Laser Interferometer Gravitational Wave Observatory
- LIGO -
CALIFORNIA INSTITUTE OF TECHNOLOGY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Document Type Technical Note	LIGO-T970211-00 -	E	12/04/97

LIGO Data Analysis System Software Specification for C, C++ and Java

Systems Integration Group

California Institute of Technology
LIGO Project - MS 51-33
Pasadena CA 91125
Phone (818) 395-2129
Fax (818) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project - MS 20B-145
Cambridge, MA 01239
Phone (617) 253-4824
Fax (617) 253-7014
E-mail: info@ligo.mit.edu

WWW: http://www.ligo.caltech.edu/

1 INTRODUCTION

The purpose of these coding standards are to facilitate the maintenance, portability and reuse of custom C, C++ and Java source code developed for the LIGO Data Analysis System (LDAS). These standards have been compiled from a variety of sources, including other standards documents, software examples from standard language references and personal experiences. As with all guidelines, there will be individual cases where full compliance is not desirable for efficiency, maintainability, or other reasons. In those particular cases, conformance should not be pursued simply for the sake of satisfying these standards.

2 FILE ORGANIZATION

2.1. File Contents

Files should be used to organize related software coded modules, either at the class level for C++ and Java or at the function level for C. The following table identifies the contents of individual files for each language:

Table 1: File Contents based on Language

File Contents	C	C++	Java
class declaration (header)	n/a	X	n/a
class definition (source)	n/a	X	X
main function	X	X	(with primary class)
functions	X	X	n/a
globals	X	X	n/a

2.2. Source File Layout

Source files should contain the follow components in the order indicated in the table below:

Table 2: Source File Content Ordering based on Language

File Element	C	C++	Java
prologue	1	1	1
package imports	n/a	n/a	2
system #includes	2	2	n/a
application #includes	3	3	n/a
external functions	4	4	n/a
external variables	5	5	n/a
constants	6	6	3
static variable initialization	7	7	4
class declaration	n/a	n/a	5
public methods	n/a	8	6
protected methods	n/a	9	7
private methods	n/a	10	8
functions	8	11	n/a

With C and C++, when possible, put #include lines in the source file instead of the header file. This will reduce unnecessary file dependencies and save on compile time.

2.3. Header File Layout

Header files should contain the following components in the order indicated in the table below (Note: Java does not use header files)

Table 3: Header File Content Ordering based on Language

File Element	C	C++	Java
file guard	1	1	n/a
prologue	2	2	n/a
system #includes	3	3	n/a
application #includes	4	4	n/a
#defines	5	5	n/a
macros	6	6	n/a
external functions	7	7	n/a
external variables	8	8	n/a
constants	9	9	n/a
structs	10	10	n/a
forward declarations	11	11	n/a
class declarations	n/a	12	n/a
public methods	n/a	13	n/a
protected methods	n/a	14	n/a
private methods	n/a	15	n/a
inline method definitions	n/a	16	n/a
functions	11	17	n/a

With C++, small inline methods may be implemented in the class definition found in the header file. Note that it is not necessary to have all file elements appear in every file, only that the ordering of the file elements follow that in the table when being implemented.

2.4. Header File Guard

In C and C++, all header files should contain a file guard mechanism to prevent multiple inclusion and improve compilation times. The name for the file guard should agree with the header file name as closely as possible. The first character should be capitalized and each new word should be capitalized. with the period removed and a capital “H” terminator character. File guards are implemented as shown below:

```
#ifndef MeaningfulNameH    // first line of the header file
#define MeaningfulNameH    // second line of the header file
.                          // body of the header file
.                          // body of the header file
.                          // body of the header file
#endif // MeaningfulNameH  // last line of header file; Note Comment!
```

3 NAMING CONVENTIONS

The following table summarizes the naming conventions to be used for identifiers found in each of the languages. The capitalization used in the table is meant to specify the capitalization used in the identifiers:

Table 4: Naming Conventions for Language Identifiers

Identifier	C	C++	Java
package	n/a	n/a	shortname
class, union, struct	MeaningfulName	MeaningfulName	MeaningfulName
exception class	n/a	MeaningfulException	MeaningfulException
interface	n/a	n/a	MeaningfulActionable
typedef	MeaningfulName	MeaningfulName	n/a
enum	MeaningfulName	MeaningfulName	n/a
function, method	meaningfulName	meaningfulName	meaningfulName
accessor method	n/a	getX, setX	getX, setX
object, variable	meaningfulName	meaningfulName	meaningfulName
#define, macro	MEANINGFUL_NAME	MEANINGFUL_NAME	n/a
const, static final variable	MEANINGFUL_NAME	MEANINGFUL_NAME	MEANINGFUL_NAME
source file name	meaningfulname.c	meaningfulname.cc	meaningfulname.java
header file name	meaningfulname.h	meaningfulname.hh	n/a
inline definition file names	n/a	meaningfulname.icc	n/a

3.1. Descriptive Names

Identifier names should be readable and self-documenting. Abbreviations and contractions are discouraged. Shorter synonyms are allowed when they follow common usage within the domain.

3.2. Valid Characters

All identifier names should begin with a letter. Under no circumstances should an identifier name begin with an underscore. Individual words in compound names are differentiated by capitalization the first letter of each word as opposed to separating with an underscore character. The use of special characters (anything other than letters and digits), including underscores is strongly discouraged. The first 31 characters of an identifier's name (including the prefix) must be unique, due to restrictions found in various platforms and compilers. The uniqueness must not be due solely to a difference in case.

3.3. File Names

File names should be in all lowercase and only contain one period to separate the file extension.

3.4. Function Names

Function names should be an action verb. Boolean valued functions should use the prefix “`is`” as in the boolean function “`isEmpty()`”. In C and C++ all functions must be prototyped, with the prototypes residing in the respective header files.

3.5. Namespaces

Namespace collisions should be minimized without introducing cryptic naming conventions by using C++ `namespace` or Java `package` constructs

4 STYLE GUIDELINES

The primary motivation for style guidelines is to facilitate long term maintenance of software. During maintenance, software developers who are usually not the original authors are responsible for understanding source code from a variety of applications. Having a common presentation format reduces confusion and speeds comprehension. The following guideline specifications are based on principles of good programming practices and code readability. In cases where two or more equally valid alternatives are available, one was selected to simplify specifications.

4.1. Lines

4.1.1. Line Lengths

All lines should be displayed without wrapping on an 80 character display. This also produces readable printouts on most line printers. If lines greater than 80 characters are required, try to break at an operator and start the next line with the operator vertically aligned. For example:

```
cout << “This is an example of a line which must be wrapped, value =”
     << value << end;
```

4.1.2. Statements Per Line

Each statement should begin on a new line.

4.2. Comments

4.2.1. Prologue

The LDAS software development will be maintained under the Concurrent Version System (CVS). As such, most of the information that would typically be found in a prologue such as author, history, dates, etc. will be better managed and accessible through CVS and not take up space and reduce readability of the source files. However, a minimal set of comments in the prologue consisting of a few lines would still serve to as useful orientation to the reader. The prologue will also be the target of any automated comment reading tool and as such, should have a standard format. It should follow the standard Java C style delimiter convention with the keywords :FILENAME:, :PURPOSE:, :REFERENCES: and :NOTES: embedded as in the following example:

```
/**
  * :FILENAME: meaningfulname.(h|c|cc|java)
  *
  * :PURPOSE: A descriptive summary of a couple of lines
  *           intended to give the purpose of this particular
  *           file and its contents.
  * :REFERENCES: A list of useful references that cover the
  *              technical background being modeled by this
  *              software if any.
  * :NOTES: Any additional comments that may be useful in the
  *         prologue.
  */
```

It is important to keep these short, simple and to the point. It is recommended that all file types have a prologue of this format, though some keywords may have no associated text.

4.2.2. Automatic Documentation Comments

For comments meant to be extracted by an automatic documentation tool, follow the Java convention of using the standard C comment delimiters with an extra asterisk on the first one as below:

```
/**
  * This is a module, class, function, or instance variable comment
  * that will be extracted by an automated documentation tool
  */
```

This will provide a consistent look across all source code files and should facilitate creation of automated documentation tools. Such comments should be used to describe classes, methods and global or instance variables.

4.2.2.1 Gotcha Keywords in Comments

Explicitly comment variables changed out of the normal control flow or other code likely to break during maintenance. Embedded keywords are used to point out issues and potential problems. Consider that an automated documentation tool will parse comments looking for keywords, strip them out, and make a report so software developers can make a focused effort where needed.

