LASER INTERFEROMETER GRAVITATIONAL WAVE OBSERVATORY

-LIGO-

CALIFORNIA INSTITUTE OF TECHNOLOGY
MASSACHUSETTS INSTITUTE OF TECHNOLOGY

Technical Note **LIGO-T010072-00-E- 00- R** 9/27/2000

LIGO Data Analysis System Software Specification for C++

James Kent Blackburn, Philip Charlton

California Institute of Technology
LIGO Project - MS 18-34
Pasadena CA 91125
Phone (626) 395-2129
Fax (626) 304-9834
E-mail: info@ligo.caltech.edu

Massachusetts Institute of Technology
LIGO Project - MS NW17-161
Cambridge, MA 02139
Phone (617) 253-4824
Fax (617) 253-4824
E-mail: info@ligo.mit.edu

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

Contents

1 INTRODUCTION

The purpose of these coding standards is to facilitate the maintenance, portability and reuse of custom C++ 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 names and extensions

Source filenames may only contain alphanumeric characters and must begin with a letter. A mixture of upper and lower case letters may be used. Valid extensions are:

`.hh` – header files.

`.cc` – source files.

`.icc` – “included” source files.

For source and header files which define a single class (or group of related classes derived from a common base) it is recommended that the filename prefix be identical to the class name, including any capitalization. For example, a class `ExampleClass` would be declared in a file called `ExampleClass.hh` and implemented in a file called `ExampleClass.cc`.

2.2 Header file layout

Header files should contain the following components in the order indicated in the list below:

1. Include guard.
2. System includes eg. `#include <iostream>`
3. Application includes eg. `#include "HeaderFile.hh"`
4. Preprocessor directives such as `#define` for constants or macros.
5. Forward declarations.
6. External function declarations.

7. External variable declarations.
8. External constant declarations.
9. Struct declarations.
10. Class declarations.
11. Inline function definitions.
12. Include for `.icc` file if necessary eg. `#include "FileName.icc"`
13. Close for include guard.

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.

2.2.1 Include guards

All header files should contain a file guard to prevent multiple inclusion and improve compilation times. The name for the file guard will be the header file name written in all capitals. Periods will be replaced by underscore characters.

Example

```
    #ifndef FILENAME_HH
    #define FILENAME_HH

    <file body>

    #endif // FILENAME_HH
```

2.3 Included source file layout

The main purpose of included source (`.icc`) files is to contain long inline functions and template definitions (see §5.11). The use of `.icc` files should be avoided wherever possible. Generally, functions should only be declared inline if they are short (usually no more than three lines), in which case they should be defined in header files after the class declaration. The only exception is template functions, which in some circumstances need to be included by source files which use them. If an `.icc` file is used it should have the following layout:

1. Include guard of the form

```
#ifndef FILENAME_ICC
#define FILENAME_ICC
```

2. System includes eg. `#include <iostream>`
3. Application includes eg. `#include "HeaderFile.hh"`
4. Inline function definitions.
5. Close for include guard of the form

   ```
   #endif // FILENAME_ICC
   ```

2.4 Source file layout

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

1. System includes eg. `#include <iostream>`
2. Application includes eg. `#include "HeaderFile.hh"`
3. Header file for this implementation eg. `#include "FileName.hh"`
4. Preprocessor directives such as `#define` for constants or macros.
5. Local class or struct definitions.
6. External function definitions.
7. External variables definitions.
8. External constant definitions.
9. Static function definitions.
10. Static variable definitions.
11. Static member definitions.
12. Public member function definitions.
13. Protected member function definitions.
14. Private member function definitions.

3 NAMING CONVENTIONS

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

Identifier	**Example**
`#define` or macro	`MEANINGFUL_NAME`
typedef or enum	`MeaningfulName`
union or struct	`MeaningfulName`
global constant	`MeaningfulName`
object or variable	`meaningfulName`
class	`ClassName`
member data	`m_memberData`
function, method	`meaningfulName()`
accessor method for `X`	`X()`
mutator method for `X`	`setX()`
query function for condition `X`	`isX()`
exception class	`MeaningfulException`

3.1 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 capitalizing 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 identifiers 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.

