Sign in

dotnet / runtime Public

Notifications Fork 4.7k Star 15.2k

Code Issues 5k+ Pull requests 324 Discussions Actions Projects 6 Securi

runtime / docs / coding-guidelines / clr-jit-coding-conventions.md

2040 lines (1496 loc) · 78.5 KB

CLR JIT Coding Conventions

May 2015

Overview

Consistent code conventions are important for several reasons:

- *Most importantly:* To make it easy to read and understand the code. Remember: you may be the only one writing the code initially, but many people will need to read, understand, modify, and fix the code over its lifetime. These conventions attempt to make this task much easier. If the code consistently follows these standards, it improves developer understanding across the entire source base.
- To make it easy to debug the code, with both the Visual Studio and windbg debuggers. It should be easy to set breakpoints, view locals, and display and view data structures.
- To make it easier to search and browse the code, both with Visual Studio IntelliSense, and with simple tools like "grep" and Notepad. This implies, for example, that names must be unique, and not require C++ parser intelligence (for example, Visual Studio IntelliSense) to distinguish.

- To attempt to improve code quality through consistency, and requiring patterns that are less likely to result in bugs either initially, or after code modification.

For these reasons, C++ code that is part of the Common Language Runtime (CLR) Just-In-Time compiler (JIT) will follow these conventions.

Note that these conventions are different from the CLR C++ Coding Conventions, documented elsewhere and used for the VM code, though they do have strong similarities. This is due to historical differences in the JIT and VM code, the teams involved in forming these conventions, as well as to technical differences in the code itself.

> Note: the JIT currently doesn't follow some of these conventions very widely. The non-conformant code should be updated, eventually.

> Note: we now use jit-format to format our code. All changes it makes supersede the conventions in this doc. Please see the jit-format documentation for instructions on running jit-format.

How to use this document

- All new code written in the JIT should adhere to these conventions.
- Existing code that does not follow these conventions should be converted to these conventions when it is modified.
  - Typically, conversion to these conventions would be done on a function basis.
  - You need to balance this suggestion against the very real value of submitting a minimal change for any individual fix or feature. Consider doing convention changes as a separate change, to avoid polluting the changes for a bug fix (or other change) with convention changes (which would make it harder to identify exactly which changes are strictly required for the bug fix).
- Code reviewers should look for adherence to the conventions.

Contents

4 Principles

As stated above, the primary purpose of these conventions is to improve readability and understandability of the source code, by making it easier for any developer, now or in the future, to

easily read, understand, and modify any portion of the source code.

It is assumed that developers should be able to use the Visual Studio editor and debugger to the fullest extent possible. Thus, the conventions should allow us to leverage Visual Studio IntelliSense, editing and formatting, debugging, and so forth. The conventions will not preclude the use of other editors. For example, function declaration commenting style should be such that IntelliSense will automatically display that comment when typing that function name at a use site. Indenting style should be such that using Visual Studio automatic formatting rules creates correctly formatted code.

5 Spaces, not tabs

Use spaces, not tabs. Files should not contain tab characters.

Indenting is 4 characters per indent level.

In Visual Studio, go to "Tools | Options ... | Text Editor | All Languages | Tabs", edit the tab size setting to be 4 spaces, and select the "Insert spaces" radio-button to enable conversion of tabs to spaces.

6 Source code line width

A source code line should be limited to a reasonable length, so it fits in a reasonably-sized editor window without scrolling or wrapping. Consider 120 characters the baseline of reasonable, adjusted for per-site judgment.

> Rationale: Modern widescreen monitors can easily display source files much wider than 120 characters, however we don't encourage (or allow) that for a number of reasons:
>
> 1. Very long lines tend to make the code more difficult to read
> 2. If the need for long lines is because there is a lot of scope-based indentation, that is an indication that refactoring is necessary to reduce the number of nested scopes.
> 3. Many people place other windows side-by-side with source code (such as additional source code windows, or Visual Studio tool windows like the Code Definition Window), or use side-by-side "diff" programs. Thus, making (most) code visible when viewed side-by-side, without scrolling, is advantageous.
> 4. Even if there are occasional uses for wide lines, it is expected that most lines will be much shorter, leaving a considerable amount of wasted space that could be used for other purposes (see #3).

Many editors support display of a vertical line at a specified column position. Enable this in your editor to easily know when you write past the specified maximum column position. Visual Studio has this feature if you install the "Productivity Power Tools": https://visualstudiogallery.msdn.microsoft.com/3a96a4dc-ba9c-4589-92c5-640e07332afd.

7 Commenting

7.1 General

A comment should never just restate what the code does. Instead it should answer a question about the code, such as why, when, or how. Comments that say we must do something need to also state why we must do this.

Avoid using abbreviations or acronyms; it harms readability.

7.1.1 Comment style

We prefer end-of-line style `//` comments to original C `/* */` comments.

One important exception is when adding a small comment within an argument list to help document the argument, e.g., `PrintIt(/* AlignIt */ true);` (However, see section FIXTHIS for the suggested alternative to the form.)

7.1.2 Spelling and grammar

Check for spelling and grammar errors in your comments: just because you can understand the comment when you write it doesn't mean somebody else will parse it in the same way. Carefully consider the poor reader, especially those for whom English is not their first language.

7.1.3 Bug IDs

Don't put the bug or issue identifier (ID) of a fixed bug or completed feature in the source code comments. Such IDs become obsolete when (and not if) the bug tracking system changes, and are an indirect source of information. Also, understanding the bug report might be difficult in the absence of fresh context. Rather, the essence of the bug fix should be distilled into an appropriate comment. The precise condition that a case covers should be specified in the comment, as for all code.

In particular, putting a bug ID in a comment is often a shortcut people use to avoid writing a complete, descriptive comment about the code. The code and comments should stand alone.

Bug IDs of active bugs may be used in the source code to prevent other people from having to debug the problem again, only to figure out that a bug has already been opened for it. These bug IDs are expected to be removed as soon as the bug is fixed.

One thing that would be useful is for a particular case in code to be associated with a test case that exercises the case. This only makes sense if the case in question is localized to one or a few locations in the code, not pervasive and spread throughout the code. However, we don't currently have a good mechanism for referencing test cases (or other external metadata).

7.1.4 Email names

Email names or full names should not be used in the source code as people move on to other projects, leave the company, leave another company when working on the JIT in the open source world, or simply stop working on the JIT for some reason. For example, a comment that states, "Talk to JohnDoe to understand this code" isn't helpful after JohnDoe has left the company or is otherwise not available.

7.1.5 TODO

"TODO" comments in the code should be used to identify areas in the code that:

- May require tuning for code quality (runtime performance) or throughput.
- Need some cleanup for better maintainability or readability.
- Are either known or thought possibly to have a bug.

Tracking bugs should be associated with the TODO items, but they are also in the source so that they are visible to the Open Source community, which may not have access to the same bug database.

This is the format to be used:

```
// TODO[-Arch][-Platform][-CQ|-Throughput|-Cleanup|-Bug|-Bug?]: description of the
```

- One type modifier (CQ, Throughput, Cleanup, Bug or Bug?) must be specified.
- The -Arch and -Platform modifiers are optional, and should generally specify actual architectures in all-caps (e.g. AMD64, X86, ARM, ARM64), and then in Pascal casing for Platforms and architecture classes (e.g. –ARMArch, –LdStArch, –XArch, –Unix, –Windows).
- This list is not intended to be exhaustive.

Examples:

```
// TODO-LdStArch-Bug: Should regTmp be a dst on the node or an internal reg?
// Either way, it is not currently being handled by Lowering.

// TODO-CQ: based on whether src type is aligned use movaps instead.

// TODO-Cleanup: Add a comment about why this is unreached() for RyuJIT backen

// TODO-Arm64-Bug: handle large constants!  Probably need something like the A
// case above: if (arm_Valid_Imm_For_Instr(ins, val)) ...
```

7.1.6 Performance

Be sure to comment the performance characteristics (memory and time) of an API, class, sensitive block of code, line of code that looks simple but actually does something complex, etc.

