GammaLib CDC

Coding and Design Conventions

Version draft
12 October 2012

Author: Jürgen Knödlseder

Institut de Recherche en Astrophysique et Planétologie (IRAP)
9, avenue du Colonel-Roche
31028 Toulouse Cedex 4
FRANCE

This page intentionally left blank

Contents

1 Introduction

This document summarises the coding and design conventions that should be following for the `GammaLib` development.

Respecting uniform and coherent coding and design conventions is crucial for software development. They improve code readability, ease code development and maintenance, ensure code portability, and provide standards for the user interface.

Coding and design conventions were introduced at a very early stage of the `GammaLib` project, and are followed as strictly as possible. The choices that were made for `GammaLib` were inspired by a large survey of existing C++ coding rules, combined with the experience of the leading code developers. Obviously, there is no single best way to code in C++, and the adopted conventions may not meet the coding usage of individual developers. They should nevertheless be respected strictly, as the `GammaLib` code base is already very large, and assuring uniformity and coherence is a primary goal of the project.

2 General coding rules

This section provides lists of general coding rules for the `GammaLib` development.

2.1 C++ rules

Code format

- Blocks are indented by 4 characters.
- Do not use tabs (code formatting should be independent of editor configurations).
- Do not exceed a line length of 80 characters (a few more characters are in exceptional cases acceptable).
- Put a blank line at the end of each file (this is required by some compilers).
- Each function starts with a curly bracket in the line following the function name. The return type is in the same line as the function name. Example:

```
void function(void)
{
    int = 0;
    return;
}
```

- Each block should be encompassed in curly brackets, even if the block consists only of a single line.
- The opening curly bracket of a block starts in the same line as the related statement. Example:

```
for (int i = 0; i < 10; ++i) {
    sum += i;
}
```

- Use spaces between code elements (see above example).
- Align successive similar lines on common elements. Example illustrating the alignment on the `=` symbol:

```
m_max    = par.m_max;
m_prompt = par.m_prompt;
sum     += par.m_sum;
```

Example illustrating the alignment in a class definition on the member function name:

```
void        log10GeV(const double& eng);
void        log10TeV(const double& eng);
std::string print(void) const;
```

Code semantics

- Each function and/or method terminates with a `return` statement.
- Each function and/or method has only a single exit point (i.e. a single `return` statement).
- Use `explicit` for constructors with arguments to prevent unintended type conversions. The only exception to this rule is the copy constructor or type conversion constructors.
- Specify `void` for function definitions without arguments.
- Use pre incrementation in loops (pre incrementation is faster than post incrementation). Example:

```
for (int i = 0; i < 10; ++i) {
    sum += i;
}
```

- Where possible (and appropriate), use `std::vector` containers instead of allocating memory. In other words: avoid direct memory allocation.
- Provide comments, comments, comments!!!

Language features

- Do not use templates.
- Do not use macros.
- Do not use namespaces.
- Do not use `#define` directives for the declaration of constants. Use `const` instead.
- Do not use `std::strncpy`, `std::memcpy` or similar as these functions are corrupted on some systems.
- If possible, pass arguments by reference.
- Use C++ (std::string) instead of C-style (char*) strings.
- Use C++ casts instead of C-style casts.

3 Coding conventions

The style summarizes coding conventions for the `GammaLib` development.

3.1 C++ classes

3.1.1 General rules

Each class should be defined in a pair of individual files:

- a header file that defines the class interface (with filename suffix `.hpp`)
- a source code file that implements the class (with filename suffix `.cpp`)

In addition, a SWIG interface file should be provided for the Python bindings (with filename suffix `.i`).

Each file should contain the `#include` directives that are necessary for compilation of the specific file. `#include` directives that are specified in the header file can be omitted in the source code file, provided that the header file is include in the source code file.

The C++ style header files should be used instead of the C style header files to ensure maximum portability. The following table provides the correspondence between C++ and C header files for headers commonly used in `GammaLib`.

