Coding Guidelines

For Microsoft C# .NET and Native C++

Table of Contents

[2 Purpose 4](#_Toc423016415)

[3 Objectives 4](#_Toc423016416)

[4 Accompanying Document 4](#_Toc423016417)

[5 Assumptions 4](#_Toc423016418)

[6 Casing 5](#_Toc423016419)

[7 Guidelines Common Across Languages 5](#_Toc423016420)

[7.1 File Header 5](#_Toc423016421)

[7.2 Naming Conventions 6](#_Toc423016422)

[7.2.1 General, non-UI Code 6](#_Toc423016423)

[7.2.2 UI Code 7](#_Toc423016424)

[7.3 Referencing 8](#_Toc423016425)

[7.4 Whitespace/Layout 8](#_Toc423016426)

[7.5 General Guidelines 8](#_Toc423016427)

[7.5.1 Variable Declaration 8](#_Toc423016428)

[7.5.2 Line Length 9](#_Toc423016429)

[7.5.3 Function Declaration 9](#_Toc423016430)

[7.5.4 Statements 9](#_Toc423016431)

[7.5.5 Enums 9](#_Toc423016432)

[7.6 Comments 10](#_Toc423016433)

[7.7 Goto 10](#_Toc423016434)

[7.8 Indentation 10](#_Toc423016435)

[8 C# Guidelines 12](#_Toc423016436)

[8.1 Naming Conventions 12](#_Toc423016437)

[8.2 Classes 12](#_Toc423016438)

[8.2.1 Layout 12](#_Toc423016439)

[8.2.2 Member Design 12](#_Toc423016440)

[8.2.3 Static Classes 14](#_Toc423016441)

[8.2.4 Abstract Classes 15](#_Toc423016442)

[8.3 Types 15](#_Toc423016443)

[8.3.1 Strings 15](#_Toc423016444)

[8.3.2 Arrays and Collections 16](#_Toc423016445)

[8.3.3 Structures vs Classes 18](#_Toc423016446)

[8.3.4 Implicitly Typed Local Variables (var) 19](#_Toc423016447)

[8.3.5 Anonymous Types (var) 19](#_Toc423016448)

[8.4 Secure Coding 19](#_Toc423016449)

[8.5 Documentation Comments 19](#_Toc423016450)

[8.6 General Guidelines 19](#_Toc423016451)

[8.7 Constants 21](#_Toc423016452)

[8.8 Programming Topics 21](#_Toc423016453)

[8.8.1 Exceptions 21](#_Toc423016454)

[8.8.2 Dispose Pattern 23](#_Toc423016455)

[8.8.3 Finalizable Types 27](#_Toc423016456)

[8.8.4 Overriding Dispose 32](#_Toc423016457)

[8.8.5 Size of Artifacts 32](#_Toc423016458)

[8.8.6 Hardcoded Values 32](#_Toc423016459)

[8.8.7 Object & Collection Initializers 33](#_Toc423016460)

[8.8.8 Lambda Expressions 33](#_Toc423016461)

[8.8.9 Reflection 33](#_Toc423016462)

[8.8.10 Language Integrated LINQ Expressions 34](#_Toc423016463)

[8.8.11 Caller Information 34](#_Toc423016464)

[8.9 Interop 34](#_Toc423016465)

[8.9.1 Regarding P/Invoke 34](#_Toc423016466)

[8.9.2 COM Interop 36](#_Toc423016467)

[9 Native C++ Guidelines 38](#_Toc423016468)

[9.1 Header Files 38](#_Toc423016469)

[9.1.1 Layout 38](#_Toc423016470)

[9.1.2 General 38](#_Toc423016471)

[9.2 General Guidelines 38](#_Toc423016472)

[9.3 Comments 38](#_Toc423016473)

[9.3.1 Class 39](#_Toc423016474)

[9.3.2 Function 39](#_Toc423016475)

[9.3.3 Member Variable 39](#_Toc423016476)

[9.4 Warnings 39](#_Toc423016477)

[9.5 Using Pointers 39](#_Toc423016478)

[9.6 Constants 40](#_Toc423016479)

[9.7 Casting 40](#_Toc423016480)

[9.8 Sizeof 40](#_Toc423016481)

[9.9 Hungarian Notation 40](#_Toc423016482)

[9.9.1 Type Tags 40](#_Toc423016483)

[9.9.2 Prefixes 40](#_Toc423016484)

[9.10 Unicode/TCHAR 41](#_Toc423016485)

[9.11 Array and Struct Init 41](#_Toc423016486)

[9.12 Macros 41](#_Toc423016487)

[9.13 Parameters 41](#_Toc423016488)

[9.14 Struct vs Class 41](#_Toc423016489)

[9.15 Classes 41](#_Toc423016490)

[9.15.1 Constructors 42](#_Toc423016491)

[9.15.2 Operator Overloading 42](#_Toc423016492)

[9.15.3 Destructors 42](#_Toc423016493)

[9.15.4 Errors and Exceptions 42](#_Toc423016494)

[9.16 Resource Handling 42](#_Toc423016495)

[9.17 Return from Function 42](#_Toc423016496)

[10 References 43](#_Toc423016497)

[11 Works Cited 43](#_Toc423016498)

# Purpose

The purpose of this document is to define a set of coding guidelines for C# and native C++.

# Objectives

The higher level aim is to define a set of coding guidelines for bringing uniformity across projects and to automate the process of ensuring that these guidelines are followed as far as possible.

Here are the objectives of the CQM project related to Codestriker and Sonarqube:

1. Defining the coding standards and guidelines which developers need to use (for Java, C#, C++)
2. Configuring them in the tools (Codestriker/Sonarqube) (the CQM developer team will take care of implementation)
3. Integrate the output of Sonarqube into Codestriker so that the points flagged by Sonarqube appear as review comments in Codestriker (the CQM developer team will take care of implementation)

# Accompanying Document

In the following sections are references into this document: All-In-One Code Framework Coding Standards.docx (Microsoft AIO). The references start with “Refer (Microsoft AIO) Section xxx” followed by the text pasted from that document.

Some VB.NET code might have come in in these references from the above document and for such cases, the reader is expected to focus on & understand the underlying concept that’s applicable to C# as well.

