Wildfire C++ Programming Style

With Rationale

by Keith Gabryelski

Wildfire Communications, Inc.
Wildfire: 617-674-1724
Fax: 617-674-1501
Email: ag@wildfire.com

---

1 Introduction

1.1 Background

This document defines the C++ coding style for Wildfire, Inc. It also tries to provide guidelines on how to use the various features found in the C++ language. The establishment of a common style will facilitate understanding and maintaining code developed by more than one programmer as well as making it easier for several people to cooperate in the development of the same program. In addition, following a common programming style will enable the construction of tools that incorporate knowledge of these standards to help in the programming task.

Using a consistent coding style throughout a particular module, package, or project is important because it allows people other than the author to easily understand and (hopefully) maintain the code. Most programming styles are somewhat arbitrary, and this one is no exception. In the places where there were choices to be made, we attempted to include the rationale for our decisions.

This document contains rationale for many of the choices made. Rationale will be presented with this paragraph style.

One more thing to keep in mind is that when modifying an existing source file, the modifications should be coded in the same style as the file being modified. A consistent style is important, even if it isn't the one you usually use.

However, there are many variations in style that do not interfere with achieving these goals. This style guide is intended to be the minimum reasonable set of rules that accomplish these ends. It does not attempt to answer all questions about where ever character should go. We rely upon the good judgement of the programmer as much as possible.

This guide presents things in "programming order", that is, notes, rules, and guidelines about a particular programming construct are grouped together. In addition, the sections are in an order that approximates that used to write programs.

The section Miscellaneous on page 32 contains many useful tidbits of information that didn't fit well into any of the other sections.

Finally, there is a Bibliography and Reading List at the end of this document that contains quite a few titles. Many of the books there should be considered mandatory reading --- if nothing else, buy and read a copy of both the *ARM* [10] by Ellis & Stroustrup and *Effective C++* [12] by Scott Meyers. Coplien's *Advanced C++ Programming Styles and Idioms* [13] is also highly recommended.

2 Fundamental MetaRule

A good style guide can enhance the quality of the code that we write. This style guide tries to present a standard set of methods for achieving that end.

It is, however, the end itself that is important. Deviations from this standard style are acceptable if they enhance readability and code maintainability. Major deviations require a explanatory comment at each point of departure so that later readers will know that you didn't make a mistake, but purposefully are doing a local variation for a good cause.

A good rule of thumb is that 10% of the cost of a project goes into writing code, while more than 50% is spent on maintaining it. Think about the trade-offs between ease-of-programming now vs. ease-of-maintenance for the next 5 to 10 years when you consider the rules presented here.

2.1 C++ is different from C

The C++ programming language differs substantially from the C programming language. In terms of usage, C is more like Pascal than it is like C++. This style guide differs from traditional C style guides in places where the "C mindset" is detrimental to the object-oriented outlook desired for C++ development.

3 Files

Code should compile without errors or warnings. "Compile" in this sense applies to `lint`-like code analyzers, a standard-validating compilers (ANSI-C++, POSIX, Style Guide Verification, etc.), and C++ compilers on all supported hardware/software platforms.

3.1 File Naming Conventions

Try to pick filenames that are meaningful and understandable. File names are not limited to 14 characters. The following table shows the file naming conventions we will use:

File Contents	**Name**
C++ Source Code	*filename*.cc
C++ Header File	*filename*.hh
C Source Code	*filename*.c
C Header File	*filename*.h
Object Code	*filename*.o
Archive Libraries	*filename*.a
Dynamic Shared Libraries	*filename*.so.<ver>
Shell Scripts	*filename*.sh
Yacc/C Source Code	*filename*.y
Yacc/C++ Source Code	*filename*.yy
Lex/C Source Code	*filename*.l
Lex/C++ Source Code	*filename*.ll
Directory Contents	README
Build rules for make	Makefile

POSIX specifies a maximum of 14 characters for filenames, but in practice this limit is too restrictive: source control systems like RCS and SCCS use 2 characters; the IDL compiler generates names with suffixes appended, etc.

3.2 File Organization

- Although there is no maximum length requirement for source files, files with more than about 1000 lines are cumbersome to deal with.
- Lines longer than 80 columns should be avoided. Use C++'s string concatenation to avoid unwieldy string literals and break long statements onto multiple lines. (See Long Lines on page 10):

```
char *s1 = "hello\n"
           "world\n";                    // s1 is exactly the same as s2,
char *s2 = "hello\nworld\n";
```

The line length limit is related to the fact that many printers and terminals are limited to an 80 character line length. Source code that has longer lines will cause either line wrapping or truncation on these devices. Both of these behaviors result in code that is hard to read.

- No `#pragma` directive should be used.

`#pragma` directives are, by definition, non-standard, and can cause unexpected behavior when compiled on other systems. On another system, a `#pragma` might even have the opposite meaning of the intended one.

In some cases `#pragma` is a necessary evil. Some compilers use `#pragma` directives to control template instantiations. In these rare cases the `#pragma` usage should be documented and, if possible, `#ifdef` directives should be to ensure other copilers don't trip over the usage. (See `#error directive` and 4.2 Conditional Compilation (`#if` and its ilk)

3.3 Header File Content

Header files should be functionally organized, with declarations of separate subsystems placed in separate header files. For class definitions, header files should be treated as interface definition files.

- Declare related class and types that are likely to be used together in a single header file.
- If a set of declarations is likely to change when code is ported from one machine to another, put them into a separate header file.
- Never *declare* static variables or non-member static function prototypes in a header file.
- Never *define* variables in a header file.
- Private header files which are used only by a specific implementation should live with that implementation's source code (for example, in the same directory), and be included using the `#include "name"` construct.
- Header files that are designed to be includable by both C and C++ code have different rules. See Interaction with C, § 12.

The required ordering in header files is as follows:

1. A "stand-alone" copyright notice such as that shown below (1):

```
// Copyright 1992 by Wildfire Communications, Inc.
// remainder of Wildfire copyright notice
```

Don't place anything other than the copyright text in this comment --- the whole comment will be replaced programmatically to update the copyright text.

1. An `#ifndef` that checks whether the header file has been previously included, and if it has, ignores the rest of the file. The name of the variable tested looks like `_WF_`*`FILE_HH`*, where "*`FILE_HH`*" is replaced by the header file name, using underscore for any character not legal in an identifier. Immediately after the test, the variable is defined.

```
#ifndef _WF_FILENAME_HH
#define _WF_FILENAME_HH
```

1. A block comment describing the contents of the file. A description of the purpose of the entities in the files is more useful than just a list of class names. Keep the description short and to the point.
2. The RCS `$Header$` variable should be placed as the end of the block comment, or in a comment immediately following it:

```
// $Header$
```

1. #include directives. Every header file should be self-sufficient, including all other header files it needs.

Since implementations will change, code that places "implementation-required #includes" in clients could cause them to become tied to a particular implementation.

The following items are a suggested order. There will be many times where this ordering is inappropriate, and should be changed.

1. `const` declarations.
2. Forward `class`, `struct`, and `union` declarations.
3. `struct` or `union` declarations.
4. `typedef` declarations.
5. `class` declarations.

The rest of these items should be in found this order at the end of the header file.

1. Global variable declarations (not definitions). Of course, global variables should be avoided, and should never be used in interfaces. A class scoped `enum` or `const` can be used to reduce the need for globals; if they are still required they should be either file-scope `static` or declared `extern` in a header file.
2. External declarations of functions implemented in this module.

3. The header guard's `#endif` need be followed by a comment describing the `#ifdef` head guard.
   After all, it is the last directive in the file and should obvious.

3.4 Source File Content

- Do not place the implementation of more than one interface in a single source file. (Classes private to an implementation may be declared and defined within the same source file.)
- The ordering of sections for implementation files is as the same as for header files through step [10], but without the `#ifndef/#endif` multiple inclusion guard (see Template for C++ Implementation files on page 45). After that the order should be:

1. Global scope variable definitions. Global variables (both external and file-static) are problematic in a multi-threaded and/or in a reentrant server context. They should be avoided. (This is also a problem for non-`const` class static member variables.)
2. File scope (`static`) variable definitions.
3. Function definitions. A comment should generally precede each function definition.

4 Preprocessor

- Preprocessor directives must always have the # in column 1. No indentation allowed for preprocessor directives.
- Don't use absolute path names when including header files. Use the form

```
    #include <module/name>
```

to get public header files from a standard place. The `-I` option of the compiler is the best way to handle the pseudo-public "package private" header files used when constructing libraries--- it permits reorganizing the directory structure without altering source files.

4.1 Macros (`#define`)

Macros are almost never necessary in C++.

- The construct `#define NAME value` should never be used. Use a `const` or `enum` instead.

The debugger can deal with them symbolically, while it can't with a `#define`, and their scope is controlled and they only occupy a particular namespace, while `#define` symbols apply everywhere except inside strings.

- Macros in C are frequently used to define "maximum" sizes for things. This results in data structures that impose arbitrary size restrictions on their usage, a particularly insidious source of bugs. Try not to carry forward this limitation into C++.
- Consider using inline functions instead of parametrized macros (but see Use and Misuse of inline, § 11.4 first!).

