Flow123d code rules

1 Introduction

This document contain code rules for Flow123d developers. These rules were created to help developers to better understand the code of each other not to make your life harder. They have no absolute priority, but please think twice if you break some of them. Through this document we use `typesetting font` for pieces of code, **bold font** to indicate main points, and *italics* for indroduction of new terms. The sections are ordered from the most important to the less important.

2 Naming conventions

Names are comments. All names has to be meaningfull and has to be specific. If we use good names we nedd less comments. **Think** when you introduce new identificator. It will be with us "forever". Use english names, learn new words by coding :-].

Variables, functions / members, methods have lowercase names with words separated by underscore. This holds also for shortcuts i.e. `read_gmsh` not `read_GMSH`
Example: `n_elements`, `make_partitioning()`, `mesh->n_elements`,

Structures, Classes, Templates, Typenames are in mixed case, the first letter i the word is in uppercase.
Example: `Element`, `WaterLinSys`

Files should contain just one principial class/structure and should have the same name as this class/structure, i.e. `Element.c`.

Preprocessor macros in upper case.

Template parameters are uppercase letters. (C++ only)]

Namespaces in lower case. (C++ only)

Generic variables should have same name as their type.
Example: `solver_solve(Solver *solver,LinSys *lin_sys);`

Plural form of names of collections. Example: `Element elements[];`

Name of class/structure should be avoided in the name of members.
Example: `line.get_length(); // NOT: line.get_line_length();`

Variables with larger scope should have longer names.

No abbreviations in indentificators. At least don't use your own abbreviations !!!

Class members with underscore suffix. Class members should be private and suffixed with underscore. This immediately say that it is not local variable of the method. (C++ only)

3 Code organization rules and guidelines.

Code for clarity rather then efficiancy.

Clean headers. The header files should contain structure/class definitions and function declarations. Function and method definitions should be in the corresponding source file. VERY short (one line) inline methods can be in class definition, larger ones directly after the class definition.

Anti-multiple inclusion mechanism has to be used for every header file. Example:

```
#ifndef Solver_hh  // macro derived form filename is used
#define Solver_hh
 // The rest of the file
#endif
```

Avoid include in headers. Think twice when including from header file. This easily leads to cross inclusion. When you need pointer to some other structure/class you can use declaration instead of includsion of the definition. Example:

```
struct Element;         // declaration of structure Element
struct Mesh {           // definition of structure that ...
    struct Element *elements; //... use pointers to Element
}
```

Function declarations with names. Always named arguments in function decclaration - simply copy the head of function definition. It really help understanding meanig of the arguments. Example:

```
void print_solver_info(FILE *out,struct Solver *solver);
```

Use typedef for every structure/class. You can use the same name since typedefs lives in a separate namespace. Example:

```
struct Solver {
        ...
}
typedef struct Solver Solver;
```

Use `const` keyword for function/method parameters that do not change. It simplify reading and involves optimization. How to use it?

```
const int       i=1;            // constant integer
const double    pi=3.114;       // real constant

void function(const char *const_str, char * const const_ptr)
// const_str is pointer to constant char (string), you can not change *const_str
// const_ptr is constant pointer to char, you cna not change const_ptr

char const *const_char;         //same as const chat *
char const * const const_all;   //constant pointer to constant value
```

Warnings has to be checked and possibly corrected.

Other guidelines:

Use explicit type conversions. Do not rely upon implicit type conversion, use explicit type conversion instead.

Use enums for named constants instead of `#define`. Example:

```
enum { red=0, green=1, blue=2 };  // anonymous enumeration, insted of macro constants
enum Colors {red=0, green=1, blue=2}; // named enum. ; we can define variables
enum Colors my_color; // my_color have named values
```

Do not assume succes of function calls. Program defensively. Check results of function calls, check prerequisities about function parameters.

