Coding Guidelines C#, C++

History

|  |  |  |  |
| --- | --- | --- | --- |
| Version | Date | Description | Author / Editor |
| 0.1 | 01.07.2014 | Initial Version | R. Bernhardt |
| 0.2 | 08.07.2014 | Draft of General and C# section | R. Bernhardt |
| 0.3 | 19.07.2014 | Extension of general and C++ section | R. Bernhardt |
| 0.4 | 24.07.2014 | Cleanup of document; C++/CLI added | R. Bernhardt |
| 1.0 | 31.07.2014 | Integration of review results | R. Bernhardt |
| 1.0.9 | 07.11.2014 | Preview version with identifiers and cross-reference for StyleCop / Code Analysis integration. | J. v.d. Smissen |
| 1.1 | 19.02.2015 | Rules partly sharper defined to allow automatically verification via StyleCop / Code Analysis. Cross reference: see CodingGuidelines\_Matrix.xlsx  **Status: Ready for final review** | O. Cox |
| 1.2 | 09.04.2015 | Review | C. Trojahn |
|  | 9.6.2015 | Chapters which needs adaption for BVMS are marked with yellow color. | O. Cox |

Approval

|  |  |  |
| --- | --- | --- |
| Version | Date | Approval |
| 1.0 | 31.07.2014 |  |
|  |  |  |

***9.6.2015 – REMARK BY COX:***

**All yellow marked chapters are adapted for BVMS in a specific Coding Guidelines document.**

**All non-marked chapters are valid for BVMS too.**

Table of Contents

1 Overview 4

1.1 Scope 4

1.2 Document conventions 4

1.3 Terminology 4

2 General Coding Standards 6

2.1 Clarity and Consistency 6

2.2 Formatting and Style 6

2.3 Projects and Solutions 7

2.4 Using Libraries 7

2.5 Usage of Code Snippets 8

2.6 Literals 9

2.7 Global Variables 9

2.8 Variable Declarations and Initializations 9

2.9 Function Declarations and Calls 10

2.10 Statements 11

2.11 Control Flow 11

2.12 Enums 13

2.13 Whitespace 16

2.14 Braces 17

2.15 Comments 18

2.16 Design Guidelines 24

3 C# Coding Standards 25

3.1 Files and Structure 25

3.2 Regions 25

3.3 Assembly Properties 26

3.4 Naming Conventions 26

3.5 Constants 29

3.6 Strings 30

3.7 Arrays and Collections 31

3.8 Structures 32

3.9 Structures vs. Classes 33

3.10 Interfaces 33

3.11 Classes 33

3.12 Errors and Exceptions 39

3.13 Resource Cleanup 43

3.14 Interop 50

3.15 LINQ – Language INtegrated Query 52

3.16 Asynchronous Programming 55

4 C++ Coding Standards 60

4.1 Compiler Options 62

4.2 Files and Structure 63

4.3 Naming Conventions 65

4.4 Pointers 66

4.5 Constants 68

4.6 Constness 68

4.7 Casting 69

4.8 Sizeof 69

4.9 Strings 70

4.10 Arrays 71

4.11 Macros 71

4.12 Expressions 72

4.13 Functions and Operators 72

4.14 Structures 76

4.15 Classes 76

4.16 Namespaces 86

4.17 Allocations 87

4.18 Errors and Exceptions 89

4.19 Control Flow 91

4.20 Debug Macro Usage 91

4.21 STL 92

5 C++/CLI 94

5.1 Bridges 94

5.2 Control Flow 95

5.3 Resource Management 95

5.4 Mixing Managed and Native Types 96

5.5 Marshalling 98

5.6 Delegates 98

5.7 Exceptions 100

6 References 101

Overview

## Scope

This document defines the C# and C++ coding guidelines used within the Endeavour products and services development. The guidelines are derived in structure and contents from the coding standards used in the [All-In-One Code Framework](http://1code.codeplex.com/) [[REF1]](#REF1). The C# part has been extended by rules that are defined in the "Framework and Design Guidelines, Krzysztof, Abrams" book [[REF2]](#REF2). This book is also partially available online [here](http://msdn.microsoft.com/en-us/library/ms229042(v=vs.110).aspx). The C++ part was extended by rules defined in [[REF3]](#REF3). All rules have been adapted to be in conformance with the C++11 standard. Existing Coding Guidelines of other projects (BVMS, UGM2040, BIS) were also taken into consideration and rules have been extracted from there where appropriate.

During development these guidelines will be continuously adjusted and extended by the lessons learned during development.

## Document conventions

Through-out this document there will be recommendations or suggestions for standards and practices. Some practices are very important and must be followed, others are guidelines that are beneficial in certain scenarios but are not applicable everywhere. In order to clearly state the intent of the standards and practices that are discussed we will use the following terminology.

|  |  |  |
| --- | --- | --- |
| Wording | Intent | Justification |
| 🗹 Do... | This standard or practice should be followed in all cases. If you think that your specific application is exempt, it probably isn't. | These standards are present to mitigate bugs. |
| 🗷 Do Not... | This standard or practice should never be applied. |
| 🗹 You should... | This standard or practice should be followed in most cases. | These standards are typically stylistic and attempt to promote a consistent and clear style. |
| 🗷 You should not... | This standard or practice should not be followed, unless there's reasonable justification. |
| 🗹 You can… | This standard or practice can be followed if you want to; it's not necessarily good or bad. There are probably implications to following the practice (dependencies, or constraints) that should be considered before adopting it. | These standards are typically stylistic, but are not ubiquitously adopted. |

### Guideline numbering

All rules in this document have a unique identifier associated with them. These identifiers can be referenced to from other documents or from code (examples). When editing this document make sure that existing numbers are not changed.

In order to make the guideline numbers more or less independent of the document structure they use abbreviations of the chapter and sector they reside in (e.g. GEN/CC). Within a sector sequential numbers are used with gaps between them to allow for insertions of future rules.

## Terminology

The following terminology is referenced throughout this document:

Access Modifier

C# keywords public, protected, internal, and private declare the allowed code-accessibility of types and their members. Although default access modifiers vary, classes and most other members use the default of private. Notable exceptions are interfaces and enums which both default to public.

Camel Case

A word with the first letter lowercase, and the first letter of each subsequent word-part capitalized.

Example: customerName

Common Type System

The .NET Framework common type system (CTS) defines how types are declared, used, and managed. All native C# types are based upon the CTS to ensure support for cross-language integration.

Identifier

A developer defined token used to uniquely name a declared type or an instance of a type.

Example: public class MyClassNameIdentifier { … }

Literals

A literal is a notation for representing a fixed value in source code. Literals are defined for simple types as integers, floating point numbers and strings.

Magic Number

Any numeric literal used within an expression (or to initialize a variable) that does not have an obvious or well-known meaning. This usually excludes the integers 0 or 1 and any other numeric equivalent precision that evaluates as zero.

Pascal Case

A word with the first letter capitalized, and the first letter of each subsequent word-part capitalized.

Example: CustomerName

POD

Plain Old Data: PODS is a data structure that is represented only as passive collections of field values, without using object-oriented features.

OSS

Open Source Software is software with its source code made available and licensed with a license in which the copyright holder provides the rights to study, change and distribute the software under conditions described by that license.

General Coding Standards

These general coding standards can be applied to all languages - they provide high-level guidance to the style, formatting and structure of your source code.

## Clarity and Consistency

G/CC10 🗹 Do ensure that clarity, readability and transparency are paramount. These coding standards strive to ensure that the resultant code is easy to understand and maintain, but nothing beats fundamentally clear, concise, self-documenting code.

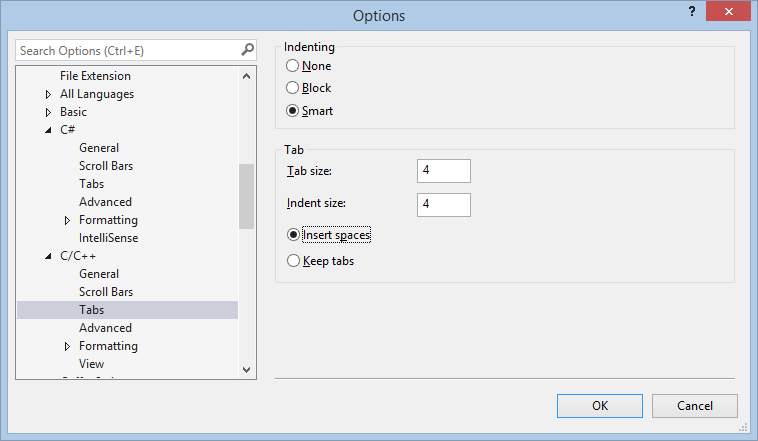
G/CC20 🗹 Do ensure that when applying these coding standards that they are applied consistently. Note that where possible automatic validation of the rules should be performed using:

* Assembly level analysis using static code analysis in Visual Studio (formerly known as FxCop before it was integrated into Visual Studio). This validation can be performed on all managed assemblies, regardless of the language that created them
* Source code level analysis using StyleCop. This tool can be used for C# code.

## Formatting and Style

G/FS10 🗷 Do not use tabs. It's generally accepted that tabs shouldn't be used in source files - different text editors use different spacing to render tabs, and this causes formatting confusion. All code should be written using four spaces for indentation.

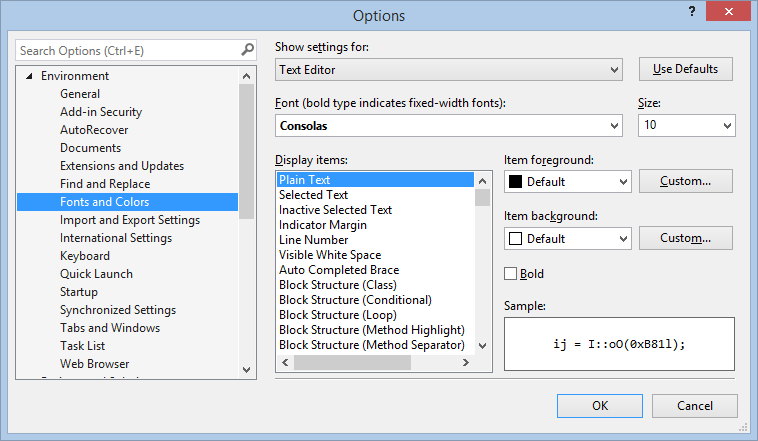
Visual Studio text editor can be configured to insert spaces for tabs.



G/FS20 🗹 You must limit the length of lines of code. Having overly long lines inhibits the readability of code. Break the code line when the line length is greater than column 120 for readability.

*Remark: You may install VisualStudio extension “Editor Guidelines” to display vertical line in text editor to visualize the character limitation.*

G/FS30 🗹 Do use a fixed-width font, typically Consolas, in your code editor.



## Projects and Solutions

G/PS10 🗷 Do not create new projects or solutions. New projects or solutions are introduced by the architects of Endeavour and are not invented in an on-demand fashion. If there seems to be a need for an additional project provide this as input to the architects in the CTO Office.

G/PS20 🗹 Do verify that references to other projects are in accordance to the allowed dependencies as described by the layer diagram of the application when adding a new reference to a project and if in doubt ask the Global Software Architecture Group (GSAG) in the CTO Office.

The properties of assemblies can be found in §3.3.

## Using Libraries

G/UL10 🗷 Do not reference unnecessary libraries, include unnecessary header files, or reference unnecessary assemblies. Paying attention to small things like this can improve build times, minimize chances for mistakes, and give readers a good impression.

G/UL20 🗷 Do not introduce yourself new libraries to be used in Endeavour. Decisions about the usage of libraries are an obligation of the architects (GSAG). If there is a need to use a new library provide this as input to the architects. The introduction of a new library must include the check of the license to be a usable in Endeavour context (e.g. is listed on the Open Source Software White List).

G/UL30 🗹 Do prefer integration of the OSS library by dynamically linking against statically linking the OSS as some OSS licenses there are more obligations when linking statically, as opposed to linking dynamically. By linking dynamically, less obligations must be fulfilled.

## Usage of Code Snippets

The general guideline for use of Open Source Software components is: always check the associated Open Source license and **get architectural approval**. The rule set below gives you some hints about the most important aspects of Open Source Software licenses. When it is in doubt whether an Open Source Software component is problematic or not, please contact your Open Source Software responsible.

G/CS10 🗹 Do prefer introducing an unchanged library in binary form to Endeavour instead of copying parts as code snippets of that library to the Endeavour code base. Follow the rules of the previous chapter when doing so.

If the preceding rule does not apply, follow these rules:

G/CS20 🗹 Do prefer introducing unchanged source files against modified code snippets.

G/CS30 🗹 Do document all changes made to the sources when changes can't be avoided. A proper documentation includes:

* Why were the changes necessary.
* Who changed it.
* What was changed.
* When was it changed.

G/CS40 🗷 Do not remove license related comments (e.g. copyright statements, change history) from any code.

G/CS50 🗹 Do follow the rules of the open source license at hand when using code snippets of that software in Endeavour. Mark clearly the region where the code snippet starts and ends and document the source where it was taken from.

Examples:

Code Project Open License

Requires to mention the author in particular:

// The following code snippet/function/class comes from the OSS component CFolderDialog

// from http://www.codeproject.com/Articles/2024/CFolderDialog-Selecting-Folders which was written

// by Armen Hakobyan and is licensed under the Code Project Open License 1.02.

Phyton Software Foundation

As another example, asks to describe changes made (besides of course a full-blown license text required in e.g. open\_source\_licenses.txt):

// The following code comes from http://xxx and is licensed under the Python Software Foundation License

// It was modified to deal with empty or unicode characters

BSD / MIT

Asks to keep also the license text at the header section of the file in which the snippet is used:

// … here: Bosch header

//

// This file uses code that was taken from <http://www.mathworks.de/matlabcentral/fileexchange/16663-simple-waitbar>

// It is licensed under the following license

// Copyright (c) 2008, Rahul PN

// All rights reserved.

//

// Redistribution and use in source and binary forms, with or without modification,

// are permitted provided that the following conditions are met:

//

//    \* Redistributions of source code must retain the above copyright

//      notice, this list of conditions and the following disclaimer.

//    \* Redistributions in binary form must reproduce the above copyright

//      notice, this list of conditions and the following disclaimer in

//      the documentation and/or other materials provided with the distribution

//

// THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"

// AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE

// IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE

// ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE

// LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR

// CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF

// SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS

// INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN

// CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)

// ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE

// POSSIBILITY OF SUCH DAMAGE.

And at the snippet:

// code snippet © Rahuld PN begins

...

// code snippet ends

G/CS60 🗷 Do not use code snippets from sources with unclear license terms.

## Literals

G/LT10 🗷 Do not use literals (especially for strings) directly in the code where the constant value is needed but define a symbolic name (constant) for that literal.

Good:

const string MAGIC\_TEXT = "Magic Text";

// Somewhere in code

foo.DoSomething(MAGIC\_TEXT);

Bad:

// Somewhere in code

foo.DoSomething("Magic Text");

This is also valid for usage of exceptions. E.g. do not use literals in ArgumentNullException.

*Hint: Please use class Foundations.Common.ArgumentGuard to simplify argument validation. In this case there is no need to use literals.*

*Sample:*

Bad:

public ResourceManager(IDependencyResolver dependencyResolver,

CultureInfo defaultCulture)

{

if (dependencyResolver == null)

throw new ArgumentNullException("dependencyResolver");

if (defaultCulture == null)

throw new ArgumentNullException("defaultCulture");

// ...

}

Good:

public ResourceManager(IDependencyResolver dependencyResolver,

CultureInfo defaultCulture)

{

ArgumentGuard.Validate(() => dependencyResolver);

ArgumentGuard.Validate(() => defaultCulture);

//…

}

## Global Variables

G/GLB10 🗹 Do minimize global variables. To use global variables properly, always pass them to functions through parameter values. Never reference them inside of functions or classes directly because doing so creates a side effect that alters the state of the global without the caller knowing. The same goes for static variables. If you need to modify a global variable, you should do so either as an output parameter or return a copy of the global.

## Variable Declarations and Initializations

G/VAR10 🗹 Do declare local variables in the minimum scope block that can contain them, typically just before use if the language allows; otherwise, at the top of that scope block.

G/VAR20 🗹 Do initialize variables when they are declared (especially pointers in C++). (Rule is valid for C++ in general and for C# for variables which are immediately used as ref parameters.)

G/VAR30 🗹 Do declare and initialize/assign local variables on a single line where the language allows it. This reduces vertical space and makes sure that a variable does not exist in an un-initialized state or in a state that will immediately change. (Rule is valid for simple data types and for pointers.)

// C++ sample:

int \*pValue = nullptr;

std::complex<double> c{ 2.71828, 3.14159 };

vector<int>     v{ 1, 2, 3, 4 };

STARTUPINFO si = { sizeof(si) };

PROCESS\_INFORMATION pi = { 0 };

// C# sample:

string name = myObject.Name;

int val = time.Hours;

G/VAR40 🗷 Do not declare multiple variables in a single line. One declaration per line is recommended since it encourages commenting, and could avoid confusion. As a Visual C++ example,

Good:

CodeExample \*pFirst = nullptr; // Pointer of the first element.

CodeExample \*pSecond = nullptr; // Pointer of the second element.

Bad:

CodeExample \*pFirst, \*pSecond;

The latter example is often mistakenly written as:

CodeExample \*pFirst, pSecond;

Which is actually equivalent to:

CodeExample \*pFirst;

CodeExample pSecond;

## Function Declarations and Calls

G/FUN10 🗹 The function/method name, return value and parameter list can take several forms. Ideally this can all fit on a single line. If there are many arguments that don't fit on a line those can be wrapped, many per line or one per line. Put the return type on the same line as the function/method name. For example,

Single Line Format:

// C++ function declaration sample:

HRESULT DoSomeFunctionCall(int param1, int param2, int \*param3);

// C++ / C# function call sample:

hr = DoSomeFunctionCall(param1, param2, param3);

Multiple Line Formats:

// 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);

… or alternatively (e.g. in C++ when parameters have comments):

// C++ function declaration 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# function call sample:

hr = DoSomeFunctionCall(

hwnd,

param1,

param2,

param3,

param4,

param5);

G/FUN30 🗹 Do order parameters, grouping the *in* parameters first, the *out* parameters last.

