CODING STANDARDS AND GUIDELINES FOR GOOD SOFTWARE ENGINEERING PRACTICE IN C++

Kosmas Karadimitriou, Ph.D.

kosmas@computer.org

Last revised: February 25, 2001

Updated versions of this document can be found at:
http://www.csc.lsu.edu/~kosmas/C++guidelines.pdf

TABLE OF CONTENTS

1. INTRODUCTION

Be a software engineer – not a programmer!

Programmer is anybody who knows the syntax of a language and can "throw together" some code that works. However, software engineer is someone who not only writes code that works correctly, but also writes *high quality* code. High quality code has the following characteristics:

- **Maintainability** (easy to find and fix bugs, easy to add new features)
- **Readability** (easy to read)
- **Understandability** (easy to understand – this is not the same as readability, e.g. redundant and repetitive code may be very readable, but not necessarily easily understandable)
- **Reusability** (easy to reuse parts of code, e.g. functions, classes, data structures, etc)
- **Robustness** (code that can handle unexpected inputs and conditions)
- **Reliability** (code that is unlikely to produce wrong results; it either works correctly, or reports errors)

A software engineer's responsibility is to produce software that is a business asset and is going to last for many years. If an engineer produces low quality code that others find it hard to understand and hard to maintain, then it might as well be thrown away and rewritten from scratch. Unfortunately this happens all too often (in the worst case, a company will first lose a lot of money trying to maintain the old code, before it realizes that it would be cheaper to throw it away and completely rewrite it). *Making code readable and maintainable is as important, or more important than making it work correctly!* If it doesn't work, it can be fixed. If it can't be maintained, it's scrap.

This document attempts to establish some good rules and guidelines that will aid engineers in producing quality software. Sometimes, coding standards are being viewed by some people as an unwelcome interference from the management that "limits creativity" in their work. Unfortunately, this way of thinking is a relic from the days when software development was mainly a "hacking" activity, when a single person could write a complete software system. But the situation has certainly changed! Now we are in the age of programming-in-the-large and mega-programming. The development of a single software product now involves tens (sometimes hundreds) of engineers and the effort can last for many years. Even in small companies, standards help to ensure that the software developers communicate more effectively with each other (and with future developers), and that everyone knows and follows the same good rules that improve code quality. It is better for engineers to be creative with algorithms and data structures, rather than being "creative" with the coding style and language tricks.

In addition to the standards, the aim of this document is to also offer some good advice on how to use more effectively the C++ language and avoid some common pitfalls, smoothing out the transition from C to C++.

2. NAMING CONVENTIONS

Overview:

- Names of variables in mixed case, beginning with lowercase.
- Function names and type names (classes, structs, enum, and typedef) in mixed case, beginning with uppercase.
- Names of #defines, constants, and enum values in all uppercase, separated with underscores.
- Global variables start with “g”.
- Class member variables start with “m_”.

2.1 Regular variables

Use mixed case, starting with lowercase. Avoid names that are too short and cryptic, but also avoid extremely long names. Do not use any prefix (e.g. Hungarian notation), except for the global variables and pointers that are discussed next.

2.2 Global variables

Global variables should start with 'g'. The reason is that it is always important to know the scope of a variable, especially for global variables that have dangerously large scope.

Example

gParser, gSimulationClock

2.3 Class names

Use mixed case starting with uppercase. There is no reason to prefix class names with “c” or “C” to indicate that they are classes, since in a C++ program most data types *should be* classes, anyway. Even in the case that you have some non-class types (e.g. struct or enum types), you want to be able to change them to classes easily, without having to modify their names and consequently the client code that uses them. Also note that you don’t gain any real programming benefit by embedding in the name the information that something is a class vs. struct or enum type.

It is OK (and very often, advisable) to declare variables with similar name as the class:

Example

```
FluidSet  fluidSet;
```

Try to limit the number of words you use in class names. Compound names of over three words are a hint that your design may be confusing various entities in your system. Revisit your design and try to see if your objects have more responsibilities than they should (also see 2.11 “Avoid extremely long names”).

2.4 Class member variables

Class member variables should start with “m_”. This increases code readability as it makes it clear what variables are member variables vs. parameters or locals. Also it allows you to reuse member variable names as function parameters. The 'm_' should always precede other name modifiers like 'p' for pointer.

Example

```
class Foo {
public:
```

```
        void Transform(Color  color, Flavor  flavor) {
                m_color = color;
                m_flavor = flavor;
        }
        …
private:
        Color  m_color;
        Flavor  m_flavor;
        String  m_pName;
        …
};
```

2.5 Structure member variables

Do NOT use "m_" for structure data members. One reason for this rule is that structure data members are usually referenced as fooStruct.dataX that does not leave room for ambiguity. Another reason is that usually structures do not include functions, in which "m_" would be useful (if some structure needs to have functions, then probably we should define it as a class instead of structure).

In any case, the use of structures is discouraged (see 6.9 "Use classes instead of structures").

2.6 Function names

Function names are in mixed case, and start with uppercase. They should normally be a verb, or a verb followed by noun. Avoid starting function names with nouns, as it may be confusing to the reader. Usually every method and function performs an action, so the name should reflect this.

Example

CheckForErrors() instead of ErrorCheck()
DumpData() instead of DataDump().
GetLength() instead of Lenth()

This also makes functions and data objects more distinguishable (note that the use of parentheses doesn't always clarify it, since in C++ parentheses can be used even with data objects, e.g. to invoke the constructor, and functions can be used without parentheses, e.g. when passed as arguments).

Example

"RegionSizeEstimate" is ambiguous. It could either refer to "an estimate of the region size", or it could refer to the action "estimate the region size". However, functions generally refer to actions, so putting the verb first makes more sense and clarifies it better:

EstimateRegionSize(), StoreContents(), FindNode(), ReduceVariance()

2.7 Enumeration types

Make sure you differentiate the label names to prevent name clashes with other enums or constants. Don't forget to put a label for an error state. It's often useful to be able to say an enum is not in any of its valid states. Also make a label for an uninitialized or error state that can be used to initialize variables. Make it the first label if possible. Put the labels in different lines, and a comment next to each label unless the name is *really* intuitive.

Example

```
//bad style: names too common, may lead to name clashes, no comments
enum VizState { RUNNING, CLOSED };

//good style: small possiblity of name clashes, comments, labels for error/unitialized state
enum VizState {
        VIZ_STATE_UNDEF, //indicates uninitialized variable
        VIZ_STARTING, //VIZ is still loading the modules
        VIZ_RUNNING, //VIZ is running, no errors
        VIZ_ERROR, //VIZ is in an error state
        VIZ_CLOSED
};
```

Note, however, that a better way to avoid name clashes is to put the enum into a class definition (see 5.10 "Put enums in class definitions").

2.8 Constants

For global constants, or constants declared inside classes, use all uppercase with underscores to separate words. Don't use #define for constants; instead, declare them as const (see 6.3 "Avoid using #define"). This facilitates catching type errors at compile-time.

Example

```
const int FOO_CONSTANT = 78.123;
```

For local variables that are declared as "const" to improve code understandability (see 5.12 "Declare variables that don't change as const"), use the regular naming conventions for variable names.

Example

```
const int dimensions = 3;
```

2.9 Keep names consistent

Avoid using different names (or different abbreviations) for the same concept. For example, instead of using all names "transform", "xform", and "xfrm" in different parts of the program to refer to the same entity, try to choose one name and use it consistently.

