C/C++语言编码规范

目录

[C/C++语言编码规范 1](#_Toc104712772)

[目录 2](#_Toc104712773)

[1. 命名规则 4](#_Toc104712774)

[1.1. 起个合适的名字 4](#_Toc104712775)

[1.1.1. 类的名称（适用于C++） 4](#_Toc104712776)

[1.1.2. 方法和函数的名称（适用于C/C++） 4](#_Toc104712777)

[1.1.3. 含有度量单位的名称（适用于C/C++） 4](#_Toc104712778)

[1.1.4. 缩写名称不要全部大写（适用于C/C++） 4](#_Toc104712779)

[1.2. 类的命名（适用于C++） 5](#_Toc104712780)

[1.3. 类库（或程序库）命名 （适用于C/C++） 5](#_Toc104712781)

[1.4. 方法和函数的命名（适用于C++） 5](#_Toc104712782)

[1.5. 类属性的命名（适用于C++） 5](#_Toc104712783)

[1.6. 方法和函数参数的命名（适用于C++） 6](#_Toc104712784)

[1.7. 局部变量的命名（适用于C/C++） 6](#_Toc104712785)

[1.8. 指针变量的命名前缀（适用于C/C++） 6](#_Toc104712786)

[1.9. 引用变量和返回引用函数的命名前缀（适用于C++） 7](#_Toc104712787)

[1.10. 全局变量的命名前缀（适用于C/C++） 7](#_Toc104712788)

[1.11. 全局常量的命名（适用于C/C++） 7](#_Toc104712789)

[1.12. 静态变量的命名前缀（适用于C++） 7](#_Toc104712790)

[1.13. 自定义类型（typedef）的命名（适用于C/C++） 7](#_Toc104712791)

[1.14. 宏定义的命名（适用于C/C++） 8](#_Toc104712792)

[1.15. C 函数的命名（适用于C/C++） 8](#_Toc104712793)

[1.16. 枚举的命名（适用于C/C++） 8](#_Toc104712794)

[2. 排版规则 10](#_Toc104712795)

[2.1. 布局和模板 10](#_Toc104712796)

[2.1.1. 类的布局模板 （适用于C++） 10](#_Toc104712797)

[2.1.2. 源文件格式（适用于C++） 11](#_Toc104712798)

[2.1.3. 保护头文件不被重复包含 （适用于C/C++） 12](#_Toc104712799)

[2.1.4. 方法和函数的布局 （适用于C/C++） 13](#_Toc104712800)

[2.2. 缩进、制表符以及空格 （适用于C/C++） 13](#_Toc104712801)

[2.3. 尽量使一行不要超过78个字母 （适用于C/C++） 13](#_Toc104712802)

[2.4. 保证一行只写一条语句 （适用于C/C++） 13](#_Toc104712803)

[2.5. 花括号 {} 规则 （适用于C/C++） 14](#_Toc104712804)

[2.5.1. 花括号的位置 14](#_Toc104712805)

[2.5.2. 什么时候应使用花括号 14](#_Toc104712806)

[2.5.3. 在花括号结束的位置加上注释 14](#_Toc104712807)

[2.5.4. 注意屏幕大小 15](#_Toc104712808)

[2.6. 圆括号 () 规则 （适用于C/C++） 15](#_Toc104712809)

[2.7. if else 语句的格式 （适用于C/C++） 15](#_Toc104712810)

[2.8. switch 格式 （适用于C/C++） 16](#_Toc104712811)

[2.9. 使用goto,continue,break 和 ?: （适用于C/C++） 16](#_Toc104712812)

[2.9.1. Goto 16](#_Toc104712813)

[2.9.2. Continue and Break 17](#_Toc104712814)

[2.9.3. ?: 17](#_Toc104712815)

[2.10. 运算符号的规则 （适用于C/C++） 17](#_Toc104712816)

[2.11. 变量声明语句块 （适用于C/C++） 17](#_Toc104712817)

[3. 文档及注释 19](#_Toc104712818)

[3.1. 文件或程序库的文档注释（适用于C/C++） 19](#_Toc104712819)

[3.2. 类文档注释（适用于C/C++） 19](#_Toc104712820)

[3.3. 函数文档注释（适用于C/C++） 19](#_Toc104712821)

[3.4. Include 语句注释 （适用于C/C++） 20](#_Toc104712822)