# Assumptions

* As far as C# is concerned, version 5.0 of the language is assumed (Side note: C# versions are not necessarily 100% backward compatible. Projects in which backward compatibility is a consideration are urged to carry out their analysis separately)
* So far, the C++ guidelines in this document are written mostly from a Microsoft/Windows perspective

# Casing

Here is a short description of the various commonly used naming conventions:

* **Pascal casing** (The first letter in the name and the first letter of each subsequent concatenated word should be capital).

Example: PascalCasing

* **Camel casing** (The first letter in the name is lower case and the first letter of each subsequent concatenated word should be capital).

Example: camelCasing

* **Upper casing** (All letters in the identifier are upper case, use of underscore is ok).

Example: UPPER\_CASING

Example: System.IO, System.Web.UI

# Guidelines Common Across Languages

Even though the programming languages maybe different, there would be common factors between them (e.g. naming of member variables), which are enlisted in this section. In other words the guidelines in this section would apply to all the programming languages covered under this document, viz. C# and C++ at this point.

The guidelines mentioned in this section would override any duplications that you might come across in the following sections.

## File Header

The following header to be inserted at the top of every source code file written by programmers (as against auto-generated files):

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

// Summary: Brief summary of the file contents and purpose.

//------------------------------------------------------------------

// Copyright and Legal notices.

//\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Naming Conventions

### General, non-UI Code

The following table is taken from [this URL](https://msdn.microsoft.com/en-us/library/ms229002%28v=vs.110%29.aspx) which is actually for C# (the topic named “Capitalization Conventions”), however the guidelines apply to C# as well as C++. Modifications have been applied to the following:

* Parameter
* Field (which are Member Variables)
* Local variables

|  |  |  |
| --- | --- | --- |
| Identifier | Casing | Example (C# unless otherwise noted) |
| Namespace | Pascal | namespace System.Security { ... } |
| Type | Pascal | public class StreamReader { ... } |
| Interface | Pascal | public interface IEnumerable { ... } |
| Method | Pascal | public class Object {   public virtual string ToString(); } |
| Property | Pascal | public class String {   public int Length { get; } } |
| Event | Pascal | public class Process {   public event EventHandler Exited; } |
| Field | Pascal | public class MessageQueue {   public static readonly TimeSpan \_infiniteTimeout; }  public struct UInt32 {  public const \_minimum = 0; } |
| Enumeration | Pascal | public enum Colors {  } |
| Enum value | C# : Pascal  C++: Upper | public enum FileMode {   Append, ... } |
| Parameter and Local Variable | Camel | public class Convert {   public static int ToInt32(string value); } |
| Structure (C++) | Upper, separate words with ‘\_’ | C++:  struct UPPER\_CASE |
| Macro, constant (C++) | Upper, separate words with ‘\_’ | C++:  #define MY\_FILE\_NAME … |
| Template Parameter (C++) | Pascal, ‘T’ prefix | C++:  TPolicy, TShoppingItem |

*Note that the “Example” column in the above table mostly applies to C# unless otherwise noted. Use simple discretion when applying to other than C# languages. Some of the things appearing under the “Identifier” column may not apply to all the programming languages in scope.*

### UI Code

UI control names should be prefixed as follows:

|  |  |
| --- | --- |
| Control Type | Prefix |
| Button | btn |
| CheckBox | chk |
| CheckedListBox | lst |
| ComboBox | cbx |
| ContextMenu | mnu |
| DataGrid | dg |
| DateTimePicker | dtp |
| Form | Form (this is a suffix) |
| GroupBox | grp |
| ImageList | imgl |
| Label | lb |
| ListBox | lst |
| ListView | lvw |
| Menu | mnu |
| MenuItem | mitm |
| NotificationIcon | nfy |
| Panel | pnl |
| PictureBox | pbx |
| ProgressBar | pbar |
| RadioButton | rbtn |
| Splitter | spl |
| StatusBar | sts |
| TabControl | tctrl |
| TabPage | tpg |
| TextBox | tbx |
| Timer | tmr |
| TreeView | tvw |

## Referencing

Make sure to not refer unnecessary libraries/assemblies and (C++) do not #include unnecessary files. Typically this happens after a code refactoring exercise where the previous code required those references/#includes and they just get left behind.

## Whitespace/Layout

1. Make appropriate use of spacing/whitespace so as to have a proper level of legibility and avoid crowding of code. E.g. in places where a lot of local variables may have been declared, group them logically and use an empty line of code to separate them rather than clubbing them all together without any separating lines.

Similarly, use empty lines to separate out logical groups of code.

1. Curly braces should be at the same level as the code outside the braces. This is called Allman or ANSI style. Even if the if or the else part has only one line of code inside it, surround it by braces. E.g.

// outer scope code

if ( a == b )

{

// inner scope code

…

return true;

}

## General Guidelines

### Variable Declaration

Declare variables at the minimum scope level where they are required to be used. Always initialize variables at declaration time. That way you are sure what state (initialize/uninitialized) the variable is at any point in the code.

Declare only one variable on a line.

**Local Variables and Arguments:** Do not use underscores (\_) in local variable names and method arguments. Local variable and argument names should start with a lower case letter and use camelCasing. E.g. int countOfInvoices

**Member Variables:** All member variables must be prefixed with underscore (\_) and the first letter should be lowercase so that they can be differentiated from local variables, arguments, etc. by simply looking at them.

e.g public class Employee

{

private string \_firstName;

}

For variable names, use sufficiently meaningful & descriptive words. Avoid using abbreviations.

**Recommended:**

string machineAddress

int firstName

**Not Recommended:**

string nam

string addrs

Do not use single character variable names such as i, j, n, etc. Use meaningful names such as “index”.

Do not use language keywords as names in your code. Don't rely on case sensitivity to avoid conflicts and get your code to work.

e.g int switch;

### Line Length

Code lines should not be more than 80 to 90 characters in length (columns). One of the reasons is that the lines then scroll off screen and it’s cumbersome to view them in typical code comparison (diff) tools.