Macros should be used to hide the `##` or `#param` features of the preprocessor and encapsulate debugging aids such as `assert()`. (Code that uses these features should be rare.) If you find that you must use macros, they must be defined so that they can be used anywhere a statement can. That is, they can not end in a semicolon. To accomplish this, multi-statement macros that cannot use the comma operator should use a `do`/`while` construct:

```
#define      ADD(sys,val)        do { \
                if (!known_##sys(val)) \
                  add_##sys(val);\
              } while(0)
```

This allows `ADD()` to be used anywhere a statement can, even inside an `if`/`else` construct:

```
  if (doAdd)
    ADD(name, "Corwin");
  else
```

```
    somethingElse();
```

It is also robust in the face of a missing semicolon between the `ADD()` and the `else`.

This technique should *not* be used for paired begin/end macros. In other words, if you have macros that bracket an operation, do not put a `do` in the begin macro and its closing `while` in the end macro.

This makes any `break` or `continue` between the begin and end macro invocations relative to the hidden `do/while` loop, not any outer containing loop.

4.2 Conditional Compilation (#if and its ilk)

In general, avoid using `#ifdef`. Modularize your code so that machine dependencies are isolated to different files and beware of hard coding assumptions into your implementation.

- The # of all preprocessor commands must always be in column 1.
- Never use indentation for preprocessor directives.
- If you use `#ifdef` to select among a set of configuration options, you need to add a final `#else` clause containing a `#error` directive so that the compiler will generate an error message if none of the options has been defined:

```
#ifdef sun

#define USE_MOTIF
#define RPC_ONC

#elif hpux

#define USE_OPENLOOK
#define RPC_OSF

#else

#error unknown machine type

#endif
```

- Test for features, not for systems, since features sometimes get added to systems. For example, if you are writing code that deals with networking, you should define and test for macros like `USE_STREAMS` or `USE_SOCKETS`, not for predefined system names like `sun`, `hpux`, `SYSV`, etc., that you happen to "know" support one or the other form.
- Never change the language's syntax via macro substitution. For example, do not do the following:

```
#define      BEGIN    {        // EXTREMELY BAD STYLE!!!
#define      when    break;case        // EXTREMELY BAD STYLE!!!
```

This makes the program unintelligible to all but the perpetrator. C++ is hard enough to read as it is.

- `#else`, `#elif`, and `#endif` should have commented tags identifying the `#if` construct to which it is attached if there are several levels of ifdefs or more than a page worth of code is placed between the `#ifdef` and `#endif`.

```
#ifdef RPC_ONC
    doONCStuff();
#endif
```

- It is considered extremely distasteful, and therefore to be avoided wherever possible, to have a preprocessor conditional that changes the blocking. When you do, the curly-brace rules may be broken:

```
#ifdef DEBUG
    if (!debug)        // #ifdef breaks standard braces rule
#endif
    {
      doSomeStuff();
      doMoreStuff();
    }
```

5 Identifier Naming Conventions

Identifier naming conventions make programs more understandable by making them easier to read. They also give information about the purpose of the identifier. Each subsystem should use the same conventions consistently. For example, if the variable `offset` holds an offset in *bytes* from the beginning of a file cache, the same name should not be used in the same subsystem to denote an offset in *blocks* from the beginning of the file.

We have made an explicit decision to not use Hungarian Notation.(2)

5.1 General Rules

- Identifiers should be meaningful. That is, they should be easy to understand and provide good documentation about themselves. Avoid abbreviations, especially ad hoc ones.

Well chosen names go a long way toward making a program self-documenting. What is an obvious abbreviation to you may be baffling to others, especially in other parts of the world. Abbreviations make it hard for others to remember the spelling of your functions and variables. They also obscure the meaning of the code that uses them.

- Single character variable names should be avoided because of the difficulty of maintaining code that uses them. However, single character names may be appropriate for variables that are essentially meaningless, such as dummy loop counters with short loop bodies or temporary pointer variables with short lifetimes.
- Avoid variables that contain mixtures of the numbers 0 & l and the letters O and 1, because they are hard to tell apart.
- Avoid identifiers that differ only in case, like `foo` and `FOO`. Having a type name and a variable differing in only in case (such as `String string;`) is permitted, but discouraged.

5.2 Identifier Style

Identifiers are either upper caps, mixed case, or lower case. If an identifier is upper caps, word separation in multi-word identifiers is done with an underscore (for example, `RUN_QUICK`). If an identifier is mixed case, it starts with a capital, and word separation is done with caps (for example, `RunQuick`). If an identifier is lower case, words are separated by underscore (for example, `run_quick`). Preprocessor identifiers and template parameters are upper case. The mixed case identifiers are global variables, function names, types (including class names), class data members, enum members. Local variables and class member functions are lower case.

Template parameter names act much like #`define` identifiers over the scope of the template. Making them upper case calls them out so they are readily identifiable in the body of the template.

An initial or trailing underscore should never be used in any user-program identifiers.(3)

Prefixes are given for identifiers with global scope (some packages may extend the prefixes for their identifiers):

Prefix	**Used for**
WF_	preprocessor
_WF_	hidden preprocessor (e.g., protecting symbols for header file)
Wf	Global scope (global variables, functions, type names).
wf	File-static scope

```
----------------------------------------------------------------------
```

File-static identifiers, are the only exception: they are mixed case, but start with a lower-case prefix. (for example, `wfFileStaticVar`).

5.3 Namespace Clashes

The goal of this section is to provide guidance to minimize potential name clashes in C++ programs and libraries.

There are two solution strategies: (1) minimize the number of clashable names, or (2) choose clashable names that minimize the probability of a clash. Strategy (1) is preferable, but clashable names cannot be totally eliminated.

Clashable names include: external variable names, external function names, top-level class names, type names in public header files, class member names in public header files, etc. (Class member names are scoped by the class, but can clash in the scope of a derived class. Explicit scoping can be used to resolve these clashes.)

There are two kinds of name clash problem:

1. Clashes that prevent two code modules from being linked together. This problem affects external variable names, external function names, and top-level class names.
2. Clashes that cause client code to fail to compile. This problem affects type names in public header files, and class member names in public header files. It is most egregious in the case of names that are intended to be private, such as the names of private class members, as a new version of the header file with new private names could cause old client code to break.

Solutions:

- Minimize the number of clashable names by:

1. Avoiding the use of external variables and functions, in favor of class data members and function members.
2. Minimizing the number of top-level classes, by using nested classes.
3. Minimizing the number of private class members declared in public header files. Private class members should be defined in public header files only where clients need to perform implementation inheritance. To minimize the number of dependencies on the data representation, define a single private data member of an opaque pointer type that points to the real data representation whose structure is not published.

- Minimize the likelihood of clashes by use distinctive prefixes in clashable identifiers.

Exception: A top-level class name used only as a naming scope can consist entirely of a distinctive prefix.

```
WfRenderingContext                (a type name)
WfPrint()                    (a function name)
WfSetTopView()                    (a function name)
WfMasterIndex                    (a variable name)
Wf::String                    (a type name --- the class name serves as prefix)
```

For components of the Wildfire program, prefixes begin with `Wf`.

5.4 Reserved Namespaces

Listed below are explicitly reserved names which should not be used in human-written code (it may be permissible for program generators to use some of these).

- From the ANSI C Specification (9899:1990(E)) 7.1.3: "All identifiers that begin with an underscore and either an uppercase letter or another underscore are always reserved".

```
_[A-Z_][0-9A-Za-z_]*
```

- "All identifiers that begin with an underscore are always reserved for use as identifiers with file scope."

```
_[a-z][0-9A-Za-z_]*
```

- The following names are also reserved by ANSI for its future expansion:

```
E[0-9A-Z][0-9A-Za-z]*                      errno values
is[a-z][0-9A-Za-z]*                      Character classification
to[a-z][0-9A-Za-z]*                      Character manipulation
LC_[0-9A-Za-z_]*                      Locale
SIG[_A-Z][0-9A-Za-z_]*                      Signals
str[a-z][0-9A-Za-z_]*                      String manipulation
mem[a-z][0-9A-Za-z_]*                      Memory manipulation
wcs[a-z][0-9A-Za-z_]*                      Wide character manipulation
```

Note that the first three namespaces are hard to avoid. In particular, many accessor methods naturally fall into the `is*` namespace, and error conditions map onto the `E*` namespace. Be aware of these conflicts and make sure that you are not redefining existing identifiers.

6 Using White Space

Blank lines and blank spaces improve readability by offsetting sections of code that are logically related. A blank line should always be used in the following circumstances:

- After the `#include` section.
- When switching from preprocessor directives to code or vice versa.
- Around `class`, `struct`, and `union` declarations.
- Around function definitions.
- Before groups of switch statement case labels that are logically grouped together.