[3.5. 语句块注释（适用于C/C++） 20](#_Toc104712823)

[4. 编码要求 22](#_Toc104712824)

[4.1. 不要忽略编译器的警告（适用于C/C++） 22](#_Toc104712825)

[4.2. 应使用源代码管理器（适用于C/C++） 22](#_Toc104712826)

[4.3. 固有的类方法成员（适用于C++） 22](#_Toc104712827)

[4.4. 使用命名空间 （适用于C++） 22](#_Toc104712828)

[4.5. 初始化所有的变量 （适用于C/C++） 22](#_Toc104712829)

[4.6. 保持函数短小精悍（适用于C/C++） 22](#_Toc104712830)

[4.7. 对空语句进行注释 （适用于C/C++） 23](#_Toc104712831)

[4.8. 不要用if语句的默认方法测试非零值（适用于C/C++） 23](#_Toc104712832)

[4.9. 布尔类型 （适用于C/C++） 23](#_Toc104712833)

[4.10. 避免在语句中内含赋值 （适用于C/C++） 24](#_Toc104712834)

[4.11. 正确的使用Const （适用于C/C++） 24](#_Toc104712835)

[4.12. 不要在头文件定义数据 （适用于C/C++） 24](#_Toc104712836)

[4.13. 不要直接使用数字 （适用于C/C++） 24](#_Toc104712837)

[4.14. 宏（适用于C/C++） 25](#_Toc104712838)

# 命名规则

## 起个合适的名字

### 类的名称（适用于C++）

* 类的名称要能告诉我们，这个类是什么。因此，类的名称通常是名词。
* 类的名字不需要告诉我们，它从哪个类继承而来的。
* 有时候加个后缀是很有用的。比如类是一个代理(Agents)时，起名叫DownloadAgent更能表达真实的意图。

### 方法和函数的名称（适用于C/C++）

* 方法和函数通常都要执行某种行为，因此，名称要能清楚的说明它做什么：CheckForErrors() 而不是ErrorCheck(), DumpDataToFile() 而不是 DataFile()。 这样也可以很容易的区别函数和数据。
* 函数名总以动词开头，后面跟随其它名称。这样看起来更自然些。
* 可以加一些必要的后缀：

Max – 表示取最大值

Cnt – 表示当前的计数值

Key – 表示键值

例如：RetryMax 表示可接收的最大数，RetryCnt表示当前接收的数量。

* 前缀也同样有用：

Is – 用于询问一些问题。只要看到Is开头，就知道这是一个查询。

Get – 用于获取一个值。

Set – 用于设置一个值。

例如：IsHitRetryLimit.

### 含有度量单位的名称（适用于C/C++）

* 如果一个变量用于表示时间，重量或其它度量单位，应把度量单位添加到名称中，以便开发人员更早一步发现问题。

例如：

uint32 mTimeoutMsecs;

uint32 mMyWeightLbs;

### 缩写名称不要全部大写（适用于C/C++）

* 无论是什么缩写名称，我们总以一个大写字母开头，后面跟随的字母全部用小写。

例如：

class FluidOz; // 而不是 FluidOZ

class NetworkAbcKey; // 而不是 NetworkABCKey

## 类的命名（适用于C++）

* 用大写字母作为单词的分隔，每个单词的首字母大写，其它字母均小写。
* 名字的第一个字母应大写
* 不含有下划线 ('\_')

例如：

class NameOneTwo;

class Name;

## 类库（或程序库）命名 （适用于C/C++）

* 使用命名空间防止名字冲突。
* 如果编译器没有实现命名空间，需要用前缀来避名名字冲突，不过前缀不要过长（2个字母比较好）。

例如：

John Johnson 完成了一个数据结构的库，它可以使用JJ作为库的前缀，所以类名就象下面这样：

class JjLinkList

{

}

## 方法和函数的命名（适用于C++）

* 使用与类名相同的规则

例如：

class NameOneTwo

{

public:

int DoIt();

void HandleError();

}

## 类属性的命名（适用于C++）

* 属性（通常是非公有数据成员）名字以字母'm'开头。
* 在 'm' 后面，使用与类名相同的规则。
* 'm' 总是位于其它修饰符（如表示指针的 'p'）的前面。

例如：

class NameOneTwo

{

public:

int VarAbc();

int ErrorNumber();

private:

int mVarAbc;

int mErrorNumber;

