Software Development and Coding Standards

Falko Kuester and David Wiley

Visualization and Interactive Systems Group
Department of Electrical and Computer Engineering
University of California, Irvine
537 Engineering Tower
Irvine, California 92697-2625

Visualization and Graphics Research Group
Center for Image Processing and Integrated Computing
Department of Computer Science
University of California, Davis, CA 95616-8562

December 10, 2001

1 Preface

Every programmer inherently learns and practices a custom programming style. The reason for this is no doubt rooted in how programmers learn to program: a snippet from this book, a line from that, an algorithm from this magazine, an array class from that. Every programmer is essentially a melting pot for the many different styles that exist. It is left to the statistically inclined reader to determine just how many combinations are possible and at which frequency. Having a custom style is generally suitable as long as the programmer abstains from interacting with other programmers and decides to be a prisoner to that particular style. Aside from the usual social discontinuities, problems surface when programmers begin to mingle. A random sample of C++ source code from the Internet will yield a variety of C++ dialects. Either you will learn some new things or your eyes will tire from poorly written code. The one constant is that you will never find two programmers that do things exactly the same way. Even more problems occur when teams of programmers must work together. In this environment source code can make round trips through programmers, changing ever so slightly in each iteration. Small scale battles can occur in these code bytes in the form of moving curly braces and parenthesis around, adding or removing spaces, tabbing this, carriage-returning that, commenting this, not commenting at all, renaming variables, or using for loops instead of while loops. We end up fighting essentially irrelevant battles and wasting time.

If everybody wrote code in the exact same way, then everything would be fine. This is obviously not the case and it is not the intention of this document to force you into writing code the way we write code. The purpose of this document is to address common problems and to provide solutions in a well-defined manner.

1.1 Where and who is Waldo?

You might remember the childrens book that challenged you to find Waldo somewhere hidden in an information overloaded picture. Software development and the maintenance of legacy code adds another twist to this challenge. What can I do if I do not find him? Well, you seriously hope that he left as many clues as possible in the form of documentation, in the code that is. Programmer Waldo, for the purpose of this document, is the manifestation of all the other programmers in the world including the next generation of programmers. Waldo also includes yourself sometime in the future because quite often it is difficult to reuse code that you wrote in the past. Had the Y2K programmers considered Waldo in their development they would not have been so infamous.

1.2 Resources

This document combines styles found in various documents including SUNs JavaDoc[1], the Microsoft Development Network (MSDN) library[2] standards and many more. The rest is based on the authors experience from a decade of C/C++ programming and commercial software development. By no means is this document considered to be the ultimate authority on C++ programming. Nevertheless, it reflects helpful concepts and ideas for good software development strategies.

1.3 Contact Information

We always enjoy debating programming issues and styles and welcome them. We find this is a very good way to learn new things about C++ and software development. We welcome feedback of (almost) any type about this document, C++ or software development issues. If you have any comments on or additions to anything in this document please let us know and we will address them as best as we can:

fkuester@ece.uci.edu
wiley@cs.ucdavis.edu

[1] http://java.sun.com/j2se/javadoc/writingdoccomments/
[2] Microsoft Foundation Library Development Guidelines

Contents

List of Figures

List of Tables

2 Introduction

This document outlines the software development and coding standards at the Visualization and Interactive Systems Group (VIS).

2.1 What are Software Development and Coding Standards and why do we need them ?

A standard is a description of precise behaviors or actions that form a methodology. As applied to a coding standard, it will describe the form and style of code. Some of the reasons to adopt a standard include.

Productivity: Includes the time for coding and maintenance. The goal of our development effort has to be to develop standard algorithms and classes once and then to reuse them.

Quality: An implication of code conformance to the design specification if one writes code in a uniform style. Ideally, the code will hold no surprises.

Maintainability: The code you write today is the legacy code of tomorrow. The maintenance effort is the longest and most tedious phase during the life-cycle of developed software. Particularly academic environments with their extremely short “lifespan” of student researchers face the challenges of dealing with legacy code if they not chose to discard it in first place.

Understandability: The original writer of the code will usually not be the same person that must maintain it.

2.2 Scope

The scope of this document is to describe a consistent approach to software development. A style that is consistent is more important than a style that enforces absolute rules. However, it is in the nature of styles that not everyone will be entirely satisfied and that it might cause initial startup problems. Nevertheless, it is expected that everyone acknowledges the importance of a standard and its role in achieving a consistent software development process.

2.3 Document layout and conventions

Requirements or rules are expressed through the verb “shall” whereas recommendations and suggestions use the verb “should”.

Type	Formatting
Source Code	
File Name	
Rule	
Exception to a rule	
Recommendation	

Table 1: Type setting

3 Directory Structure

A new dedicated VIS account forms the root of the software repository and development tree. All active VIS projects shall implement this structure.

The VIS Development Tree			
vis/			VIS source code repository and main development directory
	bin/		Binaries of VIS specific applications and standard demos
	build/		Templates for makefiles, scripts and other common tools
	cvsroot/		Root directory of the CVS source code repository
	demos/		Working demos for presentation purposes
	doc/		Documentation
	include/		Include files for VIS libraries
	lib/		VIS libraries
	log/		Log files documenting the latest software builds
	papers/		Collection of published papers
	presentations/		Collection of presentations and presentation templates
	projects/		Projects directory tree
	templates/		Standard templates for source and header files
	www/		Internal web documents
projects/			Projects directory tree
	name/		Project name
		bin/	Project binaries
		src/	Project source code directory
		obj/	Project object directory
		doc/	Project specific documentation
		lib/	Project libraries
		include/	Project include files
		qa/	Project test code and scripts
templates/			Standard VIS style templates
	template.cxx		Template for C++ source files
	template.h		Template for C++ header files
	project.html		Template for project descriptions
doc/			This document
build/			Templates for makefiles, scripts and tools
	Commondefs		Common definitions for make
	Makefile		Generic VIS makefile

Table 2: VIS development tree

All newly created projects have to provide the projects directory structure. Otherwise the build script will not work properly. Furthermore, each project must have a project.html file in its doc directory. These files will be used to automatically update the HTML page describing the project purpose and implementation state. File access permissions will be used to determine which files should be accessible publicly through the VIS web-page.