G/FUN40 🗹 Do order parameters within the *in* or *out* group, based on what will help programmers supply the right values.  
Example: If a function takes arguments named “left” and “right”, put “left” before “right” so that their place match their names.

G/FUN50 🗹 When designing a series of functions which take the same arguments, you **must** use a consistent order across the functions.  
Example: If one function takes an input handle as the first parameter, all of the related functions should also take the same input handle as the first parameter.

## Statements

G/ST10 🗷 Do not put more than one statement on a single line because it makes stepping through the code in a debugger much more difficult.

Good:

// C++ / C# sample:

a = 1;

b = 2;

Bad:

// C++ / C# sample:

a = 1; b = 2;

## Control Flow

G/CF10 🗷 You should not early return in most functions. Early returns are acceptable in some situations, but they should be avoided. Ideally functions will have a single return point at the bottom that all execution leads to.

Acceptable situations for early returns are:

* Parameter validation done at the very beginning of a function
* Extremely small functions (like assessors to member variable state)

G/CF30 🗹 Do prefer range-for/foreach statement over the "conventional" for statement.

Good:

// C#

foreach(var item in list)

{

    // ...

}

Instead of:

for (int i = 0; i < list.Count; ++i)

{

    // ...

}

// C++11

for (auto x : v)

{

    cout << x << '\n';

}

for (auto& x : v)

{

    ++x; // using a reference to allow us to change the value

}

G/CF40 In C++ the range-for statement allows to iterate through everything that can be iterated similar like an STL-sequence defined by begin() and end().The begin() (and end()) can be a member to be called x.begin() or a free-standing function to be called begin(x).   
Example:

class Range

{

public:

    class Iterator {

        friend class Range;

    public:

        long int operator \*() const

        { return i\_; }

        const Iterator &operator ++()

        { ++i\_; return \*this; }

        Iterator operator ++(int)

        { Iterator copy(\*this); ++i\_; return copy; }

        bool operator ==(const Iterator &other) const

        { return i\_ == other.i\_; }

        bool operator !=(const Iterator &other) const

        { return i\_ != other.i\_; }

    protected:

        Iterator(long int start) : i\_(start)

        { }

    private:

        unsigned long i\_;

    };

    Iterator begin() const

    { return begin\_; }

    Iterator end() const

    { return end\_; }

    Range(long int  begin, long int end)

        : begin\_(begin), end\_(end) {}

private:

    Iterator begin\_;

    Iterator end\_;

};

G/CF50 🗹 You should provide a default clause for all switch statements.

G/CF60 🗷 Do not use the *goto* statement.

G/CF70 🗷 Do not use more than 4 indentation levels for blocks (beginning on method level). If it's reasonable extract a method instead.

G/CF80 🗹 You should apply the following rules for complex conditions, which are linked with logical operators:

* Additional sub conditions are listed in a separate line, which starts by the connecting operation
* All conditions have to be listed on the same indentation level

If (complexCondition1 &&

    complexCondition2 &&

    (complexCondition3 ||

     complexCondition4) &&

    complexCondition5)

{

}

G/CF90 🗹 You should extract complex conditions to an own methods.

## Enums

G/EN10 🗹 You should use an enum to strongly type parameters, properties, and return values that represent sets of values.

G/EN20 🗹 Do favor using an enum over static constants or “#define” values . An enum is a structure with a set of static constants. The reason to follow this guideline is because you will get some additional compiler and reflection support if you define an enum versus manually defining a structure with static constants.

G/EN30 🗹 Do favor using the new enum classes of C++11 over conventional enums. This has the following advantages over conventional enums:

* Conventional enums implicitly convert to int. This causes errors when someone does not want an enumeration to act as an integer.
* Conventional enums export their enumerators to the surrounding scope. This may cause name clashes.
* The underlying type of an enum cannot be specified. This can cause confusion, compatibility problems, and makes forward declaration impossible.

Good:

// C++ sample:

enum Color

{

Red = 1,

Green = 2,

Blue = 3

};

// ... or even better

enum class Color

{

Red = 1,

Green = 2,

Blue = 3

};

//usage of the enum class

auto color = Color::Blue;

// error: no implicit conversion

int color2 = Color::Red;

// C# sample:

public enum Color

{

Red = 1,

Green = 2,

Blue = 3

}

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;

}

G/EN40 🗷 Do not use an enum for open sets (such as the operating system version, names of your friends, etc.).

G/EN50 🗹 Do provide a value of zero on simple enums. Consider calling the value something like “None.” If such value is not appropriate for this particular enum, the most common default value for the enum should be assigned the underlying value of zero.

// C++ sample:

enum Compression

{

None = 0,

GZip = 1,

Deflate = 2

};

// C# sample:

public enum Compression

{

None = 0,

GZip = 1,

Deflate = 2

}

G/EN55 🗹 Do explicitly set the value of every enum item.  
This enforces you to think about migration in case of extending the enumeration later.

Good:

// C++ sample:

enum Compression

{

None = 0,

GZip = 1,

Deflate = 2

};

// C# sample:

public enum Compression

{

None = 0,

GZip = 1,

Deflate = 2

}

Bad:

// C# sample:

public enum Compression

{

None = 0,

GZip,

Deflate

}

G/EN60 🗷 Do not use Enum.IsDefined for enum range checks in .NET. There are really two problems with Enum.IsDefined. First it loads reflection and a bunch of cold type metadata, making it a surprisingly expensive call. Second, there is a versioning issue here.

Good:

// C# sample:

if (c > Color.Black || c < Color.White)

{

throw new ArgumentOutOfRangeException(...);

}

Bad:

// C# sample:

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

{

throw new InvalidEnumArgumentException(...);

}

### Flag Enums

Flag enums are designed to support bitwise operations on the enum values. A common example of the flags enum is a list of options.

G/FEN10 🗹 Do apply the System.FlagsAttribute to flag enums in .NET. Do not apply this attribute to simple enums.

G/FEN20 🗹 Do use powers of two for the flags enum values so they can be freely combined using the bitwise OR operation. For example,

// C++ sample:

enum class AttributeTargets

{

None = 0x0000,

Assembly = 0x0001,

Class = 0x0002,

Struct = 0x0004

...

};

// C# sample:

[Flags]

public enum AttributeTargets

{

None = 0x0000,

Assembly = 0x0001,

Class = 0x0002,

Struct = 0x0004,

...

}

G/FEN30 🗹 Do provide special enum values for commonly used combinations of flags. Bitwise operations are an advanced concept and should not be required for simple tasks. FileAccess.ReadWrite is an example of such a special value. However, you should not create flag enums where certain combinations of values are invalid.

// C++ sample:

enum class FileAccess

{

None = 0x0,

Read = 0x1,

Write = 0x2,

ReadWrite = Read | Write

};

// C# sample:

[Flags]

public enum FileAccess

{

None = 0x0,

Read = 0x1,

Write = 0x2,

ReadWrite = Read | Write

}

G/FEN40 🗷 Do not use flag enum values of zero, unless the value represents “all flags are cleared” and is named appropriately as “None”. The following C# example shows a common implementation of a check that programmers use to determine if a flag is set (see the if-statement below). The check works as expected for all flag enum values except the value of zero, where the Boolean expression always evaluates to 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

{

None = 0x0,

Fixed3D = 0x1,

FixedSingle = 0x2

}

if (foo.BorderStyle == BorderStyle.None)

{

...

}

## Whitespace

### Blank Lines

G/BL10 🗹 You should use blank lines to separate groups of related statements. Omit extra blank lines that do not make the code easier to read. For example, you can have a blank line between variable declarations and code.

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(...)

{

}

}

In this example of bad usage of blank lines, there are multiple blank lines between the local variable declarations, and multiple blank likes after the ‘if’ block.

### Spaces

Spaces improve readability by decreasing code density. Here are some guidelines for the use of space characters within code:

G/SP10 🗹 You must use spaces within a line as follows.

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

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

G/SP20 🗹 Do use at least one space before and after „+“, „+=“ etc. (except for end of lines)

G/SP30 🗷 Do not use space between operators „.“ and „->“.

G/SP40 🗷 Do not use space between function/method names and brackets.

G/SP50 🗷 Do not use space after opening bracket and before closing bracket of a function/method declaration, definition or call.

## Braces

G/BR10 🗹 Do use Allman bracing style.

The Allman style is named after Eric Allman. It is sometimes referred to as "ANSI style". The style puts the brace associated with a control statement on the next line, indented to the same level as the control statement. Statements within the braces are indented to the next level.

Good:

// C++ / C# sample:

if (x > 5)

{

y = 0;

}

Bad:

// C++ / C# sample:

if (x > 5) {

y = 0;

}

G/BR20 🗹 You must use braces around single line conditionals. Doing this makes it easier to add code to these conditionals in the future and avoids ambiguities should the tabbing of the file become disturbed.

Good:

// C++ / C# sample:

if (x > 5)

{

y = 0;

}

Bad:

// C++ / C# sample:

if (x > 5) y = 0;

## Comments

G/CM10 🗹 Do use English as language for comments.

G/CM20 🗹 Do use comments that summarize what a piece of code is designed to do and why. Do not use comments to repeat the code.

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.

G/CM30 🗹 You must use ‘//’ comments instead of ‘/\* \*/’ for comments. The single-line syntax (// …) is preferred even when a comment spans multiple lines.

// 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)

{

}

G/CM40 🗹 You should indent comments at the same level as the code they describe.

G/CM50 🗹 You must use full sentences with initial caps, a terminating period and proper punctuation and spelling in comments.

Good:

// Intialize the components on the Windows Form.

InitializeComponent();

Bad:

//intialize the components on the Windows Form.

InitializeComponent();

### Inline Code Comments

G/ICM10 🗹 Inline comments **must** be included on their own line and should be indented at the same level as the code they are commenting on, with a blank line before, but none after. Comments describing a block of code should appear on a line by themselves, indented as the code they describe, with one blank line before it and one blank line after it. For example:

if (MAXVAL >= exampleLength)

{

// Reprort the error.

ReportError(GetLastError());

// The value is out of range, we cannot continue.

return E\_INVALIDARG;

}

G/ICM20 🗹 Inline comments are permissible on the same line as the actual code only when giving a brief description of a structure member, class member variable, parameter, or a short statement. In this case it is a good idea to align the comments for all variables. For example:

// C++ sample:

class Example

{

public:

...

void TestFunction()

{

...

do

{

...

}

while (!finished); // Continue if not finished.

}

private:

int m\_length; // The length of the example

float m\_accuracy; // The accuracy of the example

};

G/ICM30 🗷 You should not drown your code in comments. Commenting every line with obvious descriptions of what the code does actually hinders readability and comprehension. Single-line comments should be used when the code is doing something that might not be immediately obvious.

The following example contains many unnecessary comments:

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

}

}

A better implementation would be:

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;

}

}

G/ICM40 🗹 You should add comments to call out non-intuitive or behavior that is not obvious from reading the code.

### File Header Comments

G/HCM10 🗹 Do have a file header comment at the start of every human-created code file. The header comment templates are as follows:

C# file header comment template:

// <copyright company="Bosch Security Systems GmbH">

// PLACEHOLDER\_FOR FINAL\_FILEHEADER.

// </copyright>

C++ file header comment template:

// <copyright company="Bosch Security Systems GmbH">

// PLACEHOLDER\_FOR FINAL\_FILEHEADER.

// </copyright>

Remark: The placeholder will be replaced after final copyright text definition.

### Class Comments

G/CCM10 🗹 You must provide banner comments for all classes and structures that are non-trivial. The level of commenting should be appropriate based on the audience of the code.

In C++ use the following style which is supported by Doxygen.

C++ class comment template:

/\*\*

\* <Class description>

\*/

In C# use .NET descriptive XML Documentation comments. When you compile .NET projects with /doc the compiler will search for all XML tags in the source code and create an XML documentation file.

C# class comment template:

/// <summary>

/// <Class description>

/// </summary>

For example,

/\*\*

\* 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

{

...  
}

### Function Comments

G/FCM10 🗹 You should provide banner comments for all public and non-public functions that are not trivial. The level of commenting should be appropriate based on the audience of the code.

In C++ use the following style which is supported by Doxygen.

C++ function comment template:

/\*\*

\* <function description>

\* @param <parameter name> <parameter description>   
 \* @return <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# use descriptive XML Documentation comments. At least a <summary> element and also a <parameters> element and <returns> element, where applicable, are required. Methods that throw exceptions should make use of the <exception> element to indicate to consumers what exceptions may be thrown.

C# function comment template:

/// <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>

For example,

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

FUNCTION: IsUserInAdminGroup(HANDLE token)

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:

token – 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(token))

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>

G/FCM20 🗹 Do document all exceptions thrown by publicly callable members because of a violation of the member contract (rather than a system failure) and treat them as part of your contract.

G/FCM30 Any method or function which can fail with side-effects should have those side-effects clearly communicated in the function comment. As a general rule, code should be written so that it has no side-effects in error or failure cases; the presence of such side-effects should have some clear justification when the code is written. (Such justification is not necessary for routines which zero-out or otherwise overwrite some output-only parameter.)

### Commenting Out Code

G/OCM10 🗷 You must not comment out code to preserve an old version of a code section. Your version control system is perfectly suited to remember old versions.

G/OCM20 🗹 You can use commented code to show the usage of a class as part of its documentation.

/// <summary>

/// The GetZero method.

/// </summary>

/// <example>

/// This sample shows how to call the <see cref="GetZero"/> method.

/// <code>

/// class TestClass

/// {

///     static int Main()

///     {

///         return GetZero();

///     }

/// }

/// </code>

/// </example>

public static int GetZero()

{

    return 0;

}

## Design Guidelines

G/DG10 🗹 Do give one entity (variable, class, function, namespace, module, library) one cohesive responsibility.

G/DG20 🗹 Do prefer a correct and simple solution over a fast but complex (and therefore presumably incorrect) solution (KISS principle: Keep It Simple and Stupid).

G/DG30 🗹 Do decide when you need to program for scalability. Balance the principles "don't optimize prematurely" and "don't pessimize prematurely" [[REF3]](#REF3). Keep the O(n) notion in mind when choosing data structures and algorithm. In general prefer algorithms that scale linear or better and avoid any polynomial or exponential behavior.

G/DG40 🗹 Do minimize global and shared data. Global and shared data increases coupling and decreases maintainability and performance.

G/DG50 🗹 Do hide information. Expose abstractions instead of concrete data. See [[REF4]](#REF4).

G/DG60 🗹 Do code with concurrency requirements in mind. In particular, design types to support different instances used on different threads works without any measures taken by the client. Document how concurrency must be handled when one object is accessed from more than one thread.

C# Coding Standards

These coding standards are applied to C#.

## Files and Structure

CS/FS10 🗷 Do not have more than one type in a source file, unless they differ only in the number of generic parameters or one is nested in the other.

CS/FS20 🗹 Do name the source file with the name of the type it contains. For example, MainForm class should be in MainForm.cs file and List<T> class should be in List.cs file.

CS/FS25 🗹 You must use the following rules for order using directives:

* A using directive which declares a member of the System namespace must be placed before other using directives. All these system using directives must be ordered alphabetically.
* All other using directives must be ordered alphabetically.
* A using-alias directive must be placed after all other using directives. All these using-alias directives must be ordered alphabetically.

CS/FS30 🗹 You must use the following class layout rules:

Within a class, struct or interface use the following order

* Constant Fields
* Fields
* Constructors
* Finalizers (Destructors)
* Delegates
* Events
* Enums
* Interfaces
* Properties
* Indexers
* Methods
* Structs
* Classes

Within each of these groups order by access:

* public
* internal
* protected internal
* protected
* private

## Regions

CS/REG10 🗹 Do use regions to support the organization of the first level of the described structure of a class by defining the following regions:

Fields Region (Constant Fields, Fields)

Constructor/Finalizer Region (Constructors, Finalizers (Destructors))

Delegates Region (Delegates, Events)

Enums Region (Enums)

Properties Region (Properties, Indexers)

Methods Region (Methods)

Inner Types Region (Structs,Classes)

CS/REG20 🗷 Do not define a region with an empty content.

Example:

#region Fields

...

#endregion

#region Constructor/Finalizer

...

#endregion

#region Properties

...

#endregion

#region Methods

...

#endregion

CS/REG30 🗷 Do not embed regions in regions.

## Assembly Properties

CS/AP10 The assembly should contain the appropriate property values describing its name, copyright, and so on.

|  |  |
| --- | --- |
| Standard | Example |
| Set Copyright to Copyright © Bosch Security Systems 2014 | [assembly: AssemblyCopyright("Copyright © Bosch Security Systems 2014")] |
| Set AssemblyCompany to Bosch Security Systems | [assembly: AssemblyCompany("Bosch Security Systems")] |
| Set the AssemblyTitle to the assembly name | [assembly: AssemblyTitle("Assembly Name")] |
| Set the AssemblyProduct to Endeavour. | [assembly: AssemblyProduct("Endeavour")] |

## Naming Conventions

### General Naming Conventions

CS/GNC10 🗹 Do use meaningfull names for various types, functions, variables, constructs and types.

CS/GNC20 🗷 You should not use of shortenings or contractions as parts of identifier names. For example, use “GetWindow” rather than “GetWin”. For functions of common types, thread procs, window procedures, dialog procedures use the common suffixes for these “ThreadProc”, “DialogProc”, “WndProc”.

CS/GNC30 🗷 Do not use underscores, hyphens, or any other non-alphanumeric characters but for protected and private fields.

