Draft: Work in Progress

C++ Coding Standards and Style Guide

(Mission Applications Branch – Code 583)

Originally Developed by Wendy Shoan and Linda Jun (Code 583)

Modified for the General Mission Analysis Tool (GMAT) Project

By

Linda Jun & Wendy Shoan (Code 583)
Last Updated: 2005/05/24

Table of Contents

1 Introduction

This document is based on the "C Style Guide" (SEL-94-003). It contains recommendations for C++ implementations that build on, or in some cases replace, the style described in the C style guide. Style guidelines on any topics that are not covered in this document can be found in the "C Style Guide." An attempt has been made to indicate when these recommendations are just guidelines or suggestions versus when they are more strongly encouraged.

Using coding standards makes code easier to read and maintain. General principles that maximize the readability and maintainability of C++ are:

- Organize classes using encapsulation and information hiding techniques.
- Enhance readability through the use of indentation and blank lines.
- Add comments to header files to help users of classes.
- Add comments to implementation files to help maintainers of classes.
- Create names that are meaningful and readable.

1.1 Purpose

This document describes the Mission Applications Branch's recommended style for writing C++ programs, where code with "good style" is defined as that which is

- Organized
- Easy to read
- Easy to understand
- Maintainable
- Efficient

1.2 Audience

This document was written specifically for programmers in the Mission Applications development environment, although the majority of these standards are generally applicable to all environments. This document is intended to help programmers to produce better quality C++ programs by presenting specific guidelines for using language features.

1.3 Interpretation

This document provides guidelines for organizing the content of C++ programs and files. The guidelines are intended to help programmers write code that can be easily read, understood, and maintained. This document also discusses how C++ code can be written more efficiently.

The term "method" means a function that is a member of a class.
The term "Class Interface" means a class declaration in the header file.
The term "Class Implementation" means a class definition in the source file.

2 Names

In General, choose names that are meaningful and readable. If the name is appropriate everything fits together naturally, relationships are clear, meaning is derivable, and reasoning from common human expectations works as expected.

- When confronted with a situation where you could use all upper case abbreviation, you can do so; or instead, you can use an initial upper case letter followed by all lowercase letters. The developer should be consistent in how he/she does this.

```
class FOVPanel
class UtcDate
openDVDPlayer();
exportHtmlSource();
InertialReferenceUnit  theIRU;
FineSunSensor          theFss;
```

- Avoid use of underscores.

2.1 Class Names

- Capitalize the first letter of each word.

- A GUI component class name should be suffixed by the parent component name.

```
class MainFrame : public Frame
class DisplayPanel : public Panel
```

- Exception classes should be suffixed with Exception.

```
InvalidEulerSequenceException
```

2.2 Class Library Names

- Prevent class name clashes by using namespaces.

- When few components of a namespace are used in a file, the code should avoid using 'using' clauses and should use the scope operator '::' instead; however, when there are many uses of namespace components, it is preferable to use a 'using' clause to avoid clutter in the code.

2.3 Class Instance Names

- For instances of classes, follow conventions for variables

5

2.4   Method / Function Names

- Every method and function performs an action, so the name should make clear what it does. Names should be verbs and written in mixed case starting with upper case.

  ```
  OutputCalibrationData()
  Normalize()
  ```

- Prefixes are sometimes useful:

  - ° Is/Has/Can   -   to ask a question about something and return bool type
  - ° Set/Get   -   to set/get a value
  - ° Initialize   -   to initialize an object
  - ° Compute   -   to compute something

- The name of the class should not be duplicated in a method name.

  ```
  Vector Normalize()  // NOT: Vector NormalizeVector()
  ```

- When coding from the formal specification, match names with the spec but use no underscores.

2.5   Method / Function Argument Names

- Use the same guideline as for variables.

- When passing a class, an argument can have the same name as its type. This is not required, however, and in some cases may even be cumbersome. In that case, the name should be succinct.

  ```
  void SetForceModel(ForceModel *forceModel)
  void SetForceModel(ForceModel *fm)
  ```

- When coding from a formal specification, match argument names with the spec if possible but use no underscores.

2.6   Namespace Names

- Use the same guideline as for class names. It is suggested to use the project name as a prefix for the namespace name.

  namespace GmatTimeUtil

2.7   Variables

6

- Variables should begin with a lowercase letter, with the first letter of each word (after the first) in the name capitalized.

```
double      flatteningCoefficient;
MaVector3   initialPosition;
```

- Add a comment to a variable declaration if the meaning is not clear from the variable name.

- Internal variables should be declared at the level at which they are needed. For example, if a variable is used throughout the procedure, it should be declared at the top of the procedure. If a variable is used only in a computational block, for example, it may be declared at the top of that block.

- Internal variable declarations should be commented well, if their meaning is not clear from the variable names (particularly useful is a comment about units; units may be included in the variable name as well, e.g. `initialPositionInKm`)

- The declaration of indices may be inside a for loop or above it. (If that variable is needed after the execution of the loop, it will need to be declared above, not inside, it.)

```
for (Integer i= 0, bool done = False; i < MAX_SIZE && ! done; i++)
{
    ...
}
```

Or

```
Integer  i;                    // loop counter
bool     done;                 // have we found the matching item?
for (i = 0; i < MAX_SIZE && ! done; i++)
{
    ...
}
```

- It is preferable to use the project-defined types instead of the built-in types for the loop indices (e.g. use Integer instead of int).

2.7.1 Pointer Variables

- Place the '*' with the variable name rather than with the type:

```
MAB::String *name = new MAB::String;
MAB::String *name1, address; // note, only name1 is a pointer
```

- Take care using pointer conversions, particularly conversion from a base type to a derived type

⇒ **For Portability**

7

- For a null pointer, use "NULL".

