GammaLib CCD

Coding Conventions Document

Version draft
10 November 2011

Author: Jürgen Knödlseder
Approved by: 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

The present document summarises the coding conventions that should be followed in implementing the `GammaLib` toolbox. The respect of coherent coding conventions throughout the code improves code readibility and enhances the portability of the code.

2 General coding rules

The following general rules should be followed:

R1 Each function and/or method terminates with a `return` statement.

R2 Each function and/or method has only a single exit point (i.e. a single `return` statement).

R3 Put a blank line at the end of each file.

R4 Use `explicit` for constructors to avoid use of the constructor for unintended type conversion.

R5 Do not use tabs to make code formating independent of editor configurations.

R6 Blocks are indented by 4 characters.

3 Coding style

3.1 Code configuration

The code configuration is controlled via an include file that has to be added on top of each source file. Each source file should start with:

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

Note that the `config.h` file should **not be included in header files**, since header file are used by the outside world for which a different `config.h` include file may exist.

One of the most common options used throughout `GammaLib` is range checking. Range checking is particularily important during code development since it allows to catch memory leaks. However, range checking is time consuming and thus leads to somewhat slower code. Range checking can thus be disable during installation of `GammaLib` by using `./configure --disable-range-check` during library installation. Within the code, the following instruction adds range checking that dependens on the library configuration:

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

Range checking is provided if the `G_RANGE_CHECK` macro is defined.

3.2 Header inclusions

Each file should contain the `#include` directives that are necessary for compilation of the specific file. Only `#include` directives that are already given by the corresponding header file can be omitted.

As `GammaLib` is written in C++, the C++ style headers should be used instead of the C style headers to ensure maximum portability. Examples of common C++ style headers are `<cstdio>` (instead of `<stdio.h>`), `<cmath>` (instead of `<math.h>`), `<cstring>` (instead of `<string.h>`), etc. Functions and types should then be prefixed by `std::`. For example, `cos` becomes `std::cos`, `time_t` becomes `std::time_t`, etc. One significant change 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.

Note also that some compilers are more tolerant in omitting `#include` directives, such as `<cstdio>` or `<cstring>`, so these directives should also be included for compatibility, even if they seem not to be required on specific systems. Examples of include directives needed by common functions are:

> `<cstdio>` contains `std::fopen`, `std::fgets`, `std::fclose`, `std::fprintf`, `std::sprintf`
>
> `<cstring>` contains `std::strncpy`, `std::memcpy`

If possible, however, `std::strncpy` and `std::memcpy` should be avoided at all as these functions happend to have some compatibility problems in the past.

3.3 C++ classes

3.3.1 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 one instance `a` to the `this` instance.
- `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;
}
```

3.3.2 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;
}
```

Note that for a derived class, the assignment operator will have the form:

```
GEventCube& GEventCube::operator= (const GEventCube& cube)
{
    if (this != &cube) {
        this->GEvents::operator=(cube);    // Copy base class members
        free_members();
        init_members();
        copy_members(cube);
    }
    return *this;
}
```

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.

3.3.3 Inheritence

Class inheritence is heavily used in `GammaLib` to implement instrument specific functionnalities that satisfy a generic interface.

Derived classes should never set explicitely base class members. Base class members should be set by proper constructors, and base class constructors are to to be invoked in derived class constructors.

3.3.4 Methods

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` 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.

In addition, the following `public` method names should be used:

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

Table 1: Naming conventions for class methods.

Method	Usage	Implementation
`clear()`	Set object to initial empty state	all classes
`print()`	Print object into string	all classes (see section **??**)
`append()`	Append element to list of elements	container classes
`size()`	Return number of elements in object	container classes
`load()`	Load data from file (open, read, close)	if applicable
`save()`	Save data into file (open, write, close)	if applicable
`read()`	Read data from open file	if applicable
`write()`	Write data into open file	if applicable
`name()`	Name of object	if applicable

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`.

Valid examples are:

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

Method arguments should be generally passed as `const` and by reference.

Numeric argument types should be typically `int` and `double`. Unless absolutely required, `short int`, `long`, and `float` should be avoided. True/false checks should always be done using `bool`.

Methods should be declared as `const` if they do not alter class members. In some classes, pre-computations are done to speed up calculations, and these pre-computations will alter class members. If these pre-computation are not supposed to alter the content of a class, methods that perform pre-computations should also be declared as `const`. Internally, they have to circumvent const correctness by casting the pointer to non `const` or by using the `mutable` attribute in the declaration of the members that hold the pre-computed values (the latter option is preferred).