The guidelines for using spaces are:

- A space must follow a keyword whenever anything besides a `;` follows the keyword.
- Spaces may not be used between procedure names and their argument list.

```
  // no space between 'strcmp' and '(',

  // but space between 'if' and '('

  if (strcmp(input_value, "done") == 0)
    return 0;
```

This helps to distinguish keywords from procedure calls.

- Spaces must appear after the commas in argument lists.
- There should be no spaces on either side of `[ ] ( ) . ->`
- `All other binary operators must be separated from their operands by spaces. In other words, spaces should appear around assignment, arithmetic, relational, and logical operators, and they should not appear around .` and `->`.
- Spaces must never separate unary operators such as unary minus, address of, indirection, increment, and decrement from their operands. Some judgment is called for in the case of complex expressions, which may be clearer if the "inner" operators are not surrounded by spaces and the "outer" ones are. Remember that temporary variables are "cheap", and that several simpler expressions may be more understandable than one long complicated one.
- Spaces precede an open brace that shares a line, and follow a closing brace that shares a line.
- The expressions in a `for` statement must be separated by spaces:

```
  for (expr1; expr2; expr3) {
    ...;
  }
```

- If you know you are *constructing* an object with a cast, you should use the function form (for example, `String(sp)`) as a clue to the reader.
- Form-feeds must never be used.
- Using extra white space to line up related things in a set of lines can be worthwhile; such "violations" of the standard do not require the otherwise-mandatory expiatory comment (see Fundamental MetaRule, § 2).

```
  start= (a < b ? a : b);
  end= (a > b ? a : b);
```

6.1 Indentation

Only four-space line indentation should be used. The exact construction of the indentation (spaces only or tabs and spaces) is left unspecified. However, you may not change the settings for hard tabs in order to accomplish this. Hard tabs must be set every 8 spaces.

If this rule was not followed tabs could not be used since they would lack a well-defined meaning.

The rules for how to indent particular language constructs are described in Statements, § 10.

6.2 Long Lines

Occasionally an expression will not fit in the available space in a line; for example, a procedure call with many arguments, or a logical expression with many conditions. Such occurrences are especially likely when blocks are nested deeply or long identifiers are used.

- If a long line needs to be broken up, you need to take care that the continuation is clearly shown. For example, the expression could be broken after the last comma of a function call (never in the middle of a parameter expression), or after the last operator that fits on the line. If they are needed, subsequent continuation lines could be broken in the same manner, and aligned with each other.

```
  if (LongLogicalTest1 || LongLogicalTest2 ||
   LongLogicalTest3) {
    ...;
  }

  a = (long_identifier_term1 --- long_identifier_term2) *
     long_identifier_term3;
```

If there were some correlation among the terms of the expression, it might also be written as:

```
  if (ThisLongExpression < 0 ||
     ThisLongExpression > max_size ||
     ThisLongExpression == SomeOtherLongExpression) {
    ...;
  }
```

Placing the line break after an operator alerts the reader that the expression is continued on the next line. If the break were to be done before the operator, the continuation is less obvious.

Note also that, since temporary variables are cheap (an optimizing compiler will generate similar code whether or not you use them), they can be an alternative to a complicated expression:

```
temp1  = LongLogicalTest1;
temp2  = LongLogicalTest2;
temp3  = LongLogicalTest3;
```

```
if (temp1 || temp2 || temp3) {
    ...;
}
```

6.3 Comments

Comments should be used to give an overview of the code and provide additional information that is not readily understandable from the code itself. Comments should only contain information that is germane to reading and understanding the program.

- In general, avoid including in comments information that is likely to become out-of-date. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment in a source file. Discussion of nontrivial design decisions is appropriate, but avoid duplicating information that is present in (and clear from) the code. It is too easy for such redundant information to get out-of-date.
- C++ style comments (`//`) are preferred over C style (`/*...*/`), though both are permitted.
- Comments should never include special characters, such as form-feed and backspace.
- Frequently there is a need to leave reminders in the code about uncompleted work or special cases that are not handled correctly. These comments should be of the form:

```
//!! When we can, replace this code with a wombat -author
```

This gives maintainers some idea of whom to contact. It also allows one to easily `grep` *the source looking for unfinished areas.*

6.4 Block Comments

Block comments are used to describe a file's contents, a function's behavior, data structures, and algorithms.

- Block comments should be used at the beginning of each file and before each function.
- The comment at the beginning of the file containing `main()` should include a description of what the program does. The comments at the beginning of other files should just describe that file.
- The block comment that precedes each function should document its behavior, input parameters, algorithm, global variables used, and returned value.
- Comments may not have a right-hand line (such as the right edge of a box) drawn with asterisks or other characters.

This would require anyone changing the text in the box to continually deal with keeping the right-hand line straight.

- In many cases, block comments inside a function are appropriate and should be indented at least to the same indentation level as the code that they describe. A block comment should be preceded by a blank line, empty comment lines, or otherwise visually separated from the rest of the code. A separation after the block comment is optional, but be consistent.

```
statements;

// another block comment

// made up of C++ style comments

statements;
/*
 * Here is a C-style block comment
 * that takes up multiple lines.
 */

statements;
```

6.5 Single-Line Comments

Short comments may appear on a single line indented at least to the indentation level of the code that follows.

```
  if (argc > 1) {
    // Get option from the command line.
    ...;
  }
```

6.6 Trailing Comments

Very short comments may appear on the same line as the code they describe, but should be tabbed over far enough to separate them from the statements. Trailing comments are useful for documenting declarations.

- If multiple trailing comments are used in a block of code, they all should be tabbed to the same level.

```
  if (a == 2)
    return WfTrue;            // special case
  else
    return is_prime(a);            // works only for odd a
```

- Avoid the assembly language style of commenting every line of executable code with a trailing comment.

7 Types

7.1 Constants

- Numerical constants must be coded so that they can be changed in exactly one place. The usual method to define constants is to use `const` or `enum`. (See Macros (#define) on page 5.) The `enum` data type is the preferred way to handle situations where a variable takes on only a discrete set of values because of the added type checking done by the compiler:

```
class Foo
{
 public:
    enum {
      Success = 0, Failure = -1
    };
    ...;
}

if (foothing.foo_method("Argument") == Foo::Success) ...
```

- Unlike in ANSI C, integral typed objects in C++ that are declared `const` and initialized with compile-time expressions are themselves compile-time constants. Thus, they can be used as case labels and such.
- Well recognized constants, such as `0`, `1`, and `-1`, can often be used directly. For example if a `for` loop iterates over an array, then it is reasonable to code:

```
  for (i = 0; i < size; i++) {
    // statements using array[i];
  }
```

- Note that `<wfbase.hh>` defines the constants `WfTrue` and `WfFalse`, as well as the type `WfBoolean`, as ensures the constant `NULL` is available.
- Wherever possible, sizes should be expressed in terms of the `sizeof` operator. For example, if an array's size is determined by its initializers, the proper construct for determining the number of elements it has is:

```
double        factors[] = {
          0.1345,
          123.23451,
          0.0
        };

const int        num_factors = sizeof factors / sizeof factors[0];
```

- Wherever possible, `sizeof` operations should be applied to objects, not types. Parentheses are not allowed around the object specifier in a `sizeof` expression.

This means that if the type of an object changes, all the associated `sizeof` operations will continue to be correct. The parentheses are forbidden for data objects so that `sizeof` on types (where the compiler requires them) will be easy to see.

7.2 Use of `const`

Both ANSI C and C++ add a new modifier to declarations, `const`. This modifier declares that the specified object cannot be changed. The compiler can then optimize code, and also warn you if you do something that doesn't match the declaration.

The first example is a ***modifiable pointer to constant integers***: `foo` can be changed, but what it points to cannot be. Use this form for function parameter lists when you accept a pointer to something that you do not intend to change (for example, `strlen(const char *string)`)

```
const int *foo;

foo = &some_constant_integer_variable
```

Next is a ***constant pointer to a modifiable integer***: the pointer cannot be changed (once initialized), but the integer it points to can be changed at will:

```
int *const foo = &some_integer_variable;
```

Finally, we have a ***constant pointer to a constant integer***. Neither the pointer nor the integer it points to can be changed:

```
const int *const foo = &some_const_integer_variable;
```

Note that `const` objects can be assigned to non-`const` objects (thereby making a copy), and the modifiable copy can of course be changed. However, pointers to `const` objects cannot be assigned to pointers to non-`const` objects, although the converse is allowed. Both of these forms of assignments are legal:

```
(const int *)         = (int *);

(int *)          = (int *const);
```

But both of these forms are illegal:

```
(int *)          = (const int *);          // illegal

(int *const)          = (int *);          // illegal
```

When `const` is used in an argument list, it means that the argument will not be modified. This is especially useful when you want to pass an argument by reference, but you don't want the argument to be modified.

```
void
block_move(const void *source, void *destination, size_t length);
```

Here we are explicitly stating that the source data will not be modified, but that the destination data will be modified. (Of course, if the length is 0, then the destination won't actually be modified.)

