C/C++ Programming Style Guidelines

Fred Richards

Style guidelines and programming practices for C/C++ code for Dynamic Software Solutions. Use the checklist at the end of this document prior to submitting code for peer review.

“ *De gustibus non est disputandum.* ”

1. Introduction

This document contains the guidelines for writing C/C++ code for Dynamic Software Solutions. The point of a style guide is to greater uniformity in the appearance of source code. The benefit is enhanced readability and hence maintainability for the code. Wherever possible, we adopt stylistic conventions that have been proved to contribute positively to readability and/or maintainability.

Before code can be considered for peer review the author must check that it adheres to these guidelines. This may be considered a prerequisite for the review process. A checklist is provided at the end of this document to aid in validating the source code’s style. Where code fails to adhere to the conventions prescribed here may be considered a defect during the review process.

If you have not already, please study *Code Complete* by Steve McConnell. This book provides a detailed discussion on all things related to building software systems. It also includes references to statistical studies on many of the stylistic elements that affect program maintainability. Another valuable source of solid programming practice tips is *The Practice of Programming* by Brian W. Kernighan and Rob Pike. Scott Meyers’

books, *Effective C++* and *More Effective C++* should be considered required reading for any C++ programmer.

And what person would be considered complete without having read *The Elements of Style* by Strunk and White?

2. File Contents

Use files to group functionality. Each file should contain only one cohesive set of functions. Avoid duplicating functionality in separate files. If different files contain similar functions, consider generalizing the function sufficiently and putting it into its own file so that both function groups can use the one source. For C++ code, put only one class or closely related set of classes in each file.

Avoid strong coupling between functions and classes implemented in separate files. If two objects are so strongly coupled that one can only be used in conjunction with the other then they belong in the same file.

Use header files (`.h` suffix) to declare public interfaces, use code files (`.c`, `.cc` or `.cpp` suffix) to define implementations. Typically each cohesive set of functions you write in a single file will have one accompanying header/interface file pair. Code that uses your implementation will `#include` the header file.

Be precise with `#include` statements. Explicitly include the `.h` files you require, and only where you require them. If, for example, your code calls a function defined externally, include that function's associated `.h` in your implementation file not in your code's associated `.h` file. You should only need to include other files in your `.h` file if your public function interface or data type definitions require the definitions contained therein.

Avoid using header files to contain a set of `#include` directives simply for convenience. This "nesting" of `#include` constructs obscures file dependencies from the reader. It also creates a coupling between modules including the top-level header file. Unless the modules are cohesively coupled functionally, and each requires *all* the `.h` files included in the convenience header, it is preferable to instead include all the

individual `.h` files everywhere they are required.

2.1. Header (Interface) File Content

Header files should contain the following items in the given order.

1. Copyright statement comment
2. Module abstract comment
3. Revision-string comment
4. Multiple inclusion `#ifdef` (a.k.a. "include guard")
5. Other preprocessor directives, `#include` and `#define`
6. C/C++ `#ifdef`
7. Data type definitions (classes and structures)
8. `typedefs`
9. Function declarations
10. C/C++ `#endif`
11. Multiple inclusion `#endif`

Example 1. Standard (C) header file layout

```
/*
 * Copyright (c) 1999 Fred C. Richards.
 * All rights reserved.
 *
 * Module for computing basic statistical measures on
 * an array of real values.
 *
 * $Id$
 */
```

```
#ifndef STATISTICS_H
#define STATISTICS_H

#include <math.h>
#include <values.h>

#define MAXCOMPLEX { MAXINT, MAXINT }

#ifdef _cplusplus
extern "C" {
#endif

struct complex {
    int r; /* real part */
    int i; /* imaginary part */
};
typedef struct complex Complex;

     ...

/*
 * Compute the average of a given set.
 *   Input - array of real values, array length.
 *   Output - average, 0 for empty array.
 */
float
ave(float* v, unsigned long length);

 ...

#ifdef _cplusplus
}
#endif

#endif /* STATUS_H */
```

2.2. Code Files

C and C++ code files follow a similar structure to the header files. These files should contain the following information in the given order.