2.7.2 Reference Variables

- Put the '&' with the variable name rather than with the type.

```
MaString(const MaString &maString, unsigned int bufferLength = 0)
Quaternion Rate(const Vector3 &w) const;
```

- For class overloaded operators and methods returning a reference, put the '&' with the type.

```
const MaString& operator= (const GSSString &gssString);
const MaString& operator= (const char *string);
const MaSString& operator= (char ch);
char& operator[](unsigned int index);
const MaVector3& Normalize();
```

2.7.3 Global Variables

- Use of global variables should be avoided. Instead, use namespaces.

2.8   Type Names

- Type names should have the first letter of each word capitalized.

```
typedef unsigned int SystemType
typedef double RealType
typedef ArrayTemplate<RealType> RealArrayType;
typedef TableTemplate<RealType> RealTableType;
```

2.9   Enum Type Names and Enum Names

- Enum types should follow Class Name policy.

- Enum names should be declared using all caps and underscores between words.

```
enum DayName {SUNDAY, MONDAY, TUESDAY, WEDNESDAY, THURSDAY, FRIDAY, SATURDAY};
enum Colors {RED = 3; BLUE, DARK_BLUE, GREEN, DARK_GREEN, YELLOW = 7};
```

2.10 Constants / #define

- Constants should be declared using all **CAPS** and **underscores** between words.

```
const int MINIMUM_NUMBER_OF_BYTES = 4;
```

- The use of #define statements should be avoided. const variables or enumeration types should be used instead of constant macros. (An exception is when conditional debug is included - #define statements may be used then).

  use

```
const unsigned int MAX_NUMBER_OF_FILES  = 4;
```

  instead of

```
#define MAX_NUMBER_OF_FILES 4
```

2.11 Structure Names

- For structure names, follow conventions for class names with the word "Type"

```
struct TimeType
{
    IntType  year;
    IntType  month;
    IntType  day;
    IntType  hour;
    IntType  minute;
    RealType second;
};
```

- Use of classes is preferred to structs. However, if all data is public, a struct may be used.

- The developer may use structs to encapsulate global data (including exceptions)

```
struct AttitudeTypes
{
    static const RealType TEST_ACCURACY;       // value is 1.19209290E-07F
};
```

2.12 C Function Names

- For C functions use C style function names of all lower case letters with '_' as word delimiter.

```
get_best_fit_model()
load_best_estimate_model()
```

- There should be very few C functions used in a C++ program. They should just be used to interface between C++ and C code.

2.13 C++ File Names

- All C++ header files should have the suffix of .hpp

- Header files which contain code that is accepted by both C and C++ compilers should have the file name extension ".h".

- C++ source files should have the suffix .cpp

- With the exception of the .hpp or .cpp suffix, file names should match, as much as possible, the name of the class declared or defined within it.

    If class name is **AnalyticalModel**, the file names should be:

    ```
    AnalyticalModel.hpp
    AnalyticalModel.cpp
    ```

2.14 Generated Code File Names

- Don't change the naming convention for files generated by other programs (e.g. wxWidgets)

3 Formatting

Use of standard formatting makes code easier to read. The general principles for formatting various kinds of statements are as follows:

- Use blank lines to organize statements into paragraphs and to separate logically related statements.

- Limit the complexity of statements - break a complex statement into several simple statements if it makes the code clearer to read.

- Indent to show the logical structure of your code.

3.1 Variables

- It is preferable to declare only one variable per line in the code.

3.2 Brace {}

- Braces should be used for all blocks. The first brace should appear on the following line, lined up with the keyword.

```
for (j = 0; j < MAX_NUMBER_OF_ITERATIONS; j++)
{
    statement1;
    statement2;
    ...
}

class SolarSystemBody
{
    statement1;
    statement2;
    ...
};
```

3.3   Parentheses ()

- Always put ( ) around a condition.

- Put a space between a keyword and parentheses

3.4   Indentation

- Use three or four spaces (3 is strongly suggested) for optimum readability and maintainability.

- When the standard spaces indentation is unworkable, use some logical spacing.

- When there are several variable declarations listed together, line up the variables.

```
int                          initialByteCount = MINIMUM_NUMBER_OF_BYTES;
Earth                        theEarth;
ClientProvidedAngularVelocity   clientProvidedAngularVelocity;
Matrix33                     m;
```

3.5   Tab / Space

- **Do not** use **tabs**. Use spaces, as tabs are defined differently for different editors and printers.

- Use appropriate spacing to enhance the readability.

- Put one space after a comma and semicolons:

```
exp(2, x)
for (i = 0; i < n; ++i)
```

- Put one space around **assignment operators:**

```
c1 = c2
```

- Put space between keyword and parentheses:

```
if (   )
while (    )
```

- Always put a space around conditional operators:

```
z = (a > b) ? a : b;
```

- Do not put space before parentheses following function names:

```
z = exp(2, x)
```

- Do not put spaces between unary operators and their operands:

```
p->m       s.m    a[i]        a(i)
```

- Do not put space around the **primary operators: ->, ., [ ]**

```
++i -n      *p      &x
```

3.6   Blank Lines

- Use blank lines to create paragraphs in the code or comments to make the code more understandable.

3.7   Method/Function Arguments

- When all arguments for a function do not fit on one line, try to line up the first argument in each line.

```
void SomeFunction(unsigned int someCounter, double someScaleFactor,
                  int someOtherArgument,
                  const SolarSystemBody &solarSystemBody,
                  int theLastArgument);
```

- If the argument lists are still too long to fit on the line, you may line up the arguments with the method name instead.

3.8   If / If else

- Indent statements one level using braces. For a single statement use of braces is optional.

```
if (condition)
    statement;
else if (condition)
    statement;
else
```

```
        statement;


if (condition)
{
    statement 1;
    statement 2;
    ...
}
else if (condition)
{
    statement 1;
    statement 2;
    ...
}
else
{
    statement;
}
```

- It is recommended to use explicit comparisons.

  use

  ```
  if (theFile->EndOfData() != true)
  ```

  instead of

  ```
  if ( ! theFile->EndOfData())
  ```

- Always use braces for nested if statements.

