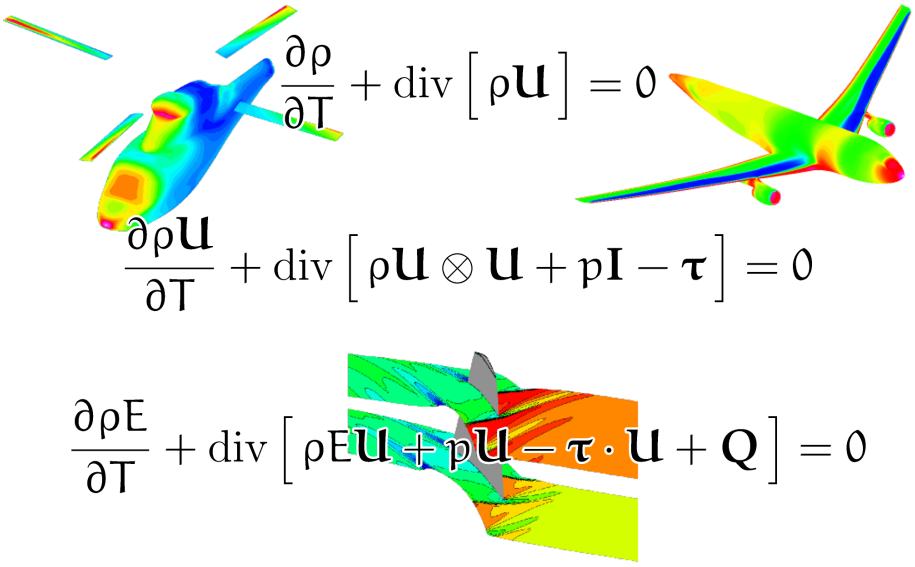
elsA programming rules



Quality	For the writers	For the reviewers	Approver
Function	Technical choices	Quality	Project leader
Name	M. Gazaix	A-M.Vuillot	L.Cambier
Visa			

Software management : ELSA SCM
Applicability date : immediate
Diffusion : see last page

HISTORY

version edition	DATE	CAUSE and/or NATURE of EVOLUTION
1.0	January 27, 2004	Creation - Successor of /ELSA/MDEV-98075 (Règles documentation dans les sources et Rgles de codage)

CONTENTS

Contents 3

1 Implementation choices 9

1.1 Why coding rules? . . . 9

2 General coding rules (C++ and Fortran source files) 11

2.1 File policy . . . 11

2.1.1 File name extension . . . 11

2.1.2 Physical file location . . . 11

2.1.3 File name prefix convention . . . 11

2.2 General coding style . . . 11

2.2.1 Use of English . . . 12

2.2.2 Avoid useless comments . . . 12

2.2.3 Do not use tabulation (CTRL-T) . . . 12

2.2.4 Avoid useless characters: . . . 12

2.2.5 Respect scrupulously alignment of code statement . . . 12

2.2.6 File header . . . 12

2.2.7 File footer . . . 13

3 C++ specific coding rules 15

3.1 C++ coding style . . . 15

3.1.1 C++ Code blocks: indentation and brace style . . . 15

3.1.2 Enclose single statement block into braces . . . 15

3.1.3 Prefer C++-style to C-style comments . . . 15

3.1.4 To improve readibility, separate each method definition with: . . . 15

3.1.5 "for" Loop coding style . . . 16

3.1.6 Avoid C-style typedef . . . 16

3.1.7 Write readable logical tests . . . 17

3.1.8 C++ Naming conventions . . . 17

3.1.9 Do not use pointer arithmetic . . . 18

3.2 Class definition and implementation . . . 19

3.2.1 Class definition . . . 19
3.2.2 Class implementation . . . 19
3.2.3 C++ class documentation . . . 19
3.2.4 Class section ordering . . . 20
3.2.5 inline methods . . . 21
3.2.6 Class attributes (data) are private (or protected), not public . . . 21
3.2.7 Get/Set interfaces: Accessors (get) and modifier (set) methods . . . 21
3.3 Do not systematically provide accessor methods . . . 22
3.3.1 Hide type implementation . . . 22
3.3.2 Prefer friend declaration for closely collaborating classes . . . 24
3.3.3 Do not give access to implementation (handle) class . . . 24
3.3.4 Do not provide modifier (set) method for a logically const attribute . . . 24
3.3.5 Breaking encapsulation with const_cast . . . 24
3.4 Choice between object, pointer and reference data members . . . 25
3.4.1 Use a value (object or primitive type) . . . 25
3.4.2 Use a pointer or a reference . . . 25
3.5 Avoid const data members . . . 27
3.6 Select the correct mode for method arguments (and return type) . . . 27
3.6.1 Use const qualifiers . . . 27
3.6.2 Argument ordering in C++ methods . . . 28
3.6.3 Use default arguments where possible. . . . 28
3.6.4 Pass primitive type arguments by values . . . 29
3.6.5 Pass non primitive type arguments by address . . . 29
3.6.6 Pass a pointer . . . 29
3.6.7 Pass objects through pass-by-reference: . . . 29
3.6.8 Pass const object arguments ("IN") as reference-to-const . . . 30
3.6.9 Non-const methods must not return a non-const pointer (or reference) to any class data member . . . 30
3.6.10 Accessor methods must be declared const . . . 30
3.6.11 Never return a reference to a local variable . . . 31
3.6.12 When creating an object inside a function, what is the best way to pass it (by argument or by return) to the caller? . . . 31
3.6.13 Be careful when modifying the address contained inside a pointer . . . 33

3.7 Header inclusion 34
3.8 Include guards 34
3.9 Avoid including unnecessary headers 34
3.10 Use E_Float, E_Int, E_Bool 35
3.11 C++ Variable declaration and initialization 35
3.11.1 Declare variable immediatly before use 35
3.11.2 Avoid implicit conversion between floats and integers 36
3.11.3 Initialize variable as soon as possible 36
3.11.4 If possible, declare local variables const 36
3.11.5 Remove unused variables (declared and not used) 36
3.11.6 Pointer and reference declaration 37
3.12 Avoid hard-coded value in C++ code 37
3.13 Use log messages with care: 38
3.14 How to introduce new error messages 38
3.15 Character strings 39
3.15.1 Use TbxString (instead of char∗) 39
3.15.2 Choice between sprintf and stringstream 40
3.16 Constructor 40
3.16.1 Constructor implementation 40
3.16.2 Copy Constructor (and assignment operator) declaration 41
3.17 Virtual methods 41
3.17.1 Virtual methods declaration 41
3.17.2 Do not ask an object its type 41
3.17.3 Do not redefine inherited non virtual methods 42
3.17.4 Do not redefine inherited argument default value of virtual methods 43
3.17.5 Do not overload virtual methods in derived classes 43
3.18 Destructor 44
3.18.1 Virtual destructor 44
3.18.2 Destroying array of objects 44
3.18.3 Use covariant return type 45
3.19 Dynamic memory allocation 45
3.19.1 Do not use malloc/free 45

3.19.2 Prefer Fld objects (FldArray, FldField) to C arrays . . . 45
3.20 Storage ownership responsibility . . . 47
3.20.1 Deleting kernel objects . . . 47
3.20.2 When possible, the creator is responsible of the object's destruction . . . 48
3.20.3 Adopt semantics . . . 48
3.21 Do not use C++ Template (except standard library templates) . . . 48
3.22 Use of STL container . . . 49
3.22.1 Do not include directly STL headers (such as <list>). . . . 49
3.22.2 Use STL container iterators in a systematic way . . . 49
3.22.3 Macro E_CONST_ITERATOR . . . 50
3.23 Do not use C++ exception . . . 50
3.24 How to use cast . . . 50
3.24.1 Avoid using cast . . . 50
3.24.2 Use C++ cast (instead of C-like cast) . . . 51
3.25 Prefer C++ methods to cpp preprocessor Macros . . . 51
3.26 How to write portable C++ code . . . 52
3.26.1 Do not include system headers, except in Def module . . . 52
3.26.2 Respect function return type . . . 52
3.26.3 Numerical "helper" functions . . . 52
3.27 Optimization of C++ code . . . 53
3.27.1 C++ code vectorization . . . 53
3.27.2 C++ code CPU optimization . . . 54
3.28 MPI insulation . . . 55
3.28.1 Do not include directly MPI header mpi.h . . . 55
3.28.2 Use Macro E_MPI . . . 55
3.29 Calling Fortran subroutine . . . 55
3.29.1 Passing scalar (E_Int or E_Float) arguments . . . 56
3.29.2 Passing array arguments . . . 57
3.29.3 Argument ordering . . . 57
3.29.4 Declare Fortran subroutine name as lowercase . . . 57
3.30 C++ common coding errors . . . 57
3.30.1 Memory corruption: Pay attention to '()' vs '[ ]' . . . 57

3.30.2 Unintended constructor conversion (use explicit keyword) . . . 57
3.30.3 Be careful with vector<>::operator[ ] . . . 58

4 Fortran specific coding rules 61
4.1 Fortran coding style . . . 61
4.1.1 Every Fortran source file must contain a single subroutine. . . . 61
4.1.2 Fortran comments . . . 61
4.1.3 Code block (DO/ENDDO, IF/THEN/ENDIF) . . . 61
4.1.4 Do not put the optional RETURN statement . . . 61
4.1.5 Use of (lower/upper)case . . . 61
4.2 Avoid using PRINT and WRITE statements . . . 61
4.3 Do not use hard-coded value inside Fortran . . . 62
4.4 Do not call Fortran subroutine (or C function) . . . 62
4.5 Do not use COMMON . . . 63
4.6 Consider using Fortran include files . . . 63
4.7 Ordering of arguments . . . 63
4.8 Do not use Fortran-90 specific constructs . . . 63
4.9 Do not use Fortran 77 implicit type convention . . . 64
4.10 Include DefFortran.h . . . 64
4.11 Use REAL_E and INTEGER_E . . . 64
4.12 Avoid using LOGICAL . . . 64
4.13 Avoid using CHARACTER . . . 64
4.14 Use adressing statement functions (GeoAdrF.h) . . . 64
4.15 Pass two-dimensional arrays as a single argument . . . 64
4.16 Fortran documentation . . . 65
4.17 Declaration section . . . 66
4.18 Do not use Fortran automatic array . . . 66
4.19 CPU optimization . . . 66
4.19.1 When possible, write vectorizable loops . . . 66
4.19.2 Consider providing scalar and vector versions . . . 67
4.20 Fortran example . . . 67

5 Python specific coding rules 69

Index **72**

1. IMPLEMENTATION CHOICES

elsA kernel is mainly written in **C++**. C++ classes, with a clean separation between interface and implementation, are the building blocks of the Object-Oriented design of elsA.

Most computationaly intensive tasks are delegated to **Fortran** subroutine, to take advantage of the very high quality of Fortran compilers when it comes to number crunching. Typically, these Fortran subroutines implement a loop acting on a range of cells or interfaces. We stress that the design is not affected by the delegation of intensive computation to such computational leaves (Fortran subroutine may be viewed as special `private` ("hidden") implementation methods).

The language for the elsA interface is **Python**, a widely used object-oriented interpreted language.

1.1 Why coding rules?

As many people have to read, modify, extend the code source, we all have to follow the same rules in order to work together efficiently.

These rules can be subject to criticism, but one has to be aware that the key point is that everybody knows the rules, instead of having one coding style per developer.

Moreover, a common coding style can be supported by tools, such as code skeleton generators and documentation automatic extractors.

Remarks:

C++ header files are the place where a library comes into contact with user code and other libraries. It is thus specially important that elsA C++ headers respect the coding rules.

Coding rules serve several purposes:

- Improve code readibility: this is specially important in a large project, where every line will probably be read many times.
- Maximize compatibility and portability: a new development should be be plug-compatible with others; reliance on as few system-dependent features (such as system calls) as possible, while still providing optional support for other useful features found only on some systems; for example, any development submitted to integration review should work fine with 32- or 64-bits addressing scheme, in single and double precision, on different computing platforms (workstations, vector supercomputers, PC clusters), in serial or parallel (MPI) mode.
- Minimize space: it should not waste space (it should obtain as little memory from the system as possible).
- Minimize CPU time: routines should be as fast as possible in the average case.
- Maximize tunability:

  - Static (compile time) hard-coded behaviour must be avoided.
  - Instead, optional features and behavior should be controllable by end users dynamically, at runtime, via control commands. To help users, default values must be provided. For instance, the number of ghost cells has a default value equal to two, and it can be changed with:

    ```
    problem.set_ghostcell(ghi1,ghi2, ghj1,ghj2, ghk1,ghk2)
    ```

- Maximize locality: code should try to allocate chunks of memory that are typically used together near each other. This helps minimize page and cache misses during program execution.
- Maximize error detection: code should provide accurate error reports, understandable by elsA users.
- Minimize compile and link time.

The following sections give the list of the "official" elsA design and coding rules. There does not exist a strict separation between *design* and *coding* rules. In future release of this document, we may try to distingish between them.

Note:

The current version of elsA violates some of the rules discussed in this document (see for example 3.17.2, 3.6.6). Please, when reading elsA source code, if you notice violations of the rules described in this document, either correct them, or contact the Developer Support team (`elsA-infodev@onera.fr`).

2. GENERAL CODING RULES (C++ AND FORTRAN SOURCE FILES)

2.1 File policy

2.1.1 File name extension

- C++ headers have a `.h` filename extension. Private header (3.3) filenames end with a a capital `P` (see for example `JoinBaseP.h`).
- C++ body (implementation) files have a `.C` extension.
- Fortran files have a `.for` extension, and terminate with a capital `F` [1]. Moreover, Fortran include files (see 4.6) have a `.h` extension.
- SWIG interface files have a `.i` filename extension; they are located in directory `src/Api/Wrapper`.