2.10 Use pronounceable, non-cryptic names

One rule of thumb is that a name that cannot be pronounced is a bad name. A long name is usually better than a short, cryptic name (however, be aware that extremely long names are not good either! See 2.11 "Avoid extremely long names"). In general, names should be meaningful and descriptive to enhance readability and comprehension.

Example

Instead of:

```
double trvlTm;
double spdRng[2];
Point stPnt;
```

use:

```
double travelTime; // total travel time
double speedRange[2]; // range of speeds during this trip
Point startPoint; //starting point for the trip
```

Put comments that fully clarify the use of each variable, if it's not obvious from the name.

2.11 Avoid extremely long names

Names for identifiers should be chosen so that they enhance readability and writeability. In general, longer names are more readable that shorter, abbreviated ones; however, there is a limit to how long a name can be before it actually starts having *detrimental effect* on readability. Names like the following:

```
fgr_n_caption_pairs_to_n_equation_sequence_numbers
FGR_ STATIC_SEQUENCE_EQUATION_MARKER_TYPE_ID
```

are hard to write, and they also make the code harder to read! How easy it is to read (or write) code that looks like the following?

```
plotter_write_figures_block_info_out(FGR_ DESCENDING_SEQUENCE_MARKER_TYPE_ID);
picture_n_caption_pairs_to_n_linear_sequence_markers(block_info_n_caption_pairs);
plotter_block_info_n_linear_sequence_markers_select(block_info_n_linear_sequence_markers);
```

A much more readable version of the previous would probably look like this:

```
plotter.Write(figures, DESCENDING);
picture.GetLinearMarkers(captionPairs);
plotter.Select(markers);
```

Sometimes, however, you may find yourself in a situation that it seems hard to avoid using long names, especially when you need to distinguish between several functions that do almost the same thing. Well, it turns out that there are a few techniques that you can use to avoid having extremely long names. C++ greatly helps on that. Consider the following C code:

```
#define TRI_SLIP_SURFEL_TYPE_ID                1
#define TRI_MOVING_SLIP_SURFEL_TYPE_ID  2
#define TRI_MASS_FLUX _SURFEL_TYPE_ID    3

void tri_scanner_read_block_info( );
static long tri_scanner_read_block_info_pressures_in( );
static long tri_scanner_read_block_info_forces_out_n( );
static long tri_scanner_read_block_info_vortex_num( );

void tri_scanner_read_block_info( )
{
        if (surfel_type == TRI_SLIP_SURFEL_TYPE_ID)
                tri_scanner_read_block_info_pressures_in( );
        else if (surfel_type == TRI_MOVING_SLIP _SURFEL_TYPE_ID)
```

```
                tri_scanner_read_block_info_forces_out_n( );
        else if (surfel_type == TRI_MASS_FLUX _SURFEL_TYPE_ID)
                tri_scanner_read_block_info_vortex_num( );
}
```

In C++, this could be written as follows:

```
class TriScanner {
public:
        enum SurfelType {
                UNDEFINED,
                SLIP,           // …blah…
                MOVING_SLIP, // …blah…
                MASS_FLUX    // …blah…
        };
        void ReadBlockInfo();
private:
        long ReadPressuresIn( );
        long ReadForcesOut( );
        long ReadVortexNum( );
};


void TriScanner::ReadBlockInfo()
{
        if (surfelType == SLIP)
                ReadPressuresIn( );
        else if (surfelType == MOVING_SLIP)
                ReadForcesOut( );
        else if (surfelType == MASS_FLUX)
                ReadVortexNum( );
}
```

The C++ version has much shorter names for two reasons:

a) The functions are now member functions of a class, therefore they don't need to embed into their names information about the module they belong
b) The enum values are also members of the class, therefore they also don't need to have long names since there is no risk of name clashes (if they need to be used outside the class, then the qualifier TriScanner:: will make them unique).

The bottom line is: use the C++ capabilities to create short names instead of sentence-like names. A rule of thumb is that names should contain about 2-4 words. Names with more than 4 words probably indicate design problems.

3. COMMENTS

3.1 General guidelines

Put always a header comment at every function definition. Use brief comments throughout the code to increase code understandability.

In general, comments should be:

- Brief
- Clear
- Don't contain information that can be easily found in the code
- Explain WHY instead of HOW (except when commenting in the large)

3.2 Comments should be brief

When writing comments, try to be brief and succinct.

Example

Instead of:

```
// The following code will calculate all the missing values of the
// pressure by applying the spline interpolation algorithm on the
// existing pressure values
```

Write:

```
// calculate missing pressures using spline interpolation
```

3.3 Don't restate code with comments

Don't give information in your comments that is obvious from the code.

Example

The following are bad comments (they give information that is obvious from the code):

```
// loop until reaching current time
while (time<currentTime) {
   // if vector is greater than screenMinSize, calculate factor
   if (vector >10) {
      factor = factor + .....
   }
   …
   // increase time
   time = time + 1;
}
```

Here is a better version (comments offer new information, and are not obvious):

```
// calculate the new factor
while (time<current_time) {
   // only vectors visible on screen affect the factor
   if (vector > screenMinSize) {
      factor = factor + .....
   }
   ....
   time = time + 1;
}
```

Another example in which comments are totally unnecessary:

```
// if file is open and it contains more than one buffer …
if (file->IsOpen() && file->GetNumBuffers() > 1) {
        //close the file and ask the file manager to create a new one
        file->Close();
        file = fileMgr.CreateNewFile();
}
```

3.4 Comments should explain WHY instead of HOW

High quality code is by definition readable and clear enough anyway, so that the average programmer can easily understand *how* it works just by reading it. However, the reasons *why* a particular algorithm was chosen, or *why* a particular data structure is used, or *why* a certain action must be taken, usually cannot be derived just be reading the code. This is the information that should be documented in comments throughout the code.

The only case that comments can be at the "HOW" side is when commenting in the large; that is, when a short comment precedes a large block of code to summarize what the code does (but without getting into the details).

Example

```
// use spline interpolation to calculate intermediate forces
{
   ….
}
```

3.5 Placement of comments

Comments should precede the block of code with which they are associated and should be indented to the same level. This makes it easier to see the code structure.

Example

```
while (....) {
    event = ...
    //weather events have priority
    if (event==WEATHER_EVENT) {
        listStart = list->Start();
        listEnd = list->End();
        //put event in the beginning of the list
        for (node=listStart; node<listEnd; node++) {
            ......
        }
    }
}
```

Exception: comments for variable definitions, or comments that are short and apply to a *single* line of code. These can be put at the same line as the code.

3.6 Variable declaration comments

If not completely obvious from the choice of the variable name, describe how the variable will be used.

Example

```
long vectorLength = 0;
Window* prevWin; //window that had the focus previously
```

3.7 Comments should be blended into the code

With the exception of function headers, most other comments should be as close to the code they refer to as possible. This "connects" them better with the code. More importantly, it reduces the risk of the comments going out-of-date when programmers make code changes (if comments are not close to the code they refer to, someone might change the code and forget to update the comment).

Example

Instead of:

```
// … details … blah blah blah blah blah
// … more details… blah blah blah blah
// … even more … blah blah blah blah.
{
        DoThis();
        DoThat()
        while(cond) {
                Foo1();
                Foo2();
                Foo3();
                MoreFoo();
        }
        CheckFoo();
}
```

use:

```
// … brief, overview comment with no details…
{
        // … local details …
        DoThis();
        DoThat()
        // … local details …
        while(notDone) {
                Foo1();
                // … local details …
                Foo2();
                Foo3();
                MoreFoo();
        }
        // … local details …
        CheckFoo();
}
```

3.8 "TODO" comments

Use //TODO: comments to remind yourself and others about work that needs to be done in the code.