7.2 File header comment

C and C++ source files (header files and implementation files) must include a file header comment at the beginning of the file that describes the file, gives the file owner, and gives some basic information about the purpose of the file, related documents, etc. The format of this header is as follows:

```
//
// Copyright (c) Microsoft. All rights reserved.
// Licensed under the MIT license. See LICENSE file in the project root for full l
//
// <summary of the purpose of the file, description of the component, overview of
//
```

Major components usually occupy their own file. The top of the file is a good place to document the design of that component, including any information that would be helpful to a new reader of the code. A reference can be made to an actual design and implementation document (specification), but that document must be co-located with the source code, and not on some server that is unlikely to remain active for as long as the source will live.

7.3 Commenting code blocks

Properly commented code blocks allow code to be scanned through and read like a book. There are a number of different commenting conventions that can be used for blocks of code, ranging from comments with significant whitespace to help visually distinguish major code segments to follow, down to single end-of-line comments to annotate individual statements or expressions. Choose the comment style that creates the most readable code.

Major blocks can be commented using the following convention (in each example, the "<comment>" line represents any number of lines with actual comment text):

```
<blank line>
//
// <comment>
//
<blank line>
<code>
```

Or:

```
<blank line>
// <comment>
<blank line>
<code>
```

Minor blocks can be commented using the following convention:

```
// <comment>
<code>
```

Beware, however, of creating a visually dense block of comments-and-code without whitespace that is difficult to read.

If the code line is short enough, and the comment is short enough, it can all be on the same line:

```
<code>  // <comment>
```

Major comments should be used to comment a body of code, and minor comments should be used to clarify the intent of sub-bodies within a main body of code.

Comments should be written with proper punctuation, including periods to end sentences. Be careful to check the spelling of your comments.

The following example illustrates correct usage of major and minor comments in a body of code:

```
CorElementType MetaSig::NextArgNormalized()
{
    // Cache where the walk starts.
    m_pLastType = m_pWalk;

    //
    // Now get the next element if one exists.
    //

    if (m_iCurArg == GetArgCount())
    {
        // We are done walking the entire signature.
        return ELEMENT_TYPE_END;
    }
    else
    {
        // Skip the current argument and go the next.
        m_iCurArg++;
        CorElementType mt = m_pWalk.PeekElemType(m_pModule,
                                                 m_typeContext);
        m_pWalk.SkipExactlyOne();
        return mt;
    }
}
```

7.4 Commenting variables

All global variables and C++ class data members must be commented at the point of declaration.

It is recommended that local variable declarations are also commented. One should not have to scan all uses of the variable to determine the exact meaning of the variable, including possible values, when it may or may not be initialized, etc. If the name of the variable is sufficient to describe the intent of the variable, then variable comments are unnecessary. However, do keep in mind that someone reading the code for the first time will not know all the rules you might have had in mind.

The following conventions should be used:

```
// <comment>
<variable declaration>
```

Or, for sufficiently concise comments:

```
<variable declaration> // <comment>
```

The following variable declarations provide an example of how to properly follow this convention when commenting variables:

```
class Thread
{
    // This is the maximum stack depth of allowed for any thread.
    // This can be set by a config setting, but might be overridden
    // if it's out of range.
    static int s_MaxStackDepth;

    bool m_fStressHeap; // Are we doing heap-stressing?
};
```

7.5 Commenting `#ifdefs`

Do specify the macro name in a comment at the end of the closing `#endif` of a long or nested `#if` / `#ifdef`.

```
#ifdef VIEWER
#ifdef MAC
...
#endif // MAC
#else // !VIEWER
...
#endif // !VIEWER
```

This is so, when you see a page of code with just the `#endif` somewhere in the page, but the `#ifdef` is somewhere off the top of the page, you don't need to page back to see if the `#ifdef` 'ed code is relevant to you; you can just look at the comment on the `#endif`. It also helps when there is nesting of `#ifdef` s, as above.

The comment on a `#else` should indicate the condition that will cause the following code to be compiled. The comment on a `#endif` should indicate the condition that caused the immediately preceding code block to be compiled. In both cases, don't simply repeat the condition of the original `#if` – that does help with matching, but doesn't help interpret the exact condition of interest.

Right:

```
#if defined(FOO) && defined(BAR)
    ...
#endif // defined(FOO) && defined(BAR)
#ifdef FOO
    ...
#else // !FOO
    ...
#endif // !FOO
```

Wrong:

```
#ifdef FOO
    ...
#else // FOO
    ...
#endif // FOO
```

Do comment why the conditional `#ifdef` is needed (just as you might comment an `if` statement).

A comment for code within an `#ifdef` should also appear within the `#ifdef`. For example:

Right:

```
#ifdef _TARGET_ARM_
    // This case only happens on ARM...
    if (...)
        ...
#endif // _TARGET_ARM_
```

Wrong:

```
// This case only happens on ARM...
#ifdef _TARGET_ARM_
```

```
    if (...)
        ...
#endif // _TARGET_ARM_
```

8 Naming Conventions

Names should be sufficiently descriptive to immediately indicate the purpose of the function or variable.

8.1 General

It is useful for names to be unique, to make it easier to search for them in the code. For example, it might make sense for every class to implement a debug-only `dump()` function. It's a simple name, and descriptive. However, if you do a simple textual search ("grep" or "findstr", or "Find in Files" in Visual Studio) for "dump", you will find far too many to be useful. Additionally, Visual Studio IntelliSense often gets confused when using "Go To Reference" or "Find All References" for such a common word that appears in many places (especially for our imprecise IntelliSense projects), rendering IntelliSense browsing also less useful.

Functions and variables should be named at their level-of-intent.

Good:

```
int ClientConnectionsRemaining[MAX_CLIENT_CONNECTIONS + 1];
```

Bad:

```
int Connections[MAX];
```

Do not use negative names, as thinking about the negation condition becomes difficult.

Good:

```
bool isVerificationEnabled,  allowStressHeap;
```

Bad:

```
bool isVerificationDisabled, dontStressHeap;
```

8.2 Hungarian or other prefix/postfix naming

We do not follow "Hungarian" naming, with a detailed set of name prefix requirements. We do suggest or require prefixes in a few cases, described below.

- Global variables should be prefixed by "g_". (Note that the JIT has very few global variables.)
- Non-static C++ class member variables should be prefixed by "m_".
- Static C++ class member variables should be prefixed by "s_".

Two common Hungarian conventions that we do not encourage are:

- Prefixing boolean variables by "f" (for "flag"). Instead, consider using an appropriate verb prefix, such as "is" or "has". For example, `bool isFileEmpty`.
- Prefixing pointer variables by "p" (one "p" for each level of pointer). There may be situations where this helps clarity, but it is not required.

It is often helpful to choose a short, descriptive prefix for all members of a class, e.g. "lv" for local variables, or "emit" for the emitter. This short prefix also helps make the name unique, and easier to find with "grep". Thus, you might have "m_lvFoo" for a non-static member variable of a class that is using a "lv" prefix.

8.3 Macro names

All macros and macro constants should have uppercase names. Words within a name must be separated by underscores. The following statements illustrate some macro and macro constant names:

```
#define PAGE_SIZE   4096
#define CONTAINING_RECORD(_address, _type, _field)    \
            ((_type*)((LONG)(_address) -              \
                      (LONG)(&((_type*)0)->_field)))
```

The use of inline functions is strongly encouraged instead of macros, for type-safety, to avoid problems of double evaluation of arguments, and to ease debugging.

The first example here, PAGE_SIZE, should probably be written instead as:

```
const int g_PageSize = 4096;
```

which eliminates the need for a `#define` at all.

Macro parameter names should start with one leading underscore. No other names (local variables names, member names, class names, parameters names, etc.) should begin with one leading underscore. This prevents problems from occurring where a macro parameter name accidentally matches a variable name at the point of macro expansion.

All macro parameter names should be surrounded by parentheses in the macro definition to guard against unintended token interactions.

