Chapter 1

Coding conventions and C++-style

This document provides an overview of the general coding conventions that are used throughout `oomph-lib`. Knowledge of these conventions will greatly facilitate the use of the library. Contributors to the library are expected to adhere to these standards.

Note that following our move to GitHub we have started to use `clang-format` to automatically format our sources (but only those in the `src` directory). The formatting rules are contained in the `.clang-format` file in the `oomph-lib` home directory. To run this on your own sources you need at least version 10.0.0 of `clang-format`. Note that some of the code shown below would be changed slightly by `clang-format` but this concerns mainly minor whitespace adjustments and the beauty of automatic formatting means that you don't have to worry about these too much.

1.1 Naming conventions

1.1.1 File names

All C++ source files end with the standard extensions $*$.h and $*$.cc.

- $*$.h: Contains the class definitions and any inline functions.
- $*$.cc: Contains all non-inline member functions that can be compiled once and for all. This includes
  - member functions of classes that do not have any template parameters
  - member functions of templated classes for which it is known a priori which instantiations are required. Examples are classes that are templated by the spatial dimension. In this case we're unlikely to require instantiations for any values other than 0,1,2 and 3.
- $*$.template.cc: Contains any non-inline member function of templated classes. This file must be included (together with the corresponding $*$.h file) when a specific instantiation of a templated class is required. For instance, most specific `Mesh` classes are templated by the element type and the mesh writer can obviously not predict which element types his/her specific mesh is going to be used with.

1.1.2 General variables

- General variables are all lowercase
- Variables that contain multiple words contain underscores to separate them, e.g.
  ```
  FiniteElement* surface_element_pt;
  ```

1.1.3 Classes

- Classes start with capital letter, e.g.
```
class Shape
```

- If the class name contains multiple words, the first letter of any subsequent word also gets capitalised, e.g.
```
class AlgebraicElement
```

  Note: No underscores in class names.

1.1.4 Private data and access functions to private data

Use a capital first letter for private data, and the all-lowercase equivalent for the access functions. Examples:

- This is a declaration for a private data member:
```
private:

  /// Pointer to boundary node
  Node* Boundary_node_pt;
```

- Here are two public access functions to the private data member:
```
public:

 /// Access to boundary node (const version)
 Node* boundary_node_pt() const {return Boundary_node_pt;}

 /// Access to boundary node
 Node*& boundary_node_pt() {return Boundary_node_pt;}
```

Note: Do **not** use public data – ever! Make it private and and provide an access function – even if it seems "perfectly obvious" at the time of writing the class that the internal storage for the data item is "never going to be changed".

1.1.5 Pointers

- Pointers and access functions to pointers are identified explicitly by the postfix *_pt* to the variable names, as shown in the previous examples.

1.1.6 Access functions to containers

Many classes have member functions that provide access to data in private containers (e.g. vectors); they are usually accompanied by a member function that returns the number of entries in that container. Naming conventions:

- Use singular for the access function to the container, i.e.
```
/// Return pointer to e-th element
FiniteElement* element_pt(const unsigned& e);
```

  rather than `elements_pt(...)`

- Use a prefix 'n' for the access function for the number of entries in the container, i.e.
```
/// Total number of elements
unsigned nelement();
```

  Notes: (i) No underscore between the "n" and the container's name; you can then use the underscore for the name of the local variable that stores the value, e.g. `unsigned n_element = my_mesh_pt->nelement();` (ii) No trailing `_pt` in the function that returns the number of objects in the container.

1.1.7 Template parameters

- Template parameters are all caps, e.g.
```
template<unsigned DIM>
class NavierStokesEquations
{
  public:
  [...]

};
```

- Sometimes it is necessary to expose the template parameter to make it accessible to the user. This should be done by adding a public `static const` copy of the parameter to the class's member data, using the prefix `TEMPLATE_PARAMETER_` , so the template parameter `DIM` in the above example would be exposed like this:
```
public:
```

```
/// Publicly exposed template parameter
static const unsigned TEMPLATE_PARAMETER_DIM = DIM;
```

Note the exposure of template parameters is optional and is only implemented when we needed it. Feel free to modify existing code yourself if required – `oomph-lib` is open source!

1.1.8 Use descriptive function/variable names

- Make sure you choose descriptive names for functions and variables, even if the names become long.

1.2 Layout etc.

1.2.1 Position of include statements

- Place include statements at the beginning of each file.

1.2.2 Layout of blocks

- Braces on separate lines (unless the content is extremely short)

```
for (unsigned i=0;i<10;i++)
  {

   [...]

   std::cout « "doing something" « std::endl;

   [...]

  }
```

1.2.3 Indentation

- Indentation of blocks etc. should follow the emacs standards.

