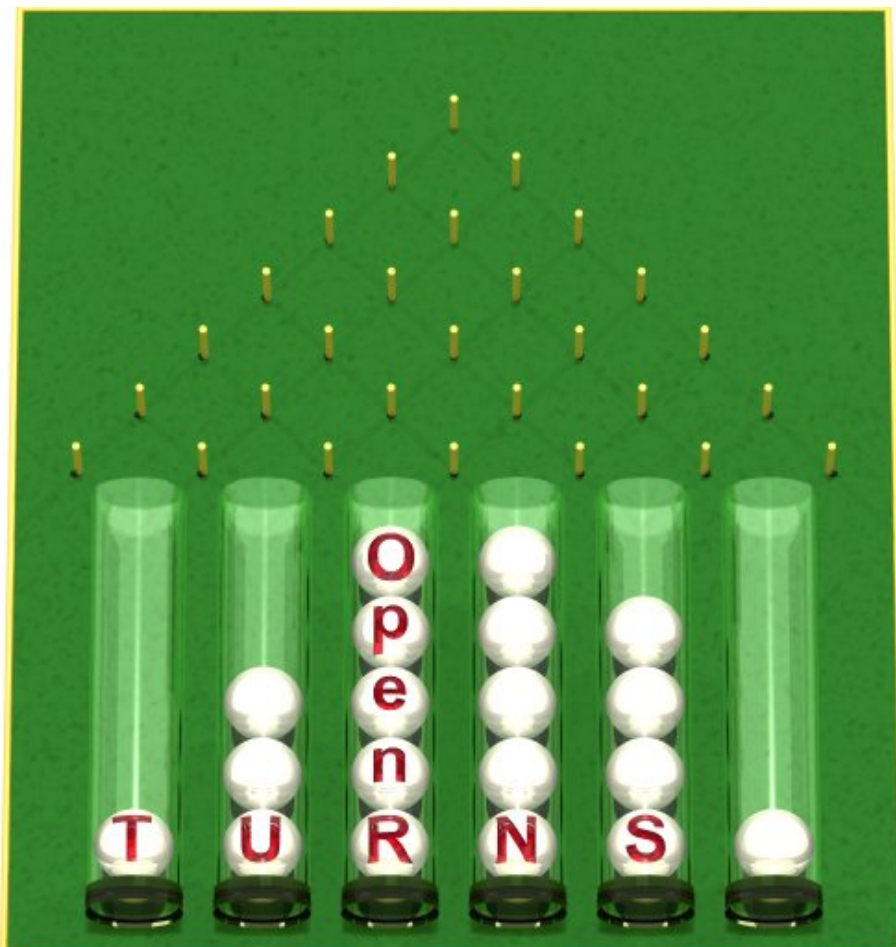


# Coding rules

Open TURNS version 1.0

Documentation built from package openturns-doc-April2012

April 20, 2012



## OPEN TURNS PROJECT: CODING RULES

*Abstract*

The purpose of this document is to present the programming rules to be applied in the Open TURNS project for the coding of classes and modules in the C++ and Python languages.

The rules are not intended to restrain the expression of developers. Following the rules allows the development of C++ and Python source code that can be understood by any developer and any newcomer to the project. Moreover, the rules help avoiding pitfalls that are common and classical in C++ and Python, thus ensuring a source code of high quality, i.e. readable, efficient and reliable. Furthermore, following the programming rules is a sign of respect towards the other developers and the future maintainers of the Open TURNS code.

The document addresses the modular structure of the Open TURNS code, the file names and the two main languages used: C++ and Python. Whenever possible, examples and counterexamples are provided to illustrate the application of each rule.

The basic rules regarding how to structure the files and directories, and how to carry out unit tests in the development environment, are not described in this document. These processes, as well as the processes for revision control and managing the evolution of the source code and modules, are described in the *Software Configuration Management Plan* of the Open TURNS project and rely on Subversion.

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Document goals . . . . .	3
1.2	Conventions . . . . .	3
<b>2</b>	<b>Package structure</b>	<b>5</b>
2.1	Packages . . . . .	5
<b>3</b>	<b>Source file naming</b>	<b>6</b>
3.1	C++ Files . . . . .	7
3.2	Header files . . . . .	7
3.3	Test files . . . . .	9
<b>4</b>	<b>C++ language</b>	<b>10</b>
4.1	C++ language standard . . . . .	10
4.2	Namespaces . . . . .	10
4.3	Names . . . . .	11
4.4	Class declaration . . . . .	14
4.5	Inheritance . . . . .	18
4.6	Function and method declaration . . . . .	19
4.7	Variable declaration . . . . .	20
4.8	Constant declaration . . . . .	20
4.9	Comments and internal documentation . . . . .	20
4.10	Memory allocation and deallocation . . . . .	21
4.11	Assignment and initialization . . . . .	23
4.12	Instructions . . . . .	23
4.13	Exceptions . . . . .	27
4.14	Error handling and error messages . . . . .	28
<b>5</b>	<b>Python language</b>	<b>29</b>
5.1	Modules and packages . . . . .	29
5.2	Names . . . . .	29
5.3	Comments and internal documentation . . . . .	29

# 1 Introduction

We would like to thank Vincent Lefebvre and Olivier Morvant from the Descartes project, who are the original authors of this document that we have almost entirely drawn upon.

## 1.1 Document goals

The coding rules edicted in this document mainly regard:

- the structure of modules and classes;
- the names of the source files;
- the use of the C++ and Python programming languages;
- the documentation included inside the source files;
- the format used to write statements in the programming languages.

