

A decorative graphic on the right side of the page. It features three concentric blue circles of different sizes, each with a lighter blue outer ring. These circles are connected by thin blue lines that extend towards the top-left corner of the page. The circles are positioned in the upper right, middle right, and bottom right areas.

一站式示例代码库编程规范

作者：葛佳亮

翻译：蒋里京

本文档描述了微软一站式代码示例库项目组所采纳的关于本地 C++ 和 .NET（C# 和 VB.NET）代码的编程风格指导规范。

鸣谢

本文档的每一章节都应该感谢 **Dan Ruder** - 微软 Principal Escalation Engineer 。 Dan 对本文档进行了斟字酌句的查阅，并根据其 20 余年的编程经验提供了大量的珍贵评论 。我很荣幸能与他共事。

我同样感谢微软的四位经理，感谢他们对该项目的一贯支持。他们是 **Vivian Luo**，**Allen Ding**，**Felix Wu** 和 **Mei Liang**。

同时，如果没有如下一站式代码示例库项目的关键成员的辛勤付出，本文档必定不会具有现在的技术深度以及完整度，我在这里要感谢他们：

Hongye Sun	Jie Wang	Ji Zhou	Michael Sun	Kira Qian	Linda Liu
Allen Chen	Yi-Lun Luo	Steven Cheng	Wen-Jun Zhang		

本文档部分章节参考自一些微软产品组的编程规范。感谢他们的慷慨共享。

本编程规范在不断改善。如果您发现某些最佳实践或者话题并没有涵盖在本文档中，请告知我们[一站式示例代码库项目组](#)，以不断充实改善本文档。我期待着您的参与。☺

声明

本编程规范文档以“如是”提供，无论明示或暗示都不包含任何形式保证，但并不限制适用于特殊目的的默认担保。

当您编写 VC++/VC#/VB.NET 代码时，敬请参考或使用本文档。但是，我们希望您能通过 oncode@microsoft.com 告知我们您正在使用本文档，或给出任何改进建议。

目录

1	概览	1
1.1	原则和主旨	1
1.2	术语	2
2	通用编程规范	3
2.1	明确和一致	3
2.2	格式和风格	3
2.3	库的使用	5
2.4	全局变量	5
2.5	变量的声明和初始化	6
2.6	函数的声明和调用	7
2.7	语句	8
2.8	枚举	9
2.9	空格	14
2.10	大括号	15
2.11	注释	17
2.12	代码块	26
3	C++ 编程规范	27
3.1	编译器选项	27
3.2	文件和结构	29
3.3	命名规范	30
3.4	指针	34
3.5	常量	34
3.6	类型转换	35
3.7	Sizeof	36
3.8	字符串	37
3.9	数组	38
3.10	宏	39
3.11	函数	40
3.12	结构体	43
3.13	类	错误!未定义书签。
3.14	COM	50
3.15	动态分配	51

3.16	错误和异常	52
3.17	资源清理	55
3.18	控制流	58
4	.NET 编码规范	61
4.1	类库开发设计规范	61
4.2	文件和结构	61
4.3	程序集属性	61
4.4	命名规范	61
4.5	常量	65
4.6	字符串	65
4.7	数组和集合	67
4.8	结构体	69
4.9	类	70
4.10	命名空间	74
4.11	错误和异常	74
4.12	资源清理	77
4.13	交互操作	90

1 概览

本文档为[一站式示例代码库](#)项目组所使用的 C++ 以及 .NET 编码规范。该规范源自于产品开发过程中的经验，并在不断完善。如果您发现一些最佳实践或者话题并没有涵盖在本文档中，请联系我们[一站式示例代码库项目组](#)，以不断充实完善本文档。

任何指导准则都可能会众口难调。本规范的目的在于帮助社区开发者提高开发效率，减少代码中可能出现的 bug，并增强代码的可维护性。万事开头难，采纳一个不熟悉的规范可能在初期会有一些棘手和困扰，但是这些不适应很快便会消失，它所带来的好处和优势很快便会显现，特别是在当您接手他人代码时。

1.1 原则和主旨

高质量的代码示例往往具有如下特质：

1. **易懂** – 代码示例必须易读且简单明确。它们必须能展示出重点所在。示例代码的相关部分应当易于重用。示例代码不可包含多余代码。它们必须带有相应文档说明。
2. **正确性** – 示例代码必须正确展示出其欲告知使用者的重点。代码必须经过测试，且可以按照文档描述进行编译和运行。
3. **一致性** – 示例代码应该按照一致的编程风格和设计来保证代码易读。同样的，不同代码示例之间也应当保持一致的风格和设计，使使用者能够很轻松的结合使用它们。一致性将我们一站式示例代码库优良的品质形象传递给使用者，展示出我们对于细节的追求。
4. **流行性** – 代码示例应当展示现行的编程实践，例如使用 Unicode，错误处理，防御式编程以及可移植性。示例代码应当使用当前推荐的运行时库和 API 函数，以及推荐的项目和生成设置。

5. **可靠性** – 代码示例必须符合法律，隐私和政策标准和规范。不允许展示入侵性或低质的编程实践，不允许永久改变机器状态。所有的安装和执行过程必须可以被撤销。

6. **安全性** - 示例代码应该展示如何使用安全的编程实践：例如最低权限原则，使用运行时库函数的安全版本，以及 SDL 推荐的项目设置。

合理使用编程实践，设计和语言特性决定了示例代码是否可以很好满足上述特性。本编程规范致力于帮助您创建代码示例以使使用者能够作为最佳实践来效仿和学习。

1.2 术语

在整个文档中，会有一些对于标准和实践的推荐和建议。一些实践是非常重要的，必须严格执行，另一些指导准则并不一定处处适用，但是会在特定的场景下带来益处。为了清楚陈述规范和实践的意图，我们会使用如下术语。

术语	意图	理由
☑ 一定请...	该规范或实践在任何情况下都应该遵守。如果您认为您的应用是例外，则可能不适用。	该规范用于减少 bug。
☒ 一定不要...	不允许应用该规范或实践。	
☑ 您应该...	该规范和实践适用于大多数情况。	该规范用于统一编程风格，保持一致和清晰的风格。
☒ 您不应该..	不应该应用该规范或实践，除非有合理的理由。	
☑ 您可以...	该标准和规范您可以按需应用。	该规范可用于编程风格，但不总是有益的。

2 通用编程规范

这些通用编程规范适用于所有语言 – 它们对代码风格，格式和结构提供了全局通用的指导。

2.1 明确性和一致性

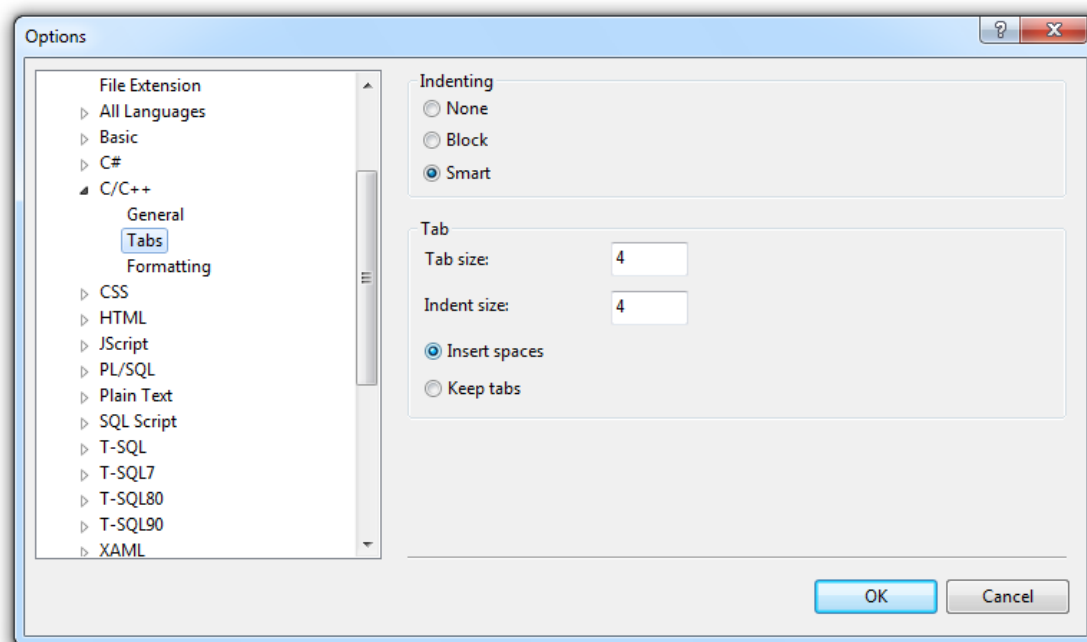
✅ **一定请** 确保代码的明确性，易读性和透明性。编程规范致力于确保代码是易懂和易维护的。没有什么胜于清晰、简洁、自描述的代码。

✅ **一定请** 确保 一旦应用了某编程规范，需在所有代码中应用，以保持一致性。

2.2 格式和风格

❌ **一定不要** 使用制表符。不同的文字编辑器使用不同的空格来生成制表符，这就带来了格式混乱。所有代码都应该使用 4 个空格来表示缩进。

可以配置 Visual Studio 文字编辑器，以空格代替制表符。



☑ 您应该 限制一行代码的最大长度。过长的代码降低了代码易读性。为了提高易读性，将代码长度设置为 78 列。若 78 列太窄，可以设置为 86 或者 90。

Visual C++ sample:

```
// Get and display whether the primary access token of the process
// belongs to user account that is a member of the local Administrators
// group even if it currently is not elevated (IsUserInAdminGroup).
HWND hInAdminGroupLabel = GetDlgItem(hWnd, IDC_INADMINGROUP_STATIC);
try
{
    BOOL const fInAdminGroup = IsUserInAdminGroup();
    SetWindowText(hInAdminGroupLabel, fInAdminGroup ? L"True" : L"False");
}
catch (DWORD dwError)
{
    SetWindowText(hInAdminGroupLabel, L"N/A");
    ReportError(L"IsUserInAdminGroup", dwError);
}
```

Column 78

Visual C# 示例:

```
// Get and display whether the primary access token of the process belongs
// to user account that is a member of the local Administrators group even
// if it currently is not elevated (IsUserInAdminGroup).
try
{
    bool fInAdminGroup = IsUserInAdminGroup();
    this.lbInAdminGroup.Text = fInAdminGroup.ToString();
}
catch (Exception ex)
{
    this.lbInAdminGroup.Text = "N/A";
    MessageBox.Show(ex.Message, "An error occurred in IsUserInAdminGroup",
        MessageBoxButtons.OK, MessageBoxIcon.Error);
}
```

Column 86

Visual Basic sample:

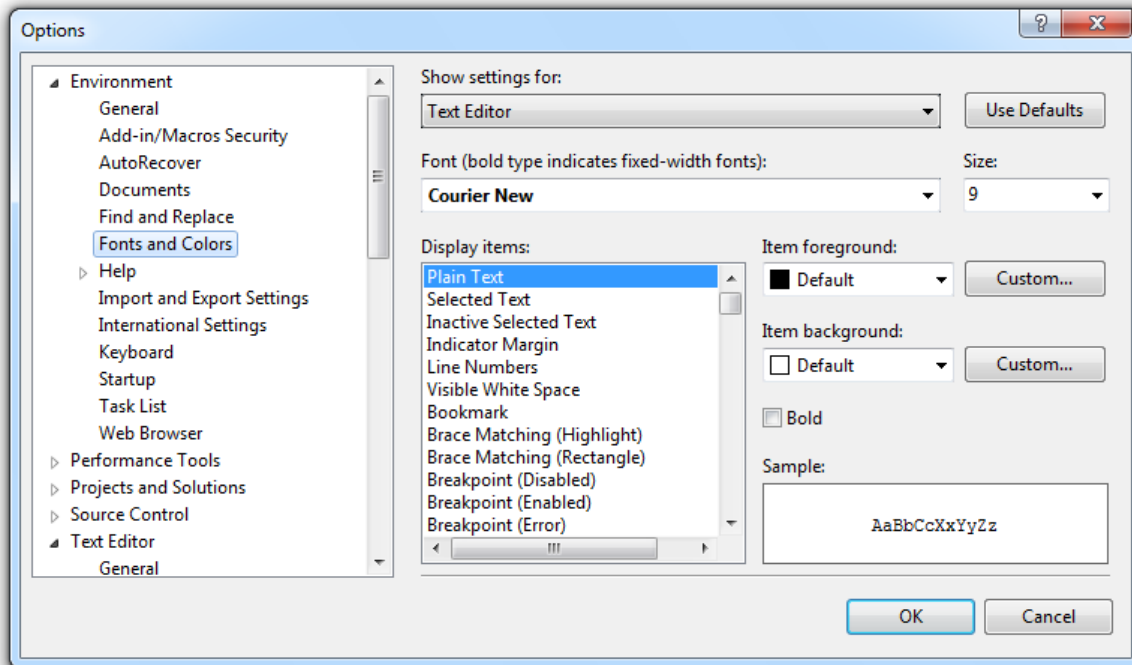

```

' Get and display whether the primary access token of the process belongs
' to user account that is a member of the local Administrators group even
' if it currently is not elevated (IsUserInAdminGroup).
Try
    Dim fInAdminGroup As Boolean = Me.IsUserInAdminGroup
    Me.lbInAdminGroup.Text = fInAdminGroup.ToString
Catch ex As Exception
    Me.lbInAdminGroup.Text = "N/A"
    MessageBox.Show(ex.Message, "An error occurred in IsUserInAdminGroup",
        MessageBoxButtons.OK, MessageBoxIcon.Error)
End Try

```

Column 86

☑ 一定请在您的代码编辑器中使用定宽字体，例如 Courier New。



2.3 库的使用

☒ 一定不要引用不必要的库，包括不必要的头文件，或引用不必要的程序集。注重细节能够减少项目生成时间，最小化出错几率，并给读者一个良好的印象。

2.4 全局变量

☑ 一定请尽量少用全局变量。为了正确的使用全局变量，一般是将它们作为参数传入函数。永远不要在函数或类内部直接引用全局变量，因为这会引起一个副作用：在调用者不知情的情况下改变了全局变量

的状态。这对于静态变量同样适用。如果您需要修改全局变量，您应该将其作为一个输出参数，或返回其一份全局变量的拷贝。

2.5 变量的声明和初始化

☑ **一定请** 在最小的，包含该局部变量的作用域块内声明它。一般，如果语言允许，就仅在使用前声明它们，否则就在作用域块的顶端声明。

☑ **一定请** 在声明变量时初始化它们。

☑ **一定请** 在语言允许的情况下，将局部变量的声明和初始化或赋值置于同一行代码内。这减少了代码的垂直空间，确保了变量不会处在未初始化的状态。

```
// C++ sample:
HANDLE hToken = NULL;
PSID pIntegritySid = NULL;
STARTUPINFO si = { sizeof(si) };
PROCESS_INFORMATION pi = { 0 };

// C# sample:
string name = myObject.Name;
int val = time.Hours;

' VB.NET sample:
Dim name As String = myObject.Name
Dim val As Integer = time.Hours
```

☒ **一定不要** 在同一行中声明多个变量。推荐每行只包含一句声明，这样有利于添加注释，也减少歧义。

例如 Visual C++ 示例，

Good:

```
CodeExample *pFirst = NULL; // Pointer of the first element.
CodeExample *pSecond = NULL; // Pointer of the second element.
```

Bad:

```
CodeExample *pFirst, *pSecond;
```

后一个代码示例经常被误写为:

```
CodeExample *pFirst, pSecond;
```

这种误写实际上等同于:

```
CodeExample *pFirst;
CodeExample pSecond;
```

2.6 函数的声明和调用

函数或方法的名称, 返回值, 参数列表可以有多种形式。原则上应该都将这些置于同一行代码内。如果带有过多参数不能置于一行代码, 可以进行换行: 多个参数一行或者一个参数一行。将返回值置于函数或方法名称的同一行。例如,

单行格式:

```
// C++ function declaration sample:
HRESULT DoSomeFunctionCall (int param1, int param2, int *param3);
// C++ / C# function call sample:
hr = DoSomeFunctionCall (param1, param2, param3);
' VB.NET function call sample:
hr = DoSomeFunctionCall (param1, param2, param3)
```

多行格式:

```
// C++ function declaration sample:
HRESULT DoSomeFunctionCall (int param1, int param2, int *param3,
    int param4, int param5);
// C++ / C# function call sample:
hr = DoSomeFunctionCall (param1, param2, param3,
    param4, param5);
' VB.NET function call sample:
hr = DoSomeFunctionCall (param1, param2, param3, _
    param4, param5)
```

将参数列表置于多行代码时, 每一个参数应该整齐排列于前一个参数的下方。第一个类型/参数对置于新行行首, 并缩进一个制表符宽度。函数或方法调用时的参数列表同样需按照这一格式。

```
// C++ sample:
HRESULT DoSomeFunctionCall (
```

```

    HWND hwnd,      // You can comment parameters, too
    T1 param1,      // Indicates something
    T2 param2,      // Indicates something else
    T3 param3,      // Indicates more
    T4 param4,      // Indicates even more
    T5 param5);    // You get the idea

// C++ / C# sample:
hr = DoSomeFunctionCall (
    hwnd,
    param1,
    param2,
    param3,
    param4,
    param5);

' VB.NET sample:
hr = DoSomeFunctionCall ( _
    hwnd, _
    param1, _
    param2, _
    param3, _
    param4, _
    param5)