3.9 Switch

- All *switch* statements should have a default case, which may be merely a "fatal error" exit.

- The default case should be last and does not require a break, but it is a good idea to put one there anyway for consistency.

- Falling through a case statement into the next case statement shall be permitted as long as a comment is included.

- Use the following format for *switch* statements: If you need to create variables put all code in a block

  ```
  switch (expression)
  {
      case aaa:
          statement[s]
          break;

      case bbb:          // fall through

      case ccc:
          {
  ```

```
        int v;
        statement[s]
        break;
    }

    default:
        statement[s]
        break;
}
```

3.10 For / *While*

- Indent statements one level using braces. For a single statement use of braces is optional.

```
for (condition)
    statement;

for (condition)
{
    statement1;
    statement2;
    ...
}

while (condition)
{
    statement[s]
}
```

3.11 *Break*

- A break statement can be used to exit an inner loop of a *for*, *while*, *do*, or *switch* statement at a logical breaking point rather than at the loop test.

```
while (condition)
{
    while (condition)
    {
        if (condition)
        {
            break;
        }
    }
}
```

3.12 Use of *goto*

- **Do not** use *goto*.

3.13 Use of *?:*

- Conditional statements are fine as long as they are not too complex.

- Put the condition in parentheses so as to set it off from other code.

```
(condition) ? statement1 : statement2;

(condition)
   ? long statement1
   : long statement2;
```

3.14 Return Statement

- Multiple return statements are allowed in a function, if it makes the code more efficient.

⇒ **For Efficiency**

- Using an expression (including a constructor call) in a return statement is more efficient than declaring a local variable and returning it (avoiding a copy-constructor and destructor call).

```
return Vector3((cosArgPer * cosRA - sinArgPer * sinRA * cosI),
               (cosArgPer * sinRA + sinArgPer * cosRA * cosI),
               (sinArgPer * sinI));
```

3.15 Maximum Characters per Line

- Lines should be **no more than 80** characters long.

4 Documentation

There are two main audiences for documentation:

- Class Users
- Class Implementors/Maintainers

Judiciously placed comments in the code can provide information that a person could not discern simply by reading the code. Comments can be added at many different levels.

- At the program level, you can include a **README** file that provides a general description of the program and explains its organization.

- At the file level, include a **file prolog** that explains the purpose of the file and provides other information.

- In the header file, include a **method prolog** for all pure virtual methods. Add useful comments for class implementators and maintainers.

- In the source file, include a **method prolog** for all methods, other than pure virtual ones. Add useful comments for class implementators and maintainers.

- Throughout the file, where data are being declared or defined, it is helpful to add comments to explain the **purpose of the variables**.

Use of automated process to extract comments from the code can save developers time in generating documentation. Use Doxygen http://www.doxygen.org to automatically extract comments from the code and make it available on line for everyone to use. Any comments to be included in the documentation should follow the JavaDoc convention to mark a comment block. Set JAVADOC_AUTOBRIEF = YES in the Doxygen configuration file. If JAVADOC_AUTOBRIEF is set to YES in the configuration file, then JavaDoc style comment blocks will automatically start a brief description which ends at the **first dot followed by a space or new line**. The detailed descriptions follow after. (See Appendix B for commonly-used Doxygen commands)

```
/**
 * brief description.
 * detailed description.
 */
```

You may use the following for a brief one line description:

```
/// one line brief description
```

Remember that comments beginning with '//' will NOT show up in the docuementation.

4.1 Header File Prolog

- All header files must begin with a header file prolog.

- PDL **should not** appear at the file level - it should be replaced by well-documented code.

- Since the main audience for the header is a user of the class (or utilities) and someone who may want to derive a class from this one, note any assumptions.

- Header file prolog should contain the following sections: Since GMAT uses CVS to control source code, change history section should not be kept. There is no reason to clutter up source files with duplicate information which is available from CVS.

  <CVS Keyword>
  <Class Name Banner>   [for a class header]
  <Project Name>
  <Legal Tag>
  <Author>
  <Created>
  <Class Description>
  <Note> - *optional*

16

- The **<CVS Keyword>** should have:

  ```
  $Header$
  ```

- The **<Class Name Banner>** is the name of the class in the following form (include the namespace name, if applicable):

  ```
  //---------------------------------------------------------------------------
  //                              Class Name
  //---------------------------------------------------------------------------
  ```

- The **<Project Name>** should have the following:

  ```
  GMAT: General Mission Analysis Tool
  ```

- The **<Legal Tag>** should have the following:

  ```
  **Legal**    [A postprocessor will be used to insert the appropriate legal
  statement before release of the software]
  ```

  It may also include current contract information.

- The **<Author>** should have the following:

  ```
  Author: Your Name
  ```

- The **<Created>** should have the following:

  ```
  Created: yyyy/mm/dd
  ```

- The **<Class Description>** should include general description of the class. Use the JavaDoc convention for marking a comment block.

  ```
  /**
   * class description.
   */
  ```

- The **<Note>** section should describe any information necessary for users of the class, including Requirement/Function Specification Reference. This field can be omitted if there are no notes for the user.

Example:

```
//$Header$
//---------------------------------------------------------------------------
//                              MAB::String
//---------------------------------------------------------------------------
// GMAT: General Mission Analysis Tool.
//
// **Legal**
//
// Author: Your Name
// Created: 2003/08/05
```

```
//
/**
 * Provides a basic character string type operations.
 *
 * @note Any notes here.
 */
//------------------------------------------------------------------------
```

4.2   Header File Pure Virtual Method/Function Prolog

- For pure virtual methods, include a method prolog (as described in Section 4.4) in the header file, focusing on what is expected for the derived classes' implementations.

4.3   Source File Prolog

- All source files must begin with a source file prolog.

- Since the main audience for the source file is a maintainer, focus commentary on issues related to development and maintenance of the code.