### Naming of Assembly, Namespace

CS/NAN10 🗹 Do name the assembly and the corresponding namespace according to the structure of the sources in the source control system with the prefix Bosch.SecuritySystems added.  
Portable projects use the same namespace as the corresponding non-portable projects.

Example:

File ISomeInterface.cs located at Endeavour\0\_Foundations\Common\Foundations.Common

namespace Bosch.SecuritySystems.Foundations.Common

{

 public interface ISomeInterface

  {

        /// ...

  }

}

CS/NAN20 🗷 Do not use the same name for a namespace and a type in that namespace.

Bad:

namespace Bosch.SecuritySystems.Foundations.Core.Device

{

public class Device

  {

/// ...

Good:

namespace Bosch.SecuritySystems.Foundations.Core.Devices

{

public class Device

  {

/// ...

CS/NAN30 🗹 Do adjust namespaces when files are moved in the source control structure.

### Capitalization Naming Rules for Identifiers

CS/NRI10 The following table describes the capitalization and naming rules for different types of identifiers.

|  |  |  |  |
| --- | --- | --- | --- |
| Identifier | Casing | Naming Structure | Example |
| Class, Structure | PascalCasing | Noun | public class ComplexNumber {...}  public struct ComplextStruct {...} |
| Namespace | PascalCasing | Noun  🗷 Do not use the same name for a namespace and a type in that namespace. | namespace Bosch.SecuritySystems.Foundations.Core |
| Enumeration | PascalCasing | Noun  🗹 Do name flag enums with plural nouns or noun phrases and simple enums with singular nouns or noun phrases. | [Flags]  public enum ConsoleModifiers  { Alt, Control } |
| Method | PascalCasing | Verb or Verb phrase | public void Print() {...}  public void ProcessItem() {...} |
| Public Property | PascalCasing | Noun or Adjective  🗹 Do name collection properties with a plural phrase describing the items in the collection, as opposed to a singular phrase followed by “List” or “Collection”.  🗹 Do name Boolean properties with an affirmative phrase (CanSeek instead of CantSeek). Optionally, you can also prefix Boolean properties with “Is,” “Can,” or “Has” but only where it adds value. | public string CustomerName {get;set;}  public ItemCollection Items {get;set;}  public bool CanRead {get;set;} |
| Private Field | \_camelCasing | Noun or Adjective.  🗹 Do use the '\_' prefix. | private string \_name; |
| Non-private Field |  | 🗷 Do not use non-private fields. Use Property instead. See "Fields" section. |  |
| Event | PascalCasing | Verb or Verb phrase  🗹 Do give events names with a concept of before and after, using the present and past tense.  🗷 Do not use “Before” or “After” prefixes or postfixes to indicate pre and post events. | // A close event that is raised after the window is closed.  public event WindowClosed  // A close event that is raised before a window is closed.  public event WindowClosing |
| Delegate | PascalCasing | 🗹 Do add the suffix ‘EventHandler’ to names of delegates that are used in events.  Note: Make use EventHandler<TEventArgs> instead of defining a specific class.  🗹 Do add the suffix ‘Callback’ to names of delegates other than those used as event handlers.  🗷 Do not add the suffix “Delegate” to a delegate. | public delegate WindowClosedEventHandler |
| Interface | PascalCasing  ‘I’ prefix | Noun | public interface IDictionary |
| Constant | All capital | Noun | private const string MESSAGETEXT = "A";  public const double PI = 3.14159...; |
| Parameter, Variable | camelCasing | Noun | int customerName; |
| Generic Type Parameter | PascalCasing  ‘T’ prefix | Noun  🗹 Do name generic type parameters with descriptive names, unless a single-letter name is completely self-explanatory and a descriptive name would not add value.  🗹 Do prefix descriptive type parameter names with T.  🗹 You should using T as the type parameter name for types with one single-letter type parameter. | T, TItem, TPolicy |
| Resource | PascalCasing | Noun  🗹 Do provide descriptive rather than short identifiers. Keep them concise where possible, but do not sacrifice readability for space.  🗹 Do use only alphanumeric characters and underscores in naming resources. | ArgumentExceptionInvalidName |

### Naming Rules for Common Types

CS/NRT10 🗹 Do follow the rules in the following table when deriving from or implementing types in the .NET framework:

|  |  |
| --- | --- |
| Base Type | Guideline |
| System.Attribute | 🗹 Do add the suffix "Attribute" to names of custom attributes. |
| System.Delegate | 🗹 Do add the suffix "EventHandler" to names of delegates that are used in events.  🗹 Do add the suffix "Callback" to names of delegates other than those used as event handlers.  🗷 Do not add the suffix “Delegate” to a delegate. |
| System.EventArgs | 🗹 Do add the suffix "EventArgs.” |
| System.Exception | 🗹 Do add the suffix "Exception.” |
| System.Collections.ICollection | 🗹 Do add the suffix "Collection.” |
| System.Collections.IDictionary | 🗹 Do add the suffix "Dictionary.” |
| System.Collections.IEnumerable | 🗹 Do add the suffix "Collection.” |
| System.Collections.Queue | 🗹 Do add the suffix "Collection.” or "Queue.” |
| System.Collections.Stack | 🗹 Do add the suffix "Collection.” or "Stack.” |
| System.Collections.Generic.ICollection<T> | 🗹 Do add the suffix "Collection.” |
| System.Collections.Generic.IDictionary<TKey, TValue> | 🗹 Do add the suffix "Dictionary.” |
| System.Data.DataSet | 🗹 Do add the suffix "DataSet.” |
| System.Data.DataTable | 🗹 Do add the suffix "Collection.” or "DataTable.” |
| System.IO.Stream | 🗹 Do add the suffix "Stream.” |
| System.Security.IPermission | 🗹 Do add the suffix "Permission.” |
| System.Security.Policy.IMembershipCondition | 🗹 Do add the suffix "Condition.” |

### Capitalizing Acronyms

CS/CA10 🗹 Do not capitalize both characters of two-character acronyms. Example: System.Io

CS/CA20 🗹 Do capitalize only the first character of acronyms, except the first word of a camel-cased identifier. Example: System.Xml

CS/CA30 🗷 Do not capitalize any character of an acronym at the beginning of a camel-cased identifier.

### Capitalizing Compound Words

CS/CCW10 🗷 Do not capitalize each word in so-called closed-form compound words. These are compound words written as a single word, such as "endpoint".

### Hungarian Notation

CS/HN10 🗷 Do not use Hungarian notation (i.e., do not encode the type of a variable in its name) in .NET.

## Constants

CS/CST10 🗹 Do use constant fields for constants that will never change. The compiler burns the values of const fields directly into calling code. Therefore const values can never be changed without the risk of breaking compatibility.

public class Int32

{

public const int MaxValue = 0x7fffffff;

public const int MinValue = unchecked((int)0x80000000);

}

CS/CST20 🗹 Do use public static (shared) readonly fields for predefined object instances. If there are predefined instances of the type, declare them as public readonly static fields of the type itself. For example,

public class ShellFolder

{

private const string PROGRAM\_DATA = "ProgramData";

public static readonly ShellFolder ProgramData = new ShellFolder(PROGRAM\_DATA);

...

}

## Strings

CS/STR10 🗷 Do not use the ‘+’ operator to concatenate many strings. Instead, you should use StringBuilder for concatenation. However, do use the ‘+’ operator 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();

}

CS/STR20 🗹 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).

CS/STR30 🗹 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.

CS/STR40 🗹 Do use string operations that are based on [CultureInfo.CurrentCulture](http://msdn.microsoft.com/en-us/library/system.stringcomparison.currentculture.aspx) when you display output to the user.

CS/STR50 🗹 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.  
  
Remark:  
The ***"Ordinal"*** setting comparison looks purely at the values of the raw byte(s) that represent the character.  
Example: "0" < "9" < "A" < "Ab" < "Z" < "a" < "aB" < "ab" < "z" < "Á" < "Áb" < "á" < "áb".  
***"InvariantCulture"*** on the other hand, uses a "standard" set of character orderings. This is in contrast to some specific locales, which may sort characters in different orders.  
Example: "0" < "9" < "a" < "A" < "á" < "Á" < "ab" < "aB" < "Ab" < "áb" < "Áb" < "z" < "Z".

CS/STR60 🗹 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))

CS/STR70 🗷 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.

CS/STR80 🗹 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

CS/AC10 🗹 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.

CS/AC20 🗷 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();

}

CS/AC30 🗹 You must 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}

};

// Multidimensional arrays

int [,] multiDimArray =

{

{1, 2, 3, 4},

{5, 6, 7, 0},

{8, 0, 0, 0},

{9, 0, 0, 0}

};

CS/AC40 🗹 **Do** use ReadOnlyCollection<T> or a subclass of ReadOnlyCollection<T> for properties or return values representing read-only collections instead of Collection<T> or a subclass of Collection<T>.

CS/AC50 🗹 You should prefer to use collections from the System.Collections.Generic namespace over unparametrized collections from the System.Collections namespace. Especially avoid the usage of ArrayList or Hashtable for type-safety and performance (boxing for value types).

CS/AC60 🗹 When you are creating a collection type, you must implement IEnumerable so that the collection can be used with LINQ to Objects.

CS/AC70 🗷 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.

CS/AC80 🗷 Do not return a null reference for 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.

List<int> list = SomeOtherFunc();

foreach (var v in list)

{

...

}

## Structures

CS/ST30 🗷 Do not define mutable value types.

Mutable value types have several problems. For example, when a property getter returns a value type, the caller receives a copy. Because the copy is created implicitly, developers might not be aware that they are mutating the copy, and not the original value. Also, some languages (dynamic languages, in particular) have problems using mutable value types because even local variables, when dereferenced, cause a copy to be made.

CS/ST40 🗹 Do implement IEquatable<T> on value types. The Object.Equals method on value types causes boxing and its default implementation is not very efficient, as it uses reflection. IEquatable<T>.Equals can have much better performance and can be implemented such that it will not cause boxing.

## Structures vs. Classes

CS/SC10 🗷 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.

## Interfaces

An interface contains only the signatures of [methods](https://msdn.microsoft.com/de-de/library/ms173114.aspx), [properties](https://msdn.microsoft.com/de-de/library/x9fsa0sw.aspx), [events](https://msdn.microsoft.com/de-de/library/awbftdfh.aspx) or [indexers](https://msdn.microsoft.com/de-de/library/6x16t2tx.aspx). A class or struct that implements the interface must implement the members of the interface that are specified in the interface definition.  
So Interface exposes contracts which are consumed by clients.

CS/INT10 🗹 Do define an interface if you need some “common API” to be supported by a set of types that includes value types.   
Example: Define an interface IFruit which is implemented by types Apple and Banana.

CS/INT20 🗹 Do use interfaces to express "can-do" relationships with the interface. For example, a FileStream "can-do" disposing.

CS/INT30 🗷 You should not use marker interfaces (interfaces with no members). There are exceptions to this rule for categorizations within frameworks.

## Classes

CS/CLS10 🗹 Do use inheritance to express “is a” relationships such as “cat is an animal”. Keep in mind that "is a" means that the derived type can be used instead of the base type and obeys the contract that is defined by the base type (Liskov substitution principle) [[REF4]](#REF4).

### Fields

CS/FLD10 🗷 Do not provide instance fields that are public or protected. Public and protected fields do not version well and are not protected by code access security demands. Instead of using publicly visible fields, use private fields and expose them through properties.

CS/FLD20 🗹 Do use public static read-only fields for predefined object instances.

CS/FLD30 🗹 Do use constant fields for constants that will never change.

CS/FLD40 🗹 Do keep in mind that read-only fields for a mutable reference type just prevents the fields from pointing to a different instance but does not prevent the instance from being modified.

A mutable type is a type with instances that can be modified after they are instantiated. For example, arrays, most collections, and streams are mutable types, but simple value types, System.Uri, and System.String are all immutable. The read-only modifier on a reference type field prevents the instance stored in the field from being replaced, but it does not prevent the field’s instance data from being modified by calling members changing the instance.

### Properties

CS/PRP10 🗹 Do use a property, rather than a method, if the value of the property is stored in the process memory and the property would just provide access to the value. Properties should look and act like a field as much as possible.

CS/PRP20 🗷 Do not use a property, but a method in the following situations:

* The operation is orders of magnitude slower than a field set would be. If you are even considering providing an asynchronous version of an operation to avoid blocking the thread, it is very likely that the operation is too expensive to be a property. In particular, operations that access the network or the file system (other than once for initialization) should most likely be methods, not properties.
* The operation is a conversion, such as the Object.ToString method.
* The operation returns a different result each time it is called, even if the parameters do not change. For example, the Guid.NewGuid method returns a different value each time it is called.
* The operation has a significant and observable side effect. Note that populating an internal cache is not generally considered an observable side effect.
* The operation returns a copy of an internal state (this does not include copies of value type objects returned on the stack).
* The operation returns an array: Use a method where the operation returns an array because to preserve the internal array, you would have to return a deep copy of the array, not a reference to the array used by the property. This fact, combined with the fact that developers use properties as though they were fields, can lead to very inefficient code.

CS/PRP30 🗹 Do create read-only properties if the caller should not be able to change the value of the property. Keep in mind that if the type of the property is a mutable reference type, the property value can be changed even if the property is get-only.

CS/PRP40 🗷 Do not provide set-only properties. If the property getter cannot be provided, use a method to implement the functionality instead. The method name should begin with Set followed by what would have been the property name.

CS/PRP50 🗹 Do provide sensible default values for all properties, ensuring that the defaults do not result in a security hole or an extremely inefficient design.

CS/PRP60 🗷 Do not design a class in a way that properties needs to be set before calling a method (that internally makes use of these properties). This would expose internal implementation details to the client.

CS/PRP70 🗹 Do allow properties to be set in any order even if this results in a temporary invalid state of the object.

CS/PRP80 🗷 You should not throw exceptions from property getters. Property getters should be simple operations without any preconditions. If a getter might throw an exception, consider redesigning the property to be a method. This recommendation does not apply to indexers. Indexers can throw exceptions because of invalid arguments. It is valid and acceptable to throw exceptions from a property setter.

### Indexed Properties

CS/IP30 🗷 Do not use indexers for other purposes but exposing elements of a class that itself represents semantically a collection of these elements.

### Constructors

CS/CN10 🗹 You should providing simple, ideally default, constructors.

CS/CN20 🗹 Do minimal work in the constructor. Constructors should not do much work other than to capture the constructor parameters and set main properties. The cost of any other processing should be delayed until required.

CS/CN30 🗹 Do use constructor parameters as shortcuts for setting main properties and use the same name for constructor parameters and a property if the constructor parameters are used to simply set the property.

CS/CN40 🗹 Do throw exceptions from instance constructors if appropriate.

CS/CN50 🗹 Do explicitly declare the public default constructor in classes, if such a constructor is required. Even though some compilers automatically add a default constructor to your class, adding it explicitly makes code maintenance easier. It also ensures the default constructor remains defined even if the compiler stops emitting it because you add a constructor that takes parameters.

CS/CN60 🗷 Do not call virtual members on an object inside its constructors. Calling a virtual member causes the most-derived override to be called regardless of whether the constructor for the type that defines the most-derived override has been called.

### Type Constructors

CS/TCN10 🗹 Do make static constructors private. A static constructor, also called a class constructor, is used to initialize a type. The CLR calls the static constructor before the first instance of the type is created or any static members on that type are called. The user has no control over when the static constructor is called. If a static constructor is not private, it can be called by code other than the CLR. Depending on the operations performed in the constructor, this can cause unexpected behavior.

Note: The C# compiler forces static constructors to be private.

CS/TCN20 🗷 Do not throw exceptions from static constructors. If an exception is thrown from a type constructor, the type is not usable in the current application domain.

CS/TCN30 🗹 You must initialize static fields inline rather than explicitly using static constructors, because the runtime is able to optimize the performance of types that don’t have an explicitly defined static constructor. In this case the order of the initialization of the static fields must not be important.

### Methods and Parameters

CS/MP10 🗹 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.

CS/MP20 🗹 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.

CS/MP40 🗹 Do use the least derived parameter type that provides the functionality required by the member.

For example, suppose you want to design a method that enumerates a collection. Such a method should take IEnumeralbe<T> as the parameter, not List<T> or T[] for example.

CS/MP50 🗹 Do be consistent in naming parameters when overriding members or implementing interface members.

CS/MP70 🗷 You should not use out or ref parameters when possible.

CS/MP80 🗷 Do not use out parameters in generic interface or base classes that may need to support covariance.

### Extension Methods

CS/EM10 🗹 You can define extension methods in any of the following scenarios:

* To provide helper functionality relevant to every implementation of an interface, if said functionality can be written in terms of the core interface. This is because concrete implementations cannot otherwise be assigned to interfaces. For example, the LINQ to Objects operators are implemented as extension methods for all IEnumerable<> types. Thus, any IEnumerable<> implementation is automatically LINQ-enabled.
* When an instance method would introduce a dependency on some type, but such a dependency would break dependency management rules. For example, a dependency from String to System.Uri is probably not desirable, and so String.ToUri() instance method returning System.Uri would be the wrong design from a dependency management perspective. A static extension method Uri.ToUri(this string str) returning System.Uri would be a much better design.

### Operator Overloads

CS/OO10 🗷 You should not defining operator overloads, except in types that should feel like primitive (built-in) types.

CS/OO20 🗹 Do define operator overloads in structs that represent numbers.

CS/OO30 🗹 Do be careful when overloading operator ==. The semantics of the operator need to be compatible with several other members, such as Object.Equals.

CS/OO40 🗷 Do not provide conversion operators if such conversion is not clearly expected by the end users.

CS/OO50 🗷 Do not provide an implicit conversion operator if the conversion is potentially lossy.

CS/OO60 🗷 Do not throw exceptions from implicit casts.

### Events and Callbacks

CS/EC10 🗹 You can use callbacks to allow users to provide custom code to be executed.

CS/EC30 🗹 Do use the Func<...>, Action<...>, or Expression<...> types instead of custom delegates, when defining APIs with callbacks.

