Version n° 1.0a

LaurTec

Artorius Humanoid Robot

Programming Practices

Author : *Mauro Laurenti*

Contributors:

email: info.laurtec@gmail.com

ID: AR-DC0003-EN

License

Additional Terms

As stated in section 4 of the license, these additional terms are included and must be preserved:

1) Disclaimer of Warranty

THERE IS NO WARRANTY FOR THE HARDWARE AND/OR SOFTWARE HEREWITH DESCRIBED, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE HARDWARE AND/OR SOFTWARE "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE HARDWARE AND/OR SOFTWARE IS WITH YOU. SHOULD THE THE HARDWARE AND/OR SOFTWARE PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

2) Limitation of Liability

IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MODIFIES AND/OR CONVEYS THE HARDWARE AND/OR SOFTWARE AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE HARDWARE AND/OR SOFTWARE (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE HARDWARE AND/OR SOFTWARE TO OPERATE WITH ANY OTHER HARDWARES AND/OR SOFTWARES), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

3) Interpretation of Sections 1 and 2

If the disclaimer of warranty and limitation of liability provided above cannot be given local legal effect according to their terms, reviewing courts shall apply local law that most closely approximates an absolute waiver of all civil liability in connection with the Hardware and/or Software, unless a warranty or assumption of liability accompanies a copy of the Hardware and/or Software in return for a fee.

All other trademarks belong to their respective owners

Table of Contents

Abstract

The times when the programmer was making all alone at home is over. High level programming languages let the programmer make programs that would have been impossible to be written in assembly, line by line. The difference between the assembly and the high level programming language is the same one among single programmer and a programming Team. A Team is not just a group of people working on the same project, a Team is group of people willing to make a project happen, communicating and collaborating with each other.

- Communication is an easy word spelled every day but just speaking, does not make it. Communication requires understanding, respect among team mates.

- Collaboration, is as hard as Communication. Collaboration requires understanding, respect among team mates.

Communication and Collaboration make out of people a Team. To let the Team properly program together as one, some basic rules should be followed. These rules are what are known as Programming Practices and Coding Styles. The rules let you read a program without recognizing the programmer and letting you get used to read code of others. Your code will be read by others more than the times you read it and you will read code of others more than yours.

Rules are made to be broken, Programming Practices and Code Styles are just guidelines but the are thought to make the code more readable and maintainable. Any exception in following these guidelines to make the code more readable can be accepted and must be done. By the way don't following a rule just because you do not like it does not respect the others, so you must follow it.

General Rules

The general rules are the ones that should not be broken.

Rule 1：Readability

Rule of the rule is readability. Everything herewith described is made to achieve high readability. If the code you are writing is having an exception in which don't following any of the rules below makes it more readable, don't follow any rule except Rule 1. Don't make of Rule 1 an excuse for not following the other rules on regular bases. Discuss with your team if you have a feeling that a rule should be added or modified.

Rule 2：Write text in English

All the names and comments must be written in English. No other languages must be used!

Example 1:

```
//Sono Italiano ma e' meglio usare l'inglese!
//Soy italiano, pero es mejor utilizar el Inglés!
//...wenn schon, denn schon!
//It's better if we use one common language!
```

...could you read all?!

Rule 3：Optimizations

Never optimize the code before you must do it. Make your code easy to read and to understand, you will always have time to optimize it. The 80-20 rule is a statistical study that has shown that 80% of the resources (time and memory) are used by 20% of the code. So don't optimize until you need it.
Once you need, search for the 20% of the code that are using the 80% of the resources. Always document the resource benefit you get once you have optimized the code. If the optimization is not worth the benefit, keep the code as it is and continue searching the 20%...
Don't optimize randomly without documenting the benefit. This practice will make your code less readable.

Rule 4：Don't write smart code write the simplest one that works

Smart code that exploit special feature or behavior can make the code hard to read. Write the code in simple and easy way. You will always have time to optimize the code.

Rule 5：License

Each source code and header file must have the project license.