Example

```
//TODO: replace these calculations with calls to Screen library
drawGrid[n].width = .....
```

TODO comments also make easy to summarize what needs to be done, by running a global search or grep for "TODO".

3.9 Function headers

Use function header comments in every function that you write. A function header should contain a brief description of what the function does. Avoid putting implementation details in the header (try to focus on WHAT the function does, not HOW it does it). Think of it as a black box, that you have to describe its usage. The header should be the "user's manual" for whoever uses this function. Describe what every input parameter is (unless it's obvious), and what this function returns.

Keep function headers short. Long function headers reduce code visibility. Avoid having empty lines and redundant information. 3-5 lines must be enough in most cases.

Do NOT put implementation details in the header, for example what are the local variables, or which subroutines are called from this function. This increases maintenance cost (function header comments must be modified along with the code; in addition, there's risk that the comments will go out-of-date if programmers forget to update them). If someone wants to find out how the function works, it's better to read directly the code. You can help by providing good and meaningful comments throughout the code (see 3.7 "Comments should be blended into the code").

Example

```
// Stores the 'labData' into the database 'db' which is in the tape with # 'tapeNum'.
// If 'compression' is TRUE, it compresses the data before storing them. Returns
// TRUE if storage operation completes successfully.
bool StoreData(long* labData, Database& db,
                        long tapeNum, bool compression)
{
  .....
}

// Returns the average salary from all employees in the input 'list'.
double GetAverage(EmployeeList& list)
{
  .......
}

// Sets a grid using 'dX' and 'dY' spacing in the X and Y directions respectively.
// Note that if the given dX and dY don't divide exactly the given 'width' and
// 'height', it decreases dX and/or dY until width%dX=0 and height%dY=0.
void SetGrid(long width, long height, long dX, long dY)
{
  .......
}
```

Do not write the function name in the header, it is unnecessary (and it adds overhead if you decide to change the function name). If software tools are used to extract documentation from the code, those tools will probably be smart enough to find what function the header belongs to; so you don't have to state it explicitly.

Example

Instead of:

```
// Function: ArchiveBlock. This function archives the block in the….
void ArchiveBlock(Param p1, Param p2)
{
        …
}
```

do:

```
// Archives the block in the….
void ArchiveBlock(Param p1, Param p2)
{
        …
}
```

Notice that even the words "this function" are unnecessary.

3.10 Preconditions - postconditions

Do NOT write comments about preconditions and postconditions in the function headers or in the function code. Instead, use TRAP statements to verify that your pre- and postconditions are true (see 8.5 “Error handling with TRAP and ERROR”). In this way pre- and postconditions become “alive” and serve a meaningful purpose. Your code will be more reliable and more correct since bugs will be caught earlier. Use TRAP to check input parameters, and check the results of your calculations. Use them even for things that you believe are “obvious”. You will be surprised to discover that things can be much different than what you thought, especially after code modifications.

3.11 Don’t overdo with comments

Too many comments are as bad as too little comments! In most cases, a comments-to-code ratio of 20%-30% should be sufficient. If you find yourself writing as much comments (or more) than code, then probably you overdo it. Too many comments impair code readability, programmers most likely will quit reading them, and in turn, comments are likely to go out-of-date when code changes are made by programmers who simply ignored those huge comment sections. To be effective, comments should be brief and describe something that is not apparent from the code.

4. FILES

4.1 Every class should be in its own file

Every class should be defined in its own header file (.h), and its implementation should be in a separate source file (.cc). Every file (either header or implementation) should contain only one class (in most cases). Sometimes, there may be two or more related classes that could be put in the same file; however this should be the exception, not the rule. Remember that the more classes in a file, the more compile-time dependencies are introduced for clients that need only a subset of the classes.

4.2 Header files (.h)

Enclose all file contents in the standard compiler directives #ifndef - #endif to protect against multiple inclusion.

Put some comments above every class definition that are similar in style to the comments in the function headers. Note that the comments need to be *directly* above the class definition so that automated tools for comment extraction could be used in the future to extract documentation from code. These comments should include a brief description of what the class is about, and any special information that you want the users of your class to be aware of.

Example:

```
#ifndef _CALENDAR_H_
#define _CALENDAR_H_

#include “Appointment.h”
#include “Holiday.h”

class Date; //forward declaration

//---------------------------------------------------------------------------------------------------
// The class Calendar provides all the operations necessary to handle dates for any
// given year. It can subtract dates, report number of holidays between two given dates,
// store and report appointments etc.
//---------------------------------------------------------------------------------------------------
class Calendar {
.....
};

#endif
```

4.3 Forward class declarations

Do not include a header file for a class that is referred to by pointer or reference only. Instead use a forward class declaration. That will reduce file dependencies and improve compilation time for everybody.

Example

Instead of :

```
#include “cat.h”
....
Cat* cat;
```

do:

```
class Cat; // forward class declaration
....
Cat* cat;
```

4.4 Don't #include header files in your header files unless your headers won't compile without them.

Sometimes you see code that #includes in class X’s header file all the header files that are needed for class X’s implementation. This is a not a good idea, as it leads to unnecessary dependencies and increases compilation time. Try to always #include only the headers that are absolutely necessary. Often some headers can be replaced by forward class declarations.

Example

File foo.h:

```
#include “force.h”
class Speed;
class Vector;

class Foo {
private:
        Force  m_force;
        Speed*  m_speedArray;
        Vector&  m_vector;
        ....
};
```

The implications are that if class Speed or Vector are changed, the files that include the header file foo.h probably will not need to be recompiled - only relinked.

4.5 Class definitions

Every class definition should be structured so that the public section appears first, followed by the protected and finally the private section.

When inlining functions, don’t put them inside the class definition unless they fit in only 1 line of code. If not, put them at the end of the header file.

Always declare a copy constructor and assignment operator in your classes to protect from shallow copy (see 7.7 “Protect classes from shallow copy”). If they are not needed in the code, make them private and don’t define a body.

Finally, right above the class definition, put some comments that provide an overview of the class (not implementation details, only information that would be useful to a client of this class) including its relationship to other classes.

Example

```
[Copyright notice goes here.]

#ifndef  _COMPONENT_FILE_H
#define  _COMPONENT_FILE_H  1


//------------------------------------------------------------------------------------------
// Class X is a … It can be used for…
// It can be used together with class Y to …
//------------------------------------------------------------------------------------------
class X {
public:
        X (); // default constructor
        virtual ~X (); // destructor
        int Foo() { /* something small that fits in one line */ }
        int LongerFoo();
protected:
private:
        X (const X& x); // copy constructor – make it public only if needed
        X& operator = (const X& x); // operator equal – make it public only if needed
};


inline int X::LongerFoo()
{
        // something that requires more than one line of code
}



#endif
```

5. WRITING READABLE CODE

5.1 Function length

Try to write small functions, no longer than 2 pages of code. This makes it easier to read and understand them, and it also promotes code reusability and maintainability. It helps you avoid duplicating code by factoring out common code into small functions. It also results in code that is more self-documented since shorter functions usually contain mostly function calls, which presumably have meaningful names that can be a great help for a reader trying to understand that code.

5.2 Function declarations

Don't omit dummy parameter names in function declarations unless the meaning is clear without them. Also always include parameter names when you have more than one parameter of the same type; otherwise it's impossible to figure out which one is which.