C++	C	Function examples
`<cctype>`		
`<cmath>`	`<math.h>`	`std::abs`, `std::cos`
`<cfloat>`		
`<cstdio>`	`<stdio.h>`	`std::fopen`, `std::fgets`, `std::fclose`, `std::fprintf`, `std::sprintf`
`<cstdlib>`		
`<cstring>`	`<string.h>`	`std::strlen`

Note that functions and types should be prefixed by `std::`. For example, `cos` becomes `std::cos`, `time_t` becomes `std::time_t`, etc. One significant change between C and C++ is that `fabs` becomes `std::abs` since the C style `abs` function only applies to integers. Here, the `std::` prefix is crucial to distinguish the C++ function (which is also defined for doubles) from the C function.

3.1.2 Header file structure

The header file defines the interface of the class. Here is an example of a header file.

```
/***************************************************************************
 *                         GClass.hpp  -  My nice class                    *
 * ----------------------------------------------------------------------- *
 *  copyright (C) 2010-2012 by Juergen Knoedlseder                         *
 * ----------------------------------------------------------------------- *
 *                                                                         *
 *  This program is free software: you can redistribute it and/or modify   *
 *  it under the terms of the GNU General Public License as published by   *
 *  the Free Software Foundation, either version 3 of the License, or      *
 *  (at your option) any later version.                                    *
 *                                                                         *
 *  This program is distributed in the hope that it will be useful,        *
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of         *
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the          *
 *  GNU General Public License for more details.                           *
```

```
 *                                                                         *
 *  You should have received a copy of the GNU General Public License      *
 *  along with this program.  If not, see <http://www.gnu.org/licenses/>.  *
 *                                                                         *
 ***************************************************************************/
/**
 * @file GClass.hpp
 * @brief Definition of my nice class interface
 * @author Juergen Knoedlseder
 */

#ifndef GCLASS_HPP
#define GCLASS_HPP

/* __ Includes ___________________________________________________________ */
#include <string>
#include <iostream>
#include "GLog.hpp"


/***************************************************************************//**
 * @class GClass
 *
 * @brief Illustration of a GammaLib class
 *
 * My nice class illustrates how a GammaLib class should be defined.
 ******************************************************************************/
class GClass {

    // I/O friends
    friend std::ostream& operator<< (std::ostream& os, const GClass& c);
    friend GLog&         operator<< (GLog& log, const GClass& c);

public:
    // Constructors and destructors
    GClass(void);
    GClass(const GClass& c);
    virtual ~GClass(void);

    // Operators
    GClass& operator= (const GClass& c);

    // Methods
    void        clear(void);
    GClass*     clone(void) const;
    std::string print(void) const;

protected:
    // Protected methods
    void init_members(void);
    void copy_members(const GClass& c);
    void free_members(void);
```

```
    // Protected data members
    std::string     m_name;          //!< Name
};

#endif /* GCLASS_HPP */
```

The header file starts with a comment containing the file name and class purpose, the copyright information and the license text. The years in the copyright information should cover the years over which the file has been modified, the author is the person who initially created the file.

Following the header comment is a comment that provides file information to the Doxygen documentation system.

The subsequent

```
    #ifndef GCLASS_HPP
    #define GCLASS_HPP
```

declarations together with the

```
    #endif /* GCLASS_HPP */
```

declaration at the end protect the file from multiple inclusions of the header. This is a crucial feature needed for proper compilation of the code.

Now all header files are included. Standard header files are included using the `< >` brackets, `GammaLib` header files are included using `" "`. A 80 character long separator precedes the header inclusion. Further 80 character long separators may be added for additional sections, such as constants, type definitions, forward declarations, etc. Use one separator to precede each additional section.

The class definition is preceded by a comment block that will be used by the Doxygen system to extract the class definition. Provide here the class name, a brief one line description of the class, and an extended detailed description of the class purpose.

The class definition is structured in several sections:

- Declaration of friend functions
- Definition of public constructors
- Definition of public operators
- Definition of public methods
- Definition of protected methods
- Definition of protected members

The definition of pure virtual methods should be done in a section that is separate from the methods that are implemented. Output stream and logging operators should be implemented for every class as friend operators.