1. Copyright statement comment
2. Module abstract comment
3. Preprocessor directives, `#include` and `#define`
4. Revision-string variable
5. Other module-specific variable definitions
6. Local function interface prototypes
7. Class/function definitions

Unlike in the header file, the implementation-file revision string should be stored as a program variable rather than in a comment. This way **ident** will be able to identify the source version from the compiled object file. For C files use:

```
static const char rcs_id[] __attribute__ ((unused)) =
"$Id$";
```

The `__attribute__` modifier is a GNU C feature that keeps the compiler from complaining about the unused variable. This may be omitted for non-GNU projects. For C++ files, use the following form for the revision string:

```
namespace { const char rcs_id[] = "$Id$"; }
```

Precede each function or class method implementation with a form-feed character (Ctrl-L) so that when printed the function starts at the start of a new page.

Example 2. Standard (C++) implementation/code file

```
//
// Copyright (c) 1999 Fred C. Richards.
// All rights reserved.
//
// Module for computing basic statistical measures on
// an array of real values.
//

#include "Class.h"
#include <string>

namespace {
    const char rcs_id[] = "$Id$";
}

// Utility for prompting user for input.
string
get_user_response();

^L
Class::Class(const int len)
{
    private_array_ = new[len];
}

Class::~Class()
{
    delete private_array_;
}

^L
 ...
```

3. File Format

The formatting style presented here is essentially that used by Stroustrup in *The C++ Programming Language*. If you use Emacs you can make this your default editing mode by adding the following to your `.emacs` file:

```
(defun my-c-mode-common-hook ()
  (c-set-style "stroustrup"))

(add-hook 'c-mode-common-hook 'my-c-mode-common-hook)
```

Format your code so that the spatial structure illustrates the logical structure. Use blank lines to help separate different ideas, use indentation to show logical relationships, and use spaces to separate functionality. Each block of code should do exactly one thing.

Start all function definitions and declarations in column zero. Put the return value type, the function interface signature (name and argument list), and the function body open bracket each on a separate line. For functions that are more than a few lines long, put the function name after the closing bracket in a comment.

Example 3. Formatting function declarations and definitions

```
void
debug(const string& message);

int
Class::method(const int x, const string& str)
{
    .
    .
    .

} // method
```

Use a single space to separate all operators from their operands. The exceptions to this rule are the "`->`", "`.`", "`()`" and "`[]`" operators. Leave no space between these operators and their operands. When breaking operations across lines, put the operator at the end of the broken line rather than at the start of the continuation line.

Use four spaces for each level of indentation. Avoid making lines longer than 80 characters. When breaking lines, use the natural logical breaks to determine where the newline goes. Indent the continuation line to illustrate its logical relationship to the rest of the code in the line. For functions, for example, this means aligning arguments with the opening parenthesis of the argument list.

Example 4. Breaking statements across multiple lines

```
new_shape = affine_transform(coords, translation,
                             rotation);

if ( ( (new_shape.x > left_border) &&
       (new_shape.x < right_border) ) &&
     ( (new_shape.y > bottom_border) &&
       (new_shape.y < top_border) ) )
{
    draw(new_shape);
}
```

Use a pure-block, fully bracketed style for blocks of code. This means put brackets around all conditional code blocks, even one-line blocks, and put the opening bracket at the end of the line with the opening statement. The exception to this rule is for conditions that are broken across multiple lines. In this case put the open bracket on a line by itself aligned with the start of the opening statement (as shown above).

Example 5. Fully bracketed, pure block style

```
if (value < max)  {
    if (value != 0)  {
        func(value);
    }
```

```
} else {
    error("The value is too big.");
}
```

Although the brackets may seem tedious for one-line blocks, they greatly reduce the probability of errors being introduced when the block is expanded later in the code's life.

3.1. Unique to C++

Start `public`, `protected`, `private`, and `friend` labels in column zero of class declarations. Use explicit `public` labels for all `struct` public fields and use explicit `private` labels for all private class members.

The members of a class should be declared in the following order. Declare all public data members and type definitions first. Declare private or protected data members or type definitions used in function member initialization lists or inline implementations next. Declare all public member functions next, starting with the constructors and destructor. Declare all remaining private or protected data members and type definitions next. Declare all private or protected function members next. Declare all friends last.

Put simple inline function definitions on the same line as their declaration. For inline functions spanning multiple lines, use a pure-block style with four-space indentation. In general, avoid putting complex function implementations `.h` files.

Example 6. Class declaration format

```
class Type : public Parent {
private:
    int x_;
    int y_;
public:
    Type();
    Type(int x) : x_(x) { }
    ~Type();
```

```
    int get_x() const { return x_; }
    void set_x(const int new_x) { x_ = new_x; }
    ...
    void display() {
        ...
    }
}
```

4. Choosing Meaningful Names

4.1. Variable Names

The name formatting conventions described here are essentially the GNU coding standards. These are available online using `info`.

