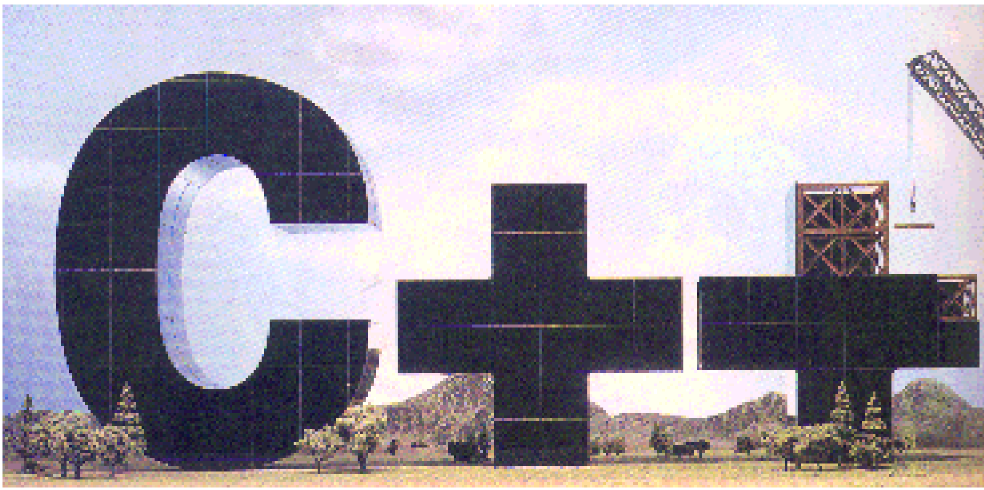
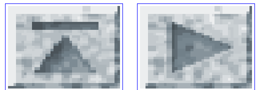




C++ LANGUAGE CODING STANDARD

Revision Date: 1 June 1995

Prepared For:

```
NRaD
271 Catalina Blvd.
San Diego, CA  92152-5000
```

Prepared By:


```
AHNTECH, Inc.

5575 Ruffin Road, Suite 100

San Diego, CA  92123
```


Abstract

This document is provided as a guideline for the creation of C++ code whose maintenance and enhancements may involve multiple programmers. In addition to providing for better team development, a standard also allows for the development of software tools to aid in the creation, testing, and

documentation of code. A list of tools automating code standardization is available for downloading and is referenced in this document.

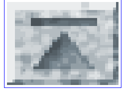
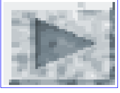
Table of Contents

1. Introduction
    - Purpose
    - Responsibility
    - Interpretation
        - Conventions
        - Terminology
        - Language Reference Manual
        - Notation
2. C++ Coding Standard
    - General Presentation Style
        - Line Lengths
        - Indentation, Spacing And Blank Lines
        - Blocks And Statements
        - Uniform Presentation Of Information
            - Prolog And Commentary
                - Prolog
                    - Content
                    - Format
                - Commentary
                    - Block Commentary
                    - In-Line Commentary
            - Layout Of Source Code Files
                - Header Files
                - Source Files
        - Size Of Code Aggregates
    - Naming Conventions
        - General
        - Abbreviations
        - Variables

            - Constants
            - Functions
            - Classes
            - Class Member Data
            - Typedefs
    - Usage Of The Implementation Language
        - Macros (#define ...)
        - Literals
            - Character Constants
            - String Literals
            - Numeric Literals
        - Variables
        - Structures, Unions and Typedefs
        - Classes
            - General
            - Member Functions
            - Constructors and Destructors
        - Templates
        - Control Structures
            - Functions
            - Branching
                - goto statements
                - if ... else statement
                - switch - case statement
                - Looping -- do, while and for statements
                - Conditionals
    - Error Processing
        - General Principles
    - C++ Language Specific Concerns
        - Location Of Declarations And C++ Scope Rules
        - Class Packaging
3. Program Design Language
    - General Description

    - PDL Commentary
4. Complete Example

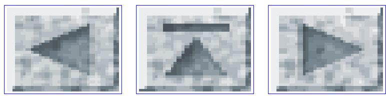
$Revision: 1.1.1.1 $ - $Date: 1995/08/24 17:15:19 $

Coding Standards

- Ellemtel
- NRAD

Bill Oakley Last modified: Mon Jun 29 10:36:00 MDT 1998


C++ LANGUAGE CODING STANDARD

Introduction

Purpose

The purpose of this document is to provide a standard for code written in the C++ programming language.

The standard is intended to increase readability, reliability, testability, maintainability and portability of C++ programs. Using a consistent style produces a final product which appears as the creation of a team working together as opposed to individuals working independently and attempting to combine their efforts late in the process.

Responsibility

This document may be applied to all programs that use the C++ language. Programs which are affected by this standard are required to either use this standard as written, or to produce a project-specific tailoring of this standard.

Interpretation

Conventions

The use of the word "shall" in this document requires that any project using this document must comply with the stated standard.

The use of the word "should" directs projects in tailoring a project-specific standard, in that the project must include, exclude, or tailor the requirement, as appropriate.

The use of the word "may" is similar to "should", in that it designates optional requirements.

Terminology

For the sake of simplicity, the use of the word "compiler" means compiler or translator.

"C++ Coding Standard" refers to this document whereas "C++ ANSI Standard" refers to the proposed C++ language definition.

The term "method" means function. Usually, a method is a function that is a member of a class.

Language Reference Manual

Subject to the provisions of this document, the following applicable documents shall be used to define the programming language, to interpret terms, and to provide examples.