The purpose of these rules is to provide a common formalism for the project in order to help the developers produce code that is globally homogeneous, readable, understandable by all and also efficient and reliable, respecting the architectural choices and constraints established to produce the code of the Open TURNS project. The scope of this document only embraces files produced in the Open TURNS project and does not apply to external software included in the project *as is*.

## 1.2 Conventions

This paragraph establishes the conventions used in this document. Each coding rules is identified by a unique number prefixed by the letter R.

**Rxx** <Description of rule>

Examples of source code in C++ and Python are framed. Correct examples are given in blue, while incorrect examples are in red.

```
<Example>
```

```
...
```

```
<correct example>
```

```
...
```

```
<incorrect example>
```

```
...
```

When an example requires comments, these are given above the said example and they appear in italics.

*Comment about example*

```
<Example>
```

Elements from the text, rules or source code cited as an example and appearing between the two characters < and > correspond to a description of the expected items or objects. The following example shows that <**condition**> represents a boolean expression and the function **compute** is called on a piece of data of type length, with its default unit.

```
if ( <condition> ) {  
  r = compute( <length in cm> )  
}
```

By extension, the following symbols are used to describe a grammar, a call syntax, etc.

<real> a real number

<integer> an integer

<string> a string

<filename> a string representing a filename

The first rule is given below this introductory paragraph and deals with the language used in the code.

**R1 Coding language:** the language used to program classes, functions and modules in Open TURNS is English. The comments included in the source code shall be written in English as well.

## 2 Package structure

In order to structure the code of the Open TURNS project, the various elements (classes, functions, libraries, data) are logically organized in packages. This chapter describes the rules to be followed for the definition, management and use of these packages.

### 2.1 Packages

The Open TURNS code is mainly located in a single library. This library is organized as a set of modules. However, there may be several interacting libraries in the future.

The Open TURNS library is interfaced with Python through a Python module exposing almost all the Open TURNS classes and operators.

**R2** Libraries at the same level are independent from each other. The source files of a function or a class can not be shared among several modules. If two libraries share the same functions or classes, these must be grouped in a common "utility" library, in order to avoid cyclic references and dependencies.

**R3** The C++ modules required for the Open TURNS code are placed in static and dynamic libraries. All libraries related to Open TURNS are prefixed by **libOT**.

For the moment, the entire set of classes is located in **libOT.so** for the dynamic part and in **libOT.a** for the static part.

**R4** Python modules required for the Open TURNS code are grouped in a package that can be, if necessary, broken down into "sub-packages".

*Example showing the import of modules via the **openturns** Python package*

```
import openturns
import openturns.base
import openturns.uncertainty
```

*Example showing the direct import of module operators or classes via the **openturns** Python package*

```
from openturns import NumericalPoint
from openturns.base import NumericalSample
from openturns.uncertainty import RandomVector
```

### 3 Source file naming

The definition of filenames obeys a few rules described below, according to the programming languages used. A general rule is preliminarily defined in order to facilitate the automatic generation of the **Makefile** files. The file names consist of sequences of characters separated by a dot. The first part of the name is called the *base* and the second is called the *suffix* (or *extension*).

**R5** The suffixes of the filenames are used to identify the nature of these files in terms of programming language. The list of suffixes used in the project is given in the table below.

Extension	Description
<b>.hxx .hh .hpp</b>	Header file containing the declaration of functions and classes and the definition of templates for C++
<b>.cxx .cc .cpp</b>	Source code file containing the definition (implementation) of C++ functions and classes
<b>.c</b>	Source code file containing the definition of C functions
<b>.h</b>	Header file containing the declaration of functions in C
<b>.f</b>	Source code file containing the definition of FORTRAN 77 functions
<b>.py</b>	Python file
<b>.R</b>	R file
<b>.a</b>	Archive file containing statically linked objects
<b>.ac</b>	Autoconf file
<b>.am</b>	Automake file
<b>.at</b>	Autotest file (for compiled tests)
<b>.atpy</b>	Autotest file (for Python tests)
<b>.bat</b>	DOS script
<b>.conf</b>	Open TURNS configuration file
<b>.csv</b>	Comma Separated Value file (for samples)
<b>.dox</b>	Doxygen file
<b>.dtd</b>	XML Document Type Definition file
<b>.i</b>	SWIG interface file
<b>.in</b>	Autoconf template file
<b>.la</b>	Libtool archive (for both static and dynamic libraries)
<b>.ll</b>	Lex scanner file
<b>.log</b>	Output log file
<b>.m4</b>	M4 script (mainly to define Autoconf macro)
<b>.mws</b>	Maple script file
<b>.nsi</b>	Windows installer file
<b>.pth</b>	Python path file for additional modules
<b>.sce</b>	Scilab script file
<b>.sh</b>	Shell script
<b>.so</b>	Shared object file containing dynamically linked objects
<b>.txt</b>	Text file
<b>.xml</b>	XML file (mainly for wrapper description file)
<b>.yy</b>	Yacc parser file

**R6** To assign a name to a file, it must be assumed that the system does not make any distinction between uppercase and lowercase letters, so that two files in the same directory cannot bear the same name.

*For example, it is not recommended to give the following names to two files in the same directory:*

```
matrix.cxx
Matrix.cxx
```

### 3.1 C++ Files

**R7** When a **.hxx** file is used to define a class, it must adopt the name of this class. As a consequence, a **.hxx** file should not be used to define more than one class unless they are tightly linked. The declaration of nested classes is allowed but it must be noted that it causes difficulties in writing SWIG interface files.

*Example: one file per class:*

```
Sample.hxx declares class Sample
Matrix.hxx declares class Matrix
...
```

*Incorrect example: one file for all classes of a model:*

```
Model.hxx # contains all the declaration of all the classes of the internal model
```

The preceding rule has one exception: in order to facilitate the use of several related classes, the header files belonging to the same module are grouped in a single header file, which bears the same name as the module and is prefixed by **OT**.