Code Comments

Writing a readable code should be one of the most important rule to follow. Write comments is one way to achieve this. By the way well written code can have reduced amount of comments because good codes should be self-explainable. In the following bunch of rules the comments within the header file and the source file are discussed separately. Furthermore since there are tools that can extract comments out of the program, the comments approach herewith described, will refer to Doxigen tool as documentation extractor. Among the commands, that Doxygen is able to parse, the ones that are compatible with Javadoc should be used. To keep the comments clear you should avoid to use too many Doxygen features, but reading the documentation of Doxigen and its functionalities is recommended. More commands could be used than the ones just described in the following rules but don't mess up your code.

Rule 6：Header File Comments

If the header file contains something different from a class, you should write a brief overview of what the header file is made for. If the header file contains a class, its description will be the class' description (see Rule 7).

Rule 7：Class Comments within the Header file

Each class should be properly commented. The comment should go before the class itself. This will let Doxygen automatically understand that the comment is made for the class.

> **Example 1:**

```
/**
 *  This is the brief description of the class.
 *  The second line is part of the more detailed description
 */

class MotorBoard {
...
};
```

Note:

- As you can see, a comment that must be extracted is started with /** instead of /*.
- The first sentence is extracted by Doxygen as brief description, while the rest of the text is considered a detailed description.

Rule 8：Method Comments within the header file

Each method should be properly commented. The comment should go before the method itself. This will let Doxigen automatically understand that the comment is made for the method. The parameters and return value should be described. Each description should contain the range in with the described parameter is valid.

Example 1:

```
/**
 *  This is the brief description of the method.
 *  The second line is part of the more detailed description.
 *
 *  @param  portNumber Number of the port to open. Range [1..10]
 *  @return 0 if the port has be opened -1 in case of error
 */

int openPort (int portNumber);
```

Note:

- Each command that is parsed by Doxygen must start with @
- The parameter description require the command @param
- The return value description require the command @return
- To describe a thrown object use the command @throw <exception-object> description

Rule 9： Code comment within the Header file

You are encouraged in *not* writing code within the header file.

Rule 10： Variables comments within the Header file

All the variables must be commented as the method parameters. In addition the initialization value should be written. The command to describe the variables is @var. Note that the comments should precede the variable, this will let you write the comment tidy.

Example 1:

```
/**  @var  portNumber Number of the port to open.
 *         Range [1..10]
 *         Initialized to 1
 */

int portNumber = 1;
```

Rule 11： Source File Comments

No comments are required on top of the file, since the description is written already in the header file. Avoid to duplicate the documentation, this could create maintainability problem.

Rule 12： Class Comments within the source file

No comments are required before the class definition, since the description is written in the header file where the class is declared. Avoid to duplicate the documentation, this could create maintainability problem.

Rule 13：Method Comments within the source file

No comments are required, since the description is written in the header file. Avoid to duplicate the documentation, this could create maintainability problem. Nevertheless it is recommended to separate each method as shown in the following Example.

Example 1:

```
//**********************************************************
//                Costructor Implementation
//**********************************************************
DataStream::DataStream () {

  openPort ();
}


//**********************************************************
//                    getBaudRate
//**********************************************************
long int DataStream::getBaudRate () {

  return (port.baudRate);
}
```

Note:
  · The methods are not even commented since the name of the method is like a comment by itself.

Rule 14：Code comment within the source file

Standard code must be properly commented to let the reader understand what the code is supposed to do. The comment should be indented as the source code. This will clearly associate the comment to the interested code.

Rule 15：Variables comments within the source file

All the variables must be commented as the method parameters. In addition the initialization value should be written. The command to describe the variables is @var. Note that the comments should precede the variable, this will let you write the comment tidy.

Example 1:

```
/**  @var  portNumber Number of the port to open.
 *         Range [1..10]
 *         Initialized to 1
 */

int portNumber = 1;
```

Naming Practices

Beside writing comments good naming is vital to make the code readable.

Rule 16： **Make the names clear, avoid abbreviations**

Abbreviation could have different meaning in different minds. Using it could create misunderstanding.