4 General Naming Conventions

Historically there exist quite a few strategies for the creation of names for variables, classes, structs, methods, etc. When Waldo reads your code however, he should be able to figure out what the names and variables are supposed to represent without having to decifer your name encryption algorithm.

4.1 File Extensions

The following rules shall be observed when determining file extension.

File type	File extension
C source code	.c
C++ source code	.cxx
C/C++ header files	.h
Perl	.pl
Python	.py
Makefiles	Makefile
Common definitions	Commondefs
Static libraries	.a
Dynamic libraries	.so
LaTeX	.tex
BibTeX	.bib
Acrobat PDF	.pdf
Adobe Postscript	.ps

Table 3: File extensions

4.2 File Names

In particular for files, expressive and natural names are easier to remember than names that contain odd abbreviations or alterations. See Table 4 for detailed examples.

- It is easier to remember complete words rather than some awkward abbreviation of a word such as Window: Wnd, Wn, Wind, etc.
- Use mixed case rather than underscores to separate words in names. Always capitalize the first letter.
- Do not invent plurals for collections of objects. Use the proper plural, this way it is more easily remembered.
- For C++ code, the file name shall be identical to the class name.

Use	Do not use
CVector	vec
CVectorIterator	vecitr
CListIterator	List_Iterator, Listiterator, listiterator, LIST_ITERATOR, LISTITERATOR
CBigVector	bigVector, Big_Vector, bigvector
CAxes	Axiss
CWindows	WindowCollection

Table 4: File naming conventions

4.3 Class Names

The class naming conventions are identical to the file naming conventions. See Table 5 for examples.

- It is easier to remember complete words rather than some awkward abbreviation of a word such as Window: Wnd, Wn, Wind, etc.
- Use mixed case rather than underscores to separate words in names. Always capitalize the first letter.
- Do not invent plurals for collections of objects. Use the proper plural, this way it is more easily remembered.
- For C++ code the classname shall be identical to the file name.
- Begin all class and struct names with C or a unique identifier.

4.4 Begin all class and struct names with C or a unique project specific identifier

The C denotes that the data type being used is a class or struct and helps distinguish between data types and variables. Alternatively you can use two or three character identifiers such as Ve (VirtualExplorer), Ldw (Large Data Visualization) to indicate to which project these files belong.

Listing 1: CVector

```
class CVector                       // or VeVector, LdwVector
{
  . . .
};

void foo()
{
  CVector v;                        // Obvious that CVector is a class based data type
  CVector vector, Vector, vct;      // Can also use a variety of combinations on the
                                    // word ``vector'' without causing a name conflict
}
```

Compared to:

Listing 2: vector

```
class vector
{
      . . .
}

void foo()
{
   vector v;          // Can not tell what 'vector' is without looking at
                      // the definition (is it a struct, enum, class,
                      // macro, etc.)
   vector vector;     // Error, name conflict
   vector Vector;     // Odd looking (to me at least)
}
```

The major advantage modifying class names with a prefix such as C is that you can then use the class name without the prefix as a variable name. How often have you named a class one thing, say line, then when you create some objects you end up mangling line to be lin, aline, line1, Line, lne, etc. If you prefix your class name you can name your variables naturally, in the case of CLine: Line, LineA, LineB, etc.

4.5 Variables

Variables should be expressive and easy to remember.

- It is easier to remember complete words rather than some awkward abbreviation of a word such as Window: Wnd, Wn, Wind, etc.
- Use mixed case rather than underscores to separate words in names. Always begin with a lower case character.
- Do not invent plurals for collections of objects. Use the proper plural, this way it is more easily remembered.
- Prefix m_ onto your member variable names.

Use	Do not use
VeVector	vec
VeVectorIterator	vecitr
VeListIterator	List_Iterator, Listiterator, listiterator
VeBigVector	bigVector, Big_Vector, bigvector
VeAxes	Axiss
VeWindows	WindowCollection

Table 5: Class naming conventions

4.6 Use Simplified Hungarian Notation for variable names

To make your life easier we suggest the use of the simplified Hungarian notation for variable naming. Consider the problem of dealing with a thousand-line function that you wrote last year. That functions should never be that long in first place is besides the point. Assume you are staring at a variable name wondering what the type of the variable is. The variable happens to be declared on the first line of the function and you are viewing the file through a very slow telnet connection. If you have lots of time to waste you might just decide to scroll and wait. The alternate solution would have been to use Hungarian notation for variable naming. An Hungarian Microsoft programmer developed the notation (hence the name). The idea is to prefix variable names with the type of the variable. Microsoft has simplified Hungarian Notation to become Simplified Hungarian Notation. For our purposes all Microsoft specific content has been removed. The recommended variable naming conventions are listed in Table 6. The simplified version was derived since the true Hungarian Notation is too cumbersome. It accounts for less than 32-bit operating systems wherein you are required to specify things like whether a pointer is far or near. In addition it also accounts for signing and bit length of types.

Prefix	Type	Description	Example
n	int	any integer type	nCount
ch	char	any character type	chLetter
f	float, double	floating point	fPercent
b	bool	any boolean type	bDone
l	long	any long type	lDistance
p	*	any pointer	pObject, pnCount
sz	*	nul terminated string of characters	szText
pfn	*	function pointer	pfnProgress
h	handle	handle to something	hMenu

Table 6: Simplified Hungarian Notation.

Variable names can then be constructed from combinations of the prefix list given in Table 6. The programmer should use common sense to determine which combinations make sense. Additional examples are listed in Table 7.

Type	Example
int	nCount
int*	pnCount
bool*	pbDone
bool&	bDone

Table 7: Example of type specific variable naming.

A paradox arises while using Hungarian Notation of any variation. What do you do with user defined types? You do not want to invent your own prefix because it may not make sense to Waldo, but by not prefixing a variable with something you are not helping the situation either. The best solution is to name your variables in such a way that you do not need to use Hungarian Notation to decipher its type

4.6.1 Class Member Variables

Use m_ for all class member variables. This will allow you to select local variables in your class member methods with the same name as the class member variable (without the m_). This allows for improved local and member variable relationships and clear differentiation between local and member variables

Listing 3: Demonstration of m_ need

```
class CVector
{
public:
  CVector(int x, int y)
  {
     x = x;  // An error frequently seen
     y = y;  // in beginning programming courses
  }
private:
    int x, y;
};
```

Listing 4: Use of m_

```
class CVector
{
```

```
public:
  CVector(int x, int y)
  {
     m_x = x;  // Obvious which parameter gets assigned to
     m_y = y;  // which member variable
  }
private:
    int m_x, m_y;
};
```

By using m_ you reduce confusion when writing code because the variable names are essentially the same, while at the same time you can easily distinguish between them. Additional prefix conventions are listed in Table 8.