CS/EC40 🗹 Do use System.EventHandler<TEventArgs> instead of manually creating new delegates to be used as event handlers.

CS/EC50 🗹 Do be prepared for arbitrary code executing in the event-handling method. One example here is that you should not hold a lock when raising an event. The handler may itself acquire a lock. This may reverse the normal lock hierarchy between the parties which in turn may result in a deadlock.

CS/EC60 🗷 Do not pass null as the event data parameter when raising an event. You should pass EventArgs.Empty if you don’t want to pass any data to the event handling method.

CS/EC70 🗹 Do understand that by calling a delegate or event, you are executing arbitrary code and that could have security, correctness, and compatibility repercussions.

### Member Overloading

CS/MO10 🗹 You should 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 way to avoid default arguments:

public void Rotate(Matrix matrix)

{

    Rotate(matrix, 180);

}

public void Rotate(Matrix matrix, int degrees)

{

    // Do rotation here

}

Solution with default arguments (which shall be avoided in this case):

public void Rotate(Matrix matrix, int degrees = 180)

{

    // Do rotation here

}

CS/MO15 🗹 You can use default arguments when it makes sense, e.g. to keep interfaces small by avoiding a list of overloaded methods. But as mentioned above, keep the version issue in mind, especially for public interfaces. So a good practice is, **never change default arguments**, after first release.

Sample for default arguments:

public void Redirect(string url, string protocol = "http", bool permanent = false);

CS/MO20 🗷 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.

CS/MO30 🗹 Do make only the longest overload virtual (if extensibility is required). Shorter overloads should simply call through to a longer overload.

CS/MO40 🗷 Do not use ref or out modifiers to overload members. Some languages cannot resolve calls to overloads like this. In addition, such overloads usually have completely different semantics and probably should not be overloads but two separate methods instead.

### Interface Members

CS/IM10 🗷 You should not implement interface members explicitly without having a strong reason to do so. Explicitly implemented members can be confusing to developers because they don’t appear in the list of public members and they can also cause unnecessary boxing of value types.

CS/IM20 🗹 You should implement interface members explicitly, if the members are intended to be called only through the interface.

CS/IM30 🗹 You should implement interface members explicitly to hide a member and add an equivalent member with a better name.

public class FileStream : IDisposable

{

    void IDisposable.Dispose()

    {

        Close();

    }

    void Close()

    {

        // implementation here

    }

}

### Virtual Members

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

CS/VM10 🗷 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.

CS/VM20 🗹 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 Classes

CS/SCL10 🗹 Do prefer non-static classes over static classes. Static classes should be used only as supporting classes for the non-static counterparts (e.g. to add extension methods to that class).

### Abstract Classes

CS/ACL10 🗷 Do not define public or protected-internal constructors in abstract classes.   
Instances of abstract classes may not be created directly. You need to derive your own implementation. So the constructor of abstract classes is an implementation detail and should not be part of the public or protected-internal contract.

CS/ACL20 🗹 Do define a protected or an internal constructor on abstract classes.   
The constructor of abstract classes is called from derived class constructors and does some initialization work of the abstract class members.

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()

{

...

}

}

### Nested types

CS/NT10 🗹 Do use nested types when the relationship between the nested type and its outer type is such that member-accessibility semantics are desirable.  
Example:

public class SomeCollection

{

    private class InnerClass : ISomeInterface

    {

        private readonly int \_index;

        private readonly SomeCollection \_outerClass;

        public InnerClass(int index, SomeCollection outerClass)

        {

            \_index = index;

            \_outerClass = outerClass;

        }

        public void DoSomething()

        {

            \_outerClass.DoSomething(\_index);

        }

    }

    private void DoSomething(int index)

    {

        /// do something based on given index

        Console.WriteLine("Do something on {0}", index);

    }

    public ISomeInterface this[int index]

    {

        get

        {

            // in real code cached

            return new InnerClass(index, this);

        }

    }

}

CS/NT20 🗷 Do not define publicly exposed nested types. The only exception to this is if variables of the nested type need to be declared only in rare scenarios such as subclassing or other advanced customization scenarios.

CS/NT30 🗷 Do not use nested types if the type is likely to be referenced outside of the containing type.

## Errors and Exceptions

### Exception Throwing

CS/ET10 🗹 Do report execution failures by throwing exceptions. Exceptions are the primary means of reporting errors. If a member cannot successfully do what it is designed to do (what the member name implies), it should be considered an execution failure and an exception should be thrown.

CS/ET12 🗷 Do not return error codes.

CS/ET15 🗹 Do throw existing exceptions residing in the System namespaces instead of creating custom exception types, when it matches the appropriate pattern, e.g. ObjectDisposedException.

CS/ET20 🗹 Do create and throw custom exceptions if you have an error condition that can be programmatically handled in a different way than any other existing exceptions. Otherwise, throw one of the existing exceptions (see next chapter).

CS/ET25 🗹 You should wrap specific exceptions thrown from a lower layer in a more appropriate exception, if the lower layer exception does not make sense in the context of the higher-layer operation. Do specify the inner exception when wrapping exceptions.

CS/ET30 🗷 Do not catch and wrap non-specific exceptions.

CS/ET35 🗹 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.

CS/ET40 🗷 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,

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

{

collection.Add(additionalNumber);

}

CS/ET45 🗷 Do not explicitly throw exceptions from finally blocks. Implicitly thrown exceptions resulting from calling methods that throw are acceptable.

CS/ET50 🗹 Do consider the performance implications of throwing exceptions. Throw rates above 100 per second are likely to noticeably impact the performance of most applications.

CS/ET55 🗹 You should use the Tester-Doer Pattern or Try-Parse Pattern for members that might throw exceptions in common scenarios to avoid performance problems related to exceptions.

ICollection<int> numbers = ...

...

if(!numbers.IsReadOnly)

{

    numbers.Add(1);

}

public struct DateTime

{

    public static DateTime Parse(string dateTime)

    {

        //...

    }

    public static bool TryParse(string dateTime, out DateTime result)

    {

        //...

    }

}

CS/ET60 🗹 Do document all exceptions thrown by publicly callable members because of a violation of the member contract (rather than a system failure) and treat them as part of your contract.

CS/ET65 🗷 Do not throw System.Exception or System.SystemException.

CS/ET70 🗷 Do not throw or derive from ApplicationException.

CS/ET75 🗹 Do throw an InvalidOperationException if the object is in an inappropriate state.

CS/ET80 🗹 Do throw ArgumentException or one of its subtypes if bad arguments are passed to a member. Prefer the most derived exception type, if applicable.

CS/ET85 🗹 Do allow publicly callable APIs to explicitly or implicitly throw NullReferenceException, AccessViolationException or IndexOutOfRangeException. These exceptions are reserved and thrown by the execution engine and in most cases indicate a bug.

CS/ET90 🗷 Do not throw StackOverflowException, OutOfMemoryException, COMException, ExecutionEngineException, and SEHException. These exception are to be thrown only by the CLR infrastructure

CS/ET95 🗹 Do provide a rich and meaningful message text targeted at the developer when throwing an exception. The message should explain the cause of the exception and clearly describe what needs to be done to avoid the exception. Avoid question marks ("?") and exclamation points ("!") in exception messages.

CS/ET100 🗷 Do not disclose security-sensitive information in exception messages without demanding appropriate permissions.

### Custom Exceptions

CS/CE10 🗹 Do derive exceptions from ErrorCodeException (Foundations.Core). This class is the base class for all project specific exceptions and represents errors that occur during application execution. ErrorCodeException provides an error code which allows localizing of error texts.

CS/CE20 🗹 Do end exception class names with the Exception suffix.

CS/CE50 🗹 You can provide exception properties for programmatic access to extra information (besides the message string) relevant to the exception.

### Exception Handling

CS/EH10 🗷 Do not swallow errors by catching nonspecific exceptions, such as System.Exception, System.SystemException, and so on in .NET code. Do catch only specific errors that the code knows how to handle. All other exceptions are to handle on application defined boundaries that have a general exception handling mechanism.  
There are cases when swallowing errors in applications is acceptable, but such cases are rare.

Good:

try

{

...

}

catch(System.NullReferenceException exc)

{

...

}

catch(System.ArgumentOutOfRangeException exc)

{

...

}

catch(System.InvalidCastException exc)

{

...

}

Bad:

try

{

...

}

catch (Exception ex)

{

...

}

CS/EH20 🗹 Do use an empty throw when catching and re-throwing an exception. This is the best way to preserve the exception call stack.

Good:

try

{

... // Do some reading with the file

}

catch

{

file.Position = position; // Unwind on failure

throw; // Rethrow

}

Bad:

try

{

... // Do some reading with the file

}

catch (Exception ex)

{

file.Position = position; // Unwind on failure

throw ex; // Rethrow

}

## Resource Cleanup

CS/RC10 🗷 Do not force garbage collections with GC.Collect.

### Try-finally Block

CS/TFB10 🗹 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,

FileStream stream = null;

try

{

stream = new FileStream(...);

...

}

finally

{

if (stream != null)

{

stream.Close();

}

}

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

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

{

...

}

Many language constructs emit try-finally blocks automatically for you. Examples are the using statement, the lock statement and the foreach statement.

### Basic Dispose Pattern

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:

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;

}

}

CS/BDP10 🗹 Do implement the Basic Dispose Pattern on types containing instances as members of disposable types.

CS/BDP20 🗹 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.

CS/BDP30 🗹 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.

CS/BDP40 🗹 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.

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;

}

CS/BDP50 🗹 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.

public void Dispose()

{

Dispose(true);

GC.SuppressFinalize(this);

}

CS/BDP60 🗷 Do not make the parameterless Dispose method virtual. The Dispose(bool) method is the one that should be overridden by subclasses.

CS/BDP70 🗷 Do 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.

CS/BDP80 🗹 Do throw an ObjectDisposedException from any member that cannot be used after the object has been disposed.

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;

}

}

### Finalizable Types

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:

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;

}

}

CS/FT10 🗹 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.

public class ComplexResourceHolder : IDisposable

{

...

~ComplexResourceHolder()

{

Dispose(false);

}

protected virtual void Dispose(bool disposing)

{

...

}

}

CS/FT20 🗹 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.

CS/FT30 🗹 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.

CS/FT40 🗹 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.

CS/FT50 🗹 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);

}

}

CS/FT60 🗷 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.

CS/FT70 🗷 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

CS/OD10 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);

}

}

}

## Interop

### P/Invoke

CS/PIV10 🗹 Do consult [P/Invoke Interop Assistant](http://clrinterop.codeplex.com/) and <http://pinvoke.net> to write P/Invoke signatures.

CS/PIV20 🗹 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.

CS/PIV30 🗷 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>.

CS/PIV40 🗹 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);

CS/PIV50 🗹 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);

}

}

CS/PIV60 🗹 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;

}

}

### COM Interop

CS/COM10 🗷 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.

CS/COM20 🗹 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.

CS/COM30 🗷 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.

## LINQ – Language INtegrated Query

### Support of LINQ

There are three means by which a type can be designed to participate in LINQ queries: implementing IEnumerable<> (or interfaces derived from it), implementing IQueryable<>, or by defining the Query Pattern on the type, with no relationship to these two interfaces.

The choice can be made as follows:

* Extend IEnumerable<> (or interfaced derived from it) if the default, LINQ to Objects support provided in the framework is sufficient. Override some/all LINQ methods if optimizations are required, but access to the query expression is not necessary.
* Extend IQueryable<> if access to the query expression is necessary.
* Define the Query Pattern on the new type if the domain makes it impossible or undesirable to implement IEnumerable<> or IQueryable<>.

The query methods are those defined by the sponsor class System.Linq.Enumerable, as extension methods to IEnumerable<>. Regardless of the means by which a type participates in LINQ queries, the following guideline applies:

CS/LNQ10 🗹 Do respect the following signature pattern when overriding LINQ methods.

// S<T> indicates a collection type (e.g. IEnumerable<>, ICollection<>)

// O<T> represents subtypes of S<T> that are ordered

// Func<> is being used, one may substitute accordingly with Expression<Func<>>

S<TSource> Where<TSource>(this S<TSource> source, Func<TSource, bool> predicate);

S<TResult> Select<TSource, TResult>(this S<TSource> source, Func<TSource, TResult> selector);

// TKey is IComparable

O<TSource> OrderBy<TSource, TKey>(this S<TSource> source, Func<TSource, TKey> keySelector);

// TKey is IComparable

O<TSource> ThenBy<TSource, TKey>(this S<TSource> source, Func<TSource, TKey> keySelector);

S<TSource> Union<TSource>(this S<TSource> first, S<TSource> second);

S<TSource> Skip<TSource>(this S<TSource> source, int count);

S<TSource> SkipWhile<TSource>(this S<TSource> source, Func<TSource, bool> predicate);

S<TSource> Take<TSource>(this S<TSource> source, int count);

TSource ElementAt<TSource>(this S<TSource> source, int index);

S<TResult> Join<TOuter, TInner, TKey, TResult>(this S<TOuter> outer,

                                                S<TInner> inner,

                                                Func<TOuter, TKey> outerKeySelector,

                                                Func<TInner, TKey> innerKeySelector,

                                                Func<TOuter, TInner, TResult> resultSelector);

S<TResult> SelectMany<TSource, TResult>(this S<TSource> source, Func<TSource, S<TResult>> selector);

S<TResult> SelectMany<TSource, TCollection, TResult>

(this S<TSource> source,

                                           Func<TSource, S<TCollection>> collectionSelector,

                                           Func<TSource, TCollection, TResult> resultSelector);

#### Extending IEnumerable<T>

CS/EIN20 🗹 You should redefine the methods in the LINQ pattern on new types implementing IEnumerable<T>, if it is desirable to override the default implementation – e.g. for optimization. It is preferred that one redefines them as type members, rather than extension methods.

CS/EIN30 🗹 You should implement ICollection<T> to improve performance of query operators.

For example, the Count<> method default behavior (as defined in System.Linq) is to simply walk the IEnumerable. Collection types can optimize their implementation of this method, since they typically offer an O(1) - complexity mechanism for finding the size of the collection.

#### Extending IQueryable<T>

CS/EIQ10 🗹 You can implement IQueryable<T> when access to the query expression is necessary.

CS/EIQ20 🗹 Do throw NotSupportedException from IQueryable<T> methods that cannot be logically supported by your data source.

#### Implementing the Query Pattern

The Query Pattern refers to defining the methods according to the signature pattern without implementing the IEnumerable<> or IQueryable<> interfaces.

CS/IQP10 🗹 Do implement at least the enumerable pattern (i.e. provide a GetEnumerator method) for types representing collections of data. This is because the expectation is that the result of the query methods be foreach-able.

CS/IQP20 🗹 Do implement the Query Pattern as instance members on the new type, if the members make sense on the type even outside of the context of LINQ. Otherwise, use extension methods.

CS/IQP30 🗹 Do represent ordered sequences as a separate type. Define on this type the “ThenBy” method.

This follows the current pattern in the LINQ-to objects implementation, as well as allows for early (compile-time) detection of errors such as applying “ThenBy” to a not “OrderBy”-ed sequence.

For example, the framework provides the IOrderedEnumerable<> type, which is returned by “OrderBy”. The “ThenBy” extension method is defined for this type, and not for IEnumerable<>.

CS/IQP40 🗹 You should design the LINQ operators to return specific enumerable types for domains that should be restricted. Essentially, one is free to return anything from a Select query method, however, the expectation is that the query result type should be at least enumerable in a foreach loop.

CS/IQP50 🗹 Do defer execution of query operator implementations. The expected behavior of most of the Query Pattern members is that they simply construct a new object which, upon enumeration, produces the elements of the set which match the query. The evaluation time is, then, at enumeration.

The following methods are exception from this rule: All, Any, Average, Contains, Count, ElementAt, Empty, First, FirstOrDefault, Last, LastOrDefault, Max, Min, Single, Sum

CS/IQP60 🗷 You should not implement just a part of the Query Pattern, if fallback to the basic IEnumerable<T> implementations is undesirable.

For example, consider a user-defined type T, which implements IEnumerable<>. T has an override for Count, but not for Where. Consider then the following example

var query=someList.Where(x => x > 100).Count();

In this example, any opportunities for optimization are lost after the Where call. The Count version used is the one defined for IEnumerable<>.

CS/IQP70 🗹 Do place query extensions methods in a “Linq” sub-namespace of the main namespace. For example, extension methods for System.Data features reside in System.Data.Linq namespace.

CS/IQP80 🗹 Do use Expression<Func<>> as a parameter instead of Func<> when it is necessary to inspect the query.

### LINQ Usage

CS/LU10 🗹 Do use meaningful names for query variables. The following example uses seattleCustomers for customers who are located in Seattle.

var seattleCustomers =  
   from customer in customers

    where customer.City == "Seattle"

    select customer.Name;

CS/LU20 🗹 Do use aliases to make sure that property names of anonymous types are correctly capitalized, using Pascal casing.

var localDistributors =

    from customer in customers

    join distributor in distributors on customer.City equals distributor.City

    select new { Customer = customer, Distributor = distributor };

CS/LU40 🗹 Do align query clauses under the from clause, as shown in the previous examples.

CS/LU50 🗹 Do use where clauses before other query clauses to ensure that later query clauses operate on the reduced, filtered set of data.

var seattleCustomers =   
 from customer in customers

    where customer.City == "Seattle"

    orderby customer.Name

    select customer.Name;

CS/LU60 🗹 Do use multiple from clauses instead of a join clause to access inner collections.

var scoreQuery =  
  from student in students

    from score in student.Scores

    where score > 90

    select new { Last = student.LastName, score };

CS/LU70 🗹 Do return a query when the caller expects a query.

CS/LU80 🗹 Do return a list when the caller expects a list.

When you design your method, decide what the caller is more likely to want, implement that, and then document it.

