Coding Guidelines

|  |  |  |
| --- | --- | --- |
| Date | Comment | Author |
| 04/03/2017 | Initial Document | Brajesh Kumar |
| 04/30/2017 | Added General Section and other changes | Brajesh Kumar |

This document must be read before starting development in the IDMS code. This document will be updated based on the changes in the language. IDMS code currently is based on C++ and C# primarily. But it is getting extended to use JavaScript, TypeScript and Java as well. This document is written after there is already good amount of code base, so it is divided into sections, where some rule needs to be followed for new code development only. C++ style part of the document is based on Google C++ style and C# is based on core [https://github.com/dotnet/corefx/blob/master/Documentation/coding-guidelines/coding-style.md](https://urldefense.proofpoint.com/v2/url?u=https-3A__github.com_dotnet_corefx_blob_master_Documentation_coding-2Dguidelines_coding-2Dstyle.md&d=DwMFAg&c=IV_clAzoPDE253xZdHuilRgztyh_RiV3wUrLrDQYWSI&r=Lw-CCo5maGqi-zfyz7uIhyQ5OxA89mVIqJ684wI4zmg&m=41qBZfW_CO_CSTUkJr-gfclXwO9Tbl3JnrjsVnvUFwk&s=1ka5iosIqNsATEjW9wD2savCIQc3tic-3H97lhnMyS0&e=) and CLR via C# book. JavaScript is based on JavaScript: The Good Parts book.

New Code in this document means, when you are adding a totally new code and changing any existing methods or classes.

Old Code means, when you are fixing a bug and so have to change a method or class.

# C++

## Header Files

In general, every .cpp file should have an associated .h file. There are some common exceptions,

such as unit tests and small .cpp files containing just a main() function.

Correct use of header files can make a huge difference to the readability, size and performance of

your code.

The following rules will guide you through the various pitfalls of using header files.

### Pragma Once (New Code)

Use pragma once over #ifndef #define include guards.

### Header File Dependencies (new code)

Don't use an #include when a forward declaration would suffice.

### Inline Functions (new code)

Define functions inline only when they are small, say, 10 lines or less.

### Function Parameter Ordering (new code, old code)

When defining a function, parameter order is: inputs, then outputs.

### Names and Order of Includes (new code, Old code)

Use standard order for readability and to avoid hidden dependencies: C library, C++ library,

other libraries' .h, your project's .h

#include "foo/public/fooserver.h" // Preferred location.

#include <sys/types.h>

#include <unistd.h>

#include <hash\_map>

#include <vector>

#include "base/basictypes.h"

#include "base/commandlineflags.h"

#include "foo/public/bar.h"

## Scoping

### Namespaces (new code)

Unnamed namespaces are allowed and even encouraged in .cc files, to avoid runtime

naming conflicts.

Do not use unnamed namespaces in .h files.

Do not declare anything in namespace std, not even forward declarations of standard library

classes. Declaring entities in namespace std is undefined behavior, i.e., not portable. To

declare entities from the standard library, include the appropriate header file.

You may not use a *usingdirective*

to make all names from a namespace available.

### Nested Classes (New code)

Although you may use public nested classes when they are part of an interface, consider a

namespace to keep declarations out of the global scope.

Do not make nested classes public unless they are actually part of the interface, e.g., a class that

holds a set of options for some method.

### Nonmember, Static Member, and Global Functions (New code, Old code)

Prefer nonmember functions within a namespace or static member functions to global functions;

use completely global functions rarely.

### Local Variables (New code, Old Code)

Place a function's variables in the narrowest scope possible, and initialize variables in the

declaration.

## Classes

### Explicit Constructors (New code, Old Code)

Use the C++ keyword explicit for constructors with one argument.

### Copy Constructors (New Code, Old Code) Exceptions allowed

Provide a copy constructor and assignment operator only when necessary. Otherwise, disable

them with DISALLOW\_COPY\_AND\_ASSIGN macro.

### Structs vs. Classes (New Code)

Use a struct only for passive objects that carry data; everything else is a class.

### Operator Overloading (New code) Exceptions allowed.

Do not overload operators except in rare, special circumstances.

### Access Control (New code, Old code)

Make data members private, and provide access to them through accessor functions as

Needed) .

IDMS constantly uses m\_memeberName as class member name and this will be continued. It is not encouraged to use variable type in the

A setter method must start with a Set prefix SetMemberName and getter method should be GetMemberName.

### Declaration Order (New code, Old code)

Use the specified order of declarations within a class: public: before private:, methods

before data members (variables), etc.