4.2.2.1.1 Gotcha keywords

- **:TODO:** - means there is more to do here, don’t forget.
- **:BUG: [bugid]** - means there is a known bug here, explain it and optionally give a bug ID.
- **:KLUDGE:** - when you’ve done something ugly say so and explain how you would do it differently next time if you had more time.
- **:TRICKY:** - tell others that the following code is very tricky so don’t go changing it without thinking.
- **:WARNING:** - beware of something.
- **:COMPILER:** - sometimes you need to work around compiler problems, document it! The problem may go away eventually.
- **:ATTRIBUTE: value** - the general form of an attribute embedded in a comment. You make

up your own attributes and they will be extracted.

4.2.2.1.2 Gotcha Formatting

- Make the gotcha keyword the first symbol in the comment.
- Comments may consist of multiple lines, but the first line should be a self containing, meaningful summary.
- The writer's name and the date of the remark should be part of the comment. This information is in the CVS source code repository, but it can take awhile to find it. Often gotchas stick around longer than they should. Embedding date information allows other programmers to make this decision. Embedding author information lets others know who to ask.

4.2.2.1.3 Gotcha Example

```
// :TODO: JKB 971203 - possible performance problem
// Should really use a hash table here but for now I've
// used a linear search.
```

4.2.3. Code Block Comments

Code block comments should precede the block, be at the same indentation level and be separated by a blank line above and below the comment. Brief comments regarding individual statements may appear at the end of the same line, and should be vertically aligned with other comments in the vicinity for readability.

- (C) code block comments should use the standard C comment delimiters `/*` and `*/`.
- (C++, Java) code block comments should use the single line comment delimiter `//`.

4.2.4. Blank Lines

Use a single blank line to separate logical groups of code to improve readability. In source files, use two blank lines to separate each class and each function.

4.3. Formatting

4.3.1. Spacing Around Operators

Spacing around operators and delimiters should be consistent. In general, use one space before and after each operator. This will improve readability. Use spaces inside parentheses around the argument list. Do not use spaces within empty argument lists `()` or non-dimensional arrays `[]`.

- (C++) Do not use spaces around the scope operator `::` .
- (C++, Java) Do not use spaces around the member access operators `.` and `->` .

```
if ( flag == 0 ) {              // correct spacing
if (flag==0){                   // incorrect spacing

void myFunction( int count );   // correct spacing
void myFunction(int count);     // incorrect spacing
```

```
count = timer->GetCount();       // correct spacing
count = timer -> GetCount();     // incorrect spacing
```

4.3.2. Indentation and Braces

The contents of all code blocks should be indented to improve readability. Use 3, 4 or 8 spaces for each level. Tabs should not be used for indentation as different editors give different indentations. Instead use 8 spaces for tabs. Some editors allow automatic substitution of spaces for tabs. Braces should be placed to show the level of indentation of the code block. Examples of clear indentation are as follows:

```
int main()
{
   doSomething();
}

struct MyStruct {
    int x;
    int y;
}
```

4.3.3. Pointers and Reference Positions

In C and C++, all declarations of pointers or reference variables and function arguments should have the dereference operator * and the address-of operator & placed adjacent to the type, not adjacent to the variable. For example:

```
char* sentence;                    // correct
char *sentence;                    // incorrect
char* myFunction( int* count );    // correct
char *myFunction( int *count );    // incorrect
```

4.4. Statements

4.4.1. Control Statements

All control statements should be followed by an indented code block enclosed with braces, even if it only contains one statement. This makes the code consistent and allows the block to be easily expanded in the future. For example:

```
if ( count == 0 )                         //
{                                         //
   myFunction( count );                   // correct
}                                         //

if ( count == 0 ) myFunction( count );   // incorrect
```

4.4.2. Conditional Statements

Conditional statements found in `if`, `while` and `do` statements should be explicit based on the data type of the variable being tested as in the following:

```
int myValue = getValue();
if ( myValue == 0 ) {              // correct test
   doSomething();
}

if ( !myValue ) {                  // incorrect test
   doSomethingElse();
}

bool myFlag = getValue();          // could be a boolean also
if ( !myValue ) {                  // correct test
   doSomethingElse();
}
```