CS/LU90 🗹 Do keep in mind that a LINQ-query is evaluated lazily. This is especially important at boundaries of components. For example a component that returns IEnumerable<T> and implements it internally by a LINQ-query may need to use ToList() to enforce query evaluation to provide a consistent state to the caller.

CS/LU100 🗹 Do use Any() to determine whether an IEnumerable<T> is empty.

## Asynchronous Programming

Three different patterns for asynchronous programming on the .NET platform are defined:

* Asynchronous Programming Model (APM) pattern (also called the [IAsyncResult](http://msdn.microsoft.com/en-us/library/system.iasyncresult(v=vs.110).aspx) pattern), where asynchronous operations require Begin and End methods (for example, BeginWrite and EndWrite for asynchronous write operations). This pattern is no longer recommended for new development. For more information, see [Asynchronous Programming Model (APM)](http://msdn.microsoft.com/en-us/library/ms228963(v=vs.110).aspx).
* Event-based Asynchronous Pattern (EAP), which requires a method that has the Async suffix, and also requires one or more events, event handler delegate types, and EventArg-derived types. EAP was introduced in the .NET Framework 2.0. It is no longer recommended for new development. For more information, see [Event-based Asynchronous Pattern (EAP)](http://msdn.microsoft.com/en-us/library/ms228969(v=vs.110).aspx).
* Task-based Asynchronous Pattern (TAP), which uses a single method to represent the initiation and completion of an asynchronous operation. TAP was introduced in the .NET Framework 4 and is the recommended approach to asynchronous programming in the .NET Framework. For more information, see [Task-based Asynchronous Pattern (TAP)](http://msdn.microsoft.com/en-us/library/hh873175(v=vs.110).aspx). With the [Async and Await](http://msdn.microsoft.com/en-us/library/hh191443.aspx) feature of C#5 this pattern is even more convenient to use because the compiler supports asynchronous programming by retaining a logical structure that resembles synchronous code.

The following code shows a quick comparison of the different patterns:

// Synchronous

public class MyClass

{

    public int Read(byte[] buffer, int offset, int count);

}

// APM

public class MyClass

{

    public IAsyncResult BeginRead(

        byte[] buffer, int offset, int count,

AsyncCallback callback, object state);

    public int EndRead(IAsyncResult asyncResult);

}

// EAP

public class MyClass

{

    public void ReadAsync(byte[] buffer, int offset, int count);

    public event ReadCompletedEventHandler ReadCompleted;

}

// TAP

public class MyClass

{

    public Task<int> ReadAsync(byte[] buffer, int offset, int count);

}

As the former ones are not recommended anymore. Only guidelines for the TAP patterns are provided here:

CS/AP10 🗹 Do use async Task-returning methods according to the TAP pattern.

CS/AP20 🗹 Do use an Async suffix after the operation name for a TAP method.

CS/AP30 🗹 Do return Task or Task<TResult> based on whether the corresponding synchronous method returns void or a type TResult.

CS/AP40 🗹 Do use the same parameters and orders if a synchronous variant of the method exists.

CS/AP50 🗷 Do not use out or ref parameters in asynchronous methods. Any data that would have been returned through an out or ref parameter should instead be returned as part of the TResult returned by Task<TResult> and should use a tuple or a custom data structure to accommodate multiple values.

CS/AP60 🗷 Do not raise an exception to be thrown out of the asynchronous method call in an asynchronous method apart from a usage error (e.g. called with wrong parameters). For all other errors, exceptions that occur when an asynchronous method is running should be assigned to the returned task, even if the asynchronous method happens to complete synchronously before the task is returned.

Note: By using the async keyword the compiler takes care that this rule is obeyed. Any exceptions that go unhandled within the body of the method are marshaled to the output task and cause the resulting task to end in the Faulted state.

Every Task will store a list of exceptions. When you await a Task, the first exception is re-thrown, so you can catch the specific exception type (such as InvalidOperationException). However, when you synchronously block on a Task using Task.Wait or Task.Result, all of the exceptions are wrapped in an AggregateException and thrown.

CS/AP70 🗷 Do not return a Task from TAP methods that is not activated (i.e. that is in the Created State).

CS/AP80 🗹 Do expose an overload of the asynchronous method that accepts a cancellation token (CancellationToken instance) when supporting cancellation of the operation. Name this parameter cancellationToken.

CS/AP90 🗷 Do not use async void except for top-level event handlers.

Async void is a "fire-and-forget" mechanism. The caller is unable to know when the async void has finished and is unable to catch exceptions.

Bad:

private async void ButtonClick(object sender, EventArgs e)

{

    try

    {

// SendData is async void and can therefore not be awaited

        SendData("http://www.google.de");

        await Task.Delay(2000);

    }

    catch (Exception)

    {

        // Exceptions of SendData can't be handled this way as there is no task

    }

}

private async void SendData(string url)

{

// ...

}

Good:

private async void ButtonClick(object sender, EventArgs e)

 {

     try

     {

         m\_response = await SendDataAsync("http://www.google.de");

     }

     catch (Exception)

     {

         // ...

     }

 }

 private async Task<string> SendDataAsync(string url)

 {

// ...

 }

CS/AP100 🗹 Do verify async lambda to be not async void. In C# the same async lambda represents an async void and async task:

Action a1 = async () => { await LoadAsync(); \_result = "done"; };

Func<Task> a2 = async () => { await LoadAsync(); \_result = "done"; };

It therefore depends on the context which one is chosen by the compiler:

// Which overload of Task.Run is chosen by the compiler ?

// If both overloads are available the compiler chooses the Task-returning one.

await Task.Run(async () => { await LoadAsync(); \_result = "done"; });

class Task

{

     static public Task Run(Action a);

     static public Task Run(Func<Task> a);

}

CS/AP110 🗹 Do use the await rather than background threads (e.g. from the threadpool) for IO-bound code. Use background threads only for CPU-bound code.

Bad:

public List<Customer> LoadCustomersInParallel(int first, int last)

{

    var loadedCustomers = new BlockingCollection<Customer>();

    Parallel.For(first, last + 1, i =>

    {

        // The access of Result blocks the thread

        Customer house = LoadCustomerAsync(i).Result;

        loadedCustomers.Add(house);

    });

    return loadedCustomers.ToList();

}

Good:

public async Task<List<Customer>> LoadHousesAsync(int first, int last)

{

    var tasks = new List<Task<Customer>>();

    for (int i = first; i <= last; i++)

    {

        Task<Customer> t = LoadCustomerAsync(i);

        tasks.Add(t);

    }

    Customer[] loadedHouses = await Task.WhenAll(tasks);

    return loadedHouses.ToList();

}

CS/AP120 🗹 You should consider adding throttling for async operations that can be initiated a huge number of times to protect your resources. Examples for throttling: limit the number for concurrent incoming or outgoing web requests or the number of concurrent database operations.

CS/AP130 🗷 You should not expose an async API if your implementation is not truly async. Don't "fake" it through use of Task.Run internally. Async method signatures implies that work is initiated and completed later. The caller expects that it is not using the threadpool or CPU.

CS/AP140 🗹 You should provide synchronous methods when you do CPU-bound work (e.g. 3D rendering) that blocks the current thread. For that purpose you can also split the work on several threads, but not more than cores are available. Please keep in mind, that the UI must be responsive.

CS/AP150 🗹 You should provide asynchronous methods when you can do so without spawning new threads.

CS/AP160 🗹 Do keep in mind that await will continue the work in the synchronization context in which it was started.

"Await task" uses the current synchronization context:

* It captures the current SyncContext before awaiting.
* Upon task completion, it calls SyncContext.Post() to resume “where you were before”

CS/AP170 🗹 You should use ConfigureAwait for Tasks returned by components that mustn't be continued on the current synchronization context. This is especially important for components that can be called in different scenarios.

CS/AP180 🗷 Do not use Task.Wait in single-threaded synchronization contexts. Use the await operator instead.

In these single-threaded synchronization contexts, it’s easy to deadlock yourself. If you spawn off a task from a single-threaded context, then wait for that task in the context, your waiting code may be blocking the background task.

C++ Coding Standards

These coding standards are applied to native C++. As a general rule the following applies:

CPP/10 🗹 Do write Modern Style C++ code using the C++ 11 standard.

The features of C++11 that are available in Visual C++ should be used whenever they are appropriate. Details about the new features can be found [here](http://www.stroustrup.com/C++11FAQ.html) [[REF6]](#REF6). The currently supported features of C++11 in Visual C++ can be found [here](http://msdn.microsoft.com/en-us/library/hh567368.aspx).

“C++11 feels like a new language.” – Bjarne Stroustrup. Some of the new language features changes the way C++ programs are written significantly: For example, classes that owns a big resource can now be returned by value without any efficiency penalty when the class implements the new "move"-semantics. This changes the way interfaces of classes are designed to be much simpler and natural:

CPP/20 🗹 You should use the benefits of move semantics to provide simpler and more natural interfaces.  
Example:

// C++98: alternatives to avoid copying

// option 1: return by pointer: no copy, but don't forget to delete

vector<int>\* make\_big\_vector();

// usage option 1

vector<int>\* result = make\_big\_vector();

// option 2: pass out by reference: no copy,

// but caller needs a named object

void make\_big\_vector(vector<int>& out);

// usage option 2

vector<int> result;

make\_big\_vector(result);

// C++11: move

// usually sufficient for 'callee-allocated out' situations

vector<int> make\_big\_vector();

// guaranteed not to copy the vector but instead use move semantics

auto result = make\_big\_vector();

CPP/30 🗹 Do use auto for variable declaration.The auto keyword instructs the compiler to infer the type from the expression. This is especially helpful if the type of an expression is hard to know or to write. It can also make the code easier to read:

// C++11

template<class T> void printall(const vector<T>& v)

{

    for (auto p = v.begin(); p != v.end(); ++p)

        cout << \*p << "\n";

}

//C++98

template<class T> void printall(const vector<T>& v)

{

    for (typename vector<T>::const\_iterator p = v.begin(); p != v.end(); ++p)

        cout << \*p << "\n";

}

CPP/40 🗹 You should prefer to use Lambdas over Functors when they have a simple body.  
Example:

Bad:

vector<Record> v = ...

vector<int> indices(v.size());

indices = ... // generate indices of v

struct Cmp\_names

{

    const vector<Record>& vr;

    Cmp\_names(const vector<Record>& r) :vr(r) { }

    bool operator()(int a, int b) const

    { return vr[a].name < vr[b].name; }

};

// sort indices in the order determined by the name field of the records

std::sort(indices.begin(), indices.end(), Cmp\_names(v));

Good:

vector<Record> v = ...

vector<int> indices(v.size());

indices = ... // generate indices of v

// sort indices in the order determined by the name field of the records:

std::sort(indices.begin(), indices.end(),

       [&](int a, int b) { return v[a].name < v[b].name; });

CPP/50 🗹 You should prefer to use Uniform Initialization and Initializer Lists over the conventional ( )-parentheses usage. When initializing a local variable whose type is non-POD or auto, continue using the familiar = syntax without extra { } braces:

// C++98 or C++11

int a = 42;

// C++ 11

// no narrowing or non-initialization is possible

auto x = begin(v);

In other cases, including everywhere that you would have used ( ) parentheses when constructing an object, prefer using { } braces instead. Using braces avoids several potential problems:

* No accidental narrowing conversions (e.g., float to int).
* No accidentally have uninitialized POD member variables or arrays.
* No C++98 surprise that your code compiles but actually declares a function rather than a variable.

// C++98

std::complex<double> c(2.71828, 3.14159);

int             a[] = { 1, 2, 3, 4 };

vector<int>      v;

for (int i = 1; i <= 4; ++i)

    v.push\_back(i);

// C++11

std::complex<double> c{ 2.71828, 3.14159 };

int              a[] { 1, 2, 3, 4 };

vector<int>      v{ 1, 2, 3, 4 };

CPP/60 🗷 You should not overuse initializer lists. The main purpose should be array/sequence and simple class initialization (as the complex number in the previous example).  
Example:

class Bar

{

private:

    //...

public:

    Bar(int i, double f)

        //...

    {}

};

class Foo

{

private:

    //...

public:

    Foo(const Bar& bar, int v)

        //...

    {}

};

Bad:

// compiles but is not easy to understand

// without the class definitions in mind

Foo foo = { { 1, 1.0 }, 1 };

Writing modern style C++ also means that features of the standard library should be used whenever appropriate:

CPP/70 🗹 Do prefer to use features of the standard library over custom libraries of a specific platform if that specific library does not provide a significant added-value.

## Compiler Options

### Enable All Warnings, and Treat Them as Errors

CPP/CO10 🗹 You should compile all code at the highest warning level.

CPP/CO20 🗹 You should treat all warnings as errors.

The warnings provided by the compiler are often useful in identifying bad practices, or even subtle bugs. You can use the compiler warnings as an extra level of verification on your code.

In Visual Studio you can enable warning level four in the properties for you project; on the ‘Property Pages’ for your project, go to “Configuration Properties”, “C/C++”, “General” and set “Warning Level” to “Level 4”.

CPP/CO30 🗹 Do prefer compile- and link-time errors to run-time errors.

Use the compiler features to enforce invariants of the program. This often means to use the type-checks inherently available in a statically-typed language.

Examples:

* Use compile-time Boolean conditions to express assumptions of the code:
* static\_assert (sizeof(int) >= 8, "Error message")
* Use compile-time polymorphism: Consider using templates instead of virtual functions.
* Use enums for symbolic constants
* Avoid using dynamic\_cast

## Files and Structure

### Header Files

CPP/HF10 🗹 Do use include guards within a header file (internal include guards) to prevent unintended multiple inclusions of the header file.

The #ifndef and #endif from the example, below, should be the first and last lines of the header file. The following example shows how to use “#ifndef/#endif” as an include guard in “CodeExample.h”;

// File header comment goes first ...

#ifndef CODE\_EXAMPLE\_H\_

#define CODE\_EXAMPLE\_H\_

class CodeExample

{

...

};

#endif //CODE\_EXAMPLE\_

Remark: Include guards must be unique. Therefore a project-specific convention must be used that ensures the uniqueness.

CPP/HF20 🗹 You can instead use “#pragma once” as an alternative to “#ifndef/#endif” include guard. It is non-standard but it is widely supported by different compilers (Microsoft Visual C++, GCC, Intel C++, Clang and others):

// File header comment goes first ...

#pragma once

class CodeExample

{

...

};

CPP/HF30 🗹 You should use the following sequence in your header file:

* Head comments
* Include guard (see. previous item)
* System includes section: Includes of files, which are base for development (operating system includes, UI class library includes, ...). For Visual C++ development, Windows specific include files are included via stdafx.h, which realizes the “precompiled header files” mechanism.
* Application includes: The files which are part of the project are included.
* The #defines section defines constants and macros, if applicable.
* Section with declaration of global variables.
* Function prototypes (ANSI-C).
* Class declaration.

CPP/HF40 🗷 You should not implement functions in header files. Header files should only contain the declarations of functions and data structures. Their implementation should be in the .cpp files. Exceptions to this are:

* Templates have to be implemented as header files only.
* Small functions can be implemented in the header file to allow inlining.
* For header-only implementations (class implementations completely in the header file) the file must have the “.hpp” file extension.

CPP/HF50 🗷 Do not use absolute path names in #includes.

CPP/HF60 🗷 Do not include a header file when a forward declaration is sufficient. Don't include headers you don't need at all.

CPP/HF70 🗹 Do make header files self-sufficient. Ensure that every header includes all dependent headers it needs.

### Implementation Files

Implementation files contain the actual function bodies for global functions, local functions, and class method functions. An implementation file has the extension .c or .cpp. Note that an implementation file does not have to implement an entire module. It can be split up and #include a common internal interface.

CPP/IF10 🗹 You should use the following sequence in your implementation file:

Head comments

Include section: In #include Section in general first the header file, which corresponds with the implementation file, is listed.

CPP/IF20 🗹 You should keep declarations that don’t have to be exported inside the implementation file. Furthermore, you should add the static keyword to limit their scope to just the compilation unit defined by the .cpp/.c file. This will reduce changes of getting “multiply-defined symbol” errors during linking when two or more .cpp files use the same internal variables.

### Class layout

CPP/CL10 🗹 You should use the following class layout rules:

Order of declaration:

First declaration of attributes and then declaration of methods

Further declarations are listed in following order:

1. public methods/attributes
2. protected methods/attributes
3. private methods/attributes
4. friend declarations

Methods can be declared in logical blocks:

* Constructor(s)
* Destructor(s)
* Operator(s) (Assignment, Comparison, etc.)
* Write and read operation(s) Set...() and Get...(), arranged as corresponding pairs.

Example:

class Foo : public Bar

{

protected:

    // The important counter

    int m\_count;

private:

    // Some secrets

    int m\_secretNumber;

public:

    Foo();

    ~Foo();

    ...

protected:

    ...

private:

    ...

};

### Regions

CPP/REG10 🗹 You can use region declarations where there is a real benefit from this organization. E.g. grouping the large amount of code by scope or functionality improves readability and structure of the code.

C++ regions:

#pragma region "Helper Functions for XX"

...

#pragma endregion

## Naming Conventions

### General Naming Conventions

CPP/GNC10 🗹 Do use meaningful names for various types, functions, variables, constructs and data structures. Their use should be plainly discernible from their name alone.

Single-character variables should only be used as counters (i, j) or as coordinates (x, y, z). As a rule-of-thumb a variable should have a more descriptive name as its scope increases.

CPP/GNC20 🗷 You should not use shortenings or contractions as parts of identifier names. For example, use “GetWindow” rather than “GetWin”. For functions of common types, thread procs, window procedures, dialog procedures use the common suffixes for these “ThreadProc”, “DialogProc”, “WndProc”.