Within each section, the declarations generally should be in the following order:

Typedefs and Enums

Constants (static const data members)

Constructors

Destructor

Methods, including static methods

Data Members (except static const data members)

Friend declarations should always be in the private section.

Method definitions in the corresponding .cc file should be the same as the declaration order, as

much as possible.

### Write Short Functions

Prefer small and focused functions.

We recognize that long functions are sometimes appropriate, so no hard limit is placed on functions

length. If a function exceeds about 50 lines, think about whether it can be broken up without

harming the structure of the program.

### Smart Pointers (New code)

Use of Smart pointers is preferred over native pointer. Unique\_ptr and shared\_ptr is recommended.

Don’t use auto\_ptr , scoped\_ptr.

## Other C++ features

### Reference Arguments (New code)

All parameters passed by reference must be labeled const.

### Function Overloading (New Code)

Use overloaded functions (including constructors) only if a reader looking at a call site can get a

good idea of what is happening without having to first figure out exactly which overload is being

called.

### Default Arguments (New Code) Exceptions allowed

We do not allow default function parameters, except in a few uncommon situations.

### Variable-Length Arrays and alloca() (New code, Old Code)

We do not allow variablelength arrays or alloca().

### Casting (New Code)

Use C++ casts like static\_cast<>(). Do not use other cast formats like int y =

(int)x; or int y = int(x);.

IDMS allows using dynamic\_cast and it will be continued.

### Use of Const (Old Code, new Code)

We strongly recommend that you use const whenever it makes sense to do so.

IDMS will use const int\* foo format.

### Preprocessor Macros (New Code, Old Code)

Be very cautious with macros. Prefer inline functions, enums, and const variables to macros.

### 0 and NULL (New code, Old Code)

Use 0 for integers, 0.0 for reals, nullptr for pointers, and '\0' for chars.

### Sizeof

Use sizeof(*varname*) instead of sizeof(*type*) whenever possible.

### Boost (New Code, Old Code)

IDMS does use BOOST. But most of the good features are already supported in C++. If you really needed to use something, please get approval from me.

### C++11

IDMS already uses the features in C++11 extensively and it is recommended to use its features.

### Naming (new code)

ClassNames:- IDMS follows Classnames starting with “C”:- for Example CNetWorkerThread and also NetWorkerThread. The new convention will be not to prefix it with “C”.

Class Member Name:- m\_recloserClosed. There is no need for type of the variable information.

Class function Name:- SetRecloserOpen()

Struct doesn’t need to be start with S.

Enum should not start with “e”.

Typedef should not start with “T”:- TEventSubscriptionMap (Current Code is very inconsistent)

Local Variables:- use substationName; No type information is needed.

Constant :- Global variable should start with upper case K. Local variables should start with lower case k. (Current Code is inconsistent)

FunctionNames:- Functions should start with a capital letter and have a capital letter for each new word. No

underscores. AddTableEntry()

Macro:- all uppercase with underscore in between words DNOM\_API.

### FileNames (New Code, Old Code)

Use existing format all in lower case with no underscore in between.

### Comment Style (New Code, Old Code)

Use /\* \*/ for multi-line comments.

Use // for one line comment.

### Function Comments(New Code)

Every function should have a summary of the function at the top of the function, which clearly states the intention of the programmer.

### Spaces vs. Tabs (New Code, Old Code)

Use only spaces, and indent 4 spaces at a time.

### Function Declarations and Definitions (New code, Old Code)

Return type on the same line as function name, parameters on the same line if they fit.

### Conditionals (New Code)

Prefer no spaces inside parentheses. The else keyword belongs on a new line.

### Loops and Switch Statements (New Code)

Switch statements may use braces for blocks. Empty loop bodies should use {} or continue

### Pointer and Reference Expressions (New Code, Old Code)

No spaces around period or arrow. Pointer operators do not have trailing spaces.

### Return Values (New Code)

Do not needlessly surround the return expression with parentheses.

### Variable and Array Initialization (New Code, Old Code)

IDMS will use = for this.

### Preprocessor Directives (New Code)

The hash mark that starts a preprocessor directive should always be at the beginning of the line.

### Class Format (New Code)

class MyClass : public OtherClass

{

public: // Note the 1 space indent!

