1. **html、css和js代码规范**
2. **文件规范**
3. html,css,js,images文件均归档至<系统开发规范>约定的目录中;
4. html文件命名:英文命名,后缀.htm.同时将对应界面稿放于同目录中,若界面稿命名为中文,请重命名与html文件同名,以方便后端添加功能时查找对应页面;

3. css文件命名:英文命名,后缀.css.共用base.css,首页index.css,其他页面依实际模块需求命名.;

4. Js文件命名:英文命名,后缀.js.共用common.js,其他依实际模块需求命名.

**（2） html书写规范**

1.文档类型声明及编码:统一为html5声明类型;编码统一为,书写时利用IDE实现层次分明的缩进;

　　2.非特殊情况下样式文件必须外链至…之间;非特殊情况下JavaScript文件必须外链至页面底部;

　　3.引入样式文件或JavaScript文件时,须略去默认类型声明.

　　4.引入JS库文件,文件名须包含库名称及版本号及是否为压缩版,比如jquery-1.7.1.min.js;引入插件,文件名格式为库名称+插件名称,比如jQuery.cookie.js;

　　5.所有编码均遵循xhtml标准,标签&属性&属性命名必须由小写字母及下划线数字组成,且所有标签必须闭合;属性值必须用双引号包括;

　　6.充分利用无兼容性问题的html自身标签,比如span,em,strong,optgroup,label,等等;需要为html元素添加自定 义属性的时候,首先要考虑下有没有默认的已有的合适标签去设置,如果没有,可以使用须以”data-”为前缀来添加自定义属性，避免使用”data:”等 其他命名方式;

　　7.语义化html,如标题根据重要性用h(同一页面只能有一个h1),段落标记用p,列表用ul,内联元素中不可嵌套块级元素;

　　8.尽可能减少div嵌套

　　9.书写链接地址时,必须避免重定向，例如：href=”http://www.example.com/”,即须在URL地址后面加上“/”；

　　10.在页面中尽量避免使用style属性,即style=”…”;

　　11.必须为含有描述性表单元素(input,textarea)添加label

　　12.能以背景形式呈现的图片,尽量写入css样式中;

　　13.重要图片必须加上alt属性;给重要的元素和截断的元素加上title;

　　14.给区块代码及重要功能(比如循环)加上注释,方便后台添加功能;

　　15.特殊符号使用:尽可能使用代码替代:比如<(<)&>(>)&空格()&»(»)等等;

　　16.书写页面过程中,请考虑向后扩展性;

　　17.class&id参见css书写规范.

**（3）css书写规范**

1.编码统一为utf-8;

　　2.协作开发及分工:i会根据各个模块,同时根据页面相似程序,事先写好大体框架文件,分配给前端人员实现内部结构&表现&行为;共 用css文件base.css由i书写,协作开发过程中,每个页面请务必都要引入,此文件包含reset及头部底部样式,此文件不可随意修改;

　　3.class与id的使用:id是唯一的并是父级的,class是可以重复的并是子级的,所以id仅使用在大的模块上,class可用在重复使用率高及子级中;id原则上都是由我分发框架文件时命名的,为JavaScript预留钩子的除外;

　　4.为JavaScript预留钩子的命名,请以js\_起始,比如:js\_hide,js\_show;

　　5.class与id命名:大的框架命名比如header/footer/wrapper/left/right之类的在2中由i统一命名.其他样式 名称由小写英文&数字&\_来组合命名,如i\_comment,fontred,width200;避免使用中文拼音,尽量使用简易的单词 组合;总之,命名要语义化,简明化.

