

# The astronomia library

---

## Help to developer

**Yannick Tailliez** - `Yannickt@users.sourceforge.net`

Copyright © 2001 Yannick Tailliez

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided also that the sections entitled “Copying” and “GNU General Public License” are included exactly as in the original, and provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the above conditions for modified versions, except that this permission notice may be stated in a translation approved by the Free Software Foundation.

---

# 1 Decription

This document is a help for people how works on the Astronomia Library. The current CVS version of this document is \$Id\$.

In this document, you can find a lot of "*how-to*" to help in some task. To keep a clean distribution of the sources, please follow these rules.



## 2 Naming convention

### 2.1 General rules

Write all name of classes/function in lower case except the first letter of each word. Of course, use English for names.

### 2.2 Choose a correct name for classes

- For classes witch represents astronomical notion, use CAstro prefix.
- For classes witch represents mathematical notion, use CMath prefix.
- For utilites classes, use CUtils prefix.
- See the existing name.

The name of the class must take precises information about it. For example, use `CAstroOrbitalElementsHeliocentric` instead of `CAstroOrbit`.

About the order of word in the name, I choose a order to sort classes when files are listed in alphabetic order.

For example, see this two lists. Classes are listed in alphabetic order.

```
CAstroBarycentricOrbitaleElements
CAstroCoordonate
CAstroHeliocentricOrbitaleElements
CAstroCoordonate
CAstroOrbitalElementsBarycentric
CAstroOrbitalElementsHeliocentric
```

In the second list, classes are automatically group by "subject".

### 2.3 Members of classes

For data members use the `m_` prefix. For function use the lower/upper case rule.

### 2.4 Constants & define

For all constants and define, use upper case.



## 3 Files organisation

### 3.1 Directory structure

The libastronomia is a part of Astronomia project, so you probably find the distribution in a `./astronomia/libastronomia`. All files we need are in the `libastronomia` directory. List of subdirectory is :

- `./` File in this directory are files need by automake/autoconf and GNU Standard information files.
- `./doc/` All documentations and file need for generate them are here.
- `./include/` C++ header file must be here. It's important for people who use the librarie because there find classes and functions prototype here.
- `./src/` All sources files.
- `./VisualCPP/` Files need to work with the Microsoft Visual C++ IDE.

### 3.2 Files maintenance

#### 3.2.1 Files management

All files for the project must be listed in the `Makefile.am` of the directory which contents it. Example with `./libastronomia/include/Makefile.am` :

```
EXTRA_DIST = attic
lib_LTLIBRARIES = libastronomia.la
libastronomia_la_SOURCES = \
  CAstroCoordonate.h \
  CAstroCoordonateRectangular.h \
  CAstroCoordonateSpheric.h \
  ...
```

First line indicate list of files or directories to keep as it is. Third and following line(s) indicates sources files.

**All files does not appear in `Makefile.am` and which not automake and autoconf system file should be remove during the use of make command. So when you add, remove or change name of a file don't forget to modify the `Makefile.am`.**

After modify a `Makefile.am`, you must update the autoconf/automake scripts. To do that, execute the following commands in the root directory of the project :

```
$ automake -a
$ ./configure
```

For more informations, see the info page about automake/autoconf :

```
$ info autoconf
```

#### 3.2.2 Files generation and cleaning

All operation such as binaries generation, cleaning of the distribution ... are make by a makefile script automatically generate for you by `./configure` (itself make by automake/autoconf).

Some possibilities of the makefile script :

- `$ make clean` : Remove all no need files.
- `$ make` : Build/rebuild binaries and documentation

- `$ make doc` : Build/rebuild documentation
- `$ make distclean` : Remove all no need files and make a `.tar.gz` archive with them.
- `$ make install` : Put the library in the corect directory of you system and register it.
- `$ make uninstall` : Remove the library of you system.

A message like "*No target for make*" means the makefile script don't exist. Generate it with  
`$ ./configure`

### 3.2.3 Verify the library content

In order to verify if a class is already in the library you can use the `nm` command. This does show a list of all function available in a library.

```
$ nm /usr/local/lib/libastronomia.so
```

## 4 Writing code

### 4.1 Syntax convention

#### 4.1.1 Indentation

The source must be clearly indented to be readable. The indentation length in the source of the project is 4 spaces.

#### 4.1.2 Braces

The position of braces must be the same in all files. Use the following style

```
void CAstroFoo::foo(int a) {
    // code
}
```

instead of this

```
void CAstroFoo::fon(int i)
{
    // code
}
```

Why ? Because in the first style, when you follow the column of the closed brace, you find directly the opening instruction. A other reason is to no confuse with this :

```
int main (int argc, char * argv[]) {
    int a=7;
    {
        int b=0;
        for(int i=0;i<a;i++) {
            b+=i;
        }
        printf("Result : %d",b);
    }
}
```

If only one instruction must be into braces, C syntax permit to don't use braces. Please, forget this rule and always put it.