- Source file prolog format is the same as the header file prolog format (see **Header File Prolog** section).

4.4   Source File Method/Function Prolog

- All methods and global functions (that are not pure virtual) declared in the header must have prologs in the source file. The prologs should focus comments on development issues and maintenance of the code. The prolog for a method/function appears just before the methods/function's implementation.

- Each prolog should describe the method/function clearly and concisely as to its purpose, inputs, return value (if any), and possible exceptions or abnormal conditions.

  - The exception section should list exceptions thrown in the class. This field can be omitted if no exceptions thrown.

  - Notes about the ownership and deletion responsibilities for pointer arguments can be included in the argument description.

- The source file function prolog should contain the following information:

```
//------------------------------------------------------------------------
//   function signature
//------------------------------------------------------------------------
/**
 * brief description of this function.
 * detailed description of this function if any.
 *
```

18

```
 * @param  - if applicable
 * @return - if applicable
 * @exception - if applicable
 * @see - if applicable
 * @note - if applicable
 */
//----------------------------------------------------------------------------
```

Example:

```
//----------------------------------------------------------------------------
// MAB::Date (IntType year, IntType month, IntType day, IntType hour,
//           IntType minute, RealType second)
//----------------------------------------------------------------------------
/**
 * A constructor.
 * Constructs objects with split calendar format date and time.
 *
 * @param <year>    input year
 * @param <month>   input month of year
 * @param <day>     input day of month
 * @param <hour>    input hour
 * @param <minute>  input minute
 * @param <second>  input second
 *
 * @exception TimeRangeError when a date or time is out of the specified
 */              range
//----------------------------------------------------------------------------
MAB::Date::MAB::Date (IntType year, IntType month, IntType day, IntType hour,
                      IntType minute, RealType second)
{
    ...
}
```

4.5   Comments in General

- The C++ comment indicator(//) should be used exclusively, except where comments are intended for documentation. In that case, the comments should use Doxygen style.

- **Do not** include Program Design Language (PDL).   Instead, care should be taken to clearly, yet succinctly comment the code. i.e. put a block comment before each major section of code, document variable declarations where needed, refer the reader to specifications or documents for further information if appropriate, etc.

- Include units in comments for variable declarations, if units are not included in the variable name.

  Example:

  ```
  Vector3 initialPosition(0.0,0.0,0,0);  // initial position vector in km
  ```

- Line up comments for declared variables.

- In particular, when coding from a formal specifications document, make sure to include the reference in the file prolog; and if coding a specific algorithm, include a reference to the

source of the algorithm in the function prolog/epilog (and possibly in the file prologs as well).

- It is also useful when coding from a formal specification, to include commentary, at each step, specifying the step number or brief description from the specifications (this makes it easier for code-readers to follow). However, comments should not be too verbose or restate the obvious.

Example

```
// Compute precession (Vallado, Eq. 3-56)
Real zeta  = ( 2306.2181*tTDB + 0.30188*tTDB2 + 0.017998*tTDB3 )
              *RAD_PER_ARCSEC;
Real Theta = ( 2004.3109*tTDB - 0.42665*tTDB2 - 0.041833*tTDB3 )
              *RAD_PER_ARCSEC;
Real     z = ( 2306.2181*tTDB + 1.09468*tTDB2 + 0.018203*tTDB3 )
              *RAD_PER_ARCSEC;

// Compute trigonometric quantities
Real cosTheta = Cos(Theta);
Real cosz     = Cos(z);
Real coszeta  = Cos(zeta);
Real sinTheta = Sin(Theta);
Real sinz     = Sin(z);
Real sinzeta  = Sin(zeta);

// Compute Rotation matrix for transformations from FK5 to MOD
// (Vallado Eq. 3-57)
Rmatrix33  PREC;
...
```

5 Class

5.1 Class Declaration (Header File)

- Public data should not be used without overriding efficiency justification.

- Provide access methods (Get/Set) for data as needed for access by other classes or code (e.g., GetPosition(), SetRadius())

- Declare a destructor to be virtual if there is any possibility that a class could be derived from this class (particularly if there are any virtual methods declared for this class).

- Declare a method to be virtual only if necessary (as virtual functions are less efficient).

- When declaring a function, put its return type on the same line as the function name. However, if the return type is very long, it is preferable to put the method name on the next line, lined up with the list of methods.

```
int                GetParamCount();

virtual std::string  GetRefObjectName(const Gmat::ObjectType type) const;
```

20

```
virtual const StringArray&
                       GetRefObjectNameArray(const Gmat::ObjectType type);
```

- Include preprocessor commands to avoid multiple definitions of items in a header file. Capitalize the preprocessor command the same way as the class name for easier future text substitution.

    For the AnalyticalModel class:

    ```
    #ifndef AnalyticalModel_hpp
    #define AnalyticalModel_hpp
          ...
    // class definition or type definitions (etc.)
          ...
    #endif  // AnalyticalModel_hpp
    ```

5.1.1 Required Methods for a Class

- Always Include the following methods for each class, to avoid having the compiler declare one for you. Declare them private (and possibly unimplemented) to limit or disable usage.

    - default constructor
    - copy constructor
    - destructor
    - assignment operator

5.1.2 Class Method/Function Declaration Layout

- The class declaration should include **public, protected**, and (if applicable) **private** sections (in that order). Since the user of the class needs the class interface, not the implementation, it make sense to have the public interface first.

    Example:

    ```
    class SolarSystemBody
    {
    public:
          ...
    protected:
          ...
    private:
          ...
    };
    ```

5.1.3 Include

- Include statements must be located at the top of a file only.

21

- Include statements should be sorted and grouped. Sorted by their hierarchical position in the system with low level files included first.

- Use C++ libraries, instead of C libraries, whenever possible. For example,

  use   `#include <iostream>`
  instead of `#include <stdio.h>`

  ⇒ **Notes:**

  ° This guideline may be ignored in cases of optimization, where the C library routines are proved to be more efficient than the C++ library routines)

- For system Include files, put a comment explaining why a particular file was included.