3.2 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. As a general rule, the larger the scope of a variable, the longer and more descriptive its name should be.

3.3 Function names

Function names should be an action verb. Boolean valued functions should use the prefix `is` as in the Boolean function `isEmpty()`. All functions must be prototyped, with the prototypes residing in the appropriate header files.

3.4 Namespaces

Name collisions should be minimized by using namespaces.

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 immediately preceding an operator and start the next line with the operator vertically aligned.

Example

```
cout << "This is an example of a line which must be wrapped "
     << value << endl;
```

4.1.2 Statements per line

Each statement should begin on a new line.

4.1.3 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.

4.2 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 style delimiter convention with the embedded keywords `:FILENAME:`, `:PURPOSE:`, `:REFERENCES:` and `:NOTES:`.

Example

```
//
// :FILENAME:   $Id$
//
// :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

This section needs to be rewritten

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 comm
  * 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.

4.2.3 “Gotcha” keywords in comments

Variables changed out of the normal control flow, or other code likely to break during maintenance, should have explicit comments. Embedded keywords are used to highlight important issues and potential problems. Consider that an automated documentation tool will be used to scan comments for keywords, strip them out, and make a report so that software developers can make a focused effort where needed.

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.

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.

Example

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

4.2.4 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 comments should use the comment delimiters `/* ... */`. C++ comments should use the single line comment delimiter `//`.

4.3 Formatting

4.3.1 Include directives

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 `#include <sysheader.h>` notation, while application and all other includes should use the `#include "appheader.h"` notation. Path names should never be explicitly used in `#include` statements (with the exception of vendor library files such as Motif), since this is non-portable.

Example

```
   #include <iostream>                    // all
   #include <stdlib.h>                    // of
   #include <math.h>                      // these
   #include <Xm/Xm.h>                     // are
   #include "FileName.hh"                 // correct

   #include </proj/util/FileName.hh>      // these
   #include "stdlib.h"                    // are
   #include </usr/include/stdio.h>        // all
   #include "Xm/Xm.h"                     // incorrect
```

4.3.2 Indentation and braces

The contents of all code blocks should be indented to improve readability. Use 3, 4 or 5 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.

Example

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

int main()
{
    if (condition)
    {
        doSomething();
    }
    else
    {
        doSomethingElse();
    }
}
```

The "Kernighan & Ritchie" style of placing braces eg.

```
if (condition) {
    doSomething();
}
else {
    doSomethingElse();
}
```

is also acceptable, but consistency within each source file must be maintained.

4.3.3 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 `[]`. Do not use spaces around the scope operator `::` or the member access operators `.` and `->`, or between a function and the parentheses enclosing its arguments.

Example

```
    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
```

4.4 Declarations

4.4.1 Enumerated types

The `enum` type name and enumerated constants should each reside on a separate line. Constants and comments should be aligned vertically.

Example

```
    enum CompassPoints {
        North,
        South,
        East,
        West
    };
```

4.4.2 Struct and unions

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.

Example

```
    struct AggregateData {
        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.

4.4.3 Classes

When appropriate, class definitions should include a default constructor, (virtual) destructor, copy constructor and assignment operator (`operator=`). If the class has pointer members, provide a deep copy constructor which allocates memory and copies the object being pointed to, not just the value of the pointers. 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 against 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 member variables be prefixed by `m_`. This has the following benefits:

- In member functions, it is immediately obvious which variables are local to the function and which are members (effectively, global to the class).
- Constructors, mutators and accessors can use the same variable names as the members, with the prefix removed.

Definitions of inline functions should appear after the class declaration. Functions should not be declared `inline` within the class declaration, only in the function definition. That way, if a function is changed from inlined to outlined the class declaration doesn't need to be modified.

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.

Example