> **Example 1:**
>
> you should not use:
>
> *no* instead of number
> *ln* instead of link, natural log...
> *comp* instead of compare, compute...
>
> and so on...

If abbreviation are used within a convention, they must be documented and accepted. Every one must then use it.

Rule 17： **Class naming**

The class should have a meaningful name that let the user understand what the class is doing and made for. The class name should start with upper case and each following word in the name must be capitalized as well.

> **Example 1:**
>
> ```
> class MotorBoard {
> ...
> };
> ```
>
> **Example 2:**
>
> ```
> class WordWord {
> ...
> };
> ```

Rule 18： **Methods naming**

Each method within a class should have a meaningful name that should let the user understand what the method is made for. The method name should start with lower case and each following word must be capitalized. The object name should be avoided within the name of the method itself (see Example 2).

> **Example 1:**

```
int getID (void);
void  wordWord (void);
```

Methods that retrieve data should have the `get` prefix, while method to set data should have the `set` prefix. The prefix `is` could be used for boolean variables.

Example 2:

```
//The method is well named
determinant = Matrix.getDeterminant ();

//The method is NOT well named since determinant is repeated
determinant = Matrix.getMatrixDeterminant ();
```

Rule 19： Namespace naming

Namespaces should be named using lower case.

Rule 20： File naming

All the file names, either source code or header file, should have the name of the class that is defined or implemented in it. Each file should therefore contain either one class definition or implementation. In case of nested classes, the file will get the name of the main class.

Rule 21： Variables naming

The variable name, as any other naming rule, should have a meaningful name. Short name should be avoided. Make the name clear. The name should start lower case and the inner words should be capitalized.

Example 1:

```
string personName;      //good name
string psName;     // bad name
string nm;         //Too short. Do I mean name, number...!

int index;       //it could be fine but it is too generic
int motorIndex;   //it could be a better one
```

Example 2:

```
int x;     // row
int y;     // column

for (x; x<10; x++)
  for (y; y<10; y++)
     matrix[x,y]= x * y;
```

this could be better written:

```cpp
int row;          // Matrix row
int column;       // Matrix column

for (row; row =< MAX_ROW; row++)
  for (column; column =< MAX_COLUMN; column++)
      matrix[row,column]= row * column;
```

Having changed the x,y variables with row and column, make the code more readable. The magic number 10 has been changed with a constant. Some time if the variable scope is small, name like i,x,y, could be accepted, but within nested **for** , variables should have good names.

Rule 22： Constants naming

Use of constant make the code easier to change and avoid the use of magic numbers within equations. The constant naming should be meaningful. The constant name is all upper case and each word is separated by an underscore.

   Example 1:

   MAX_MOTOR_SPEED

Rule 23： Variables should never have multiple meanings

Avoid side effect and confusion due to variables with different meanings in different scopes.

Rule 24： File extensions

C++ allows different files extensions, by the way the following are used within the project:

•   .h: for the header file
•   .cpp: for the source code

Don't forget it

Rule 25： Avoid magic numbers

To simplify code changes and improve readability and maintainability, you should avoid magic numbers. If you use magic numbers and you must refactor the code, you could have trouble searching the magic numbers to change, and you could change the wrong one. So, don't' use magic numbers.

Example 1:

```
for (int speed; speed<100; speed++) //bad code with magic number

for (int speed; speed<MAX_MOTOR_SPEED; speed++) //good code
```

Example 2:

If you need to run an infinite loop, avoid this structure:

```
while (1) {

//do your stuff

}
```

prefer this one instead:

```
while (true) {

//do your stuff

}
```

Rule 26： Avoid #define, use const

C++ is offering new opportunity to get read of preprocessor directives. One of this, is the well known #define, generally used to define constants and macros. The directive #define has no type check so can lead to malicious and hidden bugs. For that reason to define a constant use the keyword const. The keyword const allows type check at compile time.

