3 Few Hints on the C++ Coding Style

Written by Prof. Bogusław Cyganek © March 2021

The main goal of this note is to provide some practical hints on a coding style, which belongs to the group of "soft" skills and colleagues' guidelines, rather than formal requirements, although some companies and organizations also stress importance of the latter.

Why such coding styles are important? As mentioned in my textbook [1], the high level code we write should be readable and understandable for their authors and colleagues involved in software productions, in the first place. Certainly the code must be also correct, to let the computer to perform exactly what was intended. Hence, a proper and consistent programming style increases code readability and in consequence influences the way we and others understand what we wish the machine to do for us. Apparently, lack of such rules and chaotic way of coding can greatly diminish this ability (for the curious visit the obfuscated programming contents).

Vast majority of the professional programmers have their own programming styles and in this short note we'll try to help in choosing the most convenient style for the beginners. Although such styles can be considered in a general scope, in this note we will focus mostly on the C++ language.

3.1 Be Flexible, Care for Readability and Code Correctness

In the role of an introduction here are some advices. These are the topics on the coding styles as well as few hints:

- Naming conventions – to choose or not to choose ones?
- Coding style – examine few and pick the one(s) that make you most comfortable with code readability.
- Prefer being agile, flexible and pragmatic over formal and restrictive.
- Stick to your style but remain open for others (no "holy wars", whose style is "better", etc.).
- Do care about code readability which leads to code correctness.
- Do not sacrifice readability and correctness over code tricks and too preliminary optimization (usually based on guesses rather than real measurements, frequently not necessary at all).
- Write useful comments – be terse, concentrate on explaining the ideas, assumptions and algorithms, rather than writing what is done (avoid trivial as `++i; // increase i by 1`)

3.2 Digressions about Coding Styles with Examples

As mentioned, a coding style can help with increasing code readability. As always there is no single best way, with many pros and cons for each – we avoid such useless discussions, providing only some glimpses that maybe will help to choose (or change) your own programming style.

3.2.1 Naming Conventions

Some companies and organizations foster special styles of object naming, such as using specific prefixes, suffixes, etc. For example, in the *Introduction to Programming with C++ for Engineers* book the Taligent notation is commonly used for classes and class members (summarized on pg. 233 in [1]). Why? Because I started to use them many years ago and since then I try to employ them consistently; they increase readability of my code, at least in my eyes.

The subject gained much attention – there are dozens of posts and hundreds of opinions. However, let's agree that the shorter and more expressive a name, the better. Here are some further hints on choosing names for objects, as well as on naming conventions:

- If you are multi-lingual always prefer English names (unless forced otherwise by your employer; but if so, then consider changing your job). They are common, well understood, no special letters, no conjugation, no declination, highly expressive, common in computer science, internationally accepted.
- Prefer simple, well understood names; avoid complicated, too long, as well as similar names, such as `sqr` which can be easily confused e.g. with `sqrt`.
- Multiple-word identifiers, such as `PropellerSpeed`, or `propellerSpeed`, or `propeller_speed`. The options are as follows:
    - Delimiter-separated words – such as `find_if` popular in Standard Library and many C++ libraries;
    - Letter case-separated words – such as `GetRoots`;

  Use whichever fits you, or both. However, be sure to devise as short as possible but meaningful names.
- Feel comfortable with a devised name; if you don't, then rethink not only the name but the entire function, variable, class or a component you try to name (e.g. is `SystemManager` a good name for a class? Isn't it too "general" or trying to express "everything"? What is it 'real' role in your system?).

For more discussion see publications [5][7][4][3]. If you are meticulous about each aspect of naming, number of function parameters, programming by contract, clean code, etc. the book by Steve McConnell will fulfill your desires [7]. For all, young, as well as more experienced programmers, the book by Thomas and Hunt is highly recommended [6].

