Contents

1 Files

Each file should contain the header included in `HEADER.TXT`, and the author of the file. The header should also provide a description of the content of the file. If you make significant changes to a file consider adding your name to the author list.

1.1 File organization

Each class should be defined a different file, whose name is the class name where the upper case letters are replaced with a underscore (_) and the lower case letter.i (Exception: if the upper case letter is the first letter, no underscore is prepended).

Header files should be placed in `include/dca/`, source files in `src/`. Tests should be placed in `test/unit/`, `test/integration/`, `test/system-level/` or `test/performance/` according to their type. `include/dca/`, `src/` and `test/unit/` should have the same tree structure.

1.2 File Names

Filenames should be all lowercase and can include numbers and underscores (_). Uppercase are allowed for abbreviations and chemical compounds. Header files use the extension `.hpp`, while source files use the extension `.cpp`.

The filenames of the tests are composed by the filename of the object tested and the suffix `_test` is appended. The filename of fake objects used in tests are composed by the prefix `mock_` and the filename of the object that is replaced.

1.3 Header files

All header files should be self-contained. This means that each header can be included without defining anything special. In particular each header file should include all the headers needed and have **#define** guards to prevent multiple inclusion.

If a template or inline function is declared in a `.hpp` file, the function has to be defined in the same file.

The symbol name of the #define guards should be PATH_FILENAME_HPP, where PATH is based on the include directory. For example, the file dca_ethz/include/dca/foo/bar.hpp should have the following guard:

```
#ifndef␣DCA_FOO_BAR_HPP
#define␣DCA_FOO_BAR_HPP
...
#endif␣␣//␣DCA_FOO_BAR_HPP
```

1.4 Includes

Header files should be included with the full path based on the include directory. For example, the file dca_ethz/include/dca/foo/bar.hpp should be included as

```
#include␣"dca/foo/bar.hpp"
```

Even if the included file is located in the same directory as the including file this rule should be obeyed.

For readability it is suggested to use the standard order of includes:

1. Related header
2. C library
3. C++ library
4. Other libraries' .h
5. Your project's .h

In each section the included files should be sorted in alphabetical order.

2 Naming

The names should be as descriptive as possible.

2.1 Type and Class Names

Type and class names should start with an uppercase character and should have a capital letter for each new word. Underscores (_) are not allowed.
The **using** syntax is preferred to **typedef** for type definitions.

2.2 Namespace Names

Namespace names should be one word, lowercase.

2.3 Variable Names

2.3.1 Variable Names

Variable names should contain only lowercase characters and underscores (_). The underscore is not allowed as first or last character.

2.3.2 Class Data Members

Class data members names follows the convention of variable names with a trailing underscore (_).

2.4 Function Names

Function names should start with a lowercase character and should have a capital letter for each new word. Underscores (_) are not allowed.

The only exception to this rule are getter and setter methods for class data members. They follow the naming convention `set_variable` and `get_variable`, respectively (see 4.13).

2.5 Lambda Expressions

Named lambda expressions follow the naming convention for variables:

```
auto my_lambda = [](int i) { return i + 4; };
```

2.6 Macro Names

Macro names should be all uppercase and can include underscores (_). The underscore is not allowed as first or last character.

3 Comments

3.1 Comment Style

Use the `// Comment` syntax. The `/* ... */` syntax is allowed only for unused variables in function declarations.

3.2 Function Comments

Each function, whose name does not fully describe the operations performed should have comments that describe what the function does.

For function parameters whose type is non-const reference or pointer to non-const memory, it should be specified if they are input (In:), output (Out:) or input-output parameters (InOut:).

Example:

```
// Updates foo and computes bar using in_1 .. in_5.
// In: in_3, in_5
```

```
// In/Out: foo
// Out: bar
void computeFooBar(Type in_1, const Type& in_2, Type& in_3,
                   const Type* in_4, Type* in_5, Type& foo,
                   Type& bar);
```

3.3 Variable Comments

No comments, since the name should be self-descriptive.

3.4 Class Comments

Every class should have a short description of what it is and what is does. Comments for public class member functions follow the same rules as general function comments. Comments for private members are allowed, but not mandatory.

3.5 Implementation Comments

Tricky, complicated or important code blocks should have comments before them. Line comments should be separated from the code by 2 spaces. If multiple subsequent lines have a comment they should be aligned.
Example:

```
statement;        // Comment here so the comments line up.
longerstatement;  // Two spaces between code and comment.
```

If you modify a piece of code, also adapt the comments that belong to it if necessary.