### Capitalization Naming Rules for Identifiers

The following table describes the capitalization and naming rules for different types of identifiers.

|  |  |  |  |
| --- | --- | --- | --- |
| Identifier | Casing | Naming Structure | Example |
| Class | PascalCasing | Noun | class ComplexNumber {...};  class CodeExample {...};  class StringList {...}; |
| Enumeration | PascalCasing | Noun | Enum class Type {...}; |
| Function, Method | PascalCasing | Verb or  Verb-Noun | void Print()  void ProcessItem()  🗹 Do name selectors with a Get-prefix: void Get<Value>()  🗹 Do name modifiers with a Set-prefix:  void Set<Value>()  🗹 Do name verifiers with a Is-prefix.  void Is<Criteria>() |
| Interface | PascalCasing  ‘I’ prefix | Noun | class IDictionary {...}; |
| Structure | All capital, separate words with ‘\_’ | Noun | struct FORM\_STREAM\_HEADER |
| Macro,  Constant | All capital, separate words with ‘\_’ |  | #define BEGIN\_MACRO\_TABLE(name) ...  #define MACRO\_TABLE\_ENTRY(a, b, c) ...  #define END\_MACRO\_TABLE() ...  const int BLACK = 3; |
| Parameter, Variable | camelCasing | Noun | exampleText, dwCount |
| Template parameter | PascalCasing  ‘T’ prefix | Noun | T, TItem, TPolicy |

### Prefixes for Identifiers

CPP/GNC30 🗷 You should not use Hungarian notation in parameter and variable names when writing new code. Hungarian notation is a relic that makes code refactoring harder; i.e. change a type of a variable and you need to rename it everywhere. Nevertheless, if you are working on a legacy component that uses Hungarian Notation be consistent and follow the style used there.

The Hungarian Notation conventions can be found [here](http://msdn.microsoft.com/en-us/library/windows/desktop/aa378932(v=vs.85).aspx).

CPP/GNC40 🗹 Do use prefixes for identifiers according to the following conventions:

|  |  |
| --- | --- |
| Prefix | Description |
| p | A pointer (32bit or 64 bit depending on platform). |
| sp | A ‘smart’ pointer, i.e. a class that has pointer-like semantics. |
| c | A count. For example, cBuffer means the byte count of a buffer. It is acceptable if "c" is not followed by a tag. |
| m\_ | A member variable in a class. |
| s\_ | A static member variable in a class. |
| g\_ | A global variable. |
| I, \_I | COM interface (\_I for outgoing Interface (Events)) |

## Pointers

🗹 You should always initialize pointers when you declare them and you should reinitialize them to nullptr after freeing them. This prevents the rest of the code from using an uninitialized pointer to corrupt the process’s address space by accidentally reading/writing to an unknown location. For example:

Good:

BINARY\_TREE \*pDirectoryTree = nullptr;

DWORD \*pdw = (DWORD \*)LocalAlloc(LPTR, 512);

...

if (pdw != nullptr)

{

LocalFree(pdw);

pdw = nullptr;

}

...

if (pDirectoryTree!= nullptr)

{

// Free directoryTree with match to way it was allocated

FreeBinaryTree(pDirectoryTree);

directoryTree = nullptr;

}

🗹 You should put no space between the '\*' character(s) and the type when specifying a pointer type/variable, but there should be a space between the '\*' character(s) and the variable. Setting this rule is to be consistent and uniform in code. Here are some examples:

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);

🗹 You should use the nullptr as null pointer constant to set pointers to an invalid value. nullptr is less vulnerable to misuse and works better in most situations.  
Example:

// defined elsewhere

void f(std::pair<const char \*, double>);

// compile error as the macro NULL expands to 0,

// so that the call std::make\_pair(0, 3.14) returns std::pair<int, double>

// which is not convertible to std::pair<const char \*, double>

f(std::make\_pair(NULL, 3.14));

// successfully compiles because std::make\_pair(nullptr, 3.14) returns

// std::pair<std::nullptr\_t, double>, which is convertible to

// std::pair<const char \*, double>

f(std::make\_pair(nullptr, 3.14));

Note: In C++/CLI the nullptr is already in use and is not interchangeable with the ISO Standard C++ keyword. You can use \_nullptr when the native interpretation is needed.

## Constants

🗹 Do define named constants as ‘const’ values, instead of “#define” values. For example:

Good:

const int BLACK = 3;

Bad:

#define BLACK 3

When you use const values, the compiler will enforce type checking and add the constants to the symbol table, which makes debugging easier. In contrast, the preprocessor does neither.

🗹 You should define groups of related constants using enum. This allows the group of constants to share a unique type and improves function interfaces. For example:

Good:

enum class DayOfWeek {Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

enum class 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);

## Constness

🗹 Do use const whenever possible. It makes the code more safe and integrates well with the C++ type system. The usage of const simplifies the code as it expresses explicitly the intent that this function call /variable usage will not modify the value. The reader of the code must only lookup the initialization of the variable to know the value of the variable for its whole scope:

void Fun(const std::vector<int>& v)

{

    const std::size\_t len = v.size();

}

🗹 You should use ‘const’ when passing and returning parameters where appropriate. By applying ‘const’ the intent of the code is clearly spelled out, and the compiler can provide an added level of verification that the code isn’t modifying values that it shouldn’t be.

|  |  |  |
| --- | --- | --- |
| Const usage | Meaning | Description |
| const int \*x; | Pointer to a const int | Value pointed to by x can’t change |
| int \* const x; | Const pointer to an int | x cannot point to a different location. |
| const int \*const x; | Const pointer to a const int | Both the pointer and the value pointed to cannot change. |

🗷 Do not cast away const except to call a const-incorrect function.

🗹 Do implement logical constness with mutable members.

class A

{

private:

    int mutable m\_callCounter;

public:

    void DoSomething() const

    {

        ++m\_callCounter;

        // do something

    }

};

## Casting

🗹 You should code more clearly. The following types of C++ casts are defined:

1. dynamic\_cast is almost exclusively used for handling polymorphism. You can cast a pointer or reference to any polymorphic type to any other class in the class hierarchy.
2. static\_cast handles related types. It can be used for ordinary type conversions.
3. reinterpret\_cast handles conversion between unrelated types. Do not cast from a DWORD to a pointer or visa-versa. It will not compile under 64 bits. Do comment your reinterpret\_cast<> usage; this is needed to relieve the concern that future readers will have when they see the cast.
4. const\_cast is used to cast away the ‘const’ness of an object.

The syntax for all three is similar:

DerivedClass \*pDerived = HelperFunction();

BaseClass \*pBase = static\_cast<BaseClass \*>(pDerived);

🗷 You should not use ‘const\_cast’ unless absolutely necessary. Having to use ‘const\_cast’ typically means that an API is not using ‘const’ appropriately. Note; The Win32 API doesn’t always use ‘const’ for passing parameters and it may be necessary to use const\_cast when using the Win32 API.

## Sizeof

🗹 Do use sizeof(var) instead of sizeof(TYPE) whenever possible. To be explicit about the size value being used whenever possible write sizeof(var) instead of sizeof(TYPE\_OF\_VAR). Do not code the known size or type of the variable. Do reference the variable name instead of the variable type in sizeof:

Good:

MY\_STRUCT s;

ZeroMemory(&s, sizeof(s));

Bad:

MY\_STRUCT s;

ZeroMemory(&s, sizeof(MY\_STRUCT));

🗷 Do not use sizeof for arrays to get the element number. Use ARRAYSIZE. If you have the option use the array class of the standard library instead of C-arrays.

## Strings

🗹 Do write explicitly UNICODE code because it enables easier globalization. Don’t use TCHAR or ANSI code. This means:

Use the wide char types including wchar\_t, PWSTR, PCWSTR instead of the TCHAR versions.

Good:

HRESULT Function(PCWSTR)

Bad:

HRESULT Function(PCTSTR)

Variable names should not indicate “W” in the name.

Good:

Function(PCWSTR psz)

Bad:

Function(PCWSTR pwsz)

Don’t use the TEXT macro, instead use the L prefix for creating Unicode string constants L”string value”.

Good:

L"foo"

Bad:

TEXT("foo")

Prefer wchart\_t to WCHAR because it is the native C++ type.

Good:

L"foo"

wchar\_t szMessage[260];

Bad:

TEXT("foo")

WCHAR szMessage[260];

Never explicitly use the A/W versions of the APIs. This is bad style because the names of the APIs actually are the base name without A/W. And, it makes code hard to port when you do need to switch between ANSI/Unicode.

Good:

CreateWindow(...);

Bad:

CreateWindowW(...);

🗹 You should use std::wstring as default representation for UTF16 strings and std::string for UTF8 strings.

## Arrays

🗹 Do prefer to use the std::array type instead of the C-style arrays where appropriate.

The standard container array is a fixed-sized random-access sequence of elements defined in <array>. It has no space overheads beyond what it needs to hold its elements, it does not use free store, it can be initialized with an initializer list, it knows its size (number of elements), and doesn't convert to a pointer unless you explicitly ask it to. It is very much like a built-in array without the problems.

array<int, 6> a = { 1, 2, 3 };

a[3] = 4;

// x becomes 0 because default elements are zero initialized

int x = a[5];

// error: std::array doesn't implicitly convert to a pointer

int\* p1 = a;

// ok: get pointer to first element

int\* p2 = a.data();

### C-Style Arrays

🗹 Do use ARRAYSIZE() as the preferred way to get the size of an array. ARRAYSIZE() is declared in a way that produces an error if it is used on a non-array type. For anonymous types you need to use the less safe \_ARRAYSIZE() macro. ARRAYSIZE() should be used instead of RTL\_NUMBER\_OF(), \_countof(), NUMBER\_OF(), etc.

🗹 Do derive the array size from the variable rather than specifying the size in your code.

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

}

🗹 Do use "= {}" to zero array memory. The compiler optimizer does better with "= {}" than "= {0}" and ZeroMemory, so "= {}" is preferred.

## Macros

🗷 You should not use macros unless they are absolutely necessary. Most functionality that is typically achieved through macros can be implemented with other C++ constructs (using constants, enums, inline functions, or templates) which will yield clearer, safer, and more understandable code.

Examples:

1. Use const or enum class for constants.
2. Use inline to avoid function call overhead
3. Use templates for a family of functions or types
4. Use namespaces to avoid name clashes

Good:

// Defined in header file; This enables the compiler to inline the function.

PCWSTR PCWSTRFromBSTR(\_\_in BSTR bstr)

{

return bstr ? bstr : L"";

}

Bad:

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

🗷 Do not use “#define” values for constant values.

🗷 Do not use the following existing macros: SIZEOF(), IID\_PPV\_ARG() (use IID\_PPV\_ARGS() instead).

## Expressions

🗹 Do use brackets in an expression, which include operators with different priority, so that the evaluation order can be seen explicitly. Remark: It has to be avoided, that the developer has to look into documentation in order to find out the evaluation order.

🗷 You should not use expressions, for which the order of evaluation within the expression influences the result. An exception to this rule is the usage of boolean expression that relies on the boolean short-circuit evaluation (e.g. ptr != nullptr && ptr->IsSomething().

🗹 You should call the prefix-form of the ++/-- operators if you have the choice between both kinds.

Using the prefix-form is more efficient than the postfix-form as the compiler must provide the old value a value for the expression. This is especially important when incrementing iterators.

## Functions and Operators

### Validating Parameters

🗹 Do validate parameters to functions that will be used by the public.

### Parameters

🗹 Do declare parameters of functions and methods appropriately by value, (smart) pointer or reference.

The right choice depends on whether the parameter is an input, output or input/output parameter. The following rules should be applied:

For input-only parameters:

* Const-qualify all (smart) pointers or references
* Prefer taking primitive type and value objects that are cheap to copy/move.
* Consider pass-by-value instead of reference when the function requires a copy anyway.

For output and input/output parameters:

* Prefer passing by (smart) pointer if the parameter is optional or if the functions stores the pointer.
* Prefer passing by reference if the parameter is required and the function will not store a pointer (i.e. ownership is not affected).

### Unreferenced Parameters

When implementing methods in an interface or standard export it is common for some of the parameters to not be referenced. The compiler detects unused parameters and will produce a warning that some components treat as an error. To avoid this comment out the unused parameter using the /\* param\_name \*/ syntax, don’t use the UNREFERENCED\_PARAMETER() macro since that is 1) less concise, 2) does not ensure that the parameter is in fact unreferenced.

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);

...

}

### Output String Parameters

A common way to return a string from a function is to have the caller specify the address where the value should be stored and the length of that buffer as a count of characters. This is the “pszBuf/cchBuf” pattern. In the method, you need to explicitly test the buffer size > 0 first.

In COM applications, you can return \_\_out string parameters as strings allocated by CoTaskMemAlloc / SysAllocString. This avoids string size limitations in the code. The caller is responsible for calling CoTaskMemFree / SysFreeString.

### Boolean Parameters

Very often a Boolean parameter in the parameter list of a function/method is an indicator for having multiple tasks. If this is the case, two separate functions/methods shall be implemented.

### Return Values

🗹 Do test the return of a function, not the out parameters, in the caller. Some functions communicate the success or failed state in multiple ways; for example in COM methods it is visible in the HRESULT and the out parameter. For example both of the following are correct.

Good:

IShellItemImageFactory \*psiif;

if (SUCCEEDED(psi->QueryInterface(IID\_PPV\_ARGS(&psiif))))

{

// Use psiff

psiif->Release();

}

Bad:

IShellItemImageFactory \*psiif;

psi->QueryInterface(IID\_PPV\_ARGS(&psiif));

if (psiif)

{

// Use psiff

psiif->Release();

}

The reasons are:

* The HRESULT is the more prominent result from the function and is the most appropriate to test.
* Usually the value of the HRESULT caries important information that needs to be propagated, this reduces the chance of mapping or treating all failures to E\_FAIL.
* Testing the HRESULT makes this code consistent with cases where the out params are not zeroed on failure (many win32 APIs).
* Testing the HRESULT enables putting the call and the test on the same line.
* Testing the HRESULT is more efficient in terms of code generation.

### Function Call

🗷 Do not write code that depends on the evaluation order of function arguments. As the evaluation order of expressions in functions arguments is not defined in C++ the compiler is free to reorder the expressions:

int i = 2;

// order of evaluation is unknown

// Func(3,4) or Func(4,3) ?

Func(++i, ++i);

### Operators

🗹 Do preserve natural semantics for overloaded operators.

🗷 Do not overload &&, || or comma because the built-in operators have a special treatment by the compiler (e.g. left-to-right evaluation with short-circuit evaluation for the Boolean operators).

🗹 Do consider to make an operator explicit to avoid unintentional conversions.  
Example:

Bad:

struct Testable

{

    operator bool() const

    {

        return false;

    }

};

struct AnotherTestable

{

    operator bool() const

    {

        return true;

    }

};

Good:

struct Testable

{

    explicit operator bool() const

    {

        return false;

    }

};

struct AnotherTestable

{

    explicit operator bool() const

    {

        return true;

    }

};

// The following comparisons are accidental and are not intended

// but the compiler would compile them without the explicit keyword

// for the operators.

Testable a;

AnotherTestable b;

if (a == b)

{

    // ...

}

if (a < 0)

{

    // ...

}

🗹 You should provide overloads for common cases to avoid unnecessary implicit type conversions.  
Example:

class String

{

    String(const char\* text); // enables implicit conversion

};

bool operator== (const String&, const String&);

//... somewhere in the code...

if (someString == "Hello")

{

   //...

}

// is replaced by the compiler with:

if (someString == String("Hello"))

{

    //...

}

// is prevented by providing:

bool operator==(const String& Ihs, const String& rhs);

bool operator==(const String& Ihs, const char\* rhs);

bool operator==(const char\* Ihs, const String& rhs);

## Structures

### Structure Initialization

🗹 Do use "= {}" to zero structure memory.

PROPVARIANT pv = {};

When a structure has a byte size field as the first member, you could use the following shortcut to initialize the size field and zero initialize the other fields:

SHELLEXECUTEINFO sei = { sizeof(sei) };

sei.lpFile = ...

### Structures vs. Classes

🗹 Do use a structure to define a data aggregate that does not contain functions. Use a class if the data structure includes member functions. In C++, a struct can have member functions and operators and everything else that a class can have. In fact, the only difference between a class and a struct is that all members default to public access in a struct but private access in a class. To match the normal intuition, we use a class if and only if there are member functions included.

## Classes

🗹 You should define only one class in one header and implementation file. There are cases, where deviants are reasonable, e.g. for very small classes with (logical) relationship to another class.

🗹 Do follow the rules for specific kind of classes as follows.

Different kind of classes require a different approach when being implemented [[REF3]](#REF3):

Value classes (such as std::pair or std::vector) are modeled as built-in types:

* Has a public destructor, copy constructor, and assignment with value semantics, and an optional move constructor and assignment operator.
* Has no virtual functions (including the destructor).
* Is intended to be used as a concrete class, not as a base class.
* Is usually instantiated on the stack or as a directly held member of another class.

Base classes that are the building blocks of class hierarchies:

* Has a destructor that is public and virtual or else protected and non-virtual, and a nonpublic copy constructor and assignment operator and an optional move constructor and assignment operator.
* Establishes interfaces through virtual functions.
* Is usually instantiated dynamically on the heap and used via a (smart) pointer.

Trait classes are templates that provides additional information about a type:

* Contains only typedefs and static functions. It has no modifiable state or virtuals.
* Is not usually instantiated (construction can normally be disabled).

Policy classes (normally templates) are fragments of pluggable behavior.

* May or may not have state or virtual functions.
* Is not usually instantiated standalone, but only as a base or member.

Exception classes exhibit an unusual mix of value and reference semantics: They are

thrown by value but should be caught by reference:

* Has a public destructor and no-fail constructors (especially a no-fail copy constructor; throwing from an exception's copy constructor will abort your program).
* Has virtual functions, and often implements cloning and visitation.
* Preferably derives virtually from std::exception.

Ancillary classes typically support specific idioms (e.g., RAII). They should be easy to use correctly and hard to use incorrectly.

🗹 Do prefer small classes to monolithic classes. Small classes are easier to write, get correct, test and use.