When using the `switch` statement, each `case` should have an associated `break`. Falling through one `case` statement into the next `case` statement should be permitted as long as a comment is included. The `default` case should always be present and trigger an error if it is somehow reached when it should not be. For example:

```
switch ( myValue ) {
    case 1:
           doSomething();
           break;
    case 2:
           // case 2 and 3 require the same action, fall through
    case 3:
           doSomethingElse();
           break;
    default:
            defaultReached();  // trigger error if shouldn't be here
            break;
}
```

4.4.3. Include Statements

For C and C++, in both source and header files, all `#include` statements should be grouped together near the top of the file after the file guards and/or prologue as indicated in Table 3. Includes should be logically grouped together, with the groups separated by a blank line. System includes should use the `<sysheader.h>` notation, while application and all other includes should use the `"appheader.h"` notation. Path names should never be explicitly used in #include statements (with the exception of vender library files such as Motif), since this is inherently non-portable. Here are some examples:

```
#include <stdio.h>             // all
#include <stdlib.h>            // of
#include <Xm/Xm.h>             // these
```

```
                              // are
#include “meaningfulname.h”   // correct


#include </proj/util/MeaningfulName.h>    // these
#include <stdlib.h>                       // are
#include </usr/include/stdio.h>           // all
#include “Xm/Xm.h”                        // incorrect
```

4.5. Declarations

4.5.1. Variable Declarations

Each variable should be individually declared on a separate line. Variables may be grouped by type, with groups separated by a blank line. Variable names should be aligned vertically for readability. There is no required ordering of types, however some platforms will give optimal performance if declarations are ordered from largest to smallest types (e.g. `double`, `long`, `int`, `short`, `char`). For example:

```
double  x;          // correct
double  y;          //
double  z;          //
long    i;          //
int*    j;          //
short   k;          //
char    a;          //

double xArray[10];  // new grouping, correct also
long   iArray[10];  //
char   string[80];  //

int*  a, b, c;      // incorrect, not individually declared
int*  a,            // also incorrect, not individually declared
      b,            // even though variables appear on separate
      c;            // lines
```

The two incorrect `int*` examples are prone to misinterpretation. Notice that `a` is declared as a pointer to `integer` type and `b` and `c` are declared as integers. They are not all declared as pointers.

4.5.2. External Variable Declaration

In C and C++ all external variables should be placed in header files. In general, the use of global variables is discouraged. Use the following method to allow external variables to be created only once while using a single declaration; In the header file which declares the global variable, use a flag to cause the default action on inclusion to be a reference of an externally created variable. Only in the source file that needs to actually create the variable will this flag be defined.

In the header file meaningfulname.h,

```
#ifdef MEANINGFUL_NAME_INIT     // flag called MEANINGFUL_NAME_INIT
#define EXTERN                  // create the variable (only in main)
#else
#define EXTERN extern           // just a reference (default)
#endif
EXTERN int intVariable;         // determine file scope of variable
#undef EXTERN
```

All of the source files should include this header file as normally done:

```
#include “meaningfulname.h”
```

while the following should appear only in the source file where you actually need to declare the variable and allocate memory for it (typically in the main source file).

```
#define  MEANINGFUL_NAME_INIT
#include “meaningfulname.h”
#undef   MEANINGFUL_NAME_INIT
```

4.5.3. Numeric Constant Declaration

Use only the uppercase suffixes (e.g. `L`, `X`, `U`, `E`, `F`) when defining numeric constants. For example:

```
const int     I_VALUE = A73B2X;    // correct, hexadecimal constant
const double  D_VALUE = 1.23E9;    // correct, scientific notation

const float   F_VALUE = 1.23e9;    // incorrect use of lowercase
```

4.5.4. Enumerated Type Declaration

In C and C++ the `enum` type name and enumerated constants should each reside on a separate line. Constants and comments should be aligned vertically. The following is an example of the correct `enum` declaration:

```
enum CompassPoints {     // enum type used to specify direction
        North = 0,       //
        South = 1,       // each enum constant defined!
        East  = 2,       //
        West  = 3        //
};
```

4.5.5. Struct and Union Declaration

In C and C++ the `struct` type name and structure members should each reside on a separate line. This format separates the members for easy reading, is easy to comment, and eliminates line wrapping for large sets of members. Each `struct` should have a one line description on the same line as the type name. Each member should have a comment describing what it is and units if applicable. Members and comments should be aligned vertically. The following is an example of the correct `struct` declaration:

```
struct AgregateData {           // a structure of some data
        int     firstInt;       // the first integer
        int     secondInt;      // the second integer
        double  firstDouble;    // the first double
        double  secondDouble;   // the second double
};
```

Similar formatting applies to the `union` type and its members.