　　6.规避class与id命名(此条重要,若有不明白请及时与i沟通):

　　a）通过从属写法规避,示例见d;

　　b）取父级元素id/class命名部分命名,示例见d;

　　c）重复使用率高的命名,请以自己代号加下划线起始,比如i\_clear;

　　d）a,b两条,适用于在2中已建好框架的页面,如,要在2中已建好框架的页面代码中加入新的div元素

　　7.css属性书写顺序,建议遵循:布局定位属性–>自身属性–>文本属性–>其他属性.此条可根据自身习惯书写,但尽量保证同类 属性写在一起.属性列举:布局定位属性主要包括:display&list-style&position（相应的 top,right,bottom,left）＆float&clear＆visibility＆overflow；自身属性主要包 括:width&height&margin&padding&border&background;文本属性 主要包括：color&font&text-decoration&text-align&vertical- align&white-space&其他&content;我所列出的这些属性只是最常用到的,并不代表全部;

　　8.书写代码前,考虑并提高样式重复使用率;

　　9.充分利用html自身属性及样式继承原理减少代码量,比如:

　　10.样式表中中文字体名,请务必转码成unicode码,以避免编码错误时乱码;

　　11.背景图片请尽可能使用sprite技术,减小http请求,考虑到多人协作开发,sprite按模块制作;

　 12.使用table标签时(尽量避免使用table标签),请不要用width/height/cellspacing/cellpadding等 table属性直接定义表现,应尽可能的利用table自身私有属性分离结构与表现,如 thead,tr,th,td,tbody,tfoot,colgroup,scope;(cellspaing及cellpadding的css控制方 法:table{border:0;margin:0;border- collapse:collapse;}tableth,tabletd{padding:0;},base.css文件中我会初始化表格样式)

　　13.如何可以请少使用兼容;

　　14.用png图片做图片时,要求图片格式为png-8格式,若png-8实在影响图片质量或其中有半透明效果,请为ie6单独定义背景:

　　\_background:none;\_filter:progid:DXImageTransform.Microsoft.AlphaImageLoader(sizingMethod=crop,src=’img/bg.png’);

　　15.避免兼容性属性的使用,比如text-shadow||css3的相关属性;

　　16.减少使用影响性能的属性,比如position:absolute||float;

　　17.必须为大区块样式添加注释,小区块适量注释;

18.代码缩进与格式:建议单行书写,可根据自身习惯,后期优化会统一处理;

**（4）JavaScript书写规范**

1.文件编码统一为utf-8,书写过程过,每行代码结束必须有分号;原则上所有功能均根据XXX项目需求原生开发,以避免网上down下来的代码造成的代码污染(沉冗代码||与现有代码冲突||…);

　　2.库引入:原则上仅引入jQuery库,若需引入第三方库,须与团队其他人员讨论决定;

　　3.变量命名:驼峰式命名.原生JavaScript变量要求是纯英文字母,首字母须小写,如iTaoLun;

　　jQuery变量要求首字符为’\_’,其他与原生JavaScript规则相同,如:\_iTaoLun;

　　另,要求变量集中声明,避免全局变量.

　　4.类命名:首字母大写,驼峰式命名.如ITaoLun;

　　5.函数命名:首字母小写驼峰式命名.如iTaoLun();

　　6.命名语义化,尽可能利用英文单词或其缩写;

　　7.尽量避免使用存在兼容性及消耗资源的方法或属性,比如eval\_r()&innerText;

　　8.后期优化中,JavaScript非注释类中文字符须转换成unicode编码使用,以避免编码错误时乱码显示;

　　9.代码结构明了,加适量注释.提高函数重用率;

10.注重与html分离,减小reflow,注重性能.

