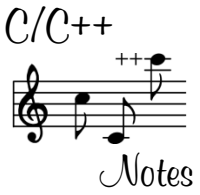
Appendix G



C/C++ Coding Style Guidelines

© 1992-2016 Ray Mitchell

NOTE G.1

C/C++ Coding Style Guidelines

1. Introduction

The C and C++ languages are free form, placing no significance on the column or line where a token is located. This means that in the extreme a program could be written either all on one line with token separators only where required, or with each token on a separate line with blank lines in between. Such variations in programming style can be compared to accents in a spoken language. As an accent gets stronger the meaning of the conversation becomes less understandable, eventually becoming gibberish. This document illustrates some of the most commonly accepted conventions used in writing C/C++ programs, providing the consistent guidelines needed to write "accentless" code.

2. Implementation Files and Header Files

Implementation files typically have a *.c* or *.cpp* extension and will minimally contain any needed non-inline function definitions and external variable definitions, although they often also contain many of same types of things contained in header files. Header files typically have a *.h* or no extension and are included in other files via a #include directive. They typically contain items like macro definitions, inline function definitions, function prototypes, external variable referencing declarations, templates, typedefs, and type definitions for classes, structures, unions, and enumerations. They also contain #include directives when necessary to support other items in the file. Header files must never contain function definitions, external variable definitions, "using" directives or statements, or run time code that is not part of a macro or an inline function. In applications where multiple implementation files include many of the same header files it is sometimes acceptable to place all of these #include directives in a single header file and include it instead.

Standard header files are typically supplied with the compiler, while 3rd-party library vendors provide custom header files to support their products. In addition, programmers are always free to create their own custom header files to meet their needs. Header files must always be logically organized, includable in any order, and each must implement an "Include Guard" to prevent the multiple inclusion of its contents in another file, even if the header is included multiple times.

3. File Organization

The suggested order for some common file items is as follows, with a blank line placed between each group. The most important consideration is that a something that relies on something else be placed after it. Except for item 1, which is always required, not all files will contain all of these while some files may contain items not shown:

1. A block comment containing information about the contents and functionality of the file, the developer, the revision history, and anything else that might be helpful in understanding and maintaining it;
2. #include directives;
3. "using" statements/directives (C++);
4. #define directives (C);
5. **typedef**s and custom data type definitions;
6. **const** variable declarations (C++);
7. External variable declarations;
8. Function prototypes;
9. Function definitions in some meaningful order, with a block comment before each describing its syntax and purpose.

 © 1992-2016 Ray Mitchell

NOTE G.2

4. Comments

Comments should be used wherever there might be a question about the workings or purpose of an algorithm, statement, macro definition, or anything else that would not be obvious to a programmer unfamiliar with the code. It should never be necessary to "reverse-engineer" a program. Comments should explain algorithmic operations, not semantics. For example, an appropriate comment for the statement *distance = rate * time;* might be "position of projectile", while something like "multiply rate by time and assign to distance" says nothing useful. With a wise choice of identifiers code can often be made somewhat self-documenting, thereby reducing the need for as many comments.

The format of a comment is normally determined by its length. Comments occupying more than one line, such as a file title block or algorithm description, should be in block comment form. Except for "partial-line" comments, comments should be placed prior to and aligned with the code they comment as follows. Depending upon the compiler, C++ style comments could cause compatibility issues in C89/C90 code.

A C style block comment (also acceptable in C++):

```
/*
 *  Use a block comment whenever comments that occupy more than one line are needed.  Note
 *  the opening / is aligned with the code being commented, all asterisks are aligned with each
 * other, and that all comment text is left aligned.
 */
for (nextValue = getchar(); nextValue != EOF; nextValue = getchar())
```

An equivalent C++ style block comment (also acceptable in C since C99):

```
//
//  Use a block comment whenever comments that occupy more than one line are needed.  Note
//  the opening / is aligned with the code being commented, all // are aligned with each other, and
//  that all comment text is left aligned.
//
for (nextValue = getchar(); nextValue != EOF; nextValue = getchar())
```

C and C++ style "full-line" comments:

```
/* This is a full-line C style comment. */
// This is a full-line C++ style comment.
for (nextValue = getchar(); nextValue != EOF; nextValue = getchar())
```