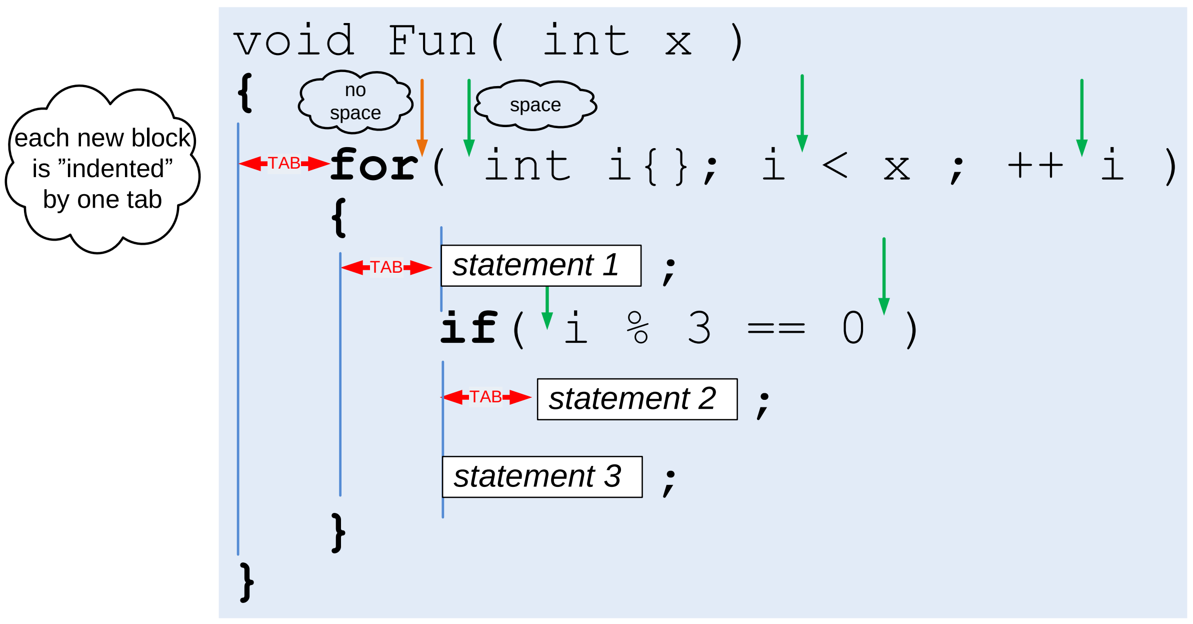
3.2.2 Indentations and Spacing

Indentation means a specific horizontal placement of a line of text. Depending on the habits and used editor, it is usually achieved either by (i) entering TAB(s) symbol(s) or simply by (ii) a sufficient number of the SPACE symbol.

In C++ (also in C)[1] the role of indentation is to increase code readability by emphasizing program structure, i.e. to show the hierarchy of the control structures, such as `for` loops and `if` conditionals (i.e. what is 'external', what is 'internal'), as well as to properly delineate space of the `struct` and `class` definitions.

[1] In some languages, such as Python, indentation is the only way to define the control structures (there are no special symbols such as `{}`).

Figure 3-1 shows the indentations style used in the book *Introduction to Programming with C++ for Engineers* [1]. It is equivalent to the Allman's style except for the space placement in the control statement (the orange arrow). That is, each control statement such as `for` or `if` is not separated by a space from its nearest opening parenthesis symbol `(`. However, one space follows this opening `(`. More importantly each block is delimited by the braces `{ }`, where each brace symbol is placed in *a separate and exclusive line*. All statements inside a block are indented by 1 TAB, such as *statement 1*, `if` and *statement 3*. However, each new control statement introduces new 1 TAB indentation, even if no `{ }` are necessary, such as *statement 2*.



```
void Fun( int x )
{
	for( int i{}; i < x ; ++ i )
	{
		statement 1 ;
		if( i % 3 == 0 )
			statement 2 ;
		statement 3 ;
	}
}
```

Figure 3-1. C++ indentation style used in the book by Cyganek [1]. Except for the space placement in the control statement equivalent to the Allman's style. Each block is delimited by the braces `{ }`, each placed in a separate line. All statements inside a block are indented by 1 TAB, such as *statement 1*, `if` and *statement 3*. However, each new control statement introduces new 1 TAB indentation, even if no `{ }` are necessary, such as *statement 2*.

There are many other indentation styles and their mutations. The three very characteristic to C++ are shown in Listing 3-1. Since C++ has its roots in C, popularity of the K&R style (middle column in Listing 3-1) is not a big surprise. Its characteristic feature is placement of the opening brace `{` at the end of the same line as the control statement. However, an exception is `{` opening the function body definition. On the other hand, the closing brace `}` always occupies a single line at the end of its associated block, and is aligned with its control statement. Each statement inside the block is identically indented (by 1 TAB or an equivalent number of SPACEs). It was presented in the book by Kernighan and Ritchie [8] and became very useful in the era of text-only terminals, which allowed for a very limited number of visible lines and characters in a line on a screen. Then it has been adopted and modified by Bjarne Stroustrup, and therefore remains very popular also in the C++ community.

On the other hand, the Whiteshmiths' style, presented in the right column of Listing 3-1, places `{}` in separate lines and *already indented*. However, the inner statements are not further right indented – instead they are aligned with their enclosing braces `{}`. This style was used in the Microsoft books on programming with C and C++.

Listing 3-1. Comparison of few indentation styles, commonly used in C++ programming.

