## Table of Contents

Introduction

The main goal of these rules and recommendations is to have future software development respect the following criteria:

- Correctness
- Maintainability
- Consistency
- Readability
- Comprehensiveness
- Portability

# 1. General Recommendations

## 1.1. File Header

Rule 1: All source files should contain the standard Copyright Notice as well as a standard header describing useful information.

This standard header is as follows:

| | | | |
|---|---|---|---|
| File Description | <Description of the file> | | |
| Contents | <List of functions defined> | | |
| Author | <Full Name> | | |
| Date | <Creation date> | | |
| Change History | <Author | Date | Description> |

Note: Dates should be given in the format "January 24, 2005"

## 1.2. Module Header

Rule 1: Every software module should be preceded by a standard header describing dedicated information.

The following are considered to be software modules:

1. A class/struct/union/typedef declaration
2. A global function declaration
3. A global/member function definition
4. A macro definition

This standard header is as follows:

| | | | |
|---|---|---|---|
| AUTHOR | <Full name> | | |
| DATE | <February 7, 2005> | | |
| NAME | <Module name> | | |
| DESCRIPTION | <Description of module> | | |
| ARGUMENTS | <if any, otherwise mark "None"> | | |
| RETURNS | <if any, otherwise mark "None"> | | |
| NOTES | <Comments> | | |
| CHANGE HISTORY | <Author | Date | Description> |

## 1.3. Comments

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program.

Rule 1: All comments are to be written in English.

Rule 2: All declarations should have a comment indicating the purpose of the declaration.

Rule 3: A header file has to contain comments describing class and functions.

Note: Modules headers are intended to describe what the whole module does. Every other comment is intended to describe what some small section of code (one line or one block) does and must be placed before this code using the same indentation.

The following keywords can be included in the comments:

TODO- topic
Means there's more to do here, don't forget.

CR- [bugid] topic
Means there's a Known bug here, explain it and optionally give a bug ID.

TRICKY- topic
Tells somebody that the following code is very tricky so don't go changing it without thinking.

WARNING- topic
Beware of something.

COMPILER- topic
Sometimes you need to work around a compiler problem. Document it. The problem may go away eventually.

QUESTION- topic
Sometimes you have an un-answered question. Document it. The question may go away eventually.

Note: If the characters // are consistently used for writing comments, then the combination /* */ may be used to make comments out of entire sections of code during the development and debugging phases.

Note: Sometimes large blocks of code need to be commented out for testing and C++ does not allow comments to be nested using /* */. For this you can use a #if 0 block
Ex:

```
void
example()
{
        great looking code
        #if 0
        lots of code
        #endif
        more code
}
```

# 2. Source Management

## 2.1. Source Organization

Rule 1: An include file should not contain more than one class definition.

Rule 2: Inline function definitions should be placed in a separate file.

Rule 3: Class implementation file should only contain member function/data definitions related to that class

Rule 4: The definitions of the member function of a class declaration *must* be in the same order as presented in the class declaration.

Source files must be organized as follows:

1. #includes should be grouped as
   <system header files> //use < brackets >
   "project header files" // use ""
   "module header files" // use ""
2. Non-class static data and const definitions
3. class static data member initializations
4. class friend functions' definitions
5. class member functions' definitions

## 2.2. File Naming

Rule 1: Include files in C++ would have the file name extension ".h".

Rule 2: Implementation files in C++ would have the file name extension ".cpp".

Rule 3: Source files always have unique names

Rule 4: An include file for a class should have a file name related to its corresponding class name

Since class names must generally be unique within a large context, it is appropriate to use this characteristic when naming its include file. This convention also makes it easy to locate a class definition using a file-based tool.

## 2.3. Include Files

Rule 1: Every include file must contain a mechanism that prevents multiple inclusions of the file

Note: The easiest way to avoid multiple includes of files is by using an #ifndef/#define block at the beginning of the file and an #endif at the end of the file.
Example:

```
#ifndef __LTKINC_H
#define __LTKINC_H

#include <iostream>
#include <string>
.....
#endif
```

Rule 2: Definitions of classes that are only accessed via pointers (*) or references (&) shall *not* be included as include files

Rule 3: Avoid specifying relative files names in #include directives

Rule 4: Every implementation files must *strictly* include the relevant file that contains declarations used in the functions that are implemented in the file

Rule 5: Use the directive #include “filename.h” for user-defined include files

Rule 6: Use the directive #include <filename.h> for include files coming with third-party libraries or standard C/C++ libraries

# 3. Source Presentation

## 3.1. Naming Conventions

Rule 1: Naming Classes, Interfaces, Namespaces and Packages: C++ uses a standard convention for naming classes. Hungarian notations will be used to name C++ classes.