2.1.2 Physical file location

Put elsA files (C++ and Fortran) in a sub-directory of a Module directory: for example, `OperTerm.h` is located in directory `Kernel/src/Oper/Term.`

Note:

This rule admits two exceptions (`Tmo/Solver/Time` and `Sio/Access/Fortran`). This will be corrected in future release.

2.1.3 File name prefix convention

Each file name is prefixed by the module identifier `Key` (see 3.2).

2.2 General coding style

Note:

At first glance, specifying rigid formatting rules may seem too restrictive; however, recall that in many cases, CVS will **not** be able to resolve conflicts arising from such "benign" formatting issues. Instead of spending time resolving conflicts, it is much better to conform to a small number of rules.

[1] '`.for`' extension may be changed to `.F` in future release

2.2.1 Use of English

All comments are written in correct English. Identifiers (name of classes, functions, ...) are chosen to be self informative in English.

2.2.2 Avoid useless comments

Comments must bring some information. Avoid comments such as:

```
/** This is the destructor */
  ~SomeClass()
...
/** virtual method */
  virtual void  f();

C     DO Loop
      DO i=1
      END DO
```

Such comments duplicate language constructs, and do not add any information.

2.2.3 Do not use tabulation (CTRL-T)

2.2.4 Avoid useless characters:

- blank characters at end of line;
- blank lines at end of file;
- in C++, blank characters before the `';'` statement termination.

2.2.5 Respect scrupulously alignment of code statement

Code reading, checking and understanding will be much easier. This applies also to commented code:

```
   if (x)
   {
     // correctly commented code
// badly commented code
   }
```

2.2.6 File header

Every source file must begin with a standardized header:

```
// ============================================================================
// Project: elsA - Copyright (c) 1998-2003 by ONERA
// Type   : <2834957738 6236> C++ header file (or C++ body file, or For-
tran file)
```

```
// File   : SomeModule/SomeSubDir/SomeFile
// Vers   : $Revision: 1.56 $
// Chrono : No  DD/MM/YYYY Author   V   Comments
//          1.1 17/10/2000 YZ       2.0 Add _nbErrror&  _nbErrorMax
//          1.0 01/01/1999 XY       0.6 Creation
// ============================================================================
```

2.2.7 File footer

The last line of each file should be:

- for C++ files:

  ```
  // ==== SomeModule/SomeSubDir/SomeFile.[Ch] ====
  ```

- for Fortran files:

  ```
  C ==== SomeModule/SomeSubDir/SomeFile.for ====
  ```

Empty page

3. C++ SPECIFIC CODING RULES

3.1 C++ coding style

3.1.1 C++ Code blocks: indentation and brace style

- Coding blocks (`for {}, if {}, switch {}`) must be indented by 2 blank characters. Example:

```
for (E_Int j=0; j<ni; i++)
for (E_Int i=0; i<nj; i++)
{
  some C++ code ...
}
```

- Braces delimiting blocks must be the first non blank characters; to improve readibility, they must be carefully aligned.

3.1.2 Enclose single statement block into braces

To avoid potential errors, prefer:

```
if (a == b)
{
  return;
}

for (E_Int i=0; i<nj; i++)
{
  a(i) = b(i);
}
```

to:

```
if (a == b)
  return;

for (E_Int i=0; i<nj; i++)
  a(i) = b(i);
```

3.1.3 Prefer C++-style to C-style comments

```
  // Preferred C++ comment style
  /* Avoid C-style
     comment  */
```

3.1.4 To improve readibility, separate each method definition with:

```
// ================================================================
```

3.1.5 *"for" Loop coding style*

- when possible, declare the loop counter inside the "for" definition:

```
Prefer:
for (E_Int l=0; l<lmax; l++)

to:
E_Int l;
...
for (l=0; l<lmax; l++)
```

  This is a special case of rule discussed in 3.11.1.

- Prefer:

```
for (E_Int l=0; l<lmax; l++)
```

  to:

```
for (E_Int l=1; l<=lmax; l++)
```

- Prefer:

```
for (E_Int j=0; j<ni; i++)
for (E_Int i=0; i<nj; i++)
{
  some C++ code ...
}
```

  to:

```
for (E_Int j=0; j<ni; i++)
{
  for (E_Int i=0; i<nj; i++)
  {
    some C++ code ...
  }
}
```

- To loop inside a STL container (see 3.22.2), use the following style:

```
for (list<E_Int>::const_iterator iter  = someList.begin();
                                 iter != someList.end();
                                 iter++)
{
  some code
}
```

3.1.6 *Avoid C-style typedef*

Avoid C-style `typedef`:

```
typedef enum  {C0, C1, C2} SomeEnum; // To be avoided
enum SomeEnum {C0, C1, C2};          // Preferred syntax
```

3.1.7 *Write readable logical tests*

- Prefer:

```
if (a == b   &&   c != d)
```

to:

```
if((a==b)&&(c!=d))
}
```

- Avoid performing complex operations inside logical test

```
// Too complex logical test:
if (!getSubSystem(nsys).advanceIteration(computeLocalTimeStep,
                                         computeResidual,
                                         resToComp,
                                         _driverType))
```

- Conditional expressions that involve negation are always hard to understand. Instead of:

```
if (!(blockId < act)   ||   !(blockId >= unb))
```

state the test positively:

```
if (blockId >= act   ||   blockId <  unb)
```

3.1.8 *C++ Naming conventions*

- Macro names must be capitalized.
- The first letter of identifier of **types** (`class`, `typedef`, `enum`) is capitalized; Example:

```
class DefError {...};
typedef FldFieldI FldIntF;
enum GeoDimension { Geo1D, Geo2D, Geo2DAxi, Geo3D };
```

- Use lowercase, except for "composed" name: `SomeClass::someMethod()`.
- Class attributes are prefixed by a single underscore (`'_'`) [1]. Do not put too much semantics in attribute's name: prefer `_bndCTor` to `_allBndVirtualConstructors`.
- Entities belonging to a restricted scope can have short identifier length, since name conflicts are easy to avoid.

1. For example:

```
for (E_Int i=0; i<4; i++)
{
}
```

Here, `i` is not known outside the `for` block.

2. Another example concerns class `enum`. Outside the class implementation, they are refered with their full name:

[1] Do not use two `'_'`, since system headers often use identifiers with *t*wo leading underscores.

```
FldArrayB operToComp(OperBase::E_MAX_OPERTOCOMP);
```

It is thus rather unlikely that name conflicts will arise.

- Conversely, global entity names should be chosen to avoid any potential conflicts:

```
// Global variables:
E_GHOST_I1, E_FLD_STACK_SIZE_F

// Global enum :
enum EosElemSysType{eosel_meanFlow, ...
```

- Be careful of not interacting with system header macros (for instance, `BUFSIZE`, `BUFFSIZE`, `CHARSIZE`,...).
- Do not encode type information into names; for instance, avoid `gridObj`, `ptrToGrid`, `refGrid`:
  - this does not add security, since the compiler performs type checking anyway;
  - entity type may change during software maintenance and evolution; it would be awkward to have to change names.
- Method names must be informative, specially public ones; however, avoid providing *implementation* clues: if implementation changes, it will be confusing:
  - prefer `FactBase::registerBnd()`, to `FactBase::registerBndVirtConstructor()`
  - `getBlankedCellArray()`: here, postfix `Array` is probably redundant in the method's name.
- Naming of function arguments must be informative.
  - use the same name in declaration and implementation

3.1.9 Do not use pointer arithmetic

Pointer arithmetic (increment and decrement operation) are forbidden, except for a relatively small number of vectorized C++ loops:

```
void
FldFieldF::mulsca_add_assign(      FldFieldF& lhs,
                             const E_Float&   scalar,
                             const FldFieldF& fld)
{
  // this loop does NOT vectorize
  // (at least with older versions of C++ compiler):
  for (E_Int i=0; i< size; i++)
  {
    lhs[i] += scalar * fld[i];
  }

  // Use of pointer arithmetic
  const E_Float* ptfld = fld.begin();
        E_Float* ptlhs = lhs.begin();
  E_Int size = lhs.getSize() * lhs.getNfld();
  for (E_Int i=0; i< size; i++)
```

```
  {
    ptlhs[i] += scalar * ptfld[i];
  }
}
```

3.2 Class definition and implementation

3.2.1 Class definition

Class name must begin with the first three letters of the module key identifier (`Des` for `Descp` module, `Tur` for `Tur` module,...). In order to find easily class definitions, the header file containing the class definition must have the same name: for example, class `SomeClass` is defined in file `SomeClass.h`.

Note:

Important abstract classes, located at the root of an inheritance tree, have been named with a `Base` suffix: `BndBase`, `JoinBase`, `LhsBase`, `OperBase`, `TurBase`, `SioBase`, `DesBase`, `DtwBaseWinFld`,... If you introduce new hierarchy, consider using this convention.

3.2.2 Class implementation

Class `SomeClass` implementation is done in file `SomeClass.C`. If the implementation file is too complex (approximately larger than 1000 lines), it may be split into several files (`SomeClass_1.C, SomeClass_2.C,...`). This should improve readibility. In such cases, consider also splitting the class into several simpler classes.

3.2.3 C++ class documentation

Every C++ class definition must be preceded by a specially formatted comment section, which can be analysed by doxygen (or Doc++) to extract automatically the documentation. Moreover, public methods are commented with a special syntax. Here is an example that speaks for itself:

```
// ============================================================================
//        Project: elsA - DSNA/ELSA - Copyright (c) 1998, 1999, 2000 by ONERA
// Type   : <3177557961 22760> C++ Public header file
// File   : Module/Dir/MyNewClass.h
// Vers   : $Revision: 1.1 $
// Chrono : No  DD/MM/YYYY Author   V   Comments
//          1.0 xx/xx/2000 UserName 3.0 Creation
// ============================================================================
#ifndef _MODULE_DIR_MYNEWCLASS_H_
#define _MODULE_DIR_MYNEWCLASS_H_

// ============================================================================
// @Name MyNewClass
// @Memo Fundamental new class.
// @See  AnOtherClass
/* @Text

Design
> Fundamental container ...
```

```
*/
// ============================================================================
class MyNewClass
{
  public:
    ///+ 1- Constructors / Destructor
    /** Constructor
      * Very clever indeed */
    MyNewClass();
    /** Destructor */
    ~MyNewClass();
    ///-

    ///+ 2-
    /** Provides the fundamental abstracion... */
    void method1();
    ///-

  private:
    // Copy constructor
    MyNewClass(const MyNewClass& rhs);
    // Assignment Operator
    MyNewClass& operator=  (const MyNewClass& rhs);
}; // ======== End class MyNewClass

#endif
// ===== Module/Dir/MyNewClass.h === Last line ===
```

3.2.4 Class section ordering

The preferred ordering inside class definition is:

1. friend declaration;
2. typedef declaration;
3. class `enum` definition;
4. public constructor(s);
5. destructor;
6. accessor (get/set) methods;
7. other public methods;
8. protected methods;
9. protected data;
10. private methods (if any, private constructor and assignment operator first);
11. private data;

In implementation files, follow as much as possible the same ordering for method implementation.

3.2.5 inline methods

- Use inline methods only when appropriate: `inline` methods increase coupling between components and with external clients: if the implementation is changed, client code must be recompiled. `inline` methods are also hard (or even impossible) to debug with a debugger. So use inlining only if:
  - the implementation is trivial, and will not change in the future;
  - profiling shows that too much time is spent inside the function.

  Avoid using inlining if complex headers have to be included; for example, `DefError` must not be used by inline methods.
- Implement inline method definition outside of class definition:

```
class C
{
  void f();  // Not implemented here: { ...}
};

inline void C::f()
{ ... }
```

3.2.6 Class attributes (data) are private (or protected), not public

Encapsulation requires that class clients do not have access to class attributes. Therefore, attributes must be `private` (preferred), or `protected`.

Note:
Protected attributes may be used in derived class implementation; in fact, derived class implementers can be viewed as special clients. This means that `protected` attributes are not fully encapsulated. They should be used with care, specially non primitive type attributes. For example, `Bnd` classes must somehow call `GeoGrid` methods. At first glance, inserting a `protected GeoGrid*` attribute in class `BndBase` may seem a good idea. However, `Bnd` classes dealing with motion treatment (`BndEulerRelFrameAbs`, `BndNoReflUnst`, `BndNSRelFrameWallLaw`,...) need a `GeoGridMotion*` pointer [2], not `GeoGrid*`. In that case, to avoid dangerous downcasts, it is much better to avoid entirely the protected attribute; instead, insert an attribute of the correct specialized type in derived concrete classes. In doubt, avoid introducing `protected` attributes in ancestor classes, instead put private attribute in derived classes.

3.2.7 Get/Set interfaces: Accessors (get) and modifier (set) methods

- Use the following convention for `get/set` methods

```
class C
{
public:
  SType        getSimple()  const;
  const CType& getComplex() const;

  void setSimple(SType);
  void setComplex(const CType&);

private:
```

[2] class `GeoGridMotion inherits from class GeoGrid`

```
  // attribute of Simple  type (primitive type or simple class)
  SType _simple;
  // attribute of Complex type (costly copy constructor)
  CType _complex;
};
```

See 3.6.10 for a thorough discussion.

Note:

One exception to the `get-` prefix naming convention is the `instance()` method used with the singleton design pattern. For instance:

```
class FactBase
{
public:
  static FactBase* instance();
...
};
```

3.3 Do not systematically provide accessor methods

It is bad practice to provide systematically `get/set` methods. In many cases, relying too much on accessor methods is a clear sign of bad design, in which the public interface has not been correctly defined. The following discussion tries to explain why.

