留空格，比如 Function(x, y, z)。如果分号';'不是一行的结束符号，其后要留空格，比如 for 空格(initialization; condition; update)。

[规则2-3-1-13] 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，比如"="、"+="、">="、"<="、"+"、"-"、"*"、*/*、"%"、"&&"、"||"、"<<","^"等二元操作符的前后应当加空格。

[规则2-3-1-14] 单目操作符如"!"、"~"、"++"、"--"、"&"（地址运算符）等前后不加空格。

[规则2-3-1-15] 比如 中括号" [ ] "、小数点"."、立即操作符"->"这类操作符前后不加空格。

[规则2-3-1-16] 修饰符 * 和 & 应当紧靠进变量名。

> 说明：若将修饰符 * 靠近数据类型,从语法上来讲比较符合，但容易引起误解。比如：int* x, y; 此处 y 容易被误解为指针变量。

建议： 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))

示例：

```
void Func(int a, int b, int c);

if (min > 1000)
if ((a >= b) && (b < c>))

for (i = 0; i < 100; i++)

x = (a < b) ? a : b;

int *p = &x;
array[6] = {0};
a.Func();
b->Func();

int *x, y;                    //变量y不会被误解为指针变量
```

- **对齐**

[规则2-3-1-17] 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。

[规则2-3-1-18] {}之内的代码块在 '{' 右边数格处**左对齐**。

示例：

```
void Function(int x)
{
    …     // program code
}
//空行
if (condition)
{
    …    // program code

}else
{
    …   // program code
}
//空行
如果出现嵌套的｛｝，则使用缩进对齐，如：
{
    …

    {
        …
    }

    …
}
```

- **长行拆分**

[规则2-3-1-19] 代码行最大长度宜控制在 70 至 80 个字符以内。

[规则2-3-1-20] 较长的语句（ >80 字符）要分成多行书写，长表达式要**在低优先级操作符处划分新行，操作符放在新行之首**，划分出的新行要进行适当的缩进，使排版整齐。

[规则2-3-1-21] 循环、判断等语句中若有较长的表达式或语句，则要进行适应的划分，**长表达式要在低优先级操作符处划分新行，操作符放在新行之首**。

示例：

```
perm_count_msg.head.len = NO7_TO_STAT_PERM_COUNT_LEN
                        + STAT_SIZE_PER_FRAM * sizeof( _UL );

act_task_table[frame_id * STAT_TASK_CHECK_NUMBER + index].occupied
                = stat_poi[index].occupied;

if ((very_longer_variable1 >= very_longer_variable12)
```

```
    && (very_longer_variable3 <= very_longer_variable14)
    && (very_longer_variable5 <= very_longer_variable16))
{
    dosomething();
}

for (very_longer_initialization;
    very_longer_condition;
    very_longer_update)
{
    dosomething();
}
```

- **类的版式**

> 类可以将数据成员和成员函数封装在一起，其中成员函数表示了类的行为（或称服务）。类提供关键字 public、protected 和 private，分别用于声明哪些数据成员和成员函数是公有的、受保护的或者是私有的。这样可以达到信息隐藏的目的。

类的版式有两种：

1. 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**。
2. 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**（或服务）。

建议：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。

- **4. 注释**

C 语言的注释符为“/.../”。C++语言中，程序块的注释常采用“/.../”，行注释 一般采用“//...”。注释通常用于：

- 版本、版权声明；
- 函数接口说明；
- 重要的代码行或段落提示。

一般情况下，源程序有效注释量必须在 0~20%以上。并且统一注释格式。

> 说明：注释的原则是有助于对程序的阅读理解，在该加的地方都加了，注释不宜太多也不能太少，注释语言必须准确、易懂、简洁。

[规则2-3-1-22] 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。

[规则2-3-1-23] (针对企业) **说明性文件头部应进行注释**，注释必须列出：版权说明、版本号、生成日期、作者、内容、功能、与其它文件的关系、修改日志等，头文件的注释中还应有函数功能简要说明。

示例：

```
/*************************************************************
版权说明
File Name:                  //文件名
Author:          Version:           Date:          //作者、版本号、日期
Description:      //描述此程序文件完成的主要功能，
                  //与其他模块或函数的接口，输出值、取值范围、含义
                  //及参数间的控制、顺序、独立或依赖关系
Others:           //其他内容的说明
Function List:    //主要函数列表，每条记录包含函数名及其功能简要说明。
1....
History:         //修改历史记录，包括修改时间、修改人及修改内容简述。
1. date:
    Author:
    Modification:
2. .....

*************************************************************/
```

[规则2-3-1-24](针对企业) 源文件头部进行注释，列出：版权说明、版本号、生成日期、作者、模块功能、主要函数其功能、修改日志等。

示例：

```
/*************************************************************
版权说明
FileName: test.c
Author:       Version:      Date:
Description:     //模块描述
Version:         //版本信息
Function List:   //主要函数及其功能
    1. .....
History:         //历史修改记录
    作者        时间       版本      修改描述

*************************************************************/
```