*Example: using all the classes of the Base module:*

```
#include "OTBase.hxx"
```

### 3.2 Header files

The header files are used to declare functions and classes (they are sometimes called *interface definition* or *interface specification*).

**R8** All header files must have an unique name.

**R9** In order to avoid multiple inclusions, the content of a **.hxx** file must be enclosed between the instructions shown below. There are two possible forms: either with the **#ifndef** compiler directive, or with the combined use of the **#if** directive and the logical expression **!defined(...)**.

*Example for a file named **Sample.hxx***

```
#ifndef OPENTURNS_SAMPLE_HXX
#define OPENTURNS_SAMPLE_HXX
...
#endif /* OPENTURNS_SAMPLE_HXX */
```

*Example for a file named **Sample.hxx** - with an explicit condition*

```
#if !defined(OPENTURNS_SAMPLE_HXX)
#define OPENTURNS_SAMPLE_HXX
...
#endif /* OPENTURNS_SAMPLE_HXX */
```

**R10** The identifier following the `#ifndef` statement must use the full name of the header file (base and suffix) prefixed by `OPENTURNS_` and obtained by replacing all non-alphanumeric characters with underscores and by using uppercase letters for all the alphanumeric characters.

*Example for a file named **Sample.hxx***

```
#ifndef OPENTURNS_SAMPLE_HXX
#define OPENTURNS_SAMPLE_HXX
...
#endif /* OPENTURNS_SAMPLE_HXX */
```

**R11** The inclusion directive for the Open TURNS header files follows the form given below:

*Example of Open TURNS header file inclusion*

```
#include "OSS.hxx"
#include "NumericalPoint.hxx"
```

**R12** The inclusion directive for files related to standard or external libraries must follow this form:

*Example for the inclusion of system function or external library header files*

```
#include <cstring>
#include <sys/stat.h>
#include <boost/python.hpp>
```

**R13** The inclusion of C system function libraries in C++ must use the C++ standard header files:

C file	C++ file	Description
<code>assert.h</code>	<code>cassert</code>	Definition of the <code>assert</code> macro
<code>ctype.h</code>	<code>cctype</code>	Macros for character handling
<code>errno.h</code>	<code>cerrno</code>	System error codes and messages
<code>float.h</code>	<code>cfloat</code>	Constants for floating point real numbers
<code>iso646.h</code>	<code>ciso646</code>	
<code>limits.h</code>	<code>climits</code>	Definition of platform-specific constants and limits
<code>locale.h</code>	<code>clocale</code>	Declaration of functions and structures for localization
<code>math.h</code>	<code>cmath</code>	Declaration of mathematical functions and definition of mathematical constants
<code>signal.h</code>	<code>csignal</code>	Declaration of the functions handling process signals
<code>stdarg.h</code>	<code>cstdarg</code>	Management of lists of variable arguments
<code>stddef.h</code>	<code>cstddef</code>	Standard definitions ( <code>NULL</code> , <code>size_t</code> , <code>wchar_t</code> , <code>offsetof</code> )
<code>stdio.h</code>	<code>cstdio</code>	Declaration of C standard I/O functions
<code>stdlib.h</code>	<code>cstdlib</code>	Declaration of the C standard library functions ( <code>libc</code> )
<code>string.h</code>	<code>cstring</code>	Declaration of C functions handling strings; NB: not to be confused with the <code>string</code> file declaring the <code>std::string</code> class
<code>time.h</code>	<code>ctime</code>	Declaration of functions converting system date and time into strings
<code>wchar.h</code>	<code>cwchar</code>	Declaration of functions handling wide char strings; NB: the C++ support of wide chars must be declared with the macro for the handling of wide chars
<code>wctype.h</code>	<code>cwctype</code>	

**R14** If the inclusion of C functions in C++ cannot rely on a header file as described in rule R12, the following form must be used:

*Example for the inclusion of non standard system function header files*

```
extern "C" (  
#include <nonstandard.h>  
)
```

**R15** The Open TURNS header files which are to be ultimately installed on the system should include all of their files using < and >.

### 3.3 Test files

**R16** Each class must be able to be tested independently from the others through a set of unit tests. It is prohibited to provide a class that would not offer at least one unit test. It is advisable to produce a unit test for each class functionality in order to isolate possible bugs. It is also advisable to provide integration tests for the class itself and those on which it depends, in order to validate the cross-integration of elementary classes. It is possible to create tests for bugs that have been detected. The test files should be prefixed by `t_`, followed by the name of the class as described in rule R3, followed by an underscore, followed by an identifier describing the nature of the test.

*Example of names for test files*

```
t_Matrix_construction.cxx  
t_Matrix_assignment.cxx  
t_Matrix_bug7654.cxx  
t_Matrix_vectorMultiplication.cxx
```

## 4 C++ language

Coding rules for the C++ language in Open TURNS.

### 4.1 C++ language standard

**R17** The use of a programming language for which there is an ISO/ANSI standard involves compliance with this standard. The source files are compiled, whenever possible, with the options carrying out the ISO/ANSI standard compliance check.

*Example: GCC compilation:*

```
g++ -c -ansi ...
```

*Note: Compliance with the ISO/ANSI standard sometimes introduces significant limitations on the use of basic definitions or basic system functions. Moreover, strict checks on the compliance with the ANSI standard are switched on through options that are compiler-specific.*

### 4.2 Namespaces

**R18** The Open TURNS C++ types, classes and functions will be declared in an **OpenTURNS** namespace, which will be aliased to **OT**.