Example

Instead of

```
void DrawCircle(Color, long, long, double);
```

do:

```
void DrawCircle(Color, long centerX, long centerY, double radius);
```

5.3 Avoid having too many parameters in functions

Functions that take too many parameters often indicate poor code design and poor usage of data structures. If classes are used properly, then there is no need to pass many parameters to member functions. Think of a class as a storage place that stores data between function calls. In this way, its member functions should be able to find inside the class most of the data they need.

Too many parameters may also indicate limited/poor usage of data structures. For example, instead of passing separately the coordinates of a line as x1,y1,z1,x2,y2,z2, consider creating a class Line that contains these coordinates. Then you will only need to pass one parameter (a Line object) instead of six parameters. As another example, instead of having a function:

```
PrintImage(image, printer, pageWidth, pageHeight, inkColor, tonerQuality, resolution, copies);
```

consider creating and using a class PrintSetup that contains all printing-associated parameters, and pass this to the function:

```
PrintImage(image, printer, printSetup);
```

Even better, in C++ you could put all printer-related information into a Printer class, and then call the printImage function as follows:

```
printer.PrintImage(image);
```

5.4 Avoid deeply nested code

Avoid having too many levels of nesting in your code. As a rule of thumb, try to have no more than 5 nesting levels. If your code needs more nesting, consider creating a function for some of the nested code. Code that looks like this:

```
                        }
                    }
                }
            }
        }
    }
  }
}
```

should automatically raise a red flag in your mind. One way to limit yourself from using too many nesting levels is to use 3 or 4 spaces for indentation. In this way, every time you nest too deeply, your code will approach the right margin faster and so you'll know that it's time to think about organizing better the code.

5.5 Use 3 or 4 spaces for indentation

Using 3 or 4 spaces for indentation makes the code more readable that when only 2 spaces are used. You may be tempted to disagree with the argument that more indentation spaces would make it difficult to have many nesting levels in the code. *Exactly*! That's the point. The code shouldn't nest too deeply (see 5.4 "Avoid deeply nested code"). Another opposing argument might be that using 3 or 4 spaces for indentation makes it hard to fit function parameters and long expressions in one line. Well, think about it. In code that is not deeply nested, using 2 vs. 4 spaces of indentation doesn't make much difference, anyway. On the other hand, having trouble fitting function calls and expressions in one line may indicate deeper problems that simply the indentation width. These problems include:

a) Function and/or parameters names that are too long (see 2.11 "Avoid extremely long names")
b) Functions that have too many parameters (see 5.3 "Avoid having too many parameters in functions")
c) Expressions that are too long
d) Code that is too deeply nested (see 5.4 "Avoid deeply nested code")

You should avoid having names that are extremely long, as this decreases writeability *and* readability of your code. Also a function that takes too many parameters may indicate poor code design and poor usage of data structures. Try to think about grouping parameters into data structures/classes that can be passed easily as one object. Finally, breaking long expressions into shorter ones makes the code more readable.

To summarize, 3-4 spaces for indentation is preferable than 2 spaces for the following reasons:

a) Code becomes more readable
b) It discourages deeply nested code
c) It discourages extremely long identifier names
d) It discourages having functions with too many parameters
e) It discourages long expressions

5.6 Curly braces

The starting brace should be placed at the end of the parent code line. The closing brace at the same indentation level as the parent code line. It's good to use braces even when there is only one line of code to be enclosed (this makes it easier to add additional lines of code if needed).

Example:

```
if (temperature>70) {
    for(int n=0; n<100; n++) {
        amount = x – y * n;
    }
}
```

An exception to setting the starting brace at the parent code line is for function definitions. In this case, put both braces at the beginning of the line.

Example

```
void Foo(long x, double y)
{
   //.....function body.....
}
```

Another exception is when the parent code line does not fit in one line. In this case, use whatever makes it more readable.

Example

```
if (veryLongExpression < anotherVeryLongExpression
   || veryLongExpression >= oneMoreVeryLongExpression)
{
    //...do this...
    //...do that...
}
```

Do not put unnecessary semicolons after a closing brace.

Example

```
while(…) {
        …
};  //semicolon is unnecessary here
```

5.7 Never use goto

This rule has been established long ago, when the structured programming paradigm replaced the spaghetti code style that was prominent in old programs. However, even today there are programmers that still use it. It is extremely rare that using goto improves code readability; therefore, its use should be completely avoided. If a piece of code seems it needs to use a "goto", then this is an indication that probably that code has poor structure and design. Instead of using goto as a shortcut, more time should be spent on improving the code quality. For example, in cases that using goto seems to be an easy solution for jumping out of a complicated loop, consider using a function instead.

5.8 Use sparingly "break" and "continue" in loops

Loops are more readable if the conditions for termination (or continuation) are either at the beginning or at the end of the loop. Using "break" and "continue" is like using a "goto" inside the loop. There are a few cases though that the use of "break" may help you avoid introducing extra flags, therefore may lead to cleaner code. Use your judgment on a case-by-case basis.

5.9 Avoid returning from the middle of functions

Avoid returning from the middle of functions; this creates "jumps" in the code logic and makes the code more difficult to understand and modify. Try to always return either from the beginning or from the end of functions. Return from the beginning if, after checking the parameters, you find that the call was unnecessary; otherwise arrange the function logic so that the function will return from the end.

5.10 Put enums in class definitions

Putting enum definitions inside classes takes them out of the global namespace; therefore there is no risk of name clashes.

Example

```
class House {
public:
        enum Pet { UNDEF, DOG, CAT, HORSE, RAVEN, IGUANA };
    ...
};
…
House::Pet myPet = Foo::HORSE;
```

5.11 Declare and initialize variables where they are first used

Variables should be declared and initialized where used. That is, the C practice of declaring all variables to the top of a function body is discouraged. Also try to always initialize variables in their declaration.

Example

Instead of:

```
long n;
// ….other irrelevant code here….
n = Foo();
```

do:

```
// ….other irrelevant code here..
long n = Foo();
```

If the value of a variable is not supposed to change, then also declare it as const (see 5.12 "Declare variables that don't change as const").

5.12 Declare variables that don't change as const

When you declare variables that are only assigned a value once and then they don't change, declare them as const to improve code understandability.

Example

Instead of:

```
long a = Foo();
// … code that only reads “a” but don’t change it…
```

do:

```
const long a = Foo();
// … code that only reads “a” but don’t change it…
```

This can be a great help for someone that tries to debug or reason about the code. When a variable is declared as const, this is an assurance that its value will never change; therefore it is unnecessary to go into functions that use it to verify that they don’t change it.

5.13 Use the “virtual” keyword for all virtual methods

Once a method is declared as virtual in a base class, all derived class methods with the same type signature are virtual, even without the virtual keyword. However, you should carry the “virtual” keyword through derived classes even though it is redundant (it helps readability).

6. WRITING MAINTAINABLE CODE

6.1 Don't duplicate code

Duplicate code is one of the greatest evils in programming; it is the worst form of code reuse. If you find yourself having the same code in two or more functions, pull the duplicate code out of those functions and create a new function with it; then just call this new function. Sometimes it is difficult to create a function out of the duplicate code; in those cases you can put it in a macro. However you should first try hard to create a function, because functions are in many ways superior to macros.