The ANSI standard reference manual for C++, when formally available, shall be the preferred document to use.

Alternate reference manuals will be considered as follows:

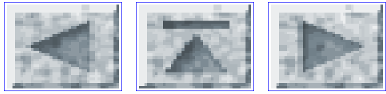
- Documents based on the draft-proposed ANSI C++ Standard until the formal ANSI C++ standard is released.
- In those cases where an ANSI C++ compiler is not available, "THE ANNOTATED C++ REFERENCE MANUAL:, by Ellis and Stroustrup, Addison Wesley, 1990.

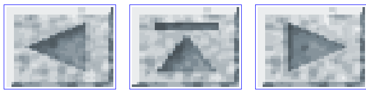
Notation

Within this document, the following notation is used:

"..." stands for any valid construct.

$Revision: 1.2 $ - $Date: 1996/01/18 16:42:22 $





C++ Coding Standard

This section defines specific restrictions and conventions placed upon the usage of the C++ programming language.

General Presentation Style

An editor which can be modified to format code according to the conventions set forth in this standard should be used. An alternative is the use of a source code formatting tool. A tool to format code specifically to this standard is available.

In the absence of such an editor, source development tool, or source code formatting tool, the following indentation and spacing conventions shall be applied.

The indentation, spacing, and alignment conventions which follow are based on the C formatting tools "indent" and "cb", available with most UNIX systems.

Some of the conventions allow two different formats. One format is labeled the default format. The other is labeled the optional format. Whenever code is delivered outside of the development group, it shall be in the default format.

Line Lengths

* Maximum length of a source code line shall be 78 characters, including preceding blank characters.

* The line length for block comments shall be 78 characters, including preceding blank characters.

Indentation, Spacing And Blank Lines

* Standard indentation shall be one tab stop.

* There shall be at least one space before and after each operator or assignment symbol.

* Leading and trailing spaces inside a parenthesis pair shall not be required. They are optional. However, the default is that both the leading and trailing spaces shall appear. For example:

```
        strcpy( destination, source );  // default: both spaces appear
        strcat(first, second);          // optional: neither space appears
```

* Function calls shall not have a space inserted between the name and the "(". For example:

```
        strcpy( destination, source );  // correct
        strcat ( destination, source ); // incorrect
```

* The indentation, in character positions, for an identifier following a declaration keyword shall be at a

multiple of standard indentations. (This rule applies only to declarations at the beginning of header files, source code files and functions, not "inline" declarations). Within class definitions, identifiers may line up within the "private", "protected" and "public" areas, not necessarily within the entire class definition. For class member data, the "_" in names should line up to further increase readability.

For example,

```
class Class_Name
{

private:

        int       _long_parameter_name1; // This is parameter 1.
        char     *_long_parameter_name2; // This is parameter 2.

protected:

        Long_Type_Name  _parameter3;    // This is parameter 3.
        int             _parameter4;    // This is parameter 4;

public:

        int      &_parameter5;          // This is parameter 5.
        int       _parameter6;          // This is parameter 6.

};
```

* There shall be only one variable for every declaration key word. For example:

```
        int counter = 0;                // Loop counter.
        int reset_flag = 0;             // Flag to reset the counter.
        int i = 0, j = 0;               // NOT ALLOWED
```

* Inline comments associated with declarations shall be aligned with one another, in a column which also is associated with a tab stop.

```
                                        +----Tab Aligned in one column
                                        V
        int counter = 0;                // Loop counter.
        int reset_flag = 0;             // Flag to reset the counter.
```

* Directives shall begin with the "#" character in column 1 with no spaces between the "#" and the directive.

```
#define ...
#include ...
#ifdef ...
```

Not

```
#       define ...
#       include ...
#       ifdef ...
```

* Continuation lines will be indented one tab stop relative to the beginning of the first line of a long statement. For example, here is how a piece of continued code should look:

```
        very_long_variable = array_of_records[array_index].element->
                another_record.another_element;
```

* If a parenthesized expression is continued, the continuation lines shall be lined up to start at the character position just after the left parenthesis. If all parameters in a parenthesized expression cannot fit on one line, then all parameters will be on their own line, lined up just after the left parenthesis. Alternately, the line can break at the equals sign for an assignment operation, or after the left parenthesis with the parameters indented one tab stop relative to the beginning of the first line of the statement.

For example, here are valid long, parenthesized statements:

```
        p_a = call_the_procedure(this_fits, so_does_this, and_this_also);
        p_b = first_procedure(second_procedure(p1, p2),
                              third_procedure(p3, p4),
                              fourth_procedure(p5, p6));
        p_c = call_another_procedure(
                first_parameter,
                second_parameter,
                third_parameter);
        long_parameter_name =
                last_procedure(now_the_parameters, will_fit, on_one_line);
```

* Blank lines shall be used to provide visual separation of source code blocks. * There shall be a blank line around every conditional compilation block (i.e., in front of every #ifdef and after every #endif.)

* There shall be a blank line after every block of declarations.

* There shall be a blank line before every block comment.

* There shall at least one blank line after every function body.

Blocks And Statements

There are two acceptable ways in which braces may be positioned when used in conjunction with controlling expressions like for(..) and while(..).

In the default rule, the statement(s) shall be formatted with the initiating brace on the line following the ending parenthesis for the controlling expression. The initiating brace shall appear at the same indentation as the initiating keyword. The terminating brace shall appear on a separate line at the same indentation as the initiating keyword. Each brace shall be the only non-blank character on the line.