**（5）开发及测试工具约定**

建议使用WebStorm||Aptana||Dw||Vim,亦可根据自己喜好选择,但须遵循如下原则:  
　　  
　　1.不可利用IDE的视图模式’画’代码;  
　　  
　　2.不可利用IDE生成相关功能代码,比如Dw内置的一些功能js;  
　　  
　　3.编码必须格式化,比如缩进;  
　　  
　　测试工具:前期开发仅测试FireFox&IE6&IE7&IE8&IE9&Opera&Chrome&Safari;

1. **Java代码规范**

1、Java注释尽可能全面

    对于方法的注释应该包含详细的入参和结果说明，有异常抛出的情况也要详细叙述：类的注释应该包含类的功能说明、作者和修改者。

2、多次使用的相同变量最好归纳成常量

    多处使用的相同值的变量应该尽量归纳为一个常量，方便日后的维护。

3、尽量少的在循环中执行方法调用

    尽量在循环中少做一些可避免的方法调用，这样可以节省方法栈的创建。例如：

    for（int i=0;i<list.size（）；i++）{

    System.out.println（i）；

    }

    可以修改为：

    for（int i=0,size=list.size（）；i<size;i++）{

    System.out.println（i）；

    }

4、常量的定义可以放到接口中

    在Java中，接口里只允许存在常量，因此把常量放到接口中声明就可以省去public static final这几个关键词。

 5、ArrayList和LinkedList的选择

    这个问题比较常见。通常程序员最好能够对list的使用场景做出评估，然后根据特性作出选择。ArrayList底层是使用数组实现的，因此随机读取数据会比LinkedList快很多，而LinkedList是使用链表实现的，新增和删除数据的速度比ArrayList快不少。

 6、String,StringBuffer和StringBuilder

    这个问题也比较常见。在进行字符串拼接处理的时候，String通常会产生多个对象，而且将多个值缓存到常量池中。例如：

    String a=“a”;

    String b=“b”;

    a=a+b;

    这种情况下jvm会产生“a”,“b”,“ab”三个对象。而且字符串拼接的性能也很低。因此通常需要做字符串处理的时候尽量采用StringBuffer和StringBuilder来。

7、包装类和基本类型的选择

    在代码中，如果可以使用基本数据类型来做局部变量类型的话尽量使用基本数据类型，因为基本类型的变量是存放在栈中的，包装类的变量是在堆中，栈的操作速度比堆快很多。

 8、尽早的将不再使用的变量引用赋给null

    这样做可以帮助jvm更快的进行内存回收。当然很多人其实对这种做法并不感冒。

 9、在finally块中对资源进行释放

    典型的场景是使用io流的时候，不论是否出现异常最后都应该在finally中对流进行关闭。

  10、在HashMap中使用一个Object作为key时要注意如何区分Object是否相同

在jdk的HashMap实现中，判断两个Object类型的key是否相同的标准是hashcode是否相同和equals方法的返回值。如果业务上需要对两个数据相同的内存对象当作不同的key存储到hashmap中就要对hashcode和equals方法进行覆盖。

1. **c++代码规范**

1. 头文件

通常每一个 .cpp 文件都有一个对应的 .h 文件. 也有一些常见例外, 如单元测试代码和只包含 main() 函数的 .cpp 文件.

正确使用头文件可令代码在可读性、文件大小和性能上大为改观.

下面的规则将引导你规避使用头文件时的各种陷阱.

1.1. #define 保护

Tip

所有头文件都应该使用 #define 防止头文件被多重包含, 命名格式当是: <PROJECT>\_<PATH>\_<FILE>\_H\_

为保证唯一性, 头文件的命名应该依据所在项目源代码树的全路径. 例如, 项目 foo 中的头文件 foo/src/bar/baz.h 可按如下方式保护:

#ifndef FOO\_BAR\_BAZ\_H\_

#define FOO\_BAR\_BAZ\_H\_

…

#endif // FOO\_BAR\_BAZ\_H\_

1.2. 头文件依赖

Tip

能用前置声明的地方尽量不使用 #include.

当一个头文件被包含的同时也引入了新的依赖, 一旦该头文件被修改, 代码就会被重新编译. 如果这个头文件又包含了其他头文件, 这些头文件的任何改变都将导致所有包含了该头文件的代码被重新编译. 因此, 我们倾向于减少包含头文件, 尤其是在头文件中包含头文件.