All of these rules apply to `class` objects as well:

```
class Foo
{
  public:
    void bar1() const;
    void bar2();
};

const Foo *foo_pointer;
foo_pointer->bar1();                    // legal

foo_pointer->bar2();                    // illegal
```

Inside a `const` member function like `bar1()`, the `this` pointer is type `(const Foo *const)`, so you really can't change the object.

However, there is a distinction between ***bit-wise const*** and ***logical const***. A bit-wise const function truly does not modify any bits of data in the object. This is what the compiler enforces for a `const` member function. A logical const function modifies the bits, but not the externally visible state; for example, it may cache a value. To users of a class, it is logical, not bit-wise, const is important. However, the compiler cannot know if a modification is logically const or not.

You get around this by casting away const, for example, by casting the pointer to be a `(Foo *)`. This should only be done if you are absolutely sure that the function remains logically const after your operation, and must always be accompanied by an explanatory comment.

7.3 `struct` and `union` Declarations

A `struct` should only be used for grouping data; there should be no member functions. If you want member functions, you should be using a `class`. Hence, `struct`s should be pretty rare.

- The opening brace should be on the same line as the `struct` or `union` name.
- The closing brace should be on a separate line followed by a semicolon, lining up with the start of the `struct` or `union` keyword.
- Declarations in a `struct` or `union` should be indented one level.

```
struct Foo {
    int    size;        // Measured in inches
    char    *name;        // Label on icon
    ...;
};
```

Note that `struct` and `enum` tag names are valid types in C++, so the following common C idiom is obsoleted because `foo` can be used wherever you used to use `Foo`:

```
      typedef struct foo {                    /* Obsolete C idiom */
          ...;
      } Foo;
```

7.4 enum Declarations

- The `enum` tag and the opening brace should be on the same line as the `enum` keyword.
- The layout for an `enum` is the same as for a `struct` if it takes up multiple lines, or it contains explicit initializers. It also can be contained on one line as shown below.

- The last item in an enum's element list should not be followed by a comma (`,').
- Where possible, the type declaration should occur within the scope of a class instead of polluting the global-scope namespace. (See Namespace Clashes on page 8.) When doing this, references to the constants outside of the class's member functions must be qualified:

```
class Color
{
  public:
    enum Component {
      Red, Green, Blue
    };
};
Color::Component foo = Color::Red;
```

- If your constants define a related set, make them an enumerated type.

```
const int        Red    = 0;              // Bad Form
const int        Blue    = 1;
const ink        Green    = 2;
enum ColorComponent {                          // Much Better
      Red,
      Blue,
      Green
};
enum ColorComponent {                          // Explicit values can be given
      Red    = 0x10,                // to each item as well...
      Blue    = 0x20,
      Green    = 0x40
};
```

This causes `ColorComponent` to become a distinct type that is type-checked by the compiler. Values of type `ColorComponent` will be automatically converted to `int` as needed, but an `int` cannot be changed to a `ColorComponent` without a cast.

- Some compilers can generate a useful warning when confronted with a `switch` statement on an `enum` variable that does not have all elements of the enum expressed as `case` labels. This situation usually indicates a logic error in the code.
- If you need a constant for the number of elements in an `enum`, make the last element of the `enum` be a *`last`* field.

```
enum Color {
      Red,
      Blue,
      Green,
      LastColor = green
};
```

This trick should only be used when you *need* the number of elements, and will only work if none of the enumeration literals are explicitly assigned values.

7.5 Classes

- Only functions should be `public` or `protected`. Member data must always be `private`.
- All inheritance must be `public`. `private` and `protected` inheritance is not allowed.
- It is very important to make sure that your class acts like a black box. The interface you export to clients and subclasses should reflect precisely what they need to know and nothing more. You should ask yourself, for every member function

you export (remember, you're not exporting any `public` or `protected` data members, right?), "Does my client (or subclass) really need to know this, or could I recast the interface to reveal less?"

- Member functions should be declared `const` whenever possible (see Use of const on page 14).

7.6 `class` Declarations

- The opening brace for a `class` should be on a separate line in the same column as the `class` keyword.

This is to help users of `vi`, which has a simple "go to beginning of paragraph" command, and which recognizes such a line as a paragraph beginning. Thus, you can, in the middle of a long class declaration, go to the beginning of the class with a simple command. The usefulness of this feature was deemed to outweigh its inconsistency (also see. section 9.2).

- The closing brace should be on a separate line followed by a semicolon, lining up with the start of the `class` keyword.
- The members of a `class` are indented similarly to those of a `struct` (see struct and union Declarations on page 15).
- The `public`, `protected`, and `private` sections of a `class` should be present (if at all) *in that order*, indented 1/2 an indent level past that of the opening brace.

The ordering is "most public first" so people who only wish to use the class can stop reading when they reach `protected/private`.

- Do not have `public` or `protected` data members --- use `private` data with `public` or `protected` access methods instead.

```
class Foo: public Blah, private Bar
{
  public:
             Foo();               // be sure to use better
            ~Foo();
    int         get_size(int    phase_of_moon) const;              // comments than
these.
    int         set_size(int    new_size);
    virtual int         override_me() = 0;

  protected:

    static int         hidden_get_size();

  private:
    int         Size;               // meaningful comment
    void         utility_method();
};
```

Public and `protected` data members affect all derived classes and violate the basic object oriented philosophy of data hiding.

7.7 Class Constructors and Destructors

Constructors and destructors are used for initializing and destroying objects and for converting an object of one type to another type. There are lots of rules and exceptions to the use of constructors and destructors in C++, and programs that rely heavily on constructors being called implicitly are hard to understand and maintain. Be careful when using this feature of C++!

Be particularly careful when writing constructors that accept only one argument (or use default arguments that may allow a multi-argument constructor to be used as if it did) since such constructors specify a conversion from their argument type to the type of its class. Such constructors need not be called explicitly and can lead to unintended implicit uses of conversions. There are also other difficulties with constructors and destructors being called implicitly by the compiler when initializing references and when copying objects.

Things to do to avoid problems with constructors and destructors:

- When passing objects as parameters to functions you will want to consider passing them by pointer or by reference. If you pass an object by value, a constructor will be called to initialize the formal parameter, which may not be what you want. Similarly, when returning from a function you may wish to return a pointer to the object instead of the object itself. Just be aware of memory "leaks" and object "hygiene" when doing this. (For an in depth exploration of this area, see Items 22 and 23 in *Effective C++* [12].)
- Be careful when copying objects --- unless you redefine the assignment `operator=`, the compiler will perform a member-wise copy, which may not be the behavior expected. Note that initialization and assignment are generally very different operations.
- If you want to make sure that for a given class no member-wise copying is allowed, define a private assignment operator for the class.

This will cause the compiler to generate a compile-time error if a member-wise copy is attempted.

- Study this area carefully. Chapter 12 of the *ARM* [10] is the authoritative reference on the subject, and *Effective C++* [12] tells you many useful things.
- The constructor and destructor declarations line up with the member function names.

```
class Foo
{
  public:
           Foo();
          ~Foo();

    int       get_size(int phase_of_moon) const;

  private:

    ...

};
```

- Constructors invoked by your constructor must be one indentation level in from the constructor declaration. For constructors declared on a single line, the `:` is on the same line as the closing parenthesis. Constructors that take multiple lines to declare have their `:` on the line following the last paramter, indented to the same level as the beginning of the constructor name.

```
BusStop::BusStop() :
    PeopleQueue(30),
    Port("Default")
{
    ...;
}
BusStop::BusStop(char *some_argument) :
    PeopleQueue(30),
    Port(some_argument)
{
    ...;
}
```

- Be careful about `static` initialization. If you design a class that depends on some other facility in its constructor, be careful about order dependencies in `static` initialization. The order in which `static` constructors (that is, the constructors of objects with `static` storage class) get called is *undefined*. You cannot count on one object being initialized before another. Therefore, if you have such a dependency, you must either document that your class cannot be used for `static` objects, or you must use "lazy evaluation" to defer the dependency until later (see Item 47 in *Effective C++* [12] for more details).

7.8 Automatically-Provided Member Functions

C++ automatically provides the following methods for your classes (unless you provide your own):

- a constructor,
- a copy constructor,
- an assignment operator,
- two address-of operators (`const` and non-`const`), and
- a destructor.

```
class Empty { };              // You write this ...
class Empty              // You really get this ...
{
   public:
                 Empty() { }                // constructor
             ~Empty() { }                // destructor
                 Empty(const Empty &rhs);                // copy constructor
        Empty        &operator=(const Empty &rhs);  // assignment operator
        Empty        *operator&();                // address-of
    const Empty            *operator&() const;                //      operators
};
```

Every class writer must consider whether the default functions are correct for that class. If they are, a comment must be provided where the function would be declared so that a reader of the class knows that the issue was considered, not forgotten.

If a class has no valid meaning for these functions, you should declare an implementation in the `private` section of the class. Such a function should probably call `abort()`, throw an exception, or otherwise generate a visible runtime error.

This ensures that the compiler will not use the default implementations, that it will not allow users to invoke that function, and that if a member function uses it by accident, it may at least be caught at runtime.

