Member of the **PICANOL** GROUP

μC & DSP TEAM

C++ Coding Guide

Jens Jonckheere

July 12, 2018

We are authors. And one thing about authors is that they have readers. Indeed, authors are responsible for communicating well with their readers. The next time you write a line of code, remember you are an author, writing for readers who will judge your effort.

– Robert C. Martin

Contents

Foreword

The different coding styles of programmers can lead to confusion and time-loss when other team-members need to understand the code, for example to add features or to debug. This document presents a common coding style for our team, so that code written by colleagues also looks familiar to us. This document also gives some tips on writing clean code and gives an overview of some essential C++ knowledge.

Chapter 1

Coding Style

1.1 Initialization

1.1.1 Fundamental Types

An object of fundamental type can be initialized in many different ways.

```
uint8_t uninitialized_var; // default intialization
uint8_t zeroed_var1(0);    // direct initialization
uint8_t zeroed_var2 = 0;   // copy initialization => preferred
uint8_t zeroed_var3{0};    // direct-list-initialization
uint8_t zeroed_var4 = {0}; // copy-list-initialization
```

Listing 1.1: Initialization of fundamental type objects

This style guide prefers copy initialization for fundamental type objects because it also works in C, it's what we are accustomed to.

Note that for fundamental types default initialization actually means uninitialized.

By leaving the parentheses empty, objects are value initialized. For fundamental types this means that all bits are made zero.

```
uint8_t zeroed_var5();
uint8_t zeroed_var6{};
uint8_t zeroed_var7 = {};
```

Listing 1.2: Value initialization of fundamental type objects

1.1.2 Arrays

Arrays can be initialized in different ways.

```
uint8_t uninitialized_array[3]; // default initialization
uint8_t array1[3]{0, 1, 2};     // direct-list-initialization
uint8_t array2[3] = {0, 1, 2};  // copy-list-initialization => preferred
```

Listing 1.3: Initialization of arrays

Default initialization of an array results in default initialization of every element.

This style guide prefers copy-list-initialization for arrays over direct-list-initialization because it also works in C.

Note that the individual array elements are initialized by copy-initialization from the initializers specified in the braced-init-list.

If the number of initializer clauses is less than the number of elements, or the initializer list is completely empty, the remaining elements are value-initialized.

```
uint8_t zeroed_array1[3]{};
uint8_t zeroed_array2[3] = {};
```

Listing 1.4: Value initialization of arrays

The preferred way of zeroing an array is shown in listing 1.5 because this also works in C.

```
uint8_t zeroed_array3[3] = {0};
```

Listing 1.5: Zeroing arrays

1.1.3 Classes

Chapter 2

Clean Code

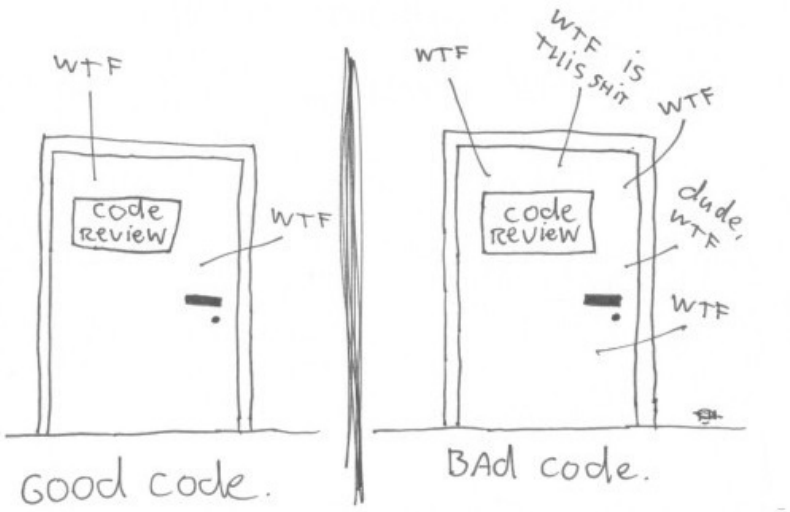

Figure 2.1: The only valid measurement of code quality: WTFs / minute [2]

2.1 Variables

2.1.1 Names

2.1.2 Horizontal Alignment

2.1.3 Vertical Alignment

2.1.4 Vertical Whitespace

2.2 Functions

2.2.1 Small, Small, Small!

2.3 Classes

2.3.1 Small, Small, Small!

2.3.2 Cohesion

Chapter 3

Essential C ++ Knowledge

3.1 List Initialization

Prefer list initialization over direct initialization for class types.