Note: All the names will be prefixed by product related string (for example: "LTK") to lessen the risk of a clash between the names a user selects and the names in other libraries the user might want to include. It also makes the code more readable so that the high-level origin of a class is obvious when the class is used.

Example:
LTKClassifierPCA
LTKClassifierDTW
LTKUnknownTypeException

Rule 2: Namespace: For naming use the company name, division name along with component names.
Example: HPLTK

Note: Don't place "using namespace" directive at global scope in a header file. This can cause lots of invisible conflicts that are hard to track. Keep "using" statements to implementation files.

Rule 3: Naming Methods: The method names should be verbs, in mixed case with the first letter lowercase and the following internal word’s first letter capitalized.

Example:

```
read (); //CORRECT
void forwardMessage (const char* messageName) throw(); //CORRECT
rollBackTransaction(LTKClassifierPCA * pt) throw(); //CORRECT
transactionRollBack(LTKClassifierPCA * pt) throw(); //WRONG
RollBackTransaction(LTKClassifierPCA * pt) throw(); //WRONG
bool equals(int i) throw(); //CORRECT
```

The argument parameters should be named in full English with the first letter of any non-initial word in uppercase.

Example:
count
timeoutValue

Rule 4: Naming Variables: Use a full English descriptor to name the variable that it represents. Variables that represent collections such as arrays, lists should be given names that are plural.

Example:
saleValue
telephoneNumbers

Use a lowercase first letter, with the first letter of each internal capitalized for naming the variables.

Note: Member variables of a class are prefixed with a "m_"
Example:
m_classId
m_featureValue

Rule 5: Naming Constants: The names of constant variables should be all uppercase letters, with underscore ("_") separating internal words.
Example:
const int DECIMAL_PLACES = 2;
enum {MT_ERROR, MT_WARNING};

## 3.2. Declarations

Rule 1: Declarations Per Line: There should be one declaration per line.
This allows short or one-line comments to be placed for the declaration. There can be one space between the type and the identifier. Following examples illustrate this concept.
int indentationLevel; // indentation level
int tableSize; // size of the table

Rule 2: Placement of Declarations: Place the declarations immediately in the beginning of the block (code surrounded by curly braces) that have relevancy to the whole of the code block.
Example:

```
void MyMethod()
{
        int int1; // beginning of method block
        if (condition)
        {
                int int2; // beginning of "if" block
                …
        }
}
```

Exception to Rule 2: The exceptions to this rule are the declaration of the variables that are needed only by the following code block and the temporary variables in the "for" loop.
Example:

```
for (int i = 0; i < maxLoops; i++)
{
        …
}
```

Note: Do not declare the same variable name in an inner block. This can confuse others.

```
int count;
…
func()
{
        if (condition)
        {
        int count; / / AVOID!
        …
        }
…
}
```

Rule 3: Initialization: Initialize local variables where they're declared.

The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

Example:

```
int numSample=0;
```

Rule 4: Class and Interface Declarations: Follow these rules when declaring C++ classes and interfaces.

1. Don't leave space between a method name and the opening parenthesis.
2. Place the Open brace "{" of the block, on a new line, at the same indentation level as the previous line.
3. Place Closing brace "}" of the block, on a new line indented to match its corresponding opening statement. The opening and closing braces are placed on the next lines at the same indentation level as the previous line, for Null statements.

Example:

```
class Sample extends Object
{
        int ivar1;
        int ivar2;
        Sample(int i, int j)
        {
                ivar1 = i;
                ivar2 = j;
        }
        int emptyMethod()
        {
        }
        …
}
```

## 3.3. Programming Style

### 3.3.1. Indentation

Rule 1: Lines should be indented by a step of 4 spaces wherever possible.

When splitting long lines, use spacing of 8 on the next line. However, when splitting multiple function arguments the indentation on the next line can be made to align with the function arguments on the previous line.

Note: Code blocks appearing within braces should begin on the line immediately below the opening brace and should be indented. Similar structure should be followed for nested code blocks.

Note: Use spaces for indentation instead of Tabs as the tab size might vary across editors.

### 3.3.2. Simple Statements

Rule 1: Write *only* one statement or object declaration on a single line and the line length is to be limited to 80 characters. Many terminals cannot handle lines longer than 80 characters and the programmer will not be able to see the whole statement on the terminal.

### 3.3.3. Brackets

The "{" brace should always be on a new line, immediately below the start of the previous command. The matching "}" brace should be in the same column position as the starting "}"

Example:

```
While (!finished)
{
        switch(CaseVal)
        {
        . . .
        }
}
```

### 3.3.4. Variable Declaration

Rule 1: Declare one variable on one line, use separate lines for each subsequent declaration and provide a comment for the usage/purpose of the variable.