*Example of OpenTURNS namespace definition for simple types:*

```

/ **
* @file      OTtypes.hxx
* ...
* /
namespace OT
{
  /* < Declarations of simple types > * /

  /* < Declarations of objects and functions > * /
} ;

// Alias for the direct use of XXX
namespace OpenTURNS = OT;
```

*Example of a class declaration in the OpenTURNS namespace and in a nested namespace*

**Uncertainty::Model:**

```

/**
* @file      RandomVector.hxx
* ...
*/
namespace OT {
  class RandomVector {
    /* ... */
  }; /* end class RandomVector */
} /* namespace OT */
```

*Example of use by making all the definitions contained in the namespace available:*

```
#include "OT.hxx"
using namespace OT ;

void f( NumericalScalar n ) ;
```

**R19** Employing the **using** directive is prohibited in header files.

*Example of namespace use:*

```
#include "OT.hxx"

void f( UnsignedLong n );
```

### 4.3 Names

**R20** Simple (or built-in) types are recoded, using the **typedef** instruction, in the `<OTtypes.hxx>` file included in all C++ source files.

*Example:*

```
#ifndef OPENTURNS_OTTYPES_HXX
#define OPENTURNS_OTTYPES_HXX

#include <string>
/**
 * basic types mapping
 */
namespace OT
{
    typedef bool          Bool ;

    typedef unsigned long UnsignedLong ;

    typedef double       NumericalScalar ;

    typedef std::string  String ;

    /* ... */
} ;

#endif /* OPENTURNS_OTTYPES_HXX */
```

**R21** When a type name (simple, structure, class) is defined by the programmer, each word composing the name must begin with an uppercase letter.

*Example:*

```
class NumericalSample {
    ...
} ; /* end class NumericalSample */
```

**R22** The names of variables outside of classes must begin with a lowercase letter.

*Example:*

```
int main( ) {
    Bool myCondition ;
    ...
}
```

**R23** The names of all class members must end with an underscore.

*Example:*

```
class Environment : public Object {
    ...
    private :
    NumericalScalar density_ ; //<! material density in environment (g/cm3)
    ...
} ; /* end class Environment */
```

NB: It is common for the underscore to be used as a prefix for private attribute names. However, this method may trigger conflicts with internal variables or definitions used by the compilers. For this reason, the underscore is used as a suffix.

**R24** The first character of the names of class member data (**static** type attributes) must be an uppercase letter. For any other attribute, it will be a lowercase letter.

*Example:*

```
class Object {
    ...
    private :
    static String ClassName_ ;
    ...
} ; /* end class Object */
```

**R25** The name of a **static** class method or a **static** function must begin with an uppercase letter. The names of other methods and functions begin with a lowercase letter.

*Example:*

```
class Object
{
    public:
    /* returns a class identifier based on its name
    static String GetClassName() ; ...
} ; /* end class Object */
```

**R26** The name of a **static** variable must begin with an uppercase letter.

*Example:*

```
int
initializeConversion( )
{
    static Bool IsInitialized = false ;
    if ( ! IsInitialized ) {
        ...
        IsInitialized = true ;
    }
} ;
```

**R27** The name of a constant must consist of a series of *uppercase* words separated by underscores.

*Example:*

```
const UnsignedLong MAXIMUM_OF_RETRIES = 5;
```

**R28** The names of types and variables must consist of a sequence of *whole* words. From the second word on, they begin with an uppercase letter.

*Example:*

```
int main()
{
    NumericalScalar reactionRate = 0. ;
    ...
}
```

**R29** The names of functions and methods must begin with an action verb in the infinitive form. This verb begins with a lowercase letter. From the second word on, the words begin with an uppercase letter.

*Example:*

```
class NumericalSample
{
    UnsignedLong getDimension ( ) const ;
    ...
} ; /* end class NumericalSample */
```

**R30** The names of variables, members, functions, methods and classes must not contain any abbreviation, except when they are self-explanatory and can not be confusing.

*Example:*

```

class NumericalSample {
} ; /* end class NumericalSample */

void removeElement(const UnsignedLong index) ;

NumericalPoint meanValue ;

```

*Example of tolerated notations:*

```

UnsignedLong i ; // loop indices i, j and k are common
UnsignedLong j ;
UnsignedLong k ;

UnsignedLong nbMaxElements ; // usual abbreviations: nb, Max

void
addPoint( NumericalPoint pt ) { // no confusion in the context
    add( pt ) ;
}

```

*Incorrect examples:*

```

NumericalScalar a, k, l, m1, m2, m3 ;
NumericalScalar zzz, zz2 ;
const char *foo, *hello, tempo, bogus ;

void adElt( NumericalPoint pt );

UnsignedLong numSmplPt ;

```

#### 4.4 Class declaration

**R31** The constituents of a class should be declared in the following order: **public**, **protected**, **private**. All static methods and members are placed before anything else.

*Example:*

```

class Buffer {
    public :
    static AThing GetThing();
    protected:
    private :
    static AThing TheThing_ ;

    public :
    NumericalScalar getValue() const;
    protected :
    NumericalScalar theValue_ ;
    private :

```

```

    /* ... */
} ; /* end class Buffer */

```

**R32** The basic structure for a class declaration is as follows: declaration of a default constructor, of a copy constructor, of a virtual destructor, of a copy assignment operator and if necessary of a comparison operator ('==') and of a stream converter, followed by the other methods of the class.

