A-*Code-Formatting-Style (ACFS)*

v1.11

©2025 este este[1]

This document describes *A-Code-Formatting-Style* (aka ACFS, hereafter called *The Style*) that I believe facilitates understanding how written code works. *The Style* prioritizes managing whitespace, density, and code element alignment so the reader can more easily discern and navigate the written code structure. *The Style* deprioritizes written code compactness. I believe that densely written code can harm code readability. I also think the desire for compact code has waned from the early days of programming with the easy access to large computer screens.

Note: *The Style* assumes use of a coding Integrated Development Environment (IDE) which displays the code using a fixed-width (monospaced) font. It is written from a "C-like" code perspective so the code formatting examples use C-like pseudocode. The ellipse (...) is used to convey "*and so on*".

1. *Text Definition*

   For this discussion, there are two types of text.

   - Code Text is the text of any code statement: an instruction for the computer to perform.
   - Comment Text is the text of any comment: an instruction for the human reader to understand.

2. *Whitespace Definition*

   Whitespace can be either vertical or horizontal.

   - Vertical whitespace is created by inserting a blank line via the newline character with the keyboard Enter key. **Note:** Vertical whitespace may always be added beyond what is stated herein if the programmer decides code readability will be improved.
   - Horizontal whitespace is created by inserting a tab character ("tabs") with the keyboard Tab key and/or a space character ("space") with the keyboard Spacebar.

   The Style specifies the use of the tab character for most horizontal whitespace, when you *meet this "tab" requirement*:

   - All the programmers on a project following *The Style* (henceforth called *The Team*) must utilize coding Integrated Development Environments (IDEs) which [a] actually insert a tab character and [b] allow a tab character to have a width equal to a user-defined number of spaces. This gives each programmer flexibility to control how horizontal whitespace appears on their individual screen. In *The Style*, the default tab character width is 2 spaces but *individual* programmers can set the tab character width in *their* IDE to a width of some other number of spaces.

   and follow this "tab" exception:

   - As will be discussed in the *Alignment Definition* section below, the limitations of the IDE(s) can necessitate use of the space character to implement horizontal whitespace for vertical alignment of code and comment elements.

   If the "tab" requirement cannot be met, *The Style* specifies the use of the space character for all horizontal whitespace.

Note: In the following discussions, the phrase "1 tab or 2 spaces" means use 1 tab character if you meet the above "tab" requirement, otherwise, use 2 space characters. This document was prepared in MS Word which cannot meet this requirement, so the examples herein implement horizontal whitespace using only the space character.

3. *Indentation Definition*

   Indentation is the skipping of horizontal whitespace at the beginning of a line, before text begins. Indentation has levels which are the amounts of whitespace to skip. When using tabs, 1 indentation level = 1 tab. When using spaces, 1 indentation level = 2 spaces. In the examples below, indentation is shown with a green highlight.

4. *Alignment Definition*

   Alignment is the use of horizontal whitespace within a line (after any indentation) to create left or right justified columns of elements across different lines. In the typical coding IDE, where the tab character is a user defined fixed width of whitespace (meeting the "tab" requirement in the *Whitespace Definition*), *The Style* specifies the use of the space character to achieve such vertical alignment (implements the "tab" exception in the *Whitespace Definition*). This is because most coding IDEs lack the tab stops needed to make tab-based vertical alignment work properly.

   However: If *The Team* is using an IDE that provides dynamic tab stops[2] (aka elastic tab stops[3]) then, depending on how well the feature is implemented, they can use the tab character for such vertical alignment when implementing *The Style*. For example, some versions of Visual Studio implement a form of dynamic / elastic tab stops. However, when the tab size is changed on a per-user basis, the dynamic / elastic tab stops may not work properly. Do adequate testing before deciding to suspend the "tab" exception in the *Whitespace Definition*.

