Guide to Style and Idiom: C/C++

Version 1.01, © 2003-5, Garth Santor

Printed: 16/01/2005

Introduction

Programmers write code to communicate ideas, both the computer that will run the program and to the other programmers that will read their code. Language syntax, grammar and semantics are essential and sufficient to communicate ideas to a computer. For human readers of code this is not so. Unlike computers, our ability to comprehend code can be helped or hindered by the style and choice of idiom employed by the author.

So what are style and idiom? And how do they affect a programmer's comprehension of code?

Style

Style has been defined as "*n.* **1.** Manner of expressing thought, in writing or speaking; distinctive or characteristic form of expression: a florid *style*; the *style* of Mark Twain. … **10.** The conventions of typography, design, usage, punctuation, etc., observed by a given publishing house, printing office, or publication." (FW-78) The essence of style in programming is not the meaning or function of code, but its presentation. The following code segments are functionally identical, but possessing radically different styles.

```
unsigned a(unsigned b){return b<=1?1:b*a(b-1);}
```

```
/** @fn unsigned factorial( unsigned n )
    @brief Recursive factorial.
    @param n [in] the top term of the factorial.
    @return The factorial of n, n! */
unsigned factorial( unsigned n ) {
    if( n <= 1 )
        return 1;
    else
        return n * factorial( n - 1 );
}
```

Idiom

Idiom is defined as "*n.* **1.** An expression peculiar to a language, not readily understandable from its grammatical construction or from the meaning of its component parts. … **3.** The special terminology or mode of expression of a class, occupational group, etc.: *legal* idiom." (FW-78). Idiom has much more to with implementation rather than presentation. Idiom focuses on the structure and sequencing of code, the choice of statements or operators and the accepted practices of professional programmers regarding these issues.

Names/Identifiers

Terminology

1. Identifiers have three parts, a prefix, a name, and a suffix.
   e.g. `strFileName` *str* = prefix, *FileName* = name
   e.g. `time_t` *time* = name, *_t* = suffix
2. *Lower-case* identifiers are composed entirely of lower case letters.
   e.g. `size`

3. *Upper-case* identifiers are composed entirely of upper case letters.
   e.g. `UCHAR`
4. *Title-case* identifiers are all lower case except for the first letter of each distinct word.
   e.g. `GetTickCount`
5. *Camelback-case* identifiers are all lower case except for the first letter of each distinct word, but not the first word.
   e.g. *`strFileName`*
6. *Saddleback-case* identifiers are all lower case except that each distinct word is separated by an underscore character.
   e.g. *`selection_sort`*
7. *Procedure* type of function that does not return a value.

Rules

1. Be consistent!
2. Use descriptive names for global and class members, short names for locals. (KP-99)
3. Variable and constant names should be nouns that describe the contents of the variable.
   e.g.
   ```
   int counter;    // bad – describes a counting device
   int count;      // good – describes the number
   ```
   a. Boolean variables should identify the true condition.
      e.g.
      ```
      bool isRunning = false;
      ```
4. Use active names for functions and procedures. Use a strong verb and perhaps a noun to describe, 1) the action being performed, and 2) the object that it will be performed on.
   e.g.
   ```
   abort();
   set_pixel();
   get_time();
   ```
   a. Functions that return Boolean values should be clear about what input returns true.
      e.g.
      ```
      if( is_active( windowHandle ) ) ...
      ```
   b. Functions that return mathematical computations should be named after the operation being performed.
      e.g.
      ```
      val = factorial( num );
      ```
   c. Nouns describing the object are not used when redundant.
      e.g.
      ```
      class Stack {
         int popStack();
      };
      Stack stk;
      stk.popStack();
      ```

      the following is preferable:

      ```
      class Stack {
         int pop();
      };
      Stack stk;
      stk.pop();
      ```
   d. Nouns describing the object are not used when *easily* inferable.
      e.g.
      ```
      template<typename T> void swap( T& a, T& b ) { ...
      ```
      replaces:
      ```
      void swap_int();
      void swap_double();
      void swap_string(); ...
      ```

5. Use camelBackCase for the names of normal variables, functions and methods.
   e.g.
```
unsigned nElements;
void glVertex3d( double, double, double );
double Timer::getTime();
```