3.3.1 Hide type implementation

Private attributes are, well, private. By exposing attribute *type* in the public interface, the accessors (`get` or `set`) expose the rather private information that the internal implementation uses this type. This is of course specially annoying for "complex" types. Any code that calls this public function can (and most probably will) develop a dependency on that particular implementation. Since it is an important point, let us give several examples.

1. `FldField[FI]::getArray()`

   ```
   Example 1
   inline const FldArrayF* FldFieldF::getArray() const
   {
     return (&_array);
   }
   ```

   Here, the "private" attribute `_array` can be used by client code. In fact, there is presently more than 100 (!) occurences(!), such as:

   ```
   File BndSubInj.C:
   _pres.setOneField(*_data.getArray(),1,1);
   ```

2. Method such as `getData()`, or `getImplementation()` are specially suspect:

   ```
   Example 2
   // bad design indeed
   inline JoinBaseP& JoinBase::getImplementation()
   {
     return *_pimpl;
   }
   ```

3. `FactBase::getDB()` (elsA v2.2)

```
Example 3
class FactBase
{
  ...
  static const FactDataBase& getDB()
  { return instance()->_db; }
};

DesNumerics.C:
void DesNumSpaceDisc::submit(DesModel& desMod)
{
  ...
  TbxString operClassName = FactBase::getDB().getOperClassName(flux);
}
```

   The client code is tightly coupled to the specific implementation detail (here, an attribute of `FactDataBase` type). If, for whatever reason, attribute `_db` is removed, or its type changed, client code will have to be *modified* (not only `compiled`).

4. A better solution (elsA v3.0) is:

```
Example 4
class FactBase
{
  static TbxString getOperClassName(const TbxString& operId);
};

void DesNumSpaceDisc::submit()
{
  TbxString flux = desNSD.getS(KEY_FLUX);
  TbxString operClassName = FactBase::getOperClassName(flux);
}
```

   Now the client does not have to care about the internal implementation. Basically, the design has been improved by finding the true public interface:

   - The responsibility of `FactBase` is to solve the indirection between flux identifier as seen from the user side and kernel flux classes (see elsA tutorial).
   - A new method, `getOperClassName` has been introduced, and the accessor `getDB()` removed.

5. `RhsEquation::getListRhsjoin()`

```
Example 5
class RhsEquation
{
  list<RhsJoinBase*> getListRhsjoin() const;
  { return _listRhsJoin; }
};

SetOfSolver::fillInGhostCellsIncr(...)
{
  ...
  list<RhsJoinBase*>
  lR = (*itr)->getRhsEquation().getListRhsjoin();
```

The client code depends on the private information that the container is implemented with a STL `list`. A minimum improvement consists of an insulating `typedef`:

```
typedef list<RhsJoinBase*> ContainerRhsJoin;
```

3.3.2 Prefer friend declaration for closely collaborating classes

In some cases, several closely coupled classes cooperate to implement a complex algorithm (`Tmo` component, for example), so that removing accessors functions is difficult. Consider declaring the accessor methods `private`, and using `friend` declaration: only explicitly declared clients will be able to "see" the internal details of the class.

3.3.3 Do not give access to implementation (handle) class

A special case of collaborative classes (as discussed in the previous section, 3.3.2) is the strong coupling between a class and its private implementation class. This pattern (sometimes called the *handle* idiom, or *pimpl* idiom) is useful to hide the implementation details to clients:

```
class SomeClass
{
  ...
private:
   SomeClassP* _pimpl;
};
```

Here, class `SomeClassP` is only used by class `SomeClass`; instead of providing `set/get` methods for class `SomeClass`, it is probably a better design that `SomeClassP` declares `SomeClass` friend. `SomeClassP` is defined in a private header, `SomeClassP.h`:

```
class SomeClassP
{
  friend class SomeClass;
private:
  ...
};
```

This private header should **not** be known to clients of `SomeClass`. It is only included by `SomeClass.C` and `SomeClassP.C`.

3.3.4 Do not provide modifier (set) method for a logically const attribute

Providing a `set()` method for an attribute which must be kept constant is silly (see also 3.5).

3.3.5 Breaking encapsulation with const_cast

Providing `get()` method, even with correct const-ness, allows (admittedly nasty) client code using `const_cast` to modify the private attribute.

3.4 Choice between object, pointer and reference data members

When inserting data members for a class, we have to select from three choices:

```
class C
{
private:
  D  _obj;  // composition
  D* _ptr;  // aggregation implemented with pointer
  D& _ref;  // aggregation implemented with reference
};
```

3.4.1 Use a value (object or primitive type)

Using a value (object or primitive type) as an attribute is sometimes called *composition*. If the attribute is not used polymorphically, this is the preferred way, since it is simpler to code, and safer: the embedded object is always created in constructor(s), so that client code does not have to take care of invalid attribute.

```
C::C(...) : _obj(...), ...  // calls D's constructor
```

Moreover, when the embedding object is destroyed, the compiler will take care of calling the destructor of the embedded object. Many examples of object attributes can be found in elsA kernel:

- `FldField` attributes are usually objects in elsA, avoiding to manage explicitly dynamic memory (pointed to by `_data` attribute). For example:
  - `BlkMesh::_coord` has type `FldNodeF`; it stores mesh point coordinates.
  - `LhsBase::_lhs` has type `FldCellF`; it is used inside the implicit algorithm.
- class `JoinAdjacentP` owns two attribute of type `AgtFrame` (`_trOppToCurr` and `_trCurrToOpp`).

3.4.2 Use a pointer or a reference

Use a pointer or a reference attribute if polymorphic behavior or increased flexiblity is required.

1. Use a pointer only when full flexibility is required:
   - at creation time, it is not required to bind the pointer attribute to a valid object: in fact, different constructors may implement different strategies:
     - `C::C() :  _ptr(new D()) {;}`
     - `C::C() :  _ptr(E_NULLPTR) {;}` Testing if `_ptr` is different from `E_NULLPTR` may be used to choose appropriate treatment:

```
void TurSA::compSource(...)
{
  ...
  if (_turCriteria != E_NULLPTR)
  {
    compTriggerTransi(listBnd,vort,sourcT);
  }
}
```

```
– C::C(D* pd) :  _ptr(pd) {;}
```

- during its lifetime, a `C` object can use different `D` objects. This is used in elsA, for example in `prepare()` methods:

```
class OperBase
{
public:
  // Note: It is better style to pass pointer
  // instead of reference
  // (Linton's convention, Murray p214)
  virtual void prepare(EosIdealGas* eos,
                       GeoGridBase* grid);
protected:
  EosIdealGas*       _eos;
  const GeoGridBase* _grid;
};

void OperBase::prepare(EosIdealGas*       eos,
                       const GeoGridBase* grid)
{
  _eos  = eos;
  _grid = grid;
}

void RhsTerm::prepareTerm(...)
{
  ...
  _oper.prepare(grid);
}
```

  Here, the same object of type `GeoGrid` may be shared among several `OperBase` objects.

- The flexibility provided by pointer attributes have also some disadvantages:
  - it may be unclear if the pointer is really valid (points to a valid object);
  - if the object pointed to is created dyanmically using a `new` call, the destructor of embedding class must remenber to delete it to avoid memory leaks.

2. Use a reference:

- as soon as a `C` object is created, its constructor must bind the reference to an **already existing** valid D object, immediately in the initializer list (before entering the constructor body) [3].

```
class JoinMatchSeqP
{
  ...
  // reference attribute
  const BlkBaseBlock& _oppositBlock;
};
JoinMatchSeqP::JoinMatchSeqP(const BlkBaseBlock& currentBlock, ...),
 : JoinMatchP(currentBlock, ...),
   _oppositBlock(oppositBlock)
```

[3] To initialize a reference attribute, we must provide an argument; this means that arrays of objects, in which individual objects are built with zero-argument constructor cannot be used easily (for small arrays, it is still feasible to use explicit constructor call). In practice, this is not a real issue, since elsA does a very limited use of object arrays.

```
{
 ...
}

JoinMatch::JoinMatch(const BlkBaseBlock& currentBlock, ...)
{
  // (_pimpl has type JoinMatchSeqP*)
  // At construction time, a valid BlkBaseBlock
  // object MUST be provided
  _pimpl = new JoinMatchSeqP(currentBlock, ...);
}
```

- This is much less flexible than the pointer solution, but simpler: Readers of the class definition are immediately aware that there is no risk of invalid reference, nor dynamic memory operation.
- Assignment operator may be difficult to write.

3.5 Avoid const data members

Class with `const` data members cannot have "reasonable" copy constructor or assignment operator. So avoid them, unless you have good reasons. In fact, since all data members are `private`, we already have adequate protection from accidental modification (see also 3.2.7). Avoid declaring a `set` method associated to this attribute.

```
class C
{
private:
  const E_Float  f;               // const useless
  E_Float*       const_ptr const; // const useless
  const E_Float* ptr_to_const;    // Useful!
  const E_Float& ref_to_const;    // Useful!
};
```

3.6 Select the correct mode for method arguments (and return type)

- Arguments can be passed by *value* or by *address*; in this latter case, C++ offers two options: passing by reference and passing by pointer. Moreover, where passing by address, they can be combined with `const`.
- A function can return an object, a pointer or a reference (in the last two cases, possibly with const modifiers).

The mode of each argument (and return type) conveys a particular meaning to the client. Hence the following rules are specially important.

3.6.1 Use const qualifiers

To make the program more robust (see 3.6.8, 3.6.9, 3.6.10 and 3.2.7):

- Declare member functions `const` when possible.
- Declare pointer arguments as pointer-to-const when possible.
- Declare reference argument as reference-to-const when possible.
- Return pointer as pointer-to-const when possible.

- Return reference as reference-to-const when possible.

Note:

Even if it is legal (some compilers give a warning) it is meaningless to declare `const` argument passed-by-value. Likewise, it is redundant to declare `const` the result returned-by-value from a function:

```
const C g()
{
  C c;
  return c;
}

const int* f(const int i, const int* p)
{
  int* pi = new int[i];
  pi[0] = p[0];
  // pi is returned with a passed-by-value semantics
  return pi;
}
```

The four `const` are meaningless. Let us stress again that `const SomeClass&` (or `const SomeClass*`) is **not** a `const` argument, it is a reference-to-const (or a pointer-to-const).

3.6.2 Argument ordering in C++ methods

When possible, place modified arguments, if any, at the beginning of the calling sequence (this may be impossible, since default parameters must be placed at the end).

3.6.3 Use default arguments where possible.

- Use default arguments to provide different ways to call the same function.
  - Example 1 (see 3.29.2):

```
class FldFieldF
{
  ...
  /** Starting iterator on a field.
      Default field is the first one. */
  iterator begin(E_Int fld = NUMFIELD0);
}

FldIntF surf (nbOfInterface, 3)
...
surf.begin(); // better than surf.begin(1);
```

  - Example 2

```
TurKL(E_Float varturbmin1, E_Float varturbmin2,
      E_Float mutRatioMx       = 1.e15,
      E_Float prandtlTurb      = .9,
      E_Float coeffMutInit     = -1.,
      TurCriteria* turcriteria = E_NULLPTR);
```

Note the use of a *pointer* (**not** a reference) for the `TurCriteria` default parameter.

- Also, when an argument value is not used in the method implementation (this may happen for instance for a given implementation of a virtual method, where this argument is only useful for some other classes), do not give it a name:

```
void C::f(E_Int, E_Float f)
{
  implementation code independent of first argument
}
```

3.6.4 *Pass primitive type arguments by values*

Passing the address of a primitive type argument (`int`, `float`, `bool`, `enum`) instead of its value does not bring any significant CPU performance improvements. So use pass-by-value, since it is much safer, and easier to understand [4]. This rule **does not** apply when calling Fortran subroutine (see 3.29). For the same reasons, return primitive types by value.

3.6.5 *Pass non primitive type arguments by address*

Pass non primitive type arguments by address, **never** by value. Pass by address (pass by reference or pass by pointer) avoids creation of temporary objects, with the associated cost of constructor and destructor calls. To achieve full security without runtime penalty, see 3.6.8.

3.6.6 *Pass a pointer*

Pass a pointer in three situations:

- Default argument (of non primitive type) cannot be passed by reference (see 3.6.3).
- Use pass-by-pointer if the callee must be able to determine if the passed argument is valid: the advantage of a pointer is that a unique value (`E_NULLPTR`, which is the `NULL` pointer), distinguishes a valid pointer from an invalid one.

```
void DefError::userError(const char* file, E_Int line,
                         const TbxString* mesg)
{
  if (mesg) e_log << "User Error : " << *mesg << endl;
  DefError error(9940); error++; error.raiseError(file, line);
}
```

  Testing the validity of the pointer *argument* is quite similar to testing the validity of a pointer class data member (see 3.4). This would not be possible with passed-by-reference arguments.

- Use pass-by-pointer to initialize (or modify) pointer data members [5]; see for example the `OperBase::prepare()` example (3.4).

3.6.7 *Pass objects through pass-by-reference:*

If the callee expects a real object (there is no uncertainty about its existence) as the argument, and if the argument is not used to modify pointer attributes (see previous rule), then pass a reference, not a pointer. Using reference arguments simplifies a little bit the code, since we do not have to dereference pointers.

[4] of course, pass-by-address is mandatory if the argument value has to be (re-)set inside the called method
[5] Lintons'convention: see Murray, p214

3.6.8 Pass const object arguments ("IN") as reference-to-const

If an object passed as argument is used as a `const` (its contents is not going to change) inside a class method (or free function), it must be passed as a reference-to-const:

```
// Preferred syntax
someMethod(const SomeClass&);
// To be avoided: a temporary object will be created
// and then  destroyed on exit from someMethod()
someMethod(SomeClass);
```

Using a reference-to-const instead of a pointer-to-const clearly demonstrates that there is no memory operation inside the method.