A common reason that programmers produce duplicate code is time pressure: "now we don't have enough time, so let's cut-and-paste for the moment, and later on we'll come back to fix it". Don't fall into this trap! It's always better to do it right the first time, mainly for two reasons:

a) the chances are that you will never come back to fix this code! Probably you will always be too busy fixing bugs or developing new code. Going back and changing working code is always a low priority, so probably it will never make it high enough in the priority list.
b) if you don't have enough time, then cut-and-paste won't buy you more time, on the opposite, it will probably cost you more time, even in the short term! Don't forget that we are talking about code under development that changes constantly; changes will have to be repeated in all parts of the duplicate code, bugs found in one duplicate part will have to be corrected everywhere, code becomes lengthier and more difficult to work with, etc. So, *do it right the first time!*

The main advantages of avoiding duplicate code are the following:

a) Modifications are easier because you only need to modify code in one location
b) Code is more reliable because you'll have to check only one place to ensure the code is right
c) Bugs are reduced. Programmers who will work your code in the future will have to fix bugs in only one place; but if there's duplicate code, then it is very common that programmers fix the bug in one place and forget to fix it in others, so bugs remain in the system
d) When putting duplicate code is in its own function, the calling code becomes shorter and in turn, more readable and understandable. For example, compare this:

```
if (node != NULL) {
        while (node->next != NULL) {
                node = node->next;
        }
        leafName = node->leaf.GetName();
}
else {
        leafName = "";
}
```

with this:

```
leafName = GetLeafName(node);
```

e) The code can become more efficient because the size of your program becomes smaller. When you replace many lines of code with one function call, more code fits into the memory and page faults can be decreased. A trip to the disk may cost as much as 2,000 function calls, so avoiding even one extra page fault can make a difference
f) Code reuse is promoted, because it is much easier for a programmer to find a function he needs and simply call it, rather than searching the code and try to cut and reuse pieces of it.

6.2 Avoid global constants

Avoid global constants that clutter the global namespace. It is much better to limit the scope of constants by declaring them inside classes. If there is a good reason for wanting them to be globally visible, then you can declare them in the "public" section of the class.

6.3 Avoid using #define

You should avoid using #define as much as possible. In C, there are three main cases where someone would use #define:

a) To define a constant
b) To define a set of related constants
c) To define a macro.

In C++, you should try to replace #define with something that the C++ compiler actually understands so that it can do better error checking: use const for constants, enum for related constants, and inline functions instead of macros.

Example
Instead of:

```
#define PI 3.1415

#define YELLOW 0
#define ORANGE 1
#define BLUE 2

#define GET_COLOR(table, color) {        \
        table_load(table);               \
        table_check(table, color);       \
        ….                               \
}
```

use:

```
const double PI = 3.1415;

enum Color {
        UNDEF,  //color is still undefined
        YELLOW,
        ORANGE,
        BLUE
};

inline void GetColor(Table& table, Color color) {
        table.Load();
        table.Check(color);
        ….
}
```

6.4 Avoid having public data members

Avoid as much as possible public data members. All data members should be private or protected. This allows class implementations to change easily without breaking client code. Also functions that return or provide access to data members should also be avoided for the same reasons. Try to hide class implementation; its interface shouldn't depend/reveal how the class is implemented. The clients of your class should not be aware or depend of what algorithms or what data structures you have used.

Example
Instead of this:

```
class TriangleCollection {
public:
        Array m_array;  // BAD!! clients now directly access your data
        GetTriangle(long index); // BAD!! clients now write code about an array
                                 // If you decide to use another data structure, (e.g. hash
                                 // table) then your clients will have to change their code
};
```

do this:

```
class TriangleCollection {
public:
        GetTriangle(Point center); // now clients cannot write code that depends on the
                                   // data structure that you're using, so you can change the
                                   // array into a hash table or list – clients won't be affected
private:
        Array m_array; // now only you control your data; no one else
};
```

6.5 The use of get and set

To provide access to the data members of a class, use the convention of "get" and "set" functions.

Example

```
class Image {
public:
        long GetWidth() { return m_width; }
        void SetWidth(long width) { m_width = width; }
private:
        int m_width;
};
```

Some people use "set" but not "get", for example:

```
Width() { return m_width; }
SetWidth(int width) { m_width = width; }
```

Even though this may read a little better, it actually doesn't help to search or grep for the accessor functions of a class. Therefore prepending the accessor functions with the word "get" is preferable.

A word of caution here: public get/set functions should be used as little as possible, as they provide access to the class's data members. Quality software is based on encapsulation, information hiding, and

interfaces - these are basic principles in software engineering. For an in-depth discussion consult any software engineering textbook. See 6.4 “Avoid having public data members” and 6.6 “Don’t overdo with get and set functions”.

6.6 Don’t overdo with get and set functions

Be careful about what get and set functions you provide. The reason is that get/set functions often expose data members and implementation to the world. In that case, they are almost equivalent to having the data member declared as public. Therefore be cautious as to what get/set functions you provide, and don’t provide more than necessary. Also avoid revealing class implementation details through these functions.

Example

```
class TotalForce {
public:
        ...
        double GetForce(int index) { return m_forces[index]; }
        void SetForce(double force, int index)  { m_forces[index] = force; }
        ....
private:
        ....
        double m_forces[MAX_MEAS];
        ....
};
```

In this example, “m_forces” has been declared as a private array. However, the get and set functions provide full access to its elements, therefore you lose most advantages of encapsulation. Now clients know that you use an array for its implementation and they can access directly its elements.

6.7 Don’t write switch statement based on the type of an object

This is exactly one of the situations that C++ was designed to improve. The three main drawbacks of the switch statements are:

a) Code may be duplicated between different cases
b) The code is not general enough; that is, if in the future the code needs to handle a new case, you will have to update all the code with the switch statements
c) The code that contains the switch statement is cluttered with details of handling the different cases

In C++ you can replace this kind of switch statements with virtual function calls. This produces code that is more elegant, shorter, and more importantly, it can handle future cases without any changes (old code that calls new code… what a concept!)

Example
Instead of:

```
void HandleStaticForce();
void HandleDynamicForce()
void HandleTangnentialForce();
```

```
// main code that handles different types of forces
// Note that it needs tochange if a new type of force is defined
switch (forceType) {
case STATIC_FORCE:
        HandleStaticForce();
        break;
case DYNAMIC_FORCE:
        HandleDynamicForce();
        break;
case TANGENTIAL_FORCE:
        HandleTangentialForce();
        break;
default:
        ERROR("Unknown force type");
        break;
}
```

do:

```
class Force {
        …
        virtual HandleForce() = 0;
};
class StaticForce : public Force {
        …
        virtual HandleForce();
};
class DynamicForce : public Force {
        …
        virtual HandleForce();
};
class TangentialForce : public Force {
        …
        virtual HandleForce();
};

//main code that handles different types of forces
//Note that it won't change if a new type of force is defined!
Force* force;
Force→HandleForce();
```

6.8 Avoid using multiple inheritance

Multiple inheritance in most cases is discouraged. It is a topic not well understood by many programmers, and it can lead to problems and considerable code complexity. The only case that multiple inheritance could be used without creating potential problems is when a class inherits from pure interfaces only (i.e. the parent classes do not contain any data or code).

6.9 Use classes instead of structures

The only difference between structures and classes in C++ is that structure data members are public by default, whereas class data members are private by default. It seems that the only real reason structures have remained in C++ is for backwards compatibility with C. Therefore the use of structures is

discouraged, as it often leads programmers to declare public data members, which should be avoided (see 6.4 “Avoid having public data members”). The use of structures should be limited to the rare case that you need to group some data together, without intending to hide them or provide any functionality with them.