6. Use saddle-back case for the names of non-member functions.
   e.g.
```
void selection_sort( std::vector<unsigned>& a );
```

7. Use UPPER_CASE for the names of constant items and macros. Upper case words are separated by an underscore character ('_').
   e.g.
```
unsigned const MAX_ELEMENTS = 42;
```

8. Do not name macros with short names often used as template parameter names such as T and U.

9. Decorate global variables (not constants) with a 'g_' prefix.
   e.g.
```
long g_processId;
```

10. Use TitleCase for the names of **class**, **struct**ures and **enum**erated types and **typedef**s.
    e.g.
```
class Timer;
struct ListNode;
```

11. Decorate member variables consistently with an 'm_' or postfix (or a prefix) underscore.
    e.g.
```
class Timer {
   ...
private:
   long m_elapsedTime;
   long elapsedTime_;
};
```
    The 'm_' style is most appropriate to code developed specifically for the Microsoft Windows platform.

12. Decorate static member variables with a 'cs_' prefix.
    e.g.
```
template<typename T>
class Singleton {
   ...
private:
   static T* cs_pSingleton_;
};
```

13. Decorate static local variables with an 's_' prefix.

14. Do not use names that start with an underscore. (SA-OCT04) Underscores are traditionally prefixed to identifiers in their object file representations (which assembly language programmers need to reference.) Prefixed underscores produce unnecessary confusion.

15. Do not use double underscores anywhere in an identifier especially as a prefix. They are often confused with system and implementation specific reserved words.
    e.g. `__argc` is defined in Microsoft Windows GUI applications.

16. Do not use names that differ only by the presence of upper- or lower-case letters.
    e.g.
```
int nStudents, nstudents;
```

17. Names should not include abbreviations that are not generally accepted.

18. Choose identifier names that suggest usage.

19. Write code in such a way that it is easy to change the prefix.

20. Encapsulate global variables, constants, enumerated types and typedefs in a namespace but prefer scoping them to a class when possible.

21. When declaring identifiers that are constant, apply the constant modifier *after* the type.

    '`int const MAX = 42;`' is prefered to '`const int MAX = 42;`'

    The value of this style becomes apparent when reading pointer declarations. '`const int * p;`' is typically read by humans as, "a constant integer pointer." Is the integer constant or the pointer constant?

    Consider: '`int * const p;`' The pointer is clearly the constant, not the integer. We read the declaration as, "a constant pointer to integer." We are unambiguous since we read the declaration right to left.

    By writing '`const int * p;`' as '`int const * p;`' and reading right to left we hear, "a pointer to constant integer", which is both correct and unambiguous.

    Therefore '`int const * const p;`' can be read as, "a constant pointer to a constant integer."

22. Don't use Hungarian notation in C++, only C. C++ is type-safe so the purpose of Hungarian notation is unnecessary duplication. It is also impossible to apply to template parameters.

Expressions and Statements

Terminology

1. RAII – Resource Allocation Is Initialization. A variable or object will receive its initial value during allocation of that variable or object.
2. Smart Pointer – Any resource management object that automatically handles resource deallocation when the object looses scope. They may or many not be reference counted.

Layout

Indenting, word spacing, line spacing and line-length are the most import aspects of code layout.

1. The amount to indent is not important to standardize. (SA-OCT04) The indent amount should however be consistent within a file.
2. Indent the body of functions, loops and if-else statements to show structure.

```
// bad
int c; for(c = 0; a[c] != 0; ++c);
return c;

// good
int c;
for( c = 0; a[c] != 0; ++c )
    ;
return c;
```

3. Format scope brackets consistently! Use only one of the standard styles throughout your code.

```
/* K&R – named after Kernighan and Ritchie, because the examples
   in their book "The C Programmer Language" are formatted this
   way.  Also known as "kernel style" or "One True Brace Style"
   (1TBS).  Tabs are 4 or 8 (usually) spaces. */
for( int i = 0; i < n; ++i ) {
    double x = sin( theta * i );
    cout << i << endl;
}

/* Allman – named after Eric Allman, who wrote many BSD-Unix
   utilities.  Also known as "BSD style" and is preferred by
   Visual Studio. Tabs are 4 (usually) or 8 spaces. */
for( int i = 0; i < n; ++i )
{
    double x = sin( theta * i );
    cout << i << endl;
}

/* GNU – named after the style used in the GNU emacs editor and
   other Free Software Foundation (FSF) code.  Tabs are 4 spaces
   and the braces are midway between the tabs. */
for( int i = 0; i < n; ++i )
  {
    double x = sin( theta * i );
    cout << i << endl;
  }

/* Whitesmith – named after the style used in examples that
   shipped with the early compiler "Whitesmiths C".  Tabs are
   4 or 8 (usually) spaces. */
for( int i = 0; i < n; ++i )
    {
    double x = sin( theta * i );
    cout << i << endl;
    }
```