```

☑ **一定请** 将参数排序，并首先将输入参数分组，再将输出参数放置最后。在参数组内，按照能够帮助程序员输入正确值的原则来将参数排序。比如，如果一个函数带有 2 个参数，“left”和“right”，将“left”置于“right”之前，则它们的放置顺序符合其参数名。当设计一系列具有相同参数的函数时，在各函数内使用一致的顺序。比如，如果一个函数带有一个输入类型为句柄的参数作为第一参数，那么所有相关函数都应该将该输入句柄作为第一参数。

2.7 代码语句

☒ **一定不要** 在同一行内放置一句以上的代码语句。这会使得调试器的单步调试变得更为困难。

```

Good:
// C++ / C# sample:
a = 1;
b = 2;

' VB.NET sample:

```

```
If (IsAdministrator ()) Then
    Console.WriteLine ("YES")
End If
```

Bad:

```
// C++ / C# sample:
a = 1; b = 2;

' VB.NET sample:
If (IsAdministrator ()) Then Console.WriteLine ("YES")
```

2.8 枚举

☑ **一定请** 将代表某些值集合的强类型参数，属性和返回值声明为枚举类型。

☑ **一定请** 在合适的情况下尽量使用枚举类型，而不是静态常量或“#define”值。枚举类型是一个具有一个静态常量集合的结构体。如果遵守这些规范，定义枚举类型，而不是带有静态常量的结构体，您便会得到额外的编译器和反射支持。

Good:

```
// C++ sample:
enum Color
{
    Red,
    Green,
    Blue
};

// C# sample:
public enum Color
{
    Red,
    Green,
    Blue
}

' VB.NET sample:
Public Enum Color
    Red
    Green
    Blue
End Enum
```

Bad:

```
// C++ sample:
const int RED    = 0;
const int GREEN  = 1;
const int BLUE   = 2;

#define RED      0
#define GREEN    1
#define BLUE     2

// C# sample:
public static class Color
{
    public const int Red = 0;
    public const int Green = 1;
    public const int Blue = 2;
}

' VB.NET sample:
Public Class Color
    Public Const Red As Integer = 0
    Public Const Green As Integer = 1
    Public Const Blue As Integer = 2
End Class
```

❌ **一定不要** 使用公开集合作为枚举（例如操作系统版本，您亲朋的姓名）。

✅ **一定请** 为简单枚举提供一个 0 值枚举量，可以考虑将之命名为“None”。如果这个名称对于特定的枚举并不合适，可以自行定义为更准确的名称。

```
// C++ sample:
enum Compression
{
    None = 0,
    GZip,
    Deflate
};

// C# sample:
public enum Compression
{
    None = 0,
    GZip,
    Deflate
}
```

```

}

' VB.NET sample:
Public Enum Compression
    None = 0
    GZip
    Deflate
End Enum

```

❌ **一定不要** 在 .NET 中使用 `Enum.IsDefined` 来检查枚举范围。 `Enum.IsDefined` 有 2 个问题。首先，它加载反射和大量类型元数据，代价极其昂贵。第二，它存在版本的问题。

```

Good:
// C# sample:
if (c > Color.Black || c < Color.White)
{
    throw new ArgumentOutOfRangeException (...);
}

' VB.NET sample:
If (c > Color.Black Or c < Color.White) Then
    Throw New ArgumentOutOfRangeException (...);
End If

Bad:
// C# sample:
if (!Enum.IsDefined (typeof (Color) , c) )
{
    throw new InvalidEnumArgumentException (...);
}

' VB.NET sample:
If Not [Enum].IsDefined (GetType (Color) , c) Then
    Throw New ArgumentOutOfRangeException (...);
End If

```

2.8.1 标志枚举

标志枚举用于对枚举值进行位运算的支持。标志枚举通常用于表示选项。

✅ **一定要** 将 `System.FlagsAttribute` 应用于标志枚举。 **一定不要** 将此属性用于简单枚举。

☑ **一定请** 利用 2 进制强大的能力，因为它可以自由的进行位异或运算。举例，

```
// C++ sample:
enum AttributeTargets
{
    Assembly = 0x0001,
    Class    = 0x0002,
    Struct   = 0x0004
    ...
};

// C# sample:
[Flags]
public enum AttributeTargets
{
    Assembly = 0x0001,
    Class    = 0x0002,
    Struct   = 0x0004,
    ...
}

' VB.NET sample:
<Flags()> _
Public Enum AttributeTargets
    Assembly = &H1
    Class    = &H2
    Struct   = &H4
    ...
End Enum
```

☑ **您应该** 提供一些特殊的枚举值，以便进行常见的标志枚举的组合运算。位运算属于高级任务，所以在简单任务中无需使用它们。`FileAccess.ReadWrite` 便是标志枚举特殊值的一个示例。然而，如果一个标志枚举中的某些值组合起来是非法的，您就不应该创建这样的标志枚举。

```
// C++ sample:
enum FileAccess
{
    Read = 0x1,
    Write = 0x2,
    ReadWrite = Read | Write
};

// C# sample:
```



```
[Flags]
public enum FileAccess
{
    Read = 0x1,
    Write = 0x2,
    ReadWrite = Read | Write
}

' VB.NET sample:
<Flags () > _
Public Enum FileAccess
    Read = &H1
    Write = &H2
    ReadWrite = Read Or Write
End Enum
```

❌ **您不应该** 在标志枚举中使用 0 值，除非它代表“所有标志被清除了”，并被恰当的命名为类似“None”的名字。如下 C# 示例展示了一常见的检查某一标志是否被设置了的实现（查看如下 if-语句）。该检查运行结果正确，但是有一处例外，便是对于 0 值的检查，它的布尔表达式结果恒为 true。

Bad:

```
[Flags]
public enum SomeFlag
{
    ValueA = 0, // This might be confusing to users
    ValueB = 1,
    ValueC = 2,
    ValueBAndC = ValueB | ValueC,
}

SomeFlag flags = GetValue();
if ((flags & SomeFlag.ValueA) == SomeFlag.ValueA)
{
    ...
}
```

Good:

```
[Flags]
public enum BorderStyle
{
    Fixed3D          = 0x1,
    FixedSingle      = 0x2,
    None             = 0x0
}

if (foo.BorderStyle == BorderStyle.None)
```

```
{
    ...
}
```

2.9 空格

2.9.1 空行

☑ **您应该**使用空行来分隔相关语句块。省略额外的空行会加大代码阅读难度。比如，您可以在变量声明和代码之间有一行空行。

Good:

```
// C++ sample:
void ProcessItem(const Item& item)
{
    int counter = 0;

    if (...)
    {
    }
}
```

Bad:

```
// C++ sample:
void ProcessItem(const Item& item)
{
    int counter = 0;

    // Implementation starts here
    //
    if (...)
    {
    }

}
```

在本例中，过多的空行造成了空行滥用，并不能使代码更易于阅读。

☑ **您应该**使用 2 行空行来分隔方法实现或类型声明。

2.9.2 空格

空格通过降低代码密度以增加可读性。以下是使用空格符的一些指导规范:

☑ 您应该 像如下般在一行代码中使用空格。

Good:

```
// C++ / C# sample:
CreateFoo ();           // No space between function name and parenthesis
Method (myChar, 0, 1);  // Single space after a comma
x = array[index];       // No spaces inside brackets
while (x == y)          // Single space before flow control statements
if (x == y)             // Single space separates operators
```

' VB.NET sample:

```
CreateFoo ()           ' No space between function name and parenthesis
Method (myChar, 0, 1)  ' Single space after a comma
x = array (index)      ' No spaces inside brackets
While (x = y)          ' Single space before flow control statements
If (x = y) Then        ' Single space separates operators
```

Bad:

```
// C++ / C# sample:
CreateFoo ( );         // Space between function name and parenthesis
Method (myChar,0,1);   // No spaces after commas
CreateFoo ( myChar, 0, 1 ); // Space before first arg, after last arg
x = array[ index ];    // Spaces inside brackets
while (x == y)         // No space before flow control statements
if (x==y)              // No space separates operators
```

' VB.NET sample:

```
CreateFoo ( )          ' Space between function name and parenthesis
Method (myChar,0,1)     ' No spaces after commas
CreateFoo ( myChar, 0, 1 ) ' Space before first arg, after last arg
x = array ( index )     ' Spaces inside brackets
While (x = y)           ' No space before flow control statements
If (x=y) Then           ' No space separates operators
```

2.10 大括号

☑ 一定请在 [一站式示例代码库](#) 的代码示例中使用 Allman 风格的大括号用法。

Allman 风格是以 Eric Allman 命名的，有时也被称为"ANSI 风格"。该风格将大括号与相关代码置于下一行内，与控制语句的缩进相同。大括号内的语句缩进一个等级。

Good:

```
// C++ / C# sample:
if (x > 5)
{
    y = 0;
}
```

```
' VB.NET sample:
If (x > 5) Then
    y = 0
End If
```

Bad (in All-In-One Code Framework samples) :

```
// C++ / C# sample:
if (x > 5) {
    y = 0;
}
```

☑ 您应该在即使是单行条件式的情况下也使用大括号。这样做使得将来增加条件式更简便，并减少制表符引起的歧义。

Good:

```
// C++ / C# sample:
if (x > 5)
{
    y = 0;
}
```

```
' VB.NET sample:
If (x > 5) Then
    y = 0
End If
```

Bad:

```
// C++ / C# sample:
if (x > 5) y = 0;

' VB.NET sample:
If (x > 5) Then y = 0
```

2.11 注释

- ☑ 您应该使用注释来解释一段代码的设计意图。一定不要 让注释仅仅是重复代码。

Good:

```
// Determine whether system is running Windows Vista or later operating
// systems (major version >= 6) because they support linked tokens, but
// previous versions (major version < 6) do not.
```

Bad:

```
// The following code sets the variable i to the starting value of the
// array. Then it loops through each item in the array.
```

- ☑ 您应该使用 `//` 注释方式，而不是 `/* */` 来为 C++ 和 C# 代码作注释。即使注释跨越多行代码，单行注释语法 (`// ...`) 仍然是首选。

```
// Determine whether system is running Windows Vista or later operating
// systems (major version >= 6) because they support linked tokens, but
// previous versions (major version < 6) do not.
if (Environment.OSVersion.Version.Major >= 6)
{
    ' Get and display the process elevation information (IsProcessElevated)
    ' and integrity level (GetProcessIntegrityLevel). The information is not
    ' available on operating systems prior to Windows Vista.
    If (Environment.OSVersion.Version.Major >= 6) Then
    End If
}
```

- ☑ 您应该将注释缩进为被其描述的代码的同一级。
- ☑ 您应该在注释中使用首字母大写的完整的句子，以及适当的标点符号和拼写。

Good:

```
// Initialize the components on the Windows Form.
InitializeComponent();

' Initialize the components on the Windows Form.
InitializeComponent()
```

Bad:

```
//intialize the components on the Windows Form.
```

```
InitializeComponent ();

'intialize the components on the Windows Form
InitializeComponent ()
```

2.11.1 内联代码注释

内联注释应该置于独立行，并与所描述的代码具有相同的缩进。其之前需放置一个空行。其之后不需要空行。描述代码块的注释应该置于独立行，与所描述的代码具有相同的缩进。其之前和之后都有一个空行。举例：

```
if (MAXVAL >= exampleLength)
{
    // Reprort the error.
    ReportError (GetLastError ()) ;

    // The value is out of range, we cannot continue.
    return E_INVALIDARG;
}
```

当内联注释只为结构体，类成员变量，参数和短语句做描述时，则内联注释允许出现在和实际代码的同一行。在本例中，最好将所有变量的注释对齐。

```
class Example
{
public:
    ...

    void TestFunction
    {
        ...
        do
        {
            ...
        }
        while (!fFinished); // Continue if not finished.
    }

private:
    int m_length;        // The length of the example
    float m_accuracy;    // The accuracy of the example
};
```

❌ **您不应该** 在代码中留有过多注释。如果每一行代码都有注释，便会影响到可读性和可理解性。单行注释应该用于代码的行为并不是那么明显易懂的情况。

如下代码包含多余注释：

```
Bad:
// Loop through each item in the wrinkles array
for (int i = 0; i <= nLastWrinkle; i++)
{
    Wrinkle *pWrinkle = apWrinkles[i];    // Get the next wrinkle
    if (pWrinkle->IsNew () &&                // Process if it's a new wrinkle
        nMaxImpact < pWrinkle->GetImpact ()) // And it has the biggest impact
    {
        nMaxImpact = pWrinkle->GetImpact (); // Save its impact for comparison
        pBestWrinkle = pWrinkle;           // Remember this wrinkle as well
    }
}
```

更好的实现如下：

```
Good:
// Loop through each item in the wrinkles array, find the Wrinkle with
// the largest impact that is new, and store it in 'pBestWrinkle'.
for (int i = 0; i <= nLastWrinkle; i++)
{
    Wrinkle *pWrinkle = apWrinkles[i];
    if (pWrinkle->IsNew () && nMaxImpact < pWrinkle->GetImpact ())
    {
        nMaxImpact = pWrinkle->GetImpact ();
        pBestWrinkle = pWrinkle;
    }
}
```

✅ **您应该** 为那些仅通过阅读很难理解其意图的代码添加注释。

2.11.2 文件头注释

✅ **一定请** 为每一份手写代码文件加入文件头注释。如下为文件头注释：

VC++ 和 VC# 文件头注释模板:

```

/***** Module Header *****/
Module Name:  <File Name>
Project:      <Sample Name>
Copyright (c) Microsoft Corporation.

<Description of the file>

This source is subject to the Microsoft Public License.
See http://www.microsoft.com/opensource/licenses.mspx#Ms-PL.
All other rights reserved.

THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.
/*****/

```

VB.NET 文件头注释模板:

```

' **** Module Header ****
' Module Name:  <File Name>
' Project:      <Sample Name>
' Copyright (c) Microsoft Corporation.
'
' <Description of the file>
'
' This source is subject to the Microsoft Public License.
' See http://www.microsoft.com/opensource/licenses.mspx#Ms-PL.
' All other rights reserved.
'
' THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
' EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED
' WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.
' ****/

```

举例,

```

/***** Module Header *****/
Module Name:  CppUACSelfElevation.cpp
Project:      CppUACSelfElevation
Copyright (c) Microsoft Corporation.

User Account Control (UAC) is a new security component in Windows Vista and

```


newer operating systems. With UAC fully enabled, interactive administrators normally run with least user privileges. This example demonstrates how to check the privilege level of the current process, and how to self-elevate the process by giving explicit consent with the Consent UI.

This source is subject to the Microsoft Public License.
See <http://www.microsoft.com/opensource/licenses.msp#Ms-PL>.
All other rights reserved.

