Picture Metadata Toolkit

C++ Style Guide

Document Version:	*1.1*
Document Authors:	*George Sotak*
Date:	*April 25, 2001*

Table of Contents

Revision History

Version	Date	Author	Description
0.1	10/19/2000	George Sotak	Initial version
0.8	10/23/2000	George Sotak	Removed requirement that local variables have semantically meaningful name. Removed requirement that typedefs and emunerations end in T and E, respectively.
0.9	10/24/2000	Dan Rupe	Removed section about changing #ifndef's. Added comment about putting default constructor, copy constructor, and assignment operator declarations into the private section of classes that do not implement them.
0.95	10/25/2000	George Sotak	Standard Information Block: removed paragraph stating warranty information. This statement is already in the license, no need to repeat it here.
0.98	10/28/2000	George Sotak	Added entire section on class documentation.
1.1	04/25/2001	George Sotak	Update class documentation section to reflect the swtich to doxygen.

Introduction

This document is intended to serve as a C++ programming style guide for the Picture Metadata Toolkit (PMT). Contributors to PMT should follow this guide to ensure a consistent appearance of the code.

Standard Information Block

The following comment block must appear at the top of every source code file:

```
/*
 * The contents of this file are subject to the Kodak Public
 * License Version 1.0 (the "License"); you may not use this file
 * except in compliance with the License. You may obtain a copy of
 * the License at http://www.kodak.com/go/drg
 *
 * The Original Code is the Picture Metadata Toolkit,
 * released November 13, 2000.
 *
 * The Initial Developer of the Original Code is Eastman
 * Kodak Company. Portions created by Kodak are
 * Copyright (C) 1999-2000 Eastman Kodak Company. All
 * Rights Reserved.
 *
 * Creation Date: mm/dd/yyyy
 *
 * Original Author:
 * FirstName LastName <email address>
 * <copyright notice>
 *
 * Contributor(s):
 * FirstName LastName <email address>
 * FirstName LastName <email address>
*/
```

Identifiers

General

Identifiers are the key to understandable code. A well thought out identifier clearly places the item in context and conveys its purpose or reason for existence.

Class

A class' identifier should convey its role or responsibility within the system. For example, the PmtAccessor class' role is to provide access to metadata stored in files. A derived class may need to carry some aspect of its parent's identifier in order to fully convey its role. For example, accessor implementations are derived from the PmtAccessor class. The identifier of the accessor implementation for the Exif image file format is PmtExifAccessor, carrying the Accessor component of the base class' identifier.

Rules

- Should convey its role or responsibility.
- Use upper case letters as word separators and lower case for rest of word.
- First character must be upper case.
- The first three characters must be ‘Pmt’.
- Do not use underscore ‘_’ as a separator.

Example

```
PmtOneTwoThree
PmtMetadata
PmtCompositeMetadata
```

Member Functions (Methods)

Rules

- Should convey its underlying behavior.
- Use upper case letters as word separators and lower case for rest of word.
- First character must be lower case.
- Can use abbreviation when convey of behavior is not affected.
- Do not use underscore ‘_’ as a separator.

Example

- Normal Member Functions

```
readMetadata( … )
open( … )
```

Member Variables (Class Attributes)

Rules

- Must convey the semantics of the value being stored
- Use upper case letters as word separators and lower case for rest of word.
- First character must be a lower case ‘m’.
- Can use abbreviation when semantics are not affected.
- Do not use underscore ‘_’ as a separator.

Examples

```
double mMemberVariable ; –or– double mMemberVar ;
int mVariable ;
```

Local Variables

Rules

- Use upper case letters as word separators and lower case for rest of word.
- The first character must be lower case.
- Do not use the underscore '_' as a separator.

Examples

```
      int localVariable ; –or– int localVar ;
```

Static Variables

Rules

- Must convey the semantics of the value being stored.
- Use upper case letters as word separators and lower case for rest of word.
- The character 's' must appear immediately after any other modifying characters.
- Can use abbreviation when semantics are not affected.
- Do not use underscore '_' as a separator.

Examples

```
      double gsGlobalVariable ; // global static
      int msMemberVariable ;    // member static (class variable)
```

Global Variables

Rules

- Must convey the semantics of the value being stored.
- Use upper case letters as word separators and lower case for rest of word.
- First character must be a lower case 'g'.
- Can use abbreviation when semantics are not affected.
- Do not use underscore '_' as a separator.