```
    class MyClass : public BaseClass {
    public:
        // Default constructor
        MyClass();

        // Copy constructor
        MyClass(const MyClass& myObject);
        // Destructor
        ˜MyClass();

        // Assignment operator
        const MyClass& operator=(const MyClass& rhs);

        // Mutator member function
        void setValue(const int value);

        // Accessor member function
        // INCORRECT - function definitions should not
        // appear in the class declaration
        int value() const { return m_value; }

    protected:
         // INCORRECT - functions should not be declared
         // inline in the class declaration
         inline
         void incrementValue();

    private:
         int m_value;
    };

    inline
    void MyClass::setValue(const int value)
    {
        m_value = value;
    }

    inline
    void MyClass::incrementValue()
    {
        m_value++;
    }
```

4.4.4 Numeric constants

Use only uppercase suffixes (eg. L, X, U, E, F) when defining numeric constants.

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
```

4.4.5 Variables and objects

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 (eg. double, long, int, short, char).

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.

4.4.6 Pointers and references

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.

Example

```
    char* sentence;                        // correct
```

```
void  myFunction(const int& count);  // correct

char *sentence;                      // incorrect
void  myFunction(const int &count);  // incorrect
```

4.5 Statements

4.5.1 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.

Example

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

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

4.5.2 Conditional statements

Conditional statements found in `if`, `while` and `do` statements should only test Boolean conditions.

Example

```
const bool valid = isValid();
if (!valid)                         // correct test
{
    doSomethingElse();
}

const int myValue = getValue();
if (myValue == 0)                  // correct test - result is
{                                  // either true or false
    doSomething();
}

if (!myValue)                      // incorrect - testing integer,
{                                  // not boolean
```

```
        doSomethingElse();
    }
```

When using the switch statement, each case should have an associated break. Falling through one case statement into the next case statement is 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.

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;
    }
```

5 RECOMMENDED PROGRAMMING PRACTICES

5.1 Constructs to avoid

The use of `#define` constants is strongly discouraged. Using `const` variables is recommended instead.

The use of `#define` macros is strongly discouraged. Using inline functions and/or template functions is recommended instead.

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

The use of `extern` 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 is not allowed.

5.2 Revision control information

Each source file should declare a `char` array named `rcsid` in the anonymous namespace as follows:

```
namespace {
    const char rcsid[] = "@(#) $Id:$";
};
```

The RCS ID will be inserted between the dollar signs 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. The embedded `"@(#)"` allows the Unix `what` command to extract the RCS ID from the compiled object.

5.3 Type declarations

5.3.1 Forward declaration and header file inclusion

If a source or header file only uses references or pointers to a class, it is often not necessary to include the header file for that class. Instead, the class may be forward declared. This lets the compiler know that the class exists but does not provide any other information. When possible, use forward to declaration to eliminate `#include` directives. This will reduce unnecessary file dependencies and save on compile time.

Example

```
// Forward declaration for Scalar class
class Scalar;

namespace std {
    // Forward declaration for STL valarray - note
    // that it is in the std namespace
    template<class T> class valarray;
};

// Ok - function only uses references
void setVector(valarray<Scalar>& v, const Scalar& scalar);

// INCORRECT - function returns an actual Scalar rather than
// a pointer or reference. Would need to include Scalar.hh
Scalar get(const valarray<Scalar>& v, size_t i);
```

5.3.2 Enumerated types

Use enumerated types instead of numeric codes. Enumerated types improve robustness by allowing the compiler to perform strong type-checking. They are also more readable and maintainable.

5.4 Variable and object definitions

5.4.1 Placement

Local variables can be defined at the start of the function, at the start of a conditional block, or at the point of first use. However, defining 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.

5.4.2 External variables and objects

In general, the use of external global variables is discouraged. If used, an external variable may be declared in multiple compilation units but may only be defined in one compilation unit. It should be declared in a header file using the `extern` storage class, and defined in a single source file in the usual way.

Example
In file `MyModule.hh`:

```
    #ifndef MYMODULE_HH
    #define MYMODULE_HH
    ... etc

    extern int externVar;    // Declaration of externVar - note that
                             // no initial value is assigned

    ... etc
    #endif // MYMODULE_HH
```

In file `MyModule.cc`:

```
    #include "MyModule.hh"
    ... etc

    int externVar = 0;       // Definition of externVar -
                             // initial value is assigned
    ... etc
```

Other compilation units wishing to refer to `externVar` must include `MyModule.hh`. Executables using `externVar` must be linked against `MyModule.o` or a library containing it.

Note that the order of initialization of global objects is left to the implementation, thus the value global objects initialized from other global objects is undefined. However, all global variables are guaranteed to be initialized before `main()` is called. If in doubt, initialize them in `main()`.

Example

```
#include "MyModule.hh"

// INCORRECT - value of myGlobal depends on whether
// myGlobal was initialized before or after externVar
int myGlobal = externVar;

main()
{
    // Ok - externVar has been set by now
    myGlobal = externVar;
}
```

5.4.3 Static objects and the anonymous namespace

Functions and variables are declared `static` or put in an anonymous namespace to render them inaccessible outside of the compilation unit in which they are declared. This means that they would normally be declared in a source file – if declared in a header file they would be accessible to any file that includes the header file. Using the anonymous namespace is preferable to `static` declaration.

Example

```
// Anonymous namespace - the namespace with no name
namespace {
    // A static object definition
    MyClass staticObject;

    // A static function declaration
    void staticFunction(const MyClass& c)
    {
       ...
    }
};
```

Static members (data or functions) still require the `static` keyword. Note that static data members are only *declared* in the class declaration. They must also be defined somewhere, usually in the implementation file for the class. They should not be defined in the header file, otherwise there may be multiple definitions at link time.

Example
In `MyClass.hh`:

```
    class MyClass {
    public:
        static int refCount() const;
    private:
        static int m_refCount;
    };
```

In `MyClass.cc`:

```
    int MyClass::m_refCount = 0;

    int MyClass::refCount() const
    {
        return m_refCount;
    }
```

5.4.4 Literals

Use constants instead of literal values whenever possible.

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
```

5.4.5 Explicit initialization

Explicitly initialize all variables before use. It is very strongly recommended that all pointers be initialized 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.

5.4.6 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 (eg. `(char*)0`).

5.4.7 Const correctness

Any variable or object which will not be modified should be declared `const`. This will help prevent unexpected side effects and guard against calling non-`const` member functions. This rule also applies to function parameters which are passed by value: although changing them won't affect the variable outside the function, declaring them `const` will prevent them from being changed by mistake inside the function. Pointers and/or the objects they point to can also be declared `const`, and should be if they are not to be modified. To declare a `const` pointer, the `const` keyword is placed immediately before the name of the pointer. To declare that the object pointed to is `const`, the `const` keyword is placed before the type of the pointer.

Member functions should also be declared `const` if they don't change the object on which they act (see §5.9.6). Remember that the object on which the function is called is an implicit argument to the function. Declaring the function `const` simply means that the object is implicitly passed by `const` reference.

Example

```
void myFunc1(const int n);      // n can't be changed in myFunc1
void myFunc2(const int& n);     // n can't be changed in myFunc2

const int* p = 0;
p++;                            // Ok
*p = 1;                         // Compiler error - p points to
                                // a const int
int* const p = 0;
p++;                            // Compiler error - p is const
*p = 1;                         // Ok

const char* const s = "Hello";  // Const ptr to const string
s++;                            // Compiler error
s[0] = 'C';                     // Compiler error
```

5.5 Macros

If macros are used, all arguments should be enclosed in parentheses to eliminate ambiguity on expansion. Keep in mind that macros only perform text substitutions and so should not be used in combination with other operations.