4. There is no standard line length. However, a single line should not wrap or exceed the width of a typical window. A good guideline is 10 words per line. (SA-OCT04)

Statements

1. Use explicit RAII and smart pointer. (SA-OCT04) This practice increases the likelihood that your code will be exception safe and able to handle multiple function exit points.

2. Declare variables as close to the first use of the variable as possible. There is a statistical relationship between the number of lines between a variables declaration and its first use and the number of bugs in a system.

e.g.
```
void bad_func() {
      bool done = false;
      ... // opportunity to accidentally modify 'done'
      while( !done ) {
         ... // loop body
      }
   }

   void good_func() {
```

```
        ... // opportunity to accidentally modify 'done'
        bool done = false;
        while( !done ) {
            ... // loop body
        }
    }
```

3. Prefer for-loop counter declaration in the for loop statement over prior to the for-loop. for-loop declaration limits the scope of the counter variable to the duration of the for loop.
   e.g.
```
int bad;
for( bad = 0; bad < n; ++bad ) { ... }
// bad still exists

for( int good = 0; good < n; ++good ) { ... }
// good does not exist
```

4. Use `for(;;)` or `while(1)` to code an infinite loop, but prefer exit conditions.

5. Don't use `goto` (unless there is no other solution).

Expressions

1. Initialize variables during declaration. Code is more compact. Many objects can perform initialization at construction more efficiently than construction followed by assignment.
```
std::string s;
s = "Kernighan";
```
   May requires two memory allocations, one during construction and one during assignment in addition to one memory de-allocation.
```
std::string s = "Kernighan";
```
   Only requires a single memory allocation and no memory de-allocations.

2. Prefer pre-increment and pre-decrement to post-increment and post-decrement. Post-increment/decrement may require the creation of a temporary value, whereas pre-increment/decrement does not.
   e.g.
```
for( int i = 0; i < n; i++ ) ... // bad
for( int i = 0; i < n; ++i ) ... // good
```

3. Prefer the arithmetic assignment operators to separate arithmetic and assignment operators.
   e.g.
```
x = x + 2; ... // bad
x += 2; ...    // good
```
   Arithmetic assignment operators can prevent some order of operations errors.
   e.g.
```
x = x * y + 2;       // bad – missing parenthesis
x = x * (y + 2);     // good – correct parenthesis
x *= y + 2;          // good – parenthesis unnecessary
```

4. Don't store a result prior to return that result.
   e.g.
```
double mean( double sum, int nElements ) {
    double average = sum / nElements;
    return average;
}
```
   Should be:
   e.g.
```
double mean( double sum, int nElements ) {
    return sum / nElements;
}
```

5. When mixing unrelated operators (e.g. `< <= == != >= >` with `&& ||`) use parentheses to remove ambiguity in the understanding of an expression.

6. Use parentheses to make your expressions clearer.

7. Use the natural form of expressions.
8. Break up complex expressions.
9. Use the `?:` operator only for short expressions, and only where it does not implement business logic.
10. Explicitly cast the result expressions of the **`?:`** operator when the result expressions are a different type than the value being assigned to.

```
double x = booleanCondition ?
                integerExpression :
                doubleExpression;
```

Should be:

```
double x = booleanCondition ?
                static_cast<double>( integerExpression ) :
                doubleExpression;
```

Compilers will typically perform an implicit cast of one result expression to the type of the other result expression. This most often confuses templated and overloaded functions.

Functions

Terminology

1. *Input parameter* is a parameter whose value originates in the calling scope and whose calling scope value is not modified by the function.
2. *Mutable parameter* or *modifiable parameter* is a parameter whose value originates in the calling scope but whose value is modified by the function.
3. *Output parameter* is a parameter whose value originates in the function and whose calling scope value is set by the function.

