How to add some code into CASToR: observing general programming guidelines

April 23, 2018

Foreword

CASToR is designed to be flexible, but also as generic as possible. Any new implementation should be thought to be usable in as many contexts as possible; among all modalities, all types of data, all types of algorithms, etc.

Before adding some new code to CASToR, it is highly recommended to read the general documentation *CASToR_general_documentation.pdf* to get a good overview of the project, as well as this HowTo guide which provides information on the syntax used throughout the code, and some general programming guidelines to be observed when contributing to CASToR. This guide does not follow any particular organization so please be sure to read it entirely.

1 Class names

Begin with a lower case letter to identify the type of the class:

- s: for singleton manager classes
- o: for an standalone class
- v: for abstract classes (pure virtual) that cannot be used on their own
- i: for a child class that inherits from a virtual class (with prefix v)
- g: for any file which is actually not a class

No underscore, all words attached and upper case at the beginning of each word. When an object is used, the first letter after the prefix must be upper case.

Listing 1: Examples of class names.

```
1 class oChicken;
2 class sMyManager;
3 class vMyVirtualClass;
4 class iChildChicken;
```

2 Class members

All class members begin with a lower case 'm', then follows the number of pointers indicated by the letter 'p', then underscore, and then begin the name of the member with two possibilities; i) first letter in lower case if it is a built-in non-object type, ii) first letter in upper case if it is an object. Then all words attached with upper case at the beginning of each word.

Listing 2: Examples of class members naming.

```
int m_nbChicken;
int* mp_chickenSizes;
int** m2p_chickenChickenSizes;
int*** m3p_chickenChickenChickenSizes;
oChicken* mp_Chicken;
```

3 Class functions

Destructor declaration and implementation is mandatory. No prefix, no underscore, all words attached, begins with an upper case, upper case at the beginning of each word. Parameters begin with prefix 'a[xp]_' followed by all words attached and upper case at the beginning of each word ('xp' for any pointers, as for class members eg: *a2p_chickenChickenSizes*), except for built-in non-object parameters where the first letter after the underscore is in lower case.

Listing 3: Examples of class functions naming.

```
int MyFunctionThatReturnsAnInteger(int a_intNumber, int* ↩
    ap_pointerToAnInteger, oChicken* ap_MyChicken);
```

As a general rule, all functions must return an integer specifying the status of the function: 0 means success and any other value means an error occured. By doing so, any error encountered at any level can be caught by the main which called the function which called the function which called the function, etc. Each time a function is called, the returning value must be tested. Exceptions include small functions that cannot possibly encounter errors and/or are called many times during the process, or functions returning a value. For the last case, as most functions return values superior or equal to 0, a returned negative value is usually used as an error indicator.

In the implementation files (*i.e.* '.cc' extension), use the following separator between two functions (one empty line before, another after):

Listing 4: Functions separator.

```
// ============================================================================
// ----------------------------------------------------------------------------
// ----------------------------------------------------------------------------
// ============================================================================
```

4 Class constructor, parameters and initialization

Most of the classes are built on the same concept. The constructors do not have parameters, must be called without parameters, and their sole role is to initialize all members with a default value. All parameters are then provided through the use of dedicated *Set()* functions. Once done, a *CheckParameters()* function is called to check that all mandatory parameters are correctly set. Follows a call to an *Initialize()* function which performs all required initialization steps, as the name suggests. Then the class can be used through its other dedicated member functions.

5 Local variables

Syntax of the local variables involves underscore to separate all words. No upper case is used, except for the first letter of the variable's name if it is an object.

Listing 5: Examples of local variables naming.

```
int nb_chicken = 2;
oChicken* p_My_chicken = NULL;
```

6 Indentation

Two spaces is the standard for indentation in the entire code. No tabulations.

7 Comments

Comment as much as possible inside the code to explain what is done and why, you will be rewarded for sure. In a new class, in the specification file, include a description of the purpose of the class. This is done using the doxygen syntax. Each function is documented in the specification file using the doxygen syntax. Documentation includes the aim of the function, description of the different parameters, and (optionally) details on the implementation. Documentation of the function in the implementation file is greatly advised, but not mandatory, and don't follow any specific format.

Inside a function, comment on the methodology using standard C++ '//' signs; do not use the '/*' then '*/' signs.

8 Source tree

One root folder, inside which there are the main programs, the makefiles, two folders for the sources (*include* and *src*) containing themselves thematic folders, one folder for the configuration files (*config*) containing itself different folders for specific purposes (e.g. *scanner*, *projector*), and one folder for utilitaries code (*toolkits*). The makefile will create a *bin* and a *tmp* folder. Each class is composed by at least 2 files when CPU only is used, and 3 files when GPU is used.

- A specification file with '.hh' extension, located inside a thematic folder of the include folder
- An implementation file for the CPU part, with '.cc' extension, located inside a thematic folder of the src folder
- An implementation file for the GPU part (if needed), with '.cu' extension, located inside a thematic folder of the src folder.