Example

```
// Note use of parentheses
#define MAX(x, y)        ( ( (x) > (y) ) ? (x) : (y) )

int x = 10;
int y = 5;

// WARNING - this expands to
//     ( ( (x++) > (y) ) ? (x++) : (y) )
// so x will be incremented twice!
int z = MAX(x++, y);
```

5.5.1 `DEBUG` macro

Code used only during development for debugging or performance monitoring should be conditionally compiled using `#ifdef DEBUG` compile-time switches. Debug statements announcing entry into a function or member function should provide the entire function name including the class.

Example

```
#ifdef DEBUG
cout << "MeaningfulName::doSomething() about to do something" << end
#endif
```

5.5.2 Preprocessor macros

The preprocessor macros `__FILE__` and `__LINE__` 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.

5.6 Casts

Avoid the use of casts except where possible, since this can introduce bugs by defeating compiler type-checking. When working with third party libraries such as X or Motif, one is often required to use casts. If used, document all casts clearly, including the reason. Use C++-style casts (`static_cast`, `reinterpret_cast`, `const_cast`, `dynamic_cast`) in preference to C-style casts.

5.7 Dynamic memory management

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 the `delete []` operator to deallocate arrays (the use of `delete` without `[]` on arrays is undefined). After deletion, set the pointer to zero to safeguard possible future calls to delete. The C++ standard guarantees that deleting 0 is harmless.

In many cases the use of `auto_ptr` can simplify memory management, particularly in constructors and cases where exceptions might be thrown, resulting in memory leaks.

5.8 Functions

5.8.1 Parameter passing

If the parameter is a “small” data type and doesn’t need to be modified, it should be passed by value. If the parameter is a “large” data type, it should be passed by reference. Use a `const` reference if the parameter doesn’t need to be modified. Parameters based on a template argument should almost always be passed by reference or `const` reference, since the size of the object being passed is unknown.

Example

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

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

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

template<class T>
void D::method(const T& someObj);   // Template based parameter
                                    // passed by reference
```

5.8.2 Default arguments

Where possible, use default arguments instead of function overloading to reduce code duplication and complexity.

5.8.3 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.

Many compilers implement return-value optimization. This optimization can be used when an anonymous instance of an object is returned using constructor arguments. Use this method in preference to returning a temporary copy of an object.

Example

```
// Supposing Complex is a class with constructor
//     Complex(re, im);
// and methods real() and imag(), the following
// can use return-value optimization if the compiler
// implements it.
Complex conjugate(const Complex& x)
{
    return Complex(x.real(), -x.imag());
}

// This version can't use return-value optimization
Complex conjugate2(const Complex& x)
{
    Complex tmp(x.real(), -x.imag());
    return tmp;
}
```

5.8.4 Declaring non-C++ functions

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.

Example

```
extern "C" {
    void nonCppFunction(void);   // single non C++ function
}

extern "C" {
#include "nonCppFunctions.h"     // library of non C++ functions
}
```

5.9 Classes

5.9.1 Constructors

Constructors should initialize all member variables to a known state. This implies that all classes should either have a default constructor defined, or else placed in the private section so that it can't be called inadvertently. For efficiency, constructors should use initializer lists rather than assignment to initialize member data.

Providing a deep copy constructor is recommended, unless the default copy constructor provided by the compiler is appropriate. The default copy constructor simply copies each member data from the source to the new object, using the member data's assignment operator. Often this is not the appropriate behaviour, for example if the class has pointer members. In addition, copy constructors should usually call constructors for any base classes of the object being constructed.

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.

5.9.2 Assignment operator

Providing a deep assignment is recommended, unless the default assignment provided by the compiler is appropriate. The default assignment operator copies each member data from the source to the new object, using the member data's assignment operator. The same caveats apply to the assignment operator as to the copy constructor, with the additional provisos that base classes should be assigned and self-assignment should be checked for.

Example