Example:

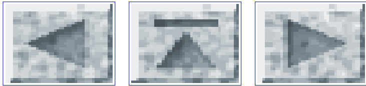
```
        while (...)
        {
                code
        }
```

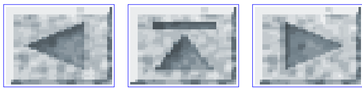
In the optional second rule, the statement(s) shall be formatted with the initiating brace on the same line as the controlling expression. The terminating brace shall appear on a separate line at the same indentation as the initiating keyword.

Example:

```
        while (...) {
                code
        }
```

$Revision: 1.1.1.1 $ - $Date: 1995/08/24 17:15:18 $





Uniform Presentation Of Information

Prolog And Commentary

Prolog

A prolog shall be created for each source file and header file and for each class definition and function definition within a file. The intent of a prolog is to provide a synopsis of the source file, header file, class definition, or function definition.

The formatting of the prologs is especially important if software tools are to be utilized to generate end user documentation. A tool to generate documentation from source code utilizing the information described in this section, called CPPLIB2DOC, is available and referred to throughout this document.

Content

A **Source File Prolog** shall contain the following information:

- The name of the source file.
- Author and creation date.

A **Header File Prolog** shall contain the following information:

- The name of the header file.
- Author and creation date.

A **Class Prolog** shall contain the following information:

- Name of the class.
- Description of the class.

The builder of the class should write the Class Prolog description as if it is the User's Guide for the class.

A **Function Prolog** shall contain the following information:

- Name of the function, and if applicable, the name of the class that contains the function. Note that arguments and return type should NOT be specified, only the name of the function.
- A description of what the function does.
- Input Parameters - Each paramater name should appear on a new comment line with the name preceded by a colon. Descriptions can span more than one comment line but may NOT contain colons. If there are no input parameters, use "void" in the list.
- Description of returned value(s). This includes documentation on modified input parameters or structures modified which were referenced by parameters. If there are no returned values, use "void" in the list. The return value description is omitted on class constructors and destructors.

The input parameters and returned values descriptions are optional for simple, in-line functions (such as member access functions) within the class definition file. Often, the description of these parameters can provide no additional information or would only repeat the information in the description of these functions.

There must be a Function Prolog for each function, wherever that function is located. That is, for pure virtual functions, member access functions, empty constructors and destructors and other functions that have no function bodies in the source file, the Function Prolog will be in the header file. Otherwise, the prolog is in the source file with the function body, and further commentary on the function in the header file is not necessarily needed.

Format

All prologs shall be presented in a block comment format, which is described in the section titled "Block Commentary"

Each item in the prolog (such as FILE) shall be labeled in all capital letters, followed by a colon (:). The colons shall be aligned.

A blank comment line shall separate each item in the prolog.

The creation date shall be placed after the name of the author, on the same line.

Any additional information, such as special characters used by revision control and configuration management tools, specified by the program shall be placed just after the prolog.

The specific format for each type of prolog shall follow the examples below.

Example of **File Prolog** (header and source):

```
//
// FILE           : Complex.h
//
// AUTHOR         : Jane P.Jones,  13 Sept 90
//
```

Example of **Class Prolog**:

```
//
// CLASS          : Circle
//
// DESCRIPTION    : To provide methods for manipulating circles.
//
```

Example of **Function Prolog**:

```
//
// FUNCTION       : Customer_Interface::read_customer_order
//
// DESCRIPTION    : Reads the order from the database.
//
// INPUTS         : order_number - which order to read
//                : order        - the order data structure
//
// RETURNS        : fills in order in the referenced order structure
```

```
//
```

Another example of **Function Prolog**, for a member access function in the header file:

```
//
// FUNCTION     : Customer_Interface::get_customer_order
//
// DESCRIPTION  : Returns the pointer to the customer's order.
//
```

Commentary

Comments shall not simply restate the C++ syntax or semantics, but shall clarify the associated objects and functions at a more descriptive level than the source code. Comments should be grammatically correct, and shall address a reader who is a C++ programmer. Certain comments, such as for class member data and global constants and variables, which will be incorporated into the documentation should be written as a sentence with appropriate capitalization and periods.

Commentary shall be associated with each occurrence of:

- class definition
- data (variables, constants, etc.) definition
- function definition
- macro definition
- loop or block structure
- "goto" statement
- "return" statement

Block Commentary

Block commentary shall be used for prologs and any other multi-line comments.

Prologs shall be aligned at the left margin. Block comments associated with compound statements shall be aligned at the same level of indentation as the compound statement being explained.

Double slash ("`//`") shall be placed at the left edge of all lines in block commentary. The slash-asterisk style of commentary ("`/*.....  */`") shall not be used in block commentary.

In-Line Commentary

All data type, variable and constant declarations should be commented when such commentary aids in understanding the code or when required to support automated document generation.

Comment statements shall be located either before the statement being clarified, or to the right of the statement, based on available space and programmer preference. Comments which are not to the right of code shall be placed at the same indentation level as the surrounding code.

For comments that must follow a statement and have alignment requirements, comments should be on the following line and indented to the correct tab stop.

Double slash ("`//`") is the required style of commenting. The slash-asterisk style ("`/*  .....  */`") is not

allowed.