Examples

```
      double gGlobalVariable ;
      int gVariable ;
```

Global Constants

Rules

- Must convey the semantics of the global constant.
- Comprises all upper case letters.
- First four characters must be ‘PMT_’
- Words separated by the underscore ‘_’ character.

Examples

```
const int PMT_GLOBAL_CONST = 10 ;
```

Type Definitions

Rules

- Must convey the semantics of the type.
- Use upper case letters as word separators and lower case for rest of word.
- Can use abbreviation when semantics are not affected.
- The first character must be upper case.
- All type definitions exposed to the user must begin with ‘Pmt’
- Do not use the underscore ‘_’ as a separator.

Examples

```
typedef unsigned short PmtUInt16
typedef EkSmartPtr<PmtMetadata> PmtMetadataPtr
```

Enumerations and Their Elements

Rules

- Enumeration’s identifier
  - Must convey semantics of the enumeration.
  - Use upper case letters as word separators and lower case for rest of word.
  - The first character must be upper case.
  - All enumerations exposed to the user must begin with “Pmt”.
  - Do not use the underscore ‘_’ as a separator.
- Enumeration element’s identifier
  - Must convey the semantics of the element.
  - Comprises all upper case letters.
  - Words separated by the underscore ‘_’ character.

Examples

```
enum PmtImageFileFormatName
{
  PMT_FORMAT_UNKNOWN,
  PMT_FORMAT_EXIF,
  PMT_FORMAT_FPX,
  PMT_FORMAT_TIFF,
  PMT_FORMAT_APS
};
```

#define and Macros

Rules

- Must convey the semantics of the #define or macro.
- Comprises all upper case letters.
- All #define's and macros exposed to user must begin with 'PMT_'
- Words separated by the underscore '_' character.

Examples

```
#define PMT_EXIF_AUDIO_TAG_TYPE 100

#define PMT_METADATA_TYPE(facKey, mdType) \
   PmtMetadata::getFactory().addEntry( facKey, new mdType("", facKey) )
```

Class Code File

Rules

- In general, one class interface / implementation per file; however, a set of small, highly related classes can be grouped into a single file.
- The root of the file name is to be the identifier of the class.
- Separate into interface and implementation.
- Interface is placed in file with ".h" as the extension.
- Implementation is placed in file with ".cpp" as the extension.
- Use interface file guards to protect against multiple inclusion. See example.

Example

```
ExampleClass.h
      #ifndef EXAMPLE_CLASS_H    // interface file guard
      #define EXAMPLE_CLASS_H
      class ExampleClass { … };
      #endif // EXAMPLE_CLASS_H

ExampleClass.cpp
      #include "ExampleClass.h"
      …
```

Class

Layout

A common class layout is critical from a code comprehension point of view. With a common layout, one does need to waste time and energy searching around the class for the protected member functions, one can instantly navigate to the desired location.

Rules

- The general structure of a class is:
  - friend declarations
  - public section of interface
  - protected section of interface
  - private section of interface
- Each section of the interface has the following sub-structure:
  - members variables
  - constructors
  - destructor
  - operators
  - accessors
  - member functions

Methods

Required

Every class needs a minimum set of class members in order to be usable. The compiler automatically generates a few of these; however, it is best to explicitly declare all members to be clear. The following is the list of required member functions:

• Default Constructor:	ClassName(void) ;
• Copy Constructor:	ClassName(const ClassName& theSrc) ;
• Destructor:	~ClassName(void) ;
• Assignment Operator:	ClassName& operator=(ClassName& from) ;

If the class does not need a default constructor, copy constructor, or assignment operator, then place the appropriate declaration(s) in the private section of the class. This will ensure a compile-time error message if one or more of these is used unwittingly.

Declare a virtual destructor in a class if and only if the class contains at least one virtual function.

Layout

- Method declaration should be contained to a single line. If method has multiple arguments and line becomes unreasonably long, break the argument list at a reasonable point and line up with the first argument.