It is a good idea to always define a constructor, copy constructor, and a destructor for every class you write, even if they don't do anything.

7.9 Function Overloading

Overloading function names must only be done if the functions do essentially the same thing. If they do not, they must not share a name. Declarations of overloaded functions should be grouped together.

7.10 Operator Overloading

Deciding when to overload operators requires careful thought. Operator overloading does not simply create a short-hand for an operation --- it creates a set of expectations in the mind of the reader, and inherits precedence from the language.

- You should only use an operator shorthand if the logical meaning of applying the operator on the type(s) involved is intuitive, either because of common usage (for example, `+` on strings concatenates, `<<` adds to a stream) or real algebra on the types (for example, a position class plus an offset gets a different position).
- If you overload one operator of a logically connected set, you *must* overload the rest of the set, if for no other reason than to generate an error if the others are called when they are not meaningful. Overloading `<` without overloading `>` or `>=` will astonish the user in unhappy ways, as will overloading `+` and `=` but not `+=` . In particular, `->` `.` and `[]` should always be considered a set:

```
foo->member()            // should be identical to
```

```
(*foo).member()          // which should also be identical to
foo[0].member()
```

Overloading `==` *requires* overloading `!=`, and vice versa.

If the expression `(a != b)` is not equivalent to `!(a == b)` we have unacceptably astonished the user.

- Note that while you can overload operators, you *cannot* change the language's precedence rules.
- If an operator in a set does not make sense, you must override it in the `private` section so that the compiler will report the error to anyone who assumes that the set is complete. However, this should be a flag for you to consider whether the operator overloading really is natural --- the strong presumption is that you are not going to override *all* members in the set then *none* of the members of the set should be overridden.
- Use type-cast operators selectively. Like so many C++ features, type casting can either clarify or obscure your code. If a type cast seems "natural", like the conversion between floating point and integers, then providing a cast function seems like a good idea. If the conversion is unusual or nonsensical, then the existence of a cast function can make it very hard to figure out what's going on. In the latter case, you should define a conversion function that must be called explicitly. If you provide a type-cast operator, you must provide an equivalent conversion function as well.

This allows the user of the class to determine if a cast is more readable than a member function invocation, for example, to avoid casts that look like they should be automatically done by the compiler, but are explicit to invoke the cast.

7.11 Protected items

When a member of a class is declared `protected`, to clients of the class it is as if the member were private. Subclasses, however, can access the member as if it were declared private to them. This means that a subclass can access the member, but only as one of its own private fields. Specifically, it cannot access a protected field of its parent class via a pointer to the parent class, only via a pointer to itself (or a descendant).

7.12 friends

When using friends remember that private member access rights do not extend to subclasses of the `friend` class. Any method that depends on `friend` access to another class cannot be rewritten in a subclass.

- When applied to a class (friend Classes, § 7.12.1, `class Base`, below), the friend keyword denotes a class-global behavior change that is being applied to the `friend` class. As such, it is not governed by the class part designation (`public`, `protected`, or `private`) currently in force. Thus, the `friend` keyword should be indented to the same level as these class part names.
- In all other cases where the friend keyword is used, (see `friend int operator==`, section 7.12.2) it should be treated as a type modifier in the same sense that `static`, `extern`, and `virtual` are. That is, the word `friend` is lined up along with the other type specifiers one indent level from the level of the class itself.
- If `friend` is needed between classes, `friend` member functions are preferred to making the entire class a friend.

The use of `friend` class or method declarations is discouraged, since the use of `friend` breaks the separation between the interface and the implementation. The only non-discouraged uses are for binary operators and for cooperating classes that need private communication, such as container/iterator sets.

7.12.1 `friend` Classes

- All `friend` class declarations must come at the end of the class declaration.
- If a `friend` class declaration is necessary and the `friend` class is intended to be subclassable, the `friend` class must be written so that its subclasses have the same access rights as the base class. To do this, any access depending on the `friend` declaration is encapsulated in a protected function:

```
class Secret
{
  private:
    int    Data;
    int    method();

  friend        Base;
};

class Base
{
  protected:
    int    secret_data(Secret *income_info);
    int    secret_method(Secret *income_info);
};

int
Base::secret_data(Secret *income_info)
{
    return income_info->Data;
}

int
Base::seccet_method(Secret *income_info)
{
    return income_info->method();
}
```

Methods of the `Secret` class should not be accessed directly by methods of the `friend` class `Base`. Direct access makes it hard to cut-and-paste code from the base to a derived class:

```
void
Base::an_example(Secret *income_info)
{
    int  a = income_info->Data;                    // BAD:  Direct access is wrong
    int  b = secret_data(income_info);                    // GOOD: Use accessor
functions!
}
```

7.12.2 friend Methods

Binary operators, except assignment operators, must almost always be `friend` member functions.

```
class String
{
  public:
            String(const char *);

  friend int operator==(const String &,const String &);
  friend int operator!=(const String &,const String &)
            { return !(string1 == string2); }
};
```

If the `operator==` were a member function, the conversion operator would only allow `(String == char *)` but not `(char * == String)` This would be quite surprising to the user of the class. Making `operator==` a `friend` member function allows the conversion implied by the constructor to work on both sides of the operator.

7.13 Templates

- The template specifier for a template class should be placed alone on the line preceding the "class" keyword or the return type for a function. The following header for the template definition should be indented 1/2 indent level:
  As an example:

```
  template<class TYPE>
  class List
  {
    public:

    TYPE     front();
    ...
  };

template<class TYPE>
TYPE
List<TYPE>::front()
{
    ...;
}
```

- The names of general template parameters should be simple and all-purpose, since their types are normally not known. On the other hand, specific template parameters should be given meaningful names to show their purpose. For example:

```
  template<class TYPE, unsigned int SIZE>
  class Vector
  {
    private:

    Type     Data[SIZE];
  }
```

Here, the type stored by the `Vector` template class is named `TYPE` because it is a general purpose parameter. The `SIZE` parameter, however, is specific since it ultimately determines the size of a `Vector<TYPE>` object; its name reflects this specific purpose.

8 Variables

8.1 Placement of Declarations

Since C++ gives the programmer the freedom to place a variable definition wherever a statement can appear, they should be placed near their use. For efficiency, it may be desirable to invoke constructors only when necessary. Thus function code may define some local variables, do some parameter checking, and once the sanity checks have passed then define the class instances and invoke their constructors.

Where possible, you should initialize variables when they are defined.

```
char      *Foo[]             = { "Hello", ", ", "World\n" };

int      max_string_length            = BUFSIZE;

String      path("/usr/tmp/gogin");
```

This minimizes "used before initialization" bugs.

8.2 `extern` Declarations

- Do not explicitly declare variables, types, or functions that you are not implementing. Include the appropriate public header files instead.
- If you are using a class, but all you need is the type name (for pointers or references), you should use the simple forward declaration instead of including the header file (if you can):

```
class ClassName;
```

- External declarations should only be placed in header files and should begin in column 1. A comment describing the role of the identifier being declared should be included.

Place them in header files to prevent inconsistent declarations in each source file that uses it.

8.3 Indentation of Variables

- The type names should be at the current indentation level.
- Type modifiers such as `*` and & should be with the identifier, not the type. The following style is forbidden:

```
int*      ip;
String&      str;
```

This style, though currently popular, lies about syntax, since `int* p1, p2;` implies `p1` and `p2` are both pointers, but one is not. Since we do not accept that only one variable should be declared on a line as a fixed rule, we cannot allow a style that lies about the meaning of multiple declarations on a line.

- Variable definitions should be indented to align the variables being declared, with identifiers lining up with each other exclusive of preceding modifiers (`*`, &, etc.). (4)

```
int       count             = 0;

char      **pointer_to_string            = &foo;
```

8.4 Number of Variables per Line

- One variable per line is recommended since it encourages commenting. In other words,

```
int      level    = 0;          // indentation level
int      size    = 0;          // size of symbol table
int      lines    = 0;          // lines read from input
```

is preferred over:

```
int      level, size, lines;             // Not Recommended
```

The latter style is frequently used for declaring several temporary variables of primitive types such as `int` or `char`, or strongly matched variables, such as x, y pairs, where changing the type of one requires changing the type of all.

- Variables and functions must not be declared on the same line:

```
long        db, OpenDb();            // Bad
long        db;          // Better
long        OpenDb();            // but still not recommended
```

```
#include <admintools/database.hh>                    // Best

Databae        db;
```

You should use a header file that contains an external function declaration of `OpenDb()` *instead of hard-coding its definition in your source file.*

8.5 Definitions Hiding Other Definitions

- Avoid local variable definitions that override (hide) variables defined at higher levels.

```
void
WfFunction()
{
    static int      boggle_count;           // Count of boggles in formyls

    if (condition) {
      int    boggle_count;           // Bad --- this hides the above instance
    }
}
```

8.6 Initialized Variables

- Opening braces on initializers must follow the = on the same line.
- If the initializers fit on one line, the closing brace should also be on that line.
- If the initializers don't fit comfortably on one line, they should be placed on separate lines, indented one level from the variable name. In this case, the closing brace should be outdented one level from the initializer list.