5. *Comment Style*

   In *The Style*, whitespace and sometimes text alignment are used to visually delineate comment text from code text. Currently, only comments using the single-line comment delimiter, //, are discussed; multi-line comments, using /* and*/ delimiters, are not addressed.

   - Line Comment Style

     This is a comment that stands alone on a line. The comment delimiter begins the line (after any appropriate indentation) and is followed by 1 space as separation from the subsequent comment text.

```
// In a line comment, there is 1 space after the comment delimiter
```

   - In-line Comment Style

     This is a comment that follows a statement of code text on the same line.

     - **Block** In-Line Comment Style

       This is an in-line comment that is part of an aligned block of such comments, i.e. it has immediately preceding and/or following lines also with in-line comments. Since it uses alignment, only spaces, not tabs, are used for alignment whitespace, per the "tab" exception specified above.

       - In the longest code text line of the block, the in-line comment delimiter is preceded by 2 spaces.

- In all other code text lines of the block, the in-line comment delimiter is preceded by the required number of spaces to align their comment delimiters with that of the longest code text line of the block.
- The comment delimiter is followed by 1 space.

```
a_long_code_statement;       // Long: 7 spaces before & 1 space after the "//"
the_longest_code_statement;  // Longest: 2 spaces before & 1 space after the "//"
a_short_one;                 // Shortest: 17 spaces before & 1 space after the "//"
```

- **Isolated** In-Line Comment Style

  This is an in-line comment that is not part of a **Block** In-Line Comment, i.e. is without immediately preceding or following lines also with in-line comments. With such an in-line comment, the comment delimiter is preceded by 1 tab or 2 spaces. The comment delimiter is followed by 1 space.

```
a_code_statement;  // Use 1 tab or 2 spaces before & 1 space after the "//"
```

However: **An Isolated In-Line Comment adopts the Block In-Line Comment Style when readability is improved** (*programmer's judgment*). This again involves alignment so spaces are used such that the comment delimiter is 2 spaces past the end of the preceding or following code text line, whichever is longer. The comment delimiter is followed by 1 space.

```
a_very_very_long_preceding_code_statement;
a_code_statement;                            // this is a comment
a_very_long_following_code_statement;
```

6. *Code Block Style*

   Code blocks are code text statements that are grouped together. Braces, {}, are used to enclose the body code for Definition Statements (e.g. of functions, classes, namespaces, etc.), Control Statements (e.g. of if, else, while, do while, for, switch etc.) and Initialization Statements (e.g. of arrays).

   In The Style, braces are always used to enclose the body code, even when such use is not specifically required by the compiler (e.g. a code block consisting of a single body code statement). Reasons for doing so are [1] maintaining a consistent "look", [2] explicitly controlling code scope, and [3] using the "code fold on bracket" ability in some IDEs.

   - Definition Statement Brace Style

     The brace indent format used here is called the "*Allman*" style[4]. The opening and closing braces stand alone on their own line and are vertically aligned, both with the beginning of the definition statement and with each other. The body code text within the braces is indented 1 tab or 2 spaces (1 level) from the definition statement. A blank line of whitespace is used to visually separate different definition statements at the same indentation level.

```
class SomeClass
{
  public:
  int someDataMember;

  void someMemberFunction()
  {
    someMemberCode;
    ...
  }
}

class SomeOtherClass
{
  a_code_statement;
  ...
}
```

This format was chosen for function, class, and namespace code blocks because they are relatively long. The format allows the reader to "sight" down the left "face" of the code block and clearly see the beginning and ending brace delimiters of the code block plus the definition statement to which the code belongs, since they all "line up".