- Header files should be included only where they are needed and whenever they are needed for clarity. i.e. the user should be able to easily follow the code to determine the origin of methods or variables used in the code.

- Wherever possible, extern declarations (of global data) should be contained in source files instead of header files.

- Use extern "C" if referencing C external variables for functions, when necessary.

- When extern data or constants are included in a header file, remember to compile and link the .cpp file(s) that contain the actual definitions of the constants or externs into your program.

⇒ **For Portability**

- You should also avoid using directory names in the include directive, since it is implementation-defined how files in such circumstances are found. Most modern compiler allow relative path names with / as separator, because such names has been standardized outside the C++ standard, for example in POSIX. Absolute path names and path names with other separators should always be avoided though.

  The file will be searched for in an implementation-defined list of places. Even if one compiler finds this file there is no guarantee that another compiler will. It is better to specify to the build environment where files may be located, since then you do not need to change any include-directives if you switch to another compiler.

```
#include "inc/MyFile.h"        // Not recommended
#include "inc\MyFile.h"        // Not portable
#include "/gui/xinterface.h"   // Not portable
#include "c:\gui\xinterf.h"    // Not portable
```

5.1.4 Inlining

- Be careful about inlining. If your compiler has an inlining switch, prefer the use of that to actually including methods' implementations in the header file. (Also, compilers are not up-to-speed on compiling when methods are inlined in the header - in some cases, a compiler will generate a larger program because of this).

- Inline member functions can be defined inside or outside the class definition. The second alternative is recommend. The class definition will be more compact and comprehensible if no implementation can be seen in the class interface.

```
class X
{
public:
    // Not recommended: function definition in class
    bool insideClass() const { return false; }
    bool outsideClass() const;
};

// Recommended: function definition outside class
inline bool X::outsideClass() const
{
    return true;
}
```

5.1.5 Class Header File Layout

- Header files should include items in this order:

    - CVS keyword
    - Class name banner
    - Header file prolog
    - Preprocessor #ifndef command
    - System include files
    - Application include files
    - Constant declarations
    - Class declaration
    - Non-member functions (global functions)
    - Preprocessor #endif command

Example:

```
//$Header$
//-----------------------------------------------------------------------------
//                              Class Name
//-----------------------------------------------------------------------------
//  Header File Prolog
//   ...
//   ...
//-----------------------------------------------------------------------------
#ifndef ClassName_hpp
#define ClassName_hpp

#includes ...
Constant declarations ...
```

23

```
class ClassName
{
public:
    ...
protected:
    ...
private:
    ...
};


//-------------------------------------
//  global declarations
//-------------------------------------
    ...
    ...

#endif // ClassName_hpp
```

5.2 Class Definition (Source File)

5.2.1 Constructors

- Do not do any real work in constructor. Initialize variables and do only actions that can't fail. Object instantiators must check an object for errors after construction.
- Avoid throwing exceptions from constructors.
- All member data should be initialized in a constructor, not elsewhere, whenever possible.

5.2.2 Exceptions

- Use exceptions for truly exceptional conditions, not for message passing (i.e. use exceptions when processing cannot continue without user action)

⇒ **For Efficiency**

- Catch exceptions by reference.

5.2.3 Class Method/Function Definition Layout

- Methods/functions should be defined in the order in which they appear in the class declaration.

- Always initialize all variables.

- The function signature, its return type, and the argument names in the definition (implementation) should match its declaration (prototype) exactly.

- When defining a function's implementation, a long **return type** may go on a line above the function name.

```
int AttitudeModelClass::GetModelNumber()
{
    ...
}

CosineMatrix
CosineMatrix::GetInverse() const
{
    ...
}
```

⇒ **For Efficiency**

- Minimize the number of constructor/destructor calls: this means minimize the number of local objects that are constructed; construct on returning from a method, rather than creating a local object, assigning to it, and then returning it; pass large objects by const reference; etc.

- Initialize member data in an **initialization list**. It is necessary that the order of the items in the initialization list match the order of declaration; also, initialize base class data first (if it is not already initialized in the base class code)

```
MaString::MaString(const char *string1, unsigned int len1,
                   const char *string2, unsigned int len2)
    :
  lengthD(len1 + len2),
  caseSensitiveD(true)
{
    ...
}
```

5.2.4 Class Source File Layout

- The source file should contain the following sections:

  - CVS keyword
  - Class name banner
  - Source file prolog
  - Source file method prolog followed by implementation
  - Include subunits if any

Example:

```
//$Header$
//------------------------------------------------------------------------
//                              Class Name
//------------------------------------------------------------------------
//  Source File Prolog
//  ...
//------------------------------------------------------------------------
```

```
#includes ...

//-----------------------------------
//  public methods
//-----------------------------------

//----------------------------------------------------------------------------
//  source file method prolog
//  ...
//----------------------------------------------------------------------------

     Implementation code ...
     ...

//-----------------------------------
//  protected methods
//-----------------------------------

//----------------------------------------------------------------------------
//  source file method prolog
//  ...
//----------------------------------------------------------------------------

     Implementation code ...
     ...

//-----------------------------------
//  private methods
//-----------------------------------

//----------------------------------------------------------------------------
//  source file method prolog
//  ...
//----------------------------------------------------------------------------

     Implementation code ...
     ...

//----------------------------------------------------------------------------
#include ClassSubunits.cpp   // If there is any
```

- Include preprocessor commands for .cpp files when necessary.

  ⇒ **Notes:**

  ° This should only be necessary for .cpp files containing **subunits.** When used, they should take the following form:

  ```
  #ifndef OrbitOutput_cpp
  #define OrbitOutput_cpp
      ...
      subunit implementations
      ...
  #endif    // OrbitOutput_cpp
  ```

6 Templates

26

Templates are in one respect very similar to an inline function. No code is generated when the compiler sees the declaration of a template; code is generated only when a template instantiation is needed.