This style is purposefully analogous to the function declaration style. It may look strange to some at first, but in the context of a complete program, it lends itself to an overall pleasing appearance of the code.

- Initializer lists must always have the optional trailing comma.

```
Cat      cats[] = {
        "Shamus",
        "Macka",
        "Tigger",
        "Xenephon",
      };
```

- Initialized objects that require only one initializer should not use braces.

```
char     *name = "Framus";
```

9 Functions

9.1 Function Declarations

- Function declarations should be lined up in accordance with Indentation of Variables, § 8.3 above.
- Function parameters should be listed as many per line as reasonable. Indention for new lines should occur at the open `(`.

```
SomeType        *WfLibraryFunctionName(void *current_pointer,
                                       size_t desired_new_size);
```

However, if a function takes only a few parameters, the declaration can be strung onto one line (if it fits):

```
int    strcmp(const char *s1, const char *s2);
```

We usually use a one-line-per-declaration form for several reasons.
(1) It is easy to comment the individual parameters,
(2) It makes it easier to read when there many parameters.
(3) It is easy to reorder the parameters, or to add one. The closing `);` is on a line by itself to make it easier to add a new parameter at the end of the parameter list.
(4) It is designed to be visually similar to the other declaration statements.
(5) It works well with long identifier names.
However, with simple declarations the weight seems too great for the benefit.

- If the function takes no parameters, both the opening and closing parenthesis must be on the same line.

```
int      getchar();
```

The ANSI C-compatible construct of `(void)` for a function with no parameters must only be used in header files designed to be included by both C and C++ (See Interaction with C on page 39.)

- Function parameter names must be included in the function declaration, not just the parameter types. This applies as well for usages where a function prototype is being used as a type (in other words, a `typedef` type). The only exception is for operators and single-argument constructors where the meaning of the parameter is clear from that context.

This provides internal documentation that can help people remember what a parameter is supposed to represent. It also allows comments in the file to refer to the parameter by name.

- Input-only function parameters must either be passed by value or as a `const &`.
- Values that may be modified by a function (input/output or output-only parameters) should be passed as references to the thing that will be modified. (This closely resembles the Pascal `var` parameter) The alternative of passing pointers is not encouraged, but is not prohibited. See References vs. Pointers, § 11.5 for more details.

9.2 Function Definitions

- Function bodies should be small.

Small functions promote clarity and correctness.

- Each function definition should be preceded by a block comment that gives its name and a short description of what the function does.
- The full type of the value returned should be alone on a line in column 1 (`int` must be specified explicitly). If the function does not return a value then it should be given the return type `void`. If the value returned requires a long explanation, it should be given in the block comment above. The function name should be alone on a line beginning in column 1 (the class name is included on the same line as the function name if the function is a method of a class).

```
char *
WfString::cstr()
{
    // ...
}
```

- Parameter declarations are analogous to those in Function Declarations, § 9.1The opening brace of the function body will be alone on a line beginning in column 1.
- In the case of a function that has unused parameters, it may be useful to comment out the name of the unused parameter in order to suppress compiler warnings. Except for callback routines, this usually is a questionable situation.

```
void
WfFoo(int param1, int /* optional_param2 */)
{
    // ...;
}
```

- All local declarations and code within the function body are indented by one indentation level.

```
int
SystemInformationObject::get_number_of_users(Name host_name, Time idle_time)
{
    int    some_variable;

    statements;
    ...;
}
```

- Never use the `this` variable in member functions to access members. In other words, you should never write `this->Anything`.

10 Statements

- Each line must contain at most one statement. In particular, do not use the comma operator to group multiple statements on one line, or to avoid using braces. For example:

```
  argv++; argc--;                         // Multiple statements are bad

  if (err)

    fprintf(stderr, "error\n"), exit(1);                      // Using `,' is worse



argv++;                                // The right way

argc--;

if (err) {

    fprintf(stderr, "error\n");

    exit(1);

}
```

10.1 Compound Statements

Compound statements are statements that contain lists of statements enclosed in braces.

- The enclosed list must be indented one more level than the compound statement itself. The opening left brace must be at the end of the line beginning the compound statement and the closing right brace must be alone on a line, positioned under the beginning of the compound statement (see example below).
- The left brace that begins a function body, a `class` definition, or a new scope are the only occurrences of a left brace that should be alone on a line.

```
{
    // New Block Scope
    int some_variable;

    statements;
}
```

- Braces are always used around any multi-line statement when it is part of a control structure, such as an `if`/`else` or `for` statement, as in:

```
if (condition) {                // braces required; following "if" is two lines
    if (other_condition)            // braces not required -- only one line follows
      statement;
}
```

Braces are not required for control structures with single-line bodies, except for `do`/`while` loops, whose always require enclosing braces. This single-line rule includes a full `if`/`else`/`else`/... statement:

```
if (condition)
    single_thing();
else if (other_condition)
    other_thing();
else
    final_thing();
```

Note that this is a "single-line rule", not a "single statement rule". It applies to things that fit on a single line.

Single-statement bodies are too simple to be worth the weight of the extra curlies.

10.2 `if/else` Statements

An `else` clause is joined to any preceding close curly brace that is part of its `if`. See also Comparing against Zero on page 34.

```
if (condition) {
    ...;
}
if (condition) {

    ...;

} else {

    ...;

}
if (condition) {

    ...;

} else if (condition) {

    ...;

} else {

    ...;

}
```

10.3 for Statements

```
  for (initialization; condition; update) {

    ...;

  }
```

If the three parts of the control structure of a for statement do not fit on one line, they each should be placed on separate lines or broken out of the loop:

```
for   (longinitialization;

      longcondition;

      longupdate

) {

    ...;

}
longinitialization;                  // Alternate form...

for (; condition; update) {

    ...;

}
```

When using the comma operator in the initialization or update clauses of a `for` statement, no more than two variables should be updated. If there are more, it is better to use separate statements outside the `for` loop (for the initialization clause), or at the end of the loop (for the update clause).

10.4 do Statements

```
  do {

    ...;

  } while (condition);
```

10.5 while Statements

```
  while (condition) {

    ...;

  }
```

10.6 Infinite Loops

The infinite loop is written using a `for` loop:

```
  for (;;) {

    ...;

  }
```

This form is better than the functionally equivalent `while (TRUE)` or `while (1)` since they imply a test against `TRUE` (or `1`), which is neither necessary nor meaningful (if `TRUE` ever is not true, then we are all in real trouble).

10.7 Empty Loops

Loops that have no body must use the `continue` keyword to make it obvious that this was intentional.

```
while (*string_pointer++ != '\0')
    continue;
```

10.8 switch Statements

- `case` labels should be on lines separate from the statements they control.
- `case` labels are indented to 1/2 an indent level beyond the level of the `switch` statement itself.

We use this indentation since the labels are conceptually part of the `switch`, but indenting by a full indent would mean that all code would be indented by two indent levels, which would be too much.

- A blank line must appear before the first `case` label in a set of `case` labels, especially if the body code is large. (But don't put a blank line right after the `switch` keyword)
- The last `break` in the `switch` is, strictly speaking, redundant, but it is required nonetheless.

This prevents a fall-through error if another `case` is added after the last one.

- In general, the fall-through feature of the `switch` statement should rarely, if ever, be used (except for multiple case labels as shown in the example). If it is used otherwise, it must be commented with:

```
    // FALLTHROUGH
```

where the `break` would normally be expected.

This makes it clear to the reader that it is this fallthrough was intentional.

- A `return` statement should not be followed by a `break`.
- `switch` statements that use members of an `enum` should not have a `default` case. This means that if you have such a `switch`, you must always have all members of the `enum` represented in explicit `case` labels, even if these only execute a `break`.

Some C++ compilers will warn you if such a `switch` is missing a member. This warning will call out situations where you add a member to an `enum` definition but forget to add a `case` for it in a given `switch`. This is usually an error.

- `switch` statements keyed on non-`enum` values should have a `default` label if the code assumes that only certain values will arrive. Such a `default` label should make sure that the erroneous situation is called to someone's attention, such as by signalling an error or generating an error message.

```
  switch (pixel_color) {

    case Color::blue:

    ...;

    break;



    case Color::red:

    found_red_one = TRUE;

    // FALLTHROUGH

    case Color::purple:

   {

      int    local_variable;

      ...;

      break;

   }



    default:          // handles green, mauve, and pink colors...

    ...;

    break;
  }
```

This is to catch unexpected inputs in more graceful ways than failing unpredictably somewhere else in the code.

10.9 `goto` Statements

While not completely avoidable, use of `goto` is discouraged. In many cases, breaking a procedure into smaller pieces, or using a different language construct will enable elimination of a `goto`.

The main place where a `goto` can be usefully employed is to break out of several nested levels of `switch`, `for`, or `while` nesting when an error is detected. In future versions of C++ exceptions should be used.

```
    for (...) {

      for (...) {

        ...;

        if (disaster) {

          goto error;

        }

      }

    }

    return true;

error:        // clean up the mess
```

- Never use a `goto` to branch to a label within a block:

```
    if (pool.is_empty()) {

      goto label;        // VERY WRONG

    }

    for (...) {

      Object obj;

label:

    }
```

Branching into a block may bypass the constructor invocations and initializations for the automatic variables in the block. Some compilers treat this as an error, others blissfully ignore it.