Type	Prefix
class member variable	m_
global variable	g_
static variable	s_

Table 8: Variable prefix conventions.

4.7 Do not use _ to prefix any identifier names

The ANSI C specification allows identifiers that begin with two underscores or single underscore followed by a capital letter to be reserved for compiler use. To make things simpler for Waldo when he tries to compile your code on the quantum computer of the future, do not begin any identifier with an underscore.

5 File Structure

The VIS source code repository provides template files for C/C++. Use these files as base for all new code development. See section 5.2 and 5.3 for details.

5.1 Header File

In the day and age of hundreds and thousands of files in a project, quite often it becomes very difficult to remember who includes what header files. Without taking any precautions you will get many “already defined” errors. Prevent these errors by:

- Using preprocessor wrappers for header files.
- Only defining one class per header.

Listing 5: Header file basics

```
#ifndef IDAV_CLASSNAME_H_   // The prefix is derived from the class name in this file
#define IDAV_CLASSNAME_H_   // Prefix the identifier with an IDAV_
.
.
.
class CClassName
{
    .
    .
    .
};

#endif   // IDAV_CLASSNAME_H_
```

This sequence of preprocessor commands allows this header file to be included more than once in any particular implementation or header file without generating “already defined” errors.
You should also try to have only one class per header and implementation file. This greatly simplifies the compilation process as well as partitioning your source code into logical divisions. For extremely large class implementations (over several thousand lines) it would be reasonable to divide the implementation into more than one file.

5.2 Header File Template

```
/*+-------------------------------------------------------------------------+
  | @BEGIN_COPYRIGHT                                                        |
  |                                                                         |
  |                      Copyright (C) 1998, 1999, 2000, 2001               |
  |                                                                         |
  |             Center for Image Processing and Integrated Computing        |
  |                          University of California                       |
  |                             One Shields Avenue                          |
  |                              Davis, CA 95616                            |
  |                                                                         |
  |  Unpublished work-rights reserved under the U.S. Copyright Act. Use,    |
  |  dublication, or disclosure is subject to restrictions.                 |
  |                                                                         |
  | @END_COPYRIGHT                                                          |
  +-------------------------------------------------------------------------+
  |                                                                         |
  | Component  : CTemplate                        Author: author name       |
  | Filename   : CTemplate.h                      Date  : 01-01-2000        |
  | Sourcefile : CTemplate.cxx                                              |
  |                                                                         |
  +---------------------------- Revision Information-------------------------+
  | $Author: fkuester $
  | $Revision: 1.1.1.1 $
  | $Date: 2000/03/31 22:36:34 $
  | $RCSfile: CTemplate.h,v $
  +-------------------------------------------------------------------------+
*/

//---------------------------------- Defines ----------------------------------
#ifndef IDAV_CTEMPLATE_H_
#define IDAV_CTEMPLATE_H_
//-----------------------------------------------------------------------------

//---------------------------------- Includes ---------------------------------
#include <CObject.h>
//-----------------------------------------------------------------------------

/**
 **  CIPIC template for C++ header files.
 **  This is the default template to be used for all projects and development
 **  code at the Center for Image Processing and Integrated Computing (CIPIC).
 **  Just copy the template and perform a "regular-expressing-replace"
 **  on \a CTemplate. I.e. exchange \a CTemplate with the name of your class.
 **  Please replace this entire section with a brief description of the
 **  implemented class.
 **
 **  @author  author name
 **  @version $Revision: 1.1.1.1 $
 **/
class CTemplate : public CObject
{
 public:
  // --------------------------- CONSTRUCTORS ---------------------------------
  // Constructor
  CTemplate(void);

  // Destructor
  virtual ~CTemplate(void);

  // ---------------------------- METHODS -------------------------------------
  // Returns the class name
  const char* getClassName(void) const;


  // ------------------------ MEMBER VARIABLES --------------------------------
```

```
  // Example of a public member variable
  int m_nPublic;

 protected:
  // ----------------------------- METHODS ------------------------------------
  // Example of a protected method
  void protectedMethod(void);

  // ------------------------- MEMBER VARIABLES -------------------------------
  // Example of a protected member variable
  int m_nProtected;

 private:
  // ----------------------------- METHODS ------------------------------------
  // Exmaple of a private method
  int privateMethod(int);

  // ------------------------- MEMBER VARIABLES -------------------------------
  // Example of a private member variable
  int m_nPrivate;
};
#endif IDAV_CTEMPLATE_H_
```

