LUMA Coding Standards

Last update: 18th April 2017

Purpose

This document sets out the rules associated with formatting, design and organisation of code which contributes to LUMA. The implementation of these rules ensures standardisation across all contributions of the development team. In other words, new features written into the code should be developer-agnostic from an appearance perspective. The specification of these rules prioritise code readability and architectural comprehension over efficiency and optimisation. This is due to the fact the LUMA is an engineering package, developed for and by engineers and not computer scientists or software engineers.

Caveat

As LUMA has evolved over time, there are numerous exceptions to these coding standards throughout the source code. With each release, many of these exceptions are being updated to reflect the new coding standards but this will take time. If all new code follows the conventions then in time we will get the desired homogeneity.

Standards

Macros

All LUMA pre-processor macro definitions should be capitalised, be prefixed by L_ and have words separated by and underscore. Definitions should not have numbers in their names. E.g.

```
#define L_NO_FLOW
```

Line Length

Wherever possible, lines should not exceed **81** columns to maximise readability.

Comments

Single line comments should leave a space between the start of the comment and the double slash. Example

```
// My Comment
```

Multi-line comments should use the “java-style” syntax with repeated asterisks. Example:

```
/* My multi-line...
 * ...Comment. */
```

Brackets and Braces

Braces, used to define code blocks should always be aligned and should be started on a new line for clarity. This principle should also extend to for loops and other commonly used constructs. Example:

```
void myMethod(…)
{
	// etc.
}
```

No extra white space is needed when using parentheses and should be used in-line unless the expression they encompass is large or exceeds the line length guidance. In such cases, the parentheses may be split across lines like braces for clarity. Example:

```
void myMethod(arg1, arg2, …
	…argn…,
	…argnn);
```

If possible, statements should be aligned with parentheses aligned like this:

```
if	(
		cond1 &&
		cond2 &&…
		…condn
	)
{
	// etc.
}
```

White Space

White space is encouraged to improve readability. In particular, leave a space after the comma in multi-argument method calls and also in between numerical operators. Example:

```
// Arithmetic
mySize = N_lim * M_lim;

// Arguments
myMethod(a, b, c);
```

Control Flow Compact Form

The compact form of single line body control flow statements (without the braces) can be used if it improves readability and obeys the line length guidance but should be on separate lines to facilitate debugging. Example

```
// First branch
if (short condition)
	doSomething();

// Second branch
else
	doSomethingElse();
```

Ternary syntax may also be used e.g.

```
Variable = (condition) ? value1 : value2;
```