3.6.9 Non-const methods must not return a non-const pointer (or reference) to any class data member

This will break data encapsulation:

```
// BAD CODE
class C
{
public:
  list<int>& f() {some code; return _list;};
  int size() {return _list.size();}

private:
  list<int> _list;
};

int main()
{
  C c;
  cerr << "c.size() = " << c.size() << endl;

  list<int>& viciousRef = c.f();
  viciousRef.push_back(1);
  cerr << "c.size() = " << c.size() << endl;
}
```

Return by value will remove any danger. If it is considered too expensive (in CPU and/or memory), return a pointer-to-const to the attribute.

3.6.10 Accessor methods must be declared const

Accessor methods are special cases of methods returning data members; they must always be declared `const`. Moreover:

- when possible, use return by value;
- if it is too costly, return by reference-to-const.

```
// CORRECT CODE
class C
{
public:
  const list<int>& getList() const {return _list;}
  E_Int getI()               const {return _i;}

private:
  E_Int     _i;
  list<int> _list;
};
```

3.6.11 *Never return a reference to a local variable*

```
SomeType& f()
{
  SomeType localObj; // local (automatic) object
  ...
  return localObj;   // Error !!!
}
// Caller code:
SomeType& ref = f();
```

Once you leave the function, the local variable `localObj` will be destroyed, so that `ref` will refer to something that is dead.

Note:
Such code has the annoying property of working on occasion; so be very careful.

3.6.12 *When creating an object inside a function, what is the best way to pass it (by argument or by return) to the caller?*

- Do not return a reference. To return a safe reference to an object from a function, the function could **not** have created that object:
  - it cannot be a local object (see previous rule);
  - who would be responsible for the storage of the newly created object? even if a header comment provide the information, it is unfair to ask to the client to take the *address* of the result and then call `delete`;
  - how will a function returning a reference to an object indicate a failure? there is no such things as `null` reference.
- The **recommended** way is to create the object dynamically (with the `new` operator). This technique is used heavily in `Fact` component (see elsA tutorial) to return polymorphic objects (`TurBase*`, `BndBase*`, `SioBase*`, ... pointers). To avoid memory leaks, you have to transfer the ownership of the new object to the caller. Two options:
  1. you can choose to *return* the pointer to the caller, which will subsequently own the memory pointed by the returned pointer (see also 3.20).
  2. alternatively, you can choose to pass a pointer argument by *address*, from the caller to the function (see example 3 of section 3.6.13). Avoid this option, since it is syntaxically more complex; the only situation where it could be useful is the (rare) case where several objects are created by the function, and have to be passed to the caller.

```
class C
{
public:
  C(float f) : _f(f) {}
private:
  float _f;
};

void f(B*& pB1, B*& pB2)
{
  pB1 = new C(1.);
  pB2 = new C(2.);
}

int main()
{
  B* pB1;
  B* pB2;
  f(pB1, pB2);
}
```

In both cases, if the function fails to create the new object (for whatever reason), it must return the `null` pointer (`E_NULLPTR`).

- Another possibility is to return by *value*, with the additional copy constructor cost. Obviously, the returned object cannot be used polymorphycally. For example:

```
list<JoinBase*>
FactJoin::getAllJoinsFor(const BlkBaseBlock& block) const
{
  list<JoinBase*> listJoin; // constructor call
  for (list<JoinBase*>::const_iterator iter  = _listJoin.begin();
                                       iter != _listJoin.end(); iter++)
  {
    JoinBase& join = *(*iter);
    if (&(join.getCurrentBlock()) == &block)
    {
      listJoin.push_back(*iter);
    }
  }
  return listJoin; // return by value : copy constructor call
} // end of function block : list<> destructor call (listJoin auto-
matic object)
```

In such a case, it is probably much better to instantiate a `list<JoinBase*>` *before* calling `getAllJoins-For()`, and to add an argument of type `list<JoinBase*>&` (passed by *reference-to-non_const*):

```
void
FactJoin::getAllJoinsFor(const BlkBaseBlock& block,
                         list<JoinBase*>& listJoin) const
{
  for (list<JoinBase*>::const_iterator iter  = _listJoin.begin();
                                       iter != _listJoin.end(); iter++)
  {
    JoinBase& join = *(*iter);
```

```
    if (&(join.getCurrentBlock()) == &block)
    {
      listJoin.push_back(*iter);
    }
  }
}
```

In fact, the semantics of this function is really to set the object *state*, not to build a new one. In similar cases, consider creating the object before calling the function, and passing it by reference.

3.6.13 Be careful when modifying the address contained inside a pointer

Pointers themselves are passed by values; in some cases, you may want to change the address contained (pointed to) by the pointer `p`. To do that, you will have to pass a *reference* to the pointer, with the syntax `SomeType*& p`. Be careful: this technique breaks polymorphism:

- a pointer to a derived class **is a** pointer to a (public) base class,
- but a pointer to a pointer to a derived class **is not** a pointer to a pointer to a (public) base class.

Let us give three examples:

1. Here, the caller asks the callee (`isBoundaryExist`) to fill a pointer (to Base class) argument with the address of a `BndPhys` (Derived class) object:

   ```
   class BndNS: public BndPhys {...};

   void isBoundaryExist(BndPhys*& bnd) {...}

   int main()
   {
     BndPhys  bPH;
     BndPhys* pPH = &bPH;
     isBoundaryExist(pPH);

     BndNS    bNS;
     BndNS*   pNS = &bNS;
     // cast required:
     // &pNS is-not-a BndPhys**
     isBoundaryExist((BndPhys*&)pNS);
   }
   ```

   Without (C-style) cast to `BndPhys*&`, the code would not compile.

2. Example 2

   ```
   void DtwBaseWinFld::addTurModel(TurBase*& turBase)
   {
     _turObjList.push_back(turBase);
   }
   ```

3. Here, the callee creates a new object and stores it in the pointer argument, such that the caller can retrieve this new address.

```
void FactBase::makeTurb(TurBase*& turBase, ...)
{
  ...
  turBase = new TurKL(...);
}
```

Be sure that you understand the syntax and the implications of such complex declarations. When possible, consider using other coding strategies.

3.7 Header inclusion

To refer to a given header, use the syntax:

```
#include "SomeModule/SomeSubDir/SomeHeader.h"
```

To compile, the Makefile system provides the correct include path, for example:

```
-I$ELSAWKSP/Kernel/src -I$ELSADIST/lib/include
```

3.8 Include guards

Wrap headers in `#ifndef / #endif` guards to minimize the chance of clashes with macro names from other's code. Use the following convention: for a class `DefError`, defined in the header file `Def/Error/DefError.h`:

```
#ifndef _DEF_ERROR_DEFERROR_H_
#define _DEF_ERROR_DEFERROR_H_
...
#endif
```

3.9 Avoid including unnecessary headers

Avoid including unnecessary headers, in order to:

- decrease compile time;
- avoid unnecessary recompilation because of false dependency; when possible [6] inside header files, prefer forward declaration: `class SomeClass;` to explicit inclusion of the file defining the type:

  File `Def/Error/DefError.h`:

  ```
  class DefErrorTable;
  ...
  class DefError
  {
  ...
     static DefErrorTable* _table;
  };
  ```

  In this example, the file `DefErrorP.h` , which defines `DefErrorTable`, is not included by `DefError.h`; instead, it is included by `DefError.C`.

[6] this is not possible with `typedef`

3.10 Use E_Float, E_Int, E_Bool

To improve portability :

- To declare floating point variables, use `E_Float` instead of `float` (or `double`).
- To declare integer variables, use `E_Int` instead of `int` (or `long`).
- To declare boolean variables, use `E_Bool` instead of `bool`. `E_Bool` entities can take two values: `E_True` or `E_False` (instead of `true` and `false`).

`E_Float`, `E_Int` and `E_Bool` are typedef in `Def/Global/DefTypes.h`

The exceptions to this rule are:

- interaction with system calls (`main(), int err = syssx(HTIMES, &_htms); ...`);
- interaction with MPI calls : use `E_MPI_INT`, `E_MPI_FLOAT`;
- postfix operator definition (`DefError& operator++(int)`).
- interaction with Python (internally, Python uses `int` and `double`):

```
FactDataBase.C:
int ret = PyArg_Parse(attr, "l", &val);


DesBase.C
int DesBase::getI(const char* key) const
{
   return static_cast<int> (getI(TbxString(key)));
}

double DesBase::getF(const char* key) const
{
   return static_cast<double> (getF(TbxString(key)));
}
```

SWIG takes care of the correct binding.

3.11 C++ Variable declaration and initialization

3.11.1 Declare variable immediatly before use

It is much better to declare variables immediatly before use since:

- variables will not be known outside current scope, thus avoiding potential name conflicts and typing errors;
- code understanding is easier: you do not have to look at function’s beginning to look for a declaration;
- maintenance is easier.

3.11.2 Avoid implicit conversion between floats and integers

Prefer:

```
E_Float b = 17.;
```

to:

```
E_Float b = 17;
```

Type conversions are confusing, and may lead to portability problems.

3.11.3 Initialize variable as soon as possible

Initialize variable as soon as possible; prefer:

```
E_Int n = 0;
SomeClass* ptr = E_NULLPTR;
```

to:

```
E_Int n;
SomeClass* ptr;
...
n = 0;
ptr = sommeAddress;
```

This rule avoids many potential bugs (see 3.1.5) and portability problems (for example, SGI compiler set unitialized pointer to `0x1`, which is different from `NULL`, which is `0x0`).

3.11.4 If possible, declare local variables const

Use const qualifier as much as possible for local variable declaration (this will remove many `const_cast`, see 3.24.1).

Note:
`const` qualifier for primitive type is not required.

3.11.5 Remove unused variables (declared and not used)

Look carefully to compiler messages such as (NEC sxc++):

```
warning(177): variable "x" was declared but never referenced
```

Remove the declaration of variables that you don’t use anymore. Code readers will not waste time figuring out what was the intent of the programmer.

3.11.6 Pointer and reference declaration

Prefer this style:

```
SomeClass* p1 = &someObject;
SomeClass& p1 = &someObject;
```

to:

```
SomeClass * p1 = &someObject;
SomeClass & p1 = &someObject;
```

This forbids declaration of several pointers (or reference) such as:

```
BAD declaration
SomeClass *a1, *a2, ..., *an;
```

In fact, this syntax would also violate the rule "Initialize variable ASAP" (see 3.11.3).

3.12 Avoid hard-coded value in C++ code

Depending on context, replace hard-coded value by:

- a class `enum`; this may be convenient to code array dimension at compile time:

  ```
  enum { BUF_SIZE=80 };
  char buff[BUF_SIZE];
  ```

- `static` class attribute:

  ```
  class OperBase {
    static const E_Int E_MAX_OPERTOCOMP;
    ...
  };
  ```

  The initialization of the static constant cannot be done inside the class definition. So, contrary to `enum` constant, it cannot be used to code array dimension. However, not storing the actual value in the header file may be convenient (no recompilation required if the constant value must be changed).

- inside C++ body file, an "internal" `const` defined in file scope:

  ```
  const TbxString PREFIX_WINDOW = TbxString("E_W_");
  ```

- some widely used **global constants**, declared in `DefCplusPlusConst.h`

  ```
  extern const E_Float ONE_HALF;
  extern const E_Float E_PI;
  ```

  These constants are defined in `DefCplusPlusConst.C`.

- finally, a few **global variables** declared in `DefCplusPlusGlobal.h`; for example:

```
...
extern E_Float E_ZERO_MACHINE = 1.e-15;
extern E_Float E_CUTOFF_GEOM  = 1.e-8;
extern E_Float E_MIN_SURFACE  = 1.e-30;
extern E_Float E_MIN_VOLUME   = 1.e-30;
extern E_Float E_TOLNWT       = 1.e-5;
...
```

Take care specially of "cutoff" values: do **not** introduce your own "personal" cutoff values, hidden inside implementation code!

3.13 Use log messages with care:

1. To log a message , you need to include `DefIostream.h` (it is not included by `DefInclude.h`, except in `DEBUG` mode).
2. Remove any debug messages prior to integration.
3. Never use `printf`;
4. Do not use `cout` or `cerr` log statements: il will clutter the log file, specially in Parallel MPI mode. Use macro `e_log` instead; however, most of the time it is better to avoid using `e_log` directly: consider using `DefError`.
5. Use `DefError` mechanism to provide, in a uniform way, useful information. See next section 3.14.

Note:
Macro `e_log` should be replaced by more sophisticated `iostream` manipulations.

3.14 How to introduce new error messages

For historical reasons, there exist two kinds or kernel errors:

1. Errors associated with a unique identifier. Such errors correspond to a unique event, and provide usually an informative error message. This error message is the concatenation of local information (for instance, an invalid value at runtime) with a pre-defined string, defined in `Def/Error/DefErrorP.h`.

   ```
   DesBoundary.C:
   DesBoundary::getState() const
     ...
     if (desState == E_NULLPTR)
     {
       TbxString mesg  = "Bad State Object for boundary : '" + _name;
                 mesg += "\nCheck state object name.";
       DefError error(2194); error++; error.raiseError(&mesg);
     }

   DefErrorP.h:
   E_ERRENTRY(2194, E_FATAL, "User Error. ", "Descp/Bnd/DesBoundary.C", "")
   ```

   Unfortunately, due to lack of time (and motivation), it is tedious to search for an unused error identifier; moreover, it is quite time-consuming to check if all the registered error numbers are still valid.