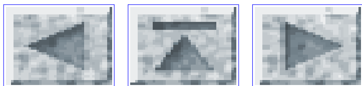
Commentary examples:

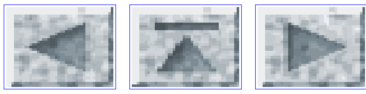
```
        // This is an example of in-line commentary
        code;           // This is also a comment

        int     var1;   // This is a data comment.
        int     variable1;
                        // This is the comment for variable1 which lines up
                        // with the comment for var1.
```

A single comment line in a class description that is surrounded by blank lines and prefaced by the keyword HEADER will become a header in the class description documentation automatically produced by CPPLIB2DOC. Therefore, these comments are a way to provide some separation between class member data and functions, and such comments shall be written as that "header" will appear in a document. For example,

```
class My_Class
{

public:
        . . .

        // HEADER       : Member Access Functions

        . . .

        int get_number(void)
        {
                return . . .
        }
```

$Revision: 1.1.1.1 $ - $Date: 1995/08/24 17:15:18 $





Layout Of Source Code Files

C++ source code files are divided into header files containing the class definition and source file(s) containing the member functions for the class. The header file and source file have the same name, the class name, with different extensions, ".h" and ".cc" respectively. There is only one class per header and source file. Source files containing global functions not associated with a class may be named whatever is appropriate.

Header Files

Header files shall be laid out in the following manner, except if modification is required by configuration management or program library tool requirements:

- The header file prolog.
- '`#include`' statements for any and all included files. The header files will be included in the following order:
  - System files
  - Library files
  - Application-specific files
  - Local files
- Any '`#define`' statements local to the header file and it's corresponding source file.
- Any exportable variable declarations (globals).
- Any local variable declarations.
- The class definition.

Example:

```
//
// <Header File Prolog>
//                . . .
//

#include . . .          // Included files
#include . . .

#define XYZ              // Constants

// Globals
const int        very_one_needs;        // global constant

//  Local to this source file (not in a class)
static char      my_char;               // variable used in this file only
static char     *my_cat(void);          // forward reference
```

```
//
// <Class Prolog>
//                  . . .
//
class My_Class
{
        . . .
}
```

Source Files

Source files shall be laid out in the following manner, except if modification is required by configuration management or program library tool requirements:

- The source file prolog.
- '`#include`' statements for any and all included files. The header files will be included in the following order:
  - System files
  - Library files
  - Application-specific files
  - Local files
  - Header file for the class
- Any '`#define`' statements local to the source file.
- Any exportable variable declarations (globals).
- Any local variable declarations.
- Functions.

Example:

```
//
// <Source File Prolog>
//                  . . .
//

#include . . .          // Included files
#include . . .

#define XYZ             // Constants

// Globals
const int        every_one_needs;       // global constant

//  Local to this source file (not in a class)
static char      my_char;               // variable used in this file only
static char     *my_cat(void);          // forward reference

//
```

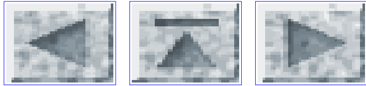
```
// <Functions Prolog>
//                 . . .
//

void my_function(void)
{
        . . .
}
```

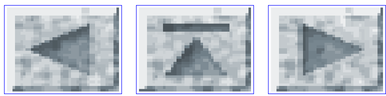
Size Of Code Aggregates

A C++ function should be limited to 100 lines of code. C++ source files should be restrained to 2 or 3 pages of printed source code, exclusive of prolog.

Clarity, readability, modularity and cohesion shall not be sacrificed to meet code size constraints.

$Revision: 1.1.1.1 $ - $Date: 1995/08/24 17:15:18 $



 C++ LANGUAGE CODING STANDARD

Naming Conventions

General

The program should define standard prefixes to be used in class names and consequently, any header and source file names. The prefix shall be two or three characters, the first letter being a capital letter. An underscore shall separate the prefix from the rest of the name, e.g. Ag_Class_Name. The list of approved prefixes should be maintained and be included in any software design documentation.

Abbreviations

A list of approved abbreviations should be established and the list be included in any software design documentation.

These abbreviations should be used in construction of C++ names. Words containing five or fewer letters should not be abbreviated. No abbreviation shall be included in the program approved list unless the following applies:

- It is a well-known and understood abbreviation or it is a commonly used acronym in the program domain.
- The abbreviation will result in significantly shorter C++ names without loss of readability.

Example:

```
typedef struct
{
. . . .
} Call_Detail_Record;
. . .
Call_Detail_Record call_detail_record_array[MAX_RECORDS];
. . .
call_detail_record_array [index].element = . . .
```

Could be replaced by:

```
typedef struct
{
. . . .
} CDR;
. . . .
CDR cdr_array[MAX_RECORDS];
. . .
```

```
        cdr_array[index].element = . . .
```

Variables

Variable names shall begin with a lower case alpha character and should be constructed from lower case alpha characters, the underscore, and numbers.

Examples:

```
        index
        customer_name
        part_number
        caller_id
        bit_3
```

Constants

Constants shall be defined via the "const" type-specifier. The "`#define <name> <value>`" construct is not recommended. Constants follow the same naming conventions as variables.

Examples:

```
const int       max_users = 20;
const int       file_read_error = -163:
const int       mask_I = OXcc;
```

Functions

Function names shall begin with a lower case alpha character and shall be constructed from lower case alpha characters, the underscore, and numbers.

Function names should describe the action or specific algorithm performed by the function.