All this being said, there still are some cases where macros are useful, such as the JIT phase list or instruction table, where data is defined in a header file by a series of macro functions that must be defined before #including the header file. This allows the macro function to be defined in different ways and the header file to be included multiple times to create different definitions from the same data. Look, for example, at instrsxarch.h. (Unfortunately, this technique confuses Visual Studio Intellisense.)

8.4 Local variables

Local function variables should be named using camelCasing:

- Multiple words should be concatenated directly, without using underscores in between.
- The first letter of all words (except the first) should be upper case. The other letters should be lower case.
- The first letter of the first word should be upper case if there a (lower-case) prefix. Otherwise it should be lower case.
- Acronyms should be treated as words, and only the first letter may be upper case.

The following variable declarations illustrate variable names that adhere to this convention:

```
CorElementType returnType;       // simple camelCasing
TypeHandle     thConstraintType; // constraint for the type argument
MethodDesc*    pOverridingMD;    // the override for the given method
```

8.5 Global variables

Global variables follow the same rules as local variable names, but should be prefixed by a "g_". The following global variable declarations illustrate variable names that adhere to this convention:

```
EEConfig*   g_Config;            // Configuration manager interface
bool        g_isVerifierEnabled; // Is the verifier enabled?
```

8.6 Function parameters

Function parameters follow the same rules as local variables

```
int Point(int x, int y) : m_x(x), m_y(y) {}
```

8.7 Non-static C++ member variables

Non-static C++ member variables should follow the same rules as local variable names, but should be prefixed by "m_".

```
class MetaSig
{
    // The module containing the metadata of the signature blob
    Module*      m_Module;

    // The size of the signature blob in bytes. This is kSizeNotSpecified if
    // the size is not specified.
    UINT32       m_SigSize;

    // This contains the offsets of the stack arguments.
    // It is valid only after the entire signature has been walked.
    // It contains the offset of only the first few arguments.
    short        m_StackOffsets[MAX_CACHED_SIG_SIZE + 1];
};
```

8.8 Static member variables

Static C++ member variables should follow the same rules as non-static member variable names, but should be prefixed by "s_" instead of "m_".

```
class Thread
{
    static int         s_MaxThreadRecord;  // Set by config file
    static HANDLE      s_FakeThreadHandle; // Initialized after startup
};
```

8.9 Functions, including member functions

Functions should be named in PascalCasing format:

- Multiple words should be concatenated directly, without using underscores in-between.
- The first letter of all words (including the first) should be upper case. The other letters should be lower case.

The following C++ method declarations illustrate method names that adhere to this convention:

```
IExecutionEngine*  GetExecutionEngine();
void               SortArgs();
void               RecordStkLevel(unsigned stkLvl);
```

It is also acceptable, and in fact encouraged, to prefix such names with a "tag" related to the function name's component or group, such as:

```
unsigned            lvaGetMaxSpillTempSize();
bool                lvaIsPreSpilled(unsigned lclNum, regMaskTP preSpillMask);
```

This makes it more likely that the names are globally unique. The tag can start either with a lower or upper case.

8.10 Classes

C++ class names should be named in PascalCasing format. A "C" prefix should not be used (thus, use `FooBar`, not `CFooBar` as is used in some conventions). The following C++ class declaration demonstrates proper adherence to this convention:

```
class SigPointer : public SigParser
{
```

```
    ...
};
```

Interfaces should use a prefix of "I" (capital letter "i"):

```
class ICorStaticInfo : public virtual ICorMethodInfo
{
    ...
};
```

8.11 Enums

Enum type names are PascalCased, like function names.

Enum values should be all-caps, and prefixed with a short prefix that is unique to the enum. This makes them easier to "grep" for.

```
enum RoundLevel
{
    ROUND_NEVER     = 0,    // Never round
    ROUND_CMP_CONST = 1,    // Round values compared against constants
    ROUND_CMP       = 2,    // Round comparands and return values
    ROUND_ALWAYS    = 3,    // Round always

    COUNT_ROUND_LEVEL,
    DEFAULT_ROUND_LEVEL = ROUND_NEVER
};
```

> The JIT is currently very inconsistent with respect to enum type and element naming standards. Perhaps we should adopt the VM standard of prefixing enum names with "k", and using PascalCasing for names, with a Hungarian-style per-enum prefix. For example, kRoundLevelNever or kRLNever / kRLCmpConst.

9 Function Structure

The term "function" here refers to both C-style functions as well as C++ member functions, unless otherwise specified.

There are two primary file types:

- A header file. This is named using the .h suffix. It contains declarations, especially those declarations needed by other components. (Declarations only needed by a single file should generally be placed in the implementation file.)
- An implementation file. This contains function definitions, etc. It is named using the .cpp suffix.

Some code uses a third type of file, an "implementation header file" (or .hpp file) in which inline functions are put, which is `#include`ed into the appropriate .h file. We don't use that. Instead, we trust that our retail (ship) build type will use the compiler's whole program optimization feature (such as link-time code generation) to do cross-module inlining. Thus, we organize the implementations logically without worrying about inlining.

It is acceptable to put very small inline member function implementations directly in the header file, at the point of declaration.

9.1 In a header file

This is the format for the declaration of a function in a header file.

For argument lists that fit within the max line length:

```
[static] [virtual] [__declspec(), etc]
return-type   FunctionName(<type-name>* <argument-name>, ...) [const];
```

For argument lists that don't fit within the max line length, or where it is deemed to be more readable:

```
[static] [virtual] [__declspec(), etc]
return-type   FunctionName(<type-name>*    <argument-name>,
                           <type-name>     <argument-name>,
                           ...
                           <type-name>     <argument-name>) [const];
```

For multi-line function declarations, both argument type names and argument names should be aligned.

Most declarations with more than one argument will be more readable by using the second format.

Functions with no arguments should just use an empty set of parenthesis, and should not use `void` for the argument list.

Right:

```
void Foo();
```

Wrong:

```
void Foo( );    // Don't put space between the parentheses
void Foo(void); // Don't use "void"
```

All arguments can be on the same line if the line fits within the maximum line length.

Right:

```
void Foo(int i);
T Min<T>(T a, T b);
```

9.1.1 Comments for function declarations

Function declarations in a header file should have a few lines of documentation using single-line comments, indicating the intent of the prototyped function.

Detailed documentation should be saved for the Function Header Comment above the function definition in the implementation file. This makes it easy to scan the API of the class in the header file.

However, note that Visual Studio IntelliSense will pick up the comment that immediately precedes the function declaration, for use when calling the function. Thus, the declaration comment should be detailed enough to aid writing a call-site using IntelliSense.

```
class MetaSig
{
public:

    // Used to avoid touching metadata for mscorlib methods.
    MetaSig(MethodDesc*    pMD,
            BinderMethodID methodId);
```

```
    // Returns type of current argument, then advances the
    // argument index.
    CorElementType NextArg();

    // Returns type of current argument. Primitive valuetypes
    // like System.Int32 are normalized to the form "int32".
    CorElementType NextArgNormalized(UINT32* pSize);

    // Checks if the calling convention of pSig is varargs.
    static
    bool IsVarArg(Module*         pModule,
                  PCCOR_SIGNATURE pSig);
};
```

9.2 In an implementation file

Typically for each header file in the project there will be an implementation file that contains the function implementations and it is named using the .cpp suffix.

The signature of a function definition in the implementation file should use the same format used in the header file.

Generally the order of the functions in the implementation file should match to the order in the header file.

9.2.1 Function size

It is recommended that functions bodies (from the opening brace to the closing brace) are no more than 200 lines of text (including any empty lines and lines with just a single brace in the function body). A large function is difficult to scan and understand.

Use your best judgment here.

9.3 Function definitions

If the header file uses simple comments for the function prototypes, then the function definition in the implementation file should include a full, descriptive function header. If the header file uses a full function header comments for the function prototypes, then the function definition in the implementation file can use a few descriptive lines of comments. That is, there should be a full descriptive comment for the function, but only in one place. The recommendation is to place the

detailed comments at the definition site. One primary reason for this choice is that most code readers spend much more time looking at the implementation files than they do looking at the header files.