2. Errors using the `__FILE__` and `__LINE__` preprocessor macro:

```
DesCfdPb.C:
TbxString mesg("Either key 'cfd_flow_root_dir' or "
               "'cfd_init_state_global' should be set.");
DefError::userError(__FILE__, __LINE__, &mesg);
```

Here, developers do not have to search for a unique error number. Admittedly, it is much more difficult to achieve a uniform error reporting style, since error messages are scattered inside the source code. Different forms are available:

- `DefError::internalError`
- `DefError::internalWarning`
- `DefError::userError`
- `DefError::notImplemented`
- `DefError::invalidNumericValue`
- `DefError::logicalError`

In addition, several helper functions are available to factorize treatment of frequent user errors (coming from Python scripting interface):

- `DesBase::missingKey`

```
void DesBase::missingKey(const TbxString& key) const
{
  TbxString mesg("Object : '");
  mesg += _name + TbxString("' needs Key : '") + key + "'";
  DefError error(2112); error++; error.raiseError(&mesg);
}
```

- `DefError::badAttr`: bad value

```
void DefError::badAttr(const TbxString& name,
                       const TbxString& attr, const TbxString& value,
                       const TbxString* mesg_complement)
{
  TbxString mesg =  "Object : '" + name + "' : '"
                  + value + "' is not a valid value for at-
tribute : '" + attr + "'";
  if (mesg_complement) e_log << *mesg_complement << endl;
  DefError error(2010); error++; error.raiseError(&mesg);
}
```

3.15 Character strings

3.15.1 Use TbxString (instead of char∗)

To manipulate character strings, elsA uses the C++ (template) class `string`, fully defined in the C++ standard (<`string`>). Not using `char`∗ avoids many nasty bugs.

To improve portability (and possible future evolution), it is better to use `TbxString` (which is just `typedef` to `string`) instead of `string`. Use of `TbxString` is restricted to a few modules (`Fact`, `Sio`, `Descp`, `Def`).

All other modules are not allowed to include directly `TbxString.h`. They may instead include `DefError.h`, to build informative error messages.

Note:

`TbxString` may be renamed to `DefString` (and moved to `Def` module) in future release.

3.15.2 Choice between sprintf and stringstream

Presently, elsA does **not** handle internal string buffer operation in a consistent way.

- use of sprintf (`PcmTaskMpi.C`):

```
char buffId[20];
sprintf (buffId, "_Pid_%d", static_cast<int>(runId));
```

- use of stringstream (`SioAccessTec.C`)

```
#ifdef _E_USE_STANDARD_IOSTREAM_
  std::istringstream inputStringI(title);
#else
  istrstream inputStringI(title.c_str());
#endif
  inputStringI >> dimI;
```

In principle, `stringstream` should be preferred to `sprintf`, since it is type-safe (no ugly `static_cast`). However, presently we do not know how to use it in a fully portable way.

3.16 Constructor

3.16.1 Constructor implementation

- A constructor must initialize all internal private data, preferably as soon as possible (See 3.11.3), i.e. in list of initializer. Use default value (`E_BADVALUE_F`, `E_BADVALUE_I`, `E_NULLPTR`) if no argument is available. Respect the same ordering in class declaration and constructor implementation:

```
File SomeClass.h:
class SomeClass
{
  SomeClass(E_Int);
...
private:
  E_Int         _i;
  E_Float       _f;
  AnOtherClass* _o;
};


File SomeClass.C:
SomeClass::SomeClass(E_Int i)
 : _i(i),
   _f(E_BADVALUE_F),
   _o(E_NULLPTR)
{
  some code ...
}
```

Initializing pointer to `E_NULLPTR` is specially important, since deleting a `NULL` pointer is always harmless.

- If the class inherits from an ancestor, you must call explicitly base class constructor [7], before the initializer list:

```
SomeDerivedClass::SomeDerivedClass(E_Int i)
 : Base(i),
   _f(E_BADVALUE_F),
   _o(E_NULLPTR)
{
  some code ...
}
```

3.16.2 Copy Constructor (and assignment operator) declaration

Copy constructor, assignment operator and other class operators are not used heavily in elsA. In fact, since class arguments are passed by address (rule discussed in 3.6.5), there should be no implicit calls of copy constructor, except those resulting from return-by-value [8]. Nevertheless:

- Each class must declare the copy constructor and the assignment operator, thus avoiding implicit generation (by the compiler) of potentially incorrect code.
- If no implementation is provided, both must be declared `private`; this will prevent passing object arguments by value, in case of typing error (see 3.6.5).
- Consider writing copy constructor for any class that contains a handle (generally pointer or reference to other data, see 3.4).

3.17 Virtual methods

3.17.1 Virtual methods declaration

Although not required, the "virtual-ness" of a method must be explicitly written in derived classes:

```
File Base.C:
class Base
{
  virtual void f();
};

File Derived.C:
class Derived: public Base
{
  virtual void f();
};
```

3.17.2 Do not ask an object its type

Method such as `BndBase::getBndType()`, `OperSource::isSouTransp()` or `OperBase::get-Identity()` are bad:

[7] Even if it has no argument, and would have been called (implicitly) anyway by the compiler.
[8] Classes `FldField` and `FldArray` are important exceptions

```
class BndBase
{
virtual BndType getBndType() const;
};

class BndEuler
{
virtual BndType getBndType() const {return bndEulerWall;}
}
```

Such methods are clear signs of wrong design. They remove the benefit of polymorphism. by (re-)introducing explicit logic switch:

```
TurKEps.C, but also TurKL.C, TurSA.C, ...
{
   ...
   BndType bndType = bnd.getBndType();
    const TurWallLine* wallLine = E_NULLPTR;
    if ( (bndType == bndViscAdiabaticWall   ) ||
         (bndType == bndViscIsothermWall    ) )
    {
         wallLine = _turCriteria->getWallLine(bnd.getGeoWindow());
    }

    else if ( (bndType == bndViscAdiabaticWallLaw) ||
              (bndType == bndViscIsothermWallLaw ) )
    {
      ...
    }
```

Such conditional statements have high maintenance costs.

3.17.3 Do not redefine inherited non virtual methods

A non virtual method is an *invariant*: every derived class inherits both interface and implementation. Prescribing an invariant behaviour is often useful, for example to implement a fixed protocol ("template" design pattern). This means that redefining a inherited non virtual method is a design error:

- either the method should be virtual,
- or the inheritance relation should be removed.

This rule will avoid potential errors:

```
class Base
{
public:
  void f() const {cout << "fOfBase\n";}
};

class Derived : public Base
{
```

```
public:
  void f() const {cout << "fOfDerived\n";}
};


int main()
{
  Base    b;
  Derived d;

  Base* pb = &b;
  Base* pd = &d;

  b.f();
  d.f();

  pb->f();
  pd->f();
}
```

At run time:

```
fOfBase
fOfDerived
fOfBase
fOfBase  // oops
```

3.17.4 Do not redefine inherited argument default value of virtual methods

```
class JoinBaseP
{
  ...
  virtual E_Bool
  putCurrentCellValue(const vector<AuxField>& keys,
                      E_Int               currentDepth,
                      E_Int               oppositDepth,
                      GeoCfdField&        cfdField,
                      E_Int               currentMaxDepth = -1,
                      E_Int               oppositMaxDepth = -1) const
}
```

It would be an error to redefine `currentMaxDepth` and `currentMaxDepth` default value (`-1`) to another value in derived classes (for example in `JoinAdjacentP`).

3.17.5 Do not overload virtual methods in derived classes

Note:

Most compilers will flag a warning (or maybe an error) in this situation.

```
class Base
{
public:
```

```
  virtual void f() const {cout << "fOfBase\n";}
};

class Derived : public Base
{
public:
  // "Typing" error : const removed (special case of overloading)
  virtual void f()       {cout << "fOfDerived\n";}
};

int main()
{
  Base    b;
  Derived d;

  Base* pb = &b;
  Base* pd = &d;

  b.f();
  d.f();

  pb->f();
  pd->f();
}
```

At run time:

```
fOfBase
fOfDerived
fOfBase
fOfBase  // oops
```

Remarks:

The base class interface itself may provide several overloaded virtual methods (same name, different arguments): see for example `BndPhys` (virtual overloaded method: `compBoundaryValues()`) or `OperFlux` (virtual overloaded method: `compInterior()`). This is perfectly legal, but maybe a little confusing. We should probably use non overloaded virtual methods in future release.

3.18 Destructor

3.18.1 Virtual destructor

Destructor of classes with `virtual` functions must be declared `virtual`. This avoids improper destruction of derived class objects.

3.18.2 Destroying array of objects

Use the appropriate syntax when deleting array of objects

```
A* array1A = new A[10];
A* array2A = new A[10];
```

```
delete    array1A; // incorrect: destructor called only once
delete [] array2A; // correct  : destructor called 10 times
```

3.18.3 Use covariant return type

Covariant return type avoids many ugly downcasts (see 3.24.1). Unfortunately, some compilers do not yet implement them. To obtain portable code, use the following technique:

```
class GeoGrid : public GeoGridBase
{
  ...
  #ifdef E_NO_COVARIANT_RETURN
  /** */
  virtual const GeoMetricsBase* getMetric()  const;
  #else
  virtual const GeoMetrics*     getMetric()  const;
  #endif
}
```

3.19 Dynamic memory allocation

3.19.1 Do not use malloc/free

Do not use the C memory allocation/deallocation functions, `malloc` and `free`. Use `new` and `delete` instead.

3.19.2 Prefer Fld objects (FldArray, FldField) to C arrays

To allocate dynamically the memory needed to store floats or integers, prefer `Fld` objects (`FldArray[FIB], FldField[FIB]`) to C arrays. For example, do not use:

```
E_Float* f = new E_Float[100];
E_Int*   i = new E_Int  [100];
```

Instead:

```
FldField f(100);
FldField i(100);
```

`Fld` classes provide several advantages (see elsA tutorial):

1. Easy control of memory initialization.
2. Check of memory access: in `DEBUG` mode, we can check that access to container elements is valid. For example

```
inline E_Float
FldArrayF::operator()(E_Int l, E_Int fld) const
{
  // PRE CONDITION
```

```
  assert (_data  !=  E_NULLPTR);
  assert (l >= 0);
  assert (l < _sizeLoc);
  assert (fld >= NUMFIELD0);
  assert (fld <= _nfldLoc);

  return (_data[(fld-NUMFIELD0)*_sizeLoc + l]);
}
```

3. Dynamic memory monitoring: `Fld` memory usage can be easily tracked and displayed (utility class `FldMemory`). For example:

```
================================================================
------------------ Heap Memory Usage Summary -------------------
================================================================
Total Number of new calls =         2038
          new [E_Float  ] =         1690
          new [E_Int    ] =          310
          new [E_Bool   ] =           38
================================================================
Maximum total allocated memory             :     10347008 bytes
Maximum allocated memory (float)           :     10271176 bytes
Maximum allocated memory (int)             :        83664 bytes
Maximum allocated memory (bool)            :        45056 bytes
----------------------------------------------------------------
----------------------------------------------------------------
----------------------------------------------------------------
```

   When any new development is reviewed before integration, it has proved very valuable to look at its memory footprint.

4. On vector machines, if highest efficiency is required, it may be useful to decrease the number of memory allocation function calls performed inside each iterative cycle. For this purpose, users can create an internal stack, where temporary memory is stored: instead of the penalty associated with `new / delete` calls, we now have a single pointer arithmetic operation:

```
E_Float* FldStack::acquireF(E_Int chunk) { _allocFloat += chunk; }
E_Float* FldStack::releaseF(E_Int chunk) { _allocFloat -= chunk; }
```

   To use this feature, you must use specific `FldField` constructor:

```
FldFieldF(E_Bool useStack, E_Int size, E_Int nfld);
FldFieldF(E_Bool useStack, GeoEntity geoLocation, E_Int size, E_Int nfld);
FldFieldI(E_Bool useStack, E_Int size, E_Int nfld);
```

   At compile time, C++ uses a complex mechanism to select which overloaded method to call. So, be very careful when invoking these constructors; note that they do **not** provide default values:

```
FldFieldF f(100,1); // will allocate array of size (100 X 1) on the heap
FldFieldF f(100);  //  identical

FldFieldF(E_True, 100, 1); // will use stack memory to allocate ar-
ray of size (100 X 1)
FldFieldF(E_True, 100);    // WRONG ! will allocate an ar-
ray of size (1 X 100) on the heap !!!
```

5. Efficient "array-like" syntax "a la Fortran 90". Many convenient arithmetic operations are available, with complete control of the implementation; in fact, code is at least as efficient as the equivalent Fortran 90, and probably better in certain cases (no temporary objects are created).

   - `FldFieldF lhs = scalar * fld`
   - `FldFieldF lhs(...); lhs = scalar * fld`
   - `FldFieldF lhs(...); lhs *= scalar * fld`
   - `FldFieldF lhs(...); lhs += scalar * fld`
   - `FldFieldF lhs = fld1 + fld2`
   - `FldFieldF lhs(...); lhs = fld1 + fld2`
   - `FldFieldF lhs = fld1 - fld2`
   - `FldFieldF lhs(...); lhs = fld1 - fld2`
   - `FldFieldF lhs = fld1 * fld2`
   - `FldFieldF lhs(...); lhs = fld1 * fld2`
   - `FldFieldF lhs = fld1 / fld2`
   - `FldFieldF lhs(...); lhs = fld1 / fld2`
   - `FldFieldF lhs(...); lhs *= fld1 / fld2`
   - `FldFieldF lhs = (fld1 * fld2) + r3`
   - `FldFieldF lhs(...); lhs = (fld1 * fld2) + r3`
   - `FldFieldF lhs = scalar * (fld1 / fld2)`
   - `FldFieldF lhs(...); lhs = scalar * (fld1 / fld2)`
   - `FldFieldF lhs(...); lhs *= scalar * (fld1 / fld2)`
   - `FldFieldF lhs = (scalar * fld1) + fld2`
   - `FldFieldF lhs(...); lhs = (scalar * fld1) + fld2`
   - `FldFieldF lhs(...); lhs *= (scalar * fld1) + fld2`
   - `FldFieldF lhs = scalar1 * fld1 + scalar2 * fld2`
   - `FldFieldF lhs(...); lhs = scalar1 * fld1 + scalar2 * fld2`
   - `FldFieldF lhs(...); lhs *= scalar1 * fld1 + scalar2 * fld2`

   See `FldField.h` and `FldFieldOptim.h` for a thorough discussion.