4.5.6. Class Declaration

In C++ and Java, all class definitions should include a constructor, (virtual) destructor, copy constructor and operator=. If the class has a pointer, provide a deep copy constructor which allocates memory and copies the object being pointed to, not just maintains a pointer to the original object. If any of these four are not currently needed, create stub versions and place them in the private section so they will not automatically be generated, then accidently used (this protects from core dumps). All classes should have public, protected and private access sections declared, in that order. Friend declarations should appear before the public section. All member variables should be either protected or private. It is recommended that definitions of inline functions follow the class declaration, although trivial inline functions (e.g. `{}` or `{ return x }`) may be defined with in the declaration itself. Each member function should be commented in the standard fashion as for regular functions. Member variables should each have a one line description. Members and comments should be aligned vertically. For example:

```
class MyClass : public BaseClass {
public:
        MyClass();                            // default constructor
        MyClass( const MyClass& myObject );   // copy constructor
        ~MyClass();                           // destructor
        void setValue ( int newValue );       // set member function
        int getValue();                       // get member function

protected:
        void incrementValue();                // increment method

private:
        int value;                            // private class data

};
```

5 RECOMMENDED PROGRAMMING PRACTICES

5.1. Placement of Declarations

Local variables can be declared at the start of the function, at the start of a conditional block, or at the point of first use. However, declaring within conditional blocks or at the first use may yield a performance advantage, since memory allocation, constructors, or class loading will not be performed at all if these statements are not reached.

5.2. Return Statements

Where practical, have only one return from a function or method as the last statement. Otherwise, minimize the number of returns. Highlight dispersed returns with comments and/or blank lines to keep them from being lost in other code. Multiple returns are generally not needed except for reducing complexity for error conditions or other exception conditions.

5.3. Casts

Avoid the use of casts except where unavoidable, since this can introduce run-time bugs by defeating compiler type-checking. When working with third party libraries such as X or Motif, one often is required to use casts. If needed, document all casts clearly, including reason.

5.4. Literals

Use constants instead of literal values whenever possible. For example:

```
const double PI = 3.14159265359;            // recommended
const char APP_NAME = “ACME Wordprocessor”  // recommended

area = 3.14159265359 * radius * radius;     // not recommended
cout << “ACME Wordprocessor” << endl;       // not recommended
```

5.5. Explicit Initialization

In general, explicitly initialize all variables before use. It is very strongly recommended that you initialize all pointers to either 0 or an object. Do not allow a pointer to have garbage in it or an address in it that will no longer be used.

5.6. Constructs to Avoid

In C and C++, the use of `#define` constants is strongly discouraged, using `const` is recommended instead.

The use of `#define` macros is strongly discouraged, using `inline` functions is recommended instead.

The use of `typedef` is discouraged since in actual fact, types such as `class`, `struct`, or `enum` would be better choices.

The use of `extern` (e.g. global) variables is strongly discouraged. The exception is for programs which benefit from having a small number of object pointers accessible globally via `extern`.

The use of the `goto` statement should not be allowed.

5.7. Macros

In C and C++, all arguments to macros should be enclosed in parentheses to eliminate ambiguity on expansion as in the example:

```
#define MAX( x, y )        ( (x) > (y) ) ? (x) : (y) )
```

5.8. Debug Compile-Time Switch

For C and C++ code used only during development for debugging or performance monitoring, the code should be conditionally compiled using `#ifdef` compile-time switches. The symbols to use are `DEBUG` and `STATS` respectively. Debug statements announcing entry into a function or member function should provide the entire function name including the class. For example:

```
#ifdef DEBUG
cout << “MeaningfulName::doSomething() about to do something” << endl;
#endif
```

The use of the C and C++ preprocessor __FILE__ and __LINE__ directives are also very useful for tracking the source file and line number at which errors or unusual conditions exist in the software at run-time. As such these should be made a permanent part of the code and not optionally compiled only when the code is being developed.

It is also recommended that each source file declare the Revision Control System (RCS) file local char array named `rcsid` to be maintained by CVS as follows:

```
static const char rcsid[] = “$Id:$”;
```

This will be filled in by CVS when the source file is checked out and the constant `rcsid` can then be used in messages, warnings and errors just as with any other constant character array.