Note that for virtual functions, the declaration site for the virtual function should provide a sufficient comment to specify the contract of that virtual function. The various implementation sites should provide implementation-specific details.

Default argument values may be repeated as comments. Example:

```
void Foo(int i /* = 0 */)
{
    <function body>
}
```

Be careful to update all call sites when changing the default parameters of a function!

Static member functions must repeat the "static" keyword as a comment. Example:

```
// static
BOOL IsVarArg(Module*         pModule,
              PCCOR_SIGNATURE pSig)
{
    <function body>
}
```

9.4 Function header comment

All functions, except trivial accessors and wrappers, should have a function header comment which describes the behavior and the implementation details of the function. The format of the function header in an implementation file is as shown below.

Within the comment, argument names (and other program-related names) should be surrounded by double quotes, to emphasize that they are program objects, and not simple English words. This helps clarify those cases where a function argument might be parsed (by a human) in either way.

Any of the sections that do not apply to a method may be skipped. For example, if a method has no arguments, the "Arguments" section can be left out.

If you can formulate any assumptions as asserts in the code itself, you should do so. The "Assumptions" section is intended to encapsulate things that are harder (or impossible) to formulate as asserts, or to

provide a place to write a more easily read English description of any assumptions that exist, even if they can be written with asserts.

```
//------------------------------------------------------------------------
// <Function name>: <Short description of the function>
//
// Arguments:
//    <argument1-name> - Description of argument 1
//    <argument2-name> - Description of argument 2
//    ... as many as the number of function arguments
//
// Return Value:
//    Description of the values this function could return
//    and under what conditions. When the return value is a
//    described as a function of the arguments, those arguments
//    should be mentioned specifically by name.
//
// Assumptions:
//    Any entry and exit conditions, such as required preconditions of
//    data structures, memory to be freed by caller, etc.
//
// Notes:
//    More detailed notes about the function.
//    What errors can the function return?
//    What other methods are related or alternatives to be considered?

<function definition>
```

9.4.1 Example

The following is a sample of a completed function definition:

```
//------------------------------------------------------------------------
// IsVarArg: Checks if the calling convention of the signature "pSig" is varargs.
//
// Arguments:
//    pModule - The module which contains the metadata of "pSig".
//    pSig    - The pointer to the signature blob.
//
// Return Value:
//    true  - The calling convention of the method is varargs.
//    false - The method is not varargs, or there was some error.
//
```

```
// Assumptions:
//    The caller must have ensured that the format of "pSig" is
//    consistent, and that the size of the signature does not extend
//    past the end of the metadata blob.
//
// Notes:
//    Call-site signature blobs include ELEMENT_TYPE_SENTINEL.
//    This method does not check for the presence of the sentinel.

// static
BOOL MetaSig::IsVarArg(Module*         pModule,
                       PCCOR_SIGNATURE pSig)
{
    <function body>
}
```

9.5 Specific function information

9.5.1 Constructor with member initialization list

This is the format to use for specifying the member initialization list

```
ClassName::ClassName(<type-name>*    <argument-name>,
                     <type-name2000> <argument-name>,
                     ...
                     <type-name>     <argument-name>)
    : m_member1(val1)
    , m_member2(val2)
    , m_member3(val3)
{
    <function_body>
}
```

Note that the order of the initializers is defined by C++ to be member declaration order, and some compilers will report an error if the order is incorrect.

10 Local Variable Declarations

Generally, variables should be declared at or close to the location of their first initialization, especially for large functions. For small functions, it is fine to declare all the locals at the start of the method.

Each variable should be declared on a separate line.

It is preferable to provide an initialization of a variable when it is declared.

Variable names should be unique within a function. This is to make it easier to do a simple textual search for the declaration and all the uses of a name in a function, without worrying about scoping.

Variables that are conditionally assigned or passed as OUT parameters to a function must be declared close to their first use. Values that are passed by reference as OUT parameters should be initialized prior to being passed.

Variable names should generally be aligned, as are function parameter names, to improve readability. Variable initializers should also be aligned, if it improves readability.

```
char*   name      = getName();
int     iPosition = 0;
int     errorCode = 0;
FindIllegalCharacter(name, &iPosition, &errorCode);
```

10.1 Pointer declarations

For pointer declaration, there should be no space between the type and the `*`, and one or more spaces between the `*` and the following symbol being declared. This emphasizes that the `*` is logically part of the type declaration (even though the C/C++ parser binds the `*` to the name).

Right:

```
int*    pi;
int*    pq;
```

Wrong:

```
int *   pi;
int   * pi;
int    *pi;
int     *pi; // the alignment is on the name, not the *
int     *pi, *pq;
```

Each local pointer variable must be declared on its own line. Combined with the fact that most local variables will be declared and initialized on the same line, this naturally prevents the confusing syntax of mixing pointer and non-pointer declarations like:

```
int* piVal1, i2, *piVal3, i4;
```

For return types, there should be no space between the type and the `*`, and one or more spaces between the `*` and the function name.

Right:

```
Module*  GetModule();
```

Wrong:

```
Module  *GetModule();  // no space between * and function name
Module * GetModule();  // one space between type and *
```

For pointer types used in casts, there should be no space between the type and the `*`.

Right:

```
BYTE*  pByte = (BYTE*)pBuffer;
```

Wrong:

```
BYTE * pByte = (BYTE *)pBuffer; // one space between type and *
```

Reference variables should use similar spacing.

Right:

```
void Foo(const BYTE&  byte);
```

Double stars should appear together with no spaces.

Right:

```
int**    ppi;
```

Wrong:

```
int **   ppi;
int * *  ppi;
```

11 Spacing

11.1 Logical and arithmetic expressions

The following example illustrates correct spacing of parentheses for expressions:

```
if (a < ((!b) + c))
{
    ...
}
```

There is a space between the `if` and the open parenthesis. This is used to distinguish C statements from functions (where the name and open parenthesis have no intervening space).

There is no space after the open parenthesis following the `if` or before the closing parenthesis.

Binary operators are separated on both sides with spaces.

There should be parentheses around all unary and binary expressions if they are contained within other expressions. We prefer to over-specify parentheses instead of requiring developers to memorize the complete C++ precedence rules. This is especially true for `&&` and `||`, whose precedence relationship is often forgotten.

Complex expressions should be broken down to use local variables to express and identify the semantics of the individual components.

While `sizeof` is a built-in operator, it does not require a space between the `sizeof` and the open parenthesis. We require the argument to `sizeof` to be surrounded by parentheses.

Wrong:

```
if(a < b)            // no space after "if"
if (a&&b)            // no space around operator "&&"
if (a || ( b && c )) // space after "(" and before ")"
if (a < b + c)       // binary expression "b + c" is not parenthesized
x = sizeof f;        // "sizeof" requires parentheses: sizeof(f)
```

11.2 Continuing statements on multiple lines

When wrapping statements, binary operators are left hanging after the left expression, so that the continuation is obvious. The right expression is indented to match the left expression. Additional spaces between the parentheses may be inserted as necessary in order to clarify how a complex conditional expression is expected to be evaluated. In fact, additional spaces are encouraged so that it is easy to read the condition.

Right:

```
if ((condition1) ||
    ((condition2A) &&
     (condition2B)))

if ( (condition1) ||
     ( (condition2A) &&
       (condition2B) ) )

if ( ((condition1A) || (condition1B)) &&
     ((condition2A) || (condition2B))    )
```

Wrong:

```
if ((condition1)
    || ((condition2A)
    && (condition2B)))

if ((condition1)
|| ((condition2A)
 && (condition2B)))
```

11.3 Function call

When calling a function, use the following formatting:

```
Value = FunctionName(argument1, argument2, argument3);
```

There is no space between the function name and the open parenthesis.

There is no space between the open parenthesis and the first argument.

There is no space between the last argument and the closing parenthesis.