### 3.3.5. Class

Rule 1: All "private" typedefs/enums/data members are placed first, followed by "protected" typedefs/enums/data members, followed by "public" typedefs/enums.

Afterwards, "public" member functions appear, followed by "protected" member functions, followed by "private" member functions.

By placing the “public” section first, everything that is of interest to a user is gathered in the beginning of the class definition. The “protected” section may be of interest to designers when considering inheriting from the class. The “private” sections contain details that should have the least general interest.

Note: Avoid member functions definition within the class definition

Class definitions are less compact and more difficult to read when they include definitions of member functions. Also, it is easier for an inline member function to become an ordinary member function that way.

Note: If there is any "friend" declaration, it will appear at the top.

Example:

```
class MtRefreshIntray
{
        friend redrawIntray(const MtRefreshIntray& intray)

        throw(MtIntrayException::BadIntray);
        private:
                //typedefs, enums, data members
        protected:
                //typedefs, enums, data members
        public:
                //typedes, enums
        public:
                //member functions
        protected:
                //member functions
        private:
                //member functions
};
```

Note: Follow the indentation standard as shown in the previous example. Leave a blank line before the access keywords (public etc).

Note: In all the access sections the data members and typedefs are grouped as follows:

1. typedefs
2. enums
3. instance data members
4. static data members

### 3.3.6. Function

Rule 1: Always provide the return type of a function explicitly

Note: When declaring a function, provide argument names (and not only their types)

Note: In a function declaration/definition, leading parenthesis, arguments and closing parenthesis are to be written on the same line as the function name, if space permits.

Note: In all the access sections the member functions are grouped as follows:

1. constructors
2. destructor
3. initialization functions
4. main interface functions
5. access/modifier functions (get/set Functions)

6. domain specific functions (such as parsing, communication etc)
7. static functions

### 3.3.7. Statements

Note:

1. In a statement, a blank space should be provided to separate a keyword and the open parenthesis.
2. When the statement is a declaration of a method, there should not be a space between the method name and the opening parenthesis.
3. The arguments in the argument list should be separated by a space after the commas.
4. The expressions in the "for" statement should be separated by a space after the semicolon mark.
5. Type cast operations should be followed by a space.
6. All binary operators except dot (.) operator should be separated from their operands by spaces.

### 3.3.8. Compound Statements

Rule 1: All the enclosed statements in the block are to be indented one level more than the compound statement.

Rule 2: The opening brace should be placed at beginning of the line following the compound statement and the closing brace should begin on a new line and be indented to match the position of the opening brace.

Rule 3: Braces are to be used around all statements, even singletons, when they are part of a control structure, such as a if-else or for statement. Statements can be added later with ease.

### 3.3.9. 4.3.9. brace.

ion of hte e begithe ed in the comments:

- Flow Control Statements

#### 3.3.9.1. if-else statement:

Rule 1: Use opening and closing braces for both the "if" section and "else" section, even if they contain a single statement. Place the inner "if" statement on the same line as the else statement of the previous "if".

Rule 2: Indent the block of code inside the curly braces as shown in the example.

Example:

```
if (condition)
{
        myMethod((byte) aNum, (Object) x);
}
else if (condition)
{
        myFunc((int) (cp + 5), ((int) (I + 3)) + 1);
}
else if (condition)
{
        while (true)
        {
```

```
            …
        }
}
```

#### 3.3.9.2. for Statement:

Here is an example on the preferred use of the “for” statement.

```
for (initialization; condition; update)
{
        statements;
}
```

The empty for statement should be coded as follows.

```
for (initialization; condition; update);
```

#### 3.3.9.3. while Statement:

The while statement should be coded as follows.

```
while (condition)
{
        statements;
}
```

The empty while statement should be coded as follows.

```
while (condition);
```

#### 3.3.9.4. do-while Statement:

A do-while statement should have the following form:

```
do
{
        statements;
} while (condition);
```

#### 3.3.9.5. switch Statement:

switch statement should have the following form:

```
switch (condition)
{
        case ABC:
                statements;
                / * falls through * /
        case DEF:
                statements;
                break;
        case XYZ:
                statements;
                break;
        default:
                statements;
                break;
}
```

Note: Document when a case falls through (doesn't include a break statement) by adding a comment where the break statement would normally be.

Note: Provide a default case for every switch block. The break in the default case prevents a fall-through error when a new case statement is added later.

#### 3.3.9.6. Pointers and References

Rule 1: The de-reference operator `*' and the address-of operator `&' should immediately follow the type names in declarations and definitions.
Moreover, their use could be avoided by using some adequate 'typedef' statements in order to avoid any kind of ambiguity

Note: The characters `*' and `&' are to be written with the type of variables instead of with the name of variables in order to emphasize that they are part of the type definition.