6.10 Prefer C++-style comments

C++ comments are easier to type: it’s one character repeated twice (in contrast, C comments require 4 characters plus the Shift key). Also C++ comments can enhance code readability. Here’s how: the following lines have a C-style comment:

```
Foo();
Bar(); /* blah */
FooTech();
```

To comment out this block, we’d have to use the preprocessor (because if you try to comment out the whole block with /* */, you’d get compilation error):

```
#if 0
        Foo();
        Bar(); /* blah, blah, blah */
        FooTech();
#endif
```

However, if the comment was C++-style, then the whole block can be commented out with /* */:

```
/*
        Foo();
        Bar(); // blah, blah, blah
        FooTech();
*/
```

So what? Well, it turns out that syntax-coloring cannot recognize that a block of code is being commented out by the preprocessor. Did you ever happen to struggle to analyze a piece of code only to suddenly realize that it was actually commented out by the preprocessor? Also tools like etags sometimes get confused with regions of code commented out through the preprocessor. Using C++ comments helps reduce the need for #if 0.

7. WRITING CORRECT CODE

7.1 Don't shadow variable names

Do not use the same name for two different variables in enclosing scopes. The second variable will "shadow" the first one, and this can lead into a hard to detect bug.

Example

```
int length;
….
if (a>0) {
        int length; //legal, but BAD! It hides the previously declared variable "length"
        ….
}
```

7.2 Don't "delete this"

Deleting the current object from within one of its own methods should be avoided. Although legal, this construct can easily lead to dangling pointers and crashes.

7.3 Avoid downcasts

Downcasting (casting a pointer of a base class to a pointer of a derived class) is the "goto" of object-oriented programming. Avoid it as much as possible. It can lead to mysterious crashes and it also makes you lose the advantages of C++ as it disables dynamic binding. If you find yourself in the need to use it, then you probably need to rethink your design. There are only very few cases where we cannot really avoid it; in those cases, use dynamic_cast.

7.4 Use the correct form of delete

When deleting an array, use the brackets [ ]. When deleting something that is not array, don't use [ ]. This is a common mistake, and the result is undefined.

Example

```
Region* region = new Region;
delete [ ] region; // WRONG
delete region; // OK

Region* regionArray = new Region[100];
delete regionArray; // WRONG
delete [ ] regionArray; // OK
```

7.5 Make all destructors virtual

Make all destructors virtual. This ensures that if this class ever has children, then all objects will be destructed properly. According to the C++ standard, when you try to delete a derived class object through a base class pointer and the base class has a non-virtual destructor, the results are undefined.

There is only a rare exception to this rule: when you have small classes that need to be as efficient as possible, you know that their destructor will be called very often, and you are certain that they will never have children. In these cases, having a non-virtual destructor can make it a little more efficient. However, in the overwhelming majority of cases it is far better to avoid serious potential bugs by always declaring destructors as virtual, rather than worrying about small efficiency improvements.

7.6 Don't use base class pointers to access arrays of derived objects

The reason is simple: pointer arithmetic in this case will be based on the base class size, however, the real array will be spaced according to the derived class size, which most likely is larger (derived classes usually contain more data members than base classes). Therefore simple expressions like a[n] will fail.

Example

```
class Derived: public Base{
        ....
        int x; // an additional int data member
};
Base* array = new Derived[10];
cout << array[2];  // this means (array + 2*sizeof(Base)), which in this case is wrong!

Derived derivedArray[20];
void PrintArray(Base array[]); // function that prints all elements of the given array
PrintArray(derivedArray); // legal syntax, but incorrect result!
```

Note, however, that mixing arrays of base class *pointers* with derived objects is perfectly fine:

```
Base* array[10];
for(int i=0; i<10; i++) {
        array[i] = new Derived;
}
cout << *array[2]; // this is fine, pointer arithmetic is based on size of pointers only
```

7.7 Protect classes from shallow copy – the "Law of Big Three"

Most classes should declare (but not necessarily define) the following methods: default constructor, copy constructor, assignment operator, and destructor. The reason is to protect against the compiler providing default definitions. You don't have to define these methods unless these are used in the code, in which case it's better to have your own methods instead of letting the compiler provide default definitions that do a sometimes dangerous bit-wise copy of your objects.

Also keep in mind the "Law of Big Three": if a class needs a destructor, or a copy constructor, or an assignment operator, then it needs them all. This simply means that every time you define a destructor, then you should also define the copy constructor and assignment operator. If you want to disallow copying the objects of this class, then make the copy constructor and the assignment operator private, and put an error message in their body.

The reason for this guideline is that when no assignment operator or copy constructor are present, then every time the assignment operator is used or when objects of the class are passed by value, then the objects are copied bit-by-bit (shallow copy). If a class needs a destructor, this means that it probably contains pointers to resources (e.g. memory) for which bit-wise copy would probably cause problems, as

multiple objects would point to the same resources. In these cases it is better to explicitly define what happens when one object is copied to another, to avoid potential bugs.

```
// Every class should have at least the following:
class X {
public:
        X (); // default constructor
        virtual ~X (); // destructor
protected:
private:
        X (const X& x); // copy constructor – make it public only if needed
        X& operator = (const X& x); // operator equal – make it public only if needed
};
```

7.8 Correct implementation of operator=

A common mistake when implementing the assignment operator is not checking for self-assignment. This could lead to a hard to detect bug. Another common mistake is forgetting to take care of the base class. In that case, the assignment would be incomplete.

To implement correctly operator=, use the following skeleton:

```
//correct implementation of operator= for class Child that inherits from class Base
Child& Child::operator=(const Child & other)
{
        if (this == &other) return *this;   // avoid self-assignment
        Base::operator=(other);    // take care of the base class

        // … assign to members of Child now …

        return *this;
}
```

7.9 Correct implementation of copy constructor

A common mistake when implementing the copy constructor is to forget the base class. We provide here the skeleton to use to avoid that mistake:

```
// correct copy constructor for class Child that inherits from class Base
Child::Child (const Child & other)
    : Base (other)     //calling the Base copy constructor
{
        // … assign to members of Child now …
};
```

8. WRITING RELIABLE CODE

8.1 Put always a "default" in switch statements

If the code is not supposed to ever execute the default statement, just use the ERROR macro in the default case which throws an error exception.

8.2 Memory ownership

If a function allocates memory and then passes the ownership of that memory to the caller, start its name with "create…" or "copy…" to emphasize the fact that new memory has been created and the caller must explicitly delete it. Also put a warning comment in the function header.

8.3 Be const correct

Use the keyword "const" for formal function parameters when the function is not supposed to change them. Use also const after function declarations when the functions are not supposed to change the object they belong to.

Note: put const from the beginning when you first start writing the code. Adding const later can be expensive, since it usually causes a ripple effect of changes through the system. Also remember that when using const, the compiler is your friend: it shows you all the places where you need to put const.

Example

```
long GetSize() const;
void CopyRegion(const Region& other, const CopyScheme* scheme);
```

8.4 Use new and delete instead of malloc and free

You should always prefer the use of new and delete. The problem with malloc and free (and their variants) is simple: they don't know about constructors and destructors. You will only need to use free for memory that is allocated by calls to old C libraries.

Also don't mix new with free, and malloc with delete: the results are undefined. Memory allocated with malloc should be deleted with free, and memory allocated with new should be deleted with delete.

