macadamian collaboration innovation

Coding Conventions for C++ and Java applications

Table of Contents

Section Contents

Subscribe to our email list to be notified by email when there is a new Column.

Archive of past columns
A full list of our software development articles from previous weeks

Coding Conventions for C++ and Java
One of our most popular pages -- Coding conventions for C++ and Java, written by our Chief Architect and used by our developers.

Source Code Organization

Files and project organization

The name of files can be more than eight characters, with a mix of upper case and lower-case. The name of files should reflect the content of the file as clearly as possible.

As a rule of thumb, files containing class definitions and implementations should contain only one class. The name of the file should be the same as the name of the class. Files can contain more than one class when inner classes or private classes are used.

Special care should be given to the naming of header files because of potential conflicts between modules. If necessary, a module prefix could be added to the filename. For example, if two modules have a garbage collector class: Database and Window, the files could be named: "DBGarbageCollector.H" and "WINDOWGarbageCollector.H".

The following table illustrates the standard file extensions used.

Extension	Description
.C	C Source Files
.CPP	C++ Source files
.H	C/C++ Header files
.INL	C++ Inline function files
.IDL (.ODL)	Interface Description language
.RC	Resource Script
.Java	Java Source file

Header Files

Include Statements

Include statements should never refer to a header file in terms of absolute path. For example, the following statement:

```
#include "/code/TelephonyMgr/Include/TMPhoneLine.h"
```

is wrong, while:

```
#include "TMPPhoneLine.h"
```

is right provided that the makefile has the appropriate paths set-up for compilation. Also:

```
#include "../include/TMPhoneLine.h"
```

is also right if the project topology is a "relative" topology where include files could be gathered in common areas.

Multiple Inclusion of a header file

As you probably know, multiple inclusion of a header file can have unexpected and unwanted effects. To avoid this, guarding pre-compiler code must be added to every header file.

```
#ifndef _TMPhoneLine_H_

#define _TMPhoneLine_H_

... Rest of Header File ...

#endif // _TMPhoneLine_H_
```

To avoid the possibility of a naming conflict, DevStudio will often append a GUID to the filename in the #define statement. This is not a necessary practice unless it is used to avoid an existing conflict.

The use of the "#pragma once" directive is allowed and encouraged to optimize compilation times. However, it doesn't replace the #ifndef guarding block because it is a non-standard extension of Microsoft's compiler and therefore not portable. Ideally, every header file should look like this.

```
#pragma once

#ifndef _Filename_H_

#define _Filename_H_

... Rest of Header File ...

#endif // _Filename_H_
```

All header files must be self-sufficient, such that a module including the file does not need to explicitly include other header files required by the interface. The header file must include other header files depended upon.

Naming Conventions

Function Names

Class member functions follow Java conventional naming. The function name is a concatenated string of words with the first letter of all word capitalized except for the first one. For example: isMemberSet, printReport, consume.

Functions exported from DLLs, that are not in a class or a namespace should include an uppercase abbreviation of the module name. For example: DEBUGTrace or DBGTrace.

C++ Specific

A good name for a routine clearly describes everything the routine does. Here are guidelines for creating effective routine names. (*The guidelines are quoted from [MCCO01]. In this context, we should not consider procedures that only return an error code as functions. They should be seen a procedures.*)

For a procedure name, use a strong verb followed by an object. A procedure with functional cohesion usually performs an operation on an object. The name should reflect what the procedure does, and an operation on an object implies a verb-plus-object name. printReport(), calcMonthlyRevenues(), and repaginateDocument() are samples of good procedure names.

In object-oriented languages, you don't need to include the name of the object in the procedure name because the object itself is included in the call.

For a function name, use a description of the return value. A function returns a value, and the function should be named for the value it returns. For example, cos(), nextCustomerID(), printerReady(), and currentPenColor() are all good function names that indicate precisely what the functions return.

Avoid meaningless or wishy-washy verbs. Some verbs are elastic, stretched to cover just about any meaning. Routine names like handleCalculation(), performServices(), processInput(), and dealWithOutput() don't tell you what the routines do. At the most, these names tell you that the routines have something to do with calculations, services, input, and output. The exception would be when the verb "handle" was used in the specific technical sense of handling an event.

Sometimes, the only problem with a routine is that its name is wishy-washy, the routine itself might actually be well designed. If handleOutput() is replaced with formatAndPrintOutput(), you have a pretty good idea of what the routine does.