3.20 Storage ownership responsibility

To be written (and checked !!!)

To avoid memory leaks or runtime crashes, it is very important to clearly define the storage ownership responsibility.

3.20.1 Deleting kernel objects

Presently, most kernel objects created inside the Factory are *not* destroyed explicitly. This has to be corrected in future release.

3.20.2 When possible, the creator is responsible of the object's destruction

3.20.3 Adopt semantics

To be written

3.21 Do not use C++ Template (except standard library templates)

C++ templates are potentially very powerful, and can provide a useful tool to improve software design and implementation (generic programming). In fact, they have been used in the first releases of elsA. However, we have changed our mind, and programmers are not allowed to introduce new templates, for the following reasons:

- increased compilation time;
- increased link time and complexity (template instantiation problem);
- cryptic error messages; for example:

```
cc-1132 CC: ERROR File = .Obj/sgi_r8/Base/FactTurb.C, Line = 160
  The class "std::_Rb_tree_iterator<std::_Rb_tree<std::map<FactBase::TurTypeId,
          FactBase::PtrVirtConsTur, std::less<FactBase::BndTypeId>,
          std::allocator<FactBase::PtrVirtConsTur>>::key_type,
          std::map<FactBase::TurTypeId, FactBase::PtrVirtConsTur,
          std::less<FactBase::BndTypeId>,
          std::allocator<FactBase::PtrVirtConsTur>>::value_type,
          std::_Select1st<std::map<FactBase::TurTypeId,
          FactBase::PtrVirtConsTur, std::less<FactBase::BndTypeId>,
          std::allocator<FactBase::PtrVirtConsTur>>::value_type>,
          std::map<FactBase::TurTypeId, FactBase::PtrVirtConsTur,
          std::less<FactBase::BndTypeId>,
          std::allocator<FactBase::PtrVirtConsTur>>::key_compare,
          std::allocator<FactBase::PtrVirtConsTur>>::value_type,
          std::_Rb_tree<std::map<FactBase::TurTypeId,
          FactBase::PtrVirtConsTur, std::less<FactBase::BndTypeId>,
          std::allocator<FactBase::PtrVirtConsTur>>::key_type,
          std::map<FactBase::TurTypeId, FactBase::PtrVirtConsTur,
          std::less<FactBase::BndTypeId>,
          std::allocator<FactBase::PtrVirtConsTur>>::value_type,
          std::_Select1st<std::map<FactBase::TurTypeId,
          FactBase::PtrVirtConsTur, std::less<FactBase::BndTypeId>,
          std::allocator<FactBase::PtrVirtConsTur>>::value_type>,
          std::map<FactBase::TurTypeId, FactBase::PtrVirtConsTur,
          std::less<FactBase::BndTypeId>,
          std::allocator<FactBase::PtrVirtConsTur>>::key_compare,
          std::allocator<FactBase::PtrVirtConsTur>>::const_reference,
          std::_Rb_tree<std::map<FactBase::TurTypeId,
          FactBase::PtrVirtConsTur, std::less<FactBase::BndTypeId>,
          std::allocator<FactBase::PtrVirtConsTur>>::key_type,
          std::map<FactBase::TurTypeId, FactBase::PtrVirtConsTur,
          std::less<FactBase::BndTypeId>,
          std::allocator<FactBase::PtrVirtConsTur>>::value_type,
          std::_Sel      turBase = iter.second;
```

- portability problems.

The only allowed template classes come from the standard library (STL containers, `string` and `iostream`).

Note:

This choice can be reversed when compiler and linker technology will be mature enough. In elsA, templates may be specially useful to implement in a generic way:

- smart pointers;
- singleton design pattern;
- generic factory (see elsA tutorial).

3.22 Use of STL container

3.22.1 Do not include directly STL headers (such as <list>).

Instead, to use (templatized) STL containers, you must include the `Def/Global/DefCompiler.h` header:

```
#define _ISO_MAP_
#include "Def/Global/DefCompiler.h"
```

You can use the predefined macros (`DefCompiler.h`):

elsA Macro	STL container	Comments
`_ISO_LIST_`	`list`	doubly-linked list
`_ISO_STACK_`	`stack`	
`_ISO_QUEUE_`	`queue`	
`_ISO_VECTOR_`	`vector`	
`_ISO_MAP_`	`map`	associative container
`_ISO_PAIR_`	`pair`	

Note:

In future release, we may replace this "macro + include" syntax by:

`#include "Def/Global/DefList.h"`

where `DefList.h` (`DefVector.h`,...) will simply include the correct STL header.

3.22.2 Use STL container iterators in a systematic way

- Example 1:

```
FactTurb.C:
  DicoTur::const_iterator iter = _allTurCtor.find(TurModelId);
  if (iter == _allTurCtor.end())
  {
    error message
  }
  else
  {
    turBase = iter->second (desNSDGlb, desModGlb, desBlock, grid,
                            vectorWindows, vecWakes, vecWalls, vecTurCrite-
riaFile,
                            coeffMutInit);
  }
```

See 3.1.5 for a `for-loop` example.

Note:
`const_iterator` should be understood as a pointer-to-const.

3.22.3 Macro E_CONST_ITERATOR

Interpretation of semantics of `const_iterator` appears to be controversial:

```
class A
{
public:
  A(int i) {_i = i;}
  void show() const {cerr << _i << endl;}

private:
  int _i;
};

void f(list<const A*>& l)
{
  for (list<const A*>::const_iterator iter  = l.begin();
                                      iter != l.end();
                                      iter++)
    (**iter).show();
}
```

Contrary to most C++ compilers, with `aCC` (HP-UX C++ compiler), this code does not compile:

```
Error 902: "t2.C", line 21 # Template deduction failed to find a match
for the call to 'operator !=' with signature
'void (list<const A *,allocator>::const_iterator,list<const A *,allocator>::iterator)'.
                                        iter != l.end();
```

Here, a simple (and recommended) solution is to declare reference-to-const the argument to function f, `l`. However, in many situations this is not possible without unpleasant `const_cast`. In this situation, you will have to use the `cpp` symbol `E_CONST_ITERATOR` (defined in `DefCompiler.h`) instead of `const_iterator` [9].

3.23 Do not use C++ exception

To improve portability, and to improve (slightly) CPU efficiency, do not use C++ exceptions.

3.24 How to use cast

3.24.1 Avoid using cast

Avoid as much as possible using `casts`:

[9] see also `const_IdentIterator, IdentIterator` in `OperTerm::computeAllFields`

- to avoid `const_cast`, use `const` declaration in a consistent manner;
- to avoid `dynamic_cast`, use polymorphic objects in a sensible way.

3.24.2 Use C++ cast (instead of C-like cast)

When casts are really unavoidable, use the C++ qualified cast:

```
const_cast<SomeType*>,   const_cast<SomeType&>
static_cast<SomeType*>,  static_cast<SomeType&>
dynamic_cast<SomeClass*> dynamic_cast<SomeClass&>
```

`dynamic_cast` is not supported by all compilers (notably NEC SX5). So we have defined the macro `E_DYNAMIC_CAST` (file `Def/Global/DefInclude.h`):

```
#ifdef E_RTTI
#define E_DYNAMIC_CAST(a) dynamic_cast<a>
#else
#define E_DYNAMIC_CAST(a) (a)
#endif
```

Old-style casts are sorely lacking in type safety, suffer poor readability, and are difficult to locate with search tools.

3.25 Prefer C++ methods to cpp preprocessor Macros

When possible, avoid using preprocessor macro:

- Macros are hard to debug.
- Macros are difficult to understand.
- They have no scope, so that potentially dangerous name conflicts may arise.

The preprocessor is useful to:

- Tune log file (serial or MPI) through `e_log` macro;
- Implement include guard (see 3.8).
- Implement platform dependent code (see elsA tutorial) :
  - Covariant return type (see 3.18.3).
  - STL minor differences.
  - `div_t` (E_DIV_T) portability (HP-UX).
- Isolate MPI-dependent code in C++ files (macro `E_MPI`, see 3.28.2).
- Select different codings of the same algorithm for scalar and vector machines: macro `E_SCALAR_COMPUTER` (see 4.19.2).
- Select between Fortran and C++ coding of the same loop: `_E_FORTRAN_LOOPS_` (see 3.27.1).
- Simplify somewhat the typing effort required to register object creation functions (see elsA tutorial).

3.26 How to write portable C++ code

In this section, we summarize the most important rules which have to be followed to obtain portable C++ code.

3.26.1 Do not include system headers, except in Def module

System header files (for instance, `stdlib.h, signal.h`) are included only once, by a small number of header files belonging to the `Def` module. When required in other modules, these `Def` headers must be included, not the system ones. For instance:

```
#include "Def/Sys/DefSignal.h"
#include <signal.h>  # WRONG !
```

3.26.2 Respect function return type

Some compilers (SUN) do not like this:

```
E_Int f()
{
  if (...)
  {
    DefError(...); // Fatal error
  }
  else
  {
    return 1;
  }
}
```

Prefer this code:

```
E_Int f()
{
  if (...)
  {
    DefError(...); // Fatal error
    return 0;      // will never be executed, anyway
  }
  else
  {
    return 1;
  }
}
```

3.26.3 Numerical "helper" functions

Prefer `inline` functions defined in `Def/Global/DefFunction.h` (namespace `E_FUNC`). `inline` fonctions are safer than traditional C macros.

```
inline E_Float E_min(E_Float a, E_Float b)
inline E_Int   E_min(E_Int a, E_Int b)
inline E_Float E_max(E_Float a, E_Float b)
inline E_Int   E_max(E_Int a, E_Int b)
inline E_Float E_abs(E_Float a)
inline E_Int   E_abs(E_Int a)
inline E_Float E_sign(E_Float a)
inline E_Int   E_sign(E_Int a)
inline E_Bool  fEqual(E_Float lhs, E_Float rhs,  E_Float precision = E_CUTOFF )
inline E_Bool  fEqualZero(E_Float lhs,  E_Float precision = E_CUTOFF )
```

Note:

In future release, `E_min` and `E_max` may be replaced by STL templates, `std::min` and `std::max`.

3.27 Optimization of C++ code

Since most computationaly intensive tasks are coded in Fortran, impact of C++ code on overall performance is usually relatively minor. However, a few specific rules should be followed.

3.27.1 C++ code vectorization

NEC and Fujitsu C++ compiler can vectorize very simple loops. For example:

```
BndEuler::compWb1(...)
{ ...
  if (eqNb > 5)
  {
    for (E_Int eqInd = 6; eqInd <= eqNb; eqInd++)
    {
      E_Float* ptWb1 = wb1.begin(eqInd);
      const E_Float* ptWbs = wbs.begin(eqInd);
      // vectorized loop
      for (E_Int lint = 0; lint < intNbB; lint++)
        ptWb1[lint] = ptWbs[lint];
    }
  }
}
```

Nevertheless, most loops have been coded in Fortran. For historical reaons, the same loop is sometimes coded in two variants (C++ and Fortran):

```
#ifdef _E_FORTRAN_LOOPS_
      E_Float gamOverPr = gam/pr;
      compcvl_(ncell, im,jm,km, gamOverPr, _wcons.begin(), mu->begin(),
               cvf.begin());
#else
      for (E_Int l=0; l< ncell; l++)
        cvf[l] = gam / _wcons(l,1) * (*mu)[l]/pr;
```

There is no justification for this code duplication, except that, if some day C++ compilers generate good enough code, the C++ code will be available.

3.27.2 C++ code CPU optimization

- For `Fld` objects used locally (typically as work array), consider using the internal stack option (see 3.19.2).
- Do not introduce dynamic size STL containers if it is not really required.
  - For example, some earlier versions of elsA use the following attribute:

```
class EosSysEq
{
public:
  E_Int getGlobalPosition(E_Int ident) const;
private:
  vector< map<E_Int,E_Int > > _sysOfEquation;
  vector<E_Int>               _nbOfEquationsBySystem;
```

A lot of time was spent in method such as `getGlobalPosition()`:

```
E_Int EosSysEq::getGlobalPosition(E_Int ident) const
{
  E_Int  position=0;
  for (E_Int i=0; i< _nbOfSubSystem; i++)
  {
    map<E_Int,E_Int >::iterator iter;
    iter = _sysOfEquation[i].find(ident);

    if (iter != _sysOfEquation[i].end())
    {
      position +=(*iter).second;
      break;
    }
    position += _nbOfEquationsBySystem[i];
  }
  return position;
}
```

Newer versions solve the performance bottleneck with `FldArrayI` containers:

```
    FldArrayI _sysOfEquation;
    FldArrayI _nbOfEquationsBySystem;
    FldArrayI _globalPosition;
    FldArrayI _localPosition;
```

and using `inline` methods:

```
inline E_Int EosSysEq::getGlobalPosition(E_Int ident) const
{
  return _globalPosition[ident];
}
```

  - Avoid inserting attribute of complex types (costly at instantiation time) inside widely used classes. Earlier version of `FldFieldF` class included an attribute, `_agtDescp`, of type `AgtDescp*`. Some `FldFieldF` constructors had to call `AgtDescp` constructor:

```
FldFieldF::FldFieldF(const FldFieldF& rhs)
 : _array(rhs._array),
   _agtDescp(E_NULLPTR)
{
  if (rhs._agtDescp)
    _agtDescp = new AgtDescp(*rhs._agtDescp);
}
```

which itself invokes a constructor of a STL container (`vector<AgtType>`). Removing this attribute has improved slightly CPU efficiency, (and also has suppressed the unfortunate coupling between `FldFieldF` and `Agt` module).