There is a space between every comma and the next argument.

There is no space between an argument and the comma following it.

If all the arguments won't fit in the maximum line-width or you wish to add per-argument comments, enter each argument on its own line. A line must either contain all the arguments, or exactly one argument. All arguments should be aligned. It is preferred that the alignment is with the first argument that comes immediately after the opening parenthesis of the function call. If that makes the call too wide for the screen, the first argument can start on the next line, indented one tab stop. This avoids potential line-length conflicts, avoids having to realign all the arguments each time the method-call expression changes, and allows per-argument comments.

Right:

```
Value = FunctionName(argument1,
                     (argument2A + argument2B) / 2,
                     *argument3, // comment about arg3
                     argument4);
```

Acceptable:

```
Value = FunctionName(
    argument1,
    (argument2A + argument2B) / 2,
    *argument3, // comment about arg3
    argument4);

Value = TrulyVeryLongAndVerboseFunctionNameThatTakesUpALotOfHorizontalSpace(
```

```
            argument1,
            (argument2A + argument2B) / 2,
            *argument3, // comment about arg3
            argument4);
```

The following are examples of incorrect usage of spaces.

Wrong:

```
Foo( i ); // space before first argument
Foo (i); // space after function name
Foo(i,j); // space is missing between arguments
```

For arguments that are themselves function calls you should consider assigning them to a new temporary local variable and passing the local variable.

There are a couple of reasons for this:

- The C++ language allows the compiler the freedom to evaluate the arguments of a function in any order, thus making the program non-deterministic when the compiler changes.
- When debugging the program, the step into procedure becomes tedious as you have to step-in/step-out of every nested argument call before stepping into the final call.

Right:

```
GenTreePtr asgStmt = gtNewStmt(asg, ilOffset);
*pAfterStmt = fgInsertStmtAfter(block, *pAfterStmt, asgStmt);
```

11.4 Arrays

Array indices should not have spaces around them.

Right:

```
int val = array[i] + array[j * k];
```

Wrong:

```
int val = array[ i ] + array[ j * k ];
```

12 Control Structures

The structure for control-flow structures like `if`, `while`, and `do-while` blocks is as follows:

```
// Comment about the upcoming control structure
<statement>
{
    <statement 1>
    ...
    <statement n>
}
```

The opening curly for the statement block is aligned with the preceding statement block or control-flow structure, and is on the next line. Statements within a block are indented 4 spaces. Curly braces are always required, with one exception allowed for very simple `if` statements (see below).

Each distinct statement must be on a separate line. While this improves readability, it also allows for breakpoints to easily be set on any statement, since the debuggers use per-line source-level breakpoints.

It is generally a good idea to leave a blank line after a control structure, for readability.

12.1 Braces for `if`

Braces are required for all `else` blocks of all `if` statements. However, "then" blocks (the true case of an `if` statement) may omit braces if:

- the "then" block is a single-line assignment statement, function call, `return` statement, or `continue` statement, *and*
- there is no `else` block.

Braces are required for all other `if` statements.

Right:

```
if (x == 5)
    printf("5\n");

if (x == 5)
{
    printf("5\n");
}

if (x == 5)
{
    printf("5\n");
}
else
{
    printf("not 5\n");
}

if (x != 5)
{
    if (x == 6)
        printf("6\n");
}

if (x == 5)
    return;

if (x == 5)
    continue;
```

Wrong:

```
if (x == 5)
    printf("5\n");
else
    printf("not 5\n");

if (x == 5)
    printf("5\n");
else
{
    printf("not 5\n");
    printf("Might be 6\n");
}
```

```
if (x != 5)
    if (x !=6)
        printf("Neither 5 or 6\n");

if (x != 5)
    for (int i = 0; i < 10; i++)
    {
        printf("x*i = %d\n", (x * i));
    }
```

12.3 Braces for looping structures

Similar spacing should be used for `for`, `while` and `do-while` statements. These examples show correct placement of braces:

```
for (int i = 0; i < 100; i++)
{
    printf("i=%d\n", i);
}

for (int i = SomeVeryLongFunctionName(); // Each part of the "for" is aligned
     SomeVeryComplexExpression();
     i++)
{
    printf("i=%d\n", i);
}

while (i < 100)
{
    printf("i=%d\n", i);
}

do
{
    printf("i=%d\n", i);
}
while(i < 100);
```

Note that a loop body *must* have braces; an empty loop body with just a semicolon cannot be used, as it can easily be missed when reading.

Right:

```
Foo* p;
for (p = start; p != q; p = p->Next)
{
    // Empty loop body
}
```

Right:

```
for (Foo* p = start; p != q; p = p->Next)
{
    // Empty loop body
}
```

Wrong:

```
Foo* p;
for (p = start; p != q; p = p->Next);
```

12.4 `switch` statements

For `switch` statements, each `case` label must be aligned to the same column as the `switch` (and the opening brace). The code body for each `case` label should be indented one level. Note that this implies that each case label must exist on its own line; do not place multiple case labels on the same line.

A default case is always required. Use the `unreached()` macro to indicate that it is unreachable, if necessary.

Local variables should not be declared in the scope defined by the `switch` statement.

A nested statement block can be used for the body of a `case` statement if you need to declare local variables, especially if you need local variable initializers. The braces should be indented one level from the `case` label, so as to not be at the same level as the `switch` braces. As with all statement blocks, the statements within the block should be indented one level from the opening and closing braces.

Fall-through between two case statements should be indicated with the `__fallthrough;` annotation. It should be surrounded by blank lines to make it maximally visible.

`case` labels (except the first) should generally be preceded by a blank line, to increase readability.

```
//
// Comment about the purpose of the switch
//
switch (Counter)
{
case 1:
    // Comment about the action required for case 1
    [code body]

    __fallthrough;

case 2:
    // Comment about the action required for case 1 or 2
    [code body]
    break;

case 3:
    {
        // Comment about the action required for case 3
        <local variable declaration>
        [code body]
    }
    break;

default:
    unreached();
}
```

12.5 Examples

The following skeletal statements illustrate the proper indentation and placement of braces for control structures. In all cases, indentations consist of four spaces each.

```
// static
int MyClass::FooBar(int    iArgumentOne /* = 0 */,
                    ULONG* pArgumentTwo)
{
    // Braces for "if" statements are generally preferred.
    if (iArgumentOne == 0)
    {
        return 0;
    }
```

```
    // All "for" loops must have braces.
    for (int counter = 0; counter < 10; counter++)
    {
        ThenBlock ();
    }

    // Simple "if" statements with assignments or function calls
    // in the "then" block may skip braces.
    if (counter == 0)
        ThenCode();

    // All if-else statements require braces.
    if (counter == 2)
    {
        counter += 100;
    }
    else
    {
        counter += 200;
    }

    return 0;
}
```

13 C++ Classes

The format for a C++ class declaration is as follows.

The labels for the `public`, `protected`, and `private` sections should be aligned with the opening brace.

The labels for the `public`, `protected`, and `private` sections should occur in this order.