Examples:

```
        generate_report( ... )
        create_window( ... )
        sort( ... )
        get_bit_3( ... )
```

Classes

Class names shall begin with an upper case alpha character and shall be constructed from lower case alpha characters, the underscore, and numbers. If a class name contains an underscore, the character after the underscore shall be an upper case alpha character or a number.

Examples:

```
Complex
List
Employee
Order_Number
Ordinal_3
```

Class Member Data

Class member data names shall begin with an underscore and shall be constructed from lower case alpha characters, the underscore, and numbers.

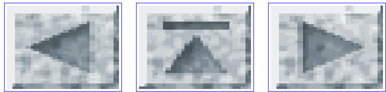
Examples:

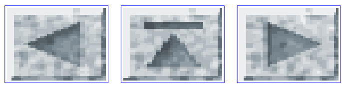
```
_filename
_order_number
_employee
```

Typedefs

Typedefs shall follow the same naming conventions as Classes.

$Revision: 1.1.1.1 $ - $Date: 1995/08/24 17:15:18 $





Usage Of The Implementation Language

Macros (#Define ...)

The use of macros should be avoided for the purpose of maintainability.

The logic of macros should be coded into inline functions instead.

Example:

```
#define  MAX(x,y)       (((x) > (y) ? (x) : (y))        // Get the maximum
```

The macro above can be replaced for integers with the following inline function with no loss of efficiency:

```
        inline int max(int x, int y)
        {
                return (x > y ? x : y);
        }
```

Macros should be used with caution because of the potential for error when invoked with an expression that has side effects.

Example:

```
        MAX(f(x),z++);
```

Literals

Literals are often referred to as constants.

Character Constants

A character constant of multiple characters enclosed in single quotes, as in 'AB', shall not be used.

String Literals

Unique inline string literals shall be allowed within the source code. String literals which represent program defined strings such as strings sent to a user, or those strings used more than once shall be defined as global strings and referenced via pointer.

Example:

```
char *unknown_command = " is an unknown command, please re-enter";

//
// <Function Prolog>
//              . . .
```

```
//

void process_command(char *command)
{
        . . .
        if (bad_command(command))
        {
                // Notify user that the command entered is unrecognized
                fprintf(stderr, " %s   %s\n", command, unknown_command);
        }
        . . .
}
```

Numeric Literals

Those literals which represent known quantities or program defined quantities shall be declared with the "const" type-specifier. "Casting away the const" shall not be used.

```
        const int index = 1;
        const int *pointer = &index;
        . . .
        (*(int *) pointer)++;           // NOT ALLOWED
```

Literals whose values are represented as hexadecimal (base 16) should use lower case letters when necessary. The prefix appearing in hexadecimal literal shall be "OX", i.e., shall use an upper case X as opposed to a lower case x.

Literals having suffixes shall use only upper case suffixes. This is particularly important for long integers. 1000000L is much more readable than 1000000l.

Example Header File:

```
const float pi = 3.14159;               //  The value of pi.
const int feet_per_mile = 5280;         //  Number of feet in a mile.
const int speed_of_light = 186281;      //  Speed of light (miles per second).
const int bit_mask = OXabc;             //  Hexidecimal constant = 2748.
```

Source example:

```
#include <header files>

const int array_size = 10;              // Size of local array.
static int loc_array[array_size];

void example_function(void)
{
        . . .
        //  Initialize array
        for (index = 0; index < array_size; loc_array[index++] = xx);
        . .
        //  Insure we can handle another user
        if (active_users < max_users)
        {
```

```
                // Process this user
                code
        }
```

Variables

Variables shall be initialized prior to usage.

Structures, Unions and Typedefs

Struct and *union* declarations shall be formatted according to the following guidelines:

- The struct or union declaration must be followed by a tag name, and an opening bracket on one line. No members should be listed following this opening to the definition.
- Components of a struct or union should be listed on separate lines with any comment describing the corresponding comment on the same line. CPPLIB2DOC does not support associating multiple lines of comments with a component of a union or struct.
- Structs and unions can be nested within one another. If one needs to define a nested struct or union (or combination thereof), the same guidelines apply for defining the nested struct/union as for the initial struct, that the keyword (struct or union), a tag name, and opening bracket be found on a separate line, and components of the nested struct be defined on separate line.

An example of the format is provided below:

```
struct funstuff
{
    char         testchar;      // char test

    union testme_out
    {
      int        testint;       // test int 1
      int       *testptr;       // ptr to int
    };

    struct morefun
    {
      int        morefun1;      // level 2 struct
      char       morefunc;      // level 2 char struct
    };
};
```

Classes

General

The class keyword shall start at the beginning of a line. The placement of braces within class definitions shall follow the default style used with compound statements ("Blocks And Statements"). The "private", "protected" and "public" labels shall not be omitted and shall be placed at the same indentation as the class keyword. All member declarations shall be placed one standard indentation past that of the class keyword.