Example 1:

```
static const int MAX_SPEED = 100;
```

Rule 27： Avoid #define, use inline functions

If you want to use #define directive to create a macro, avoid doing this and think twice. Macros can cause subtle bugs hard to find. Use inline functions instead. You will get the speed of a macro and the benefit of a function behavior.

Rule 28： Avoid enum, use const

Enforce type check using const istead of enum structure.

Rule 29： Class definition

All the class definitions must be defined within the source file and not the header file.

Example 1:

```cpp
class MyClass {

public:

  MyClass ();
  ~MyClass ();
  getPortID ( return portID);        //you should NOT implement the method

};
```

Rule 30： Multiple header file inclusion

To allow multiple header file inclusions, all the header file definitions should be contained within the following code:

```cpp
#ifndef MY_CLASS_NAME_FLAG
#define MY_CLASS_NAME_FLAG

//write your header file here

#endif
```

Example 1:

For instance if you are creating the definition for the class PortableObjects, you should do the following:

```cpp
#ifndef PORTABLE_OBJECTS_FLAG
#define PORTABLE_OBJECTS_FLAG

class PortableObjects {

//define your methods here

};

#endif
```

Rule 31： File inclusion must be sorted and grouped

Header files must be included on top of the file. Standard header files should be included first. These should be grouped leaving an empty line among the others header files.

Example 1:

```
#include <iostream>
#include <string>
#include <fstream>

#include "ClientCon.h"
#include "ServerCon.h"
```

Rule 32: Make variables scope as small as possible

C++ allows variable deceleration at any point of the source code. This feature can be exploited to reduce the variable scope, making the source code more readable. In fact, you do not have to go on top of the file to understand what the variable is and made for. Furthermore reducing the scope will release the resources once you are out of the scope, instead of keeping it for all the application life.

Example 1:

Whiting the `for` you can define the variable:

```
for (int row; row < MAX_ROW; row++){

}
```

Rule 33: Avoid using global variables

Avoid to define global variables. Make the scope as small as possible.

Rule 34: Destructor must clean up all the allocated resources

Any time you allocate resources, such as using new operator, or opening a port, a socket, a file, and so on, you must release it. The place for doing it is the destructor. Avoid to release the resources within the code itself. If an exception change the code flow, you may never release the resources.

Rule 35: Destructor must not throw exceptions

Destructor is responsible to release the resources previously allocated. If Destructor fails to release the resources you will have memory leak or worst you will get the ticket to the software undefined realm!

For that reason Destructor must always complete its cleanup goal. If it fails, it must terminate the application but should not throw an exception.

An other approach to keep the application running is to swallow the exception/error that could get generated.

If the destructor call a close method that could throw an exception is good practice to make that

method public instead of private. This way the user can call it and handle the exception if he worries about it. Otherwise the destructor will just ignore it.

Example 1:

In this example the application is terminated due to the exception . Note that closePort method is private, so the class user can not access it.

```cpp
class MyClass {

private:

   closPort () throw (ClosePortError);

public:

   MyClass ();
   ~MyClass ();
   openPort ();

};


MyClass::~MyClass (){

try {
   // Release the resources allocated for the Port
   closPort ();

} catch (ClosePortError) {

   // The resources can not be released, so the application is closed
   std::abort ()
   }
}
```

Example 2:

In this example the application continues running. Destructor just ignore the exception. Note that closePort method is now public. The class user can try to close the port by himself and handle the exception. The destructor will close the port anyway if the user won't do it.

```cpp
class MyClass {

private:

public:

   MyClass ();
   ~MyClass ();
   openPort ();
   closPort () throw (ClosePortError);

};


MyClass::~MyClass (){

try {
   // Release the resources allocated for the Port
```

```
    closPort ();

  } catch (ClosePortError) {

    // The resources can not be released, I ignore it.
    }
  }
```

Rule 36： Base class Destructor must be virtual

To avoid slicing effect due to wrong destructor call, make all the destructor within a base class virtual. This will avoid the visit into the software undefined realm!