Use lower case for all variable names. For multi-word names, use an underscore as the separator. Use all capitals for the names of constants (i.e. variables declared `const` and enumerated types). Use an underscore as a word separator.

Choose variable names carefully. While studies show that the choice of variable names has a strong influence on the time required to debug code, there are unfortunately no clear and fixed rules for how to choose good names. Review Chapter 9 of *Code Complete* periodically. In the mean time, here are some general guidelines to follow:

- Be consistent! The most important thing is to establish a clear, easily recognizable pattern to your code so that others will be able to understand your implementation and intent as quickly and reliably as possible.
- Use similar names for similar data types, dissimilar names for dissimilar types.

- Avoid names that are homophones: e.g., `foo`, `fu`, `phoo`, etc. Also, don't rely on capitalization to distinguish between variables.
- Use names that say *what the variable represents* rather than how it is used (i.e. use *nouns* for variable names); use terminology from the application domain and avoid computer jargon that reflects programming details.
- Avoid generic names such as `tmp`, `buf`, `reg`.
- Avoid intentionally misspelled words such as `lo` or `lite`.

In general, short names are acceptable for variables that serve a short-lived purpose or that have a common usage in C/C++ (e.g., index variables called `i`, `j`, `k`, etc.). Being concise can contribute to the readability of code. However, for variables that serve a unique and important purpose, or variables that persist over a significant region of your code, use descriptive and complete names. Studies have shown that minimal debugging time correlates with average variable name lengths of 10-16 characters.

4.2. Function Names

Use lower-case letters for public function names. Use an underscore as a word separator.

For functions that return no values (i.e. return type `void`), use strong verbs that indicate the function's purpose. Typically you will want to include the object of the verb in the name. For example,

```
void
remove_dc_offset(short *signal, const unsigned long length);

void
set_output_gain(const float gain);
```

Because functions tend to serve a more complex purpose than variables, longer names are more acceptable.

If a function returns a value it is sometimes better to use a name that indicates the meaning of the value returned. For instance,

```
/*
 * Compute the DC offset of the given signal.
 */
float
dc_offset(const short * const signal,
          const unsigned long length);

/*
 * Poll the D/A and return the current gain setting.
 */
float
gain(void);
```

In general, be consistent and be informative. Choose names that make your code easy to read and understand.

4.3. Classes, Structures and Type Definitions

The name formatting conventions described here are essentially those used by Stroustrup in his book on C++.

Capitalize the first letter of the name of each data type that you define. This includes all `struct`, `class`, `typedef` and `enum` types. Use an underscore as a word separator, just as for C variables and function names.

For class *instance* variables, start all names with lower-case letters. Again, use an underscore as a word separator. Apply the same rules to `public` and `protected` members, both variables and functions. Add a trailing underscore to `private` member names.

Example 7. Capitalization of user-defined types

```
/* Straight C */

struct complex {
    int r;    /* real */
    int i;    /* imaginary */
};
typedef struct complex Complex;

// C++ interface example

class Canvas {
public:
    enum Pen_style {
        NONE = 0,
        PENCIL,
        BRUSH,
        BUCKET
    };

    Canvas();
    ~Canvas();

    void set_pen_style(Pen_style p);
    ...
private:
    int cached_x_;    // to avoid recomputing coordinates
    int cached_y_;
};

// C++ usage example

Canvas sketch_pad;

sketch_pad.set_pen_style(Canvas::BRUSH);
```

When working with C++ classes and objects be mindful of redundant name elements. Remember that class members are identified by their class instance name. Thus you do not have to repeat information about the class in the member element's names.

Example 8. Poor variable names

```
// Notice how redundant "stack" becomes.

template <Type>
class Stack {
public:
    int stack_size;
    add_item_to_stack(Type item);
    ...
};

Stack my_stack;

my_stack.add_item_to_stack(4);
int tmp = my_stack.stack_size;
```

5. Comments

In general, well written code should document itself. Clear, concise variable and function names, consistent formatting and spatial structure, and clean syntactical structure all contribute to readable code. Occasionally, however, complex logic will benefit from explicit description. Be careful not to use comments to compensate for poorly written code. If you find that your code requires many comments or is often difficult to describe, perhaps you should be rewriting the code to make it simpler and clearer.

