

Coding Style Guideline for the HAEC simulator

Mario Bielert and Thomas Ilsche

July 21, 2015

Abstract

This document contains a summary of the guideline for the HAEC simulator. This aims to giving you a recommendation for a style which will lead to an unified appearance and hence an easier start for everyone. Also it should increase the maintainability.

1 About this document

1.1 Keywords

In this document contains several keywords, which aim to giving you an idea, how strong a recommendation is.

- **SHOULD** it's recommended, that you do something, but there might be reasonable exceptions.
- **SHOULD NOT** it's recommended, that you don't do something, but there might be reasonable exceptions.
- **MUST** you are strongly encouraged to always do the following.
- **MUST NOT** you are strongly encouraged to never do the following.

1.2 Changes or additions to certain guidelines

To minimize reformatting of the code base, changes and additions to style guidelines **MUST** be discussed with all developers and **SHOULD** only be done for good reasons.

1.3 Uncovered topics

If there's missing a guideline or rule for a certain topic, you **SHOULD** stick with the unwritten consent, which is already used in the project. And you **SHOULD** add a guideline about it.

2 Appearance

2.1 Automatic formatting

In the project folder is a format file for clang-format (in version 3.4). You SHOULD use clang-format with this file to format the source code. The following assumes your distribution uses clang-format-3.4, but that might as well just be clang-format.

2.1.1 Set up in Kdevelop

In Kdevelop under *Settings* → *Configure Kdevelop...* go to *Source formatter*. Under *Formatter* choose *Custom Script Formatter*. Click on *New* and name it *haec_sim*. Then click on *Edit...* and enter the following under *Command*.

```
clang-format-3.4 -style=file $FILE > $TMPFILE
```

Click on *OK* and you're done.

2.1.2 Set up in clion

Go to *File* → *Settings* → *Tools* → *External Tools*. Click on the green +-sign and choose an appropriate name, e.g. "clang-format" a group, e.g. "Formatting". Check "synchronize files after execution" and uncheck "open console".

Program: clang-format-3.4

Parameters: -i -style=file "\$FilePath\$"

Working directoy: \$ProjectFileDir\$

Now in *Keymap* you can add a shortcut, e.g. Ctrl+Alt+Shift+F. You should save the file before reformatting.

2.1.3 Usage in command line

Provided that clang compiler is installed on your machine, you can also format source files from the command line. For example, to format file code.hpp inplace, using the provided .clang-format (in the haec_sim root directory), you can use the following command.

```
clang-format-3.4 -style=file -i code.hpp
```

2.2 Length of a line

You SHOULD NOT exceed 80 characters per line, including white spaces.

2.3 Indentation

You MUST indent using 4 spaces. You MUST NOT use tabs.

2.4 Spaces

You SHOULD surround every binary operator and the ternary operator with spaces. You SHOULD NOT add spaces around parenthesis, except for the following rules. You SHOULD add spaces around C++ keywords, if applicable, including *if*, *for*, *while*. You SHOULD add a space after commas.

2.4.1 Examples

```
int a = b + c;
bool d = a > 0 ? true : b - c > 0;

printf("%s", "Hello World\n");
std::abs(-5);
for (int a=0; a<c.size(); a++);
if (do_foo());
while (1 == 0);
myclass a(b, c);
sizeof (a);
throw std::runtime_error("Chuck Norris was here.");
```

2.5 Blocks

You MUST indent the contents of the block¹, but you MUST NOT indent the block itself. You MUST place each of the curly braces surrounding the block on a new line with the same indentation level as the parental structure of the block.

2.5.1 Examples

```
void foo()
{
    for (;;)
    {
        do_something();
    }
}

auto lambda = []()
{
    std::cout << "I'm a lambda" << std::endl;
}
```

2.6 Comments

You MUST place comments above the line you want to comment, not behind.

To easily distinguish between Doxygen comments and normal comments, you SHOULD only use the single line comment format `//...` or the doxygen comment format `/** ... */`. This applies even for comments, which doesn't fit on one line.

You SHOULD NOT comment trivial commands. But you SHOULD comment parts, which aren't obvious.

2.6.1 Examples

```
/**
 * a function to convert from foo to bar
 */
```

¹Every code snippet surrounded with curly braces is considered as a block in this context.

```

bar foo2bar(foo f)
{
    // just use the explicit constructor
    return bar(f);
}

// This is a very loooooooooooooooooooooooooooooooooooooooooooooooooooooooooong
// comment, which don't fit on one line.

```

2.7 Name finding

As name finding isn't easy in general, the guidelines only apply to the format of names.

You **MUST NOT** use Hungarian notation or Camel-case in any form. You **SHOULD** use lower case and replace spaces with underscores. You **SHOULD** avoid underscores in class names, instead you **SHOULD** use a **namespace**. You **SHOULD** instead use upper case for global constants. Template parameter **SHOULD** be in CamelCase with starting upper case, but you **SHOULD** avoid to long names.

2.7.1 Examples

```

int PI = 3;
int num_events;
otf2::definition::location def;
class foo
{
    void bar();
    bool is_bar();
    bool has_something_started();
    void start_everything();

    const static int the_answer = 42;
};

```

2.8 Namespaces

You **SHOULD** use **namespaces** to structure your classes and functions.

You **SHOULD NOT** use **using namespace**, especially in header files. Instead you **SHOULD** give the full qualified name. You **SHOULD** consider the use of **using Type**; in generic code.

2.9 Types and signatures

2.9.1 Typedefs vs. using

You **SHOULD** prefer using `MyType = std::string;` over `typedef std::string MyType`, if C++11 is available.

2.9.2 References and Pointers

You **SHOULD** align the ampersand and the asterisk to the left at the type.

2.9.2.1 Examples

```
int* p = nullptr;  
std::string& name;  
char** argv;
```

2.9.3 Const

You SHOULD write the `const` as far left as possible.

2.9.3.1 Examples

```
const std::string& name;  
const char*; // pointer to const char  
char* const; // const pointer to char  
char const*; // DON'T DO THIS - still pointer to const char
```

2.10 Variable declarations

You MUST NOT declare multiple variables of different types on the same line.

2.11 Ownership and locality

You SHOULD prefer creating objects on stack. You SHOULD NOT use owning pointers, prefer smart pointers instead.

You SHOULD declare variables as locally as possible. You MUST not use global variables, instead use the Meyers singleton idiom, this solves the problem of the order in which statics are initialized.

2.12 Folders and file names

You SHOULD create one file for one class. You SHOULD create sub folders to group files together

2.13 Include guards

Include guards SHOULD start with `INCLUDE_` and SHOULD represent the path of the current file.

2.13.1 Example

If you have the file `include\haec_sim\foo_bar\some_header.hpp` an appropriate include guard is `INCLUDE_HAEC_SIM_FOO_BAR_SOME_HEADER_HPP`.