1.2.4 Layout of functions, classes, etc.

- Precede all functions by a comment block, enclosed between lines of '==='

```
// ============================================================
/// \short (add \short to make sure that multi-line descriptions
/// appear in doxygen's short documentation. Include lists with items
/// - first item
/// - second item
///   - first item of sublist
///   - second item of sublist
///   . //end of sublist
/// . //end of main list
// ============================================================
void SomeClass::some_function()
{
 for (unsigned i=0;i<10;i++)
  {
   std::cout « "doing something" « std::endl;
  }
 }
```

Note the triple slash "///" in the comment block that preceeds the function definition – comments contained in such lines are automatically extracted by doxygen and inserted into the code documentation.

1.2.5 The oomph-lib namespace

- `oomph-lib` is contained in its own namespace, `oomph`, to avoid clashes of class names with those of other (third-party) libraries. If there is no danger of name clashes, the entire `oomph` namespace may be imported at the beginning of each driver code by placing the statement

```
using namespace oomph;
```

at the beginning of the source code (after the included header files). Any additions to the library (this includes the instantiation of templated `oomph-lib` classes inside a driver code!) must be included into the `oomph` namespace by surrounding the code by

```
namespace oomph
```

```
{

   // Additions to the library go here...
   [...]

}
```

1.2.6 Namespace pollution

- To avoid namespace pollution, the namespace `std` **must not** be included globally in any header files. The statement
  `using namespace std;`
  may only be used in driver codes, in $*$.cc files, or inside specific functions in a $*$.h file.

1.2.7 Layout of class definitions and include guards.

Here is an example of a complete header file, including include guards and library includes.

```
#ifndef OOMPH_SOME_CLASS_HEADER     // Assuming that the file is
#define OOMPH_SOME_CLASS_HEADER     // called some_class.h

// Include generic oomph-lib library
#include "generic.h"

// Add to oomph-lib namespace
namespace oomph
{

// ============================================================
/// Waffle about what the class does etc.
///
// ============================================================
template<class T>
class SomeClass : public SomeBaseClass
 {
   public:

    /// Constructor: Pass coefficients n1 and n2
    SomeClass(const unsigned& n1, const T& n2) : N1(n1), N2(n2)
     {}

    /// Access function to coefficient
    inline unsigned n1() const
     {
      return N1;
     }

    /// Access function to other coefficient
    inline T& n2() const
     {
      return N2;
     }


   protected:

      /// Coefficient
      unsigned N1;

   private:

      /// Second coefficient
      T  N2;

  };

 }
 #endif
```

- Order of public/protected/private may be reversed but the declarations should always be explicit (even though everything is private by default).
- Note the prefix `OOMPH_*` in the include guard. This is to avoid clashes with include guards of other libraries.

1.3 Debugging etc.

1.3.1 The PARANOID flag and error handling

- Implement optional validation routines, self-tests, and other sanity checks via conditional compilation, using the compiler flag PARANOID, so that the relevant statements are only activated if `-DPARANOID` is specified as a compilation flag for the C++ compiler. If errors are detected, a meaningful diagnostic should be issued, by throwing an `OomphLibError`. If the code is compiled without the PARANOID flag, all sanity checks are bypassed – good for the overall execution speed, bad for error handling... The user can choose.

  Here's an example:

```
// Has a global mesh already been built?
if(Mesh_pt!=0)
 {
  std::string error_message =
   "Problem::build_global_mesh() called,\n";
  error_message += " but a global mesh has already been built:\n";
  error_message += "Problem::Mesh_pt is not zero!\n";

  throw OomphLibError(error_message,
                      OOMPH_CURRENT_FUNCTION,
                      OOMPH_EXCEPTION_LOCATION);
 }
```

- `oomph-lib` also has an object that allows warning messages to be issued in a uniform format. Here's an example of its use:

```
// Was it a duplicate?
unsigned nel_now=element_set_pt.size();
if (nel_now==nel_before)
 {
  std::ostringstream warning_stream;
  warning_stream  «"WARNING: " « std::endl
                  «"Element " « e « " in submesh " « imesh
                  «" is a duplicate \n and was ignored when assembling "
                  «"global mesh." « std::endl;
  OomphLibWarning(warning_stream.str(),
                  OOMPH_CURRENT_FUNCTION,
                  OOMPH_EXCEPTION_LOCATION);
 }
```

1.3.2 Range checking

- Most access functions that provide indexed access to a private container, do, in fact, access a private STL vector. Explicit range checking for these (frequent!) cases can be avoided by changing to container to Vector class instead. Vectors performs automatic range checking, if the `generic` library is compiled with the `RANGE_CHECKING` flag, i.e. if `-DRANGE_CHECKING` is specified as a compilation flag for the C++ compiler. **Note:** While it is generally a good idea to compile with `PARANOID` while developing code, `RANGE_CHECKING` is **very expensive** and is therefore activated via a second independent flag. We only tend to active this flag as a last resort, typically to track down particularly stubborn segmentation faults.