From the Cyganek's book [1] (~à la Allman, BSD)

Each { and } is in a separate line, aligned with the control statement. Specific to the book is connection of the control statement with the opening (, such as `for(` , rather than `for (` as in the Allman style. Characteristic to this style are also separating spaces, such as `++ i`, rather than `++i`.

```
struct SortStrategy
{
  using IntVec = std::vector< int >;

  //////////////////////////////////////////
  // Bubble sort (in situ)
  //////////////////////////////////////////
  // INPUT:   vec - vec of int to be sorted
  // OUTPUT:  none
  void operator()( IntVec & vec )
  {
    auto swapped { true };

    for( auto i { 1 }; swapped && i < vec.size(); ++ i )
    {
       swapped = false;
       for( auto j = vec.size() - 1; j >= i; -- j )
       {
          if( vec[ j - 1 ] > vec[ j ] )
          {
             std::swap( vec[ j ], vec[ j - 1 ] );
             swapped = true;
          }
       }
    }
  }

};
```

à la K&R (C, Linux) – Stroustrup (C++)

In the Kernighan & Ritchie style, each { opening a block is in the same line as the control statement, whereas closing } is in a new line. However, opening { of a function is in a separate line, whereas { opening a `struct` or `class` is in the same line (a modification by B. Stroustrup). Fosters lower number of lines.

```
struct SortStrategy_KR {  // variant B. Stroustrup

  using IntVec = std::vector< int >;

  //////////////////////////////////////////
  // Bubble sort (in situ)
  //////////////////////////////////////////
  // INPUT:   vec - vec of int to be sorted
  // OUTPUT:  none
  void operator()( IntVec& vec )
  {
     auto swapped { true };

     for (auto i {1}; swapped && i<vec.size(); ++i) {
        swapped = false;
        for (auto j = vec.size() - 1; j >= i; --j) {
          if (vec[ j-1 ] > vec[ j ]) {
              std::swap( vec[ j ], vec[ j-1 ] );
              swapped = true;
          }
        }
     }
  }

};
```

à la Whitesmiths (Windows programming)

Similar to the Allman style, i.e. the opening brace { is put on the next line just after the associated control statement, such as `for` below. However, contrary to Allman's, {} are already indented, while the inside statements are on the same level as their enclosing { }.

```
struct SortStrategy_Whitesmiths
{
  using IntVec = std::vector< int >;

  //////////////////////////////////////////
  // Bubble sort (in situ)
  //////////////////////////////////////////
  // INPUT:   vec - vec of int to be sorted
  // OUTPUT:  none
  void operator()( IntVec & vec )
  {
   auto swapped { true };

   for (auto i {1}; swapped && i<vec.size(); ++i)
      {
      swapped = false;
      for (auto j = vec.size() - 1; j >= i; -- j)
         {
         if (vec[ j-1 ] > vec[ j ])
             {
             std::swap( vec[ j ], vec[ j-1 ] );
             swapped = true;
             }
         }
      }
  }

};
```

Some practical recommendations to set up the indentations.

- For indentation prefer TAB symbol, rather than SPACE(s), since it requires less typing, cannot be mistyped (was it 3 or 4 spaces?), as well as TAB can be easily converted to the spaces if necessary.
- Set TAB to be equivalent to 3-8 SPACEs (in the book [1] a TAB equivalent of 4 SPACEs is used).
- Know you tools and make profit of their properties (read manuals and experiment, make the editor you use on a daily basis comfortable for you).

3.3 Conclusions

For more information on coding standards see the *NL: Naming and layout rules* section in the *Cpp core guidelines* document [4]. Examine also the excellent books by Thomas & Hunt [6], as well as by McConnell [7], and certainly [1]. Then choose the style that best fits your needs and try to be consistent. However, do experiment and don't be afraid of changing to other style in the future if you feel it better suits your needs. The style presented and shared in my recent book [1] worked for me well for over twenty years.

Read and continuously learn to increase your expertise, be flexible, care for readability and code correctness! Have fun!

References

[1] Cyganek B.: *Introduction to Programming with C++ for Engineers*. Wiley, 2020.
[2] https://en.wikipedia.org/wiki/Indentation_style#Allman_style
[3] https://www.stroustrup.com/bs_faq2.html#layout-style
[4] https://github.com/isocpp/CppCoreGuidelines/blob/master/CppCoreGuidelines.md
[5] https://en.wikipedia.org/wiki/Naming_convention_(programming)
[6] Thomas D., Hunt A.: *The Pragmatic Programmer*, 20th Anniversary Edition, Addison Wesley 2020.
[7] McConnell, Steve: *Code Complete: A practical handbook of software construction*. Redmond, WA: Microsoft Press, 2004.
[8] Kernighan B., Ritchie C.: *C Programming Language*, Prentice Hall, 1988.