[规则2-3-1-25] 函数头部进行注释，列出：函数的功能、输入参数、输出参数、返回值、调用关系(函数、表)等。

示例：

```
/*************************************************************
Function:       //函数名称
Description:    //函数功能、性能的描述
```

```
Calls:          //被本函数调用的函数清单
Called By:      //调用本函数的函数清单
Table Accessed: //被访问的表(数据库)
Table Updated:  //被修改的表(数据库)
Input:          //输入参数说明，每个参数的作用、取值范围及参数间的关系
Output:         //输出参数说明
Return:         //返回值说明
Others:         //其他附加说明

*****************************************************/
```

[规则2-3-1-26] 注释应与其描述的代码相近，**对代码的注释应放在其上方或右方（对单条语句的注释）相邻位置**，不可放在下面，如放于上方则需与其上面的代码用空行隔开。

示例：

```
/* get replicate sub system index and net indicator */
repssn_ind = ssn_data[index].repssn_index;
repssn_ni = ssn_data[index].ni;
```

[规则2-3-1-27] 对于所有有物理含义的变量、常量，如果其命名不是充分自注释的，在声明时都必须加以注释，说明其物理含义。**变量、常量、宏的注释应放在其上方相邻位置或右方**。

示例：

```
/* active statistic task number */
#define MAX_ACT_TASK_NUMBER 1000
#define MAX_ACT_TASK_NUMBER 1000    /* active statistic task number */
```

[规则2-3-1-28] 全局变量要有较详细的注释，包括对其功能、取值范围、哪些函数或过程存取及存取说明。

示例：

```
/* The ErrorCode when SCCP translate */
/* Global Title failure, as follows */ // 变量作用、含义
/* 0 — SUCCESS 1 — GT Table error */
/* 2 — GT error Others — no use */ // 变量取值范围
/* only function SCCPTranslate() in */
/* this modual can modify it, and other */
/* module can visit it through call */
/* the function GetGTTransErrorCode() */ // 使用方法
BYTE g_GTTranErrorCode;
```

[规则2-3-1-29] 数据结构声明(素组、结构体、类、枚举等)必须加以注释。对数据结构的注释应放在其上方；对结构中的变量注释应放在其右方。

示例：

```
/*数据结构的名称及其主要信息*/
enum  WEEKDAY            /*类型标识符(大写) */
{
    SUNDAY,
    MONDAY,
    FRIDAY,
    TUSDAY                    /*枚举值表*/
};
```

[规则2-3-1-30] 注释与解释内容同样进行缩进；并且将注释与其上方的代码使用空行隔开。

示例：

```
void code_Func(void)
{
    /*code one*/
    Code one
    //空行
    /*code two*/
    Code two
}
```

[规则2-3-1-31] 在程序块结束行的右方加注释标记，表明该程序块结束。

示例：

```
if (index > 0)
{
    //program code
    while(index < MAX_INDEX)
    {
        //program code

    }/*end of while(index < MAX_INDEX)*/

}/*end of if(index > 0)
```

- **5. 命名规则**

命名规则其实是容易对程序员形成一种约束，不能让程序员自由的进行命名。但从程序理解程度上来

说，一些共性的命名规则能够更加使别人阅读程序。以下命名规则是被大多数程序员所采纳的。

- **标识符的命名**

[规则2-3-1-32] 标识符的命名要清晰，有明确的含义，同时缩写使用大家都基本可以理解的缩写。

示例：(一些公认的缩写)

```
temp 缩写为 tmp ;
flag 缩写为 flg ;
statistic 缩写为 stat ;
increment 缩写为 inc ;
message 缩写为 msg ;
```

- **常量、变量的命名**

[规则2-3-1-33] 对于变量命名，禁止取单个字符(i,j,k...)，建议使用小写字母开头的单词组合。在局部循环变量中i，j，k...是允许的。

示例：

```
BOOL flag;
int drawMode;
```

[规则2-3-1-34] 常量全用大写的字母，用下划线分隔单词。 示例：

```
const int MAX = 100;
const int MAX_LENGTH = 100;
```

[规则2-3-1-35] 静态变量加前缀s_(static)，全局变量使用前缀g_（global） 示例：

```
static int s_initValue;
int g_howManyPeople;   //全局变量
```

- **接口的命名**

[规则2-3-1-36] ： 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。

> 说明：对接口部分的标识符应该有更严格限制，防止冲突。如可规定接口部分的变量与常量之前加上"模块"标识等。为了团队间更好的进行接口控制。

- **宏定义的命名**

[规则2-3-1-37] 使用宏定义表达式时，使用括号的进行隔离。

示例：

```
#define RECTANGLE_AREA(a, b)  (a) * (b)    //长方形的面积
```

[规则2-3-1-38] 将宏定义的多条表达式放在{}中。

示例：

```
#define INIT_RECT_VALUE(a, b)\
{\
    a = 0;\
    b = 0;\
}

for (index  = 0; index < RECT_TOTAL_NUM; index++)
{
    INIT_RECT_VALUE(rect[index].a, rect[index].b);
}
```