Example:

```
class Complex
{
private:

        double real;
        double imaginary;

public:

        Complex(double real_part, double imaginary_part);
        void add(complex another_complex_number);
};
```

Member Functions

- Member functions and their parameter names shall be descriptive. Parameters shall use the same name as member data but without the underscore if the input parameter will set a member data item. This leads to self-documenting code with the purpose of parameter being obvious. For example,

```
class My_Class
{
private:

        char    *_filename;      // Name of data file.
        char    *_mode;          // Open mode of data file;

public:

        My_Class &My_Class(const My_Class &my_class);   // Copy constructor.

        void set_data(filename, mode)
        {
            // set the member data
            _filename = filename;
            _mode = mode;
        }
};
```

- Member functions that have parameters that are not used shall not list the parameter name in the function declaration, declaring only the parameter type. Some compilers issue warnings against unused parameters. The input list in the function prolog should indicate the parameter is not used. For example,

```
        //
        // FUNCTION     : My_Class::handle_window_interface
        //
        // . . .
        //
        // INPUTS       : XtPointer * - pointer to library window
        //                              structure (not used)
        //
        // RETURNS      : void
```

```
//

void handle_window_interface(XtPointer *);
```

- Simple functions, such as member access functions, in the class definition shall use the "inline" keyword. The usage of "get_" and "set_" prefixes for member access functions is recommended, e.g. get_filename() and set_filename().

Constructors and Destructors

- All class member data shall be set in all constructors. Member data being set from input parameters shall be initialized via a member initialization list. For example,

```
My_Class::My_Class(int first_param, int next_param) :
    _first_param(first_param), _next_param(next_param)
{
    _third_param = NULL;
}
```

- If a constructor or destructor is empty, a indicating comment shall be placed within the code body. For example,

```
My_Class::My_Class(int first_param) : _first_param(first_param)
{
    // Empty
}
```

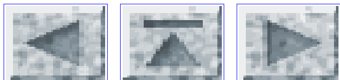
- A destructor is required, even if empty.

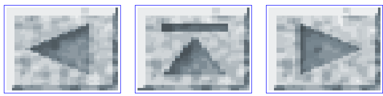
Templates

Declarations for templates of classes and functions should appear on the line immediately preceding the *class* or *function* specification. For example:

```
template<class TYPE>
class Application : public UI_Component
{ ...
```

$Revision: 1.1.1.1 $ - $Date: 1995/08/24 17:15:18 $




C++ LANGUAGE CODING STANDARD

Control Structures

The following subsections define the restrictions and approved usages of C++ control structures.

Functions

All functions (except constructors and destructors) shall include the return type in their declaration. This includes functions which return integers as well as void functions. The function type shall occur on the same line as the function declaration.

Examples:

```
//
// <Function Prolog>
//       . . .
//

static char *build_name(char *directory, char *file_name)
{
        char *full_name;        // Pointer to full pathname to be built
        . . .
        // Build the full pathname, and store in full_name
        return (full_name)
}

//
// <Function Prolog>
//       . . .
//

void destroy_name(char *name)
{
        code    // No return statement because function is void
}
```

Function parameter declarations (including parameter names) shall occur in the function definition. For function declarations with no parameters, "void" shall be used. For example,

```
        void first_function(void);      // correct
        void next_function();           // incorrect
```

Branching

goto statements

The use of "goto" statements shall be strictly limited to time critical code segments where the use of the 'goto' statement will enhance the speed of the code being executed. This usage shall be accompanied by commentary explaining the need for the use.

if ... else statement

"if" statements shall follow the general format defined for compound statements. Even if only a single statement is associated with the "if" or possible corresponding "else", braces ('{' and '}') shall be required. This practice improves readability and ensures that statements which need to be added later are inserted and controlled by the correct condition. Statements associated with the "if" and "else" statements will appear on separate lines indented one level to the right from the "if" and "else".

For compound statements, "else" statements shall begin on a new line at the same indentation as the corresponding "if" statement. The following is formatted properly:

```
if (. . .)
{
        code
}
else
{
        code
}
```

switch - case statement

Switch statements shall conform to all formatting conventions for block statements in general.

"Falling through" a case statement into the next case statement shall be permitted with explicit commentary explaining the reason for the fall through. Every other "case" within a switch shall have a terminating "break" statement.

The "default" statement should be used to handle exceptional or unexpected execution circumstances not otherwise handled in any of the specific "case" statements.

For block comments, 'case' labels shall be indented one extra level, as in:

```
switch ( . . )
{
        case 1:
                code
                break;
        case 2:
```

```
                        code
                        break;
                default:
                        break;
        }
```

Looping -- do, while and for statements

Self contained "for" and "while" statements do not require associated braces. Example: self contained loop statements

```
        // Initialize the array to all 0's
        for ( i = 0; i < MAX_ELEMENTS; array[i++] = 0 );
```

All other statements associated with any "do", "while" or "for" construct shall be enclosed within braces, even if only one statement is associated with the keyword. This practice improves readability and insures that statements which need to be added later are inserted and controlled by the correct condition.

Example: loop statements with one or more associated statements

```
        while ( <condition> )
        {
                <only_one_statement>
        }
```

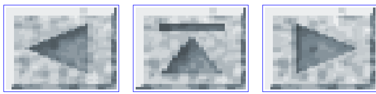
Conditionals

Complex conditional expressions should be split across multiple lines using in-line commentary to decipher the ultimate purpose of the expression.

Example:

```
        if (<condition_1> ||            // explain condition
            <condition_2> ||            // explain condition
            <condition_3> ||            // explain condition
            <condition_4> ||            // explain condition
            . . . .
            <condition_n>)
        {
                <statement_1>
                <statement_2>
                . . . . .
        }
```

$Revision: 1.1.1.1 $ - $Date: 1995/08/24 17:15:18 $



Error Processing

A standard method and format for processing errors which occur during the execution of the software should be defined. The standard format should allow for as much detail as necessary to uniquely identify the source of the error without references to external sources.

General Principles

If there exists a reasonable possibility that a given error may occur, provisions should be made for its interception or handling. Error processing should not cause uncontrolled side-effects. An error caused by erroneous user input should be reported to the user in a manner which fully explains the error.