THIS CODE AND INFORMATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND,
EITHER EXPRESSED OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND/OR FITNESS FOR A PARTICULAR PURPOSE.
*****/

2.11.3 类注释

☑ 您应该为 每一个重要的类或结构体作注释。注释的详细程度应该视代码的用户群而定。

C++ 类注释模板:

```
//
//  NAME:  class <Class name>
//  DESCRIPTION: <Class description>
//
```

C# 和 VB.NET 使用 .NET 描述性 XML 文档化 注释。当您编译.NET 项目，并带有 /doc 命令时，编译器会在源代码中搜索 XML 标签，并生成 XML 文档。

C# 类注释模板:

```
/// <summary>
/// <Class description>
/// </summary>
```

VB.NET 类注释模板:

```
''' <summary>
''' <Class description>
''' </summary>
```

举例,

```
//
//  NAME: class CodeExample
//  DESCRIPTION: The CodeExample class represents an example of code, and
//  tracks the length and complexity of the example.
//
class CodeExample
{
    ...
};

/// <summary>
/// The CodeExample class represents an example of code, and tracks
/// the length and complexity of the example.
/// </summary>
public class CodeExample
{
    ...
}
```

2.11.4 函数注释

☑ 您应该 为所有重要的 Public 或非 Public 函数作注释。注释的详细程度应该视代码的用户群而定。

C++ 函数注释模板:

```
/*-----
FUNCTION: <Funtion prototype>

PURPOSE:
    <Function description>

PARAMETERS:
    <Parameter name> -<Parameter description>

RETURN VALUE:
    <Description of function return value>

EXCEPTION:
    <Exception that may be thrown by the function>

EXAMPLE CALL:
    <Example call of the function>
```

```
REMARKS:
    <Additional remarks of the function>
    -----*/
```

C# 和 VB.NET 使用 .NET 描述性 XML 文档化注释。在一般情况下，至少需要一个 <summary> 元素，一个 <parameters> 元素和 <returns> 元素。会抛出异常的方法应使用 <exception> 元素来告知使用者哪些异常会被抛出。

C# 函数注释模板:

```
/// <summary>
/// <Function description>
/// </summary>
/// <param name="Parameter name">
/// <Parameter description>
/// </param>
/// <returns>
/// <Description of function return value>
/// </returns>
/// <exception cref="<Exception type>">
/// <Exception that may be thrown by the function>
/// </exception>
```

VB.NET 函数注释模板:

```
''' <summary>
''' <Function description>
''' </summary>
''' <param name="Parameter name">
''' <Parameter description>
''' </param>
''' <returns>
''' <Description of function return value>
''' </returns>
''' <exception cref="<Exception type>">
''' <Exception that may be thrown by the function>
''' </exception>
```

举例,

```

/*-----
FUNCTION: IsUserInAdminGroup (HANDLE hToken)

PURPOSE:
    The function checks whether the primary access token of the process
    belongs to user account that is a member of the local Administrators
    group, even if it currently is not elevated.

PARAMETERS:
    hToken - the handle to an access token.

RETURN VALUE:
    Returns TRUE if the primary access token of the process belongs to user
    account that is a member of the local Administrators group. Returns FALSE
    if the token does not.

EXCEPTION:
    If this function fails, it throws a C++ DWORD exception which contains
    the Win32 error code of the failure.

EXAMPLE CALL:
    try
    {
        if (IsUserInAdminGroup (hToken) )
            wprintf (L"User is a member of the Administrators group\n");
        else
            wprintf (L"User is not a member of the Administrators group\n");
    }
    catch (DWORD dwError)
    {
        wprintf (L"IsUserInAdminGroup failed w/err %lu\n", dwError);
    }
-----*/

/// <summary>
/// The function checks whether the primary access token of the process
/// belongs to user account that is a member of the local Administrators
/// group, even if it currently is not elevated.
/// </summary>
/// <param name="token">The handle to an access token</param>
/// <returns>
/// Returns true if the primary access token of the process belongs to
/// user account that is a member of the local Administrators group.
/// Returns false if the token does not.
/// </returns>
/// <exception cref="System.ComponentModel.Win32Exception">
/// When any native Windows API call fails, the function throws a

```

```

    /// Win32Exception with the last error code.
    /// </exception>

```

任何调用失败会带来副作用的方法或函数，都需要清楚地注释交代那些副作用的后果。一般而言，代码在发生错误或失败时不应该有副作用。当出现副作用时，在编写代码时应该有清楚的理由。当函数没有输出参数，或者只有一些单纯作为输出参数的情况下，无需交代理由。

2.11.5 将代码注释掉

当您用多种方法实现某些任务时，将代码注释掉便是必须的。除了第一个方法，其余实现方法都会被注释掉。使用 `[-or-]` 来分隔多个方法。举例，

```

// C++ / C# sample:
// Demo the first solution.
DemoSolution1 ();

// [-or-]

// Demo the second solution.
//DemoSolution2 ();


' VB.NET sample:
' Demo the first solution.
DemoSolution1 ();

' [-or-]

' Demo the second solution.
'DemoSolution2 ();

```

2.11.6 TODO 待办注释

 **一定不要** 在已发布的代码示例中使用 `TODO` 待办注释。每一个代码示例都必须完整，在代码中不能有未完成任务。

2.12 代码块

☑ 一定请在大量代码会因为更具结构化而获益时，使用代码块声明。通过作用域或者功能性分类，将大量代码分组，会改善代码易读性和结构。

C++ 代码块:

```
#pragma region "Helper Functions for XX"
...
#pragma endregion
```

C# 代码块:

```
#region Helper Functions for XX
...
#endregion
```

VB.NET 代码块:


```
#Region "Helper Functions for XX"
...
#End Region
```

3 C++ 编程规范

以下编程规范适用于本地 C++ 代码。

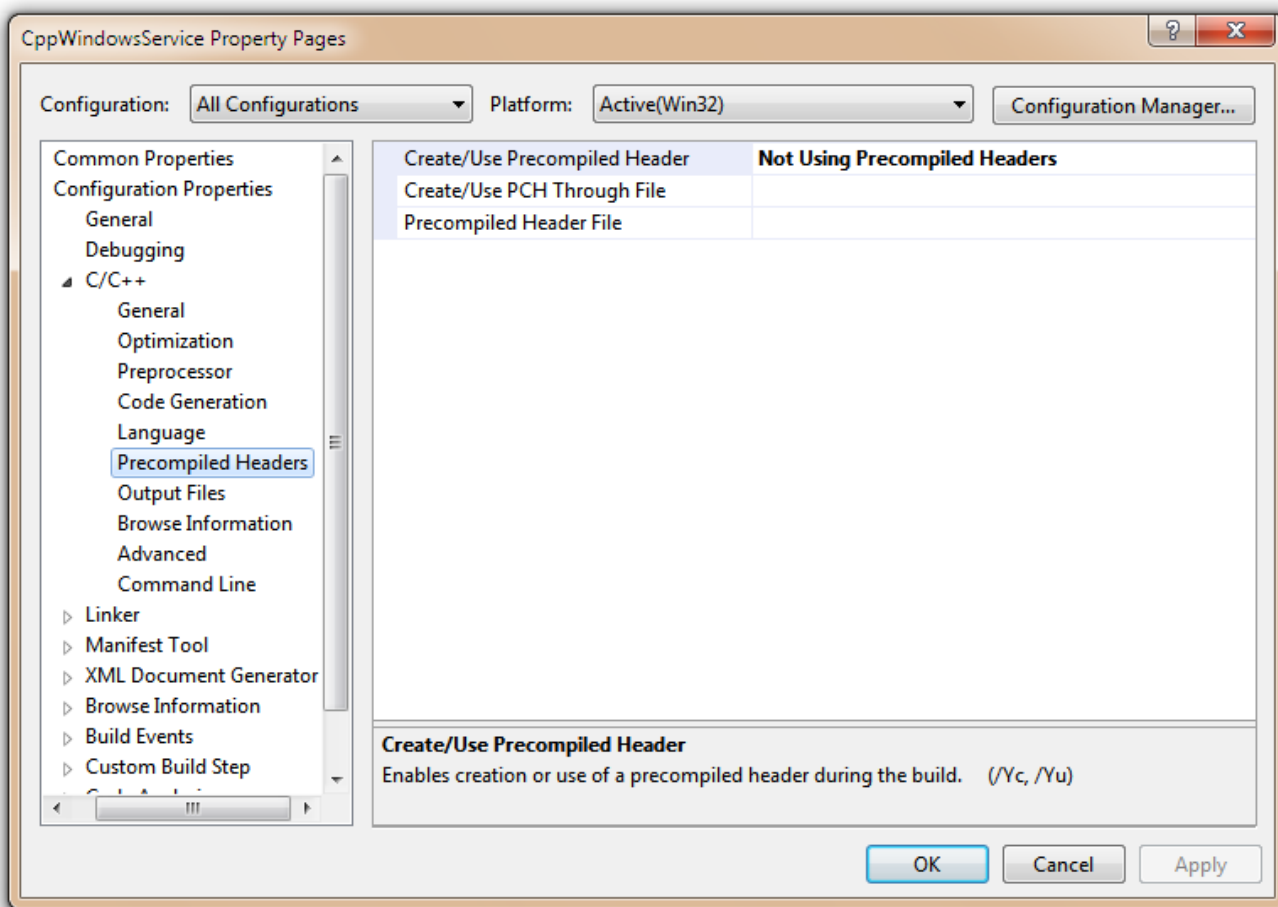
3.1 编译器选项

3.1.1 预编译头

 一定不要使用预编译头。

Visual C++ 项目默认使用预编译头文件。其原理是当生成 `stdafx.h/cpp` 文件时，巨大的 Windows 头文件只被编译一次。项目中其他任何一个 .CPP 文件都需要首先包含 `#include "stdafx.h"`，这样项目才能正确生成。当编译器找到 "stdafx.h" 时，便知道何时插入预编译头信息。

在代码示例中，必须关闭预编译头选项。在您的项目属性中，找到 C/C++ 标签页，选择 **Precompiled headers** 节点。点击 **Not using precompiled headers** 单选按钮，之后点击 OK。请确保修改了所有的配置（包括 Debug 配置 和 Release 配置）。之后，移除所有源文件的 `#include<stdafx.h>`。



3.1.2 开启所有警告，并当做错误来对待

☑ 您应该 以最高警告等级来编译所有代码。

☑ 您应该 把警告当做错误来对待。

编译器提示的警告通常对于鉴别低劣的代码和隐晦的 **bug** 非常有用。您可以以编译器警告对您的代码进行额外的验证。

在 Visual Studio 中，您可以在项目的属性设置页面开启警告等级 4。在项目属性设置页面，找到“Configuration Properties”，“C/C++”，“General”，将“Warning Level”设置为“Level 4”。

3.2 文件和结构

3.2.1 stdafx.h, stdafx.cpp, targetver.h

☑ 您应该删除 Visual Studio 项目模板生成的 `stdafx.h`, `stdafx.cpp` 和 `targetver.h` 文件以保持示例的简洁。然而，如果您有许多被大量代码文件共享的标准头文件，您可以创建单独的文件来包含它们。这非常类似于 `Windows.h` 的作用。

3.2.2 头文件

☑ 一定请在头文件内使用包含保护符（include guards），来防止头文件被无意的多次包含。

以下示例代码中的 `#ifndef` 和 `#endif`，应该为头文件的第一行和最后一行代码。示例代码展示了如何在“CodeExample.h”中使用“`#ifndef/#endif`”作为包含保护符。

```
// File header comment goes first ...

#ifndef CODE_EXAMPLE_H_
#define CODE_EXAMPLE_H_

class CodeExample
{
    ...
};

#endif
```

您也可以使用“`#pragma once`”（微软编译器的一个特定拓展）来替代“`#ifndef/#endif`”包含保护符：

```
// File header comment goes first ...

#pragma once

class CodeExample
{
    ...
};
```

☒ 您不应该在头文件内实现函数。头文件只能包含函数声明和数据结构。它们的实现应置于 `.cpp` 文件。

3.2.3 实现文件

实现文件包含了全局函数，局部函数和类方法实际的函数体。实现文件是拓展名为.c 或者 .cpp 的文件。

注意，实现文件不必包含整个模块的完整实现。它可以被分隔，并包含一个公共内部接口。

✅ **您应该** 将不必导出的声明放置在实现文件中。此外，您应该为它们加上 **static** 关键字，以限制其作用域在该.cpp/.c 文件定义的编译单元。这将减少当链接 2 个或更多使用了相同内部变量的.cpp 文件时，出现“multiply-defined symbol”错误的情况。

3.3 命名规范

3.3.1 通用命名规范

✅ **一定请** 为各种类型，函数，变量，特性和数据结构选取有意义的命名。其命名应能反映其作用。

单字符变量应该仅用于计数器（i,j）或者坐标（x,y,z）。根据经验，变量作用域越大，便越应该使用描述性强的命名。

❌ **您不应该** 在标识符名中使用缩短或缩略形式的词。比如，使用“GetWindow”而不是“GetWin”。对于公共类型，线程过程，窗口过程，和对话框过程函数，为“ThreadProc”，“DialogProc”，“WndProc”等使用公共后缀。

3.3.2 标识符的大小写命名规范

如下表格描述了对不同类型标识符的大小写命名规范。

标识符	大小写命名规范	命名结构	示例
类	Pascal 规范	名词	<pre>class ComplexNumber {...}; class CodeExample {...}; class StringList {...};</pre>
枚举	Pascal 规范	名字	<pre>enum Type {...};</pre>

函数，方法	Pascal 规范	名词或 或 动名词	<code>void Print ()</code> <code>void ProcessItem ()</code>
接口	PascalC 规范， 带‘I’前缀	名字	<code>interface IDictionary {...};</code>
结构体	所有字母大 写，以‘_’分隔 单词	名词	<code>struct FORM_STREAM_HEADER</code>
宏， 常量	所有字母大 写，以‘_’分隔 单词		<code>#define BEGIN_MACRO_TABLE (name) ...</code> <code>#define MACRO_TABLE_ENTRY (a, b, c) ...</code> <code>#define END_MACRO_TABLE () ...</code> <code>const int BLACK = 3;</code>
参数，变量	Camel 规范	名词	<code>exampleText, dwCount</code>
模板参数	Pascal 规范， 带‘T’前缀	名词	<code>T, TItem, TPolicy</code>

3.3.3 匈牙利命名法

☑ 您可以对参数和变量使用匈牙利命名法。然而，匈牙利命名法相当老旧，会对代码重造成困难。例如，当改变变量类型时，您需要对所有代码都进行更改。

如下表格定义了一套配套的匈牙利命名法标记，如果您使用匈牙利命名法，建议使用它们。

类型	标记	描述
bool, BOOL, bitfield	f	一个旗标。例如, <code>BOOL fSucceeded;</code>
BYTE		一个 8 位无符号数。BYTE 应当仅用于不透明量，例如 <code>cookies</code> ，位字段等
WORD		一个 16 位无符号数。WORD 应当仅用于不透明量，例如 <code>cookies</code> ，句柄，位字段等
DWORD	dw	一个 32 位无符号数。DWORD 应当仅用于不透明量，例如 <code>cookies</code> ，句柄，位字段等
HRESULT	hr	HRESULT 一般用于 Win32 的错误或状态值。
VARIANT	vt	一个 OLE VARIANT。
HANDLE	h	一个句柄
int, unsigned int		一个 32 位序数（可以用于<, <=, >, >=等比较）。注意：在 64 位 Windows 内 整数为 32 位。
short, unsigned short		一个 16 位序数。尽量不要使用这些标志，除非在磁盘格式和堆结构体中。

long, unsigned long		一个 32 位序数。尽量不要使用这些标志，因为 int 具有其所有功能，所以我们偏向于使用 int。
__int64, LONGLONG, ULONGLONG		一个 64 位序数。
TCHAR, wchar_t, char	ch	一个字符（未指明符号）。“wchar_t”类型更适合于宽字符，因为其属于 C++ 内置类型。对于 char 和 TCHARS，我们使用相同的标志，因为我们在项目中都使用 Unicode。如果发生函数中包含 char 和 WCHAR 的特例，则使用 "ch" 代表 char 和 "wch" 代表 wchar_t。
PWSTR, PCWSTR, wchar_t *, PSTR, PCSTR, char *	psz	一个指向 0 值结尾的字符串。因为我们在项目中都使用 Unicode，我们不必为 PSTR 和 PWSTR 设置不同标志，对于 char 和 TCHARS 也一样。如果发生函数中包含 PSTR 和 PWSTR 的特例，则使用 "psz" 代表 PSTR 和 "pwsz" 代表 PWSTR。在微软接口定义语言之外，即使是接口方法也使用 PWSTR 和 PSTR。所有指针都是长指针，L 前缀已经被废弃。
wchar_t [], char []	sz	一个 0 值结尾的字符串（形式为栈上的字符串数组）。比如，wchar_t szMessage[BUFFER_SIZE];
BSTR	bstr	一个 OLE 自动化的 BSTR。
void		一个 void。对于指向 void 的指针，请加上 "p" 前缀
(*) ()		一个函数。对于指向函数的指针，请加上 "p" 前缀

举例，

```
HANDLE hMapFile = NULL;
DWORD dwError = NO_ERROR;
```

匈牙利命名法的前缀用于增加类型信息– 以下是匈牙利命名标志的前缀：

Prefix	Description
p	一个指针（32 位 或 64 位，取决于平台）。
sp	一个‘智能’指针，例如，一个拥有类指针语义的类
c	数量。比如，cbBuffer 代表缓冲（buffer）的字节数。“c”后不跟标志也是可以接受的。
m_	类中的一个成员变量
s_	类中的一个静态成员变量
g_	一个全局变量
I	COM 接口

举例，

```

UINT cch; // Count of characters
PWSTR psz; // String pointer, null terminated
wchar_t szString[] = L"foo";

```

3.3.4 用户界面 控件命名规范

用户界面控件可使用如下前缀，以及随后的资源 ID 格式。其主要目的是使代码更统一易读。

控件类型	控件句柄命名格式	MFC 控件命名格式	资源 ID (所有字母大写，以 '_' 分隔单词)
Animation Control	hXxxAnimate	aniXxx	IDC_ANIMATE_XXX
Button	hXxxButton	btnXxx	IDC_BUTTON_XXX
Check Box	hXxxCheck	chkXxx	IDC_CHECK_XXX
ComboBox	hXxxCombo	cmbXxx	IDC_COMBO_XXX
Date Time Picker	hXxxDatePicker	dtpXxx	IDC_DATETIMEPICKER_XXX
Edit Control	hXxxEdit	tbXxx	IDC_EDIT_XXX
Group Box	hXxxGroup	grpXxx	IDC_STATIC_XXX
Horizontal Scroll Bar	hXxxScroll	hsbXxx	IDC_SCROLLBAR_XXX
IP Address Control	hXxxIpAddr	ipXxx	IDC_IPADDRESS_XXX
List Box	hXxxList	lstXxx	IDC_LIST_XXX
List (View) Control	hXxxList	lvwXxx	IDC_LIST_XXX
Menu	hXxxMenu	N/A	IDM_XXX
Month Calendar Control	hXxxCalendar	mclXxx	IDC_MONTHCALENDAR_XXX
Picture Box	hXxxPicture	pctXxx	IDC_STATIC_XXX
Progress Control	hXxxProgress	prgXxx	IDC_PROGRESS_XXX
Radio Box	hXxxRadio	radXxx	IDC_RADIO_XXX
Rich Edit Control	hXxxRichEdit	rtfXxx	IDC_RICHEDIT_XXX
Slider Control	hXxxSlider	sldXxx	IDC_SLIDER_XXX
Spin Control	hXxxSpin	spnXxx	IDC_SPIN_XXX
Static Text	hXxxLabel	lbXxx	IDC_STATIC_XXX
SysLink Control	hXxxLink	lnkXxx	IDC_SYSLINK_XXX
Tab Control	hXxxTab	tabXxx	IDC_TAB_XXX
Tree (View) Control	hXxxTree	tvwXxx	IDC_TREE_XXX
Vertical Scroll Bar	hXxxScroll	vsbXxx	IDC_SCROLLBAR_XXX

