Coding Standards in C/C++

Giuseppe Lipari, Fabio Checconi
http://retis.sssup.it/~lipari

Scuola Superiore Sant'Anna – Pisa

October 1, 2009

Outline

Outline

Why it is Important to Write Clean Code?

Readability: people, and even yourself will have to understand what you write, it should be written to be easy to follow, properly documented and as uniform as possible.

Why it is Important to Write Clean Code?

Readability: people, and even yourself will have to understand what you write, it should be written to be easy to follow, properly documented and as uniform as possible.

Teamwork: many contributors may need to work on the same code, they'll appreciate the conformance to an agreed standard to better understand an modify it.

Why it is Important to Write Clean Code?

Readability: people, and even yourself will have to understand what you write, it should be written to be easy to follow, properly documented and as uniform as possible.

Teamwork: many contributors may need to work on the same code, they'll appreciate the conformance to an agreed standard to better understand an modify it.

Code re-use: clean and manageable code is easier to reuse.

Why it is Important to Write Clean Code?

Readability: people, and even yourself will have to understand what you write, it should be written to be easy to follow, properly documented and as uniform as possible.

Teamwork: many contributors may need to work on the same code, they'll appreciate the conformance to an agreed standard to better understand an modify it.

Code re-use: clean and manageable code is easier to reuse.

Code mainteinance: maintaining code involves understanding it after years of changes done by other people; the better is to understand or modify, the easier is the maintainer's work.

Coding standards

- An example of rules for writing clean code can be found here:
  `http://www.possibility.com/Cpp/CppCodingStandard.html`.
- We will summarize some of them in the following.

Spaghetti-like Programming

Writing programs with a complex control structure using unstructured constructs (e.g., goto's, exceptions, `longjmp()` etc.) can render these programs unreadable and unmanageable. This is especially true for assembly languages.

```
        int i = 0;
loop:
        f(i);
        i++;
        if (i < LOOPS)
                goto loop;

        /* ... */
        for (i = 0; i < LOOPS; i++)
                f(i);
```

Current Practice

Many open source projects require that their contributors adhere to some coding standard. Some examples:

- the Linux kernel [1] has strict rules to deterimine if code can be accepted or not. Rules are described in the `Documentation/CodingStyle` in the source tree, and are enforced checking patches with `scripts/checkpatch.pl`.
- The Free Software Foundation has its own set of coding standards, see `http://www.gnu.org/prep/standards/`.
- On BSD systems, check `man 9 style`.
- The Hungarian notation (we will not use it).

Outline

Linux Coding Standards

Any piece of code, to be accepted into the Linux kernel, has to conform to many rules, described in various files in `Documentation/`, mostly concerning the coding style of the source code, some common practices, that have to match the rest of the codebase, and the submission process.

Coding Style

The main rules (and some justifications for them) are:

- tabs are 8 characters wide. Using smaller tabs allows too dense code, that cannot be parsed easily by the average human reader.

Coding Style

The main rules (and some justifications for them) are:

- tabs are 8 characters wide. Using smaller tabs allows too dense code, that cannot be parsed easily by the average human reader.
- Lines are (at most) 80 characters wide. Again, longer lines are difficult to parse. The need for more than 80 columns should ring some bells, as it, in the average case, stems from:

Coding Style

The main rules (and some justifications for them) are:

- tabs are 8 characters wide. Using smaller tabs allows too dense code, that cannot be parsed easily by the average human reader.
- Lines are (at most) 80 characters wide. Again, longer lines are difficult to parse. The need for more than 80 columns should ring some bells, as it, in the average case, stems from:
  - identifiers that are too long;

Coding Style

The main rules (and some justifications for them) are:

- tabs are 8 characters wide. Using smaller tabs allows too dense code, that cannot be parsed easily by the average human reader.
- Lines are (at most) 80 characters wide. Again, longer lines are difficult to parse. The need for more than 80 columns should ring some bells, as it, in the average case, stems from:
  - identifiers that are too long;
  - too many intentation levels;

Coding Style

The main rules (and some justifications for them) are:

- tabs are 8 characters wide. Using smaller tabs allows too dense code, that cannot be parsed easily by the average human reader.
- Lines are (at most) 80 characters wide. Again, longer lines are difficult to parse. The need for more than 80 columns should ring some bells, as it, in the average case, stems from:
  - identifiers that are too long;
  - too many intentation levels;
  - improper (or lack of) use of functions.