### Function Declaration

The return type, function name and parameter list should be on the same line. In cases where there are too many parameters, use the following format, i.e. the return type and the function name should be on the same line. Group the IN parameters first and the OUT parameters last.

int MyFunction(

int paramOne, // short description

string firstname, // short description

string lastname, // short description

int paramThree, // short description

double dubFour) // short description

{

…

}

### Statements

There should be one statement per line. If there are more than one, stepping through code using a debugger becomes difficult.

### Enums

Use enums for sets of values. Favor them over static constants or “#define” values as enums get added support from the compiler and from reflection. Enums are strongly typed.

In declarations of enums, provide a base value of 0 and call it None.

e.g.

C++:

enum Colors

{

None = 0,

Red,

Green,

Yellow

};

C#:

public enum Colors

{

None = 0,

Red,

Green,

Yellow

};

## Comments

Write enough comments so that what the code is meant to do is described properly.

Do not use /\* \*/ comments except for in function comment headers. Aim on using // .

Use proper punctuation for the comments, as in start with a caps letter and end in a period (.), etc.

Indent comments at the same level as the code which they describe.

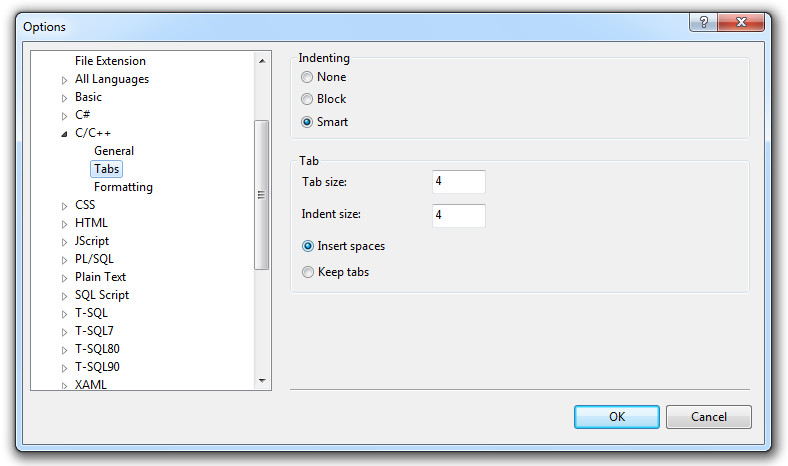
## Goto

Do not use goto.

## Indentation

Do not use Tabs for indentation. Use 4 spaces for 1 indentation. The reason being different text editors use different spacing to render tabs, which causes formatting confusion.

Configure tabs using the following settings:



Visual Studio screen for configuring tabs for C++ using spaces

e.g.

if (1 == count)

{

**----**// 4 spaces to the left here equal 1 indentation

}

# C# Guidelines

This section lays out the guidelines to be followed by .NET projects that use C# as the programming language. (Do not miss out on the points mentioned in the section addressing guidelines that are common across languages. They apply to C# as well.)

## Naming Conventions

For details of the naming convention to be used for various elements such as classes, interfaces, methods, etc., please refer here (there are several sections under this URL):

<https://msdn.microsoft.com/en-us/library/ms229002%28v=vs.110%29.aspx>

Note that the guidelines in the section “Guidelines Common Across Languages” override any that may appear in the above URL.

## Classes

### Layout

1. Use #region to group logical lines of code together. Here’s how the regions would look like when collapsed, for example:

+ Private Member Variables and Constants

+ Public Constants

+ Public Readonly Static Fields

+ Private Properties

+ Private Methods

+ Constructors

+ Events

+ Public Properties

+ Public Methods

### Member Design

Here are detailed guidelines for member (property, constructor, event, field, extension methods, operator overloads and parameter design) design:

<https://msdn.microsoft.com/en-us/library/ms229059%28v=vs.110%29.aspx>

#### Methods

Refer (Microsoft AIO) section 4.9.4

🗹 **Do** place all out parameters after all of the pass-by-value and ref parameters (excluding parameter arrays), even if this results in an inconsistency in parameter ordering between overloads.

🗹 **Do** validate arguments passed to public, protected, or explicitly implemented members. Throw System.ArgumentException, or one of its subclasses, if the validation fails: If a null argument is passed and the member does not support null arguments, throw ArgumentNullException. If the value of an argument is outside the allowable range of values as defined by the invoked method, throw ArgumentOutOfRangeException.

#### Events

Refer (Microsoft AIO) section 4.9.5

🗹 **Do** be prepared for arbitrary code executing in the event-handling method. Consider placing the code where the event is raised in a try-catch block to prevent program termination due to unhandled exceptions thrown from the event handlers.

🗷 **Do not** use events in performance sensitive APIs. While events are easier for many developers to understand and use, they are less desirable than Virtual Members from a performance and memory consumption perspective.

#### Member Overloading

Refer (Microsoft AIO) section 4.9.6

🗹 **Do** use member overloading rather than defining members with default arguments. Default arguments are not CLS-compliant and cannot be used from some languages. There is also a versioning issue in members with default arguments. Imagine version 1 of a method that sets an optional parameter to 123. When compiling code that calls this method without specifying the optional parameter, the compiler will embed the default value (123) into the code at the call site. Now, if version 2 of the method changes the optional parameter to 863, then, if the calling code is not recompiled, it will call version 2 of the method passing in 123 (version 1’s default, not version 2’s default).

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

🗷 **Do not** arbitrarily vary parameter names in overloads. If a parameter in one overload represents the same input as a parameter in another overload, the parameters should have the same name. Parameters with the same name should appear in the same position in all overloads.

🗹 **Do** make only the longest overload virtual (if extensibility is required). Shorter overloads should simply call through to a longer overload.

#### Virtual Members

Refer (Microsoft AIO) section 4.9.8

Virtual members perform better than callbacks and events, but do not perform better than non-virtual methods.

🗷 **Do not** make members virtual unless you have a good reason to do so and you are aware of all the costs related to designing, testing, and maintaining virtual members.

🗹 **You should** prefer protected accessibility over public accessibility for virtual members. Public members should provide extensibility (if required) by calling into a protected virtual member.

#### Static Members

If there are any predefined object instances, make them public static readonly members.