使用前置声明可以显著减少需要包含的头文件数量. 举例说明: 如果头文件中用到类 File, 但不需要访问 File 类的声明, 头文件中只需前置声明 class File; 而无须#include "file/base/file.h".

不允许访问类的定义的前提下, 我们在一个头文件中能对类 Foo 做哪些操作?

• 我们可以将数据成员类型声明为 Foo \* 或 Foo &.

• 我们可以将函数参数 / 返回值的类型声明为 Foo (但不能定义实现).

• 我们可以将静态数据成员的类型声明为 Foo, 因为静态数据成员的定义在类定义之外.

反之, 如果你的类是 Foo 的子类, 或者含有类型为 Foo 的非静态数据成员, 则必须包含 Foo 所在的头文件.

有时, 使用指针成员 (如果是 scoped\_ptr 更好) 替代对象成员的确是明智之选. 然而, 这会降低代码可读性及执行效率, 因此如果仅仅为了少包含头文件，还是不要这么做的好.

当然 .cpp 文件无论如何都需要所使用类的定义部分, 自然也就会包含若干头文件.

2. 类

类是 C++ 中代码的基本单元. 显然, 它们被广泛使用. 本节列举了在写一个类时的主要注意事项.

2.1. 构造函数的职责

Tip

构造函数中只进行那些没什么意义的 (trivial, YuleFox 注: 简单初始化对于程序执行没有实际的逻辑意义, 因为成员变量 “有意义” 的值大多不在构造函数中确定) 初始化, 可能的话, 使用 Init() 方法集中初始化有意义的 (non-trivial) 数据.

定义:

在构造函数体中进行初始化操作.

优点:

排版方便, 无需担心类是否已经初始化.

缺点:

在构造函数中执行操作引起的问题有:

• 构造函数中很难上报错误, 不能使用异常.

• 操作失败会造成对象初始化失败，进入不确定状态.

• 如果在构造函数内调用了自身的虚函数, 这类调用是不会重定向到子类的虚函数实现. 即使当前没有子类化实现, 将来仍是隐患.

• 如果有人创建该类型的全局变量 (虽然违背了上节提到的规则), 构造函数将先 main() 一步被调用, 有可能破坏构造函数中暗含的假设条件. 例如,gflags 尚未初始化.

结论:

如果对象需要进行有意义的 (non-trivial) 初始化, 考虑使用明确的 Init() 方法并 (或) 增加一个成员标记用于指示对象是否已经初始化成功.

2.2. 默认构造函数

Tip

如果一个类定义了若干成员变量又没有其它构造函数, 必须定义一个默认构造函数. 否则编译器将自动生产一个很糟糕的默认构造函数.

定义:

new 一个不带参数的类对象时, 会调用这个类的默认构造函数. 用 new[] 创建数组时，默认构造函数则总是被调用.

优点:

默认将结构体初始化为 “无效” 值, 使调试更方便.

缺点:

对代码编写者来说, 这是多余的工作.

结论:

如果类中定义了成员变量, 而且没有提供其它构造函数, 你必须定义一个 (不带参数的) 默认构造函数. 把对象的内部状态初始化成一致/有效的值无疑是更合理的方式.

这么做的原因是: 如果你没有提供其它构造函数, 又没有定义默认构造函数, 编译器将为你自动生成一个. 编译器生成的构造函数并不会对对象进行合理的初始化.

如果你定义的类继承现有类, 而你又没有增加新的成员变量, 则不需要为新类定义默认构造函数.

2.3. 显式构造函数

Tip

对单个参数的构造函数使用 C++ 关键字 explicit.

定义:

通常, 如果构造函数只有一个参数, 可看成是一种隐式转换. 打个比方, 如果你定义了 Foo::Foo(string name), 接着把一个字符串传给一个以 Foo 对象为参数的函数, 构造函数 Foo::Foo(string name) 将被调用, 并将该字符串转换为一个 Foo 的临时对象传给调用函数. 看上去很方便, 但如果你并不希望如此通过转换生成一个新对象的话, 麻烦也随之而来. 为避免构造函数被调用造成隐式转换, 可以将其声明为 explicit.