3.3.5 Method arguments and return values

Arguments to methods should in general be passed by reference to reduce function overhead. In addition, arguments that are not intended to be changed by a method call should be passed as `const`. Return values should also be passed by reference. To assure that the return values will not change class members inadvertably, return values passed by reference should also be declared as `const`.

If the class member is a pointer, the argument should also be a pointer to allow setting the pointer to `NULL`. The exception to this are container classes which in general do not contain `NULL` pointers in the container.

Return values should be passed by reference if they concern class members.

If the class member is a pointer, and if the pointer may be `NULL`, it should be returned as a pointer. Again, the pointer should be declared `const` to avoid inadvertable changes of the class members. The exception to this are container classes which in general do not contain `NULL` pointers in the container. Container classes should return references to container elements.

An example is given by the following excerpt of the `GObservation` class. Note that in general there are two distinct methods to set and to return a class member.

```
    void               obsname(const std::string& obsname);
    void               ebounds(const GEbounds& ebounds);
    void               gti(const GGti& gti);
    void               roi(const GRoi* roi);
    void               events(const GEvents* events);
    void               statistics(const std::string& statistics);
    const std::string& obsname(void) const;
    GTime              tstart(void) const { return m_gti.tstart(); }  // no class member
    GTime              tstop(void) const { return  m_gti.tstop(); }   // no class member
    GEnergy            emin(void) const { return m_ebounds.emin(); }  // no class member
    GEnergy            emax(void) const { return m_ebounds.emax(); }  // no class member
    const GEbounds&    ebounds(void) const;
    const GGti&        gti(void) const;
    const GRoi*        roi(void) const;
    const GEvents*     events(void) const;
    const std::string& statistics(void) const;
```

3.3.6 Output

Output stream and logging operators should be implemented for every class as friend operators. An example is:

```
#include <iostream>
#include "GLog.hpp"
class GFits {
    friend std::ostream& operator<< (std::ostream& os, const GFits& fits);
    friend GLog&         operator<< (GLog& log, const GFits& fits);
    ...
```

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;
}
```

3.3.7 Container classes

Container classes are classes that contain list of elements.

Each container class should have element access operators `operator[]` implemented, returning either a reference or a pointer to the class elements. A non-const and a const version of the operator should exist.

Similar to the C++ template classes, `at()` methods could also be implemented that perform range checking.

4 Documentation

Code documentation should be done within the source files using Doxygen compliant annotations.

The following rules apply.

4.1 File descriptor

Put a 80 character wide header comment on top of **each file** (header, source code, templates, etc.). The header contains the file name, a brief description of the file content, the development period and the name of the person which **initially** created the file, and a standard GNU Public License statement. The header comment is immediately followed by a Doxygen compliant file description that provides the file name, a brief description of the file content, and the name of the person which **initially** created the file.

```
/***************************************************************************
 *                        GMatrix.cpp  -  matrix class                     *
 * ------------------------------------------------------------------------ *
 *  copyright (C) 2006-2010 by Jurgen Knodlseder                            *
 * ------------------------------------------------------------------------ *
 *                                                                          *
 *   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 2 of the License, or     *
 *   (at your option) any later version.                                    *
 *                                                                          *
 ***************************************************************************/
/**
 * @file GMatrix.cpp
 * @brief GVector class implementation.
 * @author J. Knodlseder
 */
```

5 Miscellaneous

5.1 Version control

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

A `major_revision` of `00` designates the development version of `GammaLib`. At this level, external interfaces of `GammaLib` may change constantly, hence no interface control system is implemented. The `minor_revision` tag will be incremented once new major features become available. The `patch` tag will be incremented after adding minor features and code corrections.

Once the development phase is finished the release phase is entred. At this moment the `major_revision` number will be incremented to `01`.

During release phase, external interfaces of `GammaLib` will be under configuration control. If modifications of existing external interfaces will be done, the `major_revision` number will be incremented. At the same time, the `libtool` version number of the `GammaLib` will also change.

The `minor_revision` number will be incremented in the release phase when extensions of the existing interfaces get implemented. Extensions should always be backward compatible, i.e. any existing software should not break due to the add of extensions.

The `patch` number will be incremented in the release phase after bug corrections or code improvements. The corresponding modifications should not change the functionality of the library.