Parameters

1. Input parameters should be passed by value for primitives or constant reference for objects.
2. Mutable and output parameters should be passed by *non*-constant reference.
3. Output or mutable parameters should precede input parameters in a function parameter specification. The output parameters should be first so as to mimic the order of assignment operations.

   e.g. `char* strcat( char* dest, const char* src );`

   is the C-language equivalent to

   `string& operator += ( string& dest, const string& src );`

   which mimics

   `dest += src;`

   `void add( Matrix& res, const Matrix& lhs, const Matrix& rhs );`

   mimics

   `res = lhs + rhs;`
4. Parameters representing the left and right operands of a binary operator should be called `lhs` and `rhs` respectively.

Commenting

Commenting serves four purposes within code:

1. To document the design or purpose of the code.

2. To document the usage of the code.
3. To explain difficult code.
4. To generate code documentation.

General Principles

1. Comment as you code! Commenting is a tool to help you and others understand the code you write.
2. If a comment doesn't help understanding – don't write it.
3. Comments should as a minimum identify every file.
4. Don't go overboard! Too much commenting is almost as bad as too little.

Documenting Design

1. Use program description language (PDL)[1] which is an English-like description of the steps required to solve your problem. Guidelines for PDL usage are described by Steve McConnell[2]:
    a. Use English-like statements that precisely describe specific operations.
    b. Avoid syntactic elements from the target programming language. PDL allows you to design at a slightly higher level than the code itself. When you use programming-language constructs, you sink to a lower level, eliminating the main benefit of design at a higher level, and you saddle yourself with unnecessary syntactic restrictions.
    c. Write PDL at the level of intent. Describe the meaning of the approach rather than how the approach will be implemented in the target language.
    d. Write PDL at a low enough level that generating code from it will be nearly automatic. If the PDL is at too high a level, it can gloss over problematic details in the code. Refine the PDL in more and more detail until it seems as if it would be easier to simply write the code.
    e. Once the PDL is written, you build the code around it and the PDL turns into programming-language comments.
2. PDL comments should be written above the line or block of code whose purpose it describes.
   e.g.
```
// get a number within the guess range
int number;
do {
   cin >> number;
} while( 1 <= number && number <= 10 );
```
3. Function and method comments should document:
    a. The purpose of the function (i.e. what it does, not how it does it).
    b. The return value of the function. What is returned, what range of values are possible.
    c. Each parameter of the function. What is the purpose of the parameter, or what does it represent. Document any preconditions placed on the parameter.
    d. Document any side effects or state changes caused by calling the function. Carefully note state changes for both successful and unsuccessful execution
    e. Document any exceptions that could be generated by the function.

[1] PDL was originally developed by the company Caine, Farber & Gordon in 1975.
[2] McConnell, Steve (1993) Code Complete Microsoft Press.

e.g.
```
/** @fn int factorial( int n )
    @return The factorial of 'n'
    @param n [in] A non-negative integer.

    Calculates the factorial of a number (n!).
    @note Passing a negative integer will result in an
    invalid result */
int factorial( int n ) { … }
```

Explaining Code

1. Place comments on the end only to explain cryptic code. Cryptic code can be thought of as code that seasoned programmers would have to stop and think about what is happening.
   e.g. `a = !!b; // map 0 -> 0, !0 -> 1`
2. Do not explain the obvious – i.e. something that can be easily looked up or something that any experienced programmer would know.
   e.g. `++x;  // increment x by one`

Generating Code Documentation

DOxygen is our preferred code documentation system.