🗹 You can use the Pimpl idiom to hide the private implementation of a class from the client code. Private members in C++ are inaccessible but not invisible to the client (Layout of the class depends on it). By introducing a single pointer to an implementation class this coupling can be avoided. The benefits of the Pimpl idiom are:

* Minimization of compilation dependencies
* Separation of interface and implementation
* Portability

Example:

// MyClass.h

class MyClass

{

public:

    MyClass();

private:

    class MyImpl;

    unique\_ptr<MyImpl> m\_spImpl; // opaque type here

};

// MyClass.cpp

class MyClass::MyImpl

{

    // defined privately here

    // ... all private data and functions: all of these

    //     can now change without recompiling callers ...

public:

};

MyClass::MyClass()

    : m\_spImpl(new MyImpl)

{

    // ... set impl values ...

}

For details see [here](http://herbsutter.com/gotw/_100/).

🗹 You should prefer to write non-member non-friend functions. If you have the choice to provide a functionality based on the public interface of a type add it as function but not as a member to the type. This keeps the type simple and clean and nevertheless provides convenience function as extensions to the type.

### Data Members

🗷 Do not declare public data members. Use inline accessor functions for performance.

🗹 Do prefer initialization to assignment in constructors. For example, using initialization:

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;

std::wstring 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;

std::wstring m\_description;

float m\_accuracy;

};

🗹 Do define and initialize member variables in the same order. Members will be initialized in the order they are declared in the class definition, not the order of the initialization list.

Bad:

class Employee

{

private:

    std::string m\_email;

    std::string m\_firstName;

    std::string m\_lastName;

public:

    // bug: m\_email is declared before m\_firstName, m\_lastName

    // therefore it is initialized first

    Employee(const char\* firstName, const char\* lastName) :

        m\_firstName(firstName), m\_lastName(lastName),

        m\_email(m\_firstName, +"." + m\_lastName + "@bosch.com")

    {}

};

🗹 Do consider to initialize members in-class. This is really beneficial if your class has multiple constructors.

class A

{

public:

    A()

    {}

    A(int a)

        : m\_a(a)

    {}

private:

    int m\_a = 7;

    int m\_b = 5;

    std::wstring m\_s{ L"Constructor run" };

};

### Constructors

🗹 Do minimal work in the constructor. Constructors should not do much work other than to capture the constructor parameters and set main data members. The cost of any other processing should be delayed until required.

🗷 Do not call virtual functions in constructors and destructors. Inside constructors and destructors virtual functions behave not like virtual functions. For pure virtual functions this results in an undefined behavior.

🗹 Do define all single parameter constructors, by default, with the ‘explicit’ keyword, so that they are not conversion constructors.  
 For example,

class CodeExample

{

int m\_value;

public:

explicit CodeExample(int value) :

m\_value(value)

{

}

};

🗷 Do not provide conversion constructors unless the semantics of the class justify them.

🗹 Do use delegating constructors to avoid redundant code in constructors.

Bad:

class A

{

public:

    A()

        : m\_a(0)

    {

        // Do something

    }

    explicit A(int a)

        : m\_a(a)

    {

        if (m\_a >= 0)

        {

            // Do something

        }

        else

        {

            // Do something else

        }

    }

};

Good:

class A

{

public:

    A()

        : A(0)

    {

    }

    explicit A(int a)

        : m\_a(a)

    {

        if (m\_a >= 0)

        {

            // Do something

        }

        else

        {

            // Do something else

        }

    }

};

### Destructors

🗹 Do use a destructor to centralize the resource cleanup of a class that is freed via delete. If resources are freed before destruction, make sure the fields are reset (e.g. set pointers to null\_ptr) so that a destructor will not try to free them again.

🗹 Do declare the destructor as "virtual" for classes that contain at least one other virtual function. If the class does not contain any virtual functions, then do not declare the destructor as virtual. Having no virtual function implies that the class is not designed to be used polymorphically and should therefore also not be destroyed that way.

🗹 Do make base class destructors public and virtual or protected and non-virtual. If deletion thorough a pointer to the base class is allowed the destructor must be virtual. Otherwise it has to be disallowed.

🗷 Do not throw exceptions or let exceptions propagate out of destructors. Destructors should never fail.

Destructors are called by the compiler automatically in a number of cases (e.g. at an exit of a scope). There is no way to handle an exception for this automatic calls.

### Special Member Functions

In C++, the compiler automatically generates the default constructor, copy constructor, copy-assignment operator, and destructor for a type if it does not declare its own. These functions are known as the special member functions. C++11 brings move semantics to the language and adds the move constructor and move-assignment operator to the list of special member functions that the compiler can automatically generate.

🗹 Do explicitly enable or disable copying/moving. Think for every type about the copy/move semantics provided by the copy-constructor, assignment operator:

* Use the compiler generated ones.
* Explicitly disable both.
* Explicitly implement both.

Automatic generation of special member functions is governed by the following rules:

* If any constructor is explicitly declared, then no default constructor is automatically generated.
* If a virtual destructor is explicitly declared, then no default destructor is automatically generated.
* If a move constructor or move-assignment operator is explicitly declared, then:
  + No copy constructor is automatically generated.
  + No copy-assignment operator is automatically generated.
* If a copy constructor, copy-assignment operator, move constructor, move-assignment operator, or destructor is explicitly declared, then:
  + No move constructor is automatically generated.
  + No move-assignment operator is automatically generated.

The C++11 standards adds the following rules (currently not followed by Visual C++ (VS 2013/Update 2)):

* If a copy constructor or destructor is explicitly declared, then automatic generation of the copy-assignment operator is deprecated.
* If a copy-assignment operator or destructor is explicitly declared, then automatic generation of the copy constructor is deprecated.

The rules are further complicated in class hierarchies. It is therefore a good idea to be explicit about the usage of generated special member functions or to suppress them explicitly:

🗹 Do use explicitly defaulted or deleted special member functions even if the default compiler behavior would provide what is your intend. This makes your decision obvious to the reader of the class.

// class that can't be copied

// (but movable)

class Noncopyable

{

public:

// generates explicitly default constructor

    Noncopyable() = default;

// generates explicitly default destructor

    ~Noncopyable() = default;

// deletes explicitly the copy constructor

    Noncopyable(const Noncopyable&) = delete;

// deletes explicitly the assignment operator

    Noncopyable& operator=(const Noncopyable&) = delete;

    // implementation of move constructor

Noncopyable(const Noncopyable&& other)

    {

      // ...

    }

// implementation of move assignment operator

    Noncopyable& operator=(const Noncopyable&& other)

{

      // ...

    }

};

🗹 Do provide copy constructors, copy assignment operator, move constructor, move assignment operator and destructor in a consistent way when providing any of these.

* If you write/disable/default either the copy constructor or the copy assignment operator, you probably need to do the same for the other.
* If you write/disable/default either the move constructor or the move assignment operator, you probably need to do the same for the other.
* If you explicitly write the copying/moving functions, you probably need to write the destructor.
* If you explicitly write the destructor, you probably need to explicitly write or disable copying/moving.

Example: See the MemoryBlock class using the no-fail swap idiom.

🗹 Do use the canonical form of assignment with one of the following signatures:

T& operator=(const T&);

T& operator=(T); // to be used if you need to copy anyway

If the constructor was defined as “T(T& other)” or even “T(T& other, int value = 0)”, they would still be copy constructors. By standardizing on “const T&”, the constructor will work for both const and non-const values, with the added safety that const-ness brings.

🗹 You should consider to provide a no-fail swap when providing an copy assignment operator and follow the copy-and-swap idiom. Details can be found [here](http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms/Copy-and-swap).

class MemoryBlock

{

public:

    // Simple constructor that initializes the resource.

    explicit MemoryBlock(size\_t length)

        : \_length(length)

        , \_data(new int[length])

    {

    }

    // Destructor.

    ~MemoryBlock() \_NOEXCEPT

    {

    }

    // Copy constructor.

    MemoryBlock(const MemoryBlock& other)

        : MemoryBlock(other.\_length)

    {

        std::memcpy(\_data.get(), other.\_data.get(), \_length);

    }

    MemoryBlock& operator=(const MemoryBlock& other)

    {

        // copy-and-swap idiom

        MemoryBlock tmp(other);

        tmp.swap(\*this);

        return \*this;

    }

    // no-fail swap

    void swap(MemoryBlock& other) \_NOEXCEPT

    {

        \_data.swap(other.\_data);

        std::swap(\_length, other.\_length);

    }

private:

    size\_t \_length; // The length of the resource.

    std::unique\_ptr<int> \_data; // The resource.

};

🗹 Do support move semantics where appropriate. Move semantics enables you to write code that transfers resources (such as dynamically allocated memory) from one object to another. Move semantics works because it enables resources to be transferred from temporary objects that cannot be referenced elsewhere in the program. To implement move semantics, you typically provide a move constructor, and optionally a move assignment operator (operator=), to your class. Copy and assignment operations whose sources are rvalues then automatically take advantage of move semantics.

Example:

// Move constructor.

MemoryBlock(MemoryBlock&& other)

: \_length(other.\_length), \_data(std::move(other.\_data))

{

    other.\_length = 0;

}

// Move assignment operator.

MemoryBlock& operator=(MemoryBlock&& other)

{

// just: other.swap(\*this);

    // or for efficency and other reasons explicitly implemented

    // (see Scott Meyers blog for details

    // http://scottmeyers.blogspot.de/2014/06/

// the-drawbacks-of-implementing-move.html)

    \_data.reset(other.\_data.release());

    \_length = other.\_length;

    other.\_length = 0;

     return \*this;

}

// usage: of move semantics

// Combines to memory blocks as a new one

MemoryBlock Combine(const MemoryBlock& other) const

{

// create new block on stack and return it

// Move semantics will avoid that data is copied again

    auto result = MemoryBlock(this->\_length + other.\_length);

    std::memcpy(result.\_data.get(), this->\_data.get(), this->\_length);

    std::memcpy(result.\_data.get() + this->\_length

, other.\_data.get(), other.\_length);

    return result;

}

### Operators

🗷 Do not overload operator&&, operator|| or operator,. Unlike the built-in &&, || or , operators the overloaded versions cannot be short-circuited, so the resulting behavior of using these operators typically isn’t what was expected.

🗷 You should not overload operators unless the semantics of the class justify it.

🗷 Do not change the semantics of the operators if you choose to overload them. For example, do not re-purpose the ‘+’ operator for performing subtraction.

### Conversions

🗷 You should not provide implicit conversions without a second thought. Prefer to provide explicit conversions (explicit constructors or conversion methods).

* Implicit conversions may occur at places that were not intended.
* Implicit constructors produce unexpected temporary objects.

Example:

class String

{

public:

    String(int n); // allocate n bytes to the String object

};

// will convert character 'x' to an int

// and will call String(int)

String mystring = 'x';

🗹 Do use explicit for single parameter constructors when you want to avoid an implicit conversion.

class String

{

public:

    explicit String(int n); // allocate n bytes to the String object

};

### Function Overloading

🗷 Do not arbitrarily varying 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.

🗹 You can use default parameters instead of overloads where appropriate.

### Inheritance and Virtual Functions

🗹 Do prefer composition over inheritance. Inheritance introduces a strong coupling between two classes. If substitutability is not needed and a weaker relationship is sufficient (such as composition) prefer this.

🗷 Do not inherit from classes that were not designed to be a base class.

Use composition instead that wraps the original one by the new one and use delegation for implementing the new class.

🗹 Do use public inheritance for substitutability (dynamic polymorphism). Inheritance should not be used for reuse, but to be reused.

Don't use (public) inheritance to reuse the code that exists in the base class (see also: Prefer composition over inheritance), but use it to allow existing code that already uses base objects polymorphically to be reused.

🗹 Do use virtual functions to implement dynamic polymorphism.

🗹 Do override methods safely in derived classes.

The original contract (pre-/post-conditions) of the base class must be preserved by the overridden method in the derived class. The derived class must be substitutable for the base class (Liskov substitution principle) [[REF4]](#REF4).

🗹 Do use the override specifier to express your intend when overriding a method. The override keyword will cause the compiler to ensure that the method prototype matches a virtual function in a base class. If a change is made to the prototype of a virtual function on a base class, then the compiler will generate errors for derived classes that need to be fixed.

class SomeClass

{

public:

    SomeClass();

    virtual ~SomeClass();

    virtual void Foo();

};

class SomeDerivedClass : public SomeClass

{

public:

    SomeDerivedClass();

    virtual ~SomeDerivedClass() override;

    virtual void Foo() override;

};

🗷 Do not use virtual methods unless you really should because virtual functions have overhead of calling through the vtable.

🗹 You should consider to make virtual methods non-public, and public methods non-virtual. A public virtual method serves two purposes:

* It defines the interface of the class.
* It provides an extension point derived classes to modify the behavior.

These two purposes can be separated by providing a non-virtual public method that forms the interface and a private (or protected) methods that represents the extension point for the derived classes (Non-Virtual Interface pattern similar to the Template Method [[REF3]](#REF3) design pattern).

### Abstract Classes

An abstract class provides a polymorphic base class and requires a derived class to provide implementation for virtual methods.

🗹 Do prefer to define and inherit from abstract classes. See Dependency Inversion Principle (DIP) [[REF4]](#REF4)

🗹 You can use the 'abstract' keyword to identify an abstract class. Note: this is a Microsoft specific extension to C++.

🗹 Do provide a protected constructor.

🗹 Do identify abstract methods by making them pure virtual.

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

🗹 Do explicitly provide protected copy constructor and assignment operators or delete copy constructor and assignment operators – this will cause a compilation error if a user accidentally uses an abstract base class that results in pass-by-value behavior or disallows this for the entire hierarchy.

An example of an abstract class:

class AbstractClass

{

protected:

    AbstractClass() = default;

    AbstractClass(const AbstractClass&) = default;

    AbstractClass& operator=(const AbstractClass&) = default;

public:

    virtual ~AbstractClass() = default;

    virtual int DoSomething() = 0;

};

### Interfaces

A special case of an abstract class in one which contains only pure virtual functions, thus providing a construct similar to the interface constructs of other languages.

🗹 Do provide a public, virtual destructor to allow deletion via a pointer to the interface class.

## Namespaces

🗹 Do keep the type and its nonmember functions in the same namespace. Both public member functions and nonmember functions are part of the interface of a type. C++ enforces this by applying argument-dependent lookup (ADL, also known as Koenig lookup). This ensures that code that uses an object of a specific type can use its nonmember function interface (e.g. cout << x will invoke the operator<< for X when x is of type X).

🗹 Do keep types and functions in separate namespaces unless they are designed to work together. This rule is the backside of the previous and prevents the unintended lookup of functions and operators for types by accident. Examples can be found in [[REF3]](#REF3).

🗷 Do not use namespace usings in a header file or before any #include. In implementation files you should use usings to make the code more readable. Nevertheless usings in header files or before any include may influence other code that follows in an unintended way.

Bad:

// SomeClass.h

#include <string>

using namespace std;

class SomeClass

{

public:

    wstring GetSomeText();

private:

    wstring m\_sometext;

};

Bad:

// SomeClass.cpp

using namespace std;

#include "SomeOtherClass.h"

Good:

// SomeClass.h

#include <string>

class SomeClass

{

public:

    std::wstring GetSomeText();

private:

    std::wstring m\_sometext;

};

// SomeClass.cpp

#include "SomeOtherClass.h"

using namespace std;

// use string without namespace std::

wstring SomeClass::GetSomeText()

{

    // ...

}

## Allocations

🗹 Do ensure that all allocated memory is freed using the same mechanisms. Objects allocated using ‘new’ should be freed with ‘delete’. For example:

Engine \*pEngine = new Engine();

pEngine->Process();

delete pEngine;

Allocations made using ‘vector new’ should be freed using ‘vector delete’. For example:

wchar\_t \*pszBuffer = new wchar\_t[MAX\_PATH];

SomeMethod(pszBuffer);

delete [] pszBuffer;

🗹 Do understand the allocations within your code base to ensure that they are freed correctly.

### Smart Pointers

🗹 You should ensure that resources are owned by objects. Use explicit RAII (resource allocation is initialization) and smart pointers.

Instead of raw pointers prefer to hold dynamically allocated objects by smart pointers. The previous two examples written using STL ‘smart pointer’ classes would be:

{

    std::unique\_ptr<Engine> spEngine(new Engine());

    spEngine->Process();

}

{

    std::unique\_ptr<wchar\_t> spBuffer(new wchar\_t[MAX\_PATH]);

    SomeMethod(spBuffer.get());

}

Note: C++11 has no specialization for std::shared\_ptr<T[]>. This causes shared\_ptr to call delete instead of delete []. Therefore an explict delete function is necessary there.

🗹 You should always use the standard smart pointers, and non-owning raw pointers. Never use owning raw pointers and delete, except in rare cases when implementing your own low-level data structure.

* Use unique\_ptr to express unique ownership, if you know you’re the only owner of another object.
* Use weak\_ptr to break cycles and express optionality (e.g., implementing an object cache).
* Use shared\_ptr to express shared ownership. Prefer to use make\_shared to create shared objects efficiently.

🗹 Do perform each explicit resource allocation in one statement that hands the resource to the manager objects. As the evaluation order of the functions parameters in undefined a resource leak may occur otherwise:

Bad:

// This may produce a memory leak in case of

// an exception thrown by the SomeClass constructor

Fun(std::shared\_ptr<SomeClass>(new SomeClass),

    std::shared\_ptr<SomeClass>(new SomeClass));

Good:

// Allocate only one resource per statement

auto first = std::shared\_ptr<SomeClass>(new SomeClass);

auto second = std::shared\_ptr<SomeClass>(new SomeClass);

Fun(first, second);

🗹 You can define typedefs for smart-pointers to a specific class for convenience.   
Example:

typedef shared\_ptr<Foo> FooPtr;

## Errors and Exceptions

🗹 Do prefer using exceptions over error codes to report errors. Only use status codes (e.g., return codes, errno) for errors when exceptions cannot be used (see next item), and for conditions that are not errors. Use other methods, such as graceful or ungraceful termination, when recovery is impossible or not required.

