Contents

1 Files

Each file should contain the header included in `HEADER.TXT`, and the author 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 in lower case.

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 filenames of mock object 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 do not have to require any special condition to be included. 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`, and they should be based on the full path of the tree of the `src` directory of the project. For example, the file `dca_ethz/src/foo/bar.hpp` in project foo should have the following guard:

```
#ifndef FOO_BAR_HPP
#define FOO_BAR_HPP
...
#endif // FOO_BAR_HPP
```

2 Naming

The names should be as descriptive as possible.

2.1 Type and Class Names

Function names should start with an uppercase character and should have a capital letter for each new word.

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

Variables names should start with a lowercase character and should have a capital letter for each new word.

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.

2.5 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.

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.
```

3.6 Comments excluded in releases

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

4 Formatting

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 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;
  ...
}
```

4.7 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);
```

4.8 Conditionals

Examples:

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

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

4.9 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;
}
```

4.10 Loops

Examples:

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

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

while (condition)
  statement;

while (condition) {
  statement1;
  statement2;
}

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

4.11 Preprocessor Directives

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

4.12 Class Format

Public, protected and private keywords are not indented.

Example:

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

```
  ~Foo() {}

  void Function();
  void EmpyFunction() {}

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

private:
  bool PrivateFunction();

  int var_;
  int var2_;
};
```

4.12.1 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) {}
```

4.13 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
```