3.4 指针

☑ **您应该** 总是在声明指针时进行初始化，释放之后应赋予 **NULL** 值或其他无效值。此举防止了其余代码使用未初始化的指针，意外的读写未知地址而破坏进程地址空间。举例：

Good:

```
BINARY_TREE *directoryTree = NULL;
DWORD *pdw = (DWORD *) LocalAlloc (LPTR, 512);
...
if (pdw != NULL)
{
    LocalFree (pdw);
    pdw = NULL;
}
...
if (directoryTree != NULL)
{
    // Free directoryTree with match to way it was allocated
    FreeBinaryTree (directoryTree);
    directoryTree = NULL;
}
```

☑ **您应该** 在指定一个指针类型/变量时，在 '*' 字符和类型之间留有一个空格, 而不要在 '*' 字符和变量之间留有空格。该规范使得代码更一致。 以下是示例：

Good:

```
HRESULT GetInterface (IStdInterface **ppSI);
INFO *GetInfo (DWORD *pdwCount);
DWORD *pdw = (DWORD *) pv;
IUnknown *pUknwn = static_cast<IUnknown *> (*ppv);
```

Bad:

```
HRESULT GetInterface (IStdInterface** ppSI);
INFO* GetInfo (DWORD * pdwCount);
DWORD* pdw = (DWORD*) pv;
IUnknown* pUknwn = static_cast<IUnknown*> (*ppv);
```

3.5 常量

☑ **一定请** 使用 'const' 值来定义命名常量，而不是用 "#define"。举例：

Good:

```
const int BLACK = 3;
```

Bad:

```
#define BLACK 3
```

当您使用 `const` 常量时，编译器会强制类型检查，并将该常量添加至符号表，这将使得调试时更加简便。相反，预处理器并不会带来这些好处。

☑ **您应该** 使用枚举来定义一组相关常量。这使得常量具有同样的类型，能改善函数接口。举例：

Good:

```
enum DayOfWeek {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};
enum Color {Black, Blue, White, Red, Purple};
```

```
// Note the strong type parameter checking; calling code can't reverse them.
BOOL ColorizeCalendar (DayOfWeek today, Color todaysColor);
```

Bad:

```
const int Sunday = 0;
const int Monday = 1;
const int Black  = 0;
const int Blue   = 1;
```

```
// Note the weak type parameter checking; calling code can reverse them.
BOOL ColorizeCalendar (int today, int todaysColor);
```

☑ **您应该** 在适当的时候，对传入和返回参数加上 '`const`' 修饰符。通过应用 '`const`' 修饰符便清楚表明了该参数的意图，编译器能增加一层额外的验证，以验证代码不会修改这些 `const` 值。

Const 用法	意义	描述
<code>const int *x;</code>	指向一个 <code>const int</code> 的指针	指针 <code>x</code> 指向的值不能被修改
<code>int * const x;</code>	指向 <code>int</code> 的 <code>Const</code> 指针	<code>x</code> 不能指向另外的地址
<code>const int *const x;</code>	指向 <code>const int</code> 的 <code>const</code> 指针	指针和指针指向的值都不能被 改变

3.6 类型转换

☑ **您应该** 使用 C++ 风格的类型转换，C++ 风格的类型转换更加显式，提供更好的对类型的控制，并更能表现出代码的意图。有 3 种 C++ 风格的类型转换：

1. `static_cast` 多用于将指针转换为另一同一继承体系内的指针。它是安全的类型转换。编译器会确保转换后的类型便是您所想转换的。该转换经常用于消除多重继承带来的类型歧义。
2. `reinterpret_cast` 用于转换不相关类型。警告：不要对 `DWORD` 和指针进行相互转换。在 64 位平台下，它不能通过编译。记得对您 `reinterpret_cast<>` 的用法进行注释，这有助于减少读者在看到此类型转换时的担心。
3. `const_cast` 用于移除对象的 'const' 性质。

以上三种转换的语法是类似的：

```
DerivedClass *pDerived = HelperFunction ();
BaseClass *pBase = static_cast<BaseClass *> (pDerived);
```

❌ 您不应该使用 'const_cast'，除非是绝对需要。必须使用 'const_cast' 一般意味着某个 API 没有很恰当的使用 'const'。注意：Win32 API 并不常对传入参数使用 'const'，所以使用这些 Win32 API 时，有可能需要使用 `const_cast`。

3.7 Sizeof

✅ 一定请在条件允许的情况下，使用 `sizeof`（变量），而不是 `sizeof`（类型）。代码中不要出现变量的已知大小或者类型。

Good:

```
MY_STRUCT s;
ZeroMemory (&s, sizeof (s));
```

Bad:

```
MY_STRUCT s;
ZeroMemory (&s, sizeof (MY_STRUCT));
```

❌ 一定不要对数组使用 `sizeof` 来获得元素数量。应该使用 `ARRAYSIZE`。

3.8 字符串

☑ **一定请** 编写显式 UNICODE 代码，因为其更容易全球化。不要使用 TCHAR 或者 ANSI char，这样能减少一半的测试代价，并消除了字符串 bug。这意味着：

使用宽字符类型，包括 `wchar_t`, `PWSTR`, `PCWSTR`，不要使用 TCHAR 版本的类型。

Good:

```
HRESULT Function (PCWSTR)
```

Bad:

```
HRESULT Function (PCTSTR)
```

变量名不应该指明宽字符的“W”。

Good:

```
Function (PCWSTR psz)
```

Bad:

```
Function (PCWSTR pwsz)
```

不要使用 TEXT 宏，而使用 L 前缀来创建 Unicode 字符串常量 L“字符串值”。

Good:

```
L"foo"
```

Bad:

```
TEXT ("foo")
```

相较 WCHAR，倾向于选择 `wchar_t`，因为其是原生的 C++ 类型。

Good:

```
L"foo"
wchar_t szMessage[260];
```

Bad:

```
TEXT ("foo")
WCHAR szMessage[260];
```

永远不要显式的使用 A/W 版本的 API。这是低劣的编程风格，因为 API 的名称原本便是没有 A/W 的。同样的，硬编码会对在 ANSI/Unicode 之间的移植增加难度。

Good:

```
CreateWindow (...);
```

Bad:

```
CreateWindowW (...);
```

☑您应该在条件允许的情况下，为字符串使用固定大小栈缓冲区，而不是动态分配。使用固定大小栈缓冲区有如下好处：

- 更少的错误状态，无需检验分配失败，无需编写代码来处理这些情况的发生。
- 绝不会出现内存泄露，因为栈会帮您自动回收内存。
- 更好的性能表现，不会出现临时堆的使用。

以下几种情况，应该避免使用栈缓冲区：

- 它不适用于当一些数据量是任意的情况。
注意：用户界面字符串受到 UA 指导准则和屏幕尺寸的限制，所以您一般可以为其大小设定一个上限。将您所认为的字符串大小翻倍，以适应将来在其他语言中的增长（据统计，一般有 30% 的语言增长。）
- “大数据”；大小超过 MAX_PATH（260） 几倍，或者超过 MAX_URL（2048），这些情况下便不适用。
- 递归函数

所以对于取了合理最大值的小数据，请尽量将数据置于栈上。

3.9 数组

3.9.1 数组大小

☑一定请 尽量使用 ARRAYSIZE()来获得数组大小。当用于非数组类型时，ARRAYSIZE()会产生一个错误：错误 C2784。对于匿名类型，您需要使用更低安全性的_ARRAYSIZE() 宏。使用 ARRAYSIZE() 来代替 RTL_NUMBER_OF(), _countof(), NUMBER_OF(), 等等。

☑一定请 从数组变量取得其大小，而不要在代码中指定其大小：

Good:

```
ITEM rgItems[MAX_ITEMS];
for (int i = 0; i < ARRAYSIZE (rgItems); i++)    // use ARRAYSIZE ()
{
    rgItems[i] = fn (x);
    cb = sizeof (rgItems[i]);    // specify the var, not its type
}
```

Bad:

```
ITEM rgItems[MAX_ITEMS];
// WRONG, use ARRAYSIZE (), no need for MAX_ITEMS typically
for (int i = 0; i < MAX_ITEMS; i++)
{
    rgItems[i] = fn (x);
    cb = sizeof (ITEM);    // WRONG, use var instead of its type
}
```

3.9.2 数组初始化

☑ 一定请使用 `"= {}"` 来将数组置零。编译器优化了 `"= {}"`，其性能高于 `"= {0}"` 和 `ZeroMemory`。

3.10 宏

☒ 您不应该使用宏，除非绝对需要它们。大多宏实现的功能可以由 C++ 的特性来代替（使用常量，枚举，内联函数，或者模板，它们更加明确，安全和易懂。）

Good:

```
__inline PCWSTR PCWSTRFromBSTR (__in BSTR bstr)
{
    return bstr ? bstr : L"";
}
```

Bad:

```
#define PCWSTRFromBSTR (bstr)    (bstr ? bstr : L"")
```

☒ 一定不要使用 `"#define"` 值来定义常量。请查看“常量”章节。

☒ 一定不要使用这些宏: `SIZEOF()`，`IID_PPV_ARG()`（使用 `IID_PPV_ARGS()` 来替代）。

3.11 函数

3.11.1 验证参数

✅ **一定请** 验证公开函数的参数。如果参数无效，将最后错误代码（last error）设置为 `ERROR_INVALID_PARAMETER`，并返回 `HRESULT_E_INVALIDARG`。

3.11.2 引用参数

❌ **一定不要** 使用引用参数作为输出参数。因为很难得知在函数调用处，变量是否被修改了。此时应该使用指针。比如，考虑如下函数：

```
Function ()
{
    int n = 3;
    Mystery (n) ;
    ASSERT (n == 3) ;
}
```

该断言是否有效？如果您并不知道函数的声明，您可能会想：“当然了，n 的值肯定不会被修改”。然而，`Mystery` 函数声明如下：

```
void Mystery (int &n) ;
```

那么答案便是“可能被修改，也可能没有”。如果 `Mystery` 函数有修改其实参的意图，它应该被重写为：

```
void Mystery (int *pn) ;

Function ()
{
    int n = 3;
    Mystery (&n) ;
}
```

现在，我们便知道 `Mystery` 函数可能会修改其实参。

如果您选择按引用传递对象（比如，一个结构体），您可以选择显式的通过指针来传递（如果其是一个输出参数），或者使用 `const` 引用（如果其是一个输入参数）。`const` 属性指明函数不应修改该对象。这遵守了“不带 `&` 的传入参数不会被函数修改”的规范。我们已经为对象类型的常见情景定义了一些宏，例如 `REFCLSID`, `REFIID` 和 `REFPROPERTYKEY`。

3.11.3 未引用参数

当实现接口或标准导出内的方法时，有一些参数没有被引用是相当常见的。编译器会发现未使用的参数，并产生一个警告，有些组件甚至会认为这是一个错误。为避免发生如此情况，将未使用的参数使用 `/* 参数名 */` 语法将其注释掉。不要使用 `UNREFERENCED_PARAMETER()` 宏，因为其 1) 太繁琐，2) 并不能保证参数实际上真的未被引用。

Good:

```
LRESULT WndProc (HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM /* lParam */)
{
    ...
}
```

Bad:

```
LRESULT WndProc (HWND hwnd, UINT uMsg, WPARAM wParam, LPARAM lParam)
{
    UNREFERENCED_PARAMETER (lParam);
    ...
}
```

3.11.4 输出字符串参数

一个从函数返回字符串的常见方法是让调用者指定字符串应该存储的地址以及与字符数等长的缓冲区长度。这便是“`pszBuf/cchBuf`”模式。在方法中，您首先需要显式的检验缓冲区大小是否大于 0。

在 COM 应用中，您可以返回 `__out string` 字符串参数作为由 `CoTaskMemAlloc` / `SysAllocString` 动态分配的字符串。这便避免了代码中字符串大小的限制。调用者同样负责调用 `CoTaskMemFree` / `SysFreeString`。

3.11.5 返回值

☑ **一定请** 检查函数的返回值，而不是输出参数，以判断该函数是否调用成功。一些函数会通过多种方式来传递成功或失败状态信息。比如，在 COM 方法中，在 HRESULT 和输出参数内便能得到这些信息。如下 2 个示例都是正确的，但前者更好。

Good:

```
IShellItemImageFactory *psiif;
if (SUCCEEDED (psi->QueryInterface (IID_PPV_ARGS (&psiif) )))
{
    // Use psiif
    psiif->Release ();
}
```

Bad:

```
IShellItemImageFactory *psiif;
psi->QueryInterface (IID_PPV_ARGS (&psiif) );
if (psiif)
{
    // Use psiif
    psiif->Release ();
}
```

理由：

- HRESULT 具有从函数返回的非常重要的信息，应当对它进行检验。
- 通常，HRESULT 值带有需要被传递的、非常重要的信息，这大大降低了将所有失败映射或认作 E_FAIL 的机会。
- 对 HRESULT 进行检验，使得代码能够与一些错误输出参数并非 0 的情况（许多 win32 API）保持一致性。
- 检验 HRESULT 使得调用和检验置于同一行代码。
- 检验 HRESULT 在代码生成方面更高效。

3.12 结构体

3.12.1 Typedef 结构体

✅ 您应该在创建命名结构体时使用 `typedef`。如果您想要的只是一个单结构变量，你便不必这么做。如果需要 `typedef` 标签（前置使用和自引用的 `typedef`），则它应与类型名相同 “`typedef struct FOO { int x; } FOO;`” 结构体名应该所有字母大写，并以 ‘_’ 分隔字符。举例：

```
// The format of the bytes in the data stream....
typedef struct FORM_STREAM_HEADER
{
    int cbTotalSize;
    int cbVarDataOffset;
    USHORT usVersion;
} FORM_STREAM_HEADER;
```

3.12.2 结构体初始化

✅ 一定请使用 “`= {}`” 对结构体置零。

```
PROPVARIANT pv = {};
```

当一个结构体的第一个成员变量是字节大小字段时，您可以使用如下捷径来初始化该字段，并将其余字段置零：

```
SHELLEXECUTEINFO sei = { sizeof(sei) };
sei.lpFile = ...
```

3.12.3 结构体 vs 类

✅ 一定请使用结构体来定义一个不包含函数的数据聚合。如果数据结构包含成员方法，便使用类。对于 C++，一个结构体可以拥有成员函数，操作符以及类中所有的其他特性。实际上，类与结构体差别在于所有结构体内的成员默认访问权限是 `Public` 的，但是在类中是 `Private` 的。通常情况下，当且仅当拥有成员函数时才使用类。

3.13 类

3.13.1 数据成员

❌ **一定不要** 声明 **Public** 数据成员。使用内联访问器函数以提高性能表现。

✅ **一定请** 在构造函数内，倾向使用初始化，而不是赋值操作。初始化示例：

Good:

```
class Example
{
public:
    Example (const int length, const wchar_t *description) :
        m_length (length) ,
        m_description (description) ,
        m_accuracy (0.0)
    {
    }

private:
    int m_length;
    CString m_description;
    float m_accuracy;
};
```

Bad:

```
class Example
{
public:
    Example (int length, const wchar_t *description)
    {
        m_length = length;
        m_description = description;
        m_accuracy = 0.0;
    }

private:
    int m_length;
    CString m_description;
    float m_accuracy;
};
```


☑ **一定请** 按照成员变量在类中声明的顺序来初始化它们。初始化的顺序是成员变量在类定义中声明的顺序，而不是它们在初始化列表中的顺序。如果顺序都是一致的，则代码能够反应出编译器生成代码的情况。考虑以下“CodeExample”示例：

```

Bad:
class CodeExample
{
public:
    explicit CodeExample (int size) ;
    ~CodeExample () ;

private:
    wchar_t *m_buffer;
    int m_size;
};

CodeExample::CodeExample (int size) :
    m_size (size) ,
    m_buffer ( (wchar_t*) operator new[] (m_size) )
{
}

CodeExample::~CodeExample ()
{
    delete [] m_buffer;
}

int wmain (int argc, wchar_t *argv[])
{
    CodeExample example (23) ;
    return 0;
}