## 4.2 Symbols

### 4.2.1 Macro

#### 4.2.1.1 For preprocessor

In all header file, a macro must be define with the same name than the file, like this : If the file name is `CAstroPlanet.h` the macro name is `CASTROPLANET_H`. Use this to prevent multiload of header by the pre-processor.

#### 4.2.1.2

For the multi-platform compatibility, some macro is use. By default, platform is GNU/Linux. If Microsoft Windows platform is the target, you must define the MSW macro.

## 4.2.2 Constants

Some constant are define in the `AstroConstant.h` header file. You can find, for example :

- `PI=3.1415 ...`, `TWOPI=6.2831 ...`, `PI2=1.5707 ...`;
- `SQRT2 ...` for calculation;
- `DEGTORAD`, `SECTORAD ...` to convert angle values;

For more informations, please refer to `./include/AstroConstant.h` file.

## 4.3 Compilation

With GNU/Linux system, use `$ make` to compil the project. With Microsoft Visual C++, use the standard procedure. With Microsoft Windows and Cygnus shell, follow the GNU/Linux instructions, but don't forget to define `MSW` macro.

## 4.4 Documentation comments

To always have a up to date documentation about functions, the project use documentation comments like Java with JavaDoc. A documentation comments start with `/**` and end with `*/`. This is a example of the syntax you must use :

```

/*****
 * The CAstroFoo class is ...
 *
 */
class CAstroFoo {
    /*****
     * This function is wonderfull.
     *
     * Reference : Astronomical wonderfull function, , chap 5, p301
     *
     * @exception CAstroExceptionOutOfBound The coef is out of limits.
     *
     * @param coef A great coefficient.
     * @param time Time of the event.
     *
     * @return Return a wonderfull number.
     */
    public void CAstroFoo::Foo(double coef, CAstroTime time) {
        double result=0.0;
        ...
        return result;
    }

    /** Distance between the two body in meters */
    private long m_Distance;
}

```

The asterisk ligne at the begining help to find where a function start.

## 5 Versioning of the library

In fact, it exists three independent versions numbers for the library :

- The CVS version number, which is used only by developer. It does represent the history of the source modification.
- The 'interface' version number, which tells the compatibility between different versions of the library.
- The release version number, which is the 'official' version of the library.

The two more important versions numbers are interface and release number.

### 5.1 The interface version number

It allows the system to know if a version library is compatible with a program developed with an older version. The syntax is : *CURRENT:REVISION:AGE*.

- *CURRENT* is the most recent interface number of the library. Increment this if interfaces have been changed.
- *REVISION* is the implementation number of the current interface. Increment when source code has changed and set to zero if current is incremented.
- *AGE* is the difference between current interface and current number of the older interface supported.

This version number must be updated by the maintainer.

For more complete information, see See [\[libtool\]](#), page [\[undefined\]](#).

### 5.2 The release version number

It is the 'classic' version number.

### 5.3 Where I can find version numbers ?

The version numbers can be found at the beginning of `configure.in`.

```
RELEASE_VERSION=0.0.3  
LIBRARY_VERSION=1:0:0
```



## 6 Link to the library

### 6.1 Exemple of program

```
#include<stdio.h>
#include "CAstroTime.h"

int main(int argc, char * argv[]) {
    CAstroTime instant;
    instant.SetJulianDay(564654.2);
    printf("Dynamical time : %f\n",instant.GetDynamicalTime());
    return 0;
}
```

### 6.2 Example of compilation script :

```
gcc -g -c test.cpp -o test.o
gcc -g -o test test.o -lm -lstdc++ -L/usr/local/bin -lastronomia
```

- -lm link with math library.
- -lstdc++ link with the standard c++ library.
- -lastronomia link with the astronomia library.
- -Lpath indicate the path of the libraries.

### 6.3 Execute the program :

```
LD_LIBRARY_PATH="/usr/local/lib" ./test
```



## 7 Add, change name, remove classes

### 7.1 Add a new class

#### 7.1.1 Create new files

Choose a correct name for the class. See Chapter 2 [Naming convention], page 3, for more info.

- Make the NewClass.cpp in src directory and the new NewClass.h in include directory.
- Under Visual C++, choose Insert>New class, and don't forget to change the location in the dialog box.

#### 7.1.2 Integration in the project

##### 7.1.2.1 In NewClass.cpp file

Add the following code at the beginning

```
#include "config.h"
#ifdef MSW
    #define DLL_EXPORT_NEREIDE __declspec(dllexport)
#else
    #define DLL_EXPORT_NEREIDE __declspec(dllimport)
#endif
#include "NewClass.cpp"
```

##### 7.1.2.2 In NewClass.h file

Add the following code at the beginning

```
#if !defined(NEWCLASS_H)
#define NEWCLASS_H
#ifdef MSW
    #ifndef DLL_EXPORT_NEREIDE
        #define DLL_EXPORT_NEREIDE __declspec(dllexport)
    #endif
#else
    #define DLL_EXPORT_NEREIDE __declspec(dllimport)
#endif
#endif
```

```
#include "...h" // Add all required header files here
```

Add at the end of header file

```
#endif
```

#### 7.1.3 Update the compilation files

See Chapter 3 [Files maintenance], page 5.