In other cases, the verb is vague because the operations performed by the routine are vague. The routine suffers from a weakness of purpose, and the weak name is a symptom. If that's the case, the best solution is to restructure the routine and any related routines so that they all have stronger purposes and stronger names that accurately describe them.

The properties of a class should be accessible through ***getter*** and/or ***setter*** methods. Those methods should always be declared like this:

```
public get();

public void set( a);
```

For boolean properties, the use of "is" can replace the "get":

```
public boolean is();
```

Examples:

```
Color getCurrentPenColor()

void setCurrentPenColor(Color c)

boolean isPrinterReady()

void setPrinterReady(boolean ready)
```

Class Names

Classes follow Java conventional naming. The name is a concatenated string of words with the first letter of all word capitalized. For example: FocusEvent, DeviceContext, Customer.

Interface names follow COM conventional naming. The name is a concatenated string of words with the first letter of all word capitalized. A capital "I" is used as a prefix. For example: IWindowModel, IUnknown.

Variable Names

Variable names are a concatenated string of words with the first letter of all word capitalized except for the first one. The name chosen should clearly represent the content of the variable. For example: windowHandle, eventConsumed, index.

All variables that are a member of a class should have a "m_" prefix. For example: m_windowHandle, m_eventConsumed, m_index.

Constant (static final for Java) variables make exception to all these rules. They should follow the same standard as C++ #define statements. Constant variables should be named using capitalized names separated by an underscore character ('_'). For example: MAX_ARRAY_LENGTH, SIZE_PROPERTY_NAME.

Source Documentation

When applicable, all source documentation should be in a format compatible with JavaDoc formatting. However, it is important that the formatting codes included in the comment blocks do not overwhelm the comment block. Don't forget that the comment block is meant to be read in the source first.

Inline comments should be made with the "//" comment style and should be indented at the same level as the code they describe. End of line comments should be avoided with the exception of function parameters.

Module Comments and Revision history

Module headers are block headers placed at the top of every implementation file. The block comment should contain enough information to tell the programmer if she reached her destination.

```
/*

** FILE: filename.cpp

**

** ABSTRACT:

**   A general description of the module's role in the

**   overall software architecture,  What services it

**   provides and how it interacts with other components.

**

** DOCUMENTS:

**   A reference to the applicable design documents.

**
```

```
** AUTHOR:
**   Your name here
**
 ** CREATION DATE:
**   14/03/1998
**
** NOTES:
**   Other relevant information
*/
```

Note that all the block comments illustrated in this document have no pretty stars on the right side of the block comment. This deliberate choice was made because aligning those pretty stars is a large waste of time and discourages the maintenance of in-line comments.

It is sometimes useful to include a history of changes in the source files. With all the new source control tools now available, this information is duplicated in the source control database.

If you choose, you can use a revision history block in your modules. However, only use the revision history block if you intend to maintain it. It is pretty much useless when it's not maintained properly.

Here's the preferred style for a revision history block:

```
/*
** HISTORY:
** 000 - Nov 91 - M. Taylor  - Creation
** 001 - Dec 91 - J. Brander - Insert validation for
**                             unitlength to detect
**                             buffer overflow
*/
```

Commenting Data Declarations

Comments for variable declarations describe aspects of the variable that the name can't describe.

Comment the units of numeric data. If a number represents lengths, indicate whether it is expressed in inches, feet, meters or kilometers. If its time, indicate whether it's expressed in elapsed seconds since 1-1-1980, milliseconds since the start of the program and so on.

Comment the range of allowable numeric values. If a variable has an expected range of values, document the expected range.

Document flags to the bit level. If a variable is used as a bit field, document the meaning of each bit.

Document global data. If global data is used, annotate each piece well at the point which it is declared. The annotation should indicate the purpose of the data and why it needs to be global.

Commenting Control Structures

The place before a control structure is usually a natural place to put a comment. If it is an if or a case statement, you can provide the reason for the decision and a summary of the outcome. If it is a loop, you can indicate the purpose of the loop.

Commenting Routines

Here are a few guidelines about commenting routines.

Describe each routine in one or two sentences at the top of the routine. If you can't describe the routine in a short sentence or two, you probably need to think harder about what it is supposed to do. It might be a sign that the design isn't as good as it should be. Don't be tempted to explain everything in the block header. Instead, add comments in the code close to the code that it is commenting. It will be much more tempting to maintain those comments if the code changes.