```

CodeExample 类的声明定义了 m_buffer 和 m_size，所以首先初始化 m_buffer，接着是 m_size。而构造函数的代码所表示的初始化顺序与声明顺序相反。这会出现当 m_buffer 在初始化的时候，m_size 看起来是有效的，但是事实并非如此。如果改变一下声明顺序，代码便如预期一般运作。

3.13.2 构造函数

☑ **一定请** 将构造函数的工作量降到最低。除了得到构造函数参数，设置主要数据成员，构造函数不应该有太多的工作量。其余工作量应该被推迟，直到必须。

☑ **您应该** 为类显式的声明 复制语义。复制构造函数和赋值操作符是非常特殊的方法– 如果您不提供一个实现，编译器会为您提供一个默认实现。如果类语义不支持复制，请显式的提供 **Private** 的未实现的复制构造函数和赋值操作符来禁用复制语义。举例：

```
class Example
{
private:
    Example (const Example&);
    Example& operator= (const Example&);
};
```

您不应该提供这些方法的实现，因为当您意外的使用它们时，编译器会报错以警告您。

☑ **一定请** 在定义复制构造函数时，使用 ‘const’ 引用类型。比如，对于类 T，复制构造函数应该被定义为：

```
T (const T& other)
{
    ...
}
```

如果构造函数被定义为 “T (T& other)” 或者甚至 “T (T& other, int value = 0)”，它们仍然是复制构造函数。通过使用 “const T&” 进行标准化，构造函数能用于 const 和非 const 值，同时拥有 const 带来的安全性。

☑ **一定请** 以 ‘explicit’ 关键字来定义所有单参数构造函数，这样，它们便不会成为 **转换构造函数**。举例：

```
class CodeExample
{
    int m_value;
public:
    explicit CodeExample (int value) :
        m_value (value)
    {
```

```
    }
};
```

❌ **一定不要** 提供 转换构造函数，除非类语义需要。

3.13.3 析构函数

✅ **一定请** 使用析构函数来集中类的资源清理工作（通过 `delete` 来释放资源）。如果在析构函数之前，资源被释放，请确保正确设置了该字段，（例如，将指针设置为 `NULL`），以保证析构函数不会重复释放。

✅ **一定请** 在拥有至少一个虚函数（非析构函数）的类中，将析构函数设置为 `"virtual"`。如果类没有任何虚函数，请不要声明虚析构函数。

使用虚析构函数的原则是类有其他的虚函数。假设，类 `B` 继承自类 `A`，您有一个类型 `A` 的指针 `p`。`p` 实际可以指向 `A` 或 `B`。假设 `A` 和 `B` 有一个虚函数 `F`，若 `p` 指向的是 `A`，则 `p->F()` 会调用 `A::F`，若 `p` 指向的是 `B`，则会调用 `B::F`。同样，您显然需要配对析构函数 `~A` 或 `~B`，所以析构函数也需是虚函数。如果 `F` 不是虚函数会怎样？那么无论指针指向哪个对象，您最终只会调用 `A::F`。如果指针指向 `B`，那您也只能将 `B` 当做 `A`。所以，您绝对不会希望调用 `B::F`，因为该成员函数更新了 `A` 的状态，而不是 `B`。`B` 的析构函数可能会失败，因为其修改了只适用于 `B` 的状态。滥用虚函数会造成大量 `C++` bug。

3.13.4 操作符

❌ **一定不要** 重载 `&&`，`||`，`,`，等操作符。和内置的 `&&`，`||` 或 `,` 操作符不同，这些重载版本并不能呈现出短路特性，所以使用这些操作符会引起不可意料的结果。

❌ **您不应该** 重载操作符，除非类语义需要。

❌ **一定不要** 在重载时，改变操作符原有语义，不要使用 `+` 操作符来做减运算。

❌ **您不应该** 实现 转换操作符，除非类语义需要。

3.13.5 函数重载

❌ **一定不要** 在重载时，随意的变化参数名。如果一个重载函数中的参数代表着另一个重载函数中相同的参数，该参数则应该有相同的命名。具有相同命名的参数应该在重载函数中出现在同一位置。

✅ **一定请** 仅将最长得重载函数设为虚函数（为了拓展性考虑）。短的重载函数应该调用到长重载函数。

3.13.6 虚函数

✅ **一定请** 以虚函数来实现多态。

❌ **一定不要** 使用虚函数，除非您真的需要。因为虚函数通过虚表调用，会产生额外代价。

✅ **您应该** 在重写虚函数时使用 ‘override’ 关键字。（注意：这是微软对于 C++ 的特定拓展）。Override 关键字会使编译器确保函数原型与基类中虚函数匹配。如果基类中的虚函数原型稍作修改，编译器便会为需要修改的派生类产生一个错误。

举例：

```
class Example
{
protected:
    Example ()
    {
    }

public:
    virtual ~Example ()
    {
    }

    virtual int MeasureAccuracy ();

private:
    Example (const Example&);
    Example& operator= (const Example&);
};
```

```

class ConcreteExample : public Example
{
public:
    ConcreteExample ()
    {
    }

    ~ConcreteExample ()
    {
    }

    int MeasureAccuracy () override;
};

```

3.13.7 抽象类

抽象类提供了一个多态基类，需要派生类来提供虚函数实现。

✅您可以使用 'abstract' 关键字来表示抽象类（注意: 这是微软对于 C++ 的特定拓展）。

✅一定请提供一个 Protected 构造函数。

✅一定请将抽象方法设置为纯虚函数。

✅一定请提供一个 Public 的，虚析构函数，如果您允许通过指向抽象类的指针来进行 delete 操作。若您不想如此，请提供一个 Protected 的，非虚析构函数。

✅一定请显式提供 Protected 复制构造函数和赋值操作符，或 Private 未实现的复制构造函数和赋值操作符- 如果用户意外的使用了抽象基类进行一个传值操作，便会产生一个编译错误。

抽象类示例:

```

class Example abstract
{
protected:
    Example ()
    {
    }
}

```

```

public:
    virtual ~Example ()
    {
    }

    virtual int MeasureAccuracy () = 0;

private:
    Example (const Example&) ;
    Example& operator= (const Example&) ;
};

```

3.14 COM

3.14.1 COM 接口

☑ 一定请使用 IFACEMETHODIMP 和 IFACEMETHODIMP_ 作为 COM 接口的方法声明。这些宏替代了 STDMETHODCALLTYPE 和 STDMETHODCALLTYPE_，因为他们增加了 __override SAL 注释。

示例:

```

class CNetDataObj : public IDataObject
{
public:
    // IDataObject
    IFACEMETHODIMP GetData (FORMATETC *pFmtEtc, STGMEDIUM *pmedium)

    ...

    IFACEMETHODIMP CNetDataObj::GetData (FORMATETC *pFmtEtc, STGMEDIUM *pmedium)
    {
        ...
    }
}

```

☑ 一定请 将类定义中接口方法的顺序与声明中的顺序保持一致。以下是 IUnknown 的顺序：
QueryInterface()/AddRef()/Release()。

3.14.2 COM 接口 ID

`__uuidof()` 是一个编译器支持的特性，其产生一个可能与类型相关的 GUID 值。该类型可能是接口指针，或者类。该 GUID 通过 `__declspec(uuidof-(<guid 值>))` 与该类型联系在一起。当您可以使用 `IID_PPV_ARGS()` 或者模板化的 `QueryInterface()` 时，应该避免使用 `__uuidof()`。

当您需要接口指针的 IID 而不是硬编码该变量的 IID 时，请使用 `__uuidof(变量)`。这和 `sizeof(变量)` vs `sizeof(变量类型)` 的用法一样。注意您可以使用“`sizeof 变量`”，无需圆括号。举例，

```
CoMarshalInterThreadInterfaceInStream (__uuidof (psia), psia, &pstm);
```

当面临使用 `IID_ISomeInterface` 和 `__uuidof(ISomeInterface)` 的选择时，偏向于前者，因为其更加简洁。同样适用于 `CLSID_SomecoClass` 和 `__uuidof(SomeCoClass)`。一个例外是如果显式的引用 IID 或者 CLSID 需要一个 .lib 文件来链接时，并且没有提供，那么便使用 `__uuidof()`，而不是在代码中定义这些值。

3.14.3 COM 类

☑ **一定请** 在实现了动态分配于堆上的 COM 对象的类内，声明 **Private** 析构函数（或者 **Protected**，如果您希望其余类继承该类）。这避免了用户错误的进行“`delete pObj`”等行为，这些行为只能发生在其引用计数变为 0 时。

☑ **一定请** 在实现了 COM 对象的类的构造函数里，将 `m_cRef` 初始化为 1。（注意：ATL 使用了不同的引用计数的初始化模式，将其设为 0，并期望智能指针的赋值会将其增加到 1）。这样便消除了类存在，却不能被释放的情况的发生。

☑ **一定请** 从每一个 COM 方法中返回一个 **HRESULT**。（除了 **AddRef** 和 **Release**）。

3.15 动态分配

☑ **一定请** 确保所有动态分配的内存会被相同的机制所释放。使用‘**new**’动态分配的对象，应该使用‘**delete**’来释放。举例：

```
Engine *pEngine = new Engine ();
pEngine->Process ();
```

```
delete pEngine;
```

使用 `vector new` 的动态分配应该使用 `vector delete`。举例：

```
wchar_t *pszBuffer = new wchar_t[MAX_PATH];
SomeMethod (pszBuffer);
delete [] pszBuffer;
```

☑ **一定要** 理解您代码中的动态分配，以确保他们被正确的释放了。

3.15.1 智能指针

☑ **您应该** 使用 **RAII**（初始化时分配资源）特性来帮助追踪动态分配—例如：使用智能指针。示例如下：

```
{
    CAutoPtr<Engine> spEngine (new Engine ());
    spEngine->Process ();
}

{
    CAutoVectorPtr<wchar_t> spBuffer ();
    spBuffer.Allocate (MAX_PATH);
    SomeMethod (spBuffer);
}
```

☒ **一定不要** 仅为使用 `CAutoPtr` 和 `CAutoVectorPtr` 而使用 `ATL`。

3.16 错误和异常

☑ **一定要** 在大多情况下，为了代码简洁，倾向于使用错误代码返回值，而不是异常处理。DLL 导出的 API 和方法一般都使用错误代码。

3.16.1 错误

☑ **一定请** 检查函数返回值，并恰当的处理错误。使用控制台程序，并发现错误时，请尽早打印错误信息，并处理错误。举例：

```
// Function return HRESULT.
IShellLibrary* pShellLib = NULL;
```



```

HRESULT hr = SHCreateLibrary (IID_PPV_ARGS (&pShellLib) );
if (FAILED (hr) )
{
    wprintf (L"SHCreateLibrary failed w/err 0x%08lx\n", hr);
    goto Cleanup;
}

// Function return TRUE/FALSE , set Win32 last error.
DWORD dwError = ERROR_SUCCESS;
HANDLE hToken = NULL;
if (!OpenProcessToken (GetCurrentProcess (), TOKEN_QUERY | TOKEN_DUPLICATE,
    &hToken) )
{
    dwError = GetLastError ();
    wprintf (L"OpenProcessToken failed w/err 0x%08lx\n", dwError);
    goto Cleanup;
}

```

3.16.2 异常

异常是 C++ 的特性，在正确使用之前需要很好的理解它们。在使用带有本地 C++ 异常的代码时，请确保您了解使用这些代码的含义。

本地 C++ 异常，是一个 C++ 语言强大的特性。可以降低代码复杂度，减少编写和维护的代码量。

☑ 一定请 按值抛出异常，并按引用捕获异常。 举例：

```

void ProcessItem (const Item& item)
{
    try
    {
        if (/* some test failed */)
        {
            throw _com_error (E_FAIL);
        }
    }
    catch (_com_error& comError)
    {
        // Process comError
        //
    }
}

```

✅ 当重新抛出异常时，请使用“throw”来重抛，而不要使用“throw <捕获到的异常>”。举例，

Good:

```
void ProcessItem (const Item& item)
{
    try
    {
        Item->Process ();
    }
    catch (ItemException& itemException)
    {
        wcout << L"An error occurred."
        throw;
    }
}
```

Bad:

```
void ProcessItem (const Item& item)
{
    try
    {
        Item->Process ();
    }
    catch (ItemException& itemException)
    {
        wcout << L"An error occurred."
        throw itemException;
    }
}
```

❌ 一定不要 从析构函数中抛出异常。

❌ 一定不要 使用 “catch (...)”。不应该捕获所有异常。您应该捕获更加明确的异常，或者在 Catch 块的最后一条语句处重新抛出异常。有些情况下，隐藏错误是可接受的，但是这些情况的出现几率非常低。只捕获函数能处理的特定异常。所有其他异常都不用处理。

❌ 一定不要 在控制流中使用异常。除了系统故障 或者带有潜在资源竞争条件的操作，您编写的代码都不应该抛出异常。比如，在调用可能失败或抛出异常的方法前，您可以检查其前置条件，举例：

```
if (IsWritable (list))
```

```

{
    WriteList (list);
}

```

✅ **一定要** 确保您理解可能从您所依赖的代码中抛出的异常，并确保异常不会无故传递至使用您的 API 的用户代码中。比如，STL 和 ATL 在一些特定情况下，可能抛出本地 C++ 异常 – 了解这些情况，确保您的代码恰当地处理了这些异常，以防止其向外传递。

3.17 资源清理

动态分配的内存或资源，在您退出函数之前应该被正确的清理，以防止内存或资源泄露。在函数执行过程中出现错误时，恰当的资源清理方案是非常重要的。以下是在函数中，5 种常见的资源清理模式。

Pattern	Example	Analysis
goto 清理法	<pre> HANDLE hToken = NULL; PVOID pMem = NULL; if (!OpenProcessToken (GetCurrentProcess (), TOKEN_QUERY, &hToken)) { ReportError (GetLastError ()) ; goto Cleanup; } pMem = LocalAlloc (LPTR, 10); if (pMem == NULL) { ReportError (GetLastError ()) ; goto Cleanup; } ... Cleanup: if (hToken) { CloseHandle (hToken); hToken = NULL; } if (pMem) { LocalFree (pMem); pMem = NULL; } </pre>	如果您确保代码绝不抛出异常，"goto 清理法"是最佳选择。效率比 "__try/__finally 法"高，实现比"带 RAII 封装器的提前返回法"简单。可以移植至 C。
__try / __finally 法	<pre> HANDLE hToken = NULL; PVOID pMem = NULL; __try { </pre>	__try/__finally 法 不可移植至其他系统，会使优化器无法运行，比 goto 和提前返回法的效率低下很

	<pre> if (!OpenProcessToken (GetCurrentProcess (), TOKEN_QUERY, &hToken)) { ReportError (GetLastError ()) ; __leave; } pMem = LocalAlloc (LPTR, 10) ; if (pMem == NULL) { ReportError (GetLastError ()) ; __leave; } ... } __finally { if (hToken) { CloseHandle (hToken) ; hToken = NULL; } if (pMem) { LocalFree (pMem) ; pMem = NULL; } } </pre>	<p>多。__try / __finally 禁止了大部分优化，因为编译器需要假设任何时候（表达式过程中，函数内，等等）所有意外情况都会发生。相反，"goto" 让编译器假设最坏情况只在执行 goto 时发生。</p>
嵌套 if 法	<pre> HANDLE hToken = NULL; PVOID pMem = NULL; if (OpenProcessToken (GetCurrentProcess (), TOKEN_QUERY, &hToken)) { pMem = LocalAlloc (LPTR, 10) ; if (pMem) { ... LocalFree (pMem) ; pMem = NULL; } else { ReportError (GetLastError ()) ; } CloseHandle (hToken) ; hToken = NULL; } else { ReportError (GetLastError ()) ; } </pre>	<p>嵌套 if 法经常是最坏的选择。您会很快耗尽水平空间。代码更难阅读和维护。</p>
带重复清理的提前返回	<pre> DWORD dwError = ERROR_SUCCESS; HANDLE hToken = NULL; PVOID pMem = NULL; </pre>	<p>提前返回法在 C 中效率低下，因为其以重复清理代码来结束。如果函</p>

法	<pre> if (!OpenProcessToken (GetCurrentProcess (), TOKEN_QUERY TOKEN_DUPLICATE, &hToken)) { ReportError (GetLastError ()); return FALSE; } pMem = LocalAlloc (LPTR, 10); if (pMem == NULL) { ReportError (GetLastError ()); CloseHandle (hToken); hToken = NULL; return FALSE; } CloseHandle (hToken); LocalFree (pMem); return TRUE; </pre>	<p>数没有清理工作，或代码量非常少，那么可以使用该清理方法。</p>
带 RAI 封装器的提前返回法	<pre> namespace WinRAII { class AutoFreeObjHandle { public: explicit AutoFreeObjHandle (HANDLE h) : m_hObj (h) { ; } ~AutoFreeObjHandle () { Close (); } void Close (void) { if (m_hObj) { CloseHandle (m_hObj); m_jObj = NULL; } } HANDLE Get (void) const { return m_hObj; } void Set (HANDLE h) { m_hObj = h; } private: HANDLE m_hObj; AutoFreeObjHandle (void); AutoFreeObjHandle (const AutoFreeObjHandle &); AutoFreeObjHandle & operator = (const AutoFreeObjHandle &); } } WinRAII::AutoFreeObjHandle afToken (NULL); if (!OpenThreadToken (GetCurrentThread (), TOKEN_QUERY, TRUE, &hToken)) { ReportError (GetLastError ()); return FALSE; } </pre>	<p>如果清理工作在析构函数内，那么提前返回法是可以使用的。因为 C++ 的异常处理不会带有 finally 关键字或 C# 的 using 关键字，每一个资源类型，您都需要一个 RAI-风格的封装器类。当函数退出时，编译器为每一个栈上的对象（该封装器）调用析构函数。这与抛出异常时的 __finally 相同。</p>