String\* mpName;

}

## 方法和函数参数的命名（适用于C++）

* 第一个字母必须小写。
* 第一个字母后面的单词使用与类名相同的规则。

例如：

class NameOneTwo

{

public:

int StartYourEngines(

Engine& rSomeEngine,

Engine& rAnotherEngine);

}

## 局部变量的命名（适用于C/C++）

* 所有字母都用小写
* 使用下划线 '\_' 作为单词的分隔。

例如：

int

NameOneTwo::HandleError(int errorNumber)

{

int error= OsErr();

Time time\_of\_error;

ErrorProcessor error\_processor;

}

## 指针变量的命名前缀（适用于C/C++）

* 指针变量多数情况应在前面加 'p'。
* 星号 '\*' 应靠近类型，而不是变量名。

例如：

String\* pName= new String;

特别的：String\* pName, name; 应分成两行来写：

String\* pName;

String name;

## 引用变量和返回引用函数的命名前缀（适用于C++）

* 引用必须用 'r'作前缀修饰。

例如：

class Test

{

public:

void DoSomething(StatusInfo& rStatus);

StatusInfo& rStatus();

const StatusInfo& Status() const; // 这里返回的是常量引用，所以不符合本规则

private:

StatusInfo& mrStatus;

}

## 全局变量的命名前缀（适用于C/C++）

* 全局变量总是以 'g' 作为前缀。

例如：

Logger gLog;

Logger\* gpLog;

## 全局常量的命名（适用于C/C++）

* 全局常量全部大写，并以下划线 '\_' 分隔单词。

例如：

const int A\_GLOBAL\_CONSTANT= 5;

## 静态变量的命名前缀（适用于C++）

* 静态变量以 's' 作为前缀。

例如：

class Test

{

public:

private:

static StatusInfo msStatus;

}

## 自定义类型（typedef）的命名（适用于C/C++）

* 类型定义名称指的是用typedef定义的名称。
* 类型定义名称使用与类名相同的规则，并使用Type作为后缀。

例如：

typedef uint16 ModuleType;

typedef uint32 SystemType;

## 宏定义的命名（适用于C/C++）

* 所有单词的字母都用大写，并使用下划线 '\_' 分隔.

例如：

#define MAX(a,b) blah

#define IS\_ERR(err) blah

## C 函数的命名（适用于C/C++）

* C++项目中，应尽量少用C函数。
* C函数使用GNU规范，所有字母都使用小写，并用下划线 '\_' 作为单词的分隔。

例如：

int

some\_bloody\_function()

{

}

* 特别的，为了赚容C/C++，在必要的时候，在C++中应以下面的格式定义C函数：

extern “C” int some\_bloody\_function();

* 或在C/C++中推荐使用下面的格式：

#ifdef \_\_cplusplus\_\_

extern “C” {

#endif

int

some\_bloody\_function()

{

}

#ifdef \_\_cplusplus\_\_

}

#endif

## 枚举的命名（适用于C/C++）

* 所有字母都大写，并用下划线 '\_' 作为单词分隔。

例如：

enum PinStateType

{

PIN\_OFF,

PIN\_ON

};

enum { STATE\_ERR, STATE\_OPEN, STATE\_RUNNING, STATE\_DYING};

# 排版规则

## 布局和模板

### 类的布局模板 （适用于C++）

* 请使用下面的模板来创建一个新的类：

/\*\*

\* 用一行来描述类

\*

\* #include "XX.h" <BR>

\* -llib

\*

\* 类的详细说明

\*

\* @see something

\*/

#ifndef XX\_h

#define XX\_h

// 在这里包含系统头文件

//

// 在这里包含项目头文件

//

// 在这里包含局部头文件

//

// 在这里放置前置引用

//

class XX

{

public:

// 类的生命周期控制函数，如构造和析构，以及状态机

/\*\*

\* Default constructor.

\*/

XX(void);

/\*\*

\* Copy constructor.

\*

\* @param from The value to copy to this object.

\*/

XX(const XX& from);

/\*\*

\* Destructor.

\*/

~XX(void);

// 在这里放置类的运算操作符

/\*\*

\* Assignment operator.

\*

\* @param from THe value to assign to this object.

\*

\* @return A reference to this object.