This is the format to use:

```
//-----------------------------------------
//
// <Class name>: <Description of the class>
//
// Assumptions:
//    Is the class thread-safe? How does it handle memory allocation?
```

```
//    etc.
//
// Notes:
//    More detailed notes about the class.
//    Alternative or related types to compare
//
class <class name>
{
public:
    <optional blank line>
    <public methods>
    <public data members. Ideally, there will be none>

protected:
    <optional blank line>
    <protected methods>
    <protected data members>

private:
    <optional blank line>
    <private methods>
    <private data members>
};
```

Example:

```
//------------------------------------------
//
// MetaSig: encapsulate a meta-data signature blob. This could be the
// signature of functions, fields or local variables. It provides
// facilities to inspect the properties of the signature like
// calling convention and number of arguments, as well as to iterate
// the arguments.
//
// Assumptions:
//    The caller must have ensured that the format of the signature is
//    consistent, and that the size of the signature does not extend
//    past the end of the metadata blob.
//
// Notes:
//    Note that the elements of the signature representing primitive
//    valuetype can be accessed either in their raw form
//    (e.g., System.Int32) or in their normalized form (e.g., int32).
//    Note that parsing of generic signatures requires the caller to
//    provide the SigTypeContext to use to interpret the
```

```
//    type arguments.
//    Also look at SigPointer if you need to parse a single
//    element of a signature.
//
class MetaSig
{
public:

    //
    // Constructors
    //
    MetaSig(PCCOR_SIGNATURE szMetaSig,
            DWORD           cbMetaSig,
            Module*         pModule);

    // Used to avoid touching metadata for mscorlib methods.
    MetaSig(MethodDesc*    pMD,
            BinderMethodID methodId);

    //
    // Argument iterators
    //

    // Returns type of current argument, then advances the
    // argument index.
    CorElementType NextArg();

    // Returns type of current argument. Primitive valuetypes like
```

Preview | Code | Blame — Raw

```
    //
    // Helper methods
    //

    // Checks if the calling convention of pSig is varargs
    // two given strings.
    static
    BOOL IsVarArg(Module*         module,
                  PCCOR_SIGNATURE sig);

private:

    // The module containing the metadata of the signature blob.
    Module*      m_module;
```

```
    // The size of the signature blob. This is SizeNotSpecified if
    // the size is not specified.
    UINT32       m_sigSizeBytes;

    // This contains the offsets of the stack arguments.
    // It is valid only after the entire signature has been walked.
    // It contains the offset of only the first few arguments.
    short        m_stackOffsets[MAX_CACHED_SIG_SIZE + 1];
};
```

14 Preprocessor

14.1 Conditional compilation

Prefer `#if` over `#ifdef` for conditional compilation. This allows setting the macro to 0 to disable it. `#ifdef` will not work in this case, and instead requires ensuring that the macro is not defined.

One exception: we use `#ifdef DEBUG` for debug-only code (see Section FIXME).

Right:

```
#if MEASURE_MEM_ALLOC
```

Wrong:

```
#ifdef MEASURE_MEM_ALLOC
```

Note, however, that you must be diligent in only using `#if`, because `#ifdef FOO` will be true whether FOO is defined to 0 or 1!

`#if` and other preprocessor directives should not be indented at all, and should be placed at the very start of the line.

If you have conditional `#if` / `#ifdef` in the source, explain what they do, just like you would comment an `if` statement.

Minimize conditional compilation by defining good abstractions, partitioning files better, or defining appropriate constants or macros.

14.1.1 `#if FEATURE`

If a new or existing feature is being added or modified then use a `#define FEATURE_XXX` to both highlight the code used to implement this and to allow the JIT to be compiled both with and without the feature.

```
#define FEATURE_VALNUM_CSE  0 // disable Value Number CSE optimization logic
#define FEATURE_LEXICAL_CSE 1 // enable Lexical CSE optimization logic

#if FEATURE_VALNUM_CSE
void                Compiler::optValnumCSEinit()
```

Note that periodically we do need to go through and remove FEATURE_* defines that are always enabled, and will never be disabled.

14.1.2 Disabling code

It is generally discouraged to permanently disable code by commenting it out or by putting `#if 0` around it, in an attempt to keep it around for reference. This reduces the hygiene of the code base over time and such disabled code is rarely actually useful. Instead, such disabled code should be entirely deleted. If you do disable code without deleting it, then you must add a comment as to why the code is disabled, and why it is better to leave the code disabled than it is to delete it.

One exception is that it is often useful to `#if 0` code that is useful for debugging an area, but is not otherwise useful. Even in this case, however, it is probably better to introduce a COMPlus_* variable to enable the special debugging mode.

14.1.3 Debug code

Use `#ifdef DEBUG` for debug-only code. Do not use `#ifdef _DEBUG` (with a leading underscore).

Use the `INDEBUG(x)` macro (and related macros) judiciously, for code that only runs in DEBUG, to avoid `#ifdef`s.

Use the `JITDUMP(x)` macro for printing things to the JIT dump output. Note that these things will only get printed when the `verbose` variable is set, which is when `COMPlus_JitDump=*` or when `COMPlus_JitDump=XXX` and we are JITting function XXX; or when `COMPlus_NgenDump=*` or `COMPlus_NgenDump=XXX` and we are NGENing function XXX. Do not use `JITDUMP` for all output in a debug-only function that might be useful to call from the debugger. In that case, define a function that

uses `printf` (which is a JIT-specific implementation of this function), which can be called from the debugger, and invoke that function like this:

```
DBEXEC(verbose, MyDumpFunction());
```

A common pattern in the JIT is the following:

```
#ifdef DEBUG
    if (verbose)
        printf("*************** In genGenerateCode()\n");
#endif // DEBUG
```

This could be written on fewer lines as:

```
JITDUMP("*************** In genGenerateCode()\n");
```

However, the former is preferred because it is trivial to set an unconditional breakpoint on the "printf" that triggers when we are compiling the function that matches what COMPlus_JitDump is set to – a very common debugging technique. Note that conditional breakpoints could be used, but they are more cumbersome, and are very difficult to get right in windbg.

If many back-to-back JITDUMP statements are going to be used it is preferred that they be written using printf().

Wrong:

```
JITDUMP("  TryOffset:     0x%x\n", clause.TryOffset);
JITDUMP("  TryLength:     0x%x\n", clause.TryLength);
JITDUMP("  HandlerOffset: 0x%x\n", clause.HandlerOffset);
JITDUMP("  HandlerLength: 0x%x\n", clause.HandlerLength);
```

Right:

```
#ifdef DEBUG
if (verbose)
{
    printf("  TryOffset:     0x%x\n", clause.TryOffset);
    printf("  TryLength:     0x%x\n", clause.TryLength);
```

```
    printf("  HandlerOffset: 0x%x\n", clause.HandlerOffset);
    printf("  HandlerLength: 0x%x\n", clause.HandlerLength);
}
#endif // DEBUG
```

Always put debug-only code under `#ifdef DEBUG` (or the equivalent). Do not assume the compiler will get rid of your debug-only code in a non-debug build flavor. This also documents more clearly that you intend the code to be debug-only.

14.2 `#define` constants

Use `const` or `enum` instead of `#define` for constants when possible. The value will still be constant-folded, but the `const` adds type safety.

If you do use `#define` constants, the values of multiple constant defines should be aligned.

```
#define PREFIX_CACHE_DEFAULT_SIZE   16
#define PREFIX_CACHE_MAX_SIZE       (2 * 1024)
#define DEVICE_NAME                 L"MyDevice"
```

14.3 Macro functions

Expressions (except very simple constants) should be enclosed in parentheses to prevent incorrect multiple expansion of the macro arguments.

Enclose all argument instances in parentheses.

Macro arguments should be named with two leading underscores, to prevent their names from being confused with normal source code names, such as variable or function names.

14.3.1 Macro functions versus C++ inline functions

All macro functions should be replaced with a C++ inline function or C++ inline template function if possible. This allows type checking of arguments, and avoids the problem of macro-expansion of macro arguments.

14.3.2 Line continuation

All the `\` at the end of a multi-line macro definition should be aligned with each other.

There must be no `\` on the last line of a multi-line macro definition.

14.3.3 Multi-statement macro functions

Functional macro definitions with multiple statements or with `if` statements should use `do { <statements> } while(0)` to ensure that the statements will always be compiled together as a single statement block. This ensures that those who mistake the macro for a function don't accidentally split the statements into multiple scopes when the macro is used. Example: consider a macro used like this:

```
if (fCond)
        SOME_MACRO();
```

Wrong:

```
#define SOME_MACRO() \
    Statement1;      \
    Statement2;
```

Right:

```
#define SOME_MACRO() \
    do               \
    {                \
        Statement1;  \
        Statement2;  \
    } while(0)
```

The braces ensure the statement block isn't split. The `do { ... } while(0)` ensures that uses of the macro always end with a semicolon.

