C++ Guideline

Alexandre Quenon

2017-08-24

Contents

Part I.

Pure conventions

Introduction to conventions

Conventions are rules that are necessary for working as a team with a unified style. Hence, conventions are often seen as a pure aesthetic discussion. And often it is the case.

However, conventions can also be used to improve the clarity of something. If every member of the team uses a specific style for a specific thing, this thing can be immediately identified by everyone. As a consequence, conventions are also a mean of improving the efficiency.

In this part, we present guidelines that are purely conventional. Most of the rules are generally admitted as "good" by experienced programmers. And indeed, when we see them, we are like: "this is so obvious!" But, well, it is not. We need to see them, to taste them, to try them to be convinced and to adopt them. So, let us do that.

1. Naming stuff

The purpose of the current chapter is to provide generic as well as specific rules in order to have an easy-to-read code. This if probably one of the most important part, especially when working in teams, because it helps you to track bugs and to understand each other's code.

1.1. Generic guideline

Let us start with some generic rules that can be applied in the whole code, whatever we are working on a class or a function.

Rule 1: Use explicit names that express ideas, concepts.

Compare phone_book with number_vector [1, 2].
C++ is object-oriented programming. Ideas and concepts have a higher level of abstraction.

Rule 2: Prefer underscore to camel case for readability.

Compare number_of_element with numberOfElement [2].

Rule 3: In general, avoid capital letters.

It is simply easier and faster to write non-capital letters.

Rules 2 and 3 are followed by the Standard Library.

1.2. Specific guidelines

Let us now review some rules that are intended for specific part of the codes, such as variables or functions. Of course, we may apply them for other things but, in general, we will mainly face the situation for the specified subject.

1.2.1. Variables

Rule 4: Do not embed the type in the name.

It lowers the abstraction level. Moreover, embedding the type in the variable name is a nightmare of maintenance and consistency. Suppose you decide to change your `int` to a `double`, how many names will you have to change to maintain the consistency? [1, 2]

Examples: do not use something like `char* pc_buffer` or `int i_number`.

The previous rules could also be applied to functions (about their return type, for instance). However, this is typically done for variables by programmers who are used to weak-typed languages.

1.2.2. User-defined types (`classes` and `enums`)

Rule (optional) 5: Capitalize names for user-defined types.

Examples: Widget, My_Super_Class.

For your information, rule 5 is not followed by the Standard Library (e.g., `std::vector` or `std::string`). However, to my opinion, it is easier for the programmer to identify variables, return types…that are of built-in type or part of the STL from their own classes or enumerations. Of course, it is an exception to rule 3. Eventually, following this rule or not is your choice.

1.2.3. Macros

Rule 6: Use all-capital names for macros.

Examples: CRC_CODE, BUFSIZE.

TBC Macros are more C-like programming. As a consequence, their use should be seen as something unusual. Therefore, to draw readers' attention on any used macro, rule 6 is not named optional. This is an intentional exception to rule 3 because using macros should be annoying.

Part II.

C++ Programming Techniques

Bibliography

[1] Robert C Martin. *Coder proprement*. Pearson Education France, 2009. ISBN: 978-2-7440-4104-4.

[2] Bjarne Stroustrup. *The C++ Programming Language*. 4th ed. Addison-Wesley, 2013. ISBN: 978-0-321-56384-2.