Coding Style

The main rules (and some justifications for them) are:

- tabs are 8 characters wide. Using smaller tabs allows too dense code, that cannot be parsed easily by the average human reader.
- Lines are (at most) 80 characters wide. Again, longer lines are difficult to parse. The need for more than 80 columns should ring some bells, as it, in the average case, stems from:
  - identifiers that are too long;
  - too many intentation levels;
  - improper (or lack of) use of functions.
- Multiple statements should not be put into a single line. This, along with the 8 char-tab rule allows an easy recognition of basic blocks and statements.

Tabs and Line Width: an Example

```
static int move_one_task(struct rq *this_rq, int this_cpu, struct rq *busiest,
                         struct sched_domain *sd, enum cpu_idle_type idle)
{
        const struct sched_class *class;

        for (class = sched_class_highest; class; class = class->next)
                if (class->move_one_task(this_rq, this_cpu, busiest, sd, idle))
                        return 1;

        return 0;
}
```

Coding Style: Placing Braces

- Brace placement is not random:

Coding Style: Placing Braces

- Brace placement is not random:
  - functions have their opening brace in the line after the definition.

Coding Style: Placing Braces

- Brace placement is not random:
  - functions have their opening brace in the line after the definition.
  - `for, if, while, switch` statements have their opening braces on the same line of the statement.

Coding Style: Placing Braces

- Brace placement is not random:
  - functions have their opening brace in the line after the definition.
  - `for, if, while, switch` statements have their opening braces on the same line of the statement.
  - Except for `do ... while` and `else` statements, the closing brace is on a line on its own.

Coding Style: Placing Braces

- Brace placement is not random:
  - functions have their opening brace in the line after the definition.
  - `for, if, while, switch` statements have their opening braces on the same line of the statement.
  - Except for `do ... while` and `else` statements, the closing brace is on a line on its own.
- The idea is that functions have to be clearly identified at first sight; brace placement should help identifying code blocks. There is no need to waste an additional line for statements that open a new block (except functions).

Coding Style: Using Spaces

- Use a space after keywords.

Coding Style: Using Spaces

- Use a space after keywords.
- Do not use spaces between a function name and the parenthesis enclosing its arguments.

Coding Style: Using Spaces

- Use a space after keywords.
- Do not use spaces between a function name and the parenthesis enclosing its arguments.
- Place the `*` denoting a pointer close to the pointer name instead of the type name.

Coding Style: Using Spaces

- Use a space after keywords.
- Do not use spaces between a function name and the parenthesis enclosing its arguments.
- Place the `*` denoting a pointer close to the pointer name instead of the type name.
- Use spaces around binary arithmetic operators.

Coding Style: Using Spaces

- Use a space after keywords.
- Do not use spaces between a function name and the parenthesis enclosing its arguments.
- Place the `*` denoting a pointer close to the pointer name instead of the type name.
- Use spaces around binary arithmetic operators.
- Do not use spaces before/after unary operators and around `.` and `->`.

Coding Style: Naming

- Function and variable names should help the reader understanding the code, at the same time avoiding redundancy and excessive code bloat. A good balance between properly chosen identifiers and good comments should be achieved.

Coding Style: Naming

- Function and variable names should help the reader understanding the code, at the same time avoiding redundancy and excessive code bloat. A good balance between properly chosen identifiers and good comments should be achieved.
- Global variable names should be descriptive, both to avoid namespace collisions and to ease reading the code.

Coding Style: Types

- Typedefs should not be used as shortcuts for longer type names. They should be avoided as much as possible as they often hide from the reader the true meaning of the type.