5.3 Source File Template

```
/*-----------------------------------------------------------------------------+
  | @BEGIN_COPYRIGHT                                                            |
  |                                                                             |
  |                       Copyright (C) 1998, 1999, 2000, 2001                  |
  |                                                                             |
  |             Center for Image Processing and Integrated Computing            |
  |                          University of California                           |
  |                             One Shields Avenue                              |
  |                              Davis, CA 95616                                |
  |                                                                             |
  |  Unpublished work-rights reserved under the U.S. Copyright Act. Use,        |
  |  dublication, or disclosure is subject to restrictions.                     |
  |                                                                             |
  | @END_COPYRIGHT                                                              |
  +-----------------------------------------------------------------------------+
  |                                                                             |
  | Component  : CTemplate                          Author: author name         |
  | Filename   : CTemplate.cxx                      Date  : 01-01-2000          |
  | Headerfile : CTemplate.h                                                    |
  |                                                                             |
  +----------------------------- Revision Information -------------------------+
  | $Author: fkuester $
  | $Revision: 1.1.1.1 $
  | $Date: 2000/03/31 22:36:34 $
  | $RCSfile: CTemplate.cxx,v $
  +-----------------------------------------------------------------------------+
*/

/**
 ** CIPIC template for C++ source files.
 **
 ** This is the default template to be used for all projects and development
 ** code at the Center for Image Processing and Integrated Computing (CIPIC).
 ** Just copy the template and perform a "regular-expressing-replace"
 ** on \a CTemplate. I.e. exchange \a CTemplate with the name of your class.
 ** Please replace this entire section with a brief description of the
 ** implemented class. Each method has to be preceeded
 ** by a comment block
 **
 ** @author  author name
 ** @version $Revision: 1.1.1.1 $
 **/

//--------------------------------- Defines -----------------------------------

//-----------------------------------------------------------------------------

//--------------------------------- Includes ----------------------------------
#include <CTemplate.h>
//-----------------------------------------------------------------------------


/**
 ** Constructor
 ** @param   void
 **/
CTemplate::CTemplate()
{
}


/**
 ** Destructor
 **/
CTemplate::~CTemplate()
```

```
{
}

/**
 ** Returns the class name.
 ** @param   void
 ** @return  name of this class
 **/
const char* CTemplate::getClassName() const
{
  return "CTemplate";
}

/**
 ** Description of method.
 ** The first "." terminated sentence will be used for the short documentation
 ** and should reflect the purpose of the method. The remaing lines will
 ** automatically be added to the long documentation.
 ** @param void
 **/
void CTemplate::protectedMethod()
{
  // Just an example
}

/**
 ** This is a dummy method.
 ** Method with param and return value.
 ** @param  value to be added to m_private
 ** @return m_private incremented by value
 **/
int CTemplate::privateMethod(int value)
{
  m_nPrivate += value;
  return(m_nPrivate);
}
```

6 Class Interface

6.1 Data hiding

Class member data variables should be protected via `private` or `protected` in a class declaration. Since `private` access limits a variable's usefulness, you should initially designate all class member variables as `protected` and only promote those variables that meet the following rule to `private`:

- Derived classes will rarely need access to the variable, and when they do it is generally of read only nature.

The use of private results in a well-protected class, though it can limit the derived classes' speed and maximum level of performance. To improve access to private members, use inline methods to access them.

6.2 Virtual destructor

Implement all of your destructors as virtual if your class serves as a base class and will be derived from. In other words, if there is a remote chance that other classes ever derive from it make the destructor virtual. This way proper destruction of derived objects will occur.

6.3 Order by access level

Order your class declaration top-down in the header file by access level: `public`, `protected`, and then `private`. When Waldo decides that he wants to use your class he will browse through your class declaration looking for methods that he can call on the object. By placing private members first Waldo would be immediately confronted with members he cannot use. Therefore, place publicly accessible member functions at the beginning of the header file. Do not have multiple sections of each access level. It is confusing to have to switch back and forth between access levels when trying to learn what a class can do.

6.4 getState(), setState(), and isProperty()

For a given "property" of an object, there should be two access methods. A `getProperty` method that retrieves the value, and a `setProperty` method that sets the value. The data member that actually stores this property should be hidden. You should use the methods instead of the data member wherever possible. The `getProperty` method should be declared const. Get-state methods that return boolean values can be of the form `isProperty` rather than `getProperty`.

Listing 6: Access methods

```
class CExample
{
public:
    CExample()
    {
      m_nValue = -1; m_bIsSet = FALSE;
    }

    int getValue() const
    {
      return m_nValue;
    }

    void setValue(int nVal)
    {
      m_nValue =  nVal;
      m_bIsSet = TRUE;
    }

    bool isValueSet() const
    {
      return m_bIsSet;
    }

protected:
    int    m_nValue;
    bool   m_bIsSet;
};
```

7 Parameter Passing

The appropriate use of pointers and references can simplify code, enhance readability, and greatly improve performance.

Parameter passing guidelines:

1. If the data being passed is a primitive type and/or its size in bytes is small (less than the size of a pointer to that type), then it is acceptable to pass the parameter by value.
2. Pass the parameter as a const reference if the data will not be changed within the function.
3. Pass the parameter as a non-const pointer if the data will be changed inside the function.

Exceptions:

1. If the name of the function implies a change to the passed object, it is acceptable to pass the parameter as a non-const reference, i.e. `getNext(...)`, `getText(...)`
2. If the parameters of the function where the function is frequently called are primarily pointers, then it is acceptable to pass parameters as non-const or const pointers.

The following is a demonstration of various parameter passing methods:

Listing 7: Parameter passing examples

```
typedef struct tagCMyStruct
{
  char szBuf1[5000];
  char szBuf2[5000];
  char szBuf3[5000];
}CMyStruct;

void printMyStructOne(CMyStruct ms)
{
   printf("%s,%s,%s", ms.szBuf1, ms.szBuf2, ms.szBuf3);
}

void printMyStructTwo(const CMyStruct *pms)
{
   printf("%s,%s,%s",pms->szBuf1, pms->szBuf2, pms->szBuf3);
}

void printMyStructThree(const CMyStruct &ms)
{
   printf("%s,%s,%s", ms.szBuf1, ms.szBuf2, ms.szBuf3);
}

void main (void)
{
   CMyStruct ms;
   . . . // Fill ms with some data
   printMyStructOne(ms);
   printMyStructTwo(&ms);
   printMyStructThree(ms);
}
```

`PrintMyStructOne()`, the naive implementation, copies the `CMyStruct` data into its parameter ms, thus it copies 15,000 bytes of data. It is not desirable to copy this much data when it is not necessary.
`PrintMyStructTwo()` does not copy data, but when used, as in `main()` above, you must pass it a pointer. Passing a pointer implies that the data object could very well be modified within the function that is being called. Even though the parameter is a const in the function declaration, it is not clear from the perspective of `main()` that the function being called does not modify the object. The body of `PrintMyStructTwo()` is also slightly more complicated. Now you must deal with pointers and de-referencing within your function.
`PrintMyStructThree()` does not copy data, and because you do not pass it a pointer to the object, it does not imply that the function will be changing the passed object. Note that the body of `PrintMyStructThree()` and the naive method `PrintMyStructOne()` are the same. Also Note that the call in `main()` to `PrintMyStructThree()` looks exactly the same as the call to the naive method `PrintMyStructOne()`. It is just as simple to use as the naive method and looks cleaner than the pointer approach in `PrintMyStructTwo()`.