3.18 控制流

3.18.1 提前返回

☒ **您不应该**在一个方法内提前返回。某些情况下，提前返回法是可接受的，但是应该避免对其滥用。理想状态下，所有函数都应该只在函数底部有一个返回点，所有的执行路径都会通过该点返回。

可以使用提前返回的情况如下：

- 在函数开头的参数验证。
- 代码量极其少的函数（例如，成员变量状态访问器）

3.18.2 Goto

☒ **一定不要**使用 'goto' 语句来代替结构化控制流以企图优化运行时性能表现。这样做，会导致代码难于阅读，调试和验证。

☑ **您可以**以向前跳转，并实现一个跳转至同一目的地（例如，当某资源无法分配时，跳出资源分配代码，进入资源清理代码）的跳转集合来实现结构化的风格。这样使用 **goto** 便能减少嵌套深度，使得错误处理更易理解和验证。示例：

```

BOOL IsElevatedAdministrator (HANDLE hInputToken)
{
    BOOL fIsAdmin = FALSE;
    HANDLE hTokenToCheck = NULL;

    // If caller supplies a token, duplicate it to an impersonation token
    // because CheckTokenMembership requires an impersonation token.
    if (hInputToken)
    {
        if (!DuplicateToken (hInputToken, SecurityIdentification, &hTokenToCheck))
        {
            goto CLEANUP;
        }
    }

    DWORD sidLen = SECURITY_MAX_SID_SIZE;
    BYTE localAdminsGroupSid[SECURITY_MAX_SID_SIZE];

```

```

if (!CreateWellKnownSid (WinBuiltinAdministratorsSid, NULL,
    localAdminsGroupSid, &sidLen) )
{
    goto CLEANUP;
}

// Now, determine if the user is an admin
if (CheckTokenMembership (hTokenToCheck, localAdminsGroupSid, &fIsAdmin) )
{
    // lastErr = ERROR_SUCCESS;
}

CLEANUP:
    // Close the impersonation token only if we opened it.
    if (hTokenToCheck)
    {
        CloseHandle (hTokenToCheck) ;
        hTokenToCheck = NULL;
    }

    return (fIsAdmin) ;
}

```

逻辑上，代码被结构化的分为 4 块（如下阴影）。前 2 块代码试图分配资源，如果成功，控制流则执行下一块代码。如果某一块代码中资源分配失败，控制流则执行清理代码。第 3 块代码决定函数最终结果，之后执行清理代码。如上，我们便结构化的使用了 **goto**，因为所有 **goto** 都是向前跳转，并一致的出于同一个目的。结果代码相较于嵌套方法非常简短，易于阅读和验证。这便是 **goto** 的正确用法。

```

BOOL IsElevatedAdministrator (HANDLE hInputToken)
{
    BOOL fIsAdmin = FALSE;
    HANDLE hTokenToCheck = NULL;
    // If caller supplies a token, duplicate it to an impersonation token
    // because CheckTokenMembership requires an impersonation token.
    if (hInputToken)
    {
        if (!DuplicateToken (hInputToken, SecurityIdentification, &hTokenToCheck) )
        {
            goto CLEANUP;
        }
    }

    DWORD sidLen = SECURITY_MAX_SID_SIZE;

```

```

BYTE localAdminsGroupSid[SECURITY_MAX_SID_SIZE];

if (!CreateWellKnownSid (WinBuiltinAdministratorsSid, NULL,
    localAdminsGroupSid, &sidLen) )
{
    goto CLEANUP;
}

// Now, determine if the user is an admin
if (CheckTokenMembership (hTokenToCheck, localAdminsGroupSid, &fIsAdmin) )
{
    // lastErr = ERROR_SUCCESS;
}

CLEANUP:
    // Close the impersonation token only if we opened it.
    if (hTokenToCheck)
    {
        CloseHandle (hTokenToCheck) ;
        hTokenToCheck = NULL;
    }

    return (fIsAdmin) ;
}

```



4 .NET 编码规范


以下编程规范适用于 C# 和 VB.NET。

4.1 类库开发设计规范

MSDN 上的[类库开发设计规范](#) 对如何编写优秀的托管代码进行了细致的讨论。本章节的内容着重于一些重要的规范，以及一站式代码示例库对于该规范的一些例外情况。因此，建议你同时参照这两份文档。

4.2 文件和结构

 **一定不要** 在一个源文件内拥有一个以上的 **Public** 类型，除非它们只有泛型参数个数的差别，或者具有嵌套关系。一个文件内有多个内部类型是允许的。

 **一定请** 以源文件所含的 **Public** 类名命名该文件。比如， **MainForm** 类应该在 **MainForm.cs** 文件内，而 **List<T>** 类应该在 **List.cs** 文件内。


4.3 程序集属性

程序集应当包含适当的属性值来描述其命名，版权，等等。

Standard	Example
将 copyright 设置为 Copyright © Microsoft Corporation 2010	<code>[assembly: AssemblyCopyright ("Copyright © Microsoft Corporation 2010")]</code>
将 AssemblyCompany 设置为 Microsoft Corporation	<code>[assembly: AssemblyCompany ("Microsoft Corporation")]</code>
将 AssemblyTitle 和 AssemblyProduct 设置为当前示例名	<code>[assembly: AssemblyTitle ("CSNamedPipeClient")]</code> <code>[assembly: AssemblyProduct ("CSNamedPipeClient")]</code>

4.4 命名规范

4.4.1 综合命名规范

 **一定请** 为各种类型，函数，变量，特性和数据结构选取有意义的命名。其命名应能反映其作用。

❌ 您不应该在标识符名中使用缩短或缩略形式的词。比如，使用“GetWindow”而不是“GetWin”。对于公共类型，线程过程，窗口过程，和对话框过程函数，为“ThreadProc”，“DialogProc”，“WndProc”等使用公共后缀。

❌ 一定不要使用下划线，连字号，或其他任何非字母数字的字符。

4.4.2 标识符的大小写命名规范

如下表格描述了对不同类型标识符的大小写命名规范。

标识符	规范	命名结构	示例
类，结构体	Pascal 规范	名词	<pre>public class ComplexNumber {...} public struct ComplexStruct {...}</pre>
命名空间	Pascal 规范	名词 ❌ 一定不要以相同的名称来命名命名空间和其内部的类型。	<pre>namespace Microsoft.Sample.Windows7</pre>
枚举	Pascal 规范	名词 ✅ 一定请以复数名词或名词短语来命名标志枚举，以单数名词或名词短语来命名简单枚举。	<pre>[Flags] public enum ConsoleModifiers { Alt, Control }</pre>
方法	Pascal 规范	动词或动词短语	<pre>public void Print () {...} public void ProcessItem () {...}</pre>
Public 属性	Pascal 规范	名词或形容词 ✅ 一定请以集合中项目的复数形式命名该集合，或者单数名词后面跟“List”或者“Collection”。 ✅ 一定请以肯定短语来命名布尔属性，（CanSeek，而不是CantSeek）。当以“Is,”“Can,” or “Has”作布尔属性的前缀有意义时，您也可以这样做。	<pre>public string CustomerName public ItemCollection Items public bool CanRead</pre>
非 Public 属性	Camel 规范或 _camel 规范	名词或形容词 ✅ 一定请在您使用 '_' 前缀时，保持代码一致性。	<pre>private string name; private string _name;</pre>
事件	Pascal 规范	动词或动词短语 ✅ 一定请用现在式或过去式来表	<pre>// A close event that is raised after the window is closed. public event WindowClosed</pre>

		<p>明事件之前或是之后的概念。</p> <p>❌ 一定不要 使用 “Before” 或者 “After” 前缀或后缀来指明事件的先后。</p>	<pre>// A close event that is // raised before a window is // closed. public event WindowClosing</pre>
委托	Pascal 规范	<p>✅ 一定请 为用于事件的委托增加 ‘EventHandler’ 后缀。</p> <p>✅ 一定请 为除了用于事件处理程序之外的委托增加 ‘Callback’ 后缀。</p> <p>❌ 一定不要 为委托增加 “Delegate” 后缀。</p>	<pre>public delegate WindowClosedEventHandler</pre>
接口	Pascal 规范, 带有 ‘I’ 前缀	名词	<pre>public interface IDictionary</pre>
常量	Pascal 规范用于 Public 常量; Camel 规范用于 Internal 常量; 只有 1 或 2 个字符的缩写需全部字符大写。	名词	<pre>public const string MessageText = "A"; private const string messageText = "B"; public const double PI = 3.14159...;</pre>
参数, 变量	Camel 规范	名词	<pre>int customerID;</pre>
泛型参数	Pascal 规范, 带有 ‘T’ 前缀	<p>名词</p> <p>✅ 一定请 以描述性名称命名泛型参数, 除非单字符名称已有足够描述性。</p> <p>✅ 一定请 以 T 作为描述性类型参数的前缀。</p> <p>✅ 您应该 使用 T 作为单字符类型参数的名称。</p>	<pre>T, TItem, TPolicy</pre>
资源	Pascal 规范	<p>名词</p> <p>✅ 一定请 提供描述性强的标识符。同时, 尽可能保持简洁, 但是不应该因空间而牺牲可读性。</p> <p>✅ 一定请 仅为命名资源 使用字母数字字符和下划线。</p>	<pre>ArgumentExceptionInvalidName</pre>

4.4.3 匈牙利命名法

❌ 一定不要 在 .NET 中使用匈牙利命名法 (例如, 不要在变量名称内带有其类型指示符)。

4.4.4 户界面 控件命名规范

用户控件应该使用如下前缀，其重要目的是使代码更易读。

控件类型	前缀
Button	btn
CheckBox	chk
CheckedListBox	lst
ComboBox	cmb
ContextMenu	mnu
DataGrid	dg
DateTimePicker	dtg
Form	suffix: XXXForm
GroupBox	grp
ImageList	iml
Label	lb
ListBox	lst
ListView	lvw
Menu	mnu
MenuItem	mnu
NotificationIcon	nfy
Panel	pnl
PictureBox	pct
ProgressBar	prg
RadioButton	rad
Splitter	spl
StatusBar	sts
TabControl	tab
TabPage	tab
TextBox	tb
Timer	tmr
TreeView	tvw

比如，对于“File | Save”菜单选线，“Save”菜单项应该命名为“mnuFileSave”。

4.5 常量

☑ **一定请** 将那些永远不会改变值定义为常量字段。编译器直接将常量字段嵌入调用代码处。所以常量值永远不会被改变，且并不会打破兼容性。

```
public class Int32
{
    public const int MaxValue = 0x7fffffff;
    public const int MinValue = unchecked( (int) 0x80000000 );
}

Public Class Int32
    Public Const MaxValue As Integer = &H7FFFFFFF
    Public Const MinValue As Integer = &H80000000
End Class
```

☑ **一定请** 为预定义的对象实例使用 `public static (shared) readonly` 字段。如果有预定义的类型实例，也将该类型定义为 `public static readonly`。举例，

```
public class ShellFolder
{
    public static readonly ShellFolder ProgramData = new ShellFolder("ProgramData");
    public static readonly ShellFolder ProgramFiles = new ShellFolder("ProgramData");
    ...
}

Public Class ShellFolder
    Public Shared ReadOnly ProgramData As New ShellFolder("ProgramData")
    Public Shared ReadOnly ProgramFiles As New ShellFolder("ProgramFiles")
    ...
End Class
```

4.6 字符串

☒ **一定不要** 使用 ‘+’ 操作符（VB.NET 中的 ‘&’）来拼接大量字符串。相反，您应该使用 `StringBuilder` 来实现拼接工作。然而，拼接少量的字符串时，一定请使用 ‘+’ 操作符（VB.NET 中的 ‘&’）。

```
Good:
StringBuilder sb = new StringBuilder();
for (int i = 0; i < 10; i++)
{
```

```

        sb.Append(i.ToString());
    }

    Bad:
    string str = string.Empty;
    for (int i = 0; i < 10; i++)
    {
        str += i.ToString();
    }

```

☑ **一定请** 使用 显式地指定了字符串比较规则的重载函数。一般来说，需要调用带有 [StringComparison](#) 类型参数的重载函数。

☑ **一定请** 在对文化未知的字符串做比较时，使用 [StringComparison.Ordinal](#) 和 [StringComparison.OrdinalIgnoreCase](#) 作为您的安全默认值，以提高性能表现。

☑ **一定请** 在向用户输出结果时，使用基于 [StringComparison.CurrentCulture](#) 的字符串操作。

☑ **一定请** 在比较语言无关字符串（符号，等）时，使用非语言学的 [StringComparison.Ordinal](#) 或者 [StringComparison.OrdinalIgnoreCase](#) 值，而不是基于 [CultureInfo.InvariantCulture](#) 的字符串操作。一般而言，不要使用基于 [StringComparison.InvariantCulture](#) 的字符串操作。一个例外是当你坚持其语言上有意义，而与具体文化无关的情况。

☑ **一定请** 使用 [String.Equals](#) 的重载版本来测试 2 个字符串是否相等。比如，忽略大小写后，判断 2 个字符串是否相等，

```

if (str1.Equals(str2, StringComparison.OrdinalIgnoreCase))

If (str1.Equals(str2, StringComparison.OrdinalIgnoreCase)) Then

```

☒ **一定不要** 使用 [String.Compare](#) 或 [CompareTo](#) 的重载版本来检验返回值是否为 0，来判断字符串是否相等。这 2 个函数是用于字符串排序，而非检查相等性。

☑ **一定请** 在字符串比较时，以 `String.ToUpperInvariant` 函数使字符串规范化，而不用 `String.ToLowerInvariant`。

4.7 数组和集合

☑ **您应该** 在低层次函数中使用数组，来减少内存消耗，增强性能表现。对于公开接口，则偏向选择集合。

集合提供了对于其内容更多的控制权，可以随着时间改善，提高可用性。另外，不推荐在只读场景下使用数组，因为数组克隆的代价太高。

然而，如果您把熟练开发者作为目标，对于只读场景使用数组也是个不错的主意。数组的内存占用比较低，这减少了工作区，并因为运行时的优化能更快的访问数组元素。

☒ **一定不要** 使用只读的数组字段。字段本身只读，不能被修改，但是其内部元素可以被修改。以下示例展示了使用只读数组字段的陷阱：

Bad:

```
public static readonly char[] InvalidPathChars = { '\\', '<', '>', '|' };
```

这允许调用者修改数组内的值：

```
InvalidPathChars[0] = 'A';
```

您可以使用一个只读集合（只要其元素也是不可变的），或者在返回之前进行数组克隆。然而，数组克隆的代价可能过高：

```
public static ReadOnlyCollection<char> GetInvalidPathChars ()
{
    return Array.AsReadOnly (badChars);
}

public static char[] GetInvalidPathChars ()
{
    return (char[]) badChars.Clone ();
}
```

✅ **您应该** 使用不规则数组来代替使用多维数组。一个不规则数组是指其元素本身也是一个数组。构成元素的数组可能有不同大小，这样相较于多维数组能减少一些数据集的空间浪费（例如，稀疏矩阵）。另外，CLR 能够对不规则数组的索引操作进行优化，所以在某些情景下，具有更好的性能表现。

```
// 不规则数组
int[][] jaggedArray =
{
    new int[] {1, 2, 3, 4},
    new int[] {5, 6, 7},
    new int[] {8},
    new int[] {9}
};

Dim jaggedArray As Integer()() = New Integer()() _
{
    _
    New Integer() {1, 2, 3, 4}, _
    New Integer() {5, 6, 7}, _
    New Integer() {8}, _
    New Integer() {9} _
}

// 多维数组
int[,] multiDimArray =
{
    {1, 2, 3, 4},
    {5, 6, 7, 0},
    {8, 0, 0, 0},
    {9, 0, 0, 0}
};

Dim multiDimArray(,) As Integer = _
{
    _
    {1, 2, 3, 4}, _
    {5, 6, 7, 0}, _
    {8, 0, 0, 0}, _
    {9, 0, 0, 0} _
}
```

✅ **一定请** 将代表了读/写集合的属性或返回值声明为 `Collection<T>` 或其子类，将代表了只读集合的属性或返回值声明为 `ReadOnlyCollection<T>` 或其子类。

✅ **您应该** 重新考虑对于 `ArrayList` 的使用，因为所有添加至其中的对象都被当做 `System.Object`，当从 `ArrayList` 取回值时，这些对象都会拆箱，并返回其真实的值类型。所以我们推荐您使用定制类型的集合，而不是 `ArrayList`。比如，.NET 在 `System.Collection.Specialized` 命名空间内为 `String` 提供了强类型集合 `StringCollection`。