14.3.4 Control flow

Avoid using control flow inside of preprocessor functions. Since these read like function calls in the source it is best if they also act like function calls. The expectation should be that all arguments will get evaluated one time and we should avoid strange behavior such as only evaluating an argument if a prior argument evaluates to true or evaluating some argument multiple times.

14.3.5 Scope

Macros that require a pair of macros due to the introduction of a scope are strongly discouraged in the JIT. These do exist in the VM and the convention there is to have a _BEGIN and _END suffix at the end of a common all caps macro name.

```
#define PAL_CPP_EHUNWIND_BEGIN  {
#define PAL_CPP_EHUNWIND_END    }
```

14.3.6 Examples

```
#define MIN(_x, _y)              (((__x) < (__y)) ? (__x) : (__y))

#define CONTAINING_RECORD(_address, _type, _field) \
    ((_type*)((LONG)(_address) -                    \
              (LONG)(&((_type*)0)->_field)))

#define STRESS_ASSERT(_cond)                        \
    do                                              \
    {                                               \
        if (!(_cond) && g_pConfig->IsStressOn())    \
            DebugBreak();                           \
    }                                               \
    while(0)
```

15 Language Usage Rules

The following rules are not related to formatting; they provide guidance to improve semantic clarity.

15.1 C/C++ general

15.1.1 Casts

Instead of C-style casts, use `static_cast<>`, `const_cast<>` and `reinterpret_cast<>` for pointers as they are more expressive and type-safe.

15.1.2 Globals

Avoid global variables as they pollute the global namespace and require careful handling to ensure thread safety. Prefer static class variables.

15.1.3 `bool` versus `BOOL`

`bool` is a built-in C++ language type. `bool` variables contain the value `true` or `false`. When stored (such as a member of a struct), it is one byte in size, and `true` is stored as one, `false` as zero.

`BOOL` is a typedef, either of `bool` (from clrtypes.h) or `int` (from Windows header files), whose value is one of the `#define` macros `TRUE` (1) or `FALSE` (0).

Use `bool`. Only use `BOOL` when calling an existing API that uses it.

Right:

```
bool isComplete = true;
```

Wrong:

```
BOOL isComplete = TRUE;
```

15.1.4 `NULL` and `nullptr`

Use the C++11 `nullptr` keyword when assigning a "null" to a pointer variable, or comparing a pointer variable against "null". Do not use `NULL`.

Right:

```
int* p = nullptr;
if (p == nullptr)
    ...
```

Wrong:

```
int* p = NULL;
if (p == NULL)
    ...
int* p = 0;
if (p == 0)
    ...
```

15.1.5 Use of zero

Integers should be explicitly checked against 0. Pointers should be explicitly checked against `nullptr`. Types that have a legal zero value should use a named zero value, not an explicit zero. For example, `regMaskTP` is a register mask type. Use `RBM_NONE` instead of a constant zero for it.

Right:

```
int i;
int* p = Foo();
if (p == nullptr)
    ...
if (p != nullptr)
    ...
if (i == 0)
    ...
if (i != 0)
    ...
```

Wrong:

```
int i;
int* p = Foo();
if (!p)
    ...
if (p)
    ...
if (p == 0)
    ...
if (p != 0)
    ...
if (!i)
    ...
if (i)
    ...
```

15.1.6 Nested assignment

Do not use assignments within `if` or other control-flow statements.

Right:

```
int x = strlen(szMethodName);
if (x > 5)
```

Wrong:

```
if ((x = strlen(szMethodName)) > 5)
```

15.1.7 `if` conditions

Do not place constants first in comparison checks (unless that reads more naturally), as a trick to avoid accidental assignment in a condition, as assignment within a condition will be a compiler error in our builds.

Right:

```
if (x == 5)
```

Wrong:

```
if (5 == x)
```

15.1.8 `const`

Use of the `const` qualifier is encouraged.

It is specifically encouraged to mark class member function as `const`, especially small "accessors", for example:

```
var_types TypeGet() const { return gtType; }
```

15.1.9 Ternary operators

Ternary operators `?:` are best used to make quick and simple decisions inside function invocations. Don't use it as a replacement for the `if` statement. Note that putting individual statements on their own line makes it easy to set debugging breakpoints on them. Use of nested ternary operators is

strongly discouraged. Using it for simple assignment of a single condition is fine. It's recommended that the "then" and "else" conditions of the ternary operator do not have side-effects.

Right:

```
if (a == b)
{
    Foo();
}
else
{
    Bar();
}
```

Wrong:

```
(a == b) ? Foo() : Bar(); // top-level ?: disallowed
```

Acceptable:

```
x = (a == b) ? 7 : 9;
Foo((a == b) ? "hi" : "bye");
```

Wrong:

```
x = (a == b) ? ((c == d) ? 1 : 2) : 3; // nested ?: disallowed
```

15.1.10 Use of `goto`

The `goto` statement should be avoided.

If you *must* use `goto`, the `goto` label must be all-caps, with words separated by underscores. The label should be surrounded by empty lines and otherwise made very visible, with prominent comments and/or placing it in column zero.

Example:

```
case GT_LSH: ins = INS_SHIFT_LEFT_LOGICAL;  goto SHIFT;
case GT_RSH: ins = INS_SHIFT_RIGHT_ARITHM;  goto SHIFT;
case GT_RSZ: ins = INS_SHIFT_RIGHT_LOGICAL; goto SHIFT;

SHIFT:
```

You should think very hard about other ways to code this to avoid using a `goto`. One of the biggest problems is that the `goto` label can be targeted from anyplace in the function, which makes understanding the code very difficult.

15.2 Source file organization

The general guideline is that header files should not be bigger than 1000 lines and implementation files should not be bigger than 5000 lines of code (including comments, function headers, etc.). Files larger than this should be split up and organized in some better logical fashion.

A class declaration should contain no implementation code. This is intended to make it easy to browse the API of the class. Note that our shipping "retail" build uses Visual C++ Link Time Code Generation, which can perform cross-module inlining. It is acceptable to define small accessor functions in the class declaration, for simplicity.

Maintain clear visual separation and identification of "segments" of API, and in particular of the private area of declarations. Logical chunks of APIs should be separated with comments like this:

```
<blank line>
//
// Description of the following chunk of API
//
<blank line>
```

15.3 Function declarations

15.3.1 Default arguments

Avoid default arguments values unless the argument has very little semantic impact, especially when adding a new argument to an existing method. Avoiding default values forces all call sites to think about the argument value to use, and prevents call sites from silently opting into unexpected behavior.

15.3.2 Overloading

Never overload functions on a primitive type (e.g. `Foo(int i)` and `Foo(long l)`).

Avoid operator overloading unless the overload matches the "natural" semantics of the operator when applied to integral types.

15.3.3 Enums versus primitive parameter types

Use enums rather than primitive types for function arguments as it promotes type-safety, and the function signature is more descriptive.

Specifically, declare and use enum types with two values instead of boolean for function arguments – the enum conveys more information to the reader at the callsite.

Bad:

```
Foo(true);
Foo(false);
```

Good:

```
enum DuplicateSpecification
{
    DS_ALLOW_DUPS,
    DS_UNIQUE_ONLY
};
void Foo(DuplicateSpecification useDups);
Foo(DS_ALLOW_DUPS);
Foo(DS_UNIQUE_ONLY);
```

This is especially true if the function has multiple boolean arguments.

Bad:

```
Bar(true, false);
```

Good:

```
Bar(DS_ALLOW_DUPS, FORMAT_FIT_TO_SCREEN);
```

15.3.4 Functions returning pointers

Functions that return pointers must think carefully about whether a `nullptr` return value could be ambiguous between success with a `nullptr` return value and failure.

15.3.5 Reference arguments

Never use non-const reference arguments as the call-site has no indication that the argument may change. Const reference arguments may be used as they do not have the above problem, and are also required for operators.