- Control Statement Brace Style

  The brace indent format used here is called the "*GNU*" style[5]. The opening and closing braces stand alone on their own line and are vertically aligned with each other but are indented 1 tab or 2 spaces (1 level) relative to the beginning of the statement. The body code text within the braces is indented a further 1 level. After the closing brace, the next line of code follows immediately and is outdented 1 tab or 2 spaces (1 level). *Braces are used even when the body code has a single statement.*

```
switch (token)
  {
    case 5:
      {
        doSomething;
        break;
      }
    ...
  }
while (conditionalExpression1)
  {
    foo();
    bar();
  }
if (conditionalExpression2)
  {
    x = 5;  // use braces even though not required by the compiler
  }
else if (conditionalExpression3)
  {
    y = 7;
    someOtherStatement;
  }
else
  {
    doSomethingElse;
  }
```

This format was chosen for control code blocks because they are relatively short compared to definition statement code blocks. The braces surrounding the body code block are indented because they have a different scope compared to the control statement and they are both "lined up" to provide a visual code block "start/end" cue. This means the reader can quickly visualize the control statements that are at the same scope (e.g. "switch ➔ while ➔ if ➔ else if ➔ else") by observing their alignment without any interrupting "brace clutter".

- Array Initialization Statement Brace Style

With an array initialization statement, the initial values of elements in the array are listed between braces and are delimited by commas. In this format, there is never a space between each value and its following comma delimiter (e.g. "... 1, 2,..." and not "...1 , 2 , ..."). For convenience in the discussion below, I will refer to each such value and its following comma delimiter (if any) as an element of the array.

When the initialization of an array is short (*programmer's judgment*) , the start and end braces are on the same line as the initialization statement. In such cases, the starting brace, {, is followed by 1 space and the ending brace, }, is preceded by 1 space. The separation between each element is 1 space.

```
int someVariable[5] = { 1, 2, 3, 4, 5 };  // Short, single line, initialization statement
```

When the initialization of an array is long (*programmer's judgment*), then 1 or more blocks are created having vertically aligned columns of elements within each block. This is done by breaking the array elements into a series of lines having an *appropriate* indentation for each block and with *appropriate* whitespace alignment to arrange the elements into columns for the block. As described in the *Alignment Definition* section above, such whitespace consists of spaces, not tabs. The braces follow the "*Allman*" style.

The following formats were chosen to make it easier to see more information and to visualize patterns that may exist in a long array initialization statement.

Appropriate block indentation starts at 1 tab or 2 spaces (1 level), relative to the opening and closing braces, for the first-level block and increases by a further 1 tab or 2 spaces for each next-level block. The block indentation is the starting point for creating element columns within that block. Vertically adjacent blocks can also be differentiated by insertion of a blank line of whitespace between the blocks.

Note: Element alignment is used within each block but not across blocks. The columns in one block do not necessarily align with the columns of an adjacent block.

Appropriate whitespace to create columns depends on the data type:

- Numerical Element Column Style
  (for these types: integer, float, double, long double, and Boolean when expressed as 1 or 0)
  - Elements in the columns are *right*-justified (alignment on the right).
  - In all columns of a block except the first, the longest element is preceded by 1 space. Shorter elements are preceded by enough spaces to achieve *right*-justification.
  - In the first column of a block, the longest element meets any indentation that is present. Shorter elements are preceded by enough spaces to achieve *right*-justification.

```
int someIntArray[] =   // Long initialization with alignment of elements to visualize array structure in blocks
{
  -3,  9,  0,  // First level block indents 1 level from opening brace
  11, -2,  3,  // Right justify numbers, including sign where needed
   5, 10, 55,
     0,  0,   0, 0,  0,  0,   0,   0,  30, 30,
     4, 12, -12, 0, 12, 12, -24, -24, 130,  0,
    -4, -7,   7, 0, 12, 12,  14,  14,  30, 30  // Second level block indents 2 levels from opening brace
};
----------------------------------------------------------------------------------------------
```

```
float someFloatArray[] =
{
  11.5,     1.1, 123.123, -12.7, 120.0,
   1.1, 123.123,   -12.7, 120.0,  11.5
};
----------------------------------------------------------------------------------------------
bool someBoolArray1[] =  // Boolean when expressed as 1 or 0
{
  0, 1, 1, 0,
  1, 0, 1, 0
};
```