优点:

避免不合时宜的变换.

缺点:

无

结论:

所有单参数构造函数都必须是显式的. 在类定义中, 将关键字 explicit 加到单参数构造函数前: explicit Foo(string name);

例外: 在极少数情况下, 拷贝构造函数可以不声明成 explicit. 作为其它类的透明包装器的类也是特例之一. 类似的例外情况应在注释中明确说明.

2.4. 拷贝构造函数

Tip

仅在代码中需要拷贝一个类对象的时候使用拷贝构造函数; 大部分情况下都不需要, 此时应使用 DISALLOW\_COPY\_AND\_ASSIGN.

定义:

拷贝构造函数在复制一个对象到新建对象时被调用 (特别是对象传值时).

优点:

拷贝构造函数使得拷贝对象更加容易. STL 容器要求所有内容可拷贝, 可赋值.

缺点:

C++ 中的隐式对象拷贝是很多性能问题和 bug 的根源. 拷贝构造函数降低了代码可读性, 相比传引用, 跟踪传值的对象更加困难, 对象修改的地方变得难以捉摸.

结论:

大部分类并不需要可拷贝, 也不需要一个拷贝构造函数或重载赋值运算符. 不幸的是, 如果你不主动声明它们, 编译器会为你自动生成, 而且是 public 的.

可以考虑在类的 private: 中添加拷贝构造函数和赋值操作的空实现, 只有声明, 没有定义. 由于这些空函数声明为 private, 当其他代码试图使用它们的时候, 编译器将报错. 方便起见, 我们可以使用 DISALLOW\_COPY\_AND\_ASSIGN 宏:

// 禁止使用拷贝构造函数和 operator= 赋值操作的宏

// 应该类的 private: 中使用

#define DISALLOW\_COPY\_AND\_ASSIGN(TypeName) \

TypeName(const TypeName&); \

void operator=(const TypeName&)

在 class foo: 中:

class Foo {

public:

Foo(int f);

~Foo();

private:

DISALLOW\_COPY\_AND\_ASSIGN(Foo);

};

如上所述, 绝大多数情况下都应使用 DISALLOW\_COPY\_AND\_ASSIGN 宏. 如果类确实需要可拷贝, 应在该类的头文件中说明原由, 并合理的定义拷贝构造函数和赋值操作. 注意在 operator= 中检测自我赋值的情况 (yospaly 注: 即 operator= 接收的参数是该对象本身).

为了能作为 STL 容器的值, 你可能有使类可拷贝的冲动. 在大多数类似的情况下, 真正该做的是把对象的 指针 放到 STL 容器中. 可以考虑使用 std::tr1::shared\_ptr.

2.5. 结构体 VS. 类

Tip

仅当只有数据时使用 struct, 其它一概使用 class.

在 C++ 中 struct 和 class 关键字几乎含义一样. 我们为这两个关键字添加我们自己的语义理解, 以便为定义的数据类型选择合适的关键字.

struct 用来定义包含数据的被动式对象, 也可以包含相关的常量, 但除了存取数据成员之外, 没有别的函数功能. 并且存取功能是通过直接访问位域 (field), 而非函数调用. 除了构造函数, 析构函数, Initialize(), Reset(), Validate() 外, 不能提供其它功能的函数.

如果需要更多的函数功能, class 更适合. 如果拿不准, 就用 class.

为了和 STL 保持一致, 对于仿函数 (functors) 和特性 (traits) 可以不用 class 而是使用 struct.

注意: 类和结构体的成员变量使用 不同的命名规则.