Avoid complex contitions and statements in conditions. Like:

```
if  ( (i<0) || (i>max) || (i==last) ) { } // too complex (two cond. together)

if(!(file=open(fName,"w"))) { }           // statement in condition
```

Rather use one more line:

```
bool isOutOfArray= (i<0) || (i>max);
bool isRepeatedItem= (i==last);
if  ( isOutOfArray || isRepeatedItem ) { }

file=open(fName,"w");
if (!file) { }
```

Avoid global variables. If any, refer them with leading `::` operator. Example: `::mainWindow.open();` (C++ only)

Avoid overloading of functions and operators unless it brings clear improvements. (C++ only)

Do not overuse inline. Do not use inline method when it calls another function or contain cycle. (C++ only)

Use `bool` type for true/false variables. (C++ only)

Expose by `typedef` template arguments. (C++ only) Example:

```
template <typename T>
class AFiniteElement{
public:
        typedef T ElementShape;  // this provide meaning of template parameter
};
```

4 Current bad habits

- Declaration mess. No more `headre.h` and `structs.h`.
- Explicit nullify functions.
- Assertion tests are not separeted from real code.
- Too small functions (or too big). Minimal function should be about 5 lines of real code and max. two pages of whole code.

5 Formating

Indent 4 spaces nested blocks. Set your editor to insert spaces instead of tabs. Tabs are different in defferent editors, which breaks formating.

Indent every block. Always after you open a block with brace. Indent also statemet after `if`, `for`, `while` etc. Example:

```
void fce() {
    int i;
    for (i=0; i<10; i++)
        if (i>5) {
            printf("Over five.");
        }
}
```

Align when you split the line. Insert an empty line when this is messed with next line. Example:

```
function(param1, param2,
         param3);
sum = a + b + c +
      d + e;
if ( a < b  &&
     b < c) {

    printf();
}
```

6 Debugging and assertions

Debugging is essential. We provide sort of C macros to make easy implementation of assertion tests as prerequisities and postrequisities of functions and strurture/class invariants. These macros are automaticly removed (defined as empty) when the NODEBUG macro is defined. It means normally the debugging is ON. There should be another sort of C++ macros using streams for output, which avoids separate macros for different number of parameters. Note that this problematics is ralated to the problem of system messages, which currently is supported by xprintf macro/function.

CHKERR(i) Check condition `i`, if it is `false` abort the program.

CHKMSG(i,msg) If `i` is `false` print string "msg" and abort.

CHKMSG1(i,msg,par1) Similar, but allows include one parameter into printf before abort. Similar macros up to `CHKMSG3`.

DBGMSG(msg) Print debug message. (only when NODEBUG is not presented)

DBGMSG1(msg,par1) Print debug message with parameter. And similarly up to `DBGMSG3`.

For C++ version (vision):

ERROR_MSG(M) Output stream `M` and abort the program. Should be more general like `xprintf`. Can not be suppressed by `NODEBUG`.

ASSERT(i,M) If conditon `i` is `false` output stream "M" and abort.

This have several advantages. First we take rid of several parameters. Second, one can overload the redirection operator for output/debug stream and implement output for other objects, but still can use the same assertion macro.

7 Documentation

We use *D*oxygen for source documentation. Basic function of Doxygen is to parse sources, collect documentation for every source entity (provided by special comments) and present all in HTML format with powerfull links. Doxygen process only comments begining as `/**` or `/*!` or one line comments `///` or `//!`. Every comment is linked to the next source line and if there is a definition of some entity the comment is linked to this entity. You use comment `///<` to link it to the previous entity, usefull for comments of structure members.

All names of defined entities are automaticaly recognized in comments and linked to their documentation.

In the comment you can use number of formating keywords. Basic are:

`@file` - file name documentation block

`@brief` - gives short description of the entity, appears besides title

`@param` - description of parameter of function. Name of the parameter follows the keyword, then the description. You can specify intention of the parameter [in], [out], [in,out].