7.1 What does a pointer imply?

There a couple of base assumptions you should always consider:

1. If you pass an object pointer to a function, it is highly possible that the function changes the data pointed to.
2. If you simply pass the object to a function, it is unlikely that the object will be modified within the function.

Listing 8: Harmless function call

```
void main(void)
{
  double fA(1.0), fB(2.0), fC(3.0);

  fA = sqrt(fB);  // Pretty obvious that fB will NOT be changed within sqrt()
  fC = sin(fA);   // Pretty obvious that fA will NOT be changed within sin()
}
```

Listing 9: Potentially harmfull function call

```
void main(void)
{
   MyStruct ms;

   someFunction(ms);    // Does this function modify the passed object?
                        // It is impossible to tell directly from the
                        // function name.
}
```

Or even worse:

Listing 10: Evil function

```
void someFunction(MyStruct &ms)
{
  strcpy(ms.pBuf1, "Junk");
}
```

This is simply evil to construct a function in this manner. It is not implied from the calling perspective of the function (other than the name of the function) that the object passed will be modified and in this case the name does not imply anything. An exception is something like `getNext(int &nPos)`, where the name of the function hints to the action taken on the object. In this case it is acceptable (but not recommended) to use a non-const reference.

8 Using const

Appropriate usage of the const modifier enforces data encapsulation and protection. It is conventionally ranked in importance about as high as a mall cop, but it has far more power than a mall cop when used appropriately. Use the const modifier liberally and whenever possible. In some cases liberal use of const will bring to light some serious bugs.

Use	Do not use
void foo(const CMyStruct &ms);	void foo(CMyStruct ms);
int getProp() const;	int getProp();
const char* getName() const;	CString getName();
const CMyStruct* getStruct() const;	CMyStruct* getStruct();

Table 9: Using const.

If you want to allow direct access to a member variable via a function call, make two versions of the function:

1. MyStruct& GetStruct();
2. const MyStruct& GetStruct() const;

This will allow both types of access at the appropriate time such as on the left or right side of an assignment operator.

8.1 What does const mean when it's on the right-hand side of a class member function declaration?

Technically, it defines the function to be a constant function. In as simple terms as we can explain, when C++ gets converted to C (C++ is simply a pre-processor for C you may recall), the const on the right side gets placed in front of the this pointer.
Accordingly, the function declarations:

Listing 11: Before pre-processor

```
class CMyClass
{
  .
  .
  .
  int getValueOne();
  int getValueTwo() const;
};
```

Get converted into something like:

Listing 12: After pre-processor

```
int getValueOne(CMyClass *this);
int getValueTwo(const CMyClass *this);
```

Thus, inside a const function the this pointer is declared as const so the calling object cannot be changed.

8.2 It is about as powerful as a mall cop

If you want to you can always cast a constant object to a non-constant reference and then do whatever you want to it. This is why const is a mall cop and not a marine.

8.3 Use const instead of #define

You should use const instead of #define to declare constant global values so that only one copy of the object exists.

Listing 13: Using #define

```
#define BUFSIZE   1024
#define SENTENCE "this is a sentence"

void main (void)
{
   char szText[BUFSIZE];
```

```
    strcpy(szText, SENTENCE);
    .
    .
    .
    if(strcmp(szText, SENTENCE) != 0)
      strcpy(szText, SENTENCE);
    .
    .
    .
}
```

In the listing above character string for SENTENCE is repeated several times. Thus, the storage for this is repeated several times. A better way is as follows:

Listing 14: Using const

```
const int nBufferSize    = 1024;
const char szSentence[] = "this is a sentence";

void main (void)
{
   char szText[nBufferSize];

   strcpy(szText, szSentence);
   .
   .
   .
   if(strcmp(szText, szSentence) != 0)
           strcpy(szText, szSentence);
   .
   .
   .
}
```

Now there is only one copy of the data.

9 Templates

Use templates whenever possible. As soon as you identify that you are customizing a class or method based on the type, convert it to a template. If you make more than one copy of a method because of a varying type, then you have made too many copies. Think one, there can be only one! With this philosophy you will develop much more versatile code.

Listing 15: Template candidate

```
int swap(int a, int b)
{
  int t = a;
  a = b;
  b = t;
}

float swap(float a, float b)
{
  float t = a;
  a = b;
  b = t;
}
```

Should be changed to:

Listing 16: Template based version

```
template <class TYPE>
TYPE swap(TYPE a, TYPE b)
{
  TYPE temp = a;
  a = b;
  b = temp;
}
```

9.1 Use reasonable names for your template parameter variables

You do not have to use T as being the type variable. Select something more descriptive instead. If you are expecting some sort of vector call your type variable VECTOR, if you are expecting an array call it ARRAY. T, T1, and T2, are not descriptive enough.

9.2 Use all capital letters on parameter names

All capitals distinguish the name from the other names in your code. You can then easily identify which names are template parameters and those that are not

9.3 Postfix _TYPE onto the end of parameter names

If the template parameter is a data type, postfix _TYPE onto the name to more easily identify it as a data type parameter.

9.4 Use typedef to make using your template classes easier to use

If you have a template class that expects twenty parameters than it is a pain to write functions that use this class. Using typedef will make it easier to use:

Listing 17: typedef and templates

```
template <class BASECLASS, class VAR, class CHEESE, class VECTOR, int COUNT>
class CTemplateClass : public BASECLASS
{
  .
  .
  .
};

typedef CTemplateClass<CMyClass, int, float, CMyVector, 5> CMyNewType;
```

```
void Print(const CMyNewType &Obj)
{
   .
   .
   .
}
```

9.4.1 Provide easy access to your class template parameters

Provide easy access to your template parameters in by using typedef and static functions:

Listing 18: Easy access to your template parameters

```
template <class BASECLASS, class VAR, class CHEESE, class VECTOR, int COUNT>
class CTemplateClass : public BASECLASS
{
public:
  typedef BASECLASS                          CBaseClass;
  typedef VAR                                CVarType;
  typedef CHEESE                             CCheeseType;
  typedef VECTOR                             CVectorType;
  static int GetCount()                      {return COUNT;}
  typedef CTemplateClass<BASECLASS, VAR, CHEESE, VECTOR, COUNT> CThisClass;
  .
  .
  .
   CTemplateClass(const CThisClass &);       // Much easier to write than
                                             // the alternative
};
```

This will clean up your code and improve readability significantly. Just imagine how long the declaration of an overloaded operator for the class above would be without typedefs.