#### Auto-implemented Properties

Leverage the power added to the C# compiler such as *auto-implemented properties*, which helps in reducing the amount of code you write. Details [here](http://weblogs.asp.net/scottgu/new-c-orcas-language-features-automatic-properties-object-initializers-and-collection-initializers).

e.g.

public class Person {  
  
        private string \_firstName;

        public string FirstName {  
  
            get {  
                return \_firstName;  
            }  
            set {  
                \_firstName = value;  
            }  
        }

}

Can be written more concisely as follows, the compiler adds the required private declaration, etc.:

    public class Person {  
        public string FirstName { get; set; }

    }

#### Number of Arguments

Avoid passing more than 4-5 arguments. If more are required to be passed consider using a class for it.

### Static Classes

Refer (Microsoft AIO) section 4.9.9

🗹 **Do** use static classes sparingly. Static classes should be used only as supporting classes for the object-oriented core of the framework.

### Abstract Classes

Refer (Microsoft AIO) section 4.9.10.

🗷 **Do not** define public or protected-internal constructors in abstract types.

🗹 **Do** define a protected or an internal constructor on abstract classes.

A protected constructor is more common and simply allows the base class to do its own initialization when subtypes are created.

public abstract class Claim

{

protected Claim()

{

...

}

}

An internal constructor can be used to limit concrete implementations of the abstract class to the assembly defining the class.

public abstract class Claim

{

internal Claim()

{

...

}

}

## Types

### Strings

Refer (Microsoft AIO) section 4.6

🗷 **Do not** use the ‘+’ operator (or ‘&’ in VB.NET) to concatenate many strings. Instead, you should use StringBuilder for concatenation. However, **do** use the ‘+’ operator (or ‘&’ in VB.NET) to concatenate small numbers of strings.

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();

}

🗹 **Do** use overloads that explicitly specify the string comparison rules for string operations. Typically, this involves calling a method overload that has a parameter of type [StringComparison](http://msdn.microsoft.com/en-us/library/system.stringcomparison.aspx).

🗹 **Do** use [StringComparison.Ordinal](http://msdn.microsoft.com/en-us/library/system.stringcomparison.ordinal.aspx) or [StringComparison.OrdinalIgnoreCase](http://msdn.microsoft.com/en-us/library/system.stringcomparison.ordinalignorecase.aspx) for comparisons as your safe default for culture-agnostic string matching, and for better performance.

🗹 **Do** use string operations that are based on [StringComparison.CurrentCulture](http://msdn.microsoft.com/en-us/library/system.stringcomparison.currentculture.aspx) when you display output to the user.

🗹 **Do** use the non-linguistic [StringComparison.Ordinal](http://msdn.microsoft.com/en-us/library/system.stringcomparison.ordinal.aspx) or [StringComparison.OrdinalIgnoreCase](http://msdn.microsoft.com/en-us/library/system.stringcomparison.ordinalignorecase.aspx) values instead of string operations based on [CultureInfo.InvariantCulture](http://msdn.microsoft.com/en-us/library/system.globalization.cultureinfo.invariantculture.aspx) when the comparison is linguistically irrelevant (symbolic, for example). Do not use string operations based on StringComparison.InvariantCulture in most cases. One of the few exceptions is when you are persisting linguistically meaningful but culturally agnostic data.

🗹 **Do** use an overload of the [String.Equals](http://msdn.microsoft.com/en-us/library/system.string.equals.aspx) method to test whether two strings are equal. For example, to test if two strings are equal ignoring the case,

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

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

🗷 **Do not** use an overload of the String.[Compare](http://msdn.microsoft.com/en-us/library/system.string.compare.aspx) or [CompareTo](http://msdn.microsoft.com/en-us/library/system.string.compareto.aspx) method and test for a return value of zero to determine whether two strings are equal. They are used to sort strings, not to check for equality.

🗹 **Do** use the [String.ToUpperInvariant](http://msdn.microsoft.com/en-us/library/system.string.toupperinvariant.aspx) method instead of the [String.ToLowerInvariant](http://msdn.microsoft.com/en-us/library/system.string.tolowerinvariant.aspx) method when you normalize strings for comparison.

### Arrays and Collections

Refer (Microsoft AIO) Section 4.7.

🗹 **You should** use arrays in low-level functions to minimize memory consumption and maximize performance. In public interfaces, do prefer collections over arrays.

Collections provide more control over contents, can evolve over time, and are more usable. In addition, using arrays for read-only scenarios is discouraged as the cost of cloning the array is prohibitive.

However, if you are targeting more skilled developers and usability is less of a concern, it might be better to use arrays for read-write scenarios. Arrays have a smaller memory footprint, which helps reduce the working set, and access to elements in an array is faster as it is optimized by the runtime.

🗷 **Do not** use read-only array fields. The field itself is read-only and can’t be changed, but elements in the array can be changed. This example demonstrates the pitfalls of using read-only array fields:

Bad:

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

This allows callers to change the values in the array as follows:

InvalidPathChars[0] = 'A';

Instead, you can use either a read-only collection (only if the items are immutable) or clone the array before returning it. However, the cost of cloning the array may be prohibitive.

public static ReadOnlyCollection<char> GetInvalidPathChars()

{

return Array.AsReadOnly(badChars);

}

public static char[] GetInvalidPathChars()

{

return (char[])badChars.Clone();

}

🗹 **You should** use jagged arrays instead of multidimensional arrays. A jagged array is an array with elements that are also arrays. The arrays that make up the elements can be of different sizes, leading to less wasted space for some sets of data (e.g., sparse matrix), as compared to multidimensional arrays. Furthermore, the CLR optimizes index operations on jagged arrays, so they might exhibit better runtime performance in some scenarios.

// Jagged arrays

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} \_

}

// Multidimensional arrays

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} \_

}

🗹 **Do** use Collection<T> or a subclass of Collection<T> for properties or return values representing read/write collections, and use ReadOnlyCollection<T> or a subclass of ReadOnlyCollection<T> for properties or return values representing read-only collections.

🗹 **You should** reconsider the use of ArrayList because any objects added into the ArrayList are added as System.Object and when retrieving values back from the arraylist, these objects are to be unboxed to return the actual value type. So it is recommended to use the custom typed collections instead of ArrayList. For example, .NET provides a strongly typed collection class for String in System.Collection.Specialized, namely StringCollection.