5.1. Style

For straight C code, use `/* ... */` style comments. For C++ code, use `// ...` style comments. Adhering to these conventions, you can quickly estimate the number of lines of comments in your code with the following commands:

```
% grep "^[ \t]*\* "
% grep "^[ \t]*\/\/"
```

Too few or too many comments is an indicator of code that is likely to be difficult to maintain.

Avoid the use of end-line comments except for variable declarations and for marking `#if/#endif` statements. Make comments be the only thing on a line. For longer comments describing more complex logic, use a *block* style to offset them from the code better. Use block-style comments to describe functions. Use *bold* comments to delimit major sections of your code file. Preface all bold comments and block comments that introduce functions with a form-feed character so that they appear at the start of the printed page. The following example shows the various comment types in the C style.

Example 9. C comment types

```
^L
/*
 * ************************************************
 * Bold comment.
 * ************************************************
 */

/*
 * Block comment.
 */

/* Short (single-line) comment. */
```

```
int i; /* end-line comment */
```

5.2. Content

End-line comments are acceptable for describing variable declarations. Use a comment to describe any variable whose purpose is not obvious from its name.

Use comments to document your intent. Do not describe *how* your code works, that should be obvious from the implementation. Instead describe *why* your code does what it does. Avoid explaining especially tricky code in comments. Instead, rewrite the code to make it intrinsically more obvious. Use complete sentences with proper spelling and punctuation in all comments.

Write your comments before and as you write your code, not after. If you start by writing comments you give yourself a low-level design for the implementation. When you are finished testing your code, go back and review all comments to make sure they are still accurate.

Comment things that have wide impact. If a function makes assumptions about the condition of variable on input, document that. If a required speed optimization makes the code difficult to read, explain the need for the code in a comment. If your code uses or changes any global variables, comment that.

Use *bold* comments to delimit major sections in your code file. You may, for instance, implement a number of private utility functions. Use a bold comment to mark the start of that code. Preface each function with a *block* comment describing the function's purpose, the meaning of any input variables, and the significance of any return value(s). There is no need to include the function name since it immediately follows the comment.

Example 10. Commenting functions and function groups

^L

```
/*
 * ************************************
 * Statistics utilities used to
 * optimize performance on the fly.
 * ************************************
 */

/*
 * Compute the standard deviation or "variance"
 * for a set.
 *
 * Input: v - set of values and set size.
 *        len - size of input set.
 * Output: Dev = Expect (x - x_ave)^2
 *         0 for the empty set
 */
static float
std_dev(const float *v, const unsigned long len)
{
    ...
}
```

Use an end-line comment to mark the end of particularly long blocks of code. This applies to functions and conditional code. Include the control condition that terminates control for `if/else` branches and `for/while/do` loops.

Use an end-line comment also to identify which `#if` or `#ifdef` statement a particular `#endif` statement closes. Include the condition in the comment for blocks of code spanning more than a few lines. When using `#else` conditions, mark both the `#else` and the `#endif` statements with the negated condition (i.e. preface it with “not”).

Example 11. Commenting long code blocks

```
#ifdef DEBUG
    .
    .
```

```
    .
#else // not DEBUG
void
function()
{

    if (position != END)

        .
        .
        .

    } // position != END

    .
    .
    .

} // function()

#endif // not DEBUG
```

6. Syntax and Language Issues

The following sections outline some general practices of good defensive programming. Always assume that others will have to read and maintain your code, and try to assist them as you write. Also, assume that errors and defects are inevitable, and write so as to isolate them and limit their effect as quickly as possible. This latter practices is sometimes referred to as "fire-walling" code. Be liberal in checking the validity of input arguments within functions, and always check values returned by functions you call.

6.1. General

Avoid putting multiple instructions on the same line. Each line should do exactly one thing. This is applies in particular to control statements for branch and loop structures. Consider the following:

```
/* Bad practice! */

if (!eof && ((count = get_more()) > min_required) {
    ...
}
```

This should be rewritten so that the act of getting more data is separate from the task of checking that more data remains to be processed:

```
/* Safer version */

if (!eof) {
    count = get_more();
    if (count > min_required) {
        ...
    }
}
```

Avoid the use of side-effects. The following innocuous line may actually produce different depending on which compiler is used:

```
a[i] = i++;    /* a[0] or a[1] == 1 ? */
```

Again, each line should contain a single statement, and each statement should do exactly one thing.