10 Enumerations

Enumerations is the long lost rich uncle that every programmer forgets about. They are a goldmine of simplification.

Enumeration guidelines:

- Use enumerations instead of #define.
- Encapsulate the enumeration inside a class if you can. This way it can be identified as being closely related to that particular class. This will also prevent a rogue enumeration from appearing in a header file all by itself so that you and Waldo have to search many files to find out where it is used.
- typedef your enumerations to make them more usable.
- Prefix enumeration variable names with enum. This way, when the variable is used you know exactly what it is.

Listing 19: Class with enumeration

```
class CMyClass
{
public:
   typedef enum
   {
      enumSuccess,
      enumNotTooBad,
      enumOops,
      enumOhBoy
   }ERROR;
   .
   .
   .
};
```

Use enumerations for bit masks:

Listing 20: Enumeration as a bit mask

```
typedef enum
{
  enumSuccess   = 0x00,
  enumError1    = 0x01,
  enumError2    = 0x02,
  enumError3    = 0x04,
  enumError4    = 0x08,
  enumError5    = 0x10
} BITMASK;
```

Even for keeping track of a class version:

Listing 21: Class versioning

```
class CMyClass
{
public:
  typedef enum
  {
     enumVersion1         = 0x0100,
     enumVersion2         = 0x0101,
     enumVersion3         = 0x0200,
     enumCurrentVersion   = enumVersion3  // Yup, two vars can have
                                          // the same value
  } VERSION;
  .
  .
  .
};
```

See the Tips and Tricks section for more information on class versioning.

11 Class Member Declaration List

Overuse of declaration lists should be avoided.

Listing 22: Over use of declaration list

```
class CMyClass
{
public:
  CMyClass() : a(0), b(0), c(0), d(0), e(0), f(0), g(0.0f), h(0.0f), i(0.0f), j(0.0f) {}
  .
  .
  .
protected:
  int   a, b, c, d, e, f;
  float g, h, i, j;
};
```

Is the body of the constructor such a sacred place that normal variable declarations cannot be placed in there? Just move them into the constructor and only put things you really need to in the declaration list.

12 Reference Counting

Include the copy constructor when you need to count object instantiations. The compiler makes a default copy constructor if you do not specify one, and this one has no idea that you are trying to count object instantiations.

Listing 23: Proper reference counting

```
class CMyClass
{
public:
    CMyClass()                          {m_nRef++;...}
    CMyClass(int n)                     {m_nRef++;...}
    CMyClass(const CMyClass &mc)        {m_nRef++;...}
    virtual ~CMyClass()                 {m_nRef--;...}
    .
    .
    .
protected:
    static int m_nRef;
};
```

13 Multiple Constructors in a Class

When you have multiple constructors in a class, write a single method called `construct()` that initializes all of your member variables. Then in each constructor, call `construct()` on the first line.

Listing 24: Multiple constructors

```
class CMyClass
{
public:
   CMyClass()
   {
     construct();
   }

   CMyClass(int n)
   {
     construct();
     a = n;
   }

   CMyClass(int n, int k)
   {
     construct();
     a = n;
     b = k;
   }

   CMyClass(const CMyClass &Obj)
   {
     construct();
     *this = Obj;
   }

protected:
   void construct()
   {
     a = 0;
     b = 1;
     c = 2;
     d = 3;
     e = 4;
   }

   int a, b, c, d, e;
};
```

This way if you add new member variables, you only need to update one function rather than going to each constructor and figuring out what to do. This greatly reduces the chance of forgetting a constructor or a member variable.

14 Do not Be Too Hasty

It may seem like a good idea to do something like:

Listing 25: Destroy the virtual function table

```
class CMyClass : public CBaseClass
{
public:
   CMyClass()
   {
     memset(this, 0, sizeof(CMyClass));
   }
   .
   .
   .
};
```

In the simplest case this will work fine and achieve what you expect (initialize all variables to 0). Though, if a virtual function is declared anywhere in the class hierarchy then you would be doing something very bad. The call to memset() will overwrite the virtual function table for the object because sizeof() includes the virtual function table in its calculation. This will create all kinds of strange problems. If you really want to do something like this, a somewhat safer approach would be:

Listing 26: Do not destroy the virtual function table

```
class CMyClass : public CBaseClass
{
public:
  CMyClass()
  {
    memset(&m_pStart, 0, &m_pEnd - &m_pStart);
  }

protected:
  unsigned char m_Start;
  .       // Include variable declarations here
  .       // that you want to clear
  .
  unsigned char m_End;
};
```

This way all the variables that are declared between the start and end markers are set 0. This is still kind of hokey though. You probably should not do it, but it is worth noting so that you do it safely.

15 Namespaces

Do not prefix your class, method, or variable names with a concocted abbreviation that uniquely identifies your code. Use name spaces to make your classes and functions unique.

Listing 27: In dire need of namespaces

```
extern int mylibValue;
extern int mylibCount;

class mylibVector
{
  . . .
};

float mylibMagnitude(const mylibVector &v);
float mylibDotProduct(const mylibVector &va, const mylibVector &vb);
```

Should be changed to:

Listing 28: Uses namespace properly

```
namespace MyLib
{

  extern int nValue;
  extern int nCount;

  class CVector
  {
    . . .
  };

  float magnitude(const CVector &v);
  float dotProduct(const CVector &va, const CVector &vb);

}; // namespace MyLib
```

To use the namespace you can either place using `namespace MyLib` at the top of each file you use the namespace's members in or use the namespace selectively: `MyLib::magnitude(v)` very similar to accessing a public static member function in a class. The use of namespace clarify code and reduce a lot of extra typing.

16 Debugging

- Permanent Debug Code
- Temporary Debug Code
- Code Requiring Additional Work
- Code Requiring Additional Testing or Fixing

16.1 Permanent Debug Code

16.1.1 std::cerr and std::cout

Most of us have instrumented code with print statements before to either debug code or print status information:

Listing 29: Debug messages

```
std::cerr << ''This is a debug message to cerr''
std::cout << ''This is a debug message to cout''
```

This however introduces quite a few constraints. For examples, the user will not be able to pipe all system output to a file if `cerr` was used instead of `cout` to print information.