*Example:*

```

class AnyClass {
public :
    /** Default constructor */
    AnyClass( ) ;
    /** Copy constructor */
    AnyClass( const AnyClass & other ) ;
    /** Destructor */
    virtual ~AnyClass( ) ;
    /** Copy operator */
    AnyClass& operator = ( const AnyClass & other ) ;
    /** Comparison operator */
    Bool operator == ( const AnyClass & other ) const ;
    /** Stream converter */
    String repr( ) const ;
    String str( ) const ;

    /* other public methods ... */

private :
    UnsignedLong size_ ;
    DataType * data_ ;

    /* other private methods ... */
} ; /* end class AnyClass */

```

**R33** All attributes of a class should be placed in the **protected** or **private** sections. For classes designed to be derived, attributes can be promoted to the **protected** section.

*Example:*

```

class AnyClass {
public :
    /* ... */
private :
    UnsignedLong size_ ;
    DataType * data_ ;
} ; /* end class AnyClass */

```

**R34** All attributes must be initialized in the constructors, using their respective constructors and respecting the order of attribute declaration in the class.

*Example:*

```
class Vector {
  public :
  Vector ( Bool someProperty , UnsignedLong size , NumericalScalar elt = 0. ) ;
  private :
  Bool property_ ;
  Collection<NumericalScalar> data_ ;
} ;
```

*Example of a correct definition:*

```
Vector::Vector ( Bool someProperty , UnsignedLong size , NumericalScalar elt )
: property_(someProperty), data_( size , elt )
{ }
```

*Examples of incorrect definitions:*

```
Vector::Vector ( Bool someProperty , UnsignedLong size , NumericalScalar elt )
: data_(size , elt), property_(someProperty) // order of initialization
{ }
```

```
Vector::Vector ( Bool someProperty , UnsignedLong size , NumericalScalar elt )
{
  property_ = someProperty;
  data_ = Collection<NumericalScalar>(size , elt);
  // requires an assignment after the construction
  // processing is longer for complex objects!
}
```

**R35** The constructors of a class must not call **virtual** type methods of the class.

*Example: declaration of a pure virtual class A and of class B, derived from A:*

```
class A {
  public :
  A( ) ; // constructor
  virtual ~A( ) ; // destructor
  virtual const char * getClassName( ) = 0 ; // pure virtual method
} ;

class B : public A {
  public :
  const char * getClassName( ) { return "B" ; }
} ;
```

*Incorrect definitions leading to an execution error:*

```

A::A( ) {
    cout << getClassname( ) << "_created" << endl ; // B does not exist yet!
}

A::~~A( ) {
    cout << getClassname( ) << "_destroyed" << endl ; // B no longer exists!
}

B::B( ) : A( )
{ }

```

**R36** If an attribute is readable and/or writable by the outside world and/or derived classes, the access methods given below are reported under the **public** or **protected** sections. For access methods:

- Simple types or "built-in types" (**bool**, **int**, **float**, **long**, **double**, ...) are passed or returned as values;
- Composite types (class, structure) are passed as constant references. The return type depends on the implementation of the class. No reference to temporaries nor non-const reference to members should be returned.

Write method for the **name** attribute:

```

void          setName ( SimpleType ) ;
void setName    ( const ComposedType & ) ;

```

Read method for the **name** attribute:

```

SimpleType          getName( ) const ;
const ComposedType & getName( ) const ;

```

Example:

```

class NumericalSample {
public :
    /* return the dimension of the sample
    UnsignedLong getDimension( ) const ;

    /* return the i-th element
    NumericalPoint          operator [] (UnsignedLong i);
    const NumericalPoint & operator [] (UnsignedLong i) const ;
} ;

```

**R37** The bodies of **inline** type methods may be located outside of the class declaration, in the header file, in order to make the class easier to read.

Example:

```

class NumericalSample {
public :
    /* return the number of the rod
    inline UnsignedLong getDimension( ) const { return dimension_ ; }

    /* compute the mean point of the sample
    inline NumericalPoint computeMeanValue() const ;
};

/* inline methods
NumericalPoint computeMeanValue() const ;
{
    /* ... some non trivial processing */
    return meanValue ;
}

```

## 4.5 Inheritance

**R38** Multiple inheritance should be used *only when necessary* and its use *must be justified*.

**R39** The constructor of the parent class must be called in the constructor of the derived class prior to any member initialization.

*Example: the Point class derives from the Vector class*

```

class NumericalPoint : public std::vector<double> {
public:
    Point ( NumericalScalar x, NumericalScalar y,
            NumericalScalar z );
} ;

Point::Point( NumericalScalar x, NumericalScalar y,
             NumericalScalar z )
: std::vector<double>( 3 )
{
    (*this)[0] = x ;
    (*this)[1] = y ;
    (*this)[2] = z ;
}

```

**R40** Except for classes designed not to be derived, the destructor of a class must systematically be declared as **virtual**, in order to ensure that the destructors of potential derived classes are actually called.

*Example:*

```

class Object {
public :
    Object( ) ;
}

```

```

    virtual ~Object( );
};

```

## 4.6 Function and method declaration

**R41** Each function that is not local to a compilation unit must be subjected to a prototype declaration in an included file.

**R42** The prototype of a function or method must be as follows:

```

/** @brief <short description>
 *
 * <Long description>
 * @param argument_1 <description>
 * @param argument_2 <description>
 * @return <description>
 * @throw <description>
 */
ReturnType
functionName (
    TypeArgument_1    argument_1,
    TypeArgument_2    argument_2
);

```

**R43** For structured (i.e. different from simple types) types, parameter passing should occur by *reference* rather than by value. Structured types that are not modified by a method or a function must be declared as **const**.

*Correct example:*

```
void send( const String & message ) ;
```

*Incorrect Example:*

```
void send( String message ) ;
```

**R44** Methods that do not modify the class instance should be declared as **const**.

**R45** For a function or a method, overloading must be used preferably to a variable number of arguments or optional arguments.