#### 3.3.9.7. try-catch statements

A try-catch statement should have the following format:

```
try
{
        statements;
}
catch (ExceptionClass e)
{
        statements;
}
```

Note: Some points one should be aware of are listed below:

i) If any 'lock' is obtained in a try block, and for some reason, the underlying layer(s) throw for some reason, then make sure that the 'catch' block that is handling this try block takes care of the 'unlocking'.

Example :

```
try
{
        statements;
        GetCacheMutex().lock();
        statements;
}
catch(...)
{
        GetCacheMutex().unlock();
        statements;
}
```

ii) Judiciously avoid multiple deletes in try and cache blocks for the same object.

# 5. Design Guidelines

## 5.1. Memory Allocation

Rule 1: Do not use malloc, realloc and free statements
Do not use malloc, realloc and free. Use the new and delete operators to create and destroy dynamic objects since they are safer and more portable.

Rule 2: Use empty brackets ("[]") to delete arrays
When deleting arrays provide the empty brackets ("[]") to the delete operator otherwise only the first array entry will have its destructor called.

Example:

```
Class*  myClass = new Class[20];   // creates 20 Classes
delete[] myClass;                  // OK. Deallocate memory by calling 20 destructors
delete myClass;                    // WRONG. Deallocates by calling first destructor ONLY.
```

## 5.2. Copy Constructors and assignment Operator =

Note: Provide copy constructor and operator '= 'in all classes
Each class shall provide a copy constructor and an operator =. If the default copy constructor and the default operator = is sufficient, then that shall be specifically noted in the class.

## 5.3. Member Variables

Rule1: Do not use public and protected data members
Never use public or protected data members. If required use access functions 'get' and 'set'.

Rule 2: Data access within the class using access functions is not allowed
When protected and private data is accessed by functions in the same class the get and set functions shall NOT be used.

Rule 3: Member variables should be prefixed with a "m_" in order to distinguish them from non-members.

Rule 4: 'Get' and 'Set' access functions should return const reference

Note: Use constructor initialization instead of set functions

Note: Initialize members in the order of their declaration
During construction of an object, class members are initialized in the order of their declaration in the class, not in the order in which they are listed in the initialization list. Therefore, the initialization list must have members in the order they are declared in the header file.

## 5.4. Constant Usage

Note: Use const whenever possible

If a particular object should not be modified then make it a const object. When a object is declared as const, the compiler enforce the constraints. The placement of key word const is very important and it decides which part of the declaration is not modifiable.

Example:

```
char*       testObject              = “Some string”;    / / Non-const pointer,  non-const data
const char* testObject              = “Some string”;    / / Non-const pointer,  const data
char*  const testObject             = “Some string”;    / / const pointer,  non-const data
const char*  const testObject       = “Some string”;    / / const pointer,  const data
char const * const testObject       = “Some string”;    / / same as above. But not allowed since it violates
```

### 5.5. Pointers and References

Note: Use references instead of pointers
Use reference parameters in preference to pointers.

Note: Use typedef for pointers to improve clarity

### 5.6. Casts

Note: Do not use casting

- Avoid casting at all times if you can.
- Avoid casting away const in const objects.
- Avoid casting a base class pointer to a derived class pointer where ever possible. If casting is forced, make sure that this downward type casting is safe and shall not operate in areas beyond the memory boundary.

### 5.7. Functions without parameters

Rule1: Use C++ style function declarations

The C++ style of declaring a function without parameters shall be used.
Example:

```
MyClass(void);          / / WRONG ANSI style
MyClass();              / / C++ style
```

### 5.8. Exception handling

Rule 1: Throw exceptions whenever needed
Exceptions shall be thrown whenever an error condition arises that a piece of code cannot handle correctly or there is no suitable return type for. They shall not replace expected error conditions that can be handled by a simple return type.

Rule 2: Every thrown exception must have a catch
There shall be a “catch” for every “thrown” exception. It is bad practice to let the “unexpected” function catch any thrown exceptions (except during initial testing), as this function usually calls abort.

## 6. Doxygen Documentation Standard

Doxygen is a documentation system for C++. Doxygen can be used to generate documentation (in HTML or Latex format) from a set of documented source files. Hence, Doxygen can be used on a set of (appropriately commented) source files to extract the class structure and other relevant information. This information can be used to create/update the coding standard document for C++ according to the present standard followed in coding.

For More information on using Doxygen you can visit:
http://www.stack.nl/~dimitri/doxygen/

Doxygen is available for Windows, UNIX platforms. You can download the setup/installer from:
http://www.stack.nl/~dimitri/doxygen//download.html