16.1.2 What happens when you are done debugging your code?

Normally you will go through your source code and remove the print statements or wrap them with some sort of pre-processor flags.

Listing 30: Preprocesor wrapper

```
#ifdef DEBUG
std::cout << ''This is a debug message to cout''
#endif
```

In this case the debug information will be only included if DEBUG was defined. This is fine and dandy if you are willing to recompile your code everytime you want to debug particular code segments.
Alternatively, you can write custom macros that allow to control debug functionality through environment variables. You than can define debug categories and debug levels without having to recompile your code.

Listing 31: Using environement variables for debug purposes

```
setenv VE_DEBUG_LEVEL     1
setenv VE_DEBUG_CATEGORY ''VE RPS VUI DSO''
```

Listing 32: Improved debug statements

```
VEDEBUG(category, level) << ''This is a debug line''

VEDEBUG(category, level)
  << ''This is a multi-line debug message: ''
  << ''Category: '' << category
  << ''Level    : '' << level;
```

The debug macro can then compare debug categories and levels against the specified environment variables. If the category matches and the specified debug level is equal to or bigger than the one in the code, output is generated. Otherwise it is suppressed. This mechansim allows you instrument your code for later debugging at all critical places.

17 Development Flags

Sometimes you decicide to implement quick and dirty hacks to allow for early testing of code segments under development. Or your specifications require you to provide certain functions and you initially decide to simply implement empty wrapper functions until you find some more time to actually write the real thing. If you work on bigger source code distributions it is easily forgotten that one of these improvised code segments exists. We therefore recommend that you always flag code if it has not yet been implemented, finished or fully tested.

- XXX:INCOMPLETE

- XXX:FIXME
- XXX:NOTIMPLEMENTED

At any point in time you can then simply search for all occurences of XXX in your source code and locate all the segments that still need fixing.

17.1 Code Requiring Additional Work

Listing 33: Incomplete Code

```
/**
 ** Magically creates good code.
 **/
void CStandards::writeGoodCode()
{
  parseStandards():
  applyRules()
  :
  // XXX:INCOMPLETE
  :
}
```

17.2 Code Requiring Addtional Testing and Fixing

Listing 34: Broken Code

```
/**
 ** Adds two integers and returns the result
 ** @param a  - int value
 ** @param b  - int value
 ** @return sum of the two integer values
 **/
int CMath::add(int a, int b)
{
  // XXX:FIXME
  // Forgot how to add two numbers. Return the first one
  // until I figure out how to do it :-)
  int result = a;

  return(result)
}
```

17.3 Missing impementation

Listing 35: Missing Code

```
/**
 ** Returns the magnitude of the vector.
 ** @param   none
 ** @return  float - length of the vector
 **/
float CVector::getLength()
{
  // XXX:NOTIMPLEMENTED
  return 0.0;
}
```

18 Documentation

The following flags are readily supported by a variety of documentation tools.

- @author
- @version
- @date
- @bug
- @warning
- @param
- @return
- @exception
- @see

18.1 Class documentation

Listing 36: Class documentation block

```
/**
 ** VIS template for C++ source files.
 **
 ** This is the default template to be used for all projects and development
 ** code at the Visualization and Interactive Systems Group (VIS).
 ** Just copy the template and perform a "regular-expressing-replace"
 ** on \a CTemplate.
 **
 ** @author  author name
 ** @version $Revision$
 ** @date    $Date$
 **/
```

18.2 Function documentation

Listing 37: void function without parameters

```
/**
 ** Prints hello world.
 **/
void printHelloWord()
{
 :
}
```

Listing 38: void function with parameters

```
/**
 ** Prints hello world.
 ** @param bold - [true/false] typesets the text in bold if true, normal otherise
 **/
void printHelloWorld(bool bold)
{
  :
}
```

Listing 39: non-void function without parameters

```
/**
 ** Prints hello world.
 ** @return true if successful, false otherwise
 **/
bool printHelloWorld()
{
   :
  if(failure)
    return(false)
   :
  return(true)
}
```

Listing 40: non-void function with parameters

```
/**
 ** Prints hello world.
 ** @param  bold - [true/false] typesets the text in bold if true, normal otherise
 ** @return true if successful, false otherwise
 **/
bool printHelloWorld(int bold)
{
  :
}
```

19 Tips and Tricks

19.1 Pointers

Here are some helpful templates for handling memory deletion:

Listing 41: Helpful memory deletion templates

```
template <class TYPE>
inline void DELETE_POINTER(TYPE * &pointer)
{
  delete pointer;
  pointer = NULL;
}

template <class TYPE>
inline void DELETE_ARRAY(TYPE * &pointer)
{
  delete [] pointer;
  pointer = NULL;
}
```

Using the above templates does the following important things:

- Sets the pointer to NULL upon deletion. This prevents stale pointer variables.
- Makes the choice between “array” and “normal” memory deletion more obvious. It is too easy to forget the [] for an array. Having to type either “ARRAY” or “POINTER” makes you think about the decision a bit more.

19.2 Copy constructor and assignment operator relationship?

Since these two methods are essentially the same (aside from how they are invoked) you can simplify the copy constructor declaration by initializing the data members and then calling the assignment operator directly. This will simplify adding new member variables and eliminate the chance of the two methods getting out of synchronization.

Listing 42: Merging the copy constructor and assignment operator

```
class CMyClass
{
public:
  CMyClass()
  {
    construct();
  }

  CMyClass(const CMyClass &obj)
  {
    construct();     // Initialize members
    *this = obj;     // Call assignment operator directly
  }

  CMyClass& operator= (const CMyClass &obj)
  {
    m_nA = obj.m_nA;
    m_nB = obj.m_nB;
  }

protected:
  void construct()
  {
    m_nA = m_nB = 0;
  }

  int m_nA, m_nB;
};
```

19.3 Passing structs as parameters

There are two cases that benefit greatly from having a structs passed as the parameter rather than its individual parameter counterpart:

- When passing a large number of parameters to any method. When parameter list become long, it becomes more difficult to remember the order of the parameters and even the parameters themselves. Passing a single struct object relieves this headache.
- If function call order is important, make a single function that wraps the sequentially called methods and takes a single struct as the parameter. This eliminates the need for Waldo to know that the methods need to be called in a particular order and makes it easier to pass the parameters.