In the block header, indicate which variables are input and output variables. To help describe the interface of a method, attributes similar to the ones used in .IDL could be used. ([in], [out], [in, out]) In addition, the return of function should also be described.

Document variables where they are declared. If you're not using global data, the easiest way to document variables is to put comments next to their declarations.

Document interface assumptions. Like the fact that a variable has to be initialized before being passed.

Comment on the routine's limitations. If the routine provides a numeric result, document the accuracy of the result. If the computations are undefined under certain conditions, document the conditions. If you ran into gotchas during the development of the routine, document them too.

Document the routine's global effect. If the routine modifies global data, describe exactly what it does to the global data. Don't forget that global data is as much a part of your interface as parameters are.

Document the source of algorithms that are used. If you have used an algorithm from a book or magazine, document the volume and page number you took it from. If you developed the algorithm yourself, indicate where the reader can find the notes you've made about it.

Programming Conventions

Use of Macros

In the past, parameterized macros were used frequently instead of simple routines. This was done mostly for performance reasons. This is not necessary with compilers that are more modern because they can be replaced by inline routines.

Macros are very hard to debug because the compiler doesn't generate the proper symbols for it. Unless they are necessary for conditional compilation or a similar purpose, they should not be used and should be replaced by inline routines.

Constants and Enumerations

It is generally good practice to replace literal values (strings and numbers) by a more descriptive identifier. It is very frequent to see #define statements used to create those identifiers.

Instead of using a #define statement, constant variables should be used. Constant variables carry type information and are protected by the compiler from some dangerous forms of "implicit" translation. Furthermore, the debugger can show the value of a constant variable. It cannot show the value of a #define identifier.

For similar reasons, enumerated types should be used for defining groups of values that are related together. (Flags passed to a routine for example)

Use of return, goto and throw for flow control

According to [MCCO01]:

> *Computer scientists are zealous in their beliefs, and when the discussion turns to gotos, they get out their jousting poles, armor and maces, mount their horses and charge through the gates of Camelot to the holy wars.*

According to common programmer lore:

> *Each function must have a single entry and exit structure.*
> *The use of multiple return statements to return control from a function may only occur to handle error exits.*
> *The use of gotos should be avoided at all costs.*

Most programmers agree with these guidelines and they are good ideas in principle.

In practice however, it is not rare to see a routine that needs more than one exit structure. The major risk of using returns to prematurely leave a routine is to leave some resources behind. Cleaver use of classes makes it easier for resources to be picked-up in case of a premature exit. This is one of the ideas behind CStrings and smart pointers.

One simple way to avoid multiple exit points is to use a goto statement. This will be considered the only acceptable use of a goto statement. If your routine needs to terminate prematurely and has resources to clean-up, the use of a goto is allowed if it is used in the following manner:

```
HRESULT theRoutine()

{

    HRESULT theResult = S_OK;
```

```
        ? Some code here ?
        if (SomeExceptionalCondition)
        {
            theResult = S_FAILED;
            goto RoutineCleanup;
        }
        ? Some more code here ?
    RoutineCleanup:
        ? Free the resources used by the routine ?
        return theResult;
    }
```

In java, the same example can be written using the finally() clause:

```
    int theRoutine()
    {
      int theResult = S_OK;
      try
      {
        ? Some code here ?
        if (SomeExceptionalCondition)
        {
            throw new SuperException();
        }
        ? Some more code here ?
      }
      finally()
      {
        ? Free the resources used by the routine ?
      }
      return theResult;
    }
```

In the case of exceptional conditions, it is sometimes useful to simply throw an exception. If you want to use the throw statement, you have to exercise the same cautions as when using the return statement. Throw statements constitute an exit point of your routine and your have to make sure that all the resources that the routine uses are cleaned-up automatically. Fortunately, objects declared on the stack will be cleaned-up properly.

In addition, routines that throw exceptions must mention it in their documentation. If you are the caller of routines that can throw exceptions, it is important that you prepare for that eventuality. You can decide to catch the exception yourself and handle it. You can also decide to let it pass through. If you do, you must behave as if you were throwing the exception yourself and mention it in your documentation. For Java programs, in addition to mentioning the exception in your documentation, adding the *throws* clause is mandatory.