- Text Element Column Style
  (for these types: string, character, pointer when expressed as variable addresses, and Boolean when expressed as "true" or "false")
  - Elements in columns are *left*-justified (alignment on the left).
  - In all columns of a block except the last, the longest element is followed by 1 space. Shorter elements are followed by enough spaces to achieve *left*-justification.

```
string someStringArray[] =
{
  "left",   "right",   "top",     "bottom",  "front, "back",     // Left justify text
  "inside", "outside", "forward", "reverse", "go",   "stop now"
};
----------------------------------------------------------------------------------------------
char someCharArray[] =
{
  'a', 'b', 'c',
  'd', 'e', 'f'
};
----------------------------------------------------------------------------------------------
int *somePointerArray[] =
{
  &lensFactor, &proportion, &tranSpeed,  &pan,
  &frontUpX,   &backUpX,    &frontDownX, &backDownX,
  &frontUpY,   &backUpY,    &frontDownY, &backDownY,
  &frontUp,    &backUp,     &frontDown,  &backDown
};
----------------------------------------------------------------------------------------------
bool someBoolArray2[] =   // Boolean when expressed as true or false
{
  false, true,  true, false,
  true,  false, true, false
};
```

7. *Template Declaration Style*

   In *The Style*, the "template" keyword and associated "typename" keyword(s), used for generic functions and classes, stand alone on one line. The following line has the function or class template declaration. This is followed by the function or class template definition which uses the "*Allman*" style braces.

```
template <typename T>
T someFunction(T x, T y)
{
  someCodeStatement;
  ...
}

template <typename T>
class someClass
{
  private:
    T *ptr;
  ...
}
```

8. *Operator Style*

In *The Style*, the following types of operators have 1 space before and 1 space after the operator to separate them from the items they operate on: arithmetic (+, -, *, /, %), assignment (=, +=, -= , *=, /=, %=, <<=, >>=, &=, |=, ^=), comparison (== , != , < : , > : , <= , >=), logical (&& , || , !), bitwise (&, |, ^, ~, <<, >>).

```
int x = 5;
```

Sometimes the expression after an assignment operator can be long (*programmer's judgment*). In such cases, the statement is split across multiple lines. A line break occurs after the assignment operator then you indent 1 tab or 2 spaces (1 level) relative to the beginning of the statement. Further line breaks occur as needed after operators. HOWEVER, operators can also stand alone on a line, with the indicated indentation, if that improves readability (*programmer's judgment*). Regardless, for such multiline expression statements, the ";" stands on its own line, with the indicated indentation, to conclude the statement.

```
int x =
  someLongNamedVariable +  // Usually preferred
  someOtherLongNamedVariable
  ;

int x =
  someLongNamedVariable  // Alternative (especially for the division operator)
  /
  someOtherLongNamedVariable
  ;
```

The pointer/dereference operator (*) and the address operator (&) have no space after the operator. The increment/decrement operators (++ and --) have no space after the operator in the prefix form (++x) and no space before the operator in the postfix form (x--).

```
int *somePointer;
somePointer = &someVariable;
++x;
```

9. *Parentheses Style*

Parentheses are used in function definitions and calls, control statements, grouping expressions and C-like type casting. They can be nested and their use combined in various ways. To organize the discussion, I will distinguish between parentheses whose content is short vs. long. Specifically, short vs. long is a *programmer's judgment* about the amount of text between parentheses. It is styled on one line vs. on multiple lines, respectively.

General *Parentheses Style* Requirements:

There is no space between "(" and a following variable or literal nor between a ")" and a preceding variable or literal. There is 1 space after a comma delimiter. Wherever operators are found, the *Operator Style* applies. This means a parenthesis is separated from an operator by 1 space. Consecutive parentheses of the same type and on the same line are separated by 1 space, e.g. "))". This happens when parentheses are nested.