3.28 MPI insulation

3.28.1 Do not include directly MPI header mpi.h

Do not include directly MPI header `mpi.h`. Instead, include `Pcm/Base/PcmDefMpi.h`.

3.28.2 Use Macro E_MPI

To isolate MPI-dependent code in C++ files, use macro `E_MPI`. If MPI-specific code becomes too large, it can be quite difficult to understand the code logic; in such cases, consider introducing new files, which will be compiled only in MPI production. For example (`Pcm/Make_Obj.mk`):

```
E_PCM_COMMON_OBJS=\
Base/PcmMain.o\
Base/PcmBuffer.o\
...

E_PCM_ADD_MPI_OBJS=\
Base/PcmMPIReduce.o\
Base/PcmTaskMPI.o\
Base/PcmGroupMPIP.o\
...

E_LIBOBJLIST= $(E_PCM_COMMON_OBJS)

# this value replaces the previous one, the selection is done into
# the cfg/MakeMake.mk script

#PCM_MPI#E_LIBOBJLIST= $(E_PCM_COMMON_OBJS) $(E_PCM_ADD_MPI_OBJS)
```

3.29 Calling Fortran subroutine

Calling Fortran from C++ is straightforward; however, it is error prone. To avoid coding errors as much as possible, it is very important to follow strictly several rules.

Prior to being called from C++, Fortran subroutine must be declared; for example, for a Fortran subroutine called SOME_SUB (or some_sub, case is **not** significant in Fortran, contrary to C/C++):

```
extern "C"
{
  void
  some_sub_(E_Int&   i1, E_Int&   i2,
            E_Float& f1, E_Float& f2,
            E_Int* indir,
            const E_Float* input_array,
                  E_Float* output_array);
}
...
  E_Int i1, i2; ...; E_Float f1, f2; ...
  FldFieldI indir (...);
  FldFieldF input (...);
  FldFieldF output(...);
  some_sub_(i1, i2, f1, f2,
            indir,
            input.begin(), output.begin());
}
```

The C++ compiler can check that the declaration and call of `some_sub_` are consistent. However, it is very important to note that neither the Fortran compiler, nor the C++ compiler is able to check if arguments are passed correctly **from C++ to Fortran**. This "weakness", due to the mixing of C++ and Fortran, can unfortunately generate bugs, and it is of the programmer's responsability to check that the types of the arguments decalared by Fortran subroutine correspond to correct arguments in the declaration of this subroutine in the `.C` file [10].

3.29.1 Passing scalar (E_Int or E_Float) arguments

Scalar arguments must be declared as reference.

```
  extern "C" void some_sub_(E_Int& i1,...
  ...
  E_Int i1 = ...;
  some_sub_(i1,...);
```

Remarks:

- `const` before scalar argument declaration is optional; since scalars are nearly always kept unchanged inside Fortran subroutines, we suggest to avoid using `const` for scalar arguments, since it will reduce the amount of typing.
- Note that forgetting a single `'&'` in the declaration leads to disaster:
  - the C++ compiler does not complain, and generate code to pass argument by **value**;
  - the Fortran compiler expects an adress; so it will dereference the passed argument...
- An other valid syntax would be:

  ```
  extern "C" void some_sub_(E_Int* i1,...
  ...
  int i1 = ...;
  some_sub_(&i1,...);
  ```

  In order to keep a unique coding style, do **not** use this alternative syntax.

[10] It should be possible to write an external tool, using some simple parsing techniques, which would be able to perform cross-check, both in C++ and Fortran files. This tool would be very useful.

3.29.2 Passing array arguments

Array arguments are passed as pointers. "Bare" arrays are never used directly inside elsA C++, since they are encapsulated in Fld objects (see elsA tutorial). Method `begin()` of class `FldField` is used to pass the underlying array to Fortran. Please see elsA tutorial for explanation of `begin()`.

3.29.3 Argument ordering

Checking argument ordering is boring. To avoid as much as possible errors when calling Fortran subroutines from C++ (see also 4.7), follow the following convention for the ordering of arguments passed to the subroutine:

1. `E_Int` scalars;
   - pass grid dimensions first (`ncell`, `im`, `jm`, `km`);
   - then equation numbering: `neqTot`, `neq`, `rho`, `rou`,...
2. `E_Float` scalars;
3. `E_Int*` arrays;
4. `E_Float*` arrays.

Inside each category, put `IN` arguments before `OUT` ones.

3.29.4 Declare Fortran subroutine name as lowercase

To avoid error at link time ("unresolved symbol"), you must declare Fortran subroutine names in lowercase.

3.30 C++ common coding errors

3.30.1 Memory corruption: Pay attention to '()' vs '[]'

```
int* pi = new int(3);  // initialise pi[0] with value '3'
int* pi = new int[3];  // allocate an int array of 3 elements
```

3.30.2 Unintended constructor conversion (use explicit keyword)

Combination of default argument and C++ conversion rules may lead to unexpected behaviour, specially when constructors can be invoked with a single argument.

```
class FldFieldF
{
public:
  // constructor with  2nd default arg
  FldFieldF(E_Int n1, E_Int n2=0);
  ...
};

class BndSubInj
{
public:
```

```
  BndSubInj(const GeoGrid&         grid,
            const GeoWindowStruct& wind,
            const FldFieldF&       data,
            const EosIdealGas&     eos,
            E_Bool                 useNewtonMethod)
  ...
}

int main()
{
  E_Float data;
  BndSubInj(grid, wind,
            data,  // error
            ...);
}
```

The compiler will first convert `data` from `E_Float` to `E_Int`, and then create a temporary `FldFieldF` object, which was probably not the programmer's intent. To avoid such nasty errors, consider using the `explicit` keyword. In this example, the code does not compile if we add `explicit` to the constructor declaration. A similar situation arises when a class provides many constructors: it may be not obvious to guess which constructor will be selected by the compiler (see 3.19.2 for an example).

3.30.3 Be careful with vector<>::operator[ ]

STL container `vector<>` can grow dynamically at runtime. However this can be achieved through `push_back()` method, `not` with `operator[ ]`:

```
  // Construction vector (size = 2)
  std::vector<int> v(2);
  cerr << v.size()     << endl;
  cerr << v.capacity() << endl;
  v[0] = 0;
  v[1] = 1;
  // Very bad: does NOT increase vector size (memory corruption likely)
  v[2] = 2;
  cerr << " v.size() = " << v.size() << endl;
```

At run time:

```
 v.size() = 2
```

Access to the third vector element (`v[2]`) will probably return garbage value.

Note:

map<key,value>::operator[ ] must be used with care:

```
  #include <map.h>
  #include <iostream.h>
  int main()
  {
    map<int,int> m;
```

```
  int i   = 1;
  int key = 10;
  i = m[key];

  cerr << i << endl;
  cerr << m.size() << endl;
}
At run time:
0
1
```

Empty page

4. FORTRAN SPECIFIC CODING RULES

4.1 Fortran coding style

4.1.1 Every Fortran source file must contain a single subroutine.

However, this rule admits two exceptions:

- for closely related subroutine (algorithmic variants);
- for very small coding units (see for example `FldFortranVecF.for`).

4.1.2 Fortran comments

Use the following syntax:

- use a capital `'C'` in the first column;
- use blank lines (without leading `'C'`);
- use `'!  some comments'` for very short comments.

4.1.3 Code block (DO/ENDDO, IF/THEN/ENDIF)

Code blocks must be indented by 2 blank characters. Example:

```
DO i=1,N
  Some Fortran code
END DO
```

4.1.4 Do not put the optional RETURN statement

4.1.5 Use of (lower/upper)case

- Identifiers (arguments, local variables) are lower-case.
- Fortran keywords (`DO/ENDDO, IF/ENDIF, OPEN, ...`) are capitalized.
- Fortran intrinsics (`MAX, MIN, SQRT, SIGN, ...`) are capitalized.
- Constants are capitalized.

4.2 Avoid using PRINT and WRITE statements

Avoid `PRINT` and `WRITE` statement in Fortran. Log information, if any, should be done in C++.

4.3 Do not use hard-coded value inside Fortran

Avoid using hard-coded ("magic") value): readers of your code will prefer `E_MIN_VOLUME` to `1.e-12`.

When available, use constants defined in `Def/Global/DefFortranConst.h` or `Def/Global/DefFortran-Cutoff.h`:

File Def/Global/DefFortranConst.h:

```
C     -------------------------
C     Integer Fortran constants
C     -------------------------
...
      INTEGER_E TWO_I
      INTEGER_E E_MAXITERNWT  !Max. nb of Newton non-linear iteration
...
      PARAMETER (TWO_I  = 2)
      PARAMETER (E_MAXITERNWT = 30)
...
C     ----------------------
C     Real Fortran constants
C     ----------------------

      REAL_E    ONE_HALF
      REAL_E    TWO
      REAL_E    E_PI
...
#ifdef E_DOUBLEREAL
      PARAMETER (ONE_HALF     =  0.5D0  )
      PARAMETER (TWO          =  2.0D0  )
      PARAMETER (E_PI         = 3.14159265359)
#else
      PARAMETER (ONE_HALF     = 0.5E0  )
      PARAMETER (TWO          = 2.0E0  )
      PARAMETER (E_PI         = 3.14159265359)
#endif
```

This avoids uncertainty when switching between single and double precision.

4.4 Do not call Fortran subroutine (or C function)

Fortran subroutine are used inside elsA to perform intensive computations (see 1). Thus, Fortran subroutines are computational leafs in the program tree-like structure; this means that they should not call other routines (Fortran or C/C++). This rule admits a few exceptions; one of them is the management of errors occuring at run time inside Fortran code:

```
Sio/Access/Fortran/SioOpenFortranF.for:
...
OPEN(impw,FILE=cfich,STATUS='unknown', IOSTAT=koderr)
IF (koderr .NE. 0) CALL erriofortran(er4011)
```

Remarks:

Fortran subroutines must **not** call any MPI routines.

4.5 Do not use COMMON

Instead, information to and from the Fortran `subroutine` is passed by arguments.

Two exceptions: the number of ghost cells, and some global cutoffs:

```
File Def/Global/DefFortranCutoff.h:
      COMMON /GEOM/  E_CUTOFF,
     &               E_CUTOFF_GEOM, E_MIN_SURFACE, E_MIN_VOLUME


File Def/Global/DefFortranGlobal.h:
      COMMON /GHOST/ IFIC1, IFIC2,
     &               JFIC1, JFIC2,
     &               KFIC1, KFIC2,
     &               I2D
```

Note that `DefFortranGlobal.h` should not be included directly; instead, address functions must be obtained from `Geo/Grid/GeoAdrF.h` (see also elsA tutorial).

4.6 Consider using Fortran include files

In some cases, identical sequence of statements apperar in several files, thus leading to maintenance problem through code duplication. Most often, this situation arises inside a `DO` loop, and we cannot substitute a function implementing the comon code sequence, since this will break vectorization. The solution is to introduce a special Fortran include file. See for exmple `EosRadiusIntBorBndF.h`.

Note:

To reduce further Fortran code duplication without impairing CPU efficiency, it is planned to introduce statement function, or preprocessor macro.

4.7 Ordering of arguments

We repeat here the rule given in section 3.29.3. To avoid as much as possible errors when calling Fortran subroutines from C++, follow the following convention for the ordering of arguments passed to the subroutine:

1. `INTEGER` scalars;
2. `REAL` scalars;
3. `INTEGER` arrays;
4. `REAL` arrays;

Inside each category, put `IN` arguments before `OUT` ones.

4.8 Do not use Fortran-90 specific constructs

It is difficult to associate in a portable way Fortran 90 specific features (notably the `MODULE` concept) with C++ code. So avoid using any Fortran 90 specific construct.

4.9 Do not use Fortran 77 implicit type convention

Instead, insert `IMPLICIT NONE` immediately after subroutine statement.

4.10 Include DefFortran.h

Every Fortran file must include the Fortran header: `Def/Global/DefFortran.h`

4.11 Use REAL_E and INTEGER_E

Do not use `REAL`, `REAL*4`, `REAL*8`. Do not use `INTEGER`, `INTEGER*4`, `INTEGER*8`. Instead, use `REAL_E` and `INTEGER_E`. Depending of the Makefile switches (and of the file `DefFortran.h`), `REAL_E` and `INTEGER_E` will be substituted by the required value during the preprocessing stage.

4.12 Avoid using LOGICAL

Presently, there is no correspondence between elsA `E_Bool` and Fortran `LOGICAL`. Consequently, avoid using `LOGICAL`.

Note:
If needed, it would be easy to change this rule.

4.13 Avoid using CHARACTER

Contrary to C/C++, Fortran `CHARACTER` variables (or constants) are not "null-terminated". To avoid potential problems, avoid using them. A noteworthy exception to this rule occurs in the `Sio` module, where the Fortran library is use to perform I/O operation.