Avoid type casts and never cast pointers to `void*`. One of the strengths of C++ is its strong typing and ability to support arbitrary user types. If you feel the need to cast data to other types in C++, perhaps you should be considering defining an inheritance relationship or using templates.

Do not define any types from pointers (e.g., `typedef char* String`).

Avoid using preprocessor constants (i.e. `#defines`). Instead declare variables of the appropriate C/C++ type as `const` and use them. For related sets of integer constants, define an `enum`. Both of these techniques let the compiler perform type checking where the preprocessor `#defines` would not.

Limit variable scope as much as possible. In C++, use brackets to group functionality and temporary variables. Declare a variable just prior to using it and destroy it when you are finished with it.

Use parentheses to group logic in branch and loop control structures. Most people are not intimately familiar with operator precedence, and parentheses can make logic operations much easier for people to parse and understand. Without the additional parentheses it is not obvious that the first case below differs from the other three (which are equivalent) for only one of the eight combinations of the booleans `x`, `y` and `z`:

Example 12. One of these things is not like the other

```
(x || y  && y || z)

(x && y || z)
( (x && y) || z )
( (x || z) && (y || z) )
```

6.2. Structured Programming

Keep the structure of your code as clear as possible. Do not call **`exit()`** from library functions. Instead `return` with an appropriate error condition. Call `return` only once

for each function longer than a few lines. Avoid using `break` and `continue` to escape loop and branch code. Consider instead adding or changing the exit conditions of the the control statement. Do not use `goto`.

Prefer using `if/else/else/...` over the `switch/case/case/...` with non-trivial branch conditions. For both constructs use default conditions only to detect legitimate defaults, or to generate an error condition when there is no default behavior. Using a `switch/case` block with overlapping conditions only when the cases have identical code so that fall-through is obvious.

Prefer using `while() { ... }` instead of `do { ... } while();`. It is easier for humans to parse the control structure if they know the exit condition upon entering the block of code. The `do { ... } while();` form buries the exit criterion at the end of the loop.

Avoid overly long control structures. If you find loop or branch constructs spanning several printed pages or screens, consider rewriting the structure or creating a new function. At the very least place a comment at the end of the structure to indicate the exit conditions.

Avoid deeply nested code. Humans have a hard time keeping track of more than three or four things at a time. Try to avoid code structure that requires more than three or four levels of indentation as a general rule. Again, consider creating a new function if you have too many embedded levels of logic in your code.

Avoid the use of global variables. They make your code hard to support in a multi-threaded environment. If you do use global variables, understand how they affect the ability of your module to be reentrant.

6.3. Functions and Error Checking

Do not use preprocessor function macros. There are too many possible problems associated with them and modern computer speeds and compiler optimizations obviate any benefit they once may have had. Define a function instead.

Write function declarations/prototypes for all functions and put them either in the

module `.h` file for public functions or at the start of the `.c` file for private, internal functions. The function declaration list should read like a table of contents for your code.

Make explicit all assumptions about the condition of input data to your routines. Use assertions to test for programming errors, use exceptions (C++) or return values (C) to report error conditions detected in normal use. Do not put any implementation logic in your assertions since often they will not remain in the deployed code (especially library functions). When a library function must report both a computed value and a distinct error value, pass the computed value through a variable and `return` the error value.

Check the return values of all library function calls. This is especially important for functions providing access to system resources (e.g., **`malloc()`**, **`fopen()`**, etc.).

Generate informative error messages. Write messages that are understandable to the user. Be complete and concise and avoid using computer jargon. Suggest possible causes of the error condition.

7. Conclusion

The guidelines presented here should be followed when writing all new code. When working with someone else's code it is more important that you adhere to the conventions used there. Only if the coding style is ad hoc should you impose your own (or these) conventions.

A checklist is included at the end of this document. It covers most of the items discussed. Apply the checklist to your code prior to submitting it for peer review. Any unaddressed issue is considered a coding defect during the review process.

Obviously a style guide cannot dictate good programming practices. It can only ward off some of the more flagrant problems. With luck, a good style guide can encourage better programming habits. To that end, all C++ programmers should read *Effective C++*, by Scott Meyers. It covers far more concerns and goes into far greater detail on C++ than is appropriate here. It is critical for C++ programmers to understand these

issues when writing new code.

Appendix A. Review Checklist

File contents.