Example

```
    int aMethodWithLongArgList( int arg1, int arg2,
                                double arg3, double arg4 ) ;
```

Inline Methods

- `inline` keyword must always be present.
- One line implementations can occur either next to the method or on the line right after the method and indented four spaces.
- Multiple line implementations must go after the class declaration, see the **Class Interface Template.**

Example

```
      inline int aShortMethod( int arg ) { return arg ; }
      inline int aShortMethod( int arg )
          { return arg ; }
      inline double aLongMethod( void ) ;
```

Accessors

There are two acceptable methods for implementing accessors.

Attributes as Objects

This is the recommended approach to class attribute access.

```
         class X
         {
             public:
                 const int&       age() const     { return mAge; }
                 int&             age()           { return mAge; }

                 const string&    name() const    { return mName; }
                 string&          name()          { return mName; }
             private:
                 int              mAge;
                 string           mName;
         }
```

The above two attribute examples shows the strength and weakness of the Attributes as Objects approach.
When using an **int** type, which is not a real object, the **int** is set directly because **Age()** returns a **reference**. The object can do no checking of the value or do any representation reformatting. For many simple attributes, however, these are not horrible restrictions. A way around this problem is to use a class wrapper around base types like **int**.

When an object is returned as reference, its **operator=()** is invoked to complete the assignment. For example:

```
         X x;
         x.name()= "test";
```

This approach is also more consistent with the object philosophy: the object should do it. An object's **operator=()** can do all the checks for the assignment and it's done once in one place, in the object, where it belongs. It's also clean from a name perspective.

If the attribute is meant to be read only, do not provide the non-const version of the accessor.

One Method Name

```
         class X
         {
             public:
                 int    age() const     { return mAge; }
                 void   age(int age)    { mAge= age; }
             private:
                 int mAge;
         }
```

This method is acceptable when not using the preferred method, **Attributes as Objects**.

Initialize All Variables

What more needs to be said. More problems than you can believe are eventually traced back to a pointer or variable left uninitialized.

Do Not do Real Work in Object Constructors

Do not do any real work in an object's constructor. Inside a constructor initialize variables only and/or do only actions that can't fail.

Create an init(), open() or some other appropriately named method to complete the construction process. This method is called after object construction.

Justification

- Constructors can't return an error, object instantiators must check an object for errors after construction. This idiom is often forgotten.
- Thrown exceptions inside a constructor can leave an object in an inconsistent state.
- When an object is a member attribute of another object the constructors of the containing object's object can get called at different times depending on implementation. Assumptions about available services can be violated by these subtle changes.

Const Correctness

- Be const correct, see the C++ FAQ Lite, section 18, http://marshall-cline.home.att.net/cpp-faq-lite/

Keep Method Implementation Short

Method implementations should be as short as possible. If a method gets to be a few dozen lines of code then it may be appropriate to break the functionality down into smaller methods.

Namespaces

- Pmt does not define its own namespace. To avoid identifier collisions, append “Pmt” to identifiers.
- **std** namespace is defined by some compilers and not by others. Ideally, the desire is to choose the lowest common denominator, which in this case is to disable the use of the **std** namespace. However, Microsoft does not provide the developer with this choice, the **std** namespace is required to be use in Visual C++. In order to address this situation, the “Ek” library supplied with Pmt defines **EK_USING_STD** to `be “namespace std { }; using namespace std;`” if compiler supports the **std** namespace and defines it to nothing otherwise. This is transparently taken care of by including “`EkCompiler.h`”. For compilers that support the **std** namespace, but also allow for the select enabling / disabling of its use (like GNU g++ v2.95.2), the “Ek” library set **EK_USING_STD** if **EK_USE_NAMESPACE** is defined.

Documentation

- The “doxygen” (http://www.stack.nl/~dimitri/doxygen/)documentation program is used to generate HTML pages directly from the class header files. See doxygen’s documentation for details.
- Use the Qt commenting style:

  /*! multi-line comment block */

  //! single line comment
- Documentation will always appear just before the definition or declaration of the item being documented.
- Provide a one line summary and a long description of a class. This must appear in the class’ header file just before the class definition:

```
//! The one line summary
/*! Long Description
    of the class
    through multiple lines.
    Provide example code */
class SomeClass { … } ;
```

  The leading one line description indicator (the “//:”) is required by perceps. Leave this comment blank. Provide example code in the long description, surround the code with the \code … \endcode doxygen keywords to preserve the formatting of the code.
- Comment global typedefs with a one line description:

```
//! One line description of typedef
typedef int SomeTypedef ;
```