*Correct example:*

```

Buffer & append( UnsignedLong );
Buffer & append( const String & );
Buffer & append( NumericalScalar ) ;

```

*Incorrect Example:*

```
Buffer & append( const char *fmt, ... ) ;
Buffer & append( const char *str = 0, double d = 0., int i = 0 ) ;
```

**R46** *Inline* or *template* functions should be used rather than macro-functions.

## 4.7 Variable declaration

**R47** Do not declare more than one variable per line. *Always remember to initialize variables and to add a comment for each.* The declaration of *multiple* variables on one line is allowed if *all variables are of the same type.*

*Correct example:*

```
String          filename ("" ) ; // library file name
UnsignedLong    nbElements(0) ; // number of elements into the data file
UnsignedLong    i = 0;
UnsignedLong    j = 0;
```

*Accepted example:*

```
UnsignedLong    i = 0, j = 0, k = 0; // indices
```

*Incorrect Example:*

```
/* Multiple declarations and different types */
UnsignedLong    i, j, tab[20], *l, *numberOfElements ;
String          filename , *yourname, myname ;
```

## 4.8 Constant declaration

**R48** The use of **const** variables must be preferred over **#define**.

*Example:*

```
const UnsignedLong maximumIterations = 32;
const char printFormat [] = "%s:line %d, %s" ;
```

*Incorrect Example:*

```
#define MAXIMUM_ITERATIONS 32;
#define PRINT_FORMAT      "%s:line %d, %s"
```

## 4.9 Comments and internal documentation

**R49** An introductory comment must appear at the beginning of each file. In the case of a class declaration file, this comment will follow the structure below:

```

/**
 * @file      file name and version
 * @brief     short description
 *
 * <LGPL copyright>
 *
 * @project   OpenTURNS
 *
 * @author   first name LAST NAME (login)
 * @date     Wed Nov 24 15:32:07 MET 1997
 *
 * Copyright (C) EDF-EADS-Phimeca 2005-YYYY
 */

```

**R50** There must be a comment for each parameter of a function or method. See rule R42.

**R51** Explanatory comments precede, if necessary, a group of instructions. These comments should explain "why" these instructions are there, not "how" they were implemented, except if a very specific optimization or implementation requires further explanation.

These comments should avoid:

- mentioning the names of variables;
- being a simple transcription of the code into English.

#### 4.10 Memory allocation and deallocation

This section discusses general rules for allocating and freeing memory. It will later be supplemented by rules regarding the use of basic classes in order to manage the lifecycle of objects in memory.

**R52** The definition of automatic variables must be favored over dynamic allocation. It is preferable to use the STL containers (**list**, **vector**, **map**, **queue**, etc.).

*Example to favor:*

```

{
  NumericalScalar sections1[MAX] ; // a fixed size array
  vector<NumericalScalar> sections2 ; // an extensible vector
  list<Volume> volumes ; // a list of volumes

  /* ... */
}

```

*Example to avoid:*

```

{
  NumericalScalar *sections = new NumericalScalar[MAX] ;
  list<Volume> *volumes = new list<Volume> ;
}

```

```

    /* ... */
    delete [ ] sections ;
    delete volumes ;
}

```

**R53** In the C++ programs, the operators **new** and **delete** must be used instead of the **malloc** and **free** functions when initializing memory.

*Correct example:*

```

{
    Volume *volume = new Volume ;    // memory allocation +
    // constructor call
    /* ... */
    delete volume ;                  // destructor call +
    volume = 0 ;                      // memory deallocation
}

```

*Incorrect example:*

```

{
    Volume *volume = (Volume*) malloc(sizeof(Volume)) ;
    // memory allocation but
    // no constructor call
    /* ... */
    free( volume ) ;                 // no destructor call before
    volume = 0 ;                      // memory deallocation
}

```

**R54** In order to free the entire memory space allocated to an object vector, the string '[]' must not be forgotten after the **delete** operator, in order to call the destructor on all of the objects placed in the table. For restrictions to this rule, see rule R55.

*Example:*

```

A *    a = new A[40] ; // calls the constructor 40 times
...

```

```

delete a ; // incorrect: the table is freed,
// the ~A destructor isn't called

```

```

delete [] a ; // correct: the table is freed,
// the ~A destructor is called 40 times

```

**R55** To avoid having to manually manage the memory allocation and deallocation, the use of "smart pointers" is mandatory. It is *prohibited* to directly use C pointers. The OpenTURNS **Pointer**<**T**> class implements a smart pointer for class **T**. For restrictions to this rule, see rule R56.

List of declaration files for the smart pointer:

```
#include "Pointer.hxx"
```

**R56** Though it is forbidden to directly use C pointers (which should thus never appear in the interface of an object), there are a few tolerated exceptions. Among these, methods to convert objects to C structures are allowed. However, specific attention must be paid to avoiding memory leaks.

#### 4.11 Assignment and initialization

**R57** Initializations should be preferred over assignments.

Example:

```
String message( "finished" ) ;

String message = "done" ;
```

Example to avoid:

```
String message ;
message = "Computation_complete" ; // assignment after construction

String message() ; // declaration of a function prorotype
```

**R58** Do not make assumptions about the execution order of a sequence of initializations between different compilation units.

#### 4.12 Instructions

**R59** The indentation follows Emacs' indentation mode.

**R60** Whenever efficiency constraints allow it, the production of *clear code* must be preferred over optimized code. When it is not possible, comment on the optimization and the resulting constraints.

**R61** Limit to a reasonable number (*maximum 3*) the number of nested control structures. In the case of complex algorithms, one must use specific functions, possibly templates, to limit the depth of nesting.

**R62** For readability, one must strive to write only one statement per line. **For** loops are not affected by this rule.