MyClass(); // Regular 2 space indent.

explicit MyClass(int var);

~MyClass() {}

void SomeFunction();

void SomeFunctionThatDoesNothing()

{

}

Int GetVar() const { return m\_Var; }

private:

bool SomeInternalFunction();

int m\_Var;

};

### Things to note:

### Any base class name should be on the same line as the subclass name, subject to the 120 column

### limit.

### The public:, protected:, and private: keywords should be indented one space.

### Except for the first instance, these keywords should be preceded by a blank line. This rule is

### optional in small classes.

### Do not leave a blank line after these keywords.

### The public section should be first, followed by the protected and finally the private

### section.

### Constructor Initializer Lists (New Code)

Constructor initializer lists can be all on one line or with subsequent lines indented four spaces.

### Namespace Formatting (New Code)

The contents of namespaces are not indented.

Namespaces do not add an extra level of indentation. For example, use:

namespace {

void foo() { // Correct. No extra indentation within namespace.

...

}

} // namespace

### Loops and Conditionals (New Code)

# 

if (b)

{

//Your code after 4 spaces.

}

else

{

//Your code after 4 spaces.

}

while (test)

{

//Your code after 4 spaces.

}

### Modern C++ C#

### General

1. We use [Allman style](http://en.wikipedia.org/wiki/Indent_style#Allman_style) braces, where each brace begins on a new line. A single line statement block also needs braces. (New Code)
2. Use four spaces of indentation (no tabs). (New Code)
3. Always specify the visibility, even if it's the default (i.e. private string \_foo not string \_foo). Visibility should be the first modifier (i.e. public abstract not abstract public). (New Code)
4. Namespace imports should be specified at the top of the file, outside of namespace declarations and should be sorted alphabetically. (New Code)
5. Avoid more than one empty line at any time. For example, do not have two blank lines between members of a type. (New Code)
6. Avoid spurious free spaces. For example avoid if (someVar == 0)..., where the dots mark the spurious free spaces. Consider enabling "View White Space (Ctrl+E, S)" if using Visual Studio, to aid detection.
7. If a file happens to differ in style from these guidelines (e.g. private members are named m\_member rather than \_member), the existing style in that file takes precedence. (Old Code)
8. Only use var when it's obvious what the variable type is (i.e. var stream = new FileStream(...) not var stream = OpenStandardInput()). (New Code, Old Code)
9. Use language keywords instead of BCL types (i.e. int, string, float instead of Int32, String, Single, etc) for both type references as well as method calls (i.e. int.TryParse instead of Int32.TryParse). (New Code)
10. Use TryParse over Parse(New Code, Old Code).
11. It is recommended to use Sealed class, if it not meant to be inherited. (New Code)

| **Identifier** | **Casing** | **Example** |
| --- | --- | --- |
| Namespace | Pascal | namespace System.Security { ... } |
| Type | Pascal | public class StreamReader { ... } |
| Interface | Pascal | public interface IEnumerable { ... } |
| Method | Pascal | public class Object {     public virtual string ToString();   } |
| Property | Pascal | public class String  {     public int Length { get; }   } |
| Field | Pascal | private Boolean \_updateIncident;  public field must be avoided. |
| Event | Pascal | public class Process  {     public event EventHandler Exited;   } |
| Field | Pascal | public class MessageQueue  {     public static readonly TimeSpan    InfiniteTimeout;   }   public struct UInt32  {     public const Min = 0;   } |
| Enum value | Pascal | public enum eFileMode  {     Append,     ...   } |
| Parameter | Camel | public class Convert  {     public static int ToInt32(string valueOf);   } |

1. The Code should have the regions in the following Order: (New Code)  
   region should be in line with the code, rather than providing more indentation for the same.

#public fields

#public properties

#constants

#protected fields

#protected properties

# internal fields

#internal properties

# private fields

#private properties

#public methods

#protected methods

#internal methods

#private fields

#private methods

#private

1. Method parameters should be passed in with the input parameters first, followed by out or ref parameters.

### Parameter and Return Type Guidelines (New Code)

When declaring a method’s parameter types, you should specify the weakest type possible,

preferring interfaces over base classes. For example, if you are writing a method that

manipulates a collection of items, it would be best to declare the method’s parameter by

using an interface such as IEnumerable<T> rather than using a strong data type such as

List<T> or even a stronger interface type such as ICollection<T> or IList<T>:

// Desired: This method uses a weak parameter type

public void ManipulateItems<T>(IEnumerable<T> collection) { ... }

// Undesired: This method uses a strong parameter type

public void ManipulateItems<T>(List<T> collection) { ... }

The reason, of course, is that someone can call the first method passing in an array object,

a List<T> object, a String object, and so on—any object whose type implements

IEnumerable<T>. The second method allows only List<T> objects to be passed in; it will

not accept an array or a String object. Obviously, the first method is better because it is

much more flexible and can be used in a much wider range of scenarios.