Coding Style: Types

- Typedefs should not be used as shortcuts for longer type names. They should be avoided as much as possible as they often hide from the reader the true meaning of the type.
- In particular, typedefs for structures should be avoided, as they hide the fact that the referred type is an aggregate one.

Coding Style: Types

- Typedefs should not be used as shortcuts for longer type names. They should be avoided as much as possible as they often hide from the reader the true meaning of the type.
- In particular, typedefs for structures should be avoided, as they hide the fact that the referred type is an aggregate one.
- However typedefs can be used for completetly opaque objects or to better specify the meaning of integer types.

Coding Style: Functions

- Functions should be short and simple. They should do just one thing, hopefully well.

Coding Style: Functions

- Functions should be short and simple. They should do just one thing, hopefully well.
- A function should always fit one or two 80x24 terminal screens; for complex functions, or functions with many (i.e., more than 3) indentation levels, this may be even be too much.

Coding Style: Functions

- Functions should be short and simple. They should do just one thing, hopefully well.
- A function should always fit one or two 80x24 terminal screens; for complex functions, or functions with many (i.e., more than 3) indentation levels, this may be even be too much.
- Don’t use too many local variables (rule of thumb: can you remember all of them and their meaning while reading the code?)

Outline

What is a Module

- A module is a software component with well-defined interfaces toward the external world (i.e., its users and its building blocks).

What is a Module

- A module is a software component with well-defined interfaces toward the external world (i.e., its users and its building blocks).
- The interface of a module is defined by:

  data types: all the data manipulated by the module have a common structure, and generally this structure is available to the module only (think about C++/Java classes);

  functions: the module users invoke the services it provides using a well-defined, and hopefully documented, set of functions.

Interface vs. Implementation

- The interface of a module is whatever defines *how the module has to be used.* Data types and function declarations compose the interface of a module.
- The implementation of a module is the *definition of its behavior.* In other words, the implementation specifies how the module does all the things exposed via its interface.

C++ classes and modules

- Typically, a module only contains one class
- One header file that contains the class declaration (.h or .hpp or .hh or .hxx)
- One source file that contains the class implementation (.cpp or .cc or .cxx)
- The file name should be named after the class name
- Example: class `SynchBuffer` implementation goes into file *SynchBuffer.cpp*, and its interface/specification into *SynchBuffer.hpp*

Global variables

- Global variables must not be abused, since they pollute the global identifier namespace and create dependencies and side effects that become hard to track and control as the code grows.
- Remember:
  - The variable is *declared* in an header file by using keyword `extern`. The compiler does not allocate memory

Global variables

- Global variables must not be abused, since they pollute the global identifier namespace and create dependencies and side effects that become hard to track and control as the code grows.
- Remember:
  - The variable is *declared* in an header file by using keyword `extern`. The compiler does not allocate memory
  - The variable is *defined* in the source file (the compiler allocates memory)

Global variables

- Global variables must not be abused, since they pollute the global identifier namespace and create dependencies and side effects that become hard to track and control as the code grows.
- Remember:
  - The variable is *declared* in an header file by using keyword `extern`. The compiler does not allocate memory
  - The variable is *defined* in the source file (the compiler allocates memory)
- Example:
  - *myfile.h*

```
...
extern int myvar; /**< declaration */
...
```

  - *myfile.cpp*

```
int myvar; /**< definition */
```

Guards for Include Files

- Header files should specify interfaces only. They may be included multiple times during the compilation of a single execution unit, and this can result in compilation errors or subtle bugs if no precautions are taken.

```
#ifndef __MYINCLUDE_FILE_H__
#define __MYINCLUDE_FILE_H__

#include "pippo.h"

extern int global_var;

int myfun(int a, double b);
char *get_string(int size);

#endif
```

Outline

Commenting the Code

- Comments should say everithing that is not obvious from the code, and should help understanding it; one should focus on *what* the code does.
- If one needs a comment to say *how* the code does something, maybe that code is too involved and can be rewritten in some more readable way.
- Choosing proper identifiers and using proper function splitting can help readability more than a lot of comments.

Outline