Here any illustration of the expected structure, based on the `GObservation` class:

```
class GObservation {

    // Friend classes
```

```
    friend std::ostream& operator<< (std::ostream& os, const GObservation& obs);
    friend GLog&         operator<< (GLog& log,        const GObservation& obs);

public:
    // Constructors and destructors
    GObservation(void);
    GObservation(const GObservation& obs);
    virtual ~GObservation(void);

    // Operators
    virtual GObservation& operator= (const GObservation& obs);

    // Pure virtual methods
    virtual void          clear(void) = 0;
    virtual GObservation* clone(void) const = 0;

    // Virtual methods
    virtual double        model(const GModels& models, const GEvent& event,
                                GVector* gradient = NULL) const;
    virtual double        npred(const GModels& models, GVector* gradient = NULL) const;

    // Implemented methods
    void                  name(const std::string& name);
    void                  id(const std::string& id);

protected:
    // Protected methods
    void init_members(void);
    void copy_members(const GObservation& obs);
    void free_members(void);

    // Protected data area
    std::string m_name;         //!< Name of observation
    std::string m_id;           //!< Observation identifier
    std::string m_statistics;   //!< Optimizer statistics (default=poisson)
    GEvents*    m_events;       //!< Pointer to event container
};
```

3.1.3 Source code file structure

The source code file implements the code of the class. Here is an example of the start of a source code file.

```
/***************************************************************************
 *                        GClass.cpp  -  My nice class                     *
 * ----------------------------------------------------------------------- *
 *  copyright (C) 2010-2012 by Juergen Knoedlseder                         *
 * ----------------------------------------------------------------------- *
 *                                                                         *
 *  This program is free software: you can redistribute it and/or modify   *
 *  it under the terms of the GNU General Public License as published by   *
 *  the Free Software Foundation, either version 3 of the License, or      *
 *  (at your option) any later version.                                    *
 *                                                                         *
```

```
 *  This program is distributed in the hope that it will be useful,        *
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of         *
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the          *
 *  GNU General Public License for more details.                           *
 *                                                                         *
 *  You should have received a copy of the GNU General Public License      *
 *  along with this program.  If not, see <http://www.gnu.org/licenses/>.  *
 *                                                                         *
 ***************************************************************************/
/**
 * @file GClass.cpp
 * @brief Implementation of my nice class
 * @author Juergen Knoedlseder
 */

/* __ Includes _____________________________________________________________ */
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
#include "GClass.hpp"
#include "GTools.hpp"

/* __ Method name definitions ______________________________________________ */
#define G_CLEAR                                          "GClass::clear()"
#define G_CLONE                                   "GClass::clone() const"
#define G_PRINT                                   "GClass::print() const"

/* __ Compile options ______________________________________________________ */
#define G_USE_MY_OPTION

/* __ Debug options ________________________________________________________ */
#define G_DEBUG_PRINT

/* __ Constants ____________________________________________________________ */
const double pi = 3.14;
```

The include section starts with a conditional include of the code configuration header file (see section 4.1). This makes `GammaLib` compile options available to the source code.

The include section is followed by the declaration of method names. These method names will be used in exceptions (see section 4.2.9). Define the method names at the top of the file eases the maintainability of the code, as changes in method names or interfaces need only to be implemented in a single place. Method names need only be defined for methods throwing exceptions.

Compile options are used to control which parts of the code should be compiled. Such options may be used, for example, to compare different algorithms or computation methods. They can also be used during development, allowing an easy switch between the new and the old code for comparison.

Debug options are compile options that are used to add additional code for debugging. Often, these are print statements that allow to trace the execution of the code. For code checked into the repository, all debug options should be commented out.