Example:

```
i = 0 ;
while ( i < MAX ) {
    ++i ;
    f(i) ;
}
```

Examples to avoid if possible:

```

a = b = c = 0 ;
// multiple assignments

f(++i) ;
// readability

v = *i++ ;
// performance and understandability

for( i = 1, j = 2, k = 3; i < N; j++, i++ );
// understandability and readability

```

*Incorrect examples:*

```

buffer += "test", cout << buffer ; i = 1 ;
// heterogeneous processing &
// different objects

while( f(++i), i < MAX);
// processing carried out before the test

```

**R63** The use of the **goto** instruction must be prohibited, even for error handling.

*Prohibited example:*

```

void foo( ) {
  for( ... ) {
    phase1 :
    /* ... */
    phase2 :
    if( errno != 0 )
      goto erreur ;
    if ( /* a test */ )
      goto phase2 ;
  }
  erreur :
  /* error handling */
}

```

*Note: error handling can be easily replaced with an exception handling, and the use of **goto** for the needs of algorithms can always be replaced with a conditional structure or a loop.*

**R64** Minimize the number of temporary objects in a block of instructions. Declarations should occur at the latest moment in order to avoid useless object creation.

*Example:*

```

NumericalScalar
compute( UnsignedLong n ) {

```

```

NumericalScalar result ;
if( n < MIN || n > MAX ) {
    char msg[BUFSIZ];
    // automatic allocation for the processing
    sprintf ( msg,
        "n=%d is out of range, valid range is [%d,%d] ",
        n, MIN, MAX );
    throw Exception( msg ) ;
}
/* ... */
return result;
}

```

*Examples to avoid:*

```

NumericalScalar
compute( UnsignedLong n ) {
    NumericalScalar result ;
    Char    msg[BUFSIF] ; // allocation unnecessary if no
    // error
    if( n < MIN || n > MAX )
        ...
}

```

**R65** The **switch** keyword should be avoided. Prefer design, virtualization and polymorphism. When **switch** is used, the **default** case must always be present. In a **switch**, the instructions for the different **case** items must always end with a **break**. The cases will always use constants defined in advance.

*Correct example:*

```

switch ( errno ) {
    case ENOENT :
        msg = "..." ;

        break ;
    case EACCESS :
        msg = "..." ;
        break ;
    default :
        msg = "unknown_error" ;
        break ;
}

```

*Accepted example - processing multiple targets with the same block:*

```

switch ( errno ) {
    case ENOENT :
    case EACCESS :
        msg = "impossible_to_access_file" ;
        break ;
}

```

```

default :
/* ... */
break ;
}

```

*Incorrect example:*

```

switch ( errno ) {
  case 1 : // it is a value
    msg = "..." ; //
    // "break" is missing,
    // processing continues in ENOENT
  case ENOENT :
    msg = "..." ;
    break ;
  default : // "break" is missing at the
    // end of the "default" case
    msg = "unknown_error" ;
}

```

*Incorrect example - use of the switch as a goto:*

```

switch ( phase ) {
  case PHASE1 :
    doPhaseOne() ;
  case PHASE2 :
    doPhaseTwo() ;
    break ;
  default :
    /* ... */
}

```

**R66** Programs must always end with a return code via the **exit** (<integer>) function or via the **return** <integer>; instruction in the main function, using the **EXIT\_SUCCESS** or **EXIT\_FAILURE** values. The **EXIT\_FAILURE** constant may be replaced, if necessary, with a catalogued return code whose value is between 2 and 127. Numbers starting from 128 are reserved for the operating system.

*Example:*

```

int
main ( int argc , char *argv [] )
{
  /* ... */
  return EXIT_SUCCESS ;
}

```

### 4.13 Exceptions

The ability to raise and handle exceptions is one of the strongest features of C++. However, writing functions and methods that guarantee a safe behavior when faced with exceptions remains a difficult aspect of programming.

This chapter describes how to define and use exceptions in the source code.

**R67** When writing a function, the developer will bear in mind that the code must first perform all operations that may throw an exception, and then perform operations that change the state of an object or of the system (see *Exception-Safety Issues and Techniques, pages 25-68, Exceptional C++*).

**R68** Exceptions used by the Open TURNS code are inherited from the general class **Exception**. Other types of exceptions are simply processed.

*Example of Exception use*

```
class Exception {
public :
    Exception ( const char *description , const char * comment = 0 ) ;
    virtual ~Exception( ) throw();
    /* ... */
    friend ostream & operator<< (ostream &, const Exception & e ) ;
} ;
```

*Example of specialization of Exception in order to report an off-range error*

```
class OutOfBoundException : public Exception {
public:
    OutOfBoundException( /* ... */ )
    : Exception( /* ... */ ) { }
} ;
```

*Example of specialization of Exception in order to report an off-range error with a macro-instruction*

```
NEW_EXCEPTION( OutOfBoundException ) ;
```

**R69** C++ functions and methods throwing an exception should not specify it in the signature of the method or function using the **throw** keyword. It is better to mention it in the Doxygen comment describing the function or method (see rule R42). Furthermore, it is not recommended to pass an exception from function to function. If the error can not be processed, the exception must be transformed into a new, more explicit one, whose meaning is adapted to the function's level of knowledge.

**R70** Exceptions are reserved for error processing. Using an exception for a jump or as direct output of a function must be prohibited.

*Incorrect Example:*

```
try {
    // phase 1
    // phase 2
    if ( result != OK )
        throw GotoPhase4() ;
    // phase 3
```

```
/* ... */
catch ( GotoPhase4 e ) {
    /* ... */
}
// phase 4
```

**R71** The destructor of an object must not throw any exception.

**R72** The programming structure of *resource acquisition is initialization* (RAII) must be used preferably in order to get rid of complex processing when raising an exception.