\*/

XX& operator=(XX& from);

// 在这里放置类的操作

// 在这里放置属性存取

// 在这里放置类的状态查询

protected:

private:

};

// 内联方法定义

//

// 外部引用

//

#endif // \_XX\_h\_

* 定义的顺序是: public, protected, private
* 要清楚public/protected/private都应该放置哪些东西

### 源文件格式（适用于C++）

#include "XX.h" // class implemented

/////////////// PUBLIC ///////////////////////

//================= 构造函数 ====================

XX::XX()

{

}// XX

XX::XX(const XX&)

{

}// XX

XX::~XX()

{

}// ~XX

//=============== 操作符=========================

XX&

XX::operator=(XX&);

{

return \*this;

}// =

//==============类的操作 =====================

//==============属性存取 =====================

//==============状态查询 =====================

///////////// PROTECTED //////////////////

///////////// PRIVATE //////////////////

### 保护头文件不被重复包含 （适用于C/C++）

* 应使用宏定义来保护头文件不被重复包含：

#ifndef filename\_h

#define filename\_h

#endif // filename\_h

* 如果使用命名空间的时候，要把命名空间加到文件名前面：

#ifndef namespace\_filename\_h

#define namespace\_filename\_h

#endif

### 方法和函数的布局 （适用于C/C++）

* 对于有较多参数的函数的写法

如果参数较多，一行写不下，我们应该分成几行来写，并且每个参数都另起一行对齐：

int AnyMethod(

int arg1,

int arg2,

int arg3,

int arg4);

## 缩进、制表符以及空格 （适用于C/C++）

* 缩进的时候，每一层缩进3，4，或8个空格。（推荐使用4个空格）
* 不要使用TAB，用空格，大多数编辑器可以用空格代替TAB。TAB应固定8个空格，因为大多数编辑器都是这么设置的。
* 虽然没有规定缩进的层次，但是4至5层是合适的。如果缩进的层次太多，你可能需要考虑是否进行代码重构了。

例如：

void

func()

{

if (something bad)

{

if (another thing bad)

{

while (more input)

{

}

}

}

}

## 尽量使一行不要超过78个字母 （适用于C/C++）

* 有许多编辑器屏幕只有78个字母宽

## 保证一行只写一条语句 （适用于C/C++）

* 一行最多只写一条语句
* 一行只定义一个变量

例如：

不要象下面这样:

char \*\*a, \*x;

要象这样:

char\*\* a= 0; // 文档说明

char\* x= 0; // 文档说明

## 花括号 {} 规则 （适用于C/C++）

### 花括号的位置

* 在关键字的下一行单独放置括号，并且与关键字对齐，如：

if (condition)

{

...

}

while (condition)

{

…

}

### 什么时候应使用花括号

所有的 if, while 和 do 语句，要么用单行格式，要么使用花括号格式。

* 使用花括号格式：

if (1 == somevalue)

{

somevalue = 2;

}

* 单行格式：

if (1 == somevalue) somevalue = 2;

或下面这样（对于这种写法，建议使用花括号）：

if (1 == somevalue)   
somevalue = 2;

### 在花括号结束的位置加上注释

* 在花括号结束的位置加上注释是一个好习惯。假如前后花括号距离很远，注释就能帮你理解它是如何对应的。如：

while(1)

{

if (valid)

{

} // if valid

else

{

} // not valid

} // end forever

### 注意屏幕大小

* 一个语句块尽量不超过一个屏幕大小，这样，不要卷动屏幕就可以阅读代码。

## 圆括号 () 规则 （适用于C/C++）

* 圆括号与关键字之间应放一个空格。
* 圆括号与函数名之间不要有空格。
* Return 语句不要使用圆括号。

例如：

if (condition)

{

}

while (condition)

{

}

strcpy(s, s1);

return 1;

## if else 语句的格式 （适用于C/C++）

* 布局

if (条件) // 注释

{

}

else if (条件) // 注释

{

}

else // 注释

{

}

* 条件格式

总是把常量放在等号或不等于号的左边：

if ( 6 == errorNum ) ...

一个很重要的理由是，假如漏写一个等号，这种写法会产生一个编译错误，有助于马上发现问题。

比如：

if ( errorNum == 6) ...

错写成：

if ( errorNum = 6) ... // 这是一个不容易发现的灾难

## switch 格式 （适用于C/C++）

* 直通的case语句，应该放置一条注释说明这个case语句是直通到下一个case语句的。
* 总是要写default语句，不管是否是需要。
* 在case中需要定义变量的时候，应把所有代码放在语句块中。

例如：

switch (...)