🗹 **You should** reconsider the use of Hashtable. Instead, try other dictionary such as StringDictionary, NameValueCollection, HybridCollection. Hashtable can be used if less number of values is stored.

🗹 When you are creating a collection type, **you should** implement IEnumerable so that the collection can be used with LINQ to Objects.

🗷 **Do not** implement both IEnumerator<T> and IEnumerable<T> on the same type. The same applies to the nongeneric interfaces IEnumerator and IEnumerable. In other words, a type should be either a collection or an enumerator, but not both.

🗷 **Do not** return a null reference for Array or Collection. Null can be difficult to understand in this context. For example, a user might assume that the following code will work. Return an empty array or collection instead of a null reference.

int[] arr = SomeOtherFunc();

foreach (int v in arr)

{

...

}

### Structures vs Classes

Refer (Microsoft AIO) Section 4.8.1.

🗷 **Do not** define a struct unless the type has all of the following characteristics:

* It logically represents a single value, similar to primitive types (int, double, etc.).
* It has an instance size fewer than 16 bytes.
* It is immutable.
* It will not have to be boxed frequently.

In all other cases, you should define your types as classes instead of structs.

### Implicitly Typed Local Variables (var)

Local variables can be declared without explicitly declaring their types. The compiler infers the type from the right hand side expression. Such variables are strongly typed. Old timer programmers could argue that legibility suffers, that they would rather know the exact type being used explicitly. [MSDN URL](https://msdn.microsoft.com/en-us/library/bb384061.aspx).

e.g.

// in is compiled as an int

var in = 5;

// str is compiled as a string

var str = "Hello";

### Anonymous Types (var)

Make use of these [types](https://msdn.microsoft.com/en-IN/library/bb397696.aspx) where you want to define readonly properties in an object without explicitly defining the type.

e.g.

select new {name = cust.Name, phone = cust.Phone};

## Secure Coding

Please refer [here](https://msdn.microsoft.com/en-us/library/8a3x2b7f(v=vs.140).aspx) for secure coding guidelines.

## Documentation Comments

Tag Reference: <https://msdn.microsoft.com/en-us/library/5ast78ax.aspx>

Visual Studio has built in support for adding comments in your code which work with Intellisense. For e.g. on the line before a method starts if one starts typing “///”, Visual Studio inserts the required XML snippet inside which the programmer can add comments for things such as method description, parameter description, value returned, etc. Make use of this feature to add comments before the following:

* Classes
* Interfaces
* Methods
* Properties
* Member Variables

Here is a documentation sample: <https://msdn.microsoft.com/en-us/library/z04awywx.aspx>

Do not use the /\* \*/ comments.

## General Guidelines

1. Prefix boolean methods, variables & properties with “is” or similar prefixes.

Ex: private bool isCompleted

1. Enums: Do not use Enum.IsDefined for range checking as its an expensive operation.

Don’t use this:

if (Enum.IsDefined(typeof(Color), c))

{

…

}

Instead use something on these lines:

if (c > Color.None && c <= Color.Blue)

{

…

}

Use the Flag Attribute on enums when support for bitwise operations is desired. E.g.

[Flags]

public enum MyEnum

{

First=0x0001,

Second=0x0002,

Third=0x0004

};

## Constants

Use const fields for values that will never change.

e.g.

public class ConstantInts

{

public const int \_maxValue = 0x7fffffff;

}

## Programming Topics

### Exceptions

#### Exception Throwing

Refer (Microsoft AIO) section 4.11.1.

🗹 **Do** report execution failures by throwing exceptions. Exceptions are the primary means of reporting errors in frameworks. If a member cannot successfully do what it is designed to do, it should be considered an execution failure and an exception should be thrown. **Do not** return error codes.

🗹 **Do** throw the most specific (the most derived) exception that makes sense. For example, throw ArgumentNullException and not its base type ArgumentException if a null argument is passed. Throwing System.Exception as well as catching System.Exception are nearly always the wrong thing to do.

🗷 **Do not** use exceptions for the normal flow of control, if possible. Except for system failures and operations with potential race conditions, you should write code that does not throw exceptions. For example, you can check preconditions before calling a method that may fail and throw exceptions. For example,

// C# sample:

if (collection != null && !collection.IsReadOnly)

{

collection.Add(additionalNumber);

}

' VB.NET sample:

If ((Not collection Is Nothing) And (Not collection.IsReadOnly)) Then

collection.Add(additionalNumber)

End If

🗷 **Do not** throw exceptions from exception filter blocks. When an exception filter raises an exception, the exception is caught by the CLR, and the filter returns false. This behavior is indistinguishable from the filter executing and returning false explicitly and is therefore very difficult to debug.

' 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

🗷 **Do not** explicitly throw exceptions from finally blocks. Implicitly thrown exceptions resulting from calling methods that throw are acceptable.

#### Exception Handling

Do use exceptions as the mechanism for error reporting. Do not suppress exceptions by writing a catch all (i.e. catch (Exception ex)) handler that does nothing. If exceptions are suppressed the programmer will never come to know if any exceptions occurred and miss trying to analyze the reasons for their occurrence. Let the application crash if it has to. That way, chances are you will come to know of bugs in the development cycle itself.

If there is an absolute need to write a catch all handler log all the details about that exception in the handler before propagating that exception (i.e. doing a “throw”, which preserves the call stack). Prefer specific exception handlers rather than catch all handlers. Writing a highest level application handler that logs exception details also is fine.

#### Try-finally Blocks

Refer (Microsoft AIO) section 4.12.1

🗹 **Do** use try-finally blocks for cleanup code and try-catch blocks for error recovery code. **Do not** use catch blocks for cleanup code. Usually, the cleanup logic rolls back resource (particularly, native resource) allocations. For example,

// 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

C# and VB.NET provide the using statement that can be used instead of plain try-finally to clean up objects implementing the IDisposable interface.

// C# sample:

using (FileStream stream = new FileStream(...))

{

...

}

' VB.NET sample:

Using stream As New FileStream(...)

...