3.6 Comments excluded in releases

Comments formatted as `// TODO: Comment`, `// INTERNAL: Comment` and `// REVIEW: Comment` are removed in the releases.

3.7 Punctuation, Spelling and Grammar

Pay attention to correct punctuation, spelling, and grammar. This will improve the clarity and readability of the comments.

4 Formatting

Use the provided clang-format style to format `.hpp` and `.cpp` files.

4.1 Line Length

The length of each line of your code should, in principle, be at most 100 characters. This limit can be exceeded by few characters in special cases.

4.2 Scopes

Do not use scopes for formatting reason.

4.3 Horizontal Spacing

No trailing whitespaces should be added to any line. Use no space before a comma (`,`) and a semicolon (`;`) and add a space after them if they are not at the end of a line.

4.3.1 Parenthesis

Parenthesis should have no internal padding, and one space external padding. The one space external padding is not allowed in the following cases:

- between two opening of two closing parenthesis,
- between the function name and its arguments,
- between a closing parenthesis and a comma (`,`) or a semicolon (`;`).

4.3.2 Binary Operators

The assignment operator should always have spaces around it. Other operators may have spaces around them, but is not mandatory.

4.3.3 Unary Operators

Do not put any space between the unary operator and their argument.

4.3.4 Types

The angle brackets of the templates should not have any external and internal padding. (However, for `C++03` codes the space between two closing angle brackets is mandatory.)

Examples:

```
type*␣var;
type&␣var;

Class1<Class2<type1>>␣object;␣␣//␣OK␣with␣C++11,␣does␣not
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣//␣compile␣with␣C++03.
```

4.4 Vertical Spacing

Use empty lines when it helps the readability of the code, but do not use too many. Do not use empty lines after a brace which opens a scope, or before a brace which closes a scope. Each file should contain an empty line at the end of the file (Some editors add an empty line automatically, some do not) .

4.5 Indentation

Indentation consists of 2 spaces. Don't use tabs in the code.

4.6 Variable Declarations and Definitions

- Avoid to declare multiple variables in the same declaration, especially if they are not fundamental types:

```
int x, y;                           // Not recommended.
Matrix a("my-matrix"), b(size);     // Disallowed.

// Preferred way.
int x;
int y;
Matrix a("my-matrix");
Matrix b(10);
```

- Use the following order for keywords and modifiers in variable declarations:

```
// General type
[static] [const/constexpr] Type variable_name;

// Pointer
[static] [const] Type* [const] variable_name;

// Integer
// int is optional, if a signedness or size modifier is present.
[static] [const/constexpr] [signedness] [size] int variable_name;

// Examples:
static const Matrix a(10);
const double* const d(3.14);
constexpr unsigned long l(42);
```

4.7 Function Declarations and Definitions

The return type should be in the same line as the function name, and the parameters should be on the same line unless they don't fit in it. Add the parameters name also in the declaration of a function and avoid

```
Type␣function(Type1,␣Type2,␣Type3);
```

In function declarations comment the unused parameter names.
(E.g. `Type␣/*␣unused_parameter_name␣*/`)
Examples:

```
Type␣Class::function(Type1␣par1,␣Type2␣par2)␣{
␣␣statement;
␣␣...
}

Type␣LongNameClass::longNameFunction(Type1␣par1,␣Type2␣par2
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣Type3␣par3)␣{
␣␣statement;
␣␣...
}
```

In case of a long list of parameters prefer

```
Type LongNameClass::LongNameFunction(
    Type1 long_name_par1, Type2 long_name_par2,Type3 par3) {
  statement;
  ...
}
```

to

```
Type LongNameClass::LongNameFunction(Type1 long_name_par1,
                                     Type2 long_name_par2,
                                     Type3 par3) {
  statement;
  ...
}
```

4.8 Function Calls

Write the call all on a single line, if the length of the line doesn't exceed the maximum limit. If not possible, wrap the arguments at the parenthesis, or start the arguments on a new line using 4 spaces indent. Use the method which uses the smaller amount of lines.
Examples:

```
Function(par1, par2, par3);

Function(par1, par2,
         par3);

Function(
    par1, par2, par3);
```

4.9 Conditionals

Examples:

```
if (condition)
  statement;
else
  statement;

if (condition) {
  statement;
}
else if (condition2) {
  statement;
}
else {
  statement;
}
```

4.10 Switch statement

Switch statements should always have a default case.
Example

```
switch␣(var)␣{
␣␣case␣0:
␣␣␣␣statement1;
␣␣␣␣statement2;
␣␣␣␣break;

␣␣case␣1:
␣␣␣␣statement1;
␣␣␣␣statement2;
␣␣␣␣break;

␣␣default:
␣␣␣␣statement1;
␣␣␣␣statement2;
}
```

4.11 Loops

Examples:

```
for␣(statement;␣condition;␣statement)
␣␣statement;

for␣(statement;␣condition;␣statement)␣{
␣␣statement1;
␣␣statement2;
}

while␣(condition)
␣␣statement;

while␣(condition)␣{
␣␣statement1;
␣␣statement2;
}

do␣{
␣␣statement;
}
while␣(condition);
```

4.12 Preprocessor Directives

The preprocessor directives are not indented. The hash is the first character of the line.