1. Comment files with the following information located at the top of the file:

```
/**   @file: filename.cpp
      @author Garth Santor
      @author gsantor@fanshawec.ca
      @author http://infotech.fanshawec.ca/gsantor/
      @date 2003-04-17
      @version 0.1
      @note Developed for Visual C++ 7.0/GCC 3.2
      @brief Implementation of …
      */
```

2. Comment functions with the following information located immediately before the function:

```
/**   @fn unsigned count_bits( unsigned var )
      @brief Count of set bits in variable. (1 bits)
      @param var [in] variable whose bits will be counted.
      @return Number of bits in var that are set to 1. */
unsigned count_bits( unsigned var ) {
```

Macros

1. Prefer an inline function or a templated function to macro.
   ```
   #define MAX(a,b) ((a) > (b) ? a : b)
   ```
   Macro MAX allows arguments ‘a’ and ‘b’ to be of different types. The type conversion required to compare the arguments is done implicitly and therefore there is the potential for the conversion to be done incorrectly.
   ```
   template<typename T> inline T& max( T& a, T& b ) {
       return a > b ? a : b ;
   }
   ```
   Function max guarantees that arguments ‘a’ and ‘b’ are the same data type. If different data types are to be compared, the programmer must explicitly convert one of the types.
2. Prefer constants to macros. Constants have a specific data type and can be type-checked by the compiler. Macro literals can be implicitly converted to other, potentially incompatible, data types.
   ```
   #define VAL 42
   ```

VAL could be interpreted by the compiler as a **`char`**, **`int`**, **`float`**, or **`double`**.

```
const int CVAL = 42;
```

CVAL can only be type **`int`**.

Data Structures

1. For any casual use of an array, use `std::vector<T>` instead. The STL vector container provides flexibility, safety and performance levels close to or identical to arrays.

Files/Source Code

1. Name C++ source files with a .cpp extension.
2. Name C++ files the same as the primary class they define and with lower case.
3. Name C/C++ header files with a .h or .hpp extension. The .hpp extension makes it very clear that this is a C++ header and not a C header. Favour the .hpp extension for C++ specific header files.
4. Protect header files from multiple includes with a header guard.

```
#ifndef GUARD_myheader_h
#define GUARD_myheader_h
… // body of header file
#endif
```

5. Include in your header file every header file required to declare the data types used by your header file. You shouldn't assume that another header file will always include something for you.

```
/** @file NameList.hpp */
#ifndef GUARD_mystack_h
#define GUARD_mystack_h

#include <string>
#include <vector>

typedef std::vector< std::string > NameList;

#endif
```

6. When nesting header files (include a header within a header), try to minimize the nest. Only include the header files required by the declarations within your header file.
7. Never implement a non-templated or non-inlined function in a header file. The linker will get confused about which implementation to use when that header is included in more than one source file.
8. Never code a **`using`** statement in the global scope of a header file. Always fully resolve the names of classes and types in the global scope of a header file.
9. Avoid functions or methods that are too long. Break down your logic into separate steps, use functions and function calls (private member functions). There are rare exceptions where it isn't feasible to limit the size, check with your prof if you think you have such a case, they do occur. General rules of method/functions are that they:
    a. do only one thing
    b. be under 25 line (including spaces) preferably
    c. be under 50 lines if at all possible.
    d. Have only one return statement and it is the last statement in the method.
    Or possibly have two as the two cases of an extremely simple if statement.

Templates

1. Template parameters are declared as meta-type **`class`** if they can only be resolved by a user-defined type, or declared as meta-type **`typename`** if they can also be resolved by primitive types.

2. Document (via commenting) all extension points within your template. Template extension points are points within the code where the behaviour of the templated code could be modified by a template parameter.

   Example:

```
template <typename T>
void func( T obj )
{
    typename SomeTraits<T>::some_type x;
    obj.meth();
    cout << obj;
    f( obj );
}
```

   The extension points are:

   - data type '`x`'
   - the method called '`meth`'
   - the stream insertion operator – '`operator <<`'
   - the parameterized function '`f`'

   Documentation:

```
/** @fn template<typename T> void func( T obj )
    …
    @note Extension point: SomeTraits<T>::some_type
    @note Extension point: T::meth()
    @note Extension point: operator << T
    @note Extension point: f( T)
*/
```

Appendix A: References

FW-78	Funk and Wagnalls Standard College Dictionary: Canadian Edition (1978)
KP-99	Kernighan, Brian & Pike, Rob (1999) The practice of programming. Addison-Wesley.
SA-OCT04	Sutter, Herb & Alexandrescu, Andrei 'C++ Coding Standards: C/C++ Users Journal, October 2004'

Appendix B: Authorship

Garth Santor (gsantor@fanshawec.ca : http://infotech.fanshawec.ca/gsantor/)

- primary author and editor.