End Using

Many language constructs emit try-finally blocks automatically for you. Examples are C#/VB’s using statement, C#’s lock statement, VB’s SyncLock statement, C#’s foreach statement, and VB’s For Each statement.

### Dispose Pattern

Refer (Microsoft AIO) section 4.12.2

The basic implementation of the pattern involves implementing the System.IDisposable interface and declaring the Dispose(bool) method that implements all resource cleanup logic to be shared between the Dispose method and the optional finalizer. Please note that this section does not discuss providing a finalizer. Finalizable types are extensions to this basic pattern and are discussed in the next section. The following example shows a simple implementation of the basic pattern:

// 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

End If

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

🗹 **Do** implement the Basic Dispose Pattern on types containing instances of disposable types.

🗹 **Do** extend the Basic Dispose Pattern to provide a finalizer on types holding resources that need to be freed explicitly and that do not have finalizers. For example, the pattern should be implemented on types storing unmanaged memory buffers.

🗹 **You should** implement the Basic Dispose Pattern on classes that themselves don’t hold unmanaged resources or disposable objects but are likely to have subtypes that do. A great example of this is the System.IO.Stream class. Although it is an abstract base class that doesn’t hold any resources, most of its subclasses do and because of this, it implements this pattern.

🗹 **Do** declare a protected virtual void Dispose(bool disposing) method to centralize all logic related to releasing unmanaged resources. All resource cleanup should occur in this method. The method is called from both the finalizer and the IDisposable.Dispose method. The parameter will be false if being invoked from inside a finalizer. It should be used to ensure any code running during finalization is not accessing other finalizable objects. Details of implementing finalizers are described in the next section.

// C# sample:

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:

Protected Overridable Sub Dispose(ByVal disposing As Boolean)

' Protect from being called multiple times.

If disposed Then

Return

End If

If disposing Then

' Clean up all managed resources.

If (resource IsNot Nothing) Then

resource.Dispose()

End If

End If

disposed = True

End Sub

🗹 **Do** implement the IDisposable interface by simply calling Dispose(true) followed by GC.SuppressFinalize(this). The call to SuppressFinalize should only occur if Dispose(true) executes successfully.

// 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

🗷 **Do not** make the parameterless Dispose method virtual. The Dispose(bool) method is the one that should be overridden by subclasses.

🗷 Y**ou should not** throw an exception from within Dispose(bool) except under critical situations where the containing process has been corrupted (leaks, inconsistent shared state, etc.). Users expect that a call to Dispose would not raise an exception. For example, consider the manual try-finally in this C# snippet:

TextReader tr = new StreamReader(File.OpenRead("foo.txt"));

try

{

// Do some stuff

}

finally

{

tr.Dispose();

// More stuff

}

If Dispose could raise an exception, further finally block cleanup logic will not execute. To work around this, the user would need to wrap every call to Dispose (within their finally block!) in a try block, which leads to very complex cleanup handlers. If executing a Dispose(bool disposing) method, never throw an exception if disposing is false. Doing so will terminate the process if executing inside a finalizer context.

🗹 **Do** throw an ObjectDisposedException from any member that cannot be used after the object has been disposed.

// 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

Protected Overridable Sub Dispose(ByVal disposing As Boolean)

' Protect from being called multiple times.

If disposed Then

Return

End If

' Cleanup

...

disposed = True

End Sub

End Class

### Finalizable Types

Refer (Microsoft AIO) section 4.12.3

Finalizable types are types that extend the Basic Dispose Pattern by overriding the finalizer and providing finalization code path in the Dispose(bool) method. The following code shows an example of a finalizable type:

// 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

End If

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

🗹 **Do** make a type finalizable, if the type is responsible for releasing an unmanaged resource that does not have its own finalizer. When implementing the finalizer, simply call Dispose(false) and place all resource cleanup logic inside the Dispose(bool disposing) method.

// 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

🗹 **Do** be very careful to make type finalizable. Carefully consider any case in which you think a finalizer is needed. There is a real cost associated with instances with finalizers, from both a performance and code complexity standpoint.

🗹 **Do** implement the Basic Dispose Pattern on every finalizable type. See the previous section for details on the basic pattern. This gives users of the type a means to explicitly perform deterministic cleanup of those same resources for which the finalizer is responsible.

🗹 **You should** create and use a critical finalizable object (a type with a type hierarchy that contains CriticalFinalizerObject) for situations in which a finalizer absolutely must execute even in the face of forced application domain unloads and thread aborts.

🗹 **Do** prefer resource wrappers based on SafeHandle or SafeHandleZeroOrMinusOneIsInvalid (for Win32 resource handle whose value of either 0 or -1 indicates an invalid handle) to writing finalizer by yourself to encapsulate unmanaged resources where possible, in which case a finalizer becomes unnecessary because the wrapper is responsible for its own resource cleanup. Safe handles implement the IDisposable interface, and inherit from CriticalFinalizerObject so the finalizer logic will absolutely execute even in the face of forced application domain unloads and thread aborts.

/// <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);

}

}

🗷 **Do not** access any finalizable objects in the finalizer code path, as there is significant risk that they will have already been finalized. For example, a finalizable object A that has a reference to another finalizable object B cannot reliably use B in A’s finalizer, or vice versa. Finalizers are called in a random order (short of a weak ordering guarantee for critical finalization).

It is OK to touch unboxed value type fields.

Also, be aware that objects stored in static variables will get collected at certain points during an application domain unload or while exiting the process. Accessing a static variable that refers to a finalizable object (or calling a static method that might use values stored in static variables) might not be safe if Environment.HasShutdownStarted returns true.

🗷 **Do not** let exceptions escape from the finalizer logic, except for system-critical failures. If an exception is thrown from a finalizer, the CLR may shut down the entire process preventing other finalizers from executing and resources from being released in a controlled manner.

### Overriding Dispose

Refer (Microsoft AIO) section 4.12.4

If you're inheriting from a base class that implements IDisposable, you must implement IDisposable also. Always call your base class's Dispose(bool) so it cleans up.

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);

}

}

}

### Size of Artifacts

Avoid having methods or files that are too long. If it seems to start getting too long its time to refactor that code into smaller pieces.

### Hardcoded Values

Declare constants for hardcoded values and use the constants in your code. Consider hosting such values in the app config file or a database. Then those values can be changed without any code changes.