{

case 1:

...

// 继续执行case2

case 2:

{

int v;

...

}

break;

default:

}

## 使用goto,continue,break 和 ?: （适用于C/C++）

### Goto

* 尽量避免使用Goto 语句。一个合理使用goto语句的场合是，当你需要从多层循环中跳出。例如：

for (...)

{

while (...)

{

...

if (disaster) goto error; // 跳出循环

}

}

...

error:

clean up the mess

* 跳转的标号必须单独在一行的最左边。Goto语句需要有相应的注释，说明它的用途。

### Continue and Break

* Continue 和break 实际上起到与goto一样的作用，因此，尽量少用为上。并且，Continue与break最好不要连用。

### ?:

* 用括号把条件表达式括起来。
* 不要在 ? : 中写上过多的代码，操作表达式应尽可能简洁。
* 操作语句应分行写，除非它们能够简洁的放在一行当中。

例如：

(condition) ? funct1() : func2();

或

(condition)

? long statement

: another long statement;

## 运算符号的规则 （适用于C/C++）

* 一元操作符如（!、~ 等等）应贴近操作对象。  
  如：

if (!IsOk)

return ++v;

* 二元操作符如（+、\*、%、== 等等）应在前后留空格。  
  如：

if ( v1 == v2)

return v1 \* 3;

* ++ 和 -- 尽量使用前置运算。在C++中，不管 ++i 还是 i++，总是++i更容易生成优化代码。  
  如：  
  for(int i = 0; i < 10; ++i)

## 变量声明语句块 （适用于C/C++）

* 变量应该是随用随声明，不要集中在函数前（有些C语言不支持，则不在此要求之列）。特别是在for语句的循环变量，应只在for语句中定义。  
  如：  
  for(int i = 0; i < 10; ++i)
* 声明语句块必须要对齐

类型，变量，等号和初始化值要分别对齐。

例如：

DWORD mDword;

DWORD\* mpDword;

char\* mpChar;

char mChar;

mDword = 0;

mpDword = NULL;

mpChar = NULL;

mChar = 0;

# 文档及注释

应当使用文档自动生成工具，来生成相关的程序文档。

## 文件或程序库的文档注释（适用于C/C++）

可以为整个文件编写文档。

例如：

/\*\* @file file.h

\* A brief file description.

\* A more elaborated file description.

\*/

## 类文档注释（适用于C/C++）

在类定义前面应加上类说明文档。

例如：

/\*\* WindowsNT

\* @brief Windows Nice Try.

\* @author Bill Gates

\* @author Several species of small furry animals gathered together

\* in a cave and grooving with a pict.

\* @version 4.0

\* @date 1996-1998

\* @bug It crashes a lot and requires huge amounts of memory.

\* @bug The class introduces the more bugs, the longer it is used.

\* @warning This class may explode in your face.

\* @warning If you inherit anything from this class, you're doomed.

\*/

class WindowsNT {};

## 函数文档注释（适用于C/C++）

* 函数注释

所有的参数都应该有文档说明(param)，所有的返回代码都应该有文档说明(return)，所有的例外都应该有文档说明(exception)。可以使用(see)引用有关的开发资源。如：

/\*\*

\* 赋值操作符

\*

\* @param val 将要赋给本对象的值

\*

\* @return 本对象的引用

\*/

XX& operator = (XX& val);

* 注释属性

一些自动文档工具定义的属性可以包含在文档中，常用的有：

* + 前提条件 (pre)  
    定义调用这个函数的前提条件
  + 警告说明 (warning)  
    定义一些关于这个函数必须知道的事情。
  + 备注说明 (remarks)  
    定义一些关于这个函数的备注信息。
  + 将要完成的工作 (todo)  
    说明哪些事情将在不久以后完成
  + 使用例子说明 (example)  
    一个图片能表达100句话，一个好的例子能解答1000个问题。

例如：

/\*\*

\* 复制一个字串

\*

\* @pre

\* - 需要保证(from != 0)

\* - 需要保证(to != 0)

\*

\* @warning

\* 缓冲区必需足够大，以便容纳的下要拷贝的字串。

\*

\* @example teststrcpy.cpp

\*

\* @param from 要拷贝的字串

\* @param to 用于容纳字串的缓冲区

\* @return void

\*/

void strcpy(const char\* from, char\* to);