特别注意：使用宏定义时，不允许改变参数的值。

> **建议**：对于命名规则，可以参考驼峰命名法。写代码过程中在遵寻上述规则的同时，逐渐形成自己的命名规则。

2.3.2 变量、常量及表达式

- 常量

```
为什么要使用常量?
1．程序的可读性（可理解性）变差。程序员自己会忘记那些数字或字符串是什么意思，用户则更加不知它们从何处来、表示什么。
2．在程序的很多地方输入同样的数字或字符串，难保不发生书写错误。
3．如果要修改数字或字符串，则会在很多地方改动，既麻烦又容易出错。
```

常量的分类：

- 字面常量：直接出现的各种进制数，字符或字符串等。 示例：

```
x = -100.9f;
#define OPEN_SUCCESS 0x00000001
```

```
char c = 'a';
char *pChar = "adfkfas";
int *pInt = NULL;
```

- 符号常量：存在两种，用#define定义的宏常量、用const定义的常量。尽量使用含义直观的符号常量来表示程序中多次出现的数字或字符串。

示例：

```
#define MAX 100         //宏常量
const int MAX = 100;    //const常量
const float PI = 3.14; //const常量
```

- 契约性常量：契约性const常量的定义并未使用关键字const，但被看做是一个const常量。

示例：

```
void ReadValue(const int &num)
{
    printf("%d\n",num);
}

int main()
{
    int n = 0;

    ReadValue(n); //n被看做是const常量。

    return 0;
}
```

- 枚举常量：使用关键字enum来定义一些常量集合。

示例：

```
enum Gigantic
{
    SMALL = 10;
    GIGANTIC = 800000
};
```

- 在C++中const和define比较

C++ 语言可以用 const 来定义常量，也可以用 #define 来定义常量。但是前者比后 者有更多的优点：

- const 常量有数据类型，而宏常量没有数据类型。编译器可以对前者进行类型安全检查。而对后者只进行字符替换，没有类型安全检查，并且在字符替换可能会产生意料不到错误（边际效应）。
- 有些集成化的调试工具可以对 const常量进行调试，但是不能对宏常量进行调试。

- 常量使用规则

  [规则2-3-2-1] 需要对外公开的常量放在头文件中，不需要对外公开的常量放在定义文件的头部。为便于管理，可以把不同模块的常量集中存放在一个公共的头文件中。

  [规则2-3-2-2] 如果某一常量与其它常量密切相关，应在定义中包含这种关系，而不应给出一些孤立的值。

  示例：

```
const float RADIUS = 100;
const float DIAMETER = RADIUS * 2;
```

- 变量

- 表达式

  表达式属于C/C++的短语结构语法，看似简单，使用时隐患还是比较多的。

  - 表达式使用规则

  [规则2-3-2-] 如果代码中的运算符比较多，使用括号确定表达式的操作顺序，避免使用默认的优先级。

  示例：

```
word = (high << 8) | low

if ((a | b) && (a & c))
```

  [规则2-3-2-] 不要编写太复杂的复合表达式。

  [规则2-3-2-] 不要有多用途的表达式。

  [规则2-3-2-] 不要和数学表达式混淆。

  示例：

```
i = a < b && c < d && e >= f; //复合表达式过于复杂。
d = (a = b + c) + r; //过多表达式
if (a < b < c)   //数学表达式
```

2.3.3 函数设计

函数是C/C++程序的基本功能单元，在这里重要介绍函数设计的基础规则。

- **参数的规则**

  [规则2-3-3-1] 参数的书写要完整，如果函数没有参数，用void填充。

  示例：

```
void SetValue(int width,int height);  //参数命名规范
float GetValue(void);
```

  [规则2-3-3-2] 参数的命名要有意义。顺序要合理。

  示例：

```
void StringCopy(char *strDest, char *strSource);
                        //参数的顺序：一般将目的参数放在前面，源参数放在后面。
```

  [规则2-3-3-3] 如果参数是指针，且仅作输入用，则应该在类型前加const，防止指针在函数体内被意外修改。

  示例：

```
void StringCopy(char *strDest, const char *strSource);
```

- **返回值的规则**

  [规则2-3-3-4] 不要省略返回值的类型，如果函数没有返回值，那么应声明为void类型。

  [规则2-3-3-5] 函数名字与返回值类型在语义上不可冲突。

  [规则2-3-3-6] 不要将正常值和错误标志混在一起返回。

- **内部实现的规则**

- **函数功能的规则**

2.4 高质量软件编程进阶之路

2.4.1 内存管理

- **内存分配方式**

- **常见的内存错误及对策**
- **杜绝“野指针”**
- **三种传参方式**

2.4.2 程序效率

2.4.3 软件维护

第03章 课程总结

第04章 课程评价