8.5 Error handling with TRAP and ERROR

The most common ways that C/C++ programmers use for trapping and signaling errors are the following:

a) Returning appropriate error values from functions.
b) Using the macro ASSERT.
c) Throwing exceptions.

Yet another way is to define our own mechanism that can easily switch between (b) and (c) or any other error handling policy. In order to do that we can define and use a new macro, TRAP, that takes only the essential parameters: an error condition, and an appropriate error message. The simplest way to define this macro is the following:

```
#ifdef  _DEBUG
        #define TRAP(cond, msg)      ASSERT(!(cond));
#else
        #define TRAP(cond, msg)     { if(cond) throw msg; }
#endif
```

Under this definition, in debug mode the TRAP macro turns into an assertion, which conveniently stops and points to the debugger the appropriate code line when an error occurs. On the other hand, in release mode it turns into an exception-throwing mechanism.

The following more sophisticated definition of TRAP allows for more flexibility in the construction of the message string:

```
#include <iostream>
#include <strstream>
using namespace std;

#undef TRAP
#define TRAP(cond, msg) \
{  \
   if(cond) { \
      char __TRAP_msg[1000]; \
      ostrstream __TRAP_stream(__TRAP_msg, sizeof(__TRAP_msg));  \
      __TRAP_stream << msg << ends;  \
      throw(__TRAP_msg);  \
    \
}
```

Under this definition it is possible to create the message in a way similar to the cout convention, for example:

```
TRAP(count > MAX,   "Count cannot exceed: " << MAX << "but it is: " << count);
TRAP(speed < 0 || speed > 100,   "Illegal speed value: " << speed);
```

The accompanying macro ERROR is defined for convenience to be used when there is no explicit error condition, but we need to signal an error, e.g. in a "switch" statement, or in an empty bodied function that is not supposed to be called. The ERROR macro is defined as follows:

```
#define  ERROR(msg)    TRAP(true, msg)
```

The macros TRAP and ERROR should be used liberally throughout the code to catch abnormal, "impossible", or error conditions. At the minimum, they should be used to check the input parameters in every function to make sure they have legal or reasonable values, and also to check the results of every complex piece of code.

8.6 Use TRAP instead of returning error values

Using the TRAP mechanism to check and handle error conditions is better than returning error values because of the following reasons:

a) Error values work only if the programmers religiously check for them, and very often programmer simply ignore them; whereas an activated TRAP cannot be ignored, it must be caught by a "catch" statement otherwise it will terminate the program.

b) Sometimes we simply cannot indicate errors by returning error values. For example, consider a function that calculates and returns the average:

```
double CalcAverage(double* values);
```

Any returned value could potentially be valid; if the programmer wants to use return values for indicating errors, then he would be forced to change the function to return the average in a parameter:

```
int CalcAverage(double* average, double* values);
```

However, that would restrict the function from being used directly in an expression like the following:

```
result = offset + CalcAverage(valArray1) / CalcAverage(valArray2);
```

c) Checking for the return value of every function call in a statement block can be cumbersome. Exceptions nicely put all error-checking and handling code at the end of the block.

d) If the error-handling code is not the same code that called the function that returned the error, then we probably need to propagate back the error value through, potentially, many function calls. However, exceptions handle this automatically; the exception is propagated back until someone handles it.

8.7 Prefer using TRAP than ASSERT

The ASSERT macro only verifies if a certain condition is TRUE. If the programmer wants to clarify why, then he has to put an additional comment. More often than not, no such comments exist. Also when an ASSERT fails, the user receives a cryptic message "Assertion failed" with no much additional information about the reasons of the error. On the other hand, the TRAP macro includes an error condition *and* an error message. This has the following benefits:

a) it forces programmers to write (in the error message) why the trapped condition is an error
b) the error message itself becomes useful documentation when maintaining the code
c) the error message can help the users understand why the error occurred and use a workaround

Note that TRAP is highly flexible, and the error message can be constructed using the << operator:

Example

```
TRAP(speed >=MAX, "Unreasonable speed=" << speed << ". It must be less than " << MAX);
```

8.8 Why leaving TRAPs in the released code improves reliability and robustness

Using and enabling TRAPs in the released code increases the reliability and robustness of the application.

The application becomes more reliable because errors encountered will not remain unnoticed. Therefore when the application produces a result without any errors reported, we *know* that no error conditions checked by TRAPs were encountered. In contrast, if TRAPs were deactivated in the release code, then we simply don't know if any errors happened that could have been TRAPped. Therefore our confidence on the correctness of the results is smaller, and the application is less reliable.

The application becomes more robust because the programmers can actually respond and handle error conditions. An error condition, no matter how small it is, can either directly or indirectly crash the system. If we disable TRAPs, then we let errors untreated with all the bad consequences. However, by TRAPping errors we give the opportunity to programmers to write code (in a "catch" statement) that can respond gracefully to such errors. The respond can range anywhere from completely correcting the error and resuming normal execution, to simply printing a message and gracefully terminating the application. Both cases are better than a core dump.

8.9 Don't use exception handling as a control-flow mechanism

Exception handling is a powerful mechanism that can easily transfer control between parts of a system bypassing the normal calling control sequence. For example, code that processes list elements could throw an exception when the end of the list is found to transfer control conveniently to another part of the program thus avoiding the regular return path through many layers of function calls. You should avoid doing this! Exceptions should be thrown only when an error condition occurs, something that is not part of a regular program execution. To emphasize this, we use the term "error-handling" instead of "exception-handling".

The reason for this rule is that writing exception-safe programs is quite difficult; auto_ptr has to be used instead of regular pointers, care should be taken to avoid propagating exceptions from destructors, etc. Otherwise, resource leaks may occur, and the system will not be exception-safe (it won't be able to handle exceptions routinely and continue functioning with no problems). In most projects, our goal is not to create such a system. Instead, our goal is to create a system that exceptions occur rarely, are mostly caused by bugs in the program or lack of system resources (e.g. not enough memory) and are handled as serious situations that require special action (which may include notifying the user).

8.10 Avoid exceptions in destructors

Do not throw exceptions in destructors, and do not call anything that might throw an exception unless you're prepared to catch it and deal with it. Destructors that exit with an exception never finish executing, and the uncleaned resources are lost. Additionally, the ISO/ANSI draft standard states that if the exit occurs during a stack unwind (while handling an exception), and while searching for an exception handler, the special function terminate() is called.

The best way to make sure that no exceptions escape from destructors is to enclose the code of every "complex" destructor in a try-catch statement that catches all exceptions:

```
~FooClass( ) {
        try {
                // put the destructor code here
        }
        catch(…) {  // this catches all exceptions
        }
}
```

9. CODE PERFORMANCE ISSUES

9.1 Avoid passing arguments by value

It is fine to pass built-in types and value-oriented classes by value. However, many classes, particularly those that perform heap allocation or those that are large in size, are inefficient to pass by value. Use instead pass by const reference.

Example

Instead of:

```
void Foo(long x, SomeClass obj)  // NOT efficient
{
        obj.CleanAll();
        obj.SetValue(x);
}
```

do:

```
void Foo(long x, const SomeClass& obj)  // better
{
        obj.CleanAll();
        obj.SetValue(x);
}
```

9.2 Beware of the order in constructor initialization lists

If you use initialization list in a constructor, then order the data members in this list in the same order that they are declared in the class. The reason is that C++ initializes the variables in this list in the order in which they are declared in the class, regardless of their order in the list. This may lead to a subtle bug if the variables actually must be initialized in a specific order.