C++ Language Specific Concerns

The following subsections address the usage of and restrictions upon a number of constructs which apply to the C++ programming language.

Location Of Declarations And C++ Scope Rules

As in many other programming languages, C requires all automatic variables to be declared at the beginning of each function. However, C++ allows variables to be to declared near their first usage. For example, the index of a "for" loop may be declared in the loop. In addition, variables declared within a block are not defined outside that block, i.e., their scope is limited to that block.

Variables should be declared at the beginning of their block.

For example:

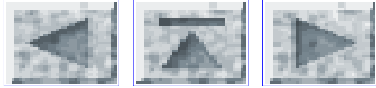
```
Grades *compute_grades(Scores scores, int students, int exams)
{
        . . . .
        for (int outer = 0; outer < students; outer++)
        {
                int sum = 0;
                for (int inner = 0; inner < exams; inner++)
                {
                        . . .
                }
        }
        . . . .
}
```

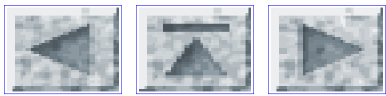
Class Packaging

In general, class declarations are placed in header files. There should be only one class declared in a header file, and the header file should have the same name as the class.

The implementation of the member functions may occur separately, and if so, are placed in one or more source code files.

$Revision: 1.1.1.1 $ - $Date: 1995/08/24 17:15:19 $



 C++ LANGUAGE CODING STANDARD

Program Design Language

When the C++ programming language is used as the Program Design Language (PDL), all of the standards and conventions previously addressed in this document shall remain in effect. In addition, the standards and conventions defined in the following paragraphs shall apply to all usage of the C++ programming language as PDL.

General Description

C++ PDL shall contain prolog and explanatory C++ commentary as described in the sections of this document pertaining to these constructs. It shall be clearly labeled as PDL.

PDL Commentary

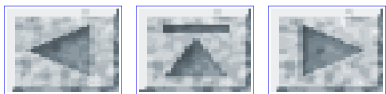
C++ PDL shall contain prolog and explanatory C++ commentary consistent with the rules put forth in this document.

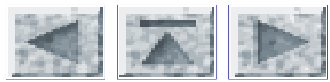
At any point in the PDL process, any C++ function which has been declared, but has not yet been expanded as part of the PDL process shall contain as part of its commentary the line:

```
// Implementation detail - Not further expanded in design
```

This notation shall be deleted when the function has been defined.

$Revision: 1.1.1.1 $ - $Date: 1995/08/24 17:15:19 $





Complete Example

The following sections contain actual C++ source code for a small, but complete example of the standard described in this document. The example consists of two files:

* Ag_File.h is the header file containing a class definition.