- Methods are to have a one line summary, a long description, parameter description, return description, and if it throws an error, a statement to that fact. The following is the format:

```
//! One line summary for methodA
/*!
   \param arg1 short description
   \param arg2 short description
   \return description of return value, if there is one.
   \exception expection-object description of exception

   Long description can go over several comment lines.
   This is an example layout of a method with
   two parameters, a return value, and throws an error.
*/
int methodA( int arg1, double arg2 ) ;
```

```
//! One line summary for methodB
/*! This is an example layout for a
    method with no arguments, no return value,
    and does not throw any errors.
*/
void methodB( void ) ;
```

NOTE: if the long description of the method is very long, just place the one line summary in the head file and place the long description in the implementation file right before the method's implementation. This keeps the header file readable.

Formatting

Indentation / Tabs / Spaces

- Statement indentation is 4 spaces from the first column of the enclosing block.
- Access keywords (public, protected, private) are indented 4 spaces from enclosing block, declarations are indented 4 spaces from beginning of access keyword.
- Use spaces instead of tabs. Most editors can automatically substitute spaces for tabs.
- Fix tabs at 4 spaces.

Statements

- There should be only one statement per line unless statements are very closely related.

Placement of Curly Braces - {}

- Place curly braces on their own line.
- Open curly brace, {, is to line up with the first character of the statement defining the block. Closing curly brace, }, is to be indented such that it is inline with its corresponding opening brace.

Example

```
if ( condition )
{
    statements…
    while ( condition )
    {
        statements…
    }
}
else
{
    statements…
}
```

Parens (), Keywords and Functions

- Do not put parens next to keywords. Put a space between.
- Put parens next to function names.
- Do not use parens in return statements when it's not necessary.

Switch Statement

- Falling through a case statement into the next case statement shall be permitted as long as a comment is included.
- The *default* case should always be present and trigger an error if it should not be reached, yet is reached.
- If you need to create variables put all the code in a block.

Example

```
switch (...)
{
     case 1:
       ...
       // FALL THROUGH COMMENT

     case 2:
     {
         int v;
         ...
     }
       break;

     default:
}
```

goto Statement

- Follow the general philosophy behind well-structured code – use **goto** only when absolutely necessary.

continue, break Statements

- **continue** and **break** are disguised **goto** statements and, therefore, should be used only when absolutely necessary.
- **continue** is especially troublesome in that its use can lead to the seemingly random bugs by inadvertently bypassing test conditions or increment / decrement statements.

Use #if 0 to Comment Out Code Blocks

If you need to comment out a large code block use "`#if 0 … #endif`". Using "`/* … */`" can be troublesome since comments cannot contain comments, and surely in a large code block there will be a comment ;-).

Class Interface Template

```
#ifndef XX_H
#define XX_H

Standard Information Block goes here

// SYSTEM INCLUDES

// PROJECT INCLUDES
#include “EkCompiler.h”

// LOCAL INCLUDES

// FORWARD REFERENCES

// TYPEDEFS

class XX
{
    public:
        //CONSTRUCTORS
        // Default constructor
        XX(void);

        // Copy constructor.
        XX(const XX& from);

        // DESTRUCTOR
        ~XX(void);

        // OPERATORS
        // Assignment operator.
        XX& operator=(XX& from);

        // Accessors

        // MEMBER FUNCTIONS

    protected:
        // MEMBER VARIABLES
        // CONSTRUCTORS
        // OPERATORS
        // Accessors
        // MEMBER FUNCTIONS

    private:
        // MEMBER VARIABLES
        // CONSTRUCTORS
        // OPERATORS
        // Accessors
        // MEMBER FUNCTIONS
};

// INLINE METHODS
//

// EXTERNAL REFERENCES
//

#endif  // _XX_H
```

Miscellaneous

Prefer Streams Over Stdio

- Streams are type safe, stdio is not, which is one of the reasons you are using C++, right? That's one good reason to use streams.
- Streams offer a standard interface for dumping out objects, << operator. This is not true of objects and stdio.
- Different types of streams are interchangeable, once an object can dump itself to a stream, it can dump itself to any stream. One stream may go to the screen, but another stream may be a serial port or network connection.