2.6. 继承

Tip

使用组合 (composition, YuleFox 注: 这一点也是 GoF 在 <<Design Patterns>> 里反复强调的) 常常比使用继承更合理. 如果使用继承的话, 定义为 public 继承.

定义:

当子类继承基类时, 子类包含了父基类所有数据及操作的定义. C++ 实践中, 继承主要用于两种场合: 实现继承 (implementation inheritance), 子类继承父类的实现代码; 接口继承 (interface inheritance), 子类仅继承父类的方法名称.

优点:

实现继承通过原封不动的复用基类代码减少了代码量. 由于继承是在编译时声明, 程序员和编译器都可以理解相应操作并发现错误. 从编程角度而言, 接口继承是用来强制类输出特定的 API. 在类没有实现 API 中某个必须的方法时, 编译器同样会发现并报告错误.

缺点:

对于实现继承, 由于子类的实现代码散布在父类和子类间之间, 要理解其实现变得更加困难. 子类不能重写父类的非虚函数, 当然也就不能修改其实现. 基类也可能定义了一些数据成员, 还要区分基类的实际布局.

结论:

所有继承必须是 public 的. 如果你想使用私有继承, 你应该替换成把基类的实例作为成员对象的方式.

不要过度使用实现继承. 组合常常更合适一些. 尽量做到只在 “是一个” (“is-a”, YuleFox 注: 其他 “has-a” 情况下请使用组合) 的情况下使用继承: 如果 Bar 的确 “是一种” Foo, Bar 才能继承 Foo.

必要的话, 析构函数声明为 virtual. 如果你的类有虚函数, 则析构函数也应该为虚函数. 注意 数据成员在任何情况下都必须是私有的.

当重载一个虚函数, 在衍生类中把它明确的声明为 virtual. 理论依据: 如果省略 virtual 关键字, 代码阅读者不得不检查所有父类, 以判断该函数是否是虚函数.

2.7. 多重继承

Tip

真正需要用到多重实现继承的情况少之又少. 只在以下情况我们才允许多重继承: 最多只有一个基类是非抽象类; 其它基类都是以 Interface 为后缀的 纯接口类.

定义:

多重继承允许子类拥有多个基类. 要将作为 纯接口 的基类和具有 实现 的基类区别开来.

优点:

相比单继承 (见 继承), 多重实现继承可以复用更多的代码.

缺点:

真正需要用到多重 实现 继承的情况少之又少. 多重实现继承看上去是不错的解决方案, 但你通常也可以找到一个更明确, 更清晰的不同解决方案.

结论:

只有当所有父类除第一个外都是 纯接口类 时, 才允许使用多重继承. 为确保它们是纯接口, 这些类必须以 Interface 为后缀.

Note

关于该规则, Windows 下有个 特例.

2.8. 接口

Tip

接口是指满足特定条件的类, 这些类以 Interface 为后缀 (不强制).

定义:

当一个类满足以下要求时, 称之为纯接口:

• 只有纯虚函数 (“=0“) 和静态函数 (除了下文提到的析构函数).

• 没有非静态数据成员.

• 没有定义任何构造函数. 如果有, 也不能带有参数, 并且必须为 protected.

• 如果它是一个子类, 也只能从满足上述条件并以 Interface 为后缀的类继承.

接口类不能被直接实例化, 因为它声明了纯虚函数. 为确保接口类的所有实现可被正确销毁, 必须为之声明虚析构函数 (作为上述第 1 条规则的特例, 析构函数不能是纯虚函数). 具体细节可参考 Stroustrup 的 The C++ Programming Language, 3rd edition 第 12.4 节.

优点:

以 Interface 为后缀可以提醒其他人不要为该接口类增加函数实现或非静态数据成员. 这一点对于 多重继承 尤其重要. 另外, 对于 Java 程序员来说, 接口的概念已是深入人心.

缺点:

Interface 后缀增加了类名长度, 为阅读和理解带来不便. 同时，接口特性作为实现细节不应暴露给用户.