`@{ ... @}` - you can group part of the source

`@name` - gives name to the group that follows

`@f$ ... @f$` - instead of ... you can write an inline Latex formula

`@f[ ... @f]` - instead of ... you can write a centered Latex formula

`@todo` - some stuff to do

For further details see :

```
/// project web page
http://www.doxygen.org
/// quick reference sheet
http://www.digilife.be/quickreferences/QRC/Doxygen%20Quick%20Reference.pdf
```

Comments should be explanatory, not formal. This is basic rule for code documentation. If something is obvious for you now, it may not be obvious later and for others. Do not use "clever tricks" unless VERY well documented.

Documentation of classes and structures. Before every class or structure definition (in header file) there should be a documentation block. It should contain brief description of class. And detailed description including implementation ideas, invariants (properties that the class should satisfy) and usage. Through the structure/classes definition there should be description for every data member. Example:

```
//*********************************************************************
/*! @brief Linear system structure accepted by Solver module.
 *
 *  The system is based on PETSc matrix A and vector of RHS (b) and solution (x),
 *  both vectors are build on the regular arrays vx, vb.
 *  CSR storage are optional and generated on demand by LinSysSetCSR.
 */
struct LinSystem
{
        int     size;
        Mat     A;      //!< Petsc matrix of the problem
        Vec     b;      //!< PETSc vector of the right hand side
        Vec     x;      //!< PETSc vector  of the solution
        double  *vx;    //!< Vector of solution
        double  *vb;    //!< RHS vector
        //! @name optional CSR storage
        /// @{
        int             *i;     //!< i-vector for CSR storage
        int             *j;     //!< j-vector for CSR storage
        double  *a;             //!< a-vector for CSR storage
        /// @}
};
typedef struct LinSystem LinSystem;
```

Documentation of method and functions. Before every method/function definition (in a source file) there should be a documentation block containing: brief description and detailed description which includes description of in-parameters and out-parameters and return value. You should also mention *prerequisities* i.e. any consideration about incoming parameters. There should be also *postrequisities* i.e. properties of out-parameters and return value. Idealy pre- and post- requisities are tested in the function by assertion checks. Example:

```
//=============================================================================
/*! @brief Solves a given linear system.
 *
 * Call user selected internal or external solver.
 * @param[in] solver solver structure to use
 * @param[in,out] system linear system to solve, conatains also result
 *
 * @pre Initialized solver. Assembled system.
 * @post Valid solution.
 */

void solve_system( struct Solver *solver, struct LinSystem *system )
```

Documentation of files. Every file should begin with documentation block. (Have to be specified...) Example:

```
/*! @file
 * @brief       Unified interface to various linear solvers
 *
 * The only internal (linked) solver is PETSC KPS which is already interface
 * to the number of direct and iterative solvers. Further several external
 * solvers are supported: MATLAB, ISOL (due to Pavel Jiranek) and
 * GM6 (due to Miroslav Tuma)
 */
```

8 File extensions - mixing C and C++

The actual Flow123d is written in pure C, but the essential utility `ngh` is written in C++. The aim is rewrite Flow123d also into C++ and include the functionality of `ngh` during this transition we can not avoid mixing of C and C++ code. To keep an order the `*.h` and `*.c` should be pure C sources. The `*.hh` and `*.cc` should be C sources with some C++ features (i.e. mixing sources). And finally `*.hpp` and "`*.cpp` are pure C++ sources. During transition the number of mixed sources should be as small as possible.

9 Performance guidelines

Code for clarity rather then efficiancy.

Performance note about function/method call. Every function call cost at least passing the parameters and the call itself. Virtual method calls take about 3 memory access more. On modern processors a (indirect) function call break some optimisations. Result: Try to make functions reasonably large to make calls less costly. Use inline for small functions. Avoid small virtual methods (these can not be inlined - obviously).

Good optimisiation should also simplify design.