Exception Handling

- Use the throw construct of the C++ language.
- Always throw an instance of the `PmtError` class. See documentation for `PmtError` for details on usage.

Example

```
if( throwError )
    {
      string msg = "PmtTranslator::assignT: Cannot convert FLOAT format
type to " ;
      msg += md->type() ;
      msg += " for key: " ;
      msg += md->key() ;
      throw PmtError(PMT_FORMAT_ENTRY_VALUE_CONVERSION_ERROR, msg, WHERE);
    }
```

UNICODE, Wide Strings

- Unicode support is enable through the defining of the preprocessor symbol “ `_UNICODE`”.
- Prefer the use of `EkStringT` over the C++ `string` class. `EkStringT` is a `typedef` that is switched between being `string` for non-Unicode builds and `wstring` for Unicode builds.

Smart Pointer

- Avoid memory management problems by using smart pointers.
- The “Ek” library provides the recommended solution
  - EkSmartPtr<PtrType> : The smart pointer class.
  - `EkRefCount<MutexType>` : Thread safe reference counting class whose interface is used by EkSmartPtr to keep track of the number of references to a pointer. Any class that will have an associated smart point must inherit from EkRefCount.
- Always `typedef` the smart pointer declaration to some meaningful identifier.

Example

```
class PmtMetadata : public EkRefCount<EK_DEFAULT_MUTEX> { … } ;

typedef EkSmartPtr<PmtMetadata*> PmtMetadataPtrT ;
```

```
PmtMetadataPtrT rootMd = PmtMetadata::create() ;
rootMd->key() ;
```

Dynamic Link Library (DLL) Support

The preferred method for DLL support is using `_declspec(dllexport)` when building the DLL and `_declspec(dllimport)` when compiling and application that uses the DLL. In addition, cross platform support requires the ability to not have any `_declspec()` present during builds. Each module of Pmt has a header file "*`Module`*`Defs.h`", where Module is replaced with the name of the module, that defines *`MODULE`*`_DECL` to be either the export or import version of the DLL declaration. The build of the DLL is indicated by defining *`MODULE`*`_BUILD_DLL` on the compiler's command line, this sets *`MODULE`*`_DECL` to `_declspec(dllexport)`. A application building against the module's DLL defines `PMT_DLL` on the compiler's command line, this sets *`MODULE`*`_DECL` to `_declspec(dllimport)`. If neither *`MODULE`*`_BUILD_DLL` nor `PMT_DLL` are defined on the compiler's command line, then *`MODULE`*`_DECL` is set to nothing (this would be the case for Solaris and Linux builds).

Example

```
Module in this case is PmtCore:
Header file: PmtCoreDefs.h

#ifdef _MSC_VER  // Windows platforms

#if defined( PMTCORE_BUILD_DLL)
#define PMTCORE_DECL __declspec(dllexport)

#elif  (defined( PMT_DLL) || defined( PMTCORE_DLL))
#define PMTCORE_DECL __declspec(dllimport)

#endif  // PMTCORE_BUILD_DLL

#endif  // _MSC_VER

#if !defined( PMTCORE_DECL ) // non Windows build, set to nothing
#define PMTCORE_DECL
#endif
```

To export the `PmtMetadata` class, add `PMTCORE_DECL` between the class keyword and the class identifier:

```
class PMTCORE_DECL PmtMetadata: public EkRefCount<EK_DEFAULT_MUTEX>
{ … } ;
```

Thread Support

TBD… Dan, can you provide the guidance here?

Standard Template Library

- Prefer the use of STL containers.

STL Map, MultiMap, Set, and MultiSet, and Dynamic Link Libraries

- Instantiations of these STL containers cannot cross DLL boundaries; therfore, when they are used, do not expose there use through the using classes interface (if the using class is exported from a DLL). Provide accessor methods to the contents of the container.

Operating System Interface

- Always use the POSIX.1 API to the operating system.

No Data Definitions in Header Files

Do not put data definitions in header files. for example:

```
/*
 * aheader.h
 */
int x = 0;
```

- It's bad magic to have space consuming code silently inserted through the innocent use of header files.
- It's not common practice to define variables in the header file so it will not occur to developers to look for this when there are problems.
- Define the variable once in a .cpp file and use an extern statement to reference it.
- If the data is an object, consider using a singleton for access to the data.