It is possible to over comment a program to the point of making it unreadable. This is especially true when code lines and comment lines are intermixed. Many comments can be reduced to the point of being small enough to fit to the right of the code being commented while still conveying meaning. To be most readable code should reside on the left side of a page with comments on the right, opposite the code they comment. Whenever possible all such comments should start in the same column as each other, as far to the right as possible. Deciding on the appropriate column is a compromise and there will usually be exceptions in any given program. Using tabs instead of spaces when positioning such comments reduces the effect of making minor code changes but may not be interpreted correctly by the printer. The following illustrates these "partial-line" comments:

```
while ((nextValue = getchar()) != EOF)      /* while source file has characters */
{
        if (nextValue == '.')               /* got sentence terminator */
                break;                      /* don't read any more characters */
        else                                // must continue reading sentence
                ++charCount;                // update characters read in sentence
}
```

© 1992-2016 Ray Mitchell

NOTE G.3

The following examples use exactly the same code but with the comments (if any) placed differently. Which programmer would you hire or be most willing to maintain code for?

WRONG – NO COMMENTS

```
while ((nextValue = getchar()) != EOF)
{
    if (nextValue == '.')
        break;
    else
        ++charCount;
}
```

WRONG – CLUTTERED & HARD TO READ

```
while ((nextValue = getchar()) != EOF)/* while source file has characters */
{
    if (nextValue == '.')/* got sentence terminator */
        break;              /* don't read any more characters */
    else/* must continue reading sentence */
        ++charCount;                    /* update characters read in sentence */
}
```

CORRECT – FULL LINE COMMENTS

```
/* while source file has characters */
while ((nextValue = getchar()) != EOF)
{
    /* got sentence terminator */
    if (nextValue == '.')
        /* don't read any more characters */
        break;
    /* must continue reading sentence */
    else
        /* update characters read in sentence */
        ++charCount;
}
```

CORRECT – PARTIAL LINE COMMENTS

```
while ((nextValue = getchar()) != EOF)      /* while source file has characters */
{
    if (nextValue == '.')                   /* got sentence terminator */
        break;                              /* don't read any more characters */
    else                                    /* must continue reading sentence */
        ++charCount;                        /* update characters read in sentence */
}
```


© 1992-2016 Ray Mitchell

NOTE G.4

5. Literals

A "**magic number**" is defined as any literal (numeric, character, or string) embedded in a program's code or comments. They usually make programs cryptic and difficult to maintain because their meaning is not obvious and, if they must be changed, the likelihood of missing one or changing the wrong one is high. There are a few cases, however, where magic numbers are acceptable, including:

> 0 & 1 as array indices or loop start/end values, 1 as an increment/decrement value, 2 as way to double/halve a value or as a divisor to check for odd/even, coefficients in some mathematical formulae, some informational strings, cases dictated by common sense.

In general, however, the #define directive (in C), **const**-qualified variables (in C++), or enumerated data (C and C++) should be used to associate meaningful names with literals. This permits values to be changed everywhere in a program by making a change in only one place. Avoid, however, the pitfall of choosing names that reflect the value they represent, such as *#define SIX 6*. This is as bad as directly coding *6* itself. Would you ever define *SIX* to be anything other than *6*?

Because an implementation may define the macro NULL as a pointer type, it must never be used in other than pointer contexts.

6. Naming Conventions

1. Choose meaningful names that reflect usage whenever possible and practical.
2. Function & function-like macro names should be verbs or ask questions; all other names should be nouns or state facts.
3. To avoid conflicts with internal names, leading or trailing underscores should be avoided in application programs.
4. Object-like macro, **const**-qualified variable, and **typedef** names should be in upper case.
5. Function, function-like macro, and **class**/**struct**/**union**/**enum** tag names should be in Pascal case (i.e., *PascalCaseName*).
6. All other names should be in camel case, (i.e., *camelCaseName*).