结论:

只有在满足上述需要时, 类才以 Interface 结尾, 但反过来, 满足上述需要的类未必一定以 Interface 结尾.

2.9. 运算符重载

Tip

除少数特定环境外，不要重载运算符.

定义:

一个类可以定义诸如 + 和 / 等运算符, 使其可以像内建类型一样直接操作.

优点:

使代码看上去更加直观, 类表现的和内建类型 (如 int) 行为一致. 重载运算符使 Equals(), Add() 等函数名黯然失色. 为了使一些模板函数正确工作, 你可能必须定义操作符.

缺点:

虽然操作符重载令代码更加直观, 但也有一些不足:

• 混淆视听, 让你误以为一些耗时的操作和操作内建类型一样轻巧.

• 更难定位重载运算符的调用点, 查找 Equals() 显然比对应的 == 调用点要容易的多.

• 有的运算符可以对指针进行操作, 容易导致 bug. Foo + 4 做的是一件事, 而 &Foo + 4 可能做的是完全不同的另一件事. 对于二者, 编译器都不会报错, 使其很难调试;

重载还有令你吃惊的副作用. 比如, 重载了 operator& 的类不能被前置声明.

结论:

一般不要重载运算符. 尤其是赋值操作 (operator=) 比较诡异, 应避免重载. 如果需要的话, 可以定义类似 Equals(), CopyFrom() 等函数.

然而, 极少数情况下可能需要重载运算符以便与模板或 “标准” C++ 类互操作 (如 operator<<(ostream&, const T&)). 只有被证明是完全合理的才能重载, 但你还是要尽可能避免这样做. 尤其是不要仅仅为了在 STL 容器中用作键值就重载 operator== 或 operator<; 相反, 你应该在声明容器的时候, 创建相等判断和大小比较的仿函数类型.

有些 STL 算法确实需要重载 operator== 时, 你可以这么做, 记得别忘了在文档中说明原因.

参考 拷贝构造函数 和 函数重载.

2.10. 存取控制

Tip

将 所有 数据成员声明为 private, 并根据需要提供相应的存取函数. 例如, 某个名为 foo\_ 的变量, 其取值函数是 foo(). 还可能需要一个赋值函数 set\_foo().

一般在头文件中把存取函数定义成内联函数.

参考 继承 和 函数命名

2.11. 声明顺序

Tip

在类中使用特定的声明顺序: public: 在 private: 之前, 成员函数在数据成员 (变量) 前;

类的访问控制区段的声明顺序依次为: public:, protected:, private:. 如果某区段没内容, 可以不声明.

每个区段内的声明通常按以下顺序:

• typedefs 和枚举

• 常量

• 构造函数

• 析构函数

• 成员函数, 含静态成员函数

• 数据成员, 含静态数据成员

宏 DISALLOW\_COPY\_AND\_ASSIGN 的调用放在 private: 区段的末尾. 它通常是类的最后部分. 参考 拷贝构造函数.

.cpp 文件中函数的定义应尽可能和声明顺序一致.

不要在类定义中内联大型函数. 通常, 只有那些没有特别意义或性能要求高, 并且是比较短小的函数才能被定义为内联函数. 更多细节参考 内联函数.

2.12. 编写简短函数

Tip

倾向编写简短, 凝练的函数.

我们承认长函数有时是合理的, 因此并不硬性限制函数的长度. 如果函数超过 40 行, 可以思索一下能不能在不影响程序结构的前提下对其进行分割.

即使一个长函数现在工作的非常好, 一旦有人对其修改, 有可能出现新的问题. 甚至导致难以发现的 bug. 使函数尽量简短, 便于他人阅读和修改代码.

在处理代码时, 你可能会发现复杂的长函数. 不要害怕修改现有代码: 如果证实这些代码使用 / 调试困难, 或者你需要使用其中的一小段代码, 考虑将其分割为更加简短并易于管理的若干函数.