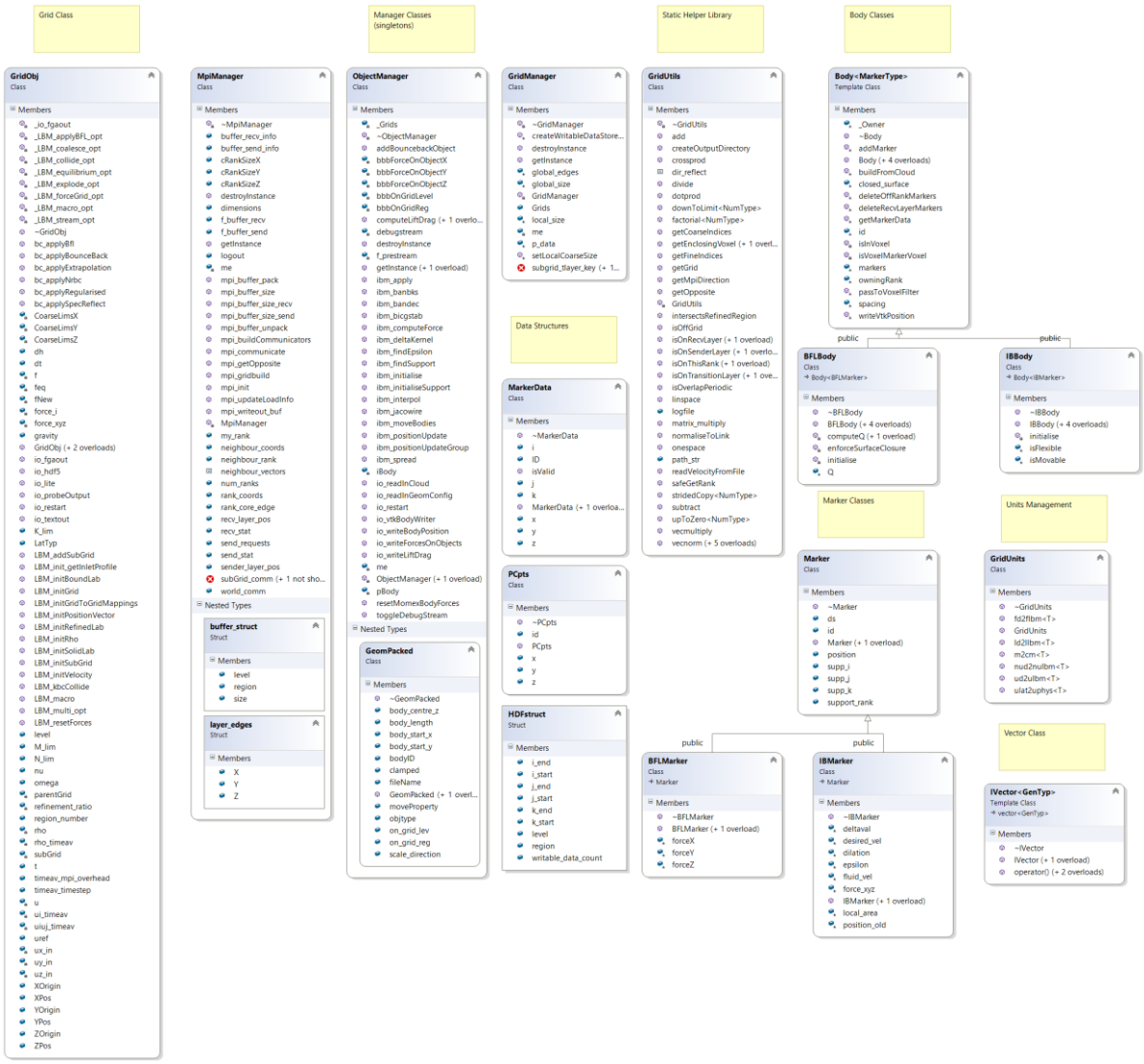
Initialiser Lists

Initialiser lists should be used wherever possible to simplify constructors. For readability, the preceding colon should be placed on the next line. Example:

```
MyClass::MyClass()
	: superMethod(), member1(arg1)
{
	// etc.
}
```

File Structure

LUMA is modularised. It is split into a number of class-based units that contain capability. The lattices and all their data is maintain in a hierarchy of GridObj classes. Three singletons act as managers MpiManager, GridManager and ObjectManager. Utilities are provided by the static GridUtils class which should not be instantiated. If new functionality cannot be rationalised as an extension of these existing units then a new class should be designed and created. The emphasis here is on *design*. When implementing a new class, time must be spent understanding the behaviour, interoperability and reusability of the class and its functionality.



The class is declared in a *.h header file and, unless it is templated, implemented in one or more *.cpp files. These files must be named using the convention given in Naming Conventions.

Getters / Setters

The use of getters and setters for private information should be avoided as it increases code complexity. Consider making a class a friend class if private access is needed or consider redesigning the class.

Documentation

LUMA uses doxygen for documentation. All classes must be documented at declaration in the *.h header file. Member variables must be documented in the header file. However, member methods must only be documented in the *.cpp file. A simple comment can be used in the header file without the doxygen syntax. The method documentation block must use tab alignment for clarity, e.g.

```
// ****************************************************************************
/// \brief    Constructor for a sub-grid.
///
///           This is not called directly but by the addSubGrid() method which
///           first performs a check to see if a sub-grid is required.
///
/// \param RegionNumber     ID indicating the region of nested refinement to which
///                         this sub-grid belongs.
/// \param pGrid            pointer to parent grid.
```

The copyright / IP protection header should be included in every source file (*.h or *.cpp).

Naming Conventions

Common conventions for naming are as follows:

- Classes (including structures) must start with an upper-case letter.
- Member methods must use camel-case with a lower-case initial letter.
- Member variables must use camel-case with an upper-case initial letter
- Macros are all caps and spaced by underscores.
- Template arguments should start with an upper-case letter.

When creating new cpp files, each file must be used to group implementation of methods with a common theme. For example, I/O methods can be grouped into their own file so those methods are easier to find in the source code. This common theme must be appended to the class name as part of the file name. e.g. for the GridObj class, I/O methods are grouped in the file *GridObj_ops_io.cpp*. Furthermore, the method should prefix the common theme e.g.

```
GridObj::io_readFromFile();
```

Structures

Structures should be used to pack and pass data to keep the number of arguments in external methods to a minimum. This is particularly relevant for I/O operations. A structure should be used where the proposed class would contain just data and no behaviour (i.e. no methods except the constructor/destructor). If the data is only to be used within a given class, the structure should be nested inside the class rather than declared as a separate h + cpp file combination.

Public, Protected, Private

All data in a class should initially be set to private access and only made protected/public as required. The use of getters and setters are discouraged as is unnecessary use of the friend keyword which should ideally be restricted for manager singletons only which are over-arching. Public data should be grouped at the top of a class declaration, followed by protected followed by private for visibility.

Deprecation

LUMA operates a deprecation model. Methods or variables that are no-longer required are first marked as deprecated using the macro, for example

```
DEPRECATED void bc_applyBounceBack(int label, int i, int j, int k);
```

The marked item will remain deprecated for one minor point release before being removed on the next point release. Do not simply delete components of classes without using the deprecation model to avoid breaking other parts of the code.

Hard-coding

Data should never be hard-coded. Any hard-coded information should be specified as a macro through the definitions file. The specification of initial values for member variables is acceptable but must be documented.

Header Guards

Header guards must always be used, even if

```
#pragma once
```

Is specified. This is for consistency. For example,

```
#ifndef GRIDOBJ_H
#define GRIDOBJ_H

	// etc…

#endif
```

Spacers

Methods within a file should be separated by an 80-character spacer

```
// ****************************************************************************
```

Defined Blocks

Pre-processor block defines, if nested should use a series of comments at the end of the definition to identify matching pairs. E.g.

```
#ifdef L_BUILD_FOR_MPI

	// etc

#ifdef L_HDF_DEBUG

	// etc

#endif // L_HDF_DEBUG


#endif // L_BUILD_FOR_MPI
```