3.2 Python classes

The Python interface for C++ classes is defined by a so-called SWIG interface file. SWIG uses these interface files to build Python wrapper files, which are C files that define the interface between `GammaLib` and Python. The structure of the SWIG interface file follows closely that of the header file, with a few exceptions. Here an example:

```
/***************************************************************************
 *                          GClass.i  -  My nice class                     *
 * ----------------------------------------------------------------------- *
 *  copyright (C) 2010-2012 by Juergen Knoedlseder                         *
 * ----------------------------------------------------------------------- *
 *                                                                         *
 *  This program is free software: you can redistribute it and/or modify   *
 *  it under the terms of the GNU General Public License as published by   *
 *  the Free Software Foundation, either version 3 of the License, or      *
 *  (at your option) any later version.                                    *
 *                                                                         *
 *  This program is distributed in the hope that it will be useful,        *
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of         *
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the          *
 *  GNU General Public License for more details.                           *
 *                                                                         *
 *  You should have received a copy of the GNU General Public License      *
 *  along with this program.  If not, see <http://www.gnu.org/licenses/>.  *
 *                                                                         *
 ***************************************************************************/
/**
 * @file GClass.i
 * @brief Python interface of my nice class
 * @author Juergen Knoedlseder
 */
%{
/* Put headers and other declarations here that are needed for compilation */
#include "GClass.hpp"
#include "GTools.hpp"
%}


/***************************************************************************//**
 * @class GClass
 *
 * @brief Illustration of a GammaLib class
 *
 * My nice class illustrates how a GammaLib class should be defined.
 ******************************************************************************/
class GClass {
public:
    // Constructors and destructors
    GClass(void);
    GClass(const GClass& c);
    virtual ~GClass(void);
```

```
    // Methods
    void        clear(void);
    GClass*     clone(void) const;
};


/***************************************************************************//**
 * @brief GClass class extension
 ******************************************************************************/
%extend GClass {
    char *__str__() {
        return tochar(self->print());
    }
    GClass copy() {
        return (*self);
    }
};
```

The code starts with a section that is enclosed in `%{ %}` brackets. In this section, all header files are specified that are needed to compile the SWIG wrapper file.

Then follows the class definition, with the following differences with respect to the definition in the header file:

- it does not include the assignment operator
- it does not include any access operator (these have to be implemented specifically, see below)
- it does not include the `print()` method (see below)
- it does not include protected or private members

Finally, there is a section with extension to the C++ class. Here, methods are implement that do not exist in the actual C++ class, but that will exist in the Python interface. In this example, the `__str__()` method is the generic Python conversion operator that converts an object into a string. It can be seen that this method calls the class's @print()@ method. By providing the conversion method, the class becomes "printable", and one my specify

```
>>> GClass c
>>> print c
```

to print out the class (i.e. to invoke the print() method of the class).

In case that an access operator needs to be implement, the `__getitem__()` and `__setitem__()` methods need to be added to the class extensions.

4 Design conventions

4.1 Code configuration

The code configuration is controlled via the `config.h` header file that is created during the configuration step of `GammaLib`. To make configuration options available the following code has to be added to the source code file:

```
/* __ Includes _____________________________________________________ */
#ifdef HAVE_CONFIG_H
#include <config.h>
#endif
```

Note that the `config.h` file should **not be included in header files**, since header files are used by the outside world without any knowledge about `config.h`.

One of the most commonly used configuration options used throughout `GammaLib` is related to range checking. Range checks are usually performed when accessing array elements, assuring that no elements outside the valid range are accessed. Range checking, however, is time consuming, in particular if many elements are accessed subsequently. `GammaLib` therefore allows to disable range checks. This can be done by specifying `./configure --disable-range-check` when configuring `GammaLib` for compilation. The `--disable-range-check` option undefines `G_RANGE_CHECK`, and optional range checking is thus achieved by adding for example

```
#if defined(G_RANGE_CHECK)
if (inx < 0 || inx >= m_num) {
    throw GException::out_of_range("GVector::operator(int)", inx, m_num);
}
#endif
```

to the code.

The following table gives a list of important configuration options that are available in `config.h` and that can be used to tune the code:

Definition	Option	Usage
`G_DEBUG`	`--enable-debug`	Code debugging
`G_PROFILE`	`--enable-profiling`	Code profiling
`G_RANGE_CHECK`	`--enable-range-check`	Performs range checking
`G_NAN_CHECK`	`--enable-nan-check`	Check for NaN and Inf values
`G_SMALL_MEMORY`	`--enable-small-memory`	Optimizes for small memory
`HAVE_OPENMP`	`--enable-openmp`	Has OpenMP multi-threading support
`HAVE_LIBREADLINE`	`--with-readline`	Has readline library
`HAVE_LIBCFITSIO`	`--with-cfitsio`	Has cfitsio library
`HAVE_PYTHON`	`--enable-python-binding`	Has Python bindings
`PACKAGE`	n.a.	`gammalib`
`PACKAGE_PREFIX`	n.a.	Installation location (e.g. `/usr/local/gamma`)
`PACKAGE_STRING`	n.a.	Full name and version of `GammaLib`
`PACKAGE_VERSION`	n.a.	Version of `GammaLib` (format: x.y.z)

Note that `enable` may be replaced by `disable` and `with` by `without` for switching off an option.