4.13 Class Format

Public, protected and private keywords are not indented.
Example:

```
class␣Foo␣:␣public␣Bar␣{
public:
␣␣Foo();
␣␣explicit␣Foo(int␣var);

␣␣void␣function();
␣␣void␣emptyFunction()␣{}

␣␣void␣set_var(const␣int␣var)␣{
␣␣␣␣var_␣=␣var;
␣␣}
␣␣int␣get_var()␣const␣{
␣␣␣␣return␣var_;
␣␣}

private:
␣␣bool␣privateFunction();

␣␣int␣var_;
␣␣int␣var2_;
};
```

4.13.1 Constructor Initializer Lists

Examples:

```
//␣When␣everything␣fits␣on␣one␣line:
Foo::Foo(int␣var)␣:␣var_(var)␣{
␣␣statement;
}

//␣If␣the␣signature␣and␣the␣initializer␣list␣do␣not
//␣fit␣on␣one␣line,␣the␣colon␣is␣indented␣by␣4␣spaces:
Foo::Foo(int␣var)
␣␣␣␣:␣var_(var),␣var2_(var␣+␣1)␣{
␣␣statement;
}

//␣If␣the␣initializer␣list␣occupies␣more␣lines
//␣they␣are␣aligned␣in␣the␣following␣way:
Foo::Foo(int␣var)
␣␣␣␣:␣some_var_(var),
␣␣␣␣␣␣some_other_var_(var␣+␣1)␣{
␣␣statement;
}

//␣No␣statements:
Foo::Foo(int␣var)
␣␣␣␣:␣some_var_(var)␣{}
```

4.14 Namespace Formatting

The content of namespaces is not indented. If nested namespaces are used a comment with the full namespace is required after each set of namespace declaration, and when a namespace is closed a comment should indicate which namespace is closed.

Example:

```
namespace␣ns{
void␣foo();
}

namespace␣ns1{
namespace␣ns2{
//␣ns1::ns2::
void␣foo();

namespace␣ns3{
//␣ns1::ns2::ns3::
void␣bar();
}␣␣//␣ns3
}␣␣//␣ns2

namespace␣ns4{
namespace␣ns5{
//␣ns1::ns4::ns5::
void␣foo();
}␣␣//␣ns5
}␣␣//␣ns4
}␣␣//␣ns1
```

5 Other C++ Features

5.1 Pre-increment and pre-decrement

Use the pre-increment (pre-decrement) operator when a variable is incremented (decremented) and the value of the expression is not used. In particular, use the pre-increment (pre-decrement) operator for loop counters:

```
for (int i = 0; i < N; ++i) {
  doSomething();
}
```

The post-increment and post-decrement operators create an unnecessary copy, that the compiler cannot optimize away in the case of iterators or other classes with overloaded increment and decrement operators.

5.2 Alternative Operator Representations

Alternative representations of operators and other tokens such as **and**, **or**, and **not** instead of `&&`, ||, and `!` are not allowed. For the reason of consistency, the far more common primary tokens should always be used.

5.3 Use of const

- Add the **const** qualifier to all function parameters that are not modified in the function body. For parameters passed by value, only add the keyword in the function definition.

```
// Declaration
int computeFoo(int bar, const Matrix& m) {

// Definition
int computeFoo(const int bar, const Matrix& m) {
  int foo = 42;

  // Compute foo without changing bar or m.
  // ...

  return foo;
}
```

5.4 In-class member initialization

In-class member initialization is recommended for members that are *always* initialized with the same value to avoid code duplication. In all other cases it is disallowed.

```
std::string getNameFromId(const int id);

class Customer {
public:
  Customer(const std::string& name = "Doe",
           const std::string& address = default_address_)
    : name_(name), address_(address) {}
  Customer(const int id,
           const std::string& address = default_address_)
    : name_(getNameFromId(id)), address_(address) {}

private:
  static const std::string default_address_;

  std::string name_;
  std::string address_;
  int num_orders_ = 0;
};

const std::string Customer::default_address_ = "Zurich";
```