## Include 语句注释 （适用于C/C++）

* 如果有必要，＃include语句也应有注释，它可以告诉我们，为什么要包含这个头文件。

## 语句块注释（适用于C/C++）

* 语句块的注释可以用在语句块的开头和结束位置：

{

// Block1 (meaningful comment about Block1)

... some code

{

// Block2 (meaningful comment about Block2)

... some code

} // End Block2

} // End Block1

# 编码要求

## 不要忽略编译器的警告（适用于C/C++）

* 编译器的警告，通常能够指示出编码存在的笔误或逻辑错误。因此，不能轻视编译器的任何警告。正确的作法是，不允许代码在编译时产生任何警告信息。

## 应使用源代码管理器（适用于C/C++）

* 根据开发规模，选择合适的源代码管理器。使用源代码管理器是非常必要的。

## 固有的类方法成员（适用于C++）

* 默认构造函数(Default Constructor)

如果构造函数的所有参数都是可选的，那么这个构造函数也是默认构造函数。如果没有定义任何普通构造函数，则编译将自动生成一个。

* 虚析构函数(Virtual Destructor)

如果一个类可以被继承，那么应该使用虚析构函数。如果没有定义虚析构函数，则编译器将自动生成一个。

* 拷贝构造函数(Copy Constructor)

如果一个类不应该被拷贝，应该定义一个私有的拷贝构造函数，并且不定义它的实现。如果不知道一个类是否应该被拷贝，就认为它是不可拷贝的，直到你确认它应该被拷贝。如果没有定义拷贝构造函数，则编译器将自动生成一个。

* 赋值操作(Assignment Operator)

如果一个类不应该被赋值，应该定义一个私有的赋值操作函数，并且不定义它的实现。如果不知道一个类是否应该被赋值，就认为它是不可赋值的，直到你确认它应该被赋值。如果没有定义赋值操作函数，则编译器将自动生成一个。

## 使用命名空间 （适用于C++）

* 命名规则

根名字一般是设计者的名字。比如公司名称等等。

* 不要在全局空间使用using语句。

## 初始化所有的变量 （适用于C/C++）

* 无论如何，都要初始化所有的变量。我们无法保证编译器会给个什么样的初值。

## 保持函数短小精悍（适用于C/C++）

* 一般情况下，一个函数最好在一个屏幕内，不要超过三个屏幕。

## 对空语句进行注释 （适用于C/C++）

* For和while语句如果跟随一个空语句，需要对此语句进行注释，并且空语句应另起一行。如：

while (\*dest++ = \*src++)

; // VOID

* 不允许写成：

while (\*dest++ = \*src++) ; // 绝对不允许这么写

## 不要用if语句的默认方法测试非零值（适用于C/C++）

* If语句只用于检测布尔值(bool)，不要用默认的方法测试非零值，比如：

建议使用：

if (FAIL != f())

不建议使用下面的表达式：

if (f())

* 宏定义的情况也一样：

#define STREQ(a, b) (strcmp((a), (b)) == 0)

或者使用内联函数：

inline bool

StringEqual(char\* a, char\* b)

{

(strcmp(a, b) == 0) ? return true : return false;

Or more compactly:

return strcmp(a, b) == 0;

}

## 布尔类型 （适用于C/C++）

* 早期的C++没有布尔类型，但新的C++标准增加了布尔类型。如果可以使用内置的布尔类型的情况下，应使用布尔类型。

早期的布尔类型定义为：

typedef int bool;

#define TRUE 1

#define FALSE 0

或：

const int TRUE = 1;

const int FALSE = 0;

* 在这种情况下，条件表达式不要比较1值(如TRUE，YES等等)，而要用0值(如FALSE,NO等等)进行比较。因为多数函数返回0表示FALSE，而非零表示TRUE。如：

if (TRUE == func()) { ... // 错误：假如func()返回 2 怎么办？

必须写成：

if (FALSE != func()) { ...