```
class Base { ... };

class Derived : public Base {

    // Copy constructor
    Derived(const Derived& d)
      : Base(d)
    {
        // do the rest of construction
        ...
    }

    // Asignment
    const Derived& operator=(const Derived& d)
    {
        // guard against self-assignment
```

```
            if (this != &d)
            {
                // assign base class
                Base::operator=(d);

                // assign other member data
                ...
            }
            return *this;
        }
    };
```

5.9.3 Destructors

Classes which allocate resources which are not automatically freed (eg. 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.

5.9.4 Object instantiation

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.

5.9.5 Encapsulation

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. Putting variables in the private section is preferable over the protected section for more complete encapsulation. Use accessor and mutator methods in either the protected or public sections if needed.

5.9.6 Const member functions

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 by objects which were either declared as `const` or passed as `const` arguments.

5.9.7 Overriding virtual functions

When overriding virtual functions in a derived class, 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 refer to the base class header file.

5.10 I/O streams

Use C++ streams and insertion/extraction operators in preference to C-style I/O functions such as `printf()`. 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.

Use the standard library classes `ostrstream` or `ostringstream` for conversion of objects to strings (eg. representing a floating point number as a string in preference to `sprintf()`.

5.11 Templates

Ideally, the code for a template class or function would be laid out in the same way as an ordinary class, that is, the template declaration would appear in a header file and the template definition would appear in a source file. In this model, templates are only instantiated at their first use and the appropriate code is only generated as needed. Many compilers provide support for this model using additional compiler passes and repositories for object files generated from templates. Unfortunately, GCC does not yet support external definitions of templates, and so some special handling is required. There are two methods which may be used.

The simplest method is to put the entire template definition into a header file, or into another file which is included by the header file such as an `.icc` file. The advantage of this method is that templates will automatically be instantiated as they are needed. The disadvantage is that the template source code will be included and recompiled each time a file using it is recompiled, even if the template is unchanged.

The second method is to put the template definition in a `.cc` file and make explicit instantiation requests for the template types which are needed. This saves the overhead of recompiling, and also guarantees that only certain template types are usable. The disadvantage is that if a new template type is required, an instantiation request must be added to the template definition and the source file recompiled, otherwise the linker will fail to find symbols for the template member functions. Where possible, this is the method which should be used.

Example
In `TemplateClass.hh`:

```
    // A simple template declaration
```

```
template<class T>
class TemplateClass {
public:
    T getValue() const;
private:
    T m_value;
};
```

In `TemplateClass.cc`:

```
#include "TemplateClass.hh"

// Definitions of member functions
template<class T>
T TemplateClass<T>::getValue() const
{
    return m_value;
}

// Explicit instantiations - only types listed
// here will be usable
template class TemplateClass<int>;
template class TemplateClass<float>;
...
```

5.12 Exceptions

For most purposes where exceptions are needed, exception classes from the standard library should be used. If there is a requirement for a new exception class, for example, when extra information needs to be associated with an exception, it should be derived from a standard library exception. This ensures that all exceptions have some part of their interface in common (at least the `what()` method) and can be caught as a `std::exception`.

6 FURTHER READING

1. Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley.
2. Scott Meyers, *Effective C++*, Addison-Wesley.
3. Scott Meyers, *More Effective C++*, Addison-Wesley.

4. Allen I. Holub, *Enough Rope to Shoot Yourself in the Foot, Rules for C and C++ Programming*, McGraw-Hill.

5. A. Koenig and B. Moo, *Ruminations on C++, a decade of programming insight and experience*, Addison Wesley.

6. Mats Henricson and Erik Nyquist, *Rules and Recommendations, Industrial Strength C++*, Prentice Hall.