A function template instantiation is needed when the template is called or its address is taken and a class template instantiation is needed when an object of the template instantiation class is declared.

The template specifier for a template class should be placed alone on the line preceding the "class" keyword or the return type for a function. Template parameters should be in upper case.

```
// template declaration
template<class T>
class ListTemplate
{
public:
    T front();
    ...
};

// template definition
template<class T>
T ListTemplate<T>::front()
{
    ...;
}
```

A big problem is that there is no standard for how code that uses templates is compiled. The compilers that require the complete template definition to be visible usually instantiate the template whenever it is needed and then use a flexible linker to remove redundant instantiations of the template member function. However, this solution is not possible on all systems; the linkers on most UNIX-systems cannot do this.

A big problem is that even though there is a standard for how templates should be compiled, there are still many compilers that do not follow the standard. Some compilers require the complete template definition to be visible, even though that is not required by the standard.

This means that we have a potential portability problem when writing code that uses templates.

=> We recommend that you to put the implementation of template functions in a separate file, a template definition file, and use conditional compilation to control whether that file is included by the header file. A macro is either set or not depending on what compiler you use. An inconvenience is that you now have to manage more files. Only inclusion of the header file should be needed.

```
#ifndef QueueTemplate_hpp
#define QueueTemplate_hpp

template <class T>
class QueueTemplate
{
    public:
        QueueTemplate();
```

```
        // ...
        void insert(const T& t);
};


//--------------------------------
// Template Definition
//--------------------------------
#ifndef EXTERNAL_TEMPLATE_DEFINITION
#include "QueueTemplate.cpp"
#endif

#endif
```

7 Program Files

- Files should all begin with a file prolog (see below).

- Organize a program into two types of files as follows:

    Header File (.hpp) - should contain:

    - A class declaration
    - Any global type declarations
    - Any exceptions
    - Any typedefs
    - Any includes for template files
    - ENUM type definitions

    Source File (.cpp) - should contain:

    - Method definitions (implementation)
    - Any global data definitions
    - Any Constant data values

- Organize header files by class (one class declaration per header file) or by logical grouping of functions (e.g. RealUtilities)

- The main procedure should reside in its own file.

- For source files which contain related functions (utilities, for example), follow guidelines for putting functions in some meaningful order.

- Do not use rows of asterisks to separate functions.

- There should be only one class per .hpp/.cpp pair.

8 Portability

- Use ANSI/ISO C++ whenever it is available.

28

- When optimizing, some thought must be given to portability issues.

- Consider optimizations right from the start, as it is much harder to go back and redesign or recode later.

- Pass "large" arguments (instances of classes or structs) to a function by **const reference** when the arguments don't need to be modified and pass as **reference** when they need to be modified

- Place typedefs for all common types (e.g. double, integer) in a central header file, accessible by all code, for easier portability to other platforms and to higher precision types.

9 Efficiency

- For efficiency, minimize the number of constructor/destructor calls: this means minimize the number of local objects that are constructed; construct on returning from a method, rather than creating a local object, assigning to it, and then returning it; pass large objects by const reference; etc.

- For efficiency, use exceptions only for truly exceptional conditions, not for message passing

- Use embedded assignments when they are proven to be more efficient than not using them.

```
while ((c = getchar()) != EOF)
{
    ...
}
```

10 Miscellaneous

10.1 Extern statements / External variables

- Avoid using extern statements in the header file. Whenever possible, the source files referencing the global data should "extern" the needed global data, so that reader will know which variables are declared external to that source file.

- Avoid declaring non-static external variables. Variables needed by more than one file should appear in a .cpp file and be externed in a source file.

10.2 Preprocessor Directives

- Include the following preprocessor directive in the main header file, accessed by all code

```
#ifndef GMAT_API
  #define GMAT_API
#endif
```

10.3 Mixing C and C++

- Include files which contain code that is accepted by both C and C++ compilers should have the file name extension ".h".

- Make the header files work correctly when they are included by both C and C++ files. If you start including a existing C header in new C++ files, fix the C header file to support C++ (as well as C), don't just `extern "c" { }` the C header file.

  In C header file:

```
#ifdef __cplusplus
extern "C"
{
#endif
int existingCfunction1(…);
int existingCfunction2(…);
#ifdef __cplusplus
}
#endif
```

10.4 CVS Keywords

- If CVS is used for configuration management, the top of every file should contain the following lines (or those agreed upon by the project/team):

  Example:

```
//$Header$
```

10.5 README file

- A README file should be used to explain what the program does and how it is organized and to document issues for the program as a whole. For example, a README file might include

  - All conditional compilation flags and their meanings.
  - Files that are machine dependent
  - Paths to reused components
  - History information about the current and previous releases
  - Information about existing major bugs and fixes

- A brief description of new features added to the system

10.6 Makefiles

- Makefiles are used on some systems to provide a mechanism for efficiently recompiling C++ code. With makefiles, the make utility recompiles files that have been changed since the last compilation. Makefiles also allow the recompilation commands to be stored, so that potentially long CC commands can be greatly abbreviated.

  The makefile

  - Lists all files that are to be included as part of the program.
  - Contains comments documenting what files are part of libraries.
  - Demonstrates dependencies, e.g., source files and associated headers using implicit and explicit rules.

10.7 Standard Libraries

- A standard library is a collection of commonly used functions combined into one file. Examples of function libraries include <iostream> which comprises a group of input/output functions and <math> which consists of mathematical functions. When using library files, include only those libraries that contain functions that your program needs. You may create your own libraries of routines and group them in header files.

- Use C++ standard libraries, instead of C libraries, whenever possible.

10.8 Use of Namespaces

The use of namespaces minimizes potential name clashes in C++ programs and libraries and eliminates the use of global types, variables, etc.