Never forget that exceptions should be used for exceptional conditions. They are a heavyweight technique for controlling flow. They shouldn't be used for "casual" flow control because they are slow.

If you are writing a routine that is accessible through a COM interface, you cannot throw exceptions as COM doesn't support it.

Pointers to objects declared on the stack will not be freed automatically when an exception is thrown.

Error Handling

Many programmers think that:

> *As a rule, all functions calls returning diagnostic data (i.e. indication of success, failure, time-out, error, etc.?) should be followed by error checking code.*
> *In certain situations, when no meaningful error handling may be devised, diagnostic return data can be ignored. Such cases, however, form the exception rather than the norm and have to be individually justified and commented.*

Again, this is a very good idea in principle. However, this can lead to two things:

- No one will write routines that send a return value.
- More than 90 percent or your program is going to be error handling code.

Let's try those guidelines instead.

A failure should never leave a class in an undefined state. The class can be put in a state where it can't do much but it should be a defined state. For example, if you are writing a file class, failure to open the physical file shouldn't put the class in limbo. The class can't do much else but it will not fail.

An error should be detected and handled if it affects the execution of the rest of a routine. For example, if you try to allocate a resource and it fails, this affects the rest of the routine if it uses that resource. This should be detected and proper action taken. However, if releasing a resource fails, it doesn't affect the rest of the routine? It can therefore be ignored. (What are you going to do about it anyway?)

In an error is detected in a routine, consider notifying your caller. If the error has a potential to affect your caller, it is important that the caller be notified. For example, the "Open" methods of a file class should return error conditions. Even if the class stays in a valid state and other calls to the class will be handled properly, the caller might be interested in doing some error handling of his own.

Don't forget that error handling code is code. It can also be defective. It is important to write a test case that will exercise that code.

Don't design a routine where error conditions are identified as a special case of an "out" parameter. Instead, use a specific error condition parameter. That will make the logic of the caller much simpler and easier to understand. For example, try and avoid methods that return null to identify a specific error case.

Style and Layout

Layout, like goto is a religious issue with most programmers. Layout is more like the aesthetic aspect of code and that is mostly a matter of personal preference.

However, one should not forget that the fundamental theorem of formatting is that good visual layout should show the logical structure of the program. A complete chapter of Code Complete [MCCO01] is dedicated to code layout.

Layout Styles

This document presents one layout style called the "Begin-end" . You can can add personal variations to the style but it is important that you stay consistent and that the layout style doesn't change through a single file.

Note: All "tabs" in the proposed layout are set to 2 spaces. The settings of your editor should be set to replace all tabs with spaces.

```
void checkSomething(
int    firstParameter,
string secondParameter)
{
  char currentChar;
  doSomething();
  while (condition)
  {
    doSomething();
    doSomethingElse();
    if (condition)
      doSomething();
    switch (condition)
    {
      case CASE_1:
        doSomething();
```

```
            break;
          case CASE_2:
          {
            doSomething();
            break;
          }
          default:
            doSomething();
        }
      }
    }
```

Complicated Expressions

For complicated expressions, separate conditions should be on separate lines.

For example:

```
    if ((?0' <= inputChar && inputChar <= ?9') || (?a' <= inputChar
    &&inputChar <= ?z') || (?A' <= inputChar && inputChar <= ?Z'))
    {
      doSomething(inputChar);
    }
```

Should be replaced by:

```
    if ( (?0' <= inputChar && inputChar <= ?9') ||
         (?a' <= inputChar && inputChar <= ?z') ||
         (?A' <= inputChar && inputChar <= ?Z') )
    {
      doSomething(inputCharput);
    }
```

Large Function Calls

When a routine has a large parameter list, or when its name is very long, typing it all in one single line makes the program hard to read. The preferred way to layout such a function is to align the parameters with the end of the function name to make it stand out.

```
   drawLine( Window.north,
             Window.south,
             Window.east,
             Window.west,
             currentWidth,
             currentHeight);
```

When the function name is very long or the variables passed as parameters have long names. The parameters should be aligned 2 characters passed the name of the method.

```
   theClientWindow.drawFilledPolyline(
                     Window.north,
                     Window.south,
                     Window.east,
                     Window.west,
                     currentWidth,
                     currentHeight,
                     normalBackgroundFillColor);
```

Testing/Debug Support

In this section, the term "non-trivial routine" Is used. There is no formal definition of a non-trivial routine. The definition is left to the programmer at coding time and to the reviewer at review time.