* Ag_File.cc is the source code file for the functions in the class.

```
//
// FILE          : Ag_File.h
//
// AUTHOR        : C.L. Zenor,  30 May 1994
//
// "$Header: /home3/source/html/CPP_Pilot/C++_Coding_Standards/CppCodingStd10.html,v
1.1.1.1 1995/08/24 17:15:18 zenor2 Exp $"

#ifndef AG_FILE_H
#define AG_FILE_H

#include
#include
#include "Ag_String.h"



//
// CLASS         : Ag_File
//
// DESCRIPTION   : This class encapsulates some common file management
//                 capabilities.
//
//                 Instantiating an Ag_File object with a disk file name allows
//                 the disk file to be deleted, renamed and copied, and the
//                 status of the disk file to be determined.
//
//                 This class will also open and close file streams for I/O
//                 operations on the physical disk file. An Ag_File object can
//                 be instantiated with a stream mode opening the disk file for
//                 I/O operations.
//

class Ag_File
{

private:

        // Private Member Data

        int     _mode;          // Mode of file stream.
        fstream _stream;        // File stream.
```

```
        // Private Member Functions

        //
        // FUNCTION     : Ag_File::Ag_File
        //
        // DESCRIPTION  : This copy constructor is hidden to prevent its use.
        //
        // INPUTS       : const Ag_File & - not used
        //

        Ag_File(const Ag_File &);

        //
        // FUNCTION     : Ag_File::operator=
        //
        // DESCRIPTION  : This assignment operator function is hidden to
        //                prevent its use.
        //
        // INPUTS       : const Ag_File & - not used
        //
        // RETURNS      : this Ag_File object
        //

        Ag_File &operator=(const Ag_File&);


protected:

        Ag_String       _filename;      // Name of disk file.


public:

        Ag_File(const Ag_String &filename);
        Ag_File(const Ag_String &filename, int mode);

        //
        // FUNCTION     : Ag_File::~Ag_File
        //
        // DESCRIPTION  : Empty destructor.
        //
        // INPUTS       : void
        //

        virtual ~Ag_File(void)
        {
                // Empty
        }


        virtual int open(int mode);
        virtual void cp(const Ag_String &dest);
```

```
        virtual int exists(void);
        virtual int mv(const Ag_String &filename);
        virtual int rm(void);

        void    close(void);
        int     size(void);


        // HEADER :     Member Access Functions

        //
        // FUNCTION     : Ag_File::get_filename
        //
        // DESCRIPTION  : This function is used to access the disk file
        //                name.
        //

        inline Ag_String get_filename(void)
        {
                return _filename;
        }

        //
        // FUNCTION     : Ag_File::get_mode
        //
        // DESCRIPTION  : This function is used to access the file stream
        //                mode. The mode will be 0 if the file is
        //                closed, else the mode the file was opened in
        //                (ios::in, ios::out, ios::app, etc.)
        //

        inline int get_mode(void)
        {
                return _mode;
        }

        //
        // FUNCTION     : Ag_File::get_istream
        //
        // DESCRIPTION  : This function is used to access the file
        //                stream to be used in input operations.
        //

        inline fstream &get_istream(void)
        {
                return _stream;
        }

        //
        // FUNCTION     : Ag_File::get_ostream
        //
        // DESCRIPTION  : This function is used to access the file
        //                stream to be used in output operations.
        //
```

```
        inline fstream& get_ostream(void)
        {
                return _stream;
        }

};

#endif

//
// FILE         : Ag_File.cc
//
// AUTHOR       : C.L. Zenor,  30 May 1994
//

static char rcsid[] = "$Header:
/home3/source/html/CPP_Pilot/C++_Coding_Standards/CppCodingStd10.html,v 1.1.1.1
1995/08/24 17:15:18 zenor2 Exp $";

extern  "C"
{
#include
#include
#include
}
#include

#include "Ag_File.h"



//
// FUNCTION     : Ag_File::Ag_File
//
// DESCRIPTION  : This constructor instantiates an Ag_File object without
//                opening the input file.
//
// INPUTS       : filename - name of disk file
//

Ag_File::Ag_File(const Ag_String &filename)  :
_filename(filename)
{
        _mode = 0;
}



//
// FUNCTION     : Ag_File::Ag_File
//
// DESCRIPTION  : This constructor opens a file stream for the input disk
//                file in the input mode.
//
// INPUTS       : filename - name of disk file
```

```
//              : mode     - open mode of file stream
//

Ag_File::Ag_File(const Ag_String &filename, int mode)  :
_filename(filename),
_mode(mode)
{
        // open a stream for input mode
        _stream.open(_filename, mode);
}



//
// FUNCTION     : Ag_File::open
//
// DESCRIPTION  : This function opens a file stream for a disk file according
//                to the input mode.
//
// INPUTS       : mode - mode for file open
//
// RETURNS      : TRUE  (1) - file opened successfully
//                FALSE (0) - open failed
//

int     Ag_File::open(int mode)
{
        _stream.open(_filename, mode);

        if (_stream.fail())
        {
                return 0;       // return false on any error
        }
        else
        {
                _mode = mode;
                return 1;       // return true if file opened OK
        }
}



//
// FUNCTION     : Ag_File::close
//
// DESCRIPTION  : This function closes an open stream.
//
// INPUTS       : void
//
// RETURNS      : void
//

void    Ag_File::close(void)
{
        // if stream is open then close
        if (_stream)
```

```
        {
                _stream.close();
        }

        _mode = 0;              // reset the open mode
        _stream.clear();
}



//
// FUNCTION     : Ag_File::cp
//
// DESCRIPTION  : This function copies the disk file to the input file.
//
// INPUTS       : dest - name of destination file
//
// RETURNS      : void
//

void    Ag_File::cp(const Ag_String &dest)
{
        // if the file exists copy it byte by byte
        if (exists())
        {
                ifstream from(_filename);
                ofstream to(dest);

                unsigned char byte;
                while (from.get(byte))
                {
                        to.put(byte);
                }
        }
}



//
// FUNCTION     : Ag_File::exists
//
// DESCRIPTION  : This function checks if the disk file exists.
//
// INPUTS       : void
//
// RETURNS      : TRUE  (1) - file exists
//                FALSE (0) - file doesn't exist
//

int     Ag_File::exists(void)
{
        struct stat statbuf;    // file info buffer

        if (stat(_filename, &statbuf) == 0)
        {
                return 1;
```

```
        }
        else
        {
                return 0;
        }
}



//
// FUNCTION     : Ag_File::mv
//
// DESCRIPTION  : This function renames the disk file for this object
//                to the input name.
//
// INPUTS       : filename - new filename
//
// RETURNS      : 0     - file renamed succesfully
//                errno - reason rename failed
//

int     Ag_File::mv(const Ag_String &filename)
{
        if (rename(_filename, filename) == 0)
        {
                // if rename OK, set filename and return true
                _filename = filename;
                return 1;
        }
        else
        {
                // return reason rename failed
                return errno;
        }
}



//
// FUNCTION     : Ag_File::rm
//
// DESCRIPTION  : This function removes the file from disk.
//
// INPUTS       : void
//
// RETURNS      : 0     - file removed succesfully
//                errno - reason remove failed
//

int     Ag_File::rm(void)
{
        int     result;
        if ((result = remove(_filename)) == 0)
        {
                // clean up data if remove OK
                _filename = "";
```

```
                if (_stream)
                {
                        _stream.close();
                }
                _mode = 0;

                return 0;
        }
        else
        {
                // return reason remove failed
                return errno;
        }
}


//
// FUNCTION     : Ag_File::size
//
// DESCRIPTION  : This function returns the size of the disk file.
//
// INPUTS       : void
//
// RETURNS      : size of file if it exists
//                -1 if file doesn't exist
//

int     Ag_File::size(void)
{
        struct stat statbuf;    // file info buffer

        // get file information
        if (!exists())
        {
                return -1;
        }

        // return function result
        stat(_filename, &statbuf);
        return statbuf.st_size;
}
```

$Revision: 1.1.1.1 $ - $Date: 1995/08/24 17:15:18 $