**R73** Processing an exception should be made as close as possible to the function or method that throws this exception.

#### 4.14 Error handling and error messages

**R74** Information, warning and error messages are constructed using a message number and the class corresponding to the message level (respectively **Log::User**, **Log::Warning** and **Log::Error**). Messages can be enhanced with comments during construction or by using the redirection operator.

*Example:*

```
String message;
Log::Debug( message );
Log::Wrapper( message );
Log::Info( message );
Log::User( message );
Log::Warning( message );
Log::Error( message );
```

## 5 Python language

These rules refer to the classes and methods in the Python layer using the services of the internal model and the Open TURNS solvers.

### 5.1 Modules and packages

**R75** For the modules, it is preferable to use `import module` or `from module import symbol1, symbol2, ... , symbolN` rather than the form allowing to load the entire content of a module, `from module import *`.

### 5.2 Names

**R76** The names of the Python objects proposed by the Open TURNSPython module follow the same rules as the C++ ones, except when the Python naming scheme or keywords prevent from doing so.

*Examples: RandomVector, NumericalSample.*

**R77** The names of attributes and functions follow the same rules as the C++ ones, except when the Python naming scheme or keywords prevent from doing so.

*Examples:*

```
rv = RandomVector()
dim = rv.getDimension()
```

### 5.3 Comments and internal documentation

**R78** The documentation string will be used to build the online help on the use of a class, a class method or or a function. Comments before the definition of the class or of its methods will be used to document the code regarding design aspects. These comments can be exploited by tools similar to Doxygen, such as pydoc or Happydoc.

*Example of documentation string for the class AnotherNumericalSample:*

```
#
# <detailed description for documentation tools such as HappyDoc>
#
class AnotherNumericalSample :
    """
    this class is designed to ...
    """
    #
    # <detailed description for developers and documentation tools>
    def __init__(self, name, type, range = None, doc = "") :
        """constructor — """
        ...
    #
    # <detailed description for developers and documentation tools>
    def computeSomething( self, value ):
```

```
"""run the well-known Schmol Algorithm...  
"""
```

## References

- [Dav] Tal Davidson. Astyle, artistic style "a free, fast and small automatic indentation filter for c, c++, java source codes". <http://astyle.sourceforge.net>.
- [DOCa] Information technology - programming languages - c++, iso/iec 14882:1998 standard.
- [DOCb] Information technology - programming languages - fortran. Part 1: Base Language, ISO/IEC 1539-1:1997 standard; Part 2: Varying length, ISO/IEC 1539-2:2000 standard; Part 3: Conditional compilation, ISO/IEC 1539-3:1999 standard.
- [DOCc] Unix manuals for gcc (g++), cc.
- [GHJV] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Programs*.
- [GMM] Matthieu GUILLO, Olivier MORVANT, and F. MOREAU. Projet descartes - règles de programmation. Note 2003 SERMA/02-3199/B, HI-27/2002/053/B.
- [Jos] Nicolai M. Josuttis. *The C++ Standard Library, A Tutorial and Reference*. Addison Wesley.
- [Lau] Jean-Jacques Lautard. Guide de programmation du logiciel cronos2. Note 1997 SERMA/GUI/001/A.
- [Lut] Mark Lutz. *Python Programming (2nd Edition)*. O'Reilly.
- [Man] John E. Grayson Manning. *Python and Tkinter Programming*.
- [Str] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Professional.
- [Sut] Herb Sutter. *Exceptional C++, 47 Engineering Puzzles, Programming Problems and Solutions*. Addison Wesley.
- [vH] Dimitri van Heesch. Doxygen, a documentation system for c++, java, idl and c. <http://www.stack.nl/~dimitri/doxygen/index.html>.

## Index

- Abbreviation, 13
- Access method, 17
- Attribute
  - initialization, 16
  - name, 12
  - private, 15
  - protected, 15
- Built-in type, 11
- Class
  - member data, 12
- Class declaration
  - basic structure, 15
- Comparison, 15
- Composite type, 17
- Const, 19
- Constant
  - name, 13
- Constructor, 15
- Control structures, nested, 23
- Copy assignment, 15
- Copy constructor, 15
- delete, 22
- Destructor, 15, 28
  - virtual, 18
- Exception, 27
  - throw, 27
- exit, 26
- Filenames
  - class, 7
  - header file, 6
  - source code, 6
- Function
  - name, 13
- goto, 24
- Inclusion
  - external files, 8
  - multiple, 7
  - Open TURNS files, 8
  - system functions, 8
- Indentation, 23
- Inheritance
  - multiple, 18
- Inline, 17, 20
- Library
  - dynamic, 5
  - static, 5
- Member data
  - of class, 12
- Memory
  - allocation, 21
  - freeing, 21
  - management, 22
- Method
  - name, 12, 13
- Modules
  - dynamic library, 5
  - static library, 5
- Multiple inclusions, 7
- Multiple inheritance, 18
- Namespace, 10
- Nested control structures, 23
- new, 22
- Parameter passing by reference, 19
- Private, 14, 15
- Protected, 14, 15
  - protected, 15
- Public, 14
- Python
  - documentation string, 29
  - from, 29
  - import, 29
- Resource acquisition is initialization, 28
- Return code, 26
- Simple type, 11, 17
- Smart pointer, 22
- Standard
  - C++, 10
- Static
  - attribute name, 12
  - method name, 12
  - variable name, 13

switch

default, 25

Template, 20

Temporary objects, 24

Type

built-in, 11

composite, 17

name, 11, 13

simple, 11, 17

Variable

automatic, 21

declaration, 20

name, 12, 13

Virtual, 16

destructor, 18