4.2 C++ classes

4.2.1 Members

Class members should be either `private` or `protected`, the latter being generally used when a derived class should be able to access base class data.

Members should be prefixed by `m_` and should be in lower case. For long member names, additional underscores may be added. Examples of valid member names are

```
m_num
m_response
m_grid_length
m_axis_dir_qual
```

Initialisation, copying and deleting of class members should be gathered in a single place to avoid memory leaks. For this purpose, each C++ class should have the following `private` or `protected` methods for memory management:

- `init_members()` initializes all member variables and pointers to well defined initial values. The class should be fully operational and consistent with these initial values. All pointers that will hold dynamically allocated memory should be initialised to `NULL`.
- `copy_members(const &A a)` copies all members from an instance `a` into the class.
- `free_members()` frees all memory that has been allocated by the class. Memory pointers should be set to `NULL` after the memory was deleted to signal that no valid memory is associated to the pointer. This allows for checking if memory has been allocated before it is accessed.

(in the above notation, `A` is the class name and `a` is an instance of the class).

An example for valid `init_members()`, `copy_members(const &A a)` and `free_members()` methods is:

```
void GEbounds::init_members(void)
{
    m_num = 0;
    m_min = NULL;
    m_max = NULL;
    return;
}

void GEbounds::copy_members(const GEbounds& ebds)
{
    m_num  = ebds.m_num;
    if (m_num > 0) {
        m_min = new GEnergy[m_num];
        m_max = new GEnergy[m_num];
        for (int i = 0; i < m_num; ++i) {
            m_min[i] = ebds.m_min[i];
            m_max[i] = ebds.m_max[i];
        }
    }
    return;
}

void GEbounds::free_members(void)
{
    if (m_min != NULL) delete [] m_min;
    if (m_max != NULL) delete [] m_max;
    m_min = NULL;
    m_max = NULL;
    return;
}
```

In this example, on may probably want to add a `alloc_members()` method for memory allocation:

```
void GEbounds::alloc_members(const int& num)
{
    if (num > 0) {
        m_min = new GEnergy[num];
        m_max = new GEnergy[num];
        for (int i = 0; i < num; ++i) {
            m_min[i] = 0.0;
            m_max[i] = 0.0;
        }
        m_num = num;
    }
    return;
}
```

This example illustrates several design conventions:

- Do always check if a pointer is not `NULL` before de-allocating memory.
- After de-allocation, always set the pointer immediately to `NULL`.
- Do never allocate zero elements (check if the number of elements to be allocated is positive).
- Do always initialise allocated memory to well defined values (do not expect that the compiler will do this for your).

4.2.2 Constructors, destructors and operators

Each class should have at least a void constructor, a copy constructor, a destructor and an assignment operator. Additional constructors and operators can be implemented as required. The following example shows the basic implementation for these 4 methods. Due to the usage of the `init_members()`, `copy_members(const &A a)` and the `free_members()` methods, most classes will have exactly this kind of syntax:

```
GEbounds::GEbounds(void)
{
    init_members();
    return;
}

GEbounds::GEbounds(const GEbounds& ebds)
{
    init_members();
    copy_members(ebds);
    return;
}

GEbounds::~GEbounds(void)
{
    free_members();
    return;
}

GEbounds& GEbounds::operator= (const GEbounds& ebds)
```

```
{
    if (this != &ebds) {
        free_members();
        init_members();
        copy_members(ebds);
    }
    return *this;
}
```

4.2.3 Inheritance

Class inheritance is central feature of the C++ language, and is largely used throughout `GammaLib`. Multiple inheritance is not used at the moment in `GammaLib`. Because of the added complexity of multiple inheritance in C++ in python there would have to be very good reasons to use it in `GammaLib`.