Naturally, if you are writing a method that requires a list (not just any enumerable object),

then you should declare the parameter type as an IList<T>. You should still avoid declaring

the parameter type as List<T>. Using IList<T> allows the caller to pass arrays and any

other objects whose type implements IList<T>.

Note that my examples talked about collections, which are designed using an interface architecture.

If we were talking about classes designed using a base class architecture, the concept

still applies. So, for example, if I were implementing a method that processed bytes from a

stream, we’d have this:

// Desired: This method uses a weak parameter type

public void ProcessBytes(Stream someStream) { ... }

// Undesired: This method uses a strong parameter type

public void ProcessBytes(FileStream fileStream) { ... }

The first method can process bytes from any kind of stream: a FileStream, a

NetworkStream, a MemoryStream, and so on. The second method can operate only on a

FileStream, making it far more limited.

On the flip side, it is usually best to declare a method’s return type by using the strongest

type possible (trying not to commit yourself to a specific type). For example, it is better to

declare a method that returns a FileStream object as opposed to returning a Stream object:

// Desired: This method uses a strong return type

public FileStream OpenFile() { ... }

// Undesired: This method uses a weak return type

public Stream OpenFile() { ... }

Here, the first method is preferred because it allows the method’s caller the option of treating

the returned object as either a FileStream object or as a Stream object. Meanwhile, the

second method requires that the caller treat the returned object as a Stream object. Basically,

it is best to let the caller have as much flexibility as possible when calling a method, allowing

the method to be used in the widest range of scenarios.

Sometimes you want to retain the ability to change the internal implementation of a method

without affecting the callers. In the example just shown, the OpenFile method is unlikely to

ever change its internal implementation to return anything other than a FileStream object

(or an object whose type is derived from FileStream). However, if you have a method that

returns a List<String> object, you might very well want to change the internal implementation

of this method in the future so that it would instead return a String[]. In the cases

in which you want to leave yourself some flexibility to change what your method returns,

choose a weaker return type. For example:

// Flexible: This method uses a weaker return type

public IList<String> GetStringCollection() { ... }

// Inflexible: This method uses a stronger return type

public List<String> GetStringCollection() { ... }

In this example, even though the GetStringCollection method uses a List<String>

object internally and returns it, it is better to prototype the method as returning an

IList<String> instead. In the future, the GetStringCollection method could change its

internal collection to use a String[], and callers of the method won’t be required to change

any of their source code. In fact, they won’t even have to recompile their code. Notice in this

example that I’m using the strongest of the weakest types.

### Properties (New Code, Old Code)

Properties must be used carefully. Using {get;set;} is not recommended, unless absolutely needed.

It is better to implement the get and set. In any case, getter property should not have much code. Lot of classes have reflection on getters for resolving queries and it slows down the query processing.

### Lambda/Anonymous methods (New Code, Old Code)

Lambda functions must be used very carefully, as it can create hidden performance issues, which is not obvious looking at the code. In any case, there should not be a nested lambda, unless it is absolutely necessary.

## **JavaScript**

## use strict

The first line of the JavaScript should start with ‘use strict’ option.

EqualTo (===) operator  
Always use (====) operator and (!===) for comparisons.

### With Statement

Must be avoided.

Eval

Must be avoided.

### Expresssion

The statement

Function foo ( ) {}

Means the same thing as

var foo = function foo( ) {};

The second form is preferred over the first one.

### Functions

Functions that are intended to be used with new should be given names with initial capital letters, and names with initial capital letters should be used only with constructor functions, that take the new prefix. Not using the new at all is recommended, if possible.

### Void

Must not be used.

### JSON Parsing

Always use JSON.parse method instead of eval.

### Global Function

There must be a single global variable for your application. It should be defined in all UPPERCASE.

### Function Variables

JavaScript does not support variable scoping, so all the function variables should be defined in the top of the function.

## General

### Performance

Be careful with nested “for loops”. If there are three levels of nesting, think about refactoring it. Any core function must never have three levels of nesting.

## Exploration Vs Exploitation

Spend more time in exploring a problem, rather than jumping into fix it immediately.

## Web Services

ADMS only support RESTful services and this should be followed when adding a new Web Service. There is no exception to this rule. Data flow in the WebService should not allow a notification to be pushed to ADMS. Notification is allowed from ADMS to other systems. ADMS typically resides in the higher security zone and notification to a higher security system is violating some of the misinterpreted NERC CIP rules. Getting the changes from a lower security zone be based on request/reply pattern. For example, when ADMS needs a list of calls from an external system, it uses a time-stamp to get the new calls.