7. Compound Statements (Block Statements)

A compound (block) statement is defined as zero or more statements enclosed in curly braces. Although initializer lists are also enclosed in braces they are not compound statements. Compound statements are a required part of function, structure, class, union, and enumeration definitions as well as **switch** statements, but they are optional with **if**, **else**, **for**, **while**, and **do** statements unless more than one statement is to be associated with those statements. Here are some examples:

Optional Compound Statement Not Used	Optional Compound Statement Is Used	Compound Statement Required

for (...)
  velocity = 1;
acceleration = 2;
	for (...)
	{
	  velocity = 1;
	}
	acceleration = 2;
		for (...)
		{
		  velocity = 1;
		  height = 8;
		}
		acceleration = 2;

© 1992-2016 Ray Mitchell

NOTE G.5

8. Placement of Braces { }

A point of contention among some C/C++ programmers is the placement of the braces used for compound statements and initializer lists. The two most popular formats are shown on the left and right below and are known as the "brace under" and "brace after (K&R)" styles, respectively. Choose one and use it exclusively and consistently. Do not intermix the two or use a different variation. The braces are sometimes omitted if only one statement is needed with **if**, **else**, **for**, **while**, or **do**:

```
int SomeFunction(void)                  int SomeFunction (void)  (Both forms are the same!)
{                                       {
    declarations/statements                 declarations/statements
}                                       }
```

```
if (...)                                if (...) {
{                                           declarations/statements
    declarations/statements             } else if (...) {
}                                           declarations/statements
else if (...)                           } else {
{                                           declarations/statements
    declarations/statements             }
}
else
{
    declarations/statements
}
```

```
for or while (...)                      for or while (...) {
{                                           declarations/statements
    declarations/statements             }
}
```

```
do                                      do {
{                                           declarations/statements
    declarations/statements             } while (...);
} while(...);
```

```
switch (...)                            switch (...) {
{                                           declarations
    declarations                        case ... :
    case ... :                              statements
        statements                          break;
        break;                          }
}
```

```
struct or class or union or enum tag    struct or class or union or enum tag {
{                                           declarations
    declarations                        };
};
```

```
double factors[] =                      double factors[] = {
{                                           initializer, initializer, ...
    initializer, initializer, ...       };
};
```

 © 1992-2016 Ray Mitchell

NOTE G.6

struct/**class**/**union**/**enum** type definitions and initialized variable declarations may be placed entirely on the same line if they will fit, but this is not appropriate for any of the other constructs:

struct *or* **class** *or* **union** *or* **enum** tag { declaration, declaration, … };
double factors[] = { initializer, initializer, ... };

9. Indenting

Indenting is used strictly to enhance a program's human readability and is totally ignored by the compiler. An indent signifies the association of one or more statements with a previous, less indented construct, as in the following examples:

```
if (...)
{
        x = 1;
        printf(...);
}
else if (...)
        z = y;
else
{
        int t = 2;

        ++v;
}
```

```
for (...)
        while (..)
        {
                x = 2;
                ++d;
        }
do
{
        y = 3;
        x = y + 9;
} while (...);
```

```
int main(void)
{
        int ch = 'A';

        putchar(ch);
        return EXIT_SUCCESS;
}
```

The rules for how and when to indent are simple:

1. **Braces must be aligned according to the placement formats previously discussed.**

2. **All declarations and other statements associated with a construct must be indented equally from the left edge of that construct.**

3. **The width used for all indents must be consistent throughout any program.**

One final consideration is the width of each indent. A default tab stop is 8 spaces wide on some systems, but this is too wide for most programs because the code ends up going off the right edge of the screen/paper or wrapping around. Because of this an indent width of 3 or 4 spaces (or at least 1/4 inch) is recommended instead. Widths smaller than this tend to be hard to discern while larger values run out of room more quickly. Actual spaces should be used instead of hard tab characters because the system printer often has no way of knowing the editor's tab setting. Thus, it often assumes that it is 8 and produces a program listing whose indenting does not match that seen within the editor itself.

10. Multiple Statements on One Line

Do not put more than one statement on a line. Similarly, put the body of an **if**, **else**, **for**, **while**, **do**, or **case** on the next line. Chained assignments such as *x = y = z = 0;* are permissible as long as all operands are logically related. Multiple variables of the same type may be declared on the same line, comma separated, unless a comment is required. The following examples illustrate some acceptable and non-acceptable formats:

© 1992-2016 Ray Mitchell

NOTE G.7

Do	**Don't**

`x = 5;`
`y = printf(...);`
`Delete();`
	`x = 5; y = printf(...); Delete();`

`for (idx = 0; idx < MAX; ++idx)` `  printf("%d", idx);`	`for (idx = 0; idx < MAX; ++idx)  printf("%d", idx);`

`if (y < x)`
`  printf("%d\n", x);`
`else`
`  printf("%d\n", y);`
	`if (y < x)  printf("%d\n", x);`
	`else  printf("%d\n", y);`