Although the inheritance philosophy may differ from class to class, the following guidelines should be respected as far as possible:

- The base class and derived class destructors should be declared `virtual`.
- Avoid overloading of base class methods by derived class methods. Preferentially, define base class methods as pure virtual.
- All base class methods that should be implemented in the derived class should be declared `virtual`. Exceptions are the `init_members()`, the `copy_members()` and the `free_members()` methods that will be implemented in the base class and the derived class.
- Base classes manage base class members, derived classes manage derived class members. By managing we mean here in particular memory allocation and de-allocation, but also proper initialization.
- Derived class constructors should invoke base class constructors for proper base class initialization. A void constructor should look like

  ```
  GEventList::GEventList(void) : GEvents()
  {
      init_members();
      return;
  }
  ```

  and a copy constructor should look like

  ```
  GEventList::GEventList(const GEventList& list) : GEvents(list)
  {
      init_members();
      copy_members(list);
      return;
  }
  ```

- Derived class operators should invoke base class operators, as illustrated by the following example:

  ```
  GEventList& GEventList::operator=(const GEventList& list)
  {
      if (this != &list) {
          this->GEvents::operator=(list);
          free_members();
  ```

```
            init_members();
            copy_members(list);
        }
        return *this;
    }
```

- The `clear()` method of a derived class show invoke the `free_members()` method of the base class, as illustrated by the following example:

```
void GCTAEventList::clear(void)
{
    free_members();
    this->GEventList::free_members();
    this->GEvents::free_members();
    this->GEvents::init_members();
    this->GEventList::init_members();
    init_members();
    return;
}
```

- Avoid as far as possible methods that are only defined in the derived class.

Also note that **for a derived class, `init_members()`, `copy_members(const &A a)` and `free_members()` should only act on derived class members but not on base class members**. Any exception from this rule needs very careful documentation since it can easily be the source of memory leaks.

4.2.4 Method naming conventions

Uniform `public` method names should be provided throughout `GammaLib` for all classes. Unless the `public` method names are very long (which should be avoided), names should not comprise underscores as separators. `Public` method names are all lowercase.

`Private` or `protected` method name may differ from this since they are hidden within the class. Yet also here, all method names should be lowercase, and the use of underscores should be limited.

Methods that set or retrieve class attributes should be named after the attribute. Here an example for the attribute `m_name`:

```
public:
    void        name(const std::string& name);
    std::string name(void) const;
protected:
    m_name;
```

A method name that is used in multiple classes should always perform an equivalent action. Here is a list of method names that are widely used in `GammaLib`, together with their typical usage. The last column specifies where these methods are used. Note that **the `clear()`, `clone()`, and `print()` methods should be implemented for all classes**.

Method	Usage	Implementation
`clear()`	Set object to initial empty state	all classes
`clone()`	Provides a deep copy of the class	all classes
`print()`	Print object into string	all classes (see section 4.2.8)
`append()`	Append element to list of elements	container classes (see section 4.2.7)
`extend()`	Append container elements to list of elements	container classes (see section 4.2.7)
`insert()`	Insert element to list of elements	container classes (see section 4.2.7)
`pop()`	Remove element from list of elements	container classes (see section 4.2.7)
`load()`	Load data from file (open, read, close)	if applicable
`save()`	Save data into file (open, write, close)	if applicable
`open()`	Open file	if applicable
`read()`	Read data from open file	if applicable
`write()`	Write data into open file	if applicable
`close()`	Close file	if applicable
`name()`	Name of object	if applicable
`type()`	Type of object	if applicable
`size()`	Size of object	if applicable
`real()`	Returns `double` precision value	if applicable
`integer()`	Returns `int` value	if applicable
`string()`	Returns `std::string` value	if applicable

Note the difference between `load()` and `read()` and between `save()` and `write()`. The `load()` and `save()` methods should take as arguments a file name, and they will open the file, read or write some data, and then close the file. In contrast, `read()` and `write()` will operate on files that are already open, and after the read or write operation the files will remain open. Typically, these methods take a `GFits*` or a `GFitsHDU*` pointer as argument.

Methods that perform checks should return a `bool` type and should start with the prefix `is` or `has`. Examples are:

```
islong()
isin()
hasedisp()
```

4.2.5 Method const declarations

All methods that do not alter accessible class members should be declared `const`. With accessible we mean here class members that can be read or written in some way by one of the methods. Non-accessible class members would be members that are only used internally, and for which no consistent state has to be preserved for the outside world. These could for example be members that hold pre-computed values.