List initialization has some advantages:

- it can be used for class types and also for fundamental types and arrays, so it can be seen as a uniform initializer;
- it avoids the most vexing parse.

```
int number;        // uninitialized
int number{};      // same as "int number = 0"
int array[100];    // uninitialized
int array[100]{}; // same as "int array[100] = {0}"
```

Listing 3.1: Value initialization syntax

3.2 Value Initialization of Fundamental Types

In C++, local variables and class members of fundamental types (int, float, int*, ...) do not get initialized by default. Value initialization can be used to zero-initialize fundamental types.

```
int number;        // uninitialized
int number{};      // same as "int number = 0"
int array[100];    // uninitialized
int array[100]{}; // same as "int array[100] = {0}"
```

Listing 3.2: Value initialization syntax

3.3 Object Initialization

3.3.1 Default Initialization

This is the initialization performed when a variable is constructed with no initializer.

```
T object;                   // constructed on stack
T* object_ptr = new T;      // constructed on heap
```

Listing 3.3: Default initialization syntax

The effects of default initialization are:

- if T is a [non-POD (until C++11)] class type, the constructors are considered and subjected to overload resolution against the empty argument list. The constructor selected (which is one of the default constructors) is called to provide the initial value for the new object;
- if T is an array type, every element of the array is default-initialized;
- otherwise, nothing is done: the objects with automatic storage duration (and their subobjects) are initialized to indeterminate values.

Notes

If no user-defined constructors of any kind are provided for a class type (struct, class, or union), the compiler will always define an empty default constructor as an inline public member of its class.

Default initialization of non-class variables with automatic and dynamic storage duration produces objects with indeterminate values (static and thread-local objects get zero initialized).

If T is a const-qualified type, it must be a class type with a user-provided default constructor.

References cannot be default-initialized.

3.3.2 Direct Initialization

Initializes an object from an explicit set of constructor arguments.

```
T object(arg1, arg2, ...); // constructed on stack
T* object_ptr = new T(arg1, arg2, ...); // constructed on heap
T(other) // prvalue temporary by functional cast
T(arg1, arg2, ...) // prvalue temporary with a parenthesized expression list
Class::Class() :
    member(args, ...) // base or a non-static member init in initializer list
{...
```

Listing 3.4: Direct initialization syntax

The effects of direct initialization are:

- if T is a class type, the constructors of T are examined and the best match is selected by overload resolution. The constructor is then called to initialize the object;
- otherwise, if T is a non-class type but the source type is a class type, the conversion functions of the source type and its base classes, if any, are examined and the best match is selected by overload resolution. The selected user-defined conversion is then used to convert the initializer expression into the object being initialized;
- otherwise, standard conversions are used, if necessary, to convert the value of other to the cv-unqualified version of T, and the initial value of the object being initialized is the (possibly converted) value.

Notes

Direct-initialization is more permissive than copy-initialization: copy-initialization only considers non-explicit constructors and non-explicit user-defined conversion functions, while direct-initialization considers all constructors and all user-defined conversion functions.

In case of ambiguity between a variable declaration using the direct-initialization syntax (with round parentheses) and a function declaration, the compiler always chooses function declaration. This disambiguation rule is sometimes counter-intuitive and has been called the most vexing parse.

3.3.3 Copy Initialization

3.3.4 List Initialization

3.3.4.1 Aggregate Initialization

3.3.5 Stack

3.3.6 Heap

3.3.7 test

Chapter 4

FreeRTOS

- Prefer inter-task queues over mutexes.

Bibliography

[1] cppreference.com. https://en.cppreference.com/w/, 2018.

[2] Robert C. Martin. *Clean Code*. Prentice Hall, 2009.