Example 1:

```
class BaseClass {

public:

  BaseClass ();
  virtual ~BaseClass ();

//write your methods

};


class DerivedClass: public BaseClass {

public:

  DerivedClass ();
  ~DerivedClass ();

//write your methods

};
```

Rule 37： Public versus Private

Encapsulation is a strong advantage whiting the object oriented programming philosophy, use of it should be enforced everywhere. All the data members should be declared private. Access to it should be given just with public methods such as *get* and *set*. This will hide the implementation, making maintenance easier. Declare private any method that the final user should not use, if you are not sure what to apply, declare it private. You will always have time to make it public without side effect.

Coding Layout

Rule 38: Use the TAB consciously

TABs can be helpful for making the code more readable, by the way you should avoid using it too much, because:

- you can make maintainability hard. People will escape from updating structures which are too complex.

    Example 1:

    ```
    /***********************
    *      This is good      *
    *         looking         *
    *                          *
    ***********************/
    ```

    This is a typical structure that has apparently a good looking...but try to add a text inside, and you will spend few minutes to get it back! Avoid using TAB to create this structures.

- TABs space is a parameter that can be changed within the editor or IDE. This means that if you open a file where the tab has been used and the editor has the TAB parameter defined with a different number of spaces, you will get everything mixed up!

    Example 2:

    This is what you could get with a different editor

    ```
    /***********************
    *            This is good                 *
    *       looking              *
    *                              *
    ***********************/
    ```

    Is it still nice!

Use TAB consciously!

Rule 39: Use the spaces consciously

Any compiler can read everything that is syntactically correct, without caring about spacing.

Example 1:

```
for(int index=0;index<NUM_MAX;index++){

   //do your stuff
}
```

surely you can read the code, but which one can you read better?

```
for (int index = 0; index < NUM_MAX; index++){

  //do your stuff
}
```

Put a space where you can enhance readability. One is enough, don't make abuse of it.

Rule 40: Use empty lines consciously

Any compiler can read everything that is syntactically correct, without caring about spacing. Empty lines are very confusing if not properly used.

Example 1:

```
int i=0;int b=5;for(int index
=0;index<NUM_MAX;index++
){b++;i++}
```

Compiler will not complain about this bunch of code, but would it be better this way?

```
int i=0;
int b=5;

for(int index =0;index < NUM_MAX; index++){

  b++;
  i++;
}
```

These rules should be followed:

- Don't use more than 3 empty lines to isolate the code. If you really think that the code you are writing is so different that you need to isolate it more than three empty lines, think about writing it in a different file!
- Use 1 empty line for isolating standard code such as the one in Example 1.
- Use 2 empty lines if the code you are writing is wort to be highlighted. For instance two methods declarations, public and private methods, include files and class definitions/implementations and so on.

Rule 41: Limit the file contents to 80 columns

IDE tools enhance code readability in many ways, by the way you should not rely on it for making the code more readable. In fact the code could get printed or can be opened with different IDE or standard editor.

One way to ensure that different IDE, editors, monitors and printers can show the code has you have initially written it, is to limit the file contents to 80 columns. IDE such as Eclipse has as editor option, the ability to show the printable margin directly on the monitor. Showing the margin will remember you to keep the contents within a printable limits.

Rule 42: Type conversion must be done explicitly

All the files must be compiled with the maximum level of warning (verbose option should be also be enabled if available). Never discard warnings that advice you about implicit conversion. Understand the implicit conversion and make it explicit using casting.

Rule 43: Use C++ casting instead of C casting

Prefer C++ casting options instead of the cast option offered by C. C++ casting make more clear which kind of casting you are using and what it is for.

Rule 44: Type Pointers and references should have their reference next to the type

Example 1:

```
char* speed;       //instead of  char * speed;  or  char *speed

char& speed;       //instead of  char & speed;  or  char &speed
```

Rule 45: Avoid implicit test

Make the code readable. If you can read the code you can limit the comments!