- Do all files contain:
  - a copyright statement?
  - an abstract/synopsis comment?
  - a revision string?
- Do all header files contain a multiple include `#ifdef`?
- Are all necessary `#includes` made explicitly (i.e. the code does not rely on nested `#includes`)?
- Are all public functions *declared* in the module's `.h` file?
- Do function declarations/prototypes exist for all functions?
- Does each code file contain exactly one cohesive set of classes or functions?
- Are functions from different files sufficiently uncoupled from one another?

File format.

- Do all function declarations and definitions begin in column zero? Are the return type, function name and open bracket each on a separate line, each beginning in column zero?
- Do functions longer than a typical screen/page have comments with their name at the close bracket?

- Is four-space indentation used throughout?
- Are all control structures in a pure-block style and fully bracketed?
- Is there a single space surrounding all operators, except `.`, `->`, `[]` and `()`?
- Do the C++ keywords `public`, `private`, `protected`, and `friend` all start in column zero?
- Are C++ class internals declared in the proper order?

    1. public data and types
    2. private or protected data and types used in the class declaration
    3. public member functions, starting with constructors and the destructor
    4. other private or protected data members
    5. other private or protected functions
    6. friends

Variable and function names.

- Are all C variable and function names lower case, with an underscore as a word separator?
- Are all C++ variable and function names lower case, with capitalization rather than an underscore indicating word boundaries?
- Do all private class member names end with an underscore?
- Do all programmer-defined types/classes start with a capital letter?
- Are all constants and enumerated types all capital letters?
- Are all variable names sufficiently informative and meaningful given their scope?
- Do variable names match the problem domain?
- Are variable names nouns?

- Are function names strong verbs (or nouns for functions whose sole purpose is to compute a value)?

Comments.

- Are bold comments used to divide code into major sections? Are block comments used to mark significant points? Are end-line comments used only for variable declarations and to mark long blocks of code?
- Does C++ code use `// ...` style comments (not `/* ... */`)?
- Do all comments contain complete sentences, with proper punctuation and spelling?
- Do comments describe intent rather than implementation details?
- Is all subtle code sufficiently explained in comments?
- Do all but the simplest functions have comments describing what they do, what data they operate on and any impact they have on the rest of the application?

Language usage.

- General
  - Does each line of code do exactly one thing?
  - Are all constants declared as `const` and not as `#defines`?
  - Does the code avoid casting variables and return values to different data types? Are there no casts to `void*`?
  - Are no `typedefs` made from pointers (e.g., `typedef char* Str`?)
  - Are there no preprocessor macros defined?
  - Are parentheses used to group items in all but the simplest logic constructs?
  - Are all `private` class members explicitly declared such?
- Program structure

- Does each function call `return` from only one place?
- Is **`exit()`** called only from within **`main()`**, and only once?
- Do final `else` blocks of `if/else` branches and `default` blocks of `switch/case` branches handle default conditions or error conditions only?
- Do all overlapping `switch/case` conditions (i.e. fall-throughs) use identical code?
- Has the use of global variables been avoided?
- Do most control structures span no more than a page or two? Is the close bracket of all longer controls structures commented with the exit criteria for the block of code?
- Does nested conditional code go no more than three or four levels?
- Has the use of structure-breaking directives such as `goto`, `continue` and `break` been avoided?

- Functions and error-checking
  - Are functions used always instead of preprocessor macros?
  - Are all assumptions about the condition of input data tested explicitly with **`assert()`**?
  - Will the code perform the same way if assertions are removed?
  - Do library functions return error values or throw exceptions (C++) wherever possible?
  - Are all function return values tested or exceptions caught?
  - Are all error messages informative to a typical user? Are messages complete sentences, with proper punctuation and spelling?

References

The C++ Programming Language, Bjarne Stroustrup, 0-201-88954-4, Addison-Wesley, 1997.

Code Complete: A Practical Handbook of Software Construction, Code Complete, Steve McConnell, 1-55615-484-4, Microsoft Press, 1993.

Effective C++: 50 Specific Ways to Improve Your Programs and Designs, Effective C++, Scott Meyers, 0-201-92488-9, Addison-Wesley, 1998.

The Elements of Style, William Strunk, Jr. and E. B. White, 0-02-418200-1, MacMillan Publishing Co., Inc., 1979.

More Effective C++: 35 New Ways to Improve Your Programs and Designs, More Effective C++, Scott Meyers, 0-201-63371-X, 1996, Addison-Wesley, 1998.

The Practice of Programming, Brian W. Kernighan and Rob Pike, 0-201-61586-X, Addison-Wesley, 1999.