Function Definition

- **Short Parentheses Content**

  The statement is on 1 line. There is no space before or after the opening parenthesis and before the closing parenthesis. Here is a simple example:

```
void FunctionWithShortParameterList(int Parameter1, int Parameter2)
```

Here is a more complex example involving nested parentheses:

```
void performOperation(int a, int b, int (*operation)(int, int) )  // function pointer passed as an argument
```

- **Long Parentheses Content**

  The statement is on multiple lines. Indent the parentheses 1 tab or 2 spaces (1 level) on their own lines relative to the beginning of the statement. Content is indented a further tab or 2 spaces (1 level) relative to the parentheses. Line breaks occur after the comma delimiter. Column creation follows the Text Element Column Style. Here is a simple example:

```
void FunctionWithLongParameterList  // Define the function
  (
    int Parameter1, int Parameter2,  // Note:  could have used a single column (programmer's judgment)
    int Parameter3, int Parameter4,
    int Parameter5, int Parameter6
  )
{
  bodyCodeHere;
}
```

Here is a more complex example involving nested parentheses:

```
void performOperation
  (
    int Parameter1, int Parameter2,
    int Parameter3, int Parameter4,
    int (*operation)(int, int)
  )
```

Function Call Statement

- **Short Parentheses Content**

  The statement is on 1 line, same as the Function Definition Statement with **Short Parentheses Content**. Here is a simple example:

```
FunctionWithShortParameterList(Argument1, Argument2);
```

Here is a more complex example involving nested parentheses:

```
FunctionWithShortParameterList(ReturnsInt1(a), ReturnsInt2(b) );
```

- **Long Parentheses Content**

  The statement is on multiple lines, same as the Function Definition Statement with **Long Parentheses Content** except there are no braces and the semicolon immediately follows the closing parenthesis. Here is a simple example:

```
FunctionWithLongParameterList  // Call the function
  (
    Argument1, Argument2,  // Note:  could have used a single column (programmer's judgment)
    Argument3, Argument4,
    Argument5, Argument6
  );
```

Here is a more complex example involving nested parentheses:

```
FunctionWithLongParameterList
  (
    ReturnsInt1(a),
    ReturnsInt2(b),
    ReturnsInt3(c),
    ReturnsInt4(d)
  );
```

Control Statement

- **Short Parentheses Content**

  The statement is on 1 line. There is 1 space before the opening parenthesis.
  Here is a simple example:

```
while (x == y)
```

  Here is a more complex example involving nested parentheses:

```
while (x == someFunction(y) )
```

- **Long Parentheses Content**

  The statement is on multiple lines. Indent the parentheses 2 tabs or 4 spaces (2 levels) relative to the beginning of the statement. Content is indented a further tab or 2 spaces (1 level) relative to the parentheses. Line breaks occur after operators. Column creation follows the Text Element Column Style. Here is a simple example:

```
if
    (
      !boolFirst && boolSecond  &&  // Line break occurs after an operator
      boolThird  && !boolFourth &&
      boolFifth  && boolSixth
    )
  {
    someCodeStatement;
  }
```

  Here is a more complex example involving nested parentheses:

```
while
    (
      (a + b) == (c + d) &&
      (x + y) != z
    )
  {
    someCodeStatement;
  }
```

Grouping Expression

Assignment statements are used for examples.