Methods that do not alter accessible members, but that modify non-accessible members, should also be declared `const`. The non-accessible class members need then to be declared `mutable` to avoid compiler errors. Alternatively, the `const_cast` declaration can be used to allow member modifications within a `const` method.

As example we show here part of the definition of `GModelSpectralPlaw2`:

```
class GModelSpectralPlaw2 : public GModelSpectral {
public:
    virtual double eval(const GEnergy& srcEng) const;
    virtual void   read(const GXmlElement& xml);
```

```
protected:
    // Protected members
    GModelPar       m_integral;        //!< Integral flux
    GModelPar       m_index;           //!< Spectral index
    GModelPar       m_emin;            //!< Lower energy limit (MeV)
    GModelPar       m_emax;            //!< Upper energy limit (MeV)

    // Cached members used for pre-computations
    mutable double  m_log_emin;        //!< Log(emin)
    mutable double  m_log_emax;        //!< Log(emax)
    mutable double  m_pow_emin;        //!< emin^(index+1)
    mutable double  m_pow_emax;        //!< emax^(index+1)
    mutable double  m_norm;            //!< Power-law normalization (for pivot energy 1 MeV)
    mutable double  m_g_norm;          //!< Power-law normalization gradient
    mutable double  m_power;           //!< Power-law factor
    mutable double  m_last_integral;   //!< Last integral flux
    mutable double  m_last_index;      //!< Last spectral index (MeV)
    mutable double  m_last_emin;       //!< Last lower energy limit (MeV)
    mutable double  m_last_emax;       //!< Last upper energy limit (MeV)
    mutable GEnergy m_last_energy;     //!< Last source energy
    mutable double  m_last_value;      //!< Last function value
    mutable double  m_last_g_integral; //!< Last integral flux gradient
    mutable double  m_last_g_index;    //!< Last spectral index gradient
```

This class has an internal cache for precomputation, which is potentially updated when `eval` is called. Here the corresponding code:

```
double GModelSpectralPlaw2::eval(const GEnergy& srcEng) const
{
    // Update precomputed values
    update(srcEng);

    // Compute function value
    double value = integral() * m_norm * m_power;

    // Return
    return value;
}
```

As the pre-computation cache is not exposed to the external world but fully handled within the class, `eval()` is declared `const` as it does not modify any of the model parameters (which are `m_integral`, `m_index`, `m_emin`, and `m_emax`). It may however modified some of the cache members, that's why these members are declared `mutable`. As there is however no way to access these cache values from the outside (no method exists to access them), the `eval()` method does not modify any *observable* property of the class, hence it is declared `const`.

4.2.6 Method arguments and return values

If possible, method arguments should always be passed by reference. To protect references from changes by the method, **arguments passed by reference should always be declared** `const`. Pointers should only be used as arguments if `NULL` should be a possible argument value. Also pointers should always be declared `const`. Here an example based on the definition of `GObservation`:

```
class GObservation {
public:
    void                  events(const GEvents* events);
    void                  statistics(const std::string& statistics);
protected:
    std::string m_statistics;   //!< Optimizer statistics (default=poisson)
    GEvents*    m_events;       //!< Pointer to event container
};
```

The `statictics` value is passed by reference because the class will hold the actual value, while `events` is passed as a pointer because the class will hold the pointer.

Numeric argument types should be either `int` or `double`. Unless absolutely necessary, avoid `short int`, `long`, and `float`.

If a method returns a class member, the return value should be passed by reference. Unless we explicitly want to modify a class member through the method call, return values passed by reference should be declared `const`.

If a method returns a base class object, a pointer should be returned. **Do never return base class objects by reference, as this will lead to code slicing if the method is used for object assignment.** Unless we explicitly want to modify a class member through the method call, the returned pointer should be declared `const`.

Here an example based on the definition of `GObservation`:

```
class GObservation {
public:
    virtual double        ontime(void) const = 0;
    const GEvents*        events(void) const;
    const std::string&    statistics(void) const { return m_statistics; }
protected:
    std::string m_statistics;   //!< Optimizer statistics (default=poisson)
    GEvents*    m_events;       //!< Pointer to event container
};
```

The `ontime()` method does return a double by value as the ontime property is not stored explicitly in the class (hence no reference can be returned to it). On the other hand, the `statistics()` method returns by reference as the statistics property is stored as a data member (hence a reference can be returned). Although we could have returned a reference to the event container, this would lead to code slicing. Therefore, the `events()` method returns a pointer. All returned references or pointers are declared `const` to prevent modification of class members.