- When a `goto` is necessary, the accompanying label must be alone on a line starting in column 1.

10.10 `return` Statements

The expressions associated with `return` statements are not required to be enclosed in parentheses.

10.11 `try/catch` Statements

The proposed C++ syntax for exception handling is *not* to be used in shared code *at this time* (5). This section specifies the syntax which will eventually be used to support the feature, but should be avoided in near term code.

We need a section describing an alternate way of handling exceptions so that we can use "boilerplate" code to do the right thing now and help ease the transition to exceptions in the future.

- The expressions associated with `throw` statements are not required to be enclosed in parentheses.
- Signaling an exception in a destructor is not a good idea, since any destructive behavior that has already taken place probably cannot be reversed.

```
try {
  statements;
} catch (type) {
  statements;
} catch (...) {                // This is the literal "..."
  statements;
}
```

11 Miscellaneous

11.1 General Comments & Rules

- When incrementally modifying existing code, follow the style of the code you are modifying, not your favorite style. Nothing is harder to read than code where the personal style changes from line to line.
- Don't use global data. Consider using file- or class-static data members instead.
- File static variables are more appropriate than class-static variables, since they hide more of the class's implementation from the reader of the header file. Of course, if you class implementation does not fit within one file, this technique will not be usable.
- In library code, don't use global or static objects that require constructors.

This can used to support C programs being linked to C++ libraries without the use of a C++-aware linker. See Interaction with C on page 39.

- Don't use global (nonmember) functions when implementing classes --- use private member functions instead (except binary operators --- see friend Methods, § 7.12.2.)
- It's possible to partially circumvent the strong type checking C++ imposes on function arguments by using unspecified (or `<stdargs.h>`) parameters. You should avoid doing this if at all possible. The classic example of this usage is:

```
void printf(const char *, ...);
```

- Do not use a "type field" in a class when a virtual function can do the job. However, if you need to be able to narrow the type of a superclass to its subclass then a type field is appropriate.
- In expressions involving mixed operators, use parentheses to ensure desired results and to enhance clarity. Overuse of parentheses tends to result in code that is difficult to read --- too few parentheses can result in expressions that are hard to modify correctly.
- Try to make the structure of your program match the intent. For example, replace:

```
if (BooleanExpression)

    return WfTrue;

else

    return WfFalse;
```

with:

```
return BooleanExpression;
```

Similarly,

```
if (condition)                    // Awkward

    return x;

return y;
```

is usually clearer when written as:

```
if (condition)                    // Clear

    return x;

else

    return y;
```

or

```
return (condition ? x : y);
```

- Do not use the assignment operator in a place where it could be easily confused with the equality operator. For instance, in the simple expression

```
if (x = y) {                    // Confusing

    ...;

}
```

it is hard to tell whether the programmer really meant assignment or the equality test.
Instead, use

```
if ((x = y) != 0) {                    // Understandable

    ...;

}
```

or something similar if the assignment is needed within the `if` statement. There is a time and a place for embedded assignments. The `++` and `--` operators count as assignments. So, for many purposes, do functions with side effects.

- Do not use embedded assignments in an attempt to improve run-time performance --- this is the job of the compiler.

Note also that side effects within expressions can result in code whose semantics are compiler-dependent, since the order of evaluation is explicitly undefined in most places. Compilers do differ.

As an aside, many of today's compilers can produce faster and/or smaller code if you don't use embedded assignments. If you are using such convoluted code to "help the compiler optimize the program", you may be doing more harm than good.

- Become familiar with existing library classes. You should not be writing your own string compare routine, or defining your own `memcpy()` function. Not only does this waste your time, but it may prevent your program from taking advantage of any hardware specific assists or other means of improving performance of these routines. It also makes your code less readable, because the reader has to figure out whether you are doing something special in the re-implemented routines to justify their existence.

11.2 Limits on numeric precision

- C++ is a superset of ANSI-C in most respects; specifically it shares the ANSI specs on the C's built in types. This is *all* you can safely assume:

Type	Minimum Value	Maximum Value	Comments
signed char	--128	127	They may hold more
unsigned char	0	255	They may hold more
char	0	127	Can't assume signed or unsigned
short	--32,768	32,767	Minimum 16 bits
signed short			
unsigned short	0	65,535	Minimum 16 bits
long	--2,147,483,648	2,147,483,647	Minimum 32 bits
signed long			
unsigned long	0	4,294,967,295	Minimum 32 bits
int	--32,768	32,767	Same as a short
signed int			
unsigned int	0	65,535	Same as an unsigned short

- A `char` may be `unsigned` or `signed`. You can't assume either. Thus, only use (unmodified) `char` if you don't care about sign extension and can live with values in the range of 0-127.
- An `int` cannot be counted on to hold more than a `short int`. It is an appropriate type to use if a `short` would be big enough but you would like to use the processor's "natural" word size to improve efficiency (on some machines, a 32-bit operation is more efficient than a 16-bit operation because there is no need to do masking). If you need something larger than a `short`, you must specify a `long` --- an `int` won't do.
- Always use the right system-defined types for values: know and use `size_t`, `ptrdiff_t`, `sigatomic_t` where appropriate.

11.3 Comparing against Zero

Comparisons against zero values must be driven by the type of the expression. This section shows the valid ways to compare for given types. Anything not permitted here is forbidden.

When maintaining code it is very useful to be able to tell what "units" a comparison is using. As an example, an equality test against the constant `0` implies that the variable being tested is an integral type; testing against an explicit `NULL` implies a pointer comparison, while an implied `NULL` implies a boolean relationship.

(See if/else Statements on page 28)

11.3.1 Boolean

Choose variable names that make sense when used as a "condition". For example,

```
if (pool.is_empty())
```

makes sense, while

```
if (pool.state())
```

just confuses people. The generic form for Boolean comparisons are

```
if (boolean_variable)

if (!boolean_variable)
```

Note that this is the only case where an implicit test is allowed; all other comparisons (`int`, `char`, pointers, etc.) must explicitly compare against a value of their type. A standalone variable should always imply a boolean value.

Never use the boolean negation operator `!` with non-boolean expressions. In particular, never use it to test for a null pointer or to test for success of the `strcmp()` function:

```
if (!strcmp(s1, s2))                     // Bad

if (strcmp(s1, s2) == 0)                    // Good
```

11.3.2 Character

```
if (char_variable != '\0')

while (*char_pointer != '\0')
```

11.3.3 Integral

```
if (integer_variable == 0)

if (integer_variable != 0)
```

11.3.4 Floating Point

```
if (floating_point_variable > 0.0)
```

Always exercise care when comparing floating point values. It is generally not a good idea to use equality or inequality type comparisons. Use relative comparisons, possibly bounded by a "fuzz" factor in cases where an equality-like functionality is required.

11.3.5 Pointer

```
if (pointer_variable != NULL    )                  // Always use an explicit test vs.
NULL
```

Implicit comparisons are not allowed:

```
if (pointer_variable)                      // WRONG
```

11.4 Use and Misuse of `inline`

- inline functions should not be used indiscriminately.
- Never use `inline` functions in public interface definitions.

Since a client using your inlined interface actually compiles your code into their executable, you can never change this part of your implementation. And no one else can provide an alternate implementation.

- Some C++ compilers support a `+w` option which will warn you of the case where things declared `inline` aren't inlined. When an `inline` function isn't inlined, it may be defined "file static" in every file that references it!

Within your implementation there may be places where you need to use inlines. Be aware that the use of inlines can easily make your (and other people's) code larger, which can overcome any efficiency gains. Here are some guidelines to help do it right.

- As explained in Chapter 7 of the ARM, inlining is not a panacea and in general should be done after the program is written, debugged and instrumented.
- Small simple functions that only increment or return a value are usually good candidates for inlining. For most functions, the time spent in a call is dominated by the time it takes to execute the body of the function and not by the cost of calling it. Indiscriminate use of inlining results in larger programs that can take longer to execute due to a larger working set that needs to reside in memory. Unless you know for sure that inlining a particular function is a win, do not use inlining.
- If your inline function just calls something else that isn't inline, that's fine, as long as the other function has identical semantics. As an example, you might have a class that defines a virtual function `is_equal()`, which compares two objects for equality. It also has an inline definition for `operator==`, as a notational convenience. Since `operator==` just turns around and calls the `is_equal()` function, it may be OK for it to be inline and not virtual.
- Do not use inlines just because your function just happens to have a one-line implementation.
- Use an inline function if efficiency is very, very important and you'll never change it.
- If you don't know (and can't prove) that your implementation must be inline, don't make it inline. Build it normally and then measure the performance. Experience has shown again and again that programmers spend lots of time optimizing code that hardly ever gets executed, while totally missing the real bottlenecks. The empirical approach is much more reliable. Experience has also shown that a better algorithm or smarter data structures will buy you a lot more performance than code tweaking.
- Long inline functions should be declared simply as `inline` in the class, with the code presented immediately after the class declaration:

```
class Dummy
{
public:
    inline int        do_something();
};

inline int
Dummy::do_something()
{
    // ... several lines of code
}
```

11.5 References vs. Pointers

The advantages of using references over pointers include (from [25]):

- A reference can only refer to an object. Its just another name for an existing object. Therefore, people reading the code should be able to recognize immediately the intent of the programmer --- namely to refer locally to some object via this one local name.
- In some situations using multiple inheritance, a reference may be somewhat more efficient.
- A reference is always "const", in the sense that the reference cannot be re-assigned to refer to another object within the

scope of the lifetime of that reference. Thus both the original programmer and the subsequent code reader can assume a powerful invariance across the scope: this reference refers to some object, and it *only* refers to that object.