Clashable names include: external variable names, external function names, top-level class names, type names in public header files, class member names in public header files, etc. (Class member names are scoped by the class, but can clash in the scope of a derived class. Explicit scoping can be used to resolve these clashes.)

It is no longer necessary to have global types, variables, constants and functions if namespaces are used. Names inside namespaces are as easy to use as global names, except that you sometimes must use the scope operator. Without namespaces it is common to add a common identifier as a prefix to the name of each class in a set of related classes.

It is recommended not to place "using namespace" directives at global scope in a header file; instead place it in a source file. This can cause lots of magic invisible conflicts that are hard to track since it will make names globally accessible to all files that include that header, which is what we are trying to avoid. Inside an implementation file, using declarations and using directives are

less dangerous and sometimes very convenient. On the other hand, too-frequent use of the scope operator is not recommended. The difference between local names and other names will be more explicit, but more code needs be rewritten if the namespaces are reorganized.

10.9 Standard Template Library (STL)

- Use Standard Template Library components, when available.

10.10 Using the new Operator

- The specification for operator "new" was changed by the standardization committee, so that it throws an exception of type std::bad_alloc when it fails. Therefore, the code may catch this exception, rather than check for a NULL value. e.g.

```
#include <stdexcept>

int someFunction()
{
    try
    {
        SomeClass *someClassList = new SomeClass[size];
    }
    catch (std::bad_alloc &ex)
    {
        ...
    }
}
```

Appendix A C++ Code Examples

A.1 Example of a header file.

```
//$Header$
//------------------------------------------------------------------------------
//                                    A1Date
//------------------------------------------------------------------------------
// GMAT: General Mission Analysis Tool
//
// **Legal**
//
// Author: Linda Jun
// Created: 2003/08/05
//
/**
 * This class provides conversions among various ways representing A1 calendar
 * dates and times.
 */
//------------------------------------------------------------------------------
#ifndef A1Date_hpp
#define A1Date_hpp

#include "IntType.h"
#include "RealType.h"
#include "TimeTypes.h"

#include "Date.hpp"
#include "UtcDate.hpp"
#include "String.hpp"

class A1Date : public MAB::Date
{
public:
    A1Date ();
    A1Date (IntType year, IntType month, IntType day, IntType hour,
                IntType minute, RealType second);
    A1Date (IntType year, IntType doy, IntType hour, IntType minute,
                RealType second);
    A1Date (IntType year, IntType month, IntType day, RealType mSecondsOfDay);
    A1Date (const MaString &timeString);
    A1Date (const MaA1Date &date);
    ~A1Date ();

    RealType  operator-  (const A1Date &date) const;
    A1Date    operator=  (const A1Date &date);
    A1Date    operator+  (const RealType seconds) const;
    A1Date&   operator+= (const RealType seconds);
    A1Date    operator-  (const RealType seconds) const;
    A1Date&   operator-= (const RealType seconds);

    RealType ModifiedJulianDate(ElapsedDays JDBias =
                                    TimeConstants::julianDateOf010541);
    Date ToUtcCDate();
protected:
private:
};
#endif // A1Date_hpp
```

33

A.2 Example of a source file.

```
//$Header$
//-------------------------------------------------------------------------------
//                              A1Date
//-------------------------------------------------------------------------------
// GMAT: General Mission Analysis Tool
//
// **Legal**
//
// Author: Linda Jun
// Created: 2003/08/05
//
/**
 * This class provides conversions among various ways representing A1 calendar
 * dates and times.
 */
//-------------------------------------------------------------------------------
#include "A1Date.hpp"
#include "Time.hpp"
#include "Date.hpp"


...
...


//---------------------------------
//   public methods
//---------------------------------

//-------------------------------------------------------------------------------
//   A1Date()
//-------------------------------------------------------------------------------
/**
 * Constructs A1Date objects with 0 second from reference (default constructor).
 *
 * @note Calls Time default constructor which creates an object with 0
 *        second from reference.
 */
//-------------------------------------------------------------------------------
A1Date::A1Date()
   :
   MAB::Date()
{
   Time t;
   *this = t.A1Split();
}


//-------------------------------------------------------------------------------
//   A1Date (IntType year, IntType month, IntType day, IntType hour,
//            IntType minute, RealType second)
//-------------------------------------------------------------------------------
/**
 * constructs A1Date objects with split calendar format of date and time.
 *
 * @param <year>   input year in YYYY.
 * @param <month>  input month of year.
 * @param <day>    input day of month.
 * @param <hour>   input hour.
 * @param <minute> input minute.
 * @param <second> input second and millisecond in ss.mmm.
 *
 * @exception TimeRangeError when a date or time is out of the specified range.
```

34

```cpp
 *
 * @note Assumes input date is in A1 time system.
 */
//-------------------------------------------------------------------------------
A1Date::A1Date (IntType year, IntType month, IntType day, IntType hour,
                        IntType minute, RealType second)
    :
    MAB::Date(year, month, day, hour, minute, second)
{
}


...
...


//-------------------------------------------------------------------------------
//  A1Date (const A1Date &date)
//-------------------------------------------------------------------------------
/**
 * A copy constructor.
 */
//-------------------------------------------------------------------------------
A1Date::A1Date (const A1Date &date)
    :
    MAB::Date(date)
{
}


//-------------------------------------------------------------------------------
//  RealType operator- (const A1Date &date) const
//-------------------------------------------------------------------------------
/**
 * Computes the time offset between two A1Date objects.
 *
 * @param <date> date object to be subtracted from "this" A1Date object.
 */
//-------------------------------------------------------------------------------
RealType A1Date::operator- (const A1Date &date) const
{
    RealType offset;

    Time t1(year, month, day, secondsOfDay);
    Time t2(date.year, date.month, date.day, date.secondsOfDay);

    offset = t1 - t2;
    return offset;
}

//-------------------------------------------------------------------------------
//  A1Date operator= (const A1Date &date)
//-------------------------------------------------------------------------------
/**
 * Assignment operator.
 *
 * @param <date> A1Date object whose values to use to set "this" A1Date object.
 *
 * @return A1Date object.
 */
//-------------------------------------------------------------------------------
A1Date A1Date::operator= (const A1Date &date)
{
    year = date.year;
    month = date.month;
    day = date.day;
```