### Object & Collection Initializers

Make use of [these initializers](http://weblogs.asp.net/scottgu/new-c-orcas-language-features-automatic-properties-object-initializers-and-collection-initializers) to make your code look elegant.

e.g.

Object initializer:

   Person person = new Person();  
   person.FirstName = "First";  
   person.LastName = "Second";  
   person.Age = 32;

There is now support to write that as:

Person person = new Person { FirstName="First", LastName="Second", Age=44 };

e.g.

Collection initializer:

  List<Person> people = new List<Person>();  
              
  people.Add( new Person { FirstName = "Anna", LastName = "Scott", Age = 32 } );  
  people.Add( new Person { FirstName = "Bill", LastName = "Gates", Age = 50 } );

There is now support to write that as:

 List<Person> people = new List<Person> {  
      new Person { FirstName = "Anna", LastName = "Scott", Age = 32 },  
      new Person { FirstName = "Bill", LastName = "Gates", Age = 50 },   };

### Lambda Expressions

Use these expressions for programming convenience and better legibility.

e.g.

In C# 2.0:

listOfObjs.Where(delegate(Obj a) { return a.Size > 16; });

which in C# 3.0 can be written as (using lambda expression):

listOfObjs.Where(a => a.Size > 16);

### Reflection

It’s alright to use reflection. Just be aware of its performance implications, be aware of the kind of code you write that results in reflection being used. MSDN [URL on reflection](https://msdn.microsoft.com/en-us/library/ms173183.aspx).

### Language Integrated LINQ Expressions

Query expressions provide the language integrated experience of [LINQ](https://msdn.microsoft.com/en-us/library/bb397926.aspx). Use them to facilitate programming convenience using an SQL like syntax. Internally the compiler translates the expression into method invocations. Beware of the performance implications of LINQ providers (e.g. LINQ to SharePoint). Conducting due diligence related to the pros and cons of using LINQ is always advisable.

### Caller Information

If writing code that can benefit from diagnostic information such as caller information, leverage the language feature mentioned [here](https://msdn.microsoft.com/en-us/library/hh156499%28v=vs.110%29.aspx).

## Interop

### Regarding P/Invoke

Refer (Microsoft AIO) section 4.13.1

🗹 **Do** consult [P/Invoke Interop Assistant](http://clrinterop.codeplex.com/) and <http://pinvoke.net> to write P/Invoke signatures.

🗹 **You can** use IntPtr for manual marshaling. By declaring parameters and fields as IntPtr, you can boost performance, albeit at the expense of ease of use, type safety, and maintainability. Sometimes it is faster to perform manual marshaling by using methods available on the Marshal class rather than to rely on default interop marshaling. For example, if large arrays of strings need to be passed across an interop boundary, but the managed code needs only a few of those elements, you can declare the array as IntPtr and manually access only those few elements that are required.

🗷 **Do not** aggressively pin short-lived objects. Pinning short-lived objects unnecessarily extends the life of a memory buffer beyond the duration of the P/Invoke call. Pinning prevents the garbage collector from relocating the bytes of the object in the managed heap, or relocating the address of a managed delegate. However, it is acceptable to pin long-lived objects, which are ideally created during application initialization, because they are not moved relative to short-lived objects. It is costly to pin short-lived objects for a long period of time, because compacting occurs most in Generation 0 and the garbage collector cannot relocate pinned objects. This results in inefficient memory management that can adversely affect performance. For more information about copying and pinning, see <http://msdn.microsoft.com/en-us/library/23acw07k.aspx>.

🗹 **Do** set CharSet = CharSet.Auto and SetLastError = true in the P/Invoke signature. For example,

// 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

🗹 **You should** wrap unmanaged resources in SafeHandle classes. The SafeHandle class is discussed in the [Finalizable Types](#_Finalizable_Types) section. For example, the handle of file mapping is wrapped as follows.

/// <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 Class

🗹 **You should** throw Win32Exception on the failure of P/Invoked functions that set the Win32 last error. If the function uses some unmanaged resources, free the resource in the finally block.

// 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

...

Finally

If (Not hMapFile Is Nothing) Then

' Close the file mapping object.

hMapFile.Close()

hMapFile = Nothing

End If

End Try

### COM Interop

Refer (Microsoft AIO) section 4.13.2

🗷 **Do not** force garbage collections with GC.Collect to release COM objects in performance sensitive APIs. A common approach for releasing COM objects is to set the RCW reference to null, and call System.GC.Collect followed by System.GC.WaitForPendingFinalizers. This is not recommended for performance reasons, because in many situations it can trigger the garbage collector to run too often. Code written by using this approach significantly compromises the performance and scalability of server applications. You should let the garbage collector determine the appropriate time to perform a collection.

🗹 **You should** use Marshal.FinalReleaseComObject or Marshal.ReleaseComObject to manage the lifetime of an RCW manually. It has much better performance than forcing garbage collections with GC.Collect.

🗷 **Do not** make cross-apartment calls. When you call a COM object from a managed application, make sure that the managed code's apartment matches the COM object's apartment type. By using matching apartments, you avoid the thread switch associated with cross-apartment calls.

# Native C++ Guidelines

## Header Files

### Layout

In every header file, have guards in place for preventing multiple #inclusion of the file. “#pragma once” which is MS specific can also be used for the same purpose. This should be the general layout of the file:

#ifndef INVOICES\_H

#define INVOICES\_H

// copyright header comment block

// #includes

// #defines

// class/function/interface declarations

// inline functions

#endif // define INVOICES\_H

### General

Methods/functions should not be defined in header files unless they are 1 line (get/set type of methods) in length.

Do not put using directives in header files.

Do not define more than one class in a given header file. Exceptions to this are ok if justifiable.

Files containing a class should have the same name as that of the class.

## General Guidelines

Inline functions: Consider making functions inline in case they are smaller than 3 lines of code. Details [here](https://msdn.microsoft.com/en-us/library/cx3b23a3.aspx).

Consider if you want to write code that supports both 32 bit as well as 64 bit.

Global Variables: Pass global variables and static variables to functions rather than directly modifying them in functions. If that’s not done the caller doesn’t know if the value was modified.