✅ **您应该** 重新考虑对于 `Hashtable` 的使用。相反，您应该尝试其他字典类，例如 `StringDictionary`，`NameValueCollection`，`HybridCollection`。除非 `Hashtable` 只存储少量值，最好不要使用 `Hashtable`。

✅ **您应该** 在实现集合类型时，为其实现 `IEnumerable` 接口，这样该集合便能用于 LINQ to Objects。

❌ **一定不要** 在同一个类型上同时实现 `IEnumerator<T>` 和 `IEnumerable<T>` 接口。同样，也不要同时实现非泛型接口 `IEnumerator` 和 `IEnumerable`。所以，一个类型只能成为一个集合或者一个枚举器，而不可二者皆得。

❌ **一定不要** 返回数组或集合的 `null` 引用。空值数组或集合的含义在代码环境中很难被理解。比如，一个用户可能假定如下代码能够正常运行，所以应该返回一个空数组或集合，而不是 `null` 引用。

```
int[] arr = SomeOtherFunc ();
foreach (int v in arr)
{
    ...
}
```

4.8 结构体

✅ **一定请** 确保将所有实例数据设置为 0 值，`false` 或者是 `null`。当创建结构体数组时，这样能防止意外创建了无效实例。

✅ **一定请** 为值类型实现 `IEquatable<T>` 接口。值类型的 `Object.Equals` 方法会引起装箱操作。且因为使用了反射特性，所以其默认实现效率不高。`IEquatable<T>.Equals` 较其有相当大的性能提升，且其实现可以不引发装箱操作。

4.8.1 结构体 vs 类

❌ **一定不要** 定义结构体，除非其具有如下特性：

- 它在逻辑上代表了一个单值，类似于原始类型（例如，`int`、`double`，等等）。
- 其示例大小小于 16 字节。
- 它是不可变的。
- 它不会引发频繁的装箱拆箱操作。

其余情况下，您应该定义类，而不是结构体。

4.9 类

✅ **一定请** 使用继承来表示“is a”关系，例如“猫是一种动物”。

✅ **一定请** 使用 接口，例如 `IDisposable`，来表示“can do”关系，例如“对象能被释放”。

4.9.1 字段

❌ **一定不要** 提供 `Public` 或 `Protected` 的实例字段。`Public` 或 `Protected` 字段并没有受到代码访问安全需求的保护。我们应该使用 `Private` 字段，并通过属性来提供访问接口。

✅ **一定请** 将预定义对象实例定义为 `public static readonly` 字段。

✅ **一定请** 将永远不会改变的字段定义为常量字段。

❌ **一定不要** 将可变类型定义为只读字段。

4.9.2 属性

✅ **一定请** 创建只读属性，如果用户不应该具有修改这些属性值的能力。

❌ **一定不要** 提供只写属性。如果没有提供属性设置器，请使用一个方法来实现相同的功能。方法名应该以 Set 开头，后面跟着属性名。

✅ **一定请** 为所有属性提供合理的默认值，并确保默认值不会引发安全漏洞或一个极端低效的设计。

❌ **您不应该** 从属性访问器内抛出异常。属性访问器应该只包含简单的操作，且不带任何前置条件。如果一个属性访问器可能抛出异常，那么考虑将其重新设计为一个方法。该推荐方法并不适用于索引器。索引器可以因为无效实参而抛出异常。从属性设置器内抛出异常是有效且可接受的。

4.9.3 构造函数

✅ **一定请** 尽量减少构造函数的工作量。除了得到构造函数参数，设置主要数据成员，构造函数不应该有太多的工作量。其余工作量应该被推迟，直到必须。

✅ **一定请** 在恰当的时候，从实例构造函数内抛出异常。

✅ **一定请** 在需要默认构造函数的情况下，显式的声明它。即使有时编译器为自动的为您的类增加一个默认构造函数，但是显式的声明使得代码更易维护。这样即使您增加了一个带有参数的构造函数，也能确保默认构造函数仍然会被定义。

❌ **一定不要** 在对象构造函数内部调用虚方法。调用虚方法时，实际调用了继承体系最底层的覆盖（override）方法，而不考虑定义了该方法的类的构造函数是否已被调用。

4.9.4 方法

✅ **一定请** 将所有输出参数放置于所有传值和传引用参数（除去参数数组）的之后，即使它引起了重载方法之前不一致的参数顺序。

☑ **一定请** 验证传递给 `Public`, `Protected` 或显式实现的成员方法的实参。如果验证失败，则抛出 `System.ArgumentException`，或者其子集：如果一个 `null` 实参传递给成员，而成员方法不支持 `null` 实参，则抛出 `ArgumentNullException`。如果实参值超出由调用方法定义的可接受范围，则抛出 `ArgumentOutOfRangeException` 异常。

4.9.5 事件

☑ **一定请** 注意事件处理方法中可能会执行任意代码。考虑将引发事件的代码放入一个 `try-catch` 块中，以避免由事件处理方法中抛出的未处理异常引起的程序终止。

☒ **一定不要** 在有性能要求的 API 中使用事件。虽然事件易于理解和使用，但是就性能和内存消耗而言，它们不如虚函数。

4.9.6 成员方法重载

☑ **一定请** 使用成员方法重载，而不是定义带有默认参数的成员方法。默认参数并不是 CLS 兼容的，所以不能被某些语言重用。同时，带有默认参数的成员方法存在一个版本问题。我们考虑成员方法的版本 1 将可选参数默认设置为 123。当编译代码调用该方法，且没有指定可选参数时，编译器在调用处直接将 123 嵌入代码中。现在，版本 2 将默认参数修改为 863，如调用代码没有重新编译，那么它会调用版本 2 的方法，并传递 123 作为其参数。（123 是版本 1 的默认参数，而不是版本 2 的。）。

Good:

```
Public Overloads Sub Rotate (ByVal data As Matrix)
    Rotate (data, 180)
End Sub
```

```
Public Overloads Sub Rotate (ByVal data As Matrix, ByVal degrees As Integer)
    ' Do rotation here
End Sub
```

Bad:

```
Public Sub Rotate (ByVal data As Matrix, Optional ByVal degrees As Integer = 180)
    ' Do rotation here
End Sub
```

❌ **一定不要**任意改动重载方法中的参数名。如果一个重载函数中的参数代表着另一个重载函数中相同的参数，该参数则应该有相同的命名。具有相同命名的参数应该在重载函数中出现在同一位置。

✅ **一定请**仅将最长重载函数设为虚函数（为了拓展性考虑）。短重载函数应该一直调用到长重载函数。

4.9.7 接口成员

❌ **您不应该**在没有合理理由的情况下显式的实现接口成员。显式实现的成员可能使开发者感到困惑，因为他们不会出现在 **Public** 成员列表内，且会造成对值类型不必要的装箱拆箱操作。

✅ **您应该**在成员只通过接口来调用的情况下，显式的实现成员接口。

4.9.8 虚成员方法

相较于回调和事件，虚成员方法性能上有更好的表现。但是比非虚方法在性能上低一点。

❌ **一定不要**在没有合理理由的情况下，将成员方法设置为虚方法，您必须意识到相关设计，测试，维护虚方法带来的成本。

✅ **您应该** 倾向于为虚成员方法设置为 **Protected** 的访问性，而不是 **Public** 访问性。**Public** 成员应该通过调用 **Protected** 的虚方法来提供拓展性（如果需要的话）。

4.9.9 静态类

✅ **一定请** 合理使用静态类。静态类应该被用于框架内基于对象的核心支持辅助类。

4.9.10 抽象类

❌ **一定不要** 在抽象类中定义 **Public** 或 **Protected-Internal** 的构造函数。

✅ **一定请** 为抽象类定义一个 **Protected**，或 **Internal** 构造函数。

Protected 构造函数更常见，因为其允许当子类创建时，基类可以完成自己的初始化工作。

```
public abstract class Claim
{
    protected Claim ()
    {
        ...
    }
}
```

internal 构造函数用于限制将抽象类的实现具化到定义该类的程序集。

```
public abstract class Claim
{
    internal Claim ()
    {
        ...
    }
}
```

4.10 命名空间

☑ **一定请** 在一站式代码示例库中使用 Visual Studio 创建的项目的默认的命名空间。无需重命名命名空间为 Microsoft.Sample.TechnologyName。

4.11 错误和异常

4.11.1 抛出异常

☑ **一定请** 通过抛出异常来告知执行失败。异常在框架内是告知错误的主要手段。如果一个成员方法不能成功的如预期般执行，便应该认为是执行失败，并抛出一个异常。一定不要返回一个错误代码。

☑ **一定请** 抛出最明确，最有意义的异常（继承体系最底层的）。比如，如果传递了一个 **null** 实参，则抛出 **ArgumentNullException**，而不是其基类 **ArgumentException**。抛出并捕获 **System.Exception** 异常通常都是没有意义的。

❌ **一定不要** 将异常用于常规的控制流。除了系统故障 或者带有潜在竞争条件的操作，您编写的代码都不应该抛出异常。比如，在调用可能失败或抛出异常的方法前， 您可以检查其前置条件，举例，

```
// C# 示例:
if (collection != null && !collection.IsReadOnly)
{
    collection.Add(additionalNumber);
}

' VB.NET 示例:
If ((Not collection Is Nothing) And (Not collection.IsReadOnly)) Then
    collection.Add(additionalNumber)
End If
```

❌ **一定不要** 从异常过滤器块内抛出异常。当一个异常过滤器引发了一个异常时，该异常会被 CLR 捕获，该过滤器返回 **false**。该行为很难与过滤器显式的执行并返回错误区分开，所以会增加调试难度。

```
' VB.NET sample
' This is bad design. The exception filter (When clause)
' may throw an exception when the InnerException property
' returns null
Try
    ...
Catch e As ArgumentException _
When e.InnerException.Message.StartsWith("File")
    ...
End Try
```

❌ **一定不要** 显式的从 **finally** 块内抛出异常。从调用方法内隐式的抛出异常是可以接受的。

4.11.2 异常处理

❌ **您不应该** 通过捕获笼统的异常，例如 **System.Exception**，**System.SystemException**，或者 .NET 代码中其他异常，来隐藏错误。一定要捕获代码能够处理的、明确的异常。您应该捕获更加明确的异常，或者在 **Catch** 块的最后一条语句处重新抛出该普通异常。以下情况隐藏错误是可以接受的，但是其发生几率很低：

Good:

```

// C# sample:
try
{
    ...
}
catch (System.NullReferenceException exc)
{
    ...
}
catch (System.ArgumentOutOfRangeException exc)
{
    ...
}
catch (System.InvalidCastException exc)
{
    ...
}

' VB.NET sample:
Try
    ...
Catch exc As System.NullReferenceException
    ...
Catch exc As System.ArgumentOutOfRangeException
    ...
Catch exc As System.InvalidCastException
    ...
End Try

Bad:
// C# sample:
try
{
    ...
}
catch (Exception ex)
{
    ...
}

' VB.NET sample:
Try
    ...
Catch ex As Exception
    ...
End Try

```


☑ **一定请** 在捕获并重新抛出异常时，倾向使用 `throw`。这是保持异常调用栈的最佳途径：

Good:

```
// C# sample:
try
{
    ... // Do some reading with the file
}
catch
{
    file.Position = position; // Unwind on failure
    throw; // Rethrow
}

' VB.NET sample:
Try
    ... ' Do some reading with the file
Catch ex As Exception
    file.Position = position ' Unwind on failure
    Throw ' Rethrow
End Try
```

Bad:

```
// C# sample:
try
{
    ... // Do some reading with the file
}
catch (Exception ex)
{
    file.Position = position; // Unwind on failure
    throw ex; // Rethrow
}

' VB.NET sample:
Try
    ... ' Do some reading with the file
Catch ex As Exception
    file.Position = position ' Unwind on failure
    Throw ex ' Rethrow
End Try
```

4.12 资源清理

☒ **一定不要** 使用 `GC.Collect` 来进行强制垃圾回收。

4.12.1 Try-finally 块

☑ 一定请使用 try-finally 块来清理资源，try-catch 块来处理错误恢复。一定不要使用 catch 块来清理资源。一般来说，清理的逻辑是回滚资源（特别是原生资源）分配。举例：

```
// C# sample:
FileStream stream = null;
try
{
    stream = new FileStream (...);
    ...
}
finally
{
    if (stream != null)
        stream.Close ();
}
```

```
' VB.NET sample:
Dim stream As FileStream = Nothing
Try
    stream = New FileStream (...)
    ...
Catch ex As Exception
    If (stream IsNot Nothing) Then
        stream.Close ()
    End If
End Try
```

为了清理实现了 IDisposable 接口的对象，C# 和 VB.NET 提供了 using 语句来替代 try-finally 块。

```
// C# sample:
using (FileStream stream = new FileStream (...))
{
    ...
}
```

```
' VB.NET sample:
Using stream As New FileStream (...)
    ...
End Using
```

许多语言特性都会自动的为您写入 try-finally 块。例如 C#/VB 的 using 语句，C# 的 lock 语句，VB 的 SyncLock 语句，C# 的 foreach 语句以及 VB For Each 语句。

4.12.2 基础 Dispose 模式

该模式的基础实现包括实现 System.IDisposable 接口，声明实现了所有资源清理逻辑的 Dispose (bool) 方法，该方法被 Dispose 方法和可选的终结器所共享。请注意，本章节并不讨论如何编写一个终结器。可终结类型是该简单模式的拓展，我们会在下个章节中讨论。如下展示了基础模式的简单实现：

```
// C# sample:
public class DisposableResourceHolder : IDisposable
{
    private bool disposed = false;
    private SafeHandle resource; // Handle to a resource

    public DisposableResourceHolder ()
    {
        this.resource = ... // Allocates the native resource
    }

    public void DoSomething ()
    {
        if (disposed)
            throw new ObjectDisposedException (...);

        // Now call some native methods using the resource
        ...
    }

    public void Dispose ()
    {
        Dispose (true);
        GC.SuppressFinalize (this);
    }

    protected virtual void Dispose (bool disposing)
    {
        // Protect from being called multiple times.
        if (disposed) return;

        if (disposing)
        {

```

```

        // Clean up all managed resources.
        if (resource != null)
            resource.Dispose ();
    }

    disposed = true;
}

' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable

    Private disposed As Boolean = False
    Private resource As SafeHandle ' Handle to a resource

    Public Sub New ()
        resource = ... ' Allocates the native resource
    End Sub

    Public Sub DoSomething ()
        If (disposed) Then
            Throw New ObjectDisposedException (...)
        End If

        ' Now call some native methods using the resource
        ...
    End Sub

    Public Sub Dispose () Implements IDisposable.Dispose
        Dispose (True)
        GC.SuppressFinalize (Me)
    End Sub

    Protected Overridable Sub Dispose (ByVal disposing As Boolean)
        ' Protect from being called multiple times.
        If disposed Then Return

        If disposing Then
            ' Clean up all managed resources.
            If (resource IsNot Nothing) Then
                resource.Dispose ()
            End If
        End If

        disposed = True
    End Sub

```

```
End Class
```

☑ **一定请** 为包含了可释放类型的类型实现基础 `Dispose` 模式。

☑ **一定请** 拓展基础 `Dispose` 模式来提供一个终结器。比如，为存储非托管内存的缓冲实现该模式。

☑ **您应该** 为即使类本身不包含非托管代码或可清理对象，但是其子类可能带有的类实现该基础 `Dispose` 模式。一个绝佳的例子便是 `System.IO.Stream` 类。虽然它是一个不带任何资源的抽象类，大多数子类却带有资源，所以应该为其实现该模式。

☑ **一定请** 声明一个 `protected virtual void Dispose (bool disposing)` 方法来集中所有释放非托管资源的逻辑。所有资源清理都应该在该方法中完成。用终结器 和 `IDisposable.Dispose` 方法来调用该方法。如果从终结器内调用，则其参数为 `false`。它应该用于确保任何在终结中运行的代码不应该被其他可终结对象访问到。下一章节我们会详细介绍终结器的细节。

```
// C# sample:
protected virtual void Dispose (bool disposing)
{
    if (disposing)
    {
        // Clean up all managed resources.
        if (resource != null)
            resource.Dispose ();
    }
}

' VB.NET sample:
Protected Overridable Sub Dispose (ByVal disposing As Boolean)
    ' Protect from being called multiple times.
    If disposed Then Return

    If disposing Then
        ' Clean up all managed resources.
        If (resource IsNot Nothing) Then
            resource.Dispose ()
        End If
    End Sub
```

```

        End If

        disposed = True
    End Sub

```

☑ **一定请** 通过简单的调用 `Dispose (true)`，以及 `GC.SuppressFinalize (this)` 来实现 `IDisposable` 接口。仅当 `Dispose (true)` 成功执行完后才能调用 `SuppressFinalize`。

```

// C# sample:
public void Dispose ()
{
    Dispose (true);
    GC.SuppressFinalize (this);
}

' VB.NET sample:
Public Sub Dispose () Implements IDisposable.Dispose
    Dispose (True)
    GC.SuppressFinalize (Me)
End Sub