35

```
        secondsOfDay = date.secondsOfDay;
        return *this;
}


...
...

//-------------------------------------------------------------------------------
//   RealType ModifiedJulianDate(ElapsedDays jdBias)
//-------------------------------------------------------------------------------
/**
 * Computes days elapsed since 0 hour of UTC julian date of reference.
 *
 * @param <jdBias> offset between modifed julian days and julian days.
 *
 * @return A1 modified julian days.
 */
//-------------------------------------------------------------------------------
RealType A1Date::ModifiedJulianDate(ElapsedDays jdBias)
{
    RealType          daysElapsed;
    ElapsedDays julianDays;

    // compute A1 julian days
    julianDays = MAB::Time::JulianDay(year, day);

    daysElapsed = (RealType)(julianDays - JDBias) - 0.5 +
        secondsOfDay / TimeConstants::secondsPerDay;
    return daysElapsed;
}


//-------------------------------------------------------------------------------
//   Date ToUtcDate()
//-------------------------------------------------------------------------------
/**
 * Converts to UTC date.
 *
 * @return UTC date.
 *
 * @note The two time systems label time differently.  At any given moment,
 *        the A.1 system is several seconds ahead of the UTC system.
 *        This offset is constant between leap insertions.  For example,
 *        the instant of time labeled July 1, 1992, 12:00:27.0343817 in the
 *        A.1 system will be labeled July 1, 1992, 12:00:00 (Noon) in the
 *        UTC system.
 */
//-------------------------------------------------------------------------------
Date A1Date::ToUtcDate()
{
    UtcDate utcDate;
    Date    tempDate;

    // convert A1 date to equivalent UTC date
    MAB::Time a1Time(*this);
    utcDate = UtcDate(a1Time);

    utcDate.YearMonDayHourMinSec(tempDate.year, tempDate.month,  tempDate.day,
                                 tempDate.hour, tempDate.minute, tempDate.second);
    return tempDate;
}
```

Appendix B Doxygen Commands

- Below is a list of widely used commands with a description of their arguments. For a full list of all commands, refer to the Doxygen Documentation Section 21 Special Commands from http://www.stack.nl/~dimitri/doxygen/download.html#latestman.

    - `@author {list of authors}` *Starts a paragraph where one or more author names may be entered.*
    - `@class <name> [<header-file>] [<header-name>]` *Indicates that a comment block contains documentation for a class with name. Optionally a header file and a header name can be specified.*
    - `@date {date description}` *Starts a paragraph where one or more dates may be entered.*
    - `@defgroup <name> (group title)` *Indicates that a comment block contains documentation for a group of classes, files or namespaces.*
    - `@endlink` *This command ends a link that is started with the `@link` command.*
    - `@enum <name>` *Indicates that a comment block contains documentation for an enumeration.*
    - `@example <file-name>` *Indicates that a comment block contains documentation for a source code example.*
    - `@exception <exception-object> {exception description}` *Starts an exception description for an exception object with name `<exception-object>`. Followed by a description of the exception.*
    - `@file [<name>]` *Indicates that a comment block contains documentation for a source or header file with name `<name>`.*
    - `@fn (function declaration)` *Indicates that a comment block contains documentation for a function (either global or as a member of a class).*
    - `@include <file-name>` *This command can be used to include a source file as a block of code. The command takes the name of an include file as an argument.*
    - `@interface <name>` *Indicates that a comment block contains documentation for an interface with name `<name>`.*
    - `@link <link-object>` *This command can be used to create a link to an object (a file, class, or member) with a user specified link-text. The link command should end with an `@endlink` command.*
    - `@name (header)` *This command turns a comment block into a header definition of a member group.*
    - `@namespace <name>` *Indicates that a comment block contains documentation for a namespace with name `<name>`.*
    - `@package <name>` *Indicates that a comment block contains documentation for a Java package with name `<name>`.*
    - `@param <parameter-name> {parameter description}` *Starts a parameter description for a function parameter with name `<parameter-name>`. Followed by a description of the parameter.*
    - `@return {description of the return value}` *Starts a return value description for a function.*
    - `@retval <return value> {description}` *Starts a return value description for a function with name `<return value>`. Followed by a description of the return value.*
    - `@struct <name> [<header-file>] [<header-name>]` *Indicates that a comment block contains documentation for a struct with name `<name>`.*
    - `@test {paragraph describing a test case}` *Starts a paragraph where a test case can be described.*
    - `@union <name> [<header-file>] [<header-name>]` *Indicates that a comment block contains documentation for a union with name `<name>`.*
    - `@var (variable declaration)` *Indicates that a comment block contains documentation for a variable or enum value (either global or as a member of a class).*
    - `@version {version number}` *Starts a paragraph where one or more version strings*

may be entered.

- **@warning {warning message}** *Starts a paragraph where one or more warning messages may be entered.*

References

1. "C Style Guide", Doland, J. et. al., SEL-94-003, Software Engineering Laboratory Series, Goddard Space Flight Center, August 1994.

2. Effective C++, Meyers. S., Addison-Wesley Professional Computing Series, 1992.

3. C++ Primer, 2nd Edition, Lippman, S., AT&T Bell Laboratories, 1991.

4. "Programming in C++ Rules and Recommendations", Henricson, M. and Nyquist, E., Ellemtel Telecommunication Systems Laboratories, 1990-1992.

5. C++ Style Guide, Version 1.0, Software and Automation Systems Branch, Goddard Space Flight Center, July 1992.

6. "C++ Programming Style Guides", Eckel, B., UNIX Review, March 1995.

7. "C++ Coding Standard", http://www.chris-lott.org/resources/cstyle/CppCodingStandard.html.