5.9. Memory Management

In C++ use `new` and `delete` instead of `malloc`, `calloc`, `realloc` and `free`. Allocate memory with `new` only when necessary for variables to remain after leaving the current scope. Use `delete []` operator to deallocate arrays (the use of `delete` without the array operator to delete arrays is undefined). After deletion, set the pointer to zero to safeguard possible future calls to `delete`. C++ guarantees that `delete 0` will be harmless.

5.10. Constructors

In C++ all class constructors should initialize all member variables to a known state. This implies that all classes should have a default constructor (i.e. MyClass();) defined. Providing a deep copy constructor is strongly recommended. If the programmer wishes not to fully implement a copy constructor, then a stub copy constructor should be written and placed in the private section so no one will accidently call it.

5.11. Destructors

In C++ all classes which allocate resources which are not automatically freed (e.g., have a pointer variable) should have a destructor which explicitly frees the resources using `delete`. Since any class may someday be used as a base class, destructors should be declared virtual even if empty.

5.12. Argument Passing

In C++, if the argument is small and will not be modified, use the default pass by value. If the argument is large and will not be modified, pass by `const` reference. If the argument will be modified, pass by reference. For example:

```
void A::method( int notChanged );       // default: pass by value

void B::method( const C& bigReadOnlyObject ); // pass const reference

void C::method( int& result );          // result passed by reference
```

5.13. Default Arguments

In C++, where possible, use default arguments instead of function overloading to reduce code duplication and complexity.

5.14. Overriding Virtual Functions

In C++, when overriding virtual functions in a new subclass, explicitly declare the functions virtual. Although not required by the compiler, this aids maintainability by making clear that the function is virtual without having to reference the base class header file.

5.15. Constant Member Functions

In C++, it is recommended that all member functions which do not modify the member variables of an object be declared `const`. This allows these functions to be called for objects which were either declared as `const` or passed as `const` arguments. Also in C++, it is recommended that all member function parameters be declared `const` when possible.

5.16. Referencing Non-C++ Functions

In C++ use the `extern “C”` mechanism to allow access to non-C++ (not just C) functions. This mechanism disables C++ name mangling, which allows the linker to resolve the function references. For example:

```
extern “C” {
    void nonCppFunction();   // single non C++ function prototype
}
extern “C” {
#include “nonCppFunctions.h” // library of non C++ functions
}
```

5.17. NULL Pointer

In C++ use the number zero (0) instead of the `NULL` macro for initialization, assignment and comparison of pointers. The use of `NULL` is not portable, since different environments may define it to be something other than zero (e.g., `(char*)0`).

5.18. Enumerated Types

In C and C++, use enumerated types instead of numeric codes. Enumeration improves robustness by allowing the compiler to perform strong type-checking and are also more readable and maintainable.

5.19. Termination Stream Output

In C++, use the `iostream` manipulator `endl` to terminate an output line, instead of the newline character, `\n`. In addition to being more readable, the `endl` manipulator not only inserts a newline character, it also flushes the output buffer.

5.20. Object Instantiation

In C++ and Java, where possible, move object declarations and instantiations out of the loops, using assignment to change the state of the object at each iteration. This minimizes overhead due to memory allocation from the heap.

5.21. Encapsulation

In C++ and Java instance variables of a class should not be declared `public`. Open access to internal variables exposes structure and does not allow methods to assume values are valid. In C++, putting variables in the `private` section is preferable over the `protected` section for more complete encapsulation. Use get and set methods in either `protected` or `public` if needed.

5.22. Default Constructor

In Java, where possible, define a default constructor (with no arguments) so that objects can be created dynamically using `Class.newInstance()`. This exploits the power of Java to dynamically link in functionality that was not present at compile time.

5.23. Import Packages

In Java use full package names instead of wildcards when importing to improve comprehensibility and provide context.

5.24. Exception Handling

In general in C++, avoid exception handling. It is sometimes needed for third party code, but in general, use return values instead. If you need it, document the reasons for using it in the code.

6 FURTHER READING

1. “Enough Rope to Shoot Yourself in the Foot, Rules for C and C++ Programming”, by Allen I. Holub, McGraw-Hill
2. Ruminations on C++, a decade of programming insight and experience”, by A. Koenig & B. Moo, Addison Wesley
3. “Rules & Recommendations, Industrial Strength C++”, by Mats Henricson & Erik Nyquist, Prentice Hall