Example

```
class SomeClass {
public:
        SomeClass(int var1, int var2, int var3)
                : m_var1(var1), m_var2(var2), m_var3(var3) {}
                // here the initialization order will be: a2, a1, a3
private:
        int m_var2;
        int m_var1;
        int m_var3;
}
```

9.3 Don't micro-tune your code – let the compiler do it

Avoid doing "performance tuning" that the compilers usually are pretty good at. For example, most compilers are pretty good at loop unrolling, so you shouldn't reduce readability of your code by doing such tricks yourself.

9.4 Don't sacrifice code quality to improve performance

Don't try to increase code efficiency by sacrificing its clarity and readability. On the long term, this will create more problems than it solves. Code with poor readability is difficult to maintain; this means more

bugs and more time needed for code maintenance. This in turn results in delaying new releases and neglecting performance tuning as the programmers are too busy fixing bugs and trying to put in new features in code that is difficult to understand.

Performance should be addressed separately and it should be more focused on the "hot spots" of the code. Don't forget the 80-20 rule: 80% of the execution time is usually spent on only 20% of the code. Randomly applying tricks to improve performance is bad; instead, after the code is written and debugged, do a systematic profile analysis to reveal the bottlenecks and focus only on the parts that really matter.

9.5 Don't tweak your code – try better algorithms

Better algorithms or smarter data structures generally buy you a lot more performance than tweaking code. Changing your code to use a O(logn) algorithm will generally pay much more than spending time trying to improve and fine tune an algorithm that is inherently O(n).

Smarter data structures can also have a significant impact on the performance of your program (e.g. classes that use lazy evaluation). Try to address the system performance in a higher level; don't simply tweak code.

10. APPENDIX: RULE SUMMARY

The following checklists summarize and group the rules and guidelines given in this document. For the reader's convenience, every item has a reference to the associated section where it is discussed in detail.

10.1 Syntax conventions

- **Keep names consistent [2.9]**
- **Use pronounceable, non-cryptic names (e.g. *firstLetter* instead of *fLtr*) [2.9]**
- **Avoid extremely long variable and function names [2.11]**
- **Try to use no more than 3 words in class names [2.3]**
- **Function names should be a verb or a verb followed by noun (e.g. *StoreContext()*) [2.6]**
- **Use the prefixes "get" and "set" for functions that access class attributes [6.5]**
- **Use 3 or 4 spaces for indentation [5.5]**
- **Put starting curly braces at the end of the codelines (except for function braces) [5.6]**
- **Typography [2.1 – 2.8]:**

```
class  SomeClass; //class names in mixed case, start with uppercase

struct  SomeStruct; //structure names in mixed case, start with uppercase

class  FooClass {
     int  m_memberVariable;   //class member variables start with "m_"
};

struct  FooStruct{
     int  memberVariable;   //struct member variables don't start with "m_"
};

int  localVariable;  //variable names in mixed case, starting with lowercase

int  gGlobalVariable;  //global variables have the prefix "g"

void  SomeFunction();  //function names in mixed case, starting with uppercase

typedef  long LocalTime;  //typedefs in mixex case, start with uppercase

#define  SOMETHING;  //everything #defined in uppercase – but avoid #defines!

const  int  CONSTANT = 100;  //constants are all uppercase

enum Colors { REG, GREEN, BLUE };  //enum values are all uppercase
```

10.2 Good coding practices for C and C++

- **Avoid writing long functions (more than 2 pages of code) [5.1]**
- **Pass objects instead of passing too many parameters in functions [5.3]**
- **Avoid deeply nested code (more than 5 nesting levels) [5.4]**
- **Put always a "default" in switch statements [8.1]**
- **Minimize the number of #includes in your header files [4.4]**
- **Protect against multiple header file inclusion [4.2]**
- **Don't duplicate code [6.1]**
- **Don't use goto [5.7]**
- **Don't omit dummy parameter names in function declarations [5.2]**
- **Use sparingly "break" and "continue" in loops [5.8]**
- **Avoid returning from the middle of functions [5.9]**
- **Don't shadow variable names [7.1]**
- **Don't sacrifice code quality (readability, maintainability) to improve performance [9.4]**
- **Don't micro-tune your code – let the compiler do it [9.3]**
- **Don't tweak your code to improve performance – try better algorithms [9.5]**

10.3 C++ usage essentials

- **Enclose constants and enums in classes – don't leave them in global scope [5.10, 6.2]**
- **Use as much as possible forward class declarations instead of #include [4.3]**
- **Avoid having public data members in classes [6.4]**
- **Don't unnecessarily expose class private data members through "get" and "set" [6.6]**
- **Use classes instead of structures [6.9]**
- **Don't write switch statement based on the type of an object – use inheritance [6.7]**
- **Avoid using multiple inheritance [6.8]**
- **Use *const* as much as possible on parameters, functions and local variables [5.12, 8.3]**
- **Use *new* and *delete* instead of *malloc* and *free* [8.4]**
- **If you define a destructor, then also define operator= and copy constructor [7.7]**
- **Follow the examples when implementing operator= or copy constructor [7.8 – 7.9]**
- **Do not "delete this" [7.1]**
- **Use "delete [ ]" for arrays and "delete" for non-arrays [7.4]**
- **Avoid downcasts [7.3]**
- **Make all destructors virtual [7.5]**
- **Don't use base class pointers to access arrays of derived objects [7.6]**
- **Beware of the element order in constructor initialization lists [9.2]**

10.4 Good C++ coding style

- **Consider putting every class in its own file [4.1]**
- **Pass arguments by reference instead of by value [9.1]**
- **Don't use #define (use const, enum & inline functions instead) [6.3]**
- **In class definitions, put sections in the following order: public, protected, private [4.5]**
- **Put *after* the class definition inline functions that don't fit in 1 line [4.5]**
- **Use the "virtual" keyword both in base and derived classes [5.13]**
- **Declare and initialize variables where they are first used [5.11]**

10.5 Error handling

- **Do error handling with TRAP, ERROR and try-catch statements [8.5]**
- **Use TRAP instead of returning error values [8.6]**
- **Prefer using TRAP than ASSERT [8.7]**
- **Leave TRAP enabled in the released code [8.8]**
- **Don't use error handling as a control-flow mechanism [8.9]**
- **Avoid TRAP, ASSERT, or anything that throws exceptions in destructors [8.9]**
- **Use TRAP in functions to check preconditions – postconditions [3.10]**

10.6 Class and function header comments

- **Use function header comments to describe the function usage to a client [3.9]**
- **Don't put implementation details in function header comments [3.9]**
- **Describe parameters and return values in function header comments [3.9]**
- **Keep function header comments short (no more than 3-5 lines) [3.9]**
- **If a function passes memory ownership to the caller, put a clear comment [8.2]**
- **Use class header comments to describe the class usage to a client [4.2]**

10.7 Code comments

- **Use "TODO" comments for work that needs to be done and keep them current [3.8]**
- **Prefer C++-style // comments [6.10]**
- **Comments should be brief [3.2]**
- **Put comments as close to the code they refer as possible - blend them with the code [3.7]**
- **Don't write useless comments that simply restate the code [3.3]**
- **Comments should explain "why" instead of "how", except when commenting in the large [3.4]**
- **Don't overdo with comments – about 30% comments-to-code ratio should be enough [3.11]**
- **Use comments in variable declarations to clarify their use [3.6]**
- **Comments should precede the code they refer to (except in variable declarations) [3.5]**
- **Comments should be indented to the same level as the code they refer to [3.5]**