## 避免在语句中内含赋值 （适用于C/C++）

* 只有一种情况可以在语句中内含赋值，它要能使代码显得更易理解，例如：

while (EOF != (c = getchar()))

{

process the character

}

* ++ 和 -- 操作也是一种赋值语句
* 内含赋值语句常常会带来一些副作用。在遇到这种情况时，我们应分成几个语句来写。比如：

a = b + c;

d = a + r;

不应该写成：

d = (a = b + c) + r;

## 正确的使用Const （适用于C/C++）

* C/C++ 提供const 关键字，用于指示不应该被修改的对象或数据。正确的使用Const既可以提供编译器的优化指示，也能够避免一些编码错误。

## 不要在头文件定义数据 （适用于C/C++）

不要把数据定义放在头文件，如：

/\*

\* aheader.h

\*/

int x = 0;

## 不要直接使用数字 （适用于C/C++）

* 直接使用数字，会使源代码难以理解和维护。如：

if (22 == foo) { start\_thermo\_nuclear\_war(); }

else if (19 == foo) { refund\_lotso\_money(); }

else if (16 == foo) { infinite\_loop(); }

else { cry\_cause\_im\_lost(); }

当一段时间过去以后，有谁会记得22和19是什么意思？假如数字改变，或者是编写错误，更是难以发现问题。

* 我们可以用#define或者常量来改变这一状况，如：

#define PRESIDENT\_WENT\_CRAZY (22)

const int WE\_GOOFED= 19;

enum

{

THEY\_DIDNT\_PAY= 16

};

if (PRESIDENT\_WENT\_CRAZY == foo) { start\_thermo\_nuclear\_war(); }

else if (WE\_GOOFED == foo) { refund\_lotso\_money(); }

else if (THEY\_DIDNT\_PAY == foo) { infinite\_loop(); }

else { happy\_days\_i\_know\_why\_im\_here(); }

## 宏（适用于C/C++）

* 如果可以，使用内联函数代替宏。

例如：

#define MAX(x,y) (((x) > (y) ? (x) : (y)) // 取最大数

使用内联函数可以达到相同的效果，而且更安全：

inline int

max(int x, int y)

{

return (x > y ? x : y);

}

* 要注意副作用

必须小心副作用，因为在调用表达式时，会发生潜在的错误。

例如：

MAX(f(x),z++);

* 表达式总是用括号括起来

在宏展开时，使用括号可以避免宏展开后产生的二义性。

例如：

#define ADD(x,y) x + y

必须写成：

#define ADD(x,y) ((x) + (y))

* 保证宏名称的唯一性

和全局变量一样，宏也会与其它名称产生冲突。下面两条规则有助于解决这个问题：

* + 在宏名称前加上库的名字

避免使用简单而常用的名字，如：MAX 和MIN。

**using** System;  
**namespace** runoob  
{  
   **public** **abstract** **class** Person  
   {  
      **public** **abstract** **string** Name  
      {  
         **get**;  
         **set**;  
      }  
      **public** **abstract** **int** Age  
      {  
         **get**;  
         **set**;  
      }  
   }  
   **class** Student : Person  
   {  
  
      **private** **string** code = "N.A";  
      **private** **string** name = "N.A";  
      **private** **int** age = 0;  
  
      *// 声明类型为 string 的 Code 属性*  
      **public** **string** Code  
      {  
         **get**  
         {  
            **return** code;  
         }  
         **set**  
         {  
            code = **value**;  
         }  
      }  
     
      *// 声明类型为 string 的 Name 属性*  
      **public** **override** **string** Name  
      {  
         **get**  
         {  
            **return** name;  
         }  
         **set**  
         {  
            name = **value**;  
         }  
      }  
  
      *// 声明类型为 int 的 Age 属性*  
      **public** **override** **int** Age  
      {  
         **get**  
         {  
            **return** age;  
         }  
         **set**  
         {  
            age = **value**;  
         }  
      }  
      **public** **override** **string** ToString()  
      {  
         **return** "Code = " + Code +", Name = " + Name + ", Age = " + Age;  
      }  
   }  
   **class** ExampleDemo  
   {  
      **public** **static** **void** Main()  
      {  
         *// 创建一个新的 Student 对象*  
         Student s = new Student();  
              
         *// 设置 student 的 code、name 和 age*  
         s.Code = "001";  
         s.Name = "Zara";  
         s.Age = 9;  
         Console.WriteLine("Student Info:- {0}", s);  
         *// 增加年龄*  
         s.Age += 1;  
         Console.WriteLine("Student Info:- {0}", s);  
         Console.ReadKey();  
       }  
   }  
}