- **Short Parentheses Content**

  The statement is on 1 line. Here is a simple example:

```
int result = (a + b) / (c + d);
```

  Here is a more complex example involving nested parentheses:

```
int result = ( (a + b) * 3) / ( (c + (d / 5) ) );
```

- **Long Parentheses Content**

  The statement is on multiple lines. A line break occurs after the assignment operator then you indent 1 tab or 2 spaces (1 level) relative to the beginning of the statement to start the assignment. Line breaks can occur after a closing parenthesis and after operators (preferably those outside of parentheses). Try to avoid breaking the grouping parentheses content between multiple lines. The ";" stands on its own line, with the indicated indentation, to conclude the statement.
  Here is a simple example:

```
int result =
  (someLongVariableName + someOtherLongVariableName)
  /
  (anotherLongVariableName + stillAnotherOtherLongVariableName)
  ;
```

  Here is a more complex example involving nested parentheses:

```
int result =
  ( (someLongVariableName + someOtherLongVariableName) * 3)
  /
  ( (anotherLongVariableName + stillAnotherOtherLongVariableName) / (a + b) )
  ;
```

C-Like Type Casting

The *Parentheses Style* for C-like type casting uses 1 space before the beginning parenthesis and no space between the ending parenthesis and the name of the variable being cast.

```
someIntegerVariable = (int)charVariable;  // cast to integer
```

10. *Directive Style*

Directives are instructions to the preprocessor.

For conditional directives (e.g. #if, #elif, #else, #ifdef, #ifndef, #endif), the Directive Style is similar to the "*GNU*" Control Statement Brace Style but applied to the conditional directives and their content. Since there are no braces, the body code text after a conditional directive is indented 1 tab or 2 spaces (1 level). An inclusion directive (#include) is formatted like any other code statements, as is a macro directive (#define) that creates a symbol representing a value (but not when the symbol represents code - see below).

The conditional directive and operator combinations of #if defined / #elif defined and #if !defined / #elif !defined are strongly preferred over the "contracted" conditional directives like #ifdef and #ifndef. This is to conform to the normal C language "if" and "else if" keywords and not the preprocessor unique #ifdef and #ifndef plus the absence of #elifdef and #elifndef conditional directives. Like the "*GNU*" Control Statement Brace Style, the sequence of associated conditional directives, i.e. an #if block, (#if, #elif, ..., #else, #endif) are at the same indentation level. A blank line of whitespace is used to visually separate the end of an #if directive from the next #if block or other code.

```
#if defined QUALITY
  #define SIGNAL 1
  #if !defined TEST
    #define STACK 200
  #elif defined SPECIAL
    #define STACK 150
  #else
    #define STACK 100
  #endif
#else
  #define SIGNAL 0
  #if STACKUSE == 1
    #define STACK 100
  #else
    #define STACK 50
  #endif
#endif

#if DLEVEL == 0
  #define STACK 0
#elif !defined SPECIAL
  #define STACK 100
#elif DLEVEL > 5
  #include "test.h"
  display( debugptr );
#else
  #define STACK 200
#endif
```

A macro directive that defines a symbol which represents code is more complex. In such cases, place the #define keyword on its own line. Then, on the next line, indent 1 tab or 2 spaces (1 level) and define the macro symbol. On subsequent lines, define the code that the preprocessor will use to replace the macro symbol wherever it appears elsewhere in the code. The code formats in the *Code Block Style* apply, and remember to use the required preprocessor line continuation character, \, which is formatted with the **Block** In-Line Comment Style.

```
// This macro defines a function so use Definition Statement Brace Style and
// align the line continuation character, "\", using the Block In-Line Comment Style.
#define                                                               \
  printfloatx(Name, Variable, Spaces, Precision, EndTxt)              \
  {                                                                   \
    Serial.print(F(Name) );                                           \
    char S[(Spaces + Precision + 3)];                                 \
    Serial.print(F(" ") );                                            \
    Serial.print(dtostrf( (float)Variable, Spaces, Precision, S) );  \
    Serial.print(F(EndTxt) );                                         \
  }
```

1. https://github.com/este-este/A-Code-Formatting-Style
2. https://en.wikipedia.org/wiki/Tab_stop
3. https://nick-gravgaard.com/elastic-tabstops/
4. https://en.wikipedia.org/wiki/Indentation_style#Allman_style
5. https://en.wikipedia.org/wiki/Indentation_style#GNU