The files' names are exactly the same as the class name with the associated extension. For independent GPU functions, the file name begins with a prefix 'g', and the specification file extension is '.cuh' and implementation file extension '.cu'.

9 Non-class files

For other files that include some general functionnalities not contained in classes, the file name begins with the lower case 'g'. Currently there is one important header file *include/management/gVariables.hh* which contains important macro definitions, two other files *include/management/gOptions.hh* and *src/management/gOptions.cc* that implement some functions related to command line parameters reading.

10 Specific rules

At the beginning of each file, include the *gVariables.hh* file. It contains all useful includes from the standard library. It also contains the line saying the *std* namespace is used, so the use of another namespace is not allowed.

11 Messages display and verbosity

Macro definitions are used to display messages to the standard output or error interfaces, namely *Cout* and *Cerr* respectively. These macros allow to automatically log any message into a log file that is also automatically created when CASToR is used. The name and localization of this log file are based on where CASToR is executed and the provided output base name. See examples in the code to know how to use these macros.

The verbosity level, whatever the class, should obey the following guidelines as much as possible (the test over the verbose level is incremental '>='):

- **0**: No output except warnings and errors.
- **VERBOSE_LIGHT 1**: The lightest setting for progression status (1 message from each involved manager starting initialization and basis status of reconstruction).
- **VERBOSE_NORMAL 2**: The standard setting for progression status (specific types of modules used during initialization and more details reconstruction).
- **VERBOSE_DETAIL**: The heaviest setting for progression status (all relevant information during initialization and reconstruction).
- **VERBOSE_DEBUG_LIGHT 4**: A message at the beginning of each function that is called only a few times, with prefix "+++++". This is done by the macro DEBUG_VERBOSE which itself is described only when CASTOR_VERBOSE is set during compilation.
- **VERBOSE_DEBUG_NORMAL 5**: A message at the beginning of each function that is called many times, excluding those at the event level, with prefix "+++++" This is done by the macro DEBUG_VERBOSE which itself is described only when CASTOR_VERBOSE is set during compilation.
- **VERBOSE_DEBUG_EVENT 6**: A message at the beginning of each function that is called many times, including those at the event level, with prefix "+++++" This is done by the macro DEBUG_VERBOSE which itself is described only when CASTOR_VERBOSE is set during compilation. Also any other message more specific than just entering a function, at the event level.
- **VERBOSE_DEBUG_MAX 7+**: Any message originating from a function below the event level (*e.g.* action on a voxel for an event), with prefix "+++++" This is done by the macro DEBUG_VERBOSE which itself is described only when CASTOR_VERBOSE is set during compilation. Also any other message more specific than just entering a function, below the event level.

The messages prompting a call to a function, including in levels 4, 5 and 6, are embedded into a specific macro definition DEBUG_VERBOSE, defined in the file *include/management/sOutputManager.hh*. At these verbose levels, the messages are not logged. For the levels 6 and 7++ (at or below the event level), the other messages are explicitely implemented within the code but should still be encapsulated within the CASTOR_VERBOSE environment variable.

Note that the harmonization of verbose levels is not an easy task, so this is a permanent work-in-progress.

12 Configuration files

There is a *config* folder that contains all possible configuration files organized in sub-folders. The *CASTOR_CONFIG* environnement variable or CMake variable must be set to the config folder of CASToR in order to be able to find the configuration files (or use the '-conf' option). Here is a list of all sub-folders including:

- *convolver* folder, which contains the configuration files specific to the different image convolvers
- *misc* folder, which contains miscellaneous configuration files, such as isotopes parameters
- *optimizer* folder, which contains the configuration files specific to the different optimizers
- *processing* folder, which contains the configuration files specific to the different image processing modules
- *projector* folder, which contains the configuration files specific to the different projectors
- *scanner* folder, which contains the scanner configuration files

13 Precision of the computation (floating point and integer numbers)

The precision of all computations in CASToR are left to the choice of the user when compiling the code. To do so, simple macros defining the types of the variables are used. They are defined in the *include/management/gVariables.hh* file. Here are the different macro-types:

FLTNB : This macro-type defines the precision of floating point numbers used in all standard operations and image matrices. IT can be *float*, *double* or *long double*. Any time you define a floating point variable, use this macro-type instead of the actual standard C type, or use the next one HPFLTNB.

HPFLTNB : This macro-type defines the precision of floating point numbers that require a high level of precision, usually at least double precision. Basically use this type for any part of the code that is sensitive to precision.

FLTNBDATA : This macro-type defines the precision of floating point numbers that are read or written, for datafiles.

FLTNBLUT : This macro-type defines the precision of floating point numbers that are read or written, for scanner LUT file.

INTNB : This macro-types defines the precision of integer numbers used for images dimensions.

14 Doxygen

Technical documentation is generated using Doxygen. Doxygen comments must only appear in the header files (*i.e.* '.hh' extension).