The following listing shows some methods in serious need of parameter passing help:

Listing 43: In need of parameter simplification

```
class CMyClass
{
public:
   CMyClass() {}

   void one(int nA, int nB, int nC, int nD, CMyClass *pObj)
   {
     pObj->m_nVal = nA * nB* nC;
     m_nVal       = nD;
   }

   void two()
   {
     m_nVal *= 4;
   }

   void three(int nE, int nF, int nG, CMyClass *pObj)
   {
     pObj->m_nVal = m_nVal * nE * nF * nG;
   }
protected;
   int m_nVal;
};
```

Should be changed to:

Listing 44: Much better

```
class CMyClass
{
public:
   CMyClass(){}

   typedef struct
   {
     int nA, nB, nC, nD, nE, nF, nG;
     CMyClass *pObj;
   }CData;

   void Method(CData *pData)
   {
     one(pData);
     two();
     three(pData);
   }

protected;
   void one(CData *pData)
   {
     pData->pObj->m_nVal = (pData->nA) * (pData->nB) * (pData->nC);
     m_nVal              = pData->nD;
   }
```

```
    void two()
    {
      m_nVal *= 4;
    }

    void three(CData *pData)
    {
      pData->pObj->m_nVal = m_nVal * (pData->nE) * (pData->nF) * (pData->nG);
    }

    int m_nVal;
};
```

19.4 Equality expression evaluation: (1 == n), (1 ¡= n), etc.

When evaluating equality expressions, try to place the constant parameter (if there is one) on the left-hand side. This will reduce the chances of accidentally forgetting an equal (or less-than) sign and thus assigning a variable incorrectly. If the parameter on the left-hand side is constant, the compiler will catch to error if you do make an assignment mistake.

19.5 Versioning

Class versioning can be handled elegantly by adding an enumeration to the class.

Listing 45: Class versioning

```
class CMyClass
{
public:
  typedef enum
  {
    enumVersion1        = 0x0100,
    enumVersion2        = 0x0101,
    enumVersion3        = 0x0200,
    enumCurrentVersion = enumVersion3
   }VERSION;
.
.
.
void save(ostream &os, const VERSION nVersion)
{
  os << nVersion;          // Save the version number
  os << m_nA;              // Version 1

  if(nVersion >= enumVersion2)
     os << m_nB;           // Version 2

  if(nVersion >= enumVersion3)
     os << m_nC;           // Version 3

   return os;
}

void load(istream &is)
{
  int nVersion;

  is >> nVersion;          // Read version
  is >> m_nA;              // Version 1

  if(nVersion >= enumVersion2)
    is >> m_nB;            // Version 2

  if(nVersion >= enumVersion3)
     is >> m_nC;           // Version 3
}
```

```
protected:
    int m_nA, m_nB, m_nC;
};
```

19.6 Line grouping and commenting

Try to group lines of code in small groups (less-than 20 lines per group) and identify the groups with comments to aid in readability. Separate groups by at least one blank line.
Here is a piece of code showing grouping.

Listing 46: Line grouping

```
void CFieldDefinition::DrawField(CDC *pDC)
{
    // Local variable definitions
    CPen     penNormal(PS_SOLID, 0, RGB(144, 144, 144));
    CPen     penRemove(PS_SOLID, 0, RGB(144, 144, 0));
    CPen     penMultiplePass(PS_SOLID, 0, RGB(0,0,225));
    CPen     penInflection(PS_SOLID, 0, RGB(192,0,0));
    CPen     penRed(PS_SOLID, 0, RGB(192,0,0));
    CBrush   brushRed(RGB(192, 0, 0));
    CPt      *pPt = NULL;
    CFont    Font;

    // Save the DC state
    pDC->saveDC();

    // Setup the font
    Font.createPointFont(70, "Arial", pDC);
    pDC->selectObject(&Font);
    pDC->setTextColor(RGB(0,0,0));
    pDC->setBkMode(TRANSPARENT);
    pDC->setTextAlign(TA_CENTER | TA_BASELINE);
    .
    .
    .
};
```

Even though you have no idea what the function is supposed to do, you can generally figure out what it is doing just by quickly scanning the line groups and comments. You can quickly scan the comments to find the section of code that you want to find. Once found, you can more closely examine the lines of code.

20 Discussion Points

20.1 Error handling

20.1.1 Exceptions

Please email me any reason to use exceptions, unless it is one of my own exceptions listed here:

- To catch real exceptions caused by hardware or OS failure: power shutoff to something, network cable unplugged, out-of-memory, etc.
- To catch other people's cop-out exceptions.
- To catch other people's laziness exceptions because they didn't want to handle an error condition.
- To catch other people's solutions to uncommon situations.

You might learn something by trying to handle some of these "uncommon" situations. There could be a very good reason for a method failure, figure it out. Exceptions obscure code readability and require Waldo to use exceptions as well to handle your errors. Far too often we have seen code that uses exceptions as a cop-out to handle reasonable errors.

20.1.2 Method success and failure

A very reasonable method of handling method success and failures is to only have methods returning either void or BOOL types. Methods returning void are assumed to work correctly every time. Methods returning a BOOL type return TRUE on success and FALSE when an error occurs. Companies have been very successful in using this type of error handling saving exceptions for things like database and file access as well as other OS failures (true "uncommon" non-deterministic cases). Success of a method is tested very easily in an if statement and errors can be propagated easily to calling methods.

20.1.3 Curly brace placement

About 60-70% of people use curly braces the same way outlined earlier, whereas the rest of the people place the opening brace on the end of a line. My argument for not placing braces at the end of lines is two-fold. First, if the line scrolls off the screen, you can't see the brace anymore. Two, when scanning code to find blocks (defined by braces) it is easier and faster to just move your eyes vertically, rather that both vertically and horizontally to find them. By placing braces at the beginning of their own lines it is much easier to decipher code blocks. Comments can even be added immediately after the opening brace (on the same line) to hint to what that block is trying to accomplish.