The advantages of using pointers over references include:

- Pointers can do and mean many things that references can't --- a pointer can be used to represent such things as:
- an array
- an element of an array
- null (no object passed)
- one past the end of an array (an end marker)
  So clearly if you must do one of these things you should be using a pointer. The disadvantage then being that *which* of *N* possible uses of the pointer is intended is not immediately apparent from the code.
- Pointers (non-constant) can be re-assigned, making them useful in the infrequent case where it would truly be inefficient or inconvenient to not be able to change the pointers value. The disadvantage then is that the code reader cannot assume invariance within the scope.

Use references where you reasonably can --- that is, when assigning a name to an already existing singular object. Use pointers for any of the other *N* meanings that pointers have traditionally held.

11.6 Portability

The advantages of portable code are well known and little appreciated. This section gives some guidelines for writing portable code, where the definition of portable is a source file that can be compiled and executed on different machines with the only source change being the inclusion of (possibly different) header files.

- Beware of making assumptions about the size of pointers. They are not always the same size as `int`. Nor are all pointers always the same size, or freely interchangeable.
- Also, beware of potential pointer alignment problems. On machines that have address alignment restrictions (for example, Sparc), the conversion of a pointer-to-`char` to a pointer-to-`int` may result in an invalid address.
- Never assume you can do anything with a `NULL` pointer except test its value. In particular, code that assumes that dereferencing a `NULL` pointer yields `'\0'` (ala VAX/BSD) will generate memory faults on other machines (for example, Sparc). Further, never write a class that assumes that `this` may be validly `NULL`.
- Don't assume that longs, floats, doubles, or long doubles can be at any even address.
- Don't assume you know the memory layout of a data type.
- Don't assume you know how a `struct` or `class` is laid out in memory, or that it can be written to a data file as is.
- *Never* assume that the rest of the world uses 7-bit US-ASCII.
- Watch out for signed characters. On some machines, for example, `char` is sign-extended when used in expressions, which is not the case on some other machines. Code that assumes either that `char` is `signed` or `unsigned` is non-portable. It is best to completely avoid using `char` to hold numbers. Manipulation of characters as if they were numbers is often non-portable. Explicitly declare character variables as `signed` or `unsigned` in cases where it matters.
- Bitfields are also `signed` on some machines and `unsigned` on others. If you use bitfields in a way that is sensitive to this difference you must be explicit.
- On some processors the bits (or bytes) are numbered from right to left within a word. Other machines number the bits from left to right. Hence any code that depends on the left-right orientation of bits in a word deserves special scrutiny. Bit fields within structure members will only be portable so long as two separate fields are never concatenated and treated as a unit.
- Alignment considerations and loader peculiarities make it very rash to assume that two consecutively declared variables are together in memory, or that a variable of one type is aligned appropriately to be used as another type.
- You should not need to know the format of the virtual tables generated by the compiler. These formats may not even be portable between different versions of the same compiler.
- If a simple integer, such as a loop counter, is being used where either 16 or 32 bits will do, then use `int`, since it will get the most efficient (natural) unit for the current machine. Word size also affects shifts and masks. The statement

```
x &= 0177770
```

will clear only the three right most bits of a 16 bit `int` on a PDP-11. On a VAX (with 32 bit `ints`) it will also clear the entire upper 16 bits. Use

```
x &= ~07
```

instead, which works as expected on all machines. The operator `|` does not have these problems, nor do bit-fields.

- The directive `#include "somefile.hh"` implies different search paths on different systems. On BSD derived systems it means

1. first look in the directory the build is being done in,
2. then use the `-I` directory list
   while on System V derived systems it means
3. first look in the directory the current file is in,
4. then the directory the build is being done in,
5. then the `-I` directory list
   Note that the BSD systems don't implicitly look in the directory that contains the source file if that directory isn't the one where the build is being done.
   As an example, assume a directory with the files `foo.cc`, `foo.hh`, and the subdirectory `obj`. Also assume that `foo.cc` does an `#include "foo.hh"`. The command

```
        CC -c -o obj/foo.o foo.cc
```

will work on both systems, but

```
        cd obj

        CC -c -o foo.o ../foo.cc
```

will fail to find `foo.hh` on many BSD based systems.

- Some things are inherently non-portable. Examples include code to deal with particular hardware registers such as the program status word, and code that is designed to support a particular piece of hardware such as a device driver. Even in these cases there are many routines and data structures that can be made machine-independent. Source files should be organized so that the machine-independent code and the machine-dependent code are in separate files. Then if the program is to be moved to a new machine, it is a much easier task to determine what needs to be changed.

12 Interaction with C

12.1 ANSI-C/C++ include files:

This is the list of the header files that ANSI-C (and thus C++) requires be provided by the language implementation. Use of any other "system" header file may not be portable.

Uses of these C header files are not required to be bracketed with the `extern "C" { }` construct.

```
#include <stddef.h>
#include <stdio.h>
#include <stdarg.h>
#include <stdlib.h>
#include <locale.h>
#include <ctype.h>
#include <string.h>
#include <time.h>
#include <limits.h>
```

```
#include <errno.h>
#include <assert.h>
#include <signal.h>
#include <setjmp.h>
#include <math.h>
#include <float.h>
```

12.2 Including C++ Header Files in C programs

Header files that must be included by both C and C++ source have slightly different rules than do C++-only header files.

- The files are named `headerfile.h` instead of C++'s `headerfile.hh`
- All C++ keywords must be avoided. The C++ keywords that are not in C are: asm, `catch`, `class`, `delete`, `friend`, `inline`, `new`, `operator`, `private`, `protected`, `public`, `template`, `this`, `throw`, `try`, `virtual`.
- Comments must use C's `/* */`, not C++'s `//`.
- Functions that take no parameters must use `(void)`, not just `()`.
- Function prototypes must always be provided.

This project has no interest in any non-ANSI-C dialects of the C language.

- The C language mapping for identifiers must be preserved (See Template for Shared C and C++ Header Files on page 44).

12.3 Including C Header Files in C++

The inclusion of every non-C++ header file must be surrounded by the extern "C" { } construct.

Note that the header files enumerated above (ANSI-C/C++ include files:, § 12.1) are considered C++ header files, and not subject to this rule.

```
extern "C" {
#include <somefile.h>
#include <otherfile.h>
}
```

12.4 C Code calling C++ Libraries

Function calls that are intended to be called from C that take input-only `struct` arguments may wish to use pointers, since C does not have references. Such pointers must, of course, be declared `const`.

In order to be able to export a C++ library to C users, you'd like to let the C users link to the library using the regular C/Unix linker.

The ***CC linker*** (also called the "C++ pre-linker" or "patch" or "munch") is the part of the C++ system that makes static constructors and static destructors work. These are the routines that get called if you have a global (or a local static) variable whose class has a constructor. The C++ pre-linker paws through your object files looking for the right pattern of mangled name that indicates constructors and/or destructors that need to be called for that file, puts them all together into an initialization routine named `_main()`, and links that synthesized `_main()` into your program. Cfront has inserted a call to `_main()` at the start of your main C++ program.

So, on SunOS 4.x, if your library has any global, file-static or local-static variables whose classes have constructors or destructors, you must use the `CC` command to link any application to that library.

SunOS 5.0 object files allow libraries to have a `.init` section that gets called to initialize the library. The constructors would be put into this section (and not `_main()`), thus avoiding the linking problems mentioned above.

Footnotes

(1)

The real, legal copyright text is still being created. It will be much longer and wordier than that which is shown here. A tool exists that allows us to simply put a comment containing the word "Copyright" somewhere in it at the beginning of the file and have the correct current copyright notice inserted automatically at a later date.

(2)

Hungarian notation is that style that has a identifier modifier (either suffix or prefix) for every potentially interesting fact about that identifier, for example, a "P" suffix for pointer, "l" prefix for local, "i" prefix for an int, "a" suffix for arrays, so each identifier can look something like liindexaP.

(3)

This does not apply to names used in libraries or generated by utilities.

(4)

Lining up the initializations, as shown in the examples, is a matter of personal preference.

(5)

As soon as exceptions are available in the compilers we use, this restriction will be revisited.