Class Invariant

According to [MCGR01]:

> *The class invariant is a statement about constraints on objects that belong to the class. These constraints should always be maintained by the implementation.*

In other words, the class invariant is a condition that always holds true for a class no matter what state it's in. For example: The invariant for a queue class might contain a clause that requires that the current size of the queue be between zero and the maximum size of the queue inclusive.

Every class should supply a function to perform a sanity check on the class that will verify if the class invariant holds.

Assertions and Defensive Programming

Each non-trivial routine in your classes should have the following parts before performing any actions.

Sanity check	The routine should call the sanity check method to verify that the class invariant still holds.
Parameter checking	The routine should verify that all the parameters passed to the routine are valid in their type, value and range. An invalid parameter that would disrupt the behavior of the routine should be caught at this point and terminate the routine.
Method Invariant	In addition to performing the sanity check for the class, the routine should make sure that the class is in a proper state for this routine to be called. For example, you can't remove an item from a list if it is empty.

All of these checks should use an ASSERT statement to complain loudly if something goes wrong. In a release build, those ASSERT statements should be compiled-out.

Don't use assertions to check error codes of a function. If a function can fail, you should handle the error with "real" error handling code, not an assertion. You can still use an assertion to complain loudly for debugging purposes.

Use assertions to guard pieces of code that should never logically be executed. Like virtual methods that you expect to be overwritten by a derived class or the default clause of a switch statement.

Don't forget that assertions are compiled-out of production code. Don't put anything else but conditions in ASSERT statements. Any code put into an assertion statement will not be executed in the production version of you program. For example, don't do:

```
ASSERT(pFile = fopen(...) != NULL);
```

How much defensive programming should be left in production code?

Remove code that hinders performance. In your debug build, special error checking code has the potential to reduce the performance of a system. If this is the case, it is important to remove that code from the production build.

Leave in code that checks for important errors. Decide which errors can slip through the cracks without too much effect. For example, a test that verifies that a string fits in a dialog is only a cosmetic check and can be removed. However, a test that verifies that states are valid for telephony events should stay in because it might result in dropped calls.

Leave in code that helps the program crash gracefully. If your routine performs validation on the parameters it receives and this helps your function degrade gracefully when parameters are wrong, that code should be left in.

See that the messages you leave in the code are friendly. Never forget that the messages you display for debugging and tracing might make it in production code.

Validation Tests

If possible, build tests into your subsystems instead of on top of them. If you know of a second algorithm that can test your first one (even if it much slower or takes more memory), code it and put it in debug code, running alongside with your production code.

If you optimize an algorithm in your system, the old algorithm can make a very good validation test. Have it run with your new algorithm and ASSERT that the results are the same.

You shouldn't worry too much about the performance of the system in this case because your validation test will be compiled-out of production code.

Conclusion

Glossary

Term	Description
Class Category	A related group of classes. Usually used to implement a feature or function of the system
COM	Component Object Model. Microsoft's Object technology
DevStudio	Microsoft's development environment.
GUID	Globally Unique Identifier

References

MAGU01	Writing Solid Code, Steve Maguire, Microsoft Press
MCCO01	Code Complete, Steve Mcconnel, Microsoft Press
MCGR01	Functional testing of classes, John D. McGregor Dept. of C.S. Clemson University

History

Date	Contributor	Comments
16/01/1998	Francis Beaudet	Creation
20/01/1998	Claude Monpetit	Added standard for function naming of property accessors. Added the mandatory use of the *throws* clause.
22/01/98	Fred Boulanger Stephane Lussier	Added the mention of the "spaces for tabs" note. Added the creation date to the module header. Clarified the portion about commenting parameters. Clarified the restriction for the "one class per file" rule.

Other Coding Conventions on the Web

Java Coding Standard by Doug Lea, Professor of Computer Science, State University of New York

Writing Robust Java Code - AmbySoft's Java Coding Standards

[HOME] [SYNDEO] [NEWS] [COMPANY] [SERVICES] [COLUMN] [CAREERS] [CONTACT US]

Macadamian Technologies Inc. - info@macadamian.com - 613.739.5976

To Comment on our Web Site - webmaster@macadamian.com