4.14 Use adressing statement functions (GeoAdrF.h)

Never compute element address directly: use statement function defined in `GeoAdrF.h`: `adrcell`, `adrnode`, `adrint`, `inccell`, `incnode`. For example:

```
      SUBROUTINE scadis(ncell,eqnb,
     &                  im,jm,km, ...
...
#include "Geo/Grid/GeoAdrF.h"
...
      l1 = inccell( 1, 0, 0, im,jm,km)
      n0 = adrcell( 0, 0, 0, im,jm,km)
```

4.15 Pass two-dimensional arrays as a single argument

When passing "2 dimensional data" (conservative values `w(nbOfCell, nbOfEquation)`, surface normal `surf(nbOfInterface, 3)`) to Fortran, prefer this style:

```
      SUBROUTINE scadis(ncell,eqnb,
     &                  im,jm,km, ...
     &                  w, dif, ...
...
      REAL_E w(0:ncell-1, eqnb)
      REAL_E dif(0:ncell-1,5)
...
      dif(nn,1) = w(np,1) - w(nm,1)
      dif(nn,2) = w(np,2) - w(nm,2)
      dif(nn,3) = w(np,3) - w(nm,3)
      dif(nn,4) = w(np,4) - w(nm,4)
      dif(nn,5) = w(np,5) - w(nm,5)
```

to:

```
      SUBROUTINE scadis(ncell,eqnb,
     &                  im,jm,km, ...
     &                  w1, w2, w3,w4,w5, dif, ...
...
      REAL_E w1(0:ncell-1)
      REAL_E w2(0:ncell-1)
      REAL_E w3(0:ncell-1)
      REAL_E w4(0:ncell-1)
      REAL_E w5(0:ncell-1)
      REAL_E dif(0:ncell-1)
...
      dif1(nn) = w1(np) - w1(nm)
      dif2(nn) = w2(np) - w2(nm)
      dif3(nn) = w3(np) - w3(nm)
      dif4(nn) = w4(np) - w4(nm)
      dif5(nn) = w5(np) - w5(nm)
```

- The code is slightly more general.
- It will be easier to *transpose* (for example for CPU optimization, see subroutine `harcorrs`, in file `FxcHarCorrF.for`).
- A smaller number of arguments reduces the possibility or errors.

4.16 Fortran documentation

Every Fortran subroutine must be preceded by a specially formatted comment section, which can be analysed by doxygen (or Doc++) to extract automatically the documentation. The comment format is very similar to C++ class documentation (see 3.2.3). This comment section is as follows:

```
#ifdef ELSA_DOCUMENTATION
// ============================================================================
// @Name Name_of_subroutine
// @Memo A single line (with no 'dot') of explanation
/* @Text

Design
```

```
> Some more elaborate discussion,
> Additional comments,...

Warning
> May be moved elsewhere in future releases

Reference
Some very important Technical Report, ONERA/DLR/CERFACS cooperation
*/
// ============================================================================
Fortran Name_of_subroutine();
#endif
```

Void sections should be removed (for example, if there is no reference available, remove entirely the keyword reference).

4.17 Declaration section

The declaration section follows immediately the `#include` statements. Variable declarations are classified in three subsections:

- `C_IN` : Argument not modified;
- `C_OUT` : Argument modified;
- `C_LOC` : Internal work variables,

4.18 Do not use Fortran automatic array

elsA code might fail at runtime (probably with a "segmentation violation") due to overflowing the stack. Most often, this behaviour comes from the use of large automatic arrays inside Fortran (for example, code has declarations like `'REAL A(N)'` where `A` is a local array and `N` is a dummy variable that can have a large value).

Most compilers (all ?) currently provide no means to specify that automatic arrays are to be allocated on the heap instead of the stack. So, other than increasing the stack size (typically using `limit stacksize` (in `csh` and derivatives) or `ulimit -s` (in `sh` and derivatives), it is much better to change your source code to avoid large automatic arrays.

4.19 CPU optimization

4.19.1 When possible, write vectorizable loops

Most Fortran loops should be vectorizable. Consider adding compiler directive `FOR_OPT_DIR_NODEP_E`. Be sure that the longest loop is vectorized!

```
FOR_OPT_DIR_NOLOOPCHG_E
      DO eq = 1, eqnb
      DO n = n0int, nfint
C       I interface
           delta(n,eq)           = w(n,ibeg) -  w(n-l1,ibeg)
C       J interface
           delta(n + ncell,eq)   = w(n,ibeg) -  w(n-l2,ibeg)
C       K interface
```

```
        delta(n + 2*ncell,eq) = w(n,ibeg) -  w(n-l3,ibeg)
      END DO
      ideb = ideb + 1
    END DO
```

Without the `FOR_OPT_DIR_NOLOOPCHG_E` directive, the NEC compiler interchanges the loop order.

4.19.2 Consider providing scalar and vector versions

For the most time consuming Fortran routines, it may make sense to write two different versions of the same algorithm:

- the *vector* version is optimized for vector supercomputers;
- the *scalar* version is optimized for non-vector platforms, where efficient use of the complex memory hierarchy (cache) is mandatory to achieve high efficiency.

Use the macro `E_SCALAR_COMPUTER` to decide at compile time which Fortran code to call from C++ (see for example `LhsLuRelaxSca_2.C`).

4.20 Fortran example

An example of Fortran code is given next:

```
C ======================================================================
C Project: elsA - Copyright (c) 2003 by ONERA
C Type   : <3491228467 1384> Fortran subroutine
C File   : Agt/Transfo/AgtTransfoGenF.for
C Vers   : $Revision: 1.1 $
C Chrono : No  DD/MM/YYYY Author   V   Comments
C          1.1 03/09/1999 Charpin  1.0 Vectorization
C          1.0 01/07/1999 XJ/AR    1.0 Creation
C  ====================================================================
#ifdef ELSA_DOCUMENTATION
// ====================================================================
// @Name saxpy
// @Memo same routine as BLAS.
/* @Text

Warning
> May be moved elsewhere in future releases
*/
// ====================================================================
Fortran saxpy();
#endif

      SUBROUTINE saxpy(length, A, x, alpha, y, z )
      IMPLICIT NONE

#include "Def/Global/DefFortran.h"
#include "Def/Global/DefFortranConst.h"

C ======================================================================
```

```
C_BEGIN_DECLARATION

C_IN
      INTEGER_E length
      REAL_E    A(1:length,1:length)
      REAL_E    x(length)
      REAL_E    alpha
      REAL_E    y(length)

C_OUT
      REAL_E    prod(length)
      REAL_E    z(length)

C_LOC
      INTEGER_E i,j

C ========================================================================

#ifdef E_F90
      prod(1:length) = ZERO
#else
      DO j=1,length
        prod(j) = ZERO
      ENDDO
#endif

      DO j=1,length
        DO i=1,length
          prod(i) = prod(i)  +  A(i,j) * x(j)
        END DO
      END DO

      DO i=1,length
        z(i) = alpha*y(i) + prod(i)
      END DO

      END
C =============== Agt/Transo/AgtTransfoGenF.for ===============
```

5. PYTHON SPECIFIC CODING RULES

To be written

Empty page

Direct access to index's alphabetical section headings :

– A – *p.* 72
– B – *p.* 72
– C – *p.* 72
– D – *p.* 72
– E – *p.* 72
– F – *p.* 73
– G – *p.* 73
– H – *p.* 73
– I – *p.* 73
– J – *p.* 73
– K – *p.* 73
– L – *p.* 73
– M – *p.* 73
– N – *p.* 74
– O – *p.* 74
– P – *p.* 74
– Q – *p.* 74
– R – *p.* 74
– S – *p.* 74
– T – *p.* 74
– U – *p.* 74
– V – *p.* 74
– W – *p.* 74
– X – *p.* 74
– Y – *p.* 74
– Z – *p.* 74

INDEX

<iostream>, 49
<list>, 49
<map>, 49
<queue>, 49
<stack>, 49
<string>, 39, 49
<vector>, 49
_E_FORTRAN_LOOPS_, 53
– A – , 72
– B – , 72
– C – , 72
– D – , 72
– E – , 73
– F – , 73
– G – , 73
– H – , 73
– I – , 73
– J – , 73
– K – , 73
– L – , 73
– M – , 73
– N – , 74
– O – , 74
– P – , 74
– Q – , 74
– R – , 74
– S – , 74
– T – , 74
– U – , 74
– V – , 74
– W – , 74
– X – , 74
– Y – , 74
– Z – , 74

– A –

(link is to index's alphabetical headings), 71
accessor, 20, 21
aggregation, 25
assignment operator, 27, 41

– B –

(link is to index's alphabetical headings), 71
Base (root class), 19

– C –

(link is to index's alphabetical headings), 71
C-array, 45
cast, 33, 50
cerr, 38
CHARACTER, 64
class (definition), 19
class (naming convention), 19
class documentation, 19
coding rules, 9
comment style, 12
COMMON (Fortran), 63
composition, 25
const attribute, 27
const data member, 27
const qualifier, 27
const reference argument, 30
const_cast, 24, 36, 50
const_iterator (STL), 16, 49
constructor, 20
constructor (implementation), 40
copy constructor, 27, 41
cout, 38
covariant return type, 45

– D –

(link is to index's alphabetical headings), 71
data member (attribute) naming, 17
default argument, 28
DefCompiler.h, 49
DefCplusPlusConst.h, 37
DefCplusPlusGlobal.h, 37
DefError, 38
DefFortran.h, 64
DefFortranCutoff.h, 62, 63
DefFortranGlobal.h, 63
Deffunction.h , 52
DefIostream.h, 38
delete, 45
delete (array of objects), 44
destructor, 20
destructor (virtual), 44
downcast, 21
dynamic_cast, 50

– E –

(link is to index's alphabetical headings), 71
E_abs, 52
E_BADVALUE_F, 40
E_BADVALUE_I, 40
E_Bool, 35
E_CONST_ITERATOR, 50
E_CUTOFF_GEOM, 38
E_DYNAMIC_CAST, 51
E_False, 35
E_Float, 35
E_FUNC, 52
E_Int, 35
e_log, 38, 51
E_max, 52
E_min, 52
E_MIN_SURFACE, 38
E_MIN_VOLUME, 38
E_MPI, 55
E_NO_COVARIANT_RETURN, 45
E_NULLPTR, 36, 40
E_NULLPTR (null pointer), 32
E_RTTI, 51
E_SCALAR_COMPUTER, 67
E_sign, 52
E_TOLNWT, 38
E_True, 35
E_ZERO_MACHINE, 38
EfEqualZero, 52
ELSADIST, 34
encapsulation, 21
enum, 20
enum (class), 17
exception (C++), 50
explicit (C++ keyword), 57, 58

– F –

(link is to index's alphabetical headings), 71
false, 35
fEqual, 52
file extension, 11
file header, 12
Fld, 45
FldMemory, 46
FldStack, 46
FOR_OPT_DIR_NODEP_E, 66
Fortran, 9
Fortran 90, 63
Fortran automatic array, 66
Fortran ordering of arguments, 63
forward declaration, 34
free (malloc), 45
friend, 20, 24

– G –

(link is to index's alphabetical headings), 71
generic factory, 49
generic programming, 48, 49
get (accessor), 21
getArray(), 22
getBndType, 41
getIdentity, 41
global (C++ constant), 37
global (C++ variable), 37
global (Fortran constants), 62
global (Fortran variable), 62

– H –

(link is to index's alphabetical headings), 71
handle, 24
header (C++), 11
header (private), 24

– I –

(link is to index's alphabetical headings), 71
IMPLICIT NONE, 64
include file (C++), 11
include file(Fortran), 11
include Fortran file, 63
include path, 34
initialize reference attribute, 26
initializer list (constructor), 26, 40
inline, 21, 52
instance() method (singleton), 22
INTEGER_E, 64
interface>file (SWIG), 11
iostream, 49
iostream (C++ I/O), 49
iterator (STL), 16, 49

– J –

(link is to index's alphabetical headings), 71

– K –

(link is to index's alphabetical headings), 71
key prefix, 11

– L –

(link is to index's alphabetical headings), 71
list (STL container), 49
list<>, 49
log message, 38
LOGICAL, 64

– M –

(link is to index's alphabetical headings), 71
macro (preprocessor), 51

malloc, 45
map (STL container), 49
map<>, 49
memory leak, 26, 31, 47
modifier, 21
MPI, 55
mpi.h, 55

– N –

(link is to index's alphabetical headings), 71
namespace, 38
new, 45
null (reference), 31
null pointer, 32

– O –

(link is to index's alphabetical headings), 71
overloaded method, 46
overloaded virtual methods, 44
overloading virtual method, 43

– P –

(link is to index's alphabetical headings), 71
pass-by-value (primitive type), 29
passing arguments, 27
PcmDefMpi.h, 55
polymorphism, 42
porting issue, 50
prepare method, 26
printf, 38
private, 20, 21
private header, 11, 24
protected, 20
protected attribute, 21
public, 20
Python, 9

– Q –

(link is to index's alphabetical headings), 71
queue (STL container), 49
queue<>, 49

– R –

(link is to index's alphabetical headings), 71
REAL_E, 64
reference-to-const argument, 30

– S –

(link is to index's alphabetical headings), 71
scalar computing platforms, 67
set (modifier), 21
single statement block, 15
singleton, 49
singleton (design pattern), 22
smart pointer, 49
sprintf, 40
stack (STL container), 49
stack<>, 49
static_cast, 50
STL, 49
STL container, 16, 49
STL containers, 49
STL iterator, 16, 49
string (class), 49
stringstream, 40
SUBROUTINE documentation, 65
SWIG, 35

– T –

(link is to index's alphabetical headings), 71
TbxString, 39
template, 49
template (C++), 48
template design pattern, 42
true, 35
typedef, 20
typedef (insulation), 24

– U –

(link is to index's alphabetical headings), 71

– V –

(link is to index's alphabetical headings), 71
vector (STL container), 49
vector supercomputers, 67
vector<>, 49
vectorization, 66
vectorization (C++), 53
vectorization (Fortran), 63
virtual destructor, 44
virtual method, 41

– W –

(link is to index's alphabetical headings), 71

– X –

(link is to index's alphabetical headings), 71

– Y –

(link is to index's alphabetical headings), 71

– Z –

(link is to index's alphabetical headings), 71

DIFFUSION SCHEMA

Software Secretariat Archives

Writers

elsA developers

END of LIST