15.3.6 Resource release

If you call a function to release a resource and pass it a pointer or handle, you must set the pointer to `nullptr` or handle to `INVALID_HANDLE`. This ensures that the pointer or handle will not be accidentally used in code that follows.

```
CloseHandle(hMyFile);
hMyFile = INVALID_HANDLE;
```

15.3.7 OUT parameters

Functions with OUT parameters must initialize them (e.g., to 0 or `nullptr`) on entry to the function. If the function fails, this protects the caller from accidental use of potentially uninitialized values.

15.4 STL usage

> JIT STL usage rules need to be specified.

15.5 C++ class design

15.5.1 Public data members

Do not declare public data members. Instead, public accessor functions should be exposed to access class members.

15.5.2 Friend functions

Avoid friend functions - they expose internals of the class to the friend function, making subsequent changes to the class more fragile. However, it is notably worse to make everything public.

15.5.3 Constructors

If you declare a constructor, make sure to initialize all the class data members.

15.5.4 Destructors

The JIT uses a specialized memory allocator that does not release memory until compilation is complete. Thus, it is generally bad to declare or require destructors and the calling of `delete`, since memory will never be reclaimed, and JIT developers used to never dealing with deallocation are also likely to omit calls to `delete`.

15.5.5 Operator overloading

Define operators such as `=`, `==`, and `!=` only if you really want and use this capability, and can make them super-efficient.

Never define an operator to do anything other than the standard semantics for built-in types.

Never hide expensive work behind an operator. If it's not super efficient then make it an explicit method call.

15.5.6 Copy constructor and assignment operator

The compiler will automatically create a default copy constructor and assignment operator for a class. If that is undesirable, use the C++11 delete functions feature to prevent that, as so:

```
private:
    // No copy constructor or operator=
    MyClass(const MyClass& info) = delete;
    MyClass& operator=(const MyClass&) = delete;
```

15.5.7 Virtual functions

An overridden virtual method is explicitly declared virtual for clarity.

Virtual functions have overhead, so don't use them unless you need polymorphism.

However, note that virtual functions are often a cleaner, clearer, and faster solution than alternatives.

15.5.8 Inheritance

Don't use inheritance just because it will work. Use it sparingly and judiciously, when it makes sense to the situation. Deeply nested hierarchies can be confusing to understand.

Be careful with inheritance vs. containment. When in doubt, use containment.

Don't use multiple implementation inheritance.

15.5.9 Global class objects

Never declare a global instance of a class that has a constructor. Such constructors run in a non-deterministic order which is bad for reliability, and they have to be executed during process startup which is bad for startup performance. It is better to use lazy initialization in such a case.

15.6 Exceptions

Exceptions should be thrown only on true error paths, not in the general execution of a function. Exceptions are quite expensive on some platforms. What is an error path is a subjective choice depending on the scenario.

Do not catch all exceptions blindly. Catching all exceptions may mask a genuine bug in your code. Also, on Win32, some exceptions such as out-of-stack, cannot be safely resumed without careful coding.

Use care when referencing local variables within the "catch" and "finally" blocks, as their values may be an undefined state if an exception occurs.

15.7 Code tuning for performance optimization

In general, code should be written to be readable first, and optimized for performance second. Don't optimize for the compiler! This will help to keep the code understandable, maintainable, and less prone to bugs.

In the case of tight loops and code that has been analyzed to be a performance bottleneck, performance optimizations take a higher priority. Talk to the performance team if in doubt.

15.8 Memory allocation

All memory required during the compilation of a method must be allocated using the `Compiler`'s arena allocator. This allocator takes care of deallocating all the memory when compilation ends, avoiding memory leaks and simplifying memory management.

However, the use of an arena allocator can increase memory usage and it's worth considering its impact when writing JIT code. Simple code changes can have a significant impact on memory usage, such as hoisting a `std::vector` variable out of a loop:

```
std::vector<int> outer; // same memory gets used for all iterations
for (...)
{
    std::vector<int> inner; // this will allocate memory on every iteration
                            // and previously allocated memory is simply wasted
}
```

Node based data structures (e.g linked lists) may benefit from retaining and reusing removed nodes, provided that maintaining free lists doesn't add significant cost.

The arena allocator should not be used directly. `Compiler::getAllocator(CompMemKind)` returns a `CompAllocator` object that wraps the arena allocator and supports memory usage tracking when `MEASURE_MEM_ALLOC` is enabled. It's best to use a meaningful memory kind (e.g. not `CMK_Generic`) but exceptions can be made for small allocations. `CompAllocator` objects are always pointer sized and can be freely copied and stored (useful to avoid repeated `CompMemKind` references).

The `new (CompAllocator)` operator should be preferred over `CompAllocator::allocate(size_t)`. The later is intended to be used only when constructors must not be run, such as when allocating arrays for containers like `std::vector`.

```
// typical object allocation
RangeCheck* p = new (compiler->getAllocator(CMK_RangeCheck)) RangeCheck(compiler);
// slightly shorter alternative
RangeCheck* p = new (compiler, CMK_RangeCheck) RangeCheck(compiler);
// allocate an array with default initialized elements
LclVarDsc* p = new (compiler->getAllocator(CMK_LvaTable)) LclVarDsc[lvaCount];
// use list initialization to zero out an array
```

```
unsigned* p = new (compiler->getAllocator(CMK_LvaTable)) unsigned[lvaTrackedCount]
// use CompAllocator::allocate to allocate memory without doing any initialization
LclVarDsc* p = compiler->getAllocator(CMK_LvaTable).allocate<LclVarDsc>(lvaCount);
// ... and construct elements in place as needed
new (&p[i], jitstd::placement_t()) LclVarDsc(compiler)
```

Note that certain classes (e.g. `GenTree`) provide their own `new` operator overloads, those should be used instead of the general purpose `new (CompAllocator)` operator.

`jitstd` container classes accept a `CompAllocator` object by implicit conversion from `CompAllocator` to `jitstd::allocator`.

Debug/checked code that needs to allocate memory outside of method compilation can use the `HostAllocator` class and the associated `new` operator. This is a normal memory allocator that requires manual memory deallocation.

15.9 Obsoleting functions, classes and macros

The Visual C++ compiler has support built in for marking various user defined constructs as deprecated. This functionality is accessed via one of two mechanisms:

```
#pragma deprecated(identifier1 [, identifier2 ...])
```

This mechanism allows you to deprecate pretty much any identifier. In particular it can be used to mark a macro as obsolete:

```
#define FOO(x) x
#pragma deprecated(FOO)
```

Attempts to utilize FOO will result in compiler warning C4995 being raised:

```
obs.cpp(18) : warning C4995: 'FOO': name was marked as #pragma deprecated
```

Note that this warning will fire at the point in the code where the macro is expanded, which may include instances where another macro is used which happens to utilize the deprecated macro (the warning will not fire at the definition of the outer macro). In order to correctly obsolete such a macro it may be necessary to refactor your code to avoid its use by any outer macros which are not being obsoleted.

Another obsoleting mechanism is:

```
__declspec(deprecated("descriptive text"))
```

This mechanism can be used with classes and methods. It cannot be applied to macros. It is more flexible than the #pragma mechanism since it provides a way to output additional information with the warning message and can be applied to a specific overload of a given method:

```
#ifdef _MSC_VER
__declspec(deprecated("This method is deprecated, use the version that takes a Sta
#endif
static Module* GetCallersModule(int skip);
```

Attempting to call the method annotated above will result in a C4996 warning being raised:

```
d:\dd\puclr\ndp\clr\src\vm\marshalnative.cpp(431) : warning C4996: 'SystemDomain::(
```

Code that legitimately still needs to use deprecated functionality (or is being grandfathered in as new functions are deprecated) can use the normal C++ mechanism to suppress the deprecation warnings:

```
#ifdef _MSC_VER
#pragma warning(push)
#pragma warning(disable:4996) // Suppress warning on call to
                              // deprecated method
#endif

Module* pModule = SystemDomain::GetCallersModule(1);

#ifdef _MSC_VER
#pragma warning(pop)
#endif
```

Note that all these techniques are specific to the Microsoft C++ compiler and must therefore be conditionally compiled out for non-Windows builds, as shown in the examples above.