`case 'A':`
`  putchar('A');`
`  break;`
	`case 'A':  putchar('A');  break;  (See below)`

A common exception is where multiple **switch** cases are almost identical. Readability may actually be enhanced by grouping them as:

```
case 'A':   putchar('A');   break;
case 'B':   putchar('B');   break;
case 'C':   putchar('C');   break;
case 'D':   putchar('D');   break;
```

11. The "else if" Convention

If an **if** follows an **else**, separated by only whitespace, the **if** should be placed on the line with the **else**, thereby forming the standard C/C++ "**else if**" construct.

Do	**Don't**

`if (a > b)`
`  ++a;`
`else if (a < x)`
`  ++b;`
	`if (a > b)`
	`  ++a;`
	`else`
	`  if (a < x)`
	`  ++b;`

12. Functions

Each function definition should be preceded by a block comment that describes its syntax and what it does. A function's return type must always be specified and never defaulted to type **int** by omitting it. Functions not returning a value must use the keyword **void**. Placing a return type on its own line in column 1 is common. C functions with no parameters must use **void** as the parameter list. Although C++ functions with no parameters may also use **void** for this purpose, it is more common to leave their parameter lists empty.

Either the definition of a function or its prototype must visible to the caller before the function is called. Explicitly declaring a function as external by using the keyword **extern** is obsolete. Prototypes for user functions needed by more than one file and all library functions, such as those in the standard library, must be kept in header files that are included by any files needing them. Do not duplicate the prototype in each file. In multi-file programs all functions not shared with other files should be declared **static** to limit their scope.

 © 1992-2016 Ray Mitchell

NOTE G.8

13. External Variables

Referencing declarations of external variables used by more than one file must be kept in header files that are included by any files needing them. Do not by duplicate the declaration in each file. In multi-file programs, all external variables (except **const**-qualified external variables in C++) not shared with other files should be declared **static** to limit their scope.

14. Automatic Variables

Automatic variables should be declared as close to the point of use as is practical and, if initialization is required, also initialized as close to the point of use as is practical. A program that initializes automatic variables when declared but doesn't use them until much later is difficult to read.

The use of extremely large automatic aggregate objects (i.e., arrays, structures, classes) can cause compiler/run-time problems due to the amount of stack space required, and initializing such objects can add significant run time overhead. For similar reasons functions should avoid the arbitrary passing/returning of aggregates. Instead, pass/return pointers or references to them. Making local objects **static** can sometimes solve stack size issues but can also result in more overall memory usage and can prevent the implementation of multi-threaded applications.

15. Operators

All ternary and binary operators except class/structure/union member operators should be separated from their operands by spaces. 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, for example:

```
if (x=y  &&  6<t  ||  ++p)
```

Spaces should be placed after, but not before, all commas and semicolons. They should also be placed after keywords that are followed by expressions in parentheses, with the exception of **sizeof** and **return**. On the other hand, macros with arguments must not and function calls should not have a space between the name and the left parenthesis. Unary operators should not be separated from their single operand.

Compound assignments and pre/post increment/decrement expressions should always be used instead of their equivalent longer forms. The shorter expressions are evaluated as is and are not expanded to the longer forms. Parentheses around the entire right side of such expressions are not needed, as the following illustrates:

Do	**Don't**
x += 2	x = x + 2
x++ or ++x	x = x + 1 or x += 1
x >>= 2	x = x >> 2
x /= 3 + y	x = x / (3 + y) or x /= (3 + y)

© 1992-2016 Ray Mitchell

NOTE G.9

16. Portability

Beware of making assumptions about:

1. The number of bits in a data type;
2. Byte ordering (big endian vs. little endian);
3. Bit-field ordering (left-to-right vs. right-to-left);
4. The size of different pointer types;
5. The compatibility of pointers with arithmetic types;
6. The internal representation of negative integer numbers;
7. The internal representation of floating point numbers;
8. The operation of a right shift on negative integer numbers (arithmetic vs. logical);
9. The padding of structures/classes/unions;
10. The machine's character set (ASCII, EBCDIC, etc.);
11. The evaluation of expressions having side effects when used as macro arguments;
12. The order of evaluation of expressions not involving sequence point operators, including expressions used as function arguments.

 © 1992-2016 Ray Mitchell