Example 1:

```
//bad one, it is implicit
if (turnON) {

}

//better one
if (turnON == 1) {

}

//even better, you read it as a sentence!
if (turnON == true) {

}
```

Rule 46: Don't make too much stuff with for (...)

Use `for ()` for loop, don't use it with too many statements!

Example 1:

```
//bad one
```

```
for (int index=0, int fontNumber=100; index<MAX_VALUE; index++,
fontNumber--) {

}

//good one
for (int index = 0; i < MAX_VALUE; index++) {

}
```

Rule 47: Avoid using do...while

Everything that is done by do...while can be done with a while. Normally do-while is less readable then a standard loop.

Rule 48: The while (true) should be used for infinite loops

Make clear that the loop is endless. Use the keyword true instead of 1.

Example 1:

```
//good one
while (true) {

}

//bad one
while (1) {

}

//bad one
if(;;) {

}
```

Rule 49: Avoid using goto

In the past goto was a good keyword. Nowadays programming practice has reduced its usage since readability and maintainability are must. Huge programs with code flow changes due to goto are difficult to read and maintain. Think twice before using it!
...if you end up on using it, think a little bit more! Rare are he cases where you really need goto!

Rule 50: Limit the use of break and continue

Code is more readable if code is not interrupted and the flow goes forward making little steps.

Rule 51: Make the return values always explicit

Methods and functions should always have the return value explicitly written even if void.

Rule 52： Class layout

This brackets layout and declaration order should be used:

Example 1:

```cpp
class MyClass {

public:

...

protected:

...

private:

...

};
```

public comes before the others because the user needs those, while private members can be ignored.

Rule 53： Methods brackets layout

This brackets layout should be used:

Example 1:

```cpp
int MyClass::MyMethod (int value) {

   //do whatever you want

}
```

Rule 54： if brackets layout

This brackets layout should be used:

Example 1:

```cpp
if (name=="Me") {

   //do whatever you want
}
```

instead of:

```
if (name=="Me")
{
   //do whatever you want
}
```

Rule 55： if-else layout

The following layout should be used:

Example 1:

```
if ([condition]) {

   [statement];
}
```

this layout is preferable even if the statement is one line, since avoids trouble if other lines will be added later.

```
if ([condition])
   [statement_1];
   [statement_2];
```

from the indentation it seams that the statment_2 is executed if the condition is verified, but this is not true, just statement_1 is executed. Brackets can limit this problems.

Example 2:

```
if ([condition]) {

   [statement];
}
else {

   [statement];
}
```

Rule 56： while brackets layout

This brackets layout should be used:

Example 1:

```
while (true) {

   //do whatever you want

}
```

instead of:

```
while (true)
{
   //do whatever you want
```

```
}
```

Rule 57： **for brackets layout**

This brackets layout should be used:

Example 1:

```cpp
for (int index; index < MAX_VALUE; index++) {

   //do whatever you want

}
```

Rule 58： **switch layout**

The following layout should be followed:

Example 1:

```cpp
switch (condition) {

   case A :
           statements;
           break;

   case B :
           statements;
           break;

   default :
           statements;
           break;
}
```

Rule 59： **try-catch layout**

The following layout should be used:

Example 1:

```cpp
try {

   //your code that could throw exceptions

} catch (Exception_1& error) {

   //exception handler 1

} catch (Exception_2& error) {

   //exception handler 2
}
```

Alphabetical Index

Bibliography

[1] www.LaurTec.com/Artorius : visit Artorius Home page for getting the latest update
[2] Code Complete (Microsoft Press, Steve McConnel)
[3] Effective C++ (Addison Wesley, Scott Meyers)
[4] More Effective C++ (Addison Wesley, Scott Meyers)
[5] Thinking in C++ (Prentice Hall, Bruce Eckel)
[6] Thinking in C++ vol.2 (Prentice Hall, Bruce Eckel, Chuck Allison)

History

Date	Version	Name	Change Description	
11.07.09	1.0	Mauro Laurenti	Original Version	
05/10/09	1.0a	Mauro Laurenti	License has been changed to GNU FDL	