## Comments

### Class

Before the definition of a class in a header file, the following comment block should exist:

//

// Name: class <Class name>

// Summary: <class description>

//

### Function

/\*---------------------------------------------------------------------------

Function: <prototype>

Purpose: <description>

Arguments:

<name>: <description>

Returns:

<return type/description if any>

Exceptions:

<Exceptions that may be thrown by the function>

---------------------------------------------------------------------------\*/

### Member Variable

Make sure to have a short description against every member variable. E.g.

class A

{

private int \_iTotalLeads; // used to hold the total number of leads

};

## Warnings

Compile code at the highest warning level. The code should compile with 0 warnings. Subtle, sometimes hard to find bugs can be captured by doing this and also bad practices can be identified.

## Using Pointers

Always initialize pointers upon declaration and if the value (e.g. memory) that has been allocated into it is undone (freed in case of memory), reset the pointer to NULL.

Put a space between the type and the ‘\*’ when declaring a pointer. There should not be any space between the ‘\*’ and the pointer variable name.

E.g.

Good

int \*pCountofHouses = NULL;

Bad

int\* pCountofHouses = NULL;

## Constants

Use const values instead of #defines

## Casting

Make appropriate use of the C++ cast expressions static\_cast, reinterpret\_cast and const\_cast (comes with its caveats) rather than direct casting such as

A \*pObjectA = (A \*)pObjectB;

## Sizeof

Use sizeof(variable) and not sizeof(type) when possible.

## Hungarian Notation

This is a legacy style, here are the guidelines in case you decide to use it:

### Type Tags

|  |  |  |
| --- | --- | --- |
| Type | Tag | Description |
| BOOL, bool, bitfield | f | A flag. For example, BOOL fSucceeded; |
| DWORD | dw | Double word, an unsigned 32 bit quantity. |
| HRESULT | hr | HRESULT values are commonly used throughout Win32/COM for error/ status values. |
| HANDLE | h | A handle. |
| TCHAR, wchar\_t, char | ch | A character. |
| PWSTR, PCWSTR, wchar\_t \*, char \* PCSTR, PSTR | psz | A pointer to a zero-terminated string. |
| char [], wchar\_t [] | sz | A zero-terminated string in the form of a character array on the stack. For example, wchar\_t szMessage[BUFFER\_SIZE]; |
| BSTR | bstr | An OLE string. |
| VARIANT | vt | An OLE VARIANT. |

### Prefixes

Use the following in identifier names:

|  |  |
| --- | --- |
| Prefix | Description |
| g\_ | A global variable. |
| p | A pointer. |
| sp | A ‘smart’ pointer. |
| c | A count. |
| \_ | A member variable in a class. |
| I | A COM interface |

## Unicode/TCHAR

Make use of the “T” types and Windows API functions where possible. Put all string literals in \_T() macros.

Don’t use the standard C library functions such as strcpy, strcat, etc. Instead use \_tcscpy, \_tcscat, etc. UNICODE (\_UNICODE) should be #defined in order to build your code for Unicode. When UNICODE (\_UNICODE) is not defined, you are building for ANSI.

## Array and Struct Init

Use “={}” to zero array and structure memory.

e.g.

MYSTRUCT myst = {};

## Macros

Do not use unless absolutely necessary. See if your functionality can be built using enums, consts, inline functions or templates.

## Parameters

In public facing functions (e.g. the public functions that you expose from your library) validation of all parameters is required. In case an invalid parameter is encountered, return the Win32 code ERROR\_INVALID\_PARAMETER or equivalent.

For unreferenced parameter, do not use UNREFERENCED\_PARAMETER(). Instead don’t define a name for the parameter in the argument list of the function:

int GetWindows( HANDLE hParentWnd,

int); // not used so name not specified

## Struct vs Class

Use struct when there are only data members that need to be aggregated together and no functions are involved, for which there are classes.

## Classes

Do not use public data members, use inline accessor functions.

Use the initializer list in constructors to initialize data members.

Keep in mind the overhead of calling virtual functions.

### Constructors

Do minimal work in constructors such as member init.

Define all constructors that take 1 argument with the explicit keyword, so that they are not conversion constructors then. E.g.

class MyClass

{

Public:

explicit MyClass(int value);

}

If copying is not supported by the class, disallow it by providing a private, unimplemented copy constructor & assignment operator. If defining a copy constructor, take a const reference as argument.

### Operator Overloading

Do not change the semantics of operators by overloading them, for example don’t overload the ‘+’ operator for performing subtraction. Don’t overload operators unless the semantics of the class justify them.

### Destructors

Provide a public virtual destructor if you allow deletion via a pointer to the base class or a protected, non-virtual destructor to disallow deletion via a pointer to the abstract class.

### Errors and Exceptions

User error code return values as against throwing exceptions. Do check the return code of functions and handle errors appropriately.

Throw exceptions by value and catch them by reference. Rethrow caught exceptions by simply using “throw” and not “throw caught\_exception”;

Do not use “catch (…)”. Catch specific exceptions and let the other go unhandled, which will help in detecting programming errors.

Do not use exceptions for controlling flow.

## Resource Handling

Make sure to cleanup/release resources you allocated for use in your program (e.g. free memory, delete objects, release handles, etc.), especially in cases where a “rollback” needs to be performed within a function that has multiple steps in it and one of the steps failed because of which the function needs to return an error.

## Return from Function

Avoid early or multiple return statements in a function as such code is difficult to maintain. Have just one exit point from a function, at the bottom (before the end) of it.

# References

For .NET

C# guidelines

<https://msdn.microsoft.com/en-us/library/ff926074.aspx>

Framework Design guidelines

Contains guidelines for various things such as naming, type design, member design, exceptions, common design patterns.

<https://msdn.microsoft.com/en-us/library/ms229042%28v=vs.110%29.aspx>

For C++

Visual Studio 6.0 C++ Guidelines

<https://msdn.microsoft.com/en-us/library/aa260844%28VS.60%29.aspx>

# Works Cited

Microsoft. (n.d.). *All-In-One Code Framework.* Retrieved from Codeplex: http://1code.codeplex.com/downloads/get/357518