1.3.3 Self test routines

- Every sufficiently complex class should come with its own
  `unsigned self_test()`

  routine which returns 1 for failure, 0 for successful test.

1.4 Other conventions

1.4.1 Const-ness

- Use const wherever applicable (arguments, member functions,...)
- Always provide const and non-const overloaded subscript operators.
- Example:
  `// Return i-th coordinate of Point`

```
 double& operator[](const unsigned& i){return x[i];}

// Return i-th coordinate of Point -- const version
const double& operator[](const unsigned& i) const {return x[i];}
```

1.4.2 Only use int if a variable can actually take negative values

- Just as the name of a variable gives some indication of its likely use, its type does too. For instance this code fragment

```
// Create a counter
int counter=0;
```

  immediately raises the question why the programmer anticipates circumstances in which the counter might be negative. Are negative values used to indicate special cases; etc? If the name of the variable was chosen correctly (i.e. if the variable really is used as a counter) then

```
// Create a counter
unsigned counter=0;
```

  is much clearer and therefore preferable, even if the two versions of the code would, of course, give the same result.

1.4.3 Only use "pass by reference"

- Arguments to functions should only be passed "by reference", not "by value". Use "pass by constant reference" if you want to ensure the const-ness of any (input) arguments.
- To "encourage" this behaviour, most `oomph-lib` objects have (deliberately) broken copy constructors and assignment operators, making a "pass by value" impossible. The only exceptions are cases in which we could see a good reason why a fully-functional, non-memory-leaking copy/assignment operator might be required.

1.4.4 Provide fully-functional or deliberately-broken copy constructors and assignment operators

- For the reasons mentioned above, "passing by value" is discouraged and we have only implemented copy constructors for very few classes. To make the use of C++'s default copy constructor impossible (as their accidental use may lead to serious memory leaks) all classes should either have a deliberately-broken copy constructor or provide a "proper" implementation (as in the case of `oomph-lib's Vector` class). The same applies to assignment operators.

  With the advent of C++11, this can be done easily using the `delete` keyword:

```
/// Broken copy constructor
Mesh(const Mesh& dummy) = delete;

/// Broken assignment operator
void operator=(const Mesh&) = delete;
```

1.4.5 Order of arguments

- If values are returned from a function, put them at the end of the argument list.
- "Time" arguments always come first, e.g.

```
/// \short Return FE interpolated coordinate x[i] at local coordinate s
/// at previous timestep t (t=0: present; t>0: previous timestep)
virtual double interpolated_x(const unsigned& t,
                              const Vector<double>& s,
                              const unsigned& i) const;
```

1.4.6 Access to elements in containers

- Avoid access via square brackets (i.e. via operators) and write access functions instead, as they can be overloaded more easily.

1.4.7 Boolean member data

- Avoid access to boolean member data via trivial wrapper functions that return references. These constructions lead to somewhat ugly driver codes and can lead to code that appears to set a boolean, when it does not. Instead the status of the boolean should be modified by two set/unset or enable/disable subroutines (i.e. returning void) and tested using a (const) has_ or is_ function that returns a bool . For example

```
private:

  /// Boolean to indicate whether documentation should be on or off
  bool Doc_flag;

public:

  /// Enable documentation
  void enable_doc() {Doc_flag=true;}

  /// Disable documentation
  void disable_doc() {Doc_flag=false;}

  /// Test whether documentation is on or off
  bool is_doc_enabled() const {return Doc_flag;}
```

1.4.8 Macros

- Don't use macros! There are two exceptions to this rule: We use the macros `OOMPH_EXCEPTION_LOCATION` and `OOMPH_CURRENT_FUNCTION` to make the file name, line number and current function name available to the `OomphLibException` object – the object that is thrown if a run-time error is detected.

1.4.9 Inlining

- Inline all simple set/get functions by placing them into the $*$.h file.
- **Careful:** Inlined functions should not contain calls to member functions of classes that are defined in other files as this can lead to triangular dependencies.

1.5 PDF file

A pdf version of this document is available.