```

☒ **一定不要** 将无参 `Dispose` 方法定义为虚函数。`Dispose (bool)` 方法应该被子类重写。

☒ **您不应该** 从 `Dispose (bool)` 内抛出异常，除非包含它的进程被破坏（内存泄露，不一致的共享状态，等等）等极端条件。用户不希望调用 `Dispose` 会引发异常。比如，考虑在 C# 代码内手动写入 `try-finally` 块：

```

TextReader tr = new StreamReader (File.OpenRead ("foo.txt"));
try
{
    // Do some stuff
}
finally
{
    tr.Dispose ();
    // More stuff
}

```

如果 `Dispose` 可能引发异常，那么 `finally` 块的清理逻辑不会被执行。为了解决这一点，用户需要将所有对于 `Dispose`（在它们的 `finally` 块内！）的调用放入 `try` 块内，这将导致一个非常复杂的清理处理程序。如果

执行 `Dispose (bool disposing)` 方法，即使终结失败也不会抛出异常。如果在一个终结器环境内这样做会终止当前流程。

☑ **一定请** 在对象被终结之后，为任何不能再被使用的成员抛出一个 `ObjectDisposedException` 异常。

```
// C# sample:
public class DisposableResourceHolder : IDisposable
{
    private bool disposed = false;
    private SafeHandle resource; // Handle to a resource

    public void DoSomething ()
    {
        if (disposed)
            throw new ObjectDisposedException (...);

        // Now call some native methods using the resource
        ...
    }

    protected virtual void Dispose (bool disposing)
    {
        if (disposed) return;
        // Cleanup
        ...
        disposed = true;
    }
}

' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable

    Private disposed As Boolean = False
    Private resource As SafeHandle ' Handle to a resource

    Public Sub DoSomething ()
        If (disposed) Then
            Throw New ObjectDisposedException (...)
        End If

        ' Now call some native methods using the resource
        ...
    End Sub
End Class
```

```

Protected Overridable Sub Dispose (ByVal disposing As Boolean)
    ' Protect from being called multiple times.
    If disposed Then Return

    ' Cleanup
    ...

    disposed = True
End Sub

End Class

```

4.12.3 可终结类型

可终结类型是通过重写终结器并在 `Dispose (bool)` 中提供终结代码路径来拓展基础 `Dispose` 模式的类型。

如下代码是一个可终结类型的示例：

```

// C# sample:
public class ComplexResourceHolder : IDisposable
{
    bool disposed = false;
    private IntPtr buffer; // Unmanaged memory buffer
    private SafeHandle resource; // Disposable handle to a resource

    public ComplexResourceHolder ()
    {
        this.buffer = ... // Allocates memory
        this.resource = ... // Allocates the resource
    }

    public void DoSomething ()
    {
        if (disposed)
            throw new ObjectDisposedException (...);

        // Now call some native methods using the resource
        ...
    }

    ~ComplexResourceHolder ()
    {
        Dispose (false);
    }
}

```



```

public void Dispose ()
{
    Dispose (true) ;
    GC.SuppressFinalize (this) ;
}

protected virtual void Dispose (bool disposing)
{
    // Protect from being called multiple times.
    if (disposed) return;

    if (disposing)
    {
        // Clean up all managed resources.
        if (resource != null)
            resource.Dispose () ;
    }

    // Clean up all native resources.
    ReleaseBuffer (buffer) ;

    disposed = true;
}
}

' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable

    Private disposed As Boolean = False
    Private buffer As IntPtr ' Unmanaged memory buffer
    Private resource As SafeHandle ' Handle to a resource

    Public Sub New ()
        buffer = ... ' Allocates memory
        resource = ... ' Allocates the native resource
    End Sub

    Public Sub DoSomething ()
        If (disposed) Then
            Throw New ObjectDisposedException (...)
        End If

        ' Now call some native methods using the resource
        ...
    End Sub

```

```

Protected Overrides Sub Finalize ()
    Dispose (False)
    MyBase.Finalize ()
End Sub

Public Sub Dispose () Implements IDisposable.Dispose
    Dispose (True)
    GC.SuppressFinalize (Me)
End Sub

Protected Overridable Sub Dispose (ByVal disposing As Boolean)
    ' Protect from being called multiple times.
    If disposed Then Return

    If disposing Then
        ' Clean up all managed resources.
        If (resource IsNot Nothing) Then
            resource.Dispose ()
        End If
    End If

    ' Clean up all native resources.
    ReleaseBuffer (Buffer)

    disposed = True
End Sub

End Class

```

☑ **一定请** 在类型应该为释放非托管资源负责，且自身没有终结器的情况下，将该类型定义为可终结的。当实现终结器时，简单的调用 `Dispose (false)`，并将所有资源清理逻辑放入 `Dispose (bool disposing)` 方法。

```

// C# sample:
public class ComplexResourceHolder : IDisposable
{
    ...
    ~ComplexResourceHolder ()
    {
        Dispose (false);
    }

    protected virtual void Dispose (bool disposing)
    {

```

```

        ...
    }
}

' VB.NET sample:
Public Class DisposableResourceHolder
    Implements IDisposable

    ...

    Protected Overrides Sub Finalize ()
        Dispose (False)
        MyBase.Finalize ()
    End Sub

    Protected Overridable Sub Dispose (ByVal disposing As Boolean)
        ...
    End Sub

End Class

```

☑ **一定请** 谨慎的定义可终结类型。仔细考虑任何一个您需要终结器的情况。带有终结器的实例从性能和复杂性角度来说，都需付出不小的代价。

☑ **一定请** 为每一个可终结类型实现基础 **Dispose** 模式。该模式的细节请参考先前章节。这给予该类型的使用者以一种显式的方式去清理其拥有的资源。

☑ **您应该** 在终结器即使面临强制的应用程序域卸载或线程中止的情况也必须被执行时，创建并使用临界可终结对象（一个带有包含了 **CriticalFinalizerObject** 的类型层次的类型）。

☑ **一定请** 尽量使用基于 **SafeHandle** 或 **SafeHandleZeroOrMinusOneIsInvalid**（对于 Win32 资源句柄，其值如果为 0 或者 -1，则代表其为无效句柄）的资源封装器，而不是自己来编写终结器。这样，我们便无需终结器，封装器会为其资源清理负责。安全句柄实现了 **IDisposable** 接口，并继承自 **CriticalFinalizerObject**，所以即使面临强制的应用程序域卸载或线程中止，终结器的逻辑也会被执行。

```

/// <summary>
/// Represents a wrapper class for a pipe handle.
/// </summary>

```

```

[SecurityCritical (SecurityCriticalScope.Everything) ,
HostProtection (SecurityAction.LinkDemand, MayLeakOnAbort = true) ,
SecurityPermission (SecurityAction.LinkDemand, UnmanagedCode = true) ]
internal sealed class SafePipeHandle : SafeHandleZeroOrMinusOneIsInvalid
{
    private SafePipeHandle ()
        : base (true)
    {
    }

    public SafePipeHandle (IntPtr preexistingHandle, bool ownsHandle)
        : base (ownsHandle)
    {
        base.SetHandle (preexistingHandle) ;
    }

    [ReliabilityContract (Consistency.WillNotCorruptState, Cer.Success) ,
DllImport ("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true) ]
    [return: MarshalAs (UnmanagedType.Bool) ]
    private static extern bool CloseHandle (IntPtr handle) ;

    protected override bool ReleaseHandle ()
    {
        return CloseHandle (base.handle) ;
    }
}

/// <summary>
/// Represents a wrapper class for a local memory pointer.
/// </summary>
[SuppressUnmanagedCodeSecurity,
HostProtection (SecurityAction.LinkDemand, MayLeakOnAbort = true) ]
internal sealed class SafeLocalMemHandle : SafeHandleZeroOrMinusOneIsInvalid
{
    public SafeLocalMemHandle ()
        : base (true)
    {
    }

    public SafeLocalMemHandle (IntPtr preexistingHandle, bool ownsHandle)
        : base (ownsHandle)
    {
        base.SetHandle (preexistingHandle) ;
    }

    [ReliabilityContract (Consistency.WillNotCorruptState, Cer.Success) ,
DllImport ("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true) ]

```

```

private static extern IntPtr LocalFree (IntPtr hMem) ;

protected override bool ReleaseHandle ()
{
    return (LocalFree (base.handle) == IntPtr.Zero) ;
}
}

```

❌ **一定不要** 在终结器代码路径内访问任何可终结对象，因为这样会有一个极大的风险：它们已经被终结了。例如，一个可终结对象 A，其拥有一个指向另一个可终结对象 B 的对象，在 A 的终结器内并不能很安心的使用 B，反之亦然。终结器是被随机调用的。

但是操作拆箱的值类型字段是可以接受的。

同时也请注意，在应用程序域卸载或进程退出的某些时间点上，存储于静态变量的对象会被回收。如果 `Environment.HasShutdownStarted` 返回 `True`，则访问引用了一个可终结对象的静态变量（或调用使用了存储于静态变量的值的静态函数）可能会不安全。

❌ **一定不要** 从终结器的逻辑内抛出异常，除非是系统严重故障。如果从终结器内抛出异常，CLR 可能会停止整个进程来阻止其他终结器被执行，并阻止资源以一个受控制的方式释放。

4.12.4 重写 Dispose

如果您继承了实现 `IDisposable` 接口的基类，您必须也实现 `IDisposable` 接口。记得要调用您基类的 `Dispose (bool)`。

```

public class DisposableBase : IDisposable
{
    ~DisposableBase ()
    {
        Dispose (false) ;
    }

    public void Dispose ()
    {
        Dispose (true) ;
        GC.SuppressFinalize (this) ;
    }
}

```

```

    }

    protected virtual void Dispose (bool disposing)
    {
        // ...
    }
}

public class DisposableSubclass : DisposableBase
{
    protected override void Dispose (bool disposing)
    {
        try
        {
            if (disposing)
            {
                // Clean up managed resources.
            }

            // Clean up native resources.
        }
        finally
        {
            base.Dispose (disposing) ;
        }
    }
}

```

4.13 交互操作

4.13.1 P/Invoke

☑ 一定请在编写 P/Invoke 签名时，查阅 [P/Invoke 交互操作助理](#) 以及 <http://pinvoke.net> 。

☑ 您可以使用 `IntPtr` 来进行手动封送处理。虽然会牺牲易用性，类型安全和维护性，但是通过将参数和字段声明为 `IntPtr`，您可以提升性能表现。有时，通过 `Marshal` 类的方法来执行手动封送处理比依赖默认交互操作封送处理的性能更高。举例来说，如果有一个字符串的大数组需要被传递跨越交互操作边界，但是托管代码只需其中几个元素，那么您就可以将数组声明为 `IntPtr`，手动的访问所需的几个元素。

❌ **一定不要** 锁定短暂存在的对象。锁定短暂存在的对象会延长在 `P/Invoke` 调用时内存缓冲区的生存期。

锁定会阻止垃圾回收器重新分配托管堆内对象内存，或者是托管委托的地址。然而，锁定长期存在的对象是可以接受的，因为其是在应用程序初始化期间创建的，相较于短暂存在对象，它们并不会被移动。

长期锁定短暂存在的对象代价非常高昂，因为内存压缩经常发生在第 0 代的垃圾回收时，垃圾回收器不能重新分配被锁定的对象。这样会造成低效的内存管理，极大的降低性能表现。更多复制和锁定请参考

<http://msdn.microsoft.com/en-us/library/23acw07k.aspx>。

✅ **一定请** 在 `P/Invoke` 签名中将 `CharSet` 设为 `CharSet.Auto`，`SetLastError` 设为 `true`。举例，

```
// C# sample:
[DllImport("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true)]
public static extern SafeFileMappingHandle OpenFileMapping (
    FileMapAccess dwDesiredAccess, bool bInheritHandle, string lpName);

' VB.NET sample:
<DllImport("kernel32.dll", CharSet:=CharSet.Auto, SetLastError:=True) > _
Public Shared Function OpenFileMapping ( _
    ByVal dwDesiredAccess As FileMapAccess, _
    ByVal bInheritHandle As Boolean, _
    ByVal lpName As String) _
    As SafeFileMappingHandle
End Function
```

✅ **您应该** 将非托管资源封装进 `SafeHandle` 类。`SafeHandle` 类已经在[可终结类型](#)章节中进行了讨论。比如，文件映射句柄被封装成如下代码：

```
/// <summary>
/// Represents a wrapper class for a file mapping handle.
/// </summary>
[SuppressUnmanagedCodeSecurity,
HostProtection (SecurityAction.LinkDemand, MayLeakOnAbort = true)]
internal sealed class SafeFileMappingHandle : SafeHandleZeroOrMinusOneIsInvalid
{
    [SecurityPermission (SecurityAction.LinkDemand, UnmanagedCode = true)]
    private SafeFileMappingHandle ()
        : base (true)
    {
    }
}
```

```

[SecurityPermission (SecurityAction.LinkDemand, UnmanagedCode = true) ]
public SafeFileMappingHandle (IntPtr handle, bool ownsHandle)
    : base (ownsHandle)
{
    base.SetHandle (handle) ;
}

[ReliabilityContract (Consistency.WillNotCorruptState, Cer.Success) ,
DllImport ("kernel32.dll", CharSet = CharSet.Auto, SetLastError = true) ]
[return: MarshalAs (UnmanagedType.Bool) ]
private static extern bool CloseHandle (IntPtr handle) ;

protected override bool ReleaseHandle ()
{
    return CloseHandle (base.handle) ;
}
}

''' <summary>
''' Represents a wrapper class for a file mapping handle.
''' </summary>
''' <remarks></remarks>
<SuppressUnmanagedCodeSecurity () , _
HostProtection (SecurityAction.LinkDemand, MayLeakOnAbort:=True) > _
Friend NotInheritable Class SafeFileMappingHandle
    Inherits SafeHandleZeroOrMinusOneIsInvalid

    <SecurityPermission (SecurityAction.LinkDemand, UnmanagedCode:=True) > _
    Private Sub New ()
        MyBase.New (True)
    End Sub

    <SecurityPermission (SecurityAction.LinkDemand, UnmanagedCode:=True) > _
    Public Sub New (ByVal handle As IntPtr, ByVal ownsHandle As Boolean)
        MyBase.New (ownsHandle)
        MyBase.SetHandle (handle)
    End Sub

    <ReliabilityContract (Consistency.WillNotCorruptState, Cer.Success) , _
    DllImport ("kernel32.dll", CharSet:=CharSet.Auto, SetLastError:=True) > _
    Private Shared Function CloseHandle (ByVal handle As IntPtr) _
    As <MarshalAs (UnmanagedType.Bool) > Boolean
    End Function

    Protected Overrides Function ReleaseHandle () As Boolean
        Return SafeFileMappingHandle.CloseHandle (MyBase.handle)
    End Function

```



```
End Function
```

```
End Class
```

☑ 您应该在调用会设置 Win32 最后错误代码的 P/Invoked 函数失败时，抛出 Win32Exception 异常。如果函数使用了非托管资源，请在 finally 块内释放资源。

```
// C# sample:
SafeFileMappingHandle hMapFile = null;
try
{
    // Try to open the named file mapping.
    hMapFile = NativeMethod.OpenFileMapping (
        FileMapAccess.FILE_MAP_READ,    // Read access
        false,                          // Do not inherit the name
        FULL_MAP_NAME                   // File mapping name
    );
    if (hMapFile.IsInvalid)
    {
        throw new Win32Exception ();
    }

    ...
}
finally
{
    if (hMapFile != null)
    {
        // Close the file mapping object.
        hMapFile.Close ();
        hMapFile = null;
    }
}


' VB.NET sample:
Dim hMapFile As SafeFileMappingHandle = Nothing
Try
    ' Try to open the named file mapping.
    hMapFile = NativeMethod.OpenFileMapping ( _
        FileMapAccess.FILE_MAP_READ, _
        False, _
        FULL_MAP_NAME)
    If (hMapFile.IsInvalid) Then
        Throw New Win32Exception
    End If
End Try
```


```


...
Finally
    If (Not hMapFile Is Nothing) Then
        ' Close the file mapping object.
        hMapFile.Close ()
        hMapFile = Nothing
    End If
End Try

```

4.13.2 COM 交互操作

 **一定不要** 以 `GC.Collect` 来进行强制垃圾回收以释放有性能要求的 API 内的 COM 对象。释放 COM 对象一个常见的方法是将 RCW 引用设置为 null，并调用 `System.GC.Collect` 以及 `System.GC.WaitForPendingFinalizers`。如果对性能有较高要求，我们并不推荐这样做，因为这样会引起垃圾回收运行太频繁。如果在服务器应用程序代码中使用该方法则极大的降低了性能和可拓展性。您应该让垃圾回收器自己决定何时执行垃圾回收。

 **您应该** 使用 `Marshal.FinalReleaseComObject` 或者 `Marshal.ReleaseComObject` 来手动管理 RCW 的生存周期。相较于 `GC.Collect` 来进行强制垃圾回收，这样做具有较高的效率。

 **一定不要** 进行跨套间调用。当您从托管应用程序调用一个 COM 对象时，请确保托管代码的套间类型和 COM 对象套间类型是相匹配的。通过使用匹配的套间，您可以避免跨套间调用时的线程切换。