🗹 Do always use error codes from DLL-exported APIs or methods. An exception to this rule is a DLL that is only used in a specific context where it is the guaranteed that the client of the DLL and the DLL itself are always compiled in a consistent way (Same compiler/ compatible compiler options concerning exceptions). In this scenario prefer to use exceptions.

### Errors

🗹 Do check return values for function calls and handle errors appropriately. When detecting an error, print the error message as early as possible in console applications, and handle the error. For example,

// Function returns 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 returns TRUE/FALSE and sets the 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;

}

### Exceptions

Native C++ exceptions are a powerful feature of the language, and can reduce the complexity of code, and reduce the amount of code that is written and maintained.

🗹 Do throw exceptions by value and catch exceptions by reference. For example,

void ProcessItem(const Item& item)

{

try

{

if (/\* some test failed \*/)

{

throw \_com\_error(E\_FAIL);

}

}

catch(\_com\_error& comError)

{

// Process comError

//

}

}

🗹 When re-throwing exceptions do re-throw exceptions using “throw;” instead of “throw <caught exception>”. For example,

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;

}

}

🗷 Do not allow exceptions to be thrown out of destructors.

🗷 Do not use “catch(…)”.General exceptions should not be caught. You should catch a more specific exception, or re-throw the general exception as the last statement in the catch block. There are cases when swallowing errors in applications is acceptable, but such cases are rare. Only catch the specific exceptions that the function knows how to handle. All others must be passed unhandled.

🗷 Do not use exceptions for control flow. 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,

if (IsWritable(list))

{

WriteList(list);

}

🗹 Do make sure that you understand the exceptions that may be thrown from code that you take a dependency on, and ensure that the exceptions aren’t unintentionally propagated to the consumers of your API. For example, STL and ATL can throw native C++ exceptions in certain scenarios – understand those scenarios and ensure that the appropriate exceptions are handled in your code to prevent propagation out.

🗷 Do not use exception specifications. Exception specifications (other than noexecpt) are officially deprecated due to some design flaws. Details can be found [here](http://www.gotw.ca/publications/mill22.htm) and in [[REF3]](#REF3).

## Control Flow

### Goto

🗷 Do not use ‘goto’ statements in place of structured control flow in attempts to “optimize” runtime performance. Doing so leads to code which is very hard to understand, debug, and verify correct.

🗹 You can use goto in a structured manner by always jumping forward and by implementing a consist set of jumps related to a single purpose such as jumping out of a series of resource allocations into cleanup code when one resource cannot be allocated. Such use of goto can reduce deep levels of nesting and make error handling much easier to see and verify. For example:

HANDLE hToken = nullptr;

PVOID pMem = nullptr;

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 = nullptr;

}

if (pMem)

{

    LocalFree(pMem);

    pMem = nullptr;

}

## Debug Macro Usage

In general one needs to distinguish between Debug Version and Release Build of programs and modules. The following topics should be covered:

🗷 You should not generate side-effect from Debug sections (e.g. Hidden initialization).

🗹 You should put Code, which is part of the debug version but not of the release build, into a #ifdef DEBUG/#endif bracket.

🗹 You can provide macros that expand only in the debug version to code.

## STL

🗹 Do use STL classes whenever possible and reasonable. E.g. std::string instread of char \*

🗹 Do prefer to use STL containers and algorithms over manual code.

🗹 Do use vector<T> as standard sequence if you have no specific requirements.

🗹 Do use vector<T> and string::c\_str() to exchange data with non-C++ APIs. The vector's storage is always continues. Therefore the address to the first element is a pointer to its content.

🗹 Do store only values and appropriate smart pointers in containers. Containers assume a value-like semantic of types they contain. Therefore use only values or smart pointers with that semantic in containers. The first choice is the std::unique\_ptr.

🗹 Do prefer push\_back over other means to expand a sequence; push\_back uses an exponential and not fixed size to expand the capacity of an sequence. This provides a amortized constant time behavior of the expansion operation. Other means may be as bad as quadratic.

🗹 Do use range operations if this is an option.

🗷 Do not use std::auto\_ptr. Because of some design flaws it has been deprecated.

🗹 Do use std::shared\_ptr, std::unique\_ptr, std::weak\_ptr. unique\_ptr should be the default smart pointer used by new C++ code that replaces pointers as much as possible. unique\_ptr represents the single ownership idiom – it cannot be copied and assigned, and it cleans up the pointed object when it’s destructed.

🗹 Do prefer to use the nonmember begin/end, because begin(x) and end(x) are extensible and can be adapted to work with all container types. If you’re using a non-STL collection type that provides iteration but not STL-style x.begin() and x.end(), you can often write your own non-member begin(x) and end(x) overloads for that type.

Example: C arrays are such a type, and the standard provides begin and end for arrays.

vector<int> v;

int a[100];

// C++98

std::sort(v.begin(), v.end());

std::sort(&a[0], &a[0] + sizeof(a) / sizeof(a[0]));

// C++11

std::sort(begin(v), end(v));

std::sort(std::begin(a), std::end(a));

🗹 Do prefer to use Lambdas for STL algorithms. Lambdas make STL algorithms usable.  
Example:

vector<int> v;

 int x = 1;

 int y = 200;

 // C++98: write a naked loop

 // (using std::find\_if is impractically difficult)

 vector<int>::iterator i = v.begin(); // because we need to use i later

 for (; i != v.end(); ++i)

 {

     if (\*i > x && \*i < y)

         break;

 }

 // C++11: use std::find\_if

 auto i = std::find\_if(begin(v), end(v),

     [=](int i) { return i > x && i < y; });

# C++/CLI

C++/CLI (Common Language Infrastructure) was designed to bring C++ to .NET as a first-class language for developing managed code applications, and specifically to simplify writing managed code with C++. As C++/CLI is kind of a hybrid the rules given for C#/native C++ applies correspondingly. This section contains only a short advice to the usage of C++/CLI due to the limited amount of published best practices [[REF9]](#REF9). It must therefore be extended during the course of development by the lesson learned from own experiences.

## Bridges

In Endeavour C++/CLI will mainly be used to write bridge code that brings native C++ code to the managed world. This method represents the highest performing interoperability between managed and native code.

The structure of a bridge is as follows:

* The bridge assembly provides managed types that implements managed interfaces defined in a pure managed assembly.
* The implementation code of that types in the bridge is very thin. The actual work is delegated to native types in pure native library. The bridge is only responsible to make the translation of managed to unmanaged types. The translation includes exceptions from the native code to meaningful exceptions in the managed world.

Example:

// Native class defined in a purely native library/DLL

class NativeClass

{

public:

    NativeClass();

    ~NativeClass();

    void DoSomething();

};

// Managed interface defined in a purely managed assembly

// as abstraction for the NativeClass

public interface IMyInterface

{

    void DoSomething();

}

// Bridge class to translate from native to managed

// defined in the bridge C++/CLI assembly

ref class BridgeClass : public IMyInterface

{

public:

    BridgeClass()

     :m\_pNativeClass(new std::unique\_ptr<NativeClass>(new NativeClass()))

{

}

    ~BridgeClass();

{

     delete m\_pNativeClass;

}

    virtual void DoSomething();

{

     m\_pNativeClass->get()->DoSomething();

}

private:

    std::unique\_ptr<NativeClass>\* m\_pNativeClass;

};

// usage of abstraction

{

    // CreateMyInterface is some mechanism to create an instance of the bridge class

// and return the interface abstraction (e.g. a dependency injection container).

// The auto\_handle is used for automatic resource management

// (see Resource Management chapter).

    msclr::auto\_handle<IMyInterface> myObj{ CreateMyInterface() };

    myObj->DoSomething();

}

## Control Flow

CPC/CF/10 🗹 Do prefer range-for/for each statement over the "conventional" for statement.

In C++/CLI for each can be used for managed and unmanaged collections while the range-for can be used for native collections. For native collections range-for should be preferred.

std::array<int, 3> nativeArray = { 1, 2, 3 };

array<int>  ^managedArray = gcnew array<int> { 4, 5, 6 };

// iterate by for each

for each (int v in managedArray)

{

    // Do something

}

// works also for unmanaged sequences

for each (int v in nativeArray)

{

    // Do something

}

// but range-for is here preferred

for (auto v : nativeArray)

{

    // Do something

}

// range-for allows to change the values in the sequence

for (auto& v : nativeArray)

{

    v += 1;

}

## Resource Management

The key for safe resource management is to ensure that all resources are contained within classes that embody the CLI's resource management pattern, as defined by the IDisposable interface. Implementing this interface correctly can be challenging (see [C# section](#_Resource_Cleanup)), particularly when it comes to object hierarchies. Fortunately, C++/CLI takes care of the details and leaves you only to write a classic destructor for the class that is wrapping the resource. The important part here is that destructor in C++/CLI depart of the syntactically similarity to the C# finalizer is used for deterministic clean-up (i.e. Dispose). It must therefore be implemented in such a way that they can be called multiple times.

🗹 Do use auto\_handle to automatic resource management to handles to managed types.

public ref class ResourceOwner

{

private:

    msclr::auto\_handle<Resource> m\_resource;

public:

    ResourceOwner()

        : m\_resource{ gcnew Resource() }

    {

        // create other resources

    }

    ~ResourceOwner()

    {

        // delete other resouces

    }

};

// usage somewhere in code

msclr::auto\_handle<ResourceOwner> resourceOwner{ gcnew ResourceOwner() };

🗹 You can use stack semantics for managed types for automatic cleanup.

{

    ResourceOwner resourceOwner;

    // Do something with resource owner

    // Destructor will be called at scope boundary

}

## Mixing Managed and Native Types

When making use of native types from within managed types one challenge is how to allocate and store them within the realm of the managed heap. As managed types are created on the managed heap, where the rules that native types take for granted are simply not available—namely, stable addresses in a process's address space.

Embedding native type per value in a managed type is not supported by the compiler:

class NativeType

{

};

ref class ManagedWrapperType

{

    // compiler error:

    NativeType  m\_native;

};

Therefore one needs to manage raw pointers in the managed type:

🗹 Do use pointers to a smart pointer to manage a native type within a managed type and implement a destructor to clean-up the smart pointer.  
Example:

ref class ManagedWrapperType

{

private:

    std::shared\_ptr<NativeType>\*  m\_pNative;

public:

    ManagedWrapperType(const std::shared\_ptr<NativeType>& pNative)

        : m\_pNative{ new std::shared\_ptr<NativeType>(pNative) }

    {

    }

    ~ManagedWrapperType()

    {

        delete m\_pNative;

    }

};

For managing managed types in native types Visual C++ provides the gcroot native template class as a type-safe garbage collected root. In other words, it allows you to store a handle to a managed object on the native heap, in such a way that the CLR will know that the handle exists, update it as the managed heap is compacted, and not reclaim the object's memory prematurely.

gcroot is a class included in the Visual C++ libraries. It wraps the GCHandle type from the .NET Framework, which does the actual work of managing the native pointers used to look up the objects in the managed heap. You can find the gcroot class in the <vcclr.h> header file. gcroot provides a type-safe, CLR-aware object handle for use on the native heap. It does not care about managed types that need to be disposed. For that, you can use the auto\_gcroot class defined in the <msclr\auto\_gcroot.h> header file. The auto\_gcroot native template class also provides transfer-of-ownership semantics. As with the gcroot class, its purpose is to simplify the task of storing managed types within native types, but it also takes care of disposing the object that it owns.

🗹 You can use gcroot to wrap a handle to a managed type.

ref class ManagedType

{

};

class NativeWrapperType

{

public:

    ~NativeWrapperType()

    {

        delete m\_managed;

    }

private:

    gcroot<ManagedType^> m\_managed;

};

🗹 Do use auto\_gcroot to wrap a resource with enforced strict ownership and proper cleanup of the handle to the managed type.

ref class ManagedType

{

};

class NativeWrapperType

{

    msclr::auto\_gcroot<ManagedType^> m\_managed;

};

## Marshalling

In mixed mode, you have to marshal your data between native and managed types. C++/CLI has a C++ support library which includes code to help to marshal and convert data in a simple way. You can use the marshaling library with or without a marshal\_context class. Some conversions are easier to handle with a context, other conversions can be implemented using the marshal\_as function. Marshaling requires a context only when you marshal from managed to native data types and the native type you are converting to does not have a destructor for automatic clean up. The following table lists most common conversions needed, whether they require a context, and what marshal file you have to include:

|  |  |  |  |
| --- | --- | --- | --- |
| From type | To type | Marshal method | Include file |
| System::String^ | const char \* | marshal\_context | marshal.h |
| const char \* | System::String^ | marshal\_as | marshal.h |
| char \* | System::String^ | marshal\_as | marshal.h |
| System::String^ | const wchar\_t\* | marshal\_context | marshal.h |
| const wchar\_t \* | System::String^ | marshal\_as | marshal.h |
| wchar\_t \* | System::String^ | marshal\_as | marshal.h |
| System::IntPtr | HANDLE | marshal\_as | marshal\_windows.h |
| HANDLE | System::IntPtr | marshal\_as | marshal\_windows.h |
| System::String^ | std::string | marshal\_as | marshal\_cppstd.h |
| std::string | System::String^ | marshal\_as | marshal\_cppstd.h |
| System::String^ | std::wstring | marshal\_as | marshal\_cppstd.h |
| std::wstring | System::String^ | marshal\_as | marshal\_cppstd.h |

Example:

using namespace Runtime::InteropServices;

using namespace msclr::interop;

//indirect via const wchar\_t\* with context usage

marshal\_context context;

std::wstring stdString = context.marshal\_as<const wchar\_t\*>(s);

// directly to wstring

std::wstring stdString = marshal\_as<std::wstring>(s);

Details can be found [here](http://msdn.microsoft.com/en-us/library/bb384865.aspx).

## Delegates

The following example extents the bridge class from §5.1 to demonstrate how to translate a managed delegate/event to an unmanaged function pointer. A managed delegate is created and GetFunctionPointerForDelegate is used to retrieve the underlying entry point for the delegate. This address is then passed to the unmanaged function. It is necessary to protect the delegate from premature garbage collection, but pinning is not necessary. If a delegate is re-located by a garbage collection, it will not affect the managed callback, so Alloc is used to add a reference to the delegate, allowing relocation of the delegate, but preventing disposal. Using GCHandle instead of pin\_ptr reduces fragmentation potential of the managed heap.

// Native class defined in a purely native library/DLL

class NativeClass

{

public:

    // ...

    void OnValue(std::function<void(int )> func)

    {

        m\_callback = func;

    }

private:

// Native function pointer

    std::function<void(int)> m\_callback;

};

// Managed interface defined in a purely managed assembly

// as abstraction for the NativeClass

public interface IMyInterface

{

// exposed event for the abstraction

event Action<int> ValueReceived;

}

// Bridge class to translate from native to managed

// defined in the bridge C++/CLI assembly

ref class BridgeClass : public IMyInterface

{

public:

    BridgeClass()

// ...

{

// ...

// create a non-generic delegate

m\_callback = gcnew  MyEventDelegate(this, &BridgeClass::OnValue);

// create a handle for the delegate

m\_callbackHandle = GCHandle::Alloc(m\_callback);

// create a function pointer

IntPtr ptr = Marshal::GetFunctionPointerForDelegate(m\_callback);

// wrap it in a std::function

auto func = std::function<void(int)>((void(\_\_cdecl\*)(int))ptr.ToPointer());

// pass it to the native class

(\*m\_pNativeClass)->OnValue(func);

}

    ~BridgeClass();

{

// reset the function pointer

(\*m\_pNativeClass)->OnValue(std::function<void(int)>());

// free the handle to the delegate

m\_callbackHandle.Free();

}

// ...

};

## Exceptions

The bridge class has the responsibility to translate exceptions or error codes exposed by the native class to appropriate managed exception. When an unmanaged object type is thrown in C++/CLI, it is wrapped with an exception of type System::Runtime.InteropServices::SEHException. When searching for the appropriate catch clause, there are two possibilities:

* If a native C++ type is encountered, the exception is unwrapped and compared to the type encountered. This comparison allows a native C++ type to be caught in the normal way.
* However, if a catch clause of type SEHException or any of its base classes is examined first, the clause will intercept the exception. Therefore:

🗹 Do place all catch clauses that catch native C++ types first before any catch clauses of CLR types.

Example:

void BridgeClass::MayCauseException(int v)

{

    try

    {

       m\_pNativeClass->get()->MayCauseException(v);

    }

    catch (std::out\_of\_range&  ex)

    {

       throw gcnew ArgumentOutOfRangeException(marshal\_as<System::String ^>(ex.what()));

    }

    catch (std::invalid\_argument&  ex)

    {

       throw gcnew ArgumentException(marshal\_as<System::String ^>(ex.what()));

    }

}

References

[1] [All-In-One Code Framework](http://1code.codeplex.com/wikipage?title=All-In-One%20Code%20Framework%20Coding%20Standards)

[2] Framework Design Guidelines: conventions, idioms, and patterns for reusable .NET Libraries. Krzysztof Cwalina, Brad Abrams. 2006.

[3] C++ Coding Standards, Herb Sutter, Andrei Alexandrescu, 2004.

[4] [Design Principles](http://www.objectmentor.com/resources/publishedArticles.html), Robert C. Martin

[5] [Herb Sutter](http://herbsutter.com/gotw/), Guru of the Week

[6] Stroustrup, [C++11 FAQ](http://www.stroustrup.com/C++11FAQ.html)

[7] Herb Sutter, [Elements of Modern C++ Style](http://herbsutter.com/elements-of-modern-c-style/),

[8] [More C++ Idioms](http://en.wikibooks.org/wiki/More_C%2B%2B_Idioms)

[9] [Best Practices for Writing Efficient and Reliable Code with C++/CLI](http://msdn.microsoft.com/en-us/library/aa730837(v=vs.80).aspx), MSDN