- Add the source file to the list in the Makefile.am of src directory
- Add the header file to the list in the Makefile.am of include directory
- In the root directory of the project, execute the following commands
 

```
$ automake -a
$ ./configure
```

## 7.2 Modify the name of a class

### 7.2.1 Class name

Choose a correct name for the class. See Chapter 2 [Naming convention], page 3, for more info.

- Change the name of the source file in src directory and the header file in include directory.
- Under Visual C++, goto files list, remove the old files by pressing `(DEL)` and add the new with Project>Add files dialog box.

### 7.2.2 Integration in the project

#### 7.2.2.1 In NewClass.h file

Modify the define at the beginnig of the file.

```
#if !defined(NEWCLASS_H)
#define NEWCLASS_H
```

### 7.2.3 Update the compilation files

See Chapter 3 [Files maintenance], page 5.

- Modify the source file name in the list in the Makefile.am of src directory
- Modify the header file name in the list in the Makefile.am of include directory
- In the root directory of the project, execute the following commands

```
$ automake -a
$ ./configure
```

## 7.3 Remove a class

### 7.3.1 Remove file

- Remove the source file and the header file. You should put them in `attic` directory.
- Under Visual C++, goto files list, remove the old files by pressing `(DEL)` and add the new with Project>Add files dialog box.

### 7.3.2 Update the compilation files

See Chapter 3 [Files maintenance], page 5.

- Remove the source file in the list in the Makefile.am of src directory
- Remove the header file in the list in the Makefile.am of include directory
- In the root directory of the project, execute the following commands

```
$ automake -a
$ ./configure
```

# Table of Contents

<b>1</b>	<b>Description .....</b>	<b>1</b>
<b>2</b>	<b>Naming convention .....</b>	<b>3</b>
	2.1 General rules .....	3
	2.2 Choose a correct name for classes .....	3
	2.3 Members of classes .....	3
	2.4 Constants & define .....	3
<b>3</b>	<b>Files organisation .....</b>	<b>5</b>
	3.1 Directory structure .....	5
	3.2 Files maintenance .....	5
	3.2.1 Files management .....	5
	3.2.2 Files generation and cleaning .....	5
	3.2.3 Verify the library content .....	6
<b>4</b>	<b>Writing code .....</b>	<b>7</b>
	4.1 Syntax convention .....	7
	4.1.1 Indentation .....	7
	4.1.2 Braces .....	7
	4.2 Symbols .....	7
	4.2.1 Macro .....	7
	4.2.1.1 For preprocessor .....	7
	4.2.1.2 .....	7
	4.2.2 Constants .....	8
	4.3 Compilation .....	8
	4.4 Documentation comments .....	8
<b>5</b>	<b>Versioning of the library .....</b>	<b>9</b>
	5.1 The interface version number .....	9
	5.2 The release version number .....	9
	5.3 Where I can found version numbers ? .....	9
<b>6</b>	<b>Link to the library .....</b>	<b>11</b>
	6.1 Exemple of program .....	11
	6.2 Example of compilation script : .....	11
	6.3 Execute the program : .....	11

<b>7</b>	<b>Add, change name, remove classes . . . . .</b>	<b>13</b>
7.1	Add a new class . . . . .	13
7.1.1	Create new files . . . . .	13
7.1.2	Integration in the project . . . . .	13
7.1.2.1	In NewClass.cpp file . . . . .	13
7.1.2.2	In NewClass.h file . . . . .	13
7.1.3	Update the compilation files . . . . .	13
7.2	Modify the name of a class . . . . .	14
7.2.1	Class name . . . . .	14
7.2.2	Integration in the project . . . . .	14
7.2.2.1	In NewClass.h file . . . . .	14
7.2.3	Update the compilation files . . . . .	14
7.3	Remove a class . . . . .	14
7.3.1	Remove file . . . . .	14
7.3.2	Update the compilation files . . . . .	14

# Index

-		
._SOURCES .....	5	
<b>A</b>		
Add a new class .....	13	
autoconf .....	5	
automake .....	5	
<b>B</b>		
Braces .....	7	
<b>C</b>		
Change a class name .....	13	
Class, add new .....	13	
Class, change name .....	13	
Class, remove .....	14	
Comments .....	8	
Compilation .....	8	
configure .....	5	
Constants (naming) .....	3	
Contants .....	7	
<b>D</b>		
Define (naming) .....	3	
Description .....	1	
Directory structure .....	5	
Distribution .....	5	
Documentation .....	8	
<b>E</b>		
EXTRA_DIST .....	5	
<b>F</b>		
Files maintenace .....	5	
Files management .....	5	
Files organisation .....	5	
<b>I</b>		
Indentation .....	7	
<b>L</b>		
Link .....	11	
<b>M</b>		
Macro .....	7	
make .....	5	
Makefile.am .....	5	
<b>N</b>		
Naming convention .....	3	
<b>R</b>		
Remove a class .....	14	
<b>S</b>		
Syntax convention .....	7	
<b>V</b>		
Versoning .....	9	