4.2.7 Container classes

Container classes are classes that contain list of elements. Two cases are distinguished here: containers holding objects, and containers holding pointers to objects.

Containers holding objects Containers holding objects should have element access operators `operator[]` implemented that return container elements by reference. A non-const and a const version of the operator should exist. Eventually, `at()` methods could be added that always perform range checking. Here is a list of mandatory methods for container classes holding objects:

Method	Usage
`e& operator[](const int&)`	Element access operator
`const e& operator[](const int&) const`	Element access operator
`void clear()`	Delete all objects in container
`void size()`	Return number of elements in container
`void append(const e&)`	Append an element to the container
`void insert(const int&, const e&)`	Insert an element into the container
`void extend(const C&)`	Append another container to the container
`void pop(const int&)`	Removes an element from the container
`std::string print()`	Print container (see section 4.2.8)

Containers holding pointers Containers holding pointers different from those holding objects in that their `operator[]` operators return a pointer, and in that they implement a `set()` method for value setting. Here is a list of mandatory methods for container classes holding pointers:

Method	Usage
`e* operator[](const int&)`	Element access operator
`const e* operator[](const int&) const`	Element access operator
`void set(const int&, const e&)`	Set an element of the container
`void clear()`	Delete all objects in container
`void size()`	Return number of elements in container
`void append(const e&)`	Append an element to the container
`void insert(const int&, const e&)`	Insert an element into the container
`void extend(const C&)`	Append another container to the container
`void pop(const int&)`	Removes an element from the container
`std::string print()`	Print container (see section 4.2.8)

4.2.8 Output

Output stream and logging operators should be implemented for every class as friend operators (see section 3.1.2). The usage of friend operators (instead of member operators) allows for correct handling of code such as

```
log << std::endl << "This is a text" << std::endl;
```

To support these friend operators (and to support also the Python interface), each class should have a `print()` method:

```
std::string print(void) const;
```

Using the `print()` method the output operators will take the following generic form:

```
std::ostream& operator<< (std::ostream& os, const GFits& fits)
{
    os << fits.print();
    return os;
}
GLog& operator<< (GLog& log, const GFits& fits)
{
```

```
    log << fits.print();
    return log;
}
```

4.2.9 Exceptions

Exceptions are largely used in `GammaLib` to handle the occurrence of unexpected events. `GammaLib` exceptions are implement by the `GException` class. For each new exception type, a new exception subclass is added.

Each exception returns the method name in which the exception occurs and an exception message. The exception message is generally build from values that are passed as arguments to the exception constructor.

Below a list of conventions for implementing and using exceptions:

- Re-use existing exceptions as far as possible.
- Pass exception arguments by reference.
- Use exceptions only for events that cannot be handled by a method. Do not use exceptions to check a value or a state. Implement appropriate methods instead.
- Never use exceptions in a destructor.
- De-allocate all memory that is not de-allocated by the destructor before throwing an exception.
- Always catch exceptions by reference.

5 Miscellaneous

5.1 Version control

`GammaLib` applies a three-number version numbering scheme: `major_revision-minor_revision-patch`.

A `major_revision` of `00` indicates that the `GammaLib` design is not yet frozen. At this level, external interfaces of `GammaLib` may change without notification, and no interface control system is implemented. The `minor_revision` tag will be incremented for each new release, signaling that new features have become available. The `patch` tag will be incremented after correcting bugs that were reported on releases.

Once the `GammaLib` design is frozen, the `major_revision` number will be incremented to `01`. From this moment on, external interfaces of `GammaLib` will be under configuration control. If existing external interfaces will be modified, the `major_revision` number will be incremented. At the same time, the `libtool` version number of the `GammaLib` will also be incremented. The `minor_revision` number will be incremented if modifications and extensions are backward compatible As before, the `patch` number will be incremented after correcting bugs that were reported on releases.