ARGoS

AUTONOMOUS ROBOTS GO SWARMING

# Coding Conventions

Carlo PINCIROLI

<cpinciro@ulb.ac.be>

Version 1.0

May 10th, 2010

*There are two ways to write error-free programs. Only the third one works.*

*If at first you don't succeed, you must be a programmer.*

*Do or do not. There is no try.*

*Normally it works.*

# Contents

# 1 Introduction

This document describes the coding guidelines to follow when developing code for ARGoS.

In this last phase of the development, it is very important to follow these guidelines. The code will be realeased soon and it must provide a minimum level of clarity and quality to be usable by external people. The aim is to give the impression that the code was developed by one person only, even though it is, in fact, a team work.

Besides clarity for external audience, this coding guidelines make it simpler for everybody to know how things are structured, with obvious benefits in terms of maintainability and ease of bug solving.

# 2 Naming Conventions

In the code, we follow a custom version of the Hungarian Notation.

## 2.1 Variables

The Hungarian notation encodes the *scope* and the *type* of a variable in its name. The scope is defined by where a variable is declared — variables can be class members, function parameters or local variables. In Table 1 we report examples of variable definitions. The reported examples apply also to variable references, following the C++ interpretation stating that a reference *is* the referenced object. For example, this is a correct declaration:

```
void MyFunction(const CMyClass& c_var);
```

On the contrary, for pointers, the Hungarian Notation prepends a 'p' to the type part of the variable name, as shown in Table 2.

Table 1: Examples of variable naming.

| *Type* | *Class Member* | *Local Variable* | *Function Parameter* |
|---|---|---|---|
| *UIntXX* | m_unMyVariable | unMyVariable | un_my_variable |
| *SIntXX* | m_nMyVariable | nMyVariable | n_my_variable |
| *Real* | m_fMyVariable | fMyVariable | f_my_variable |
| *Boolean* | m_bMyVariable | bMyVariable | b_my_variable |
| *STL Vector* | m_vecMyVariable | vecMyVariable | vec_my_variable |
| *STL Map* | m_mapMyVariable | mapMyVariable | map_my_variable |
| *STL List* | m_listMyVariable | listMyVariable | list_my_variable |
| *STL Iterators* | m_itMyVariable | itMyVariable | it_my_variable |
| *String* | m_strMyVariable | strMyVariable | str_my_variable |
| *Class* | m_cMyVariable | cMyVariable | c_my_variable |
| *Struct* | m_sMyVariable | sMyVariable | s_my_variable |
| *Enum* | m_eMyVariable | eMyVariable | e_my_variable |
| *Union* | m_uMyVariable | uMyVariable | u_my_variable |
| *Typdef'd type* | m_tMyVariable | tMyVariable | t_my_variable |

Table 2: Examples of pointer variable naming.

| *Type* | *Class Member* | *Local Variable* | *Function Parameter* |
|---|---|---|---|
| *UIntXX* | m_punMyVariable | punMyVariable | pun_my_variable |
| *SIntXX* | m_pnMyVariable | pnMyVariable | pn_my_variable |
| *Real* | m_pfMyVariable | pfMyVariable | pf_my_variable |
| *Boolean* | m_pbMyVariable | pbMyVariable | pb_my_variable |
| *STL Vector* | m_pvecMyVariable | pvecMyVariable | pvec_my_variable |
| *STL Map* | m_pmapMyVariable | pmapMyVariable | pmap_my_variable |
| *STL List* | m_plistMyVariable | plistMyVariable | plist_my_variable |
| *STL Iterators* | m_pitMyVariable | pitMyVariable | pit_my_variable |
| *String* | m_pstrMyVariable | pstrMyVariable | pstr_my_variable |
| *Class* | m_pcMyVariable | pcMyVariable | pc_my_variable |
| *Struct* | m_psMyVariable | psMyVariable | ps_my_variable |
| *Enum* | m_peMyVariable | peMyVariable | pe_my_variable |
| *Union* | m_puMyVariable | puMyVariable | pu_my_variable |
| *Typdef'd type* | m_ptMyVariable | ptMyVariable | pt_my_variable |

Table 3: Examples of user defined type naming.

| *Type* | *Declaration* |
|---|---|
| *Class* | **class** CMyClass {...}; |
| *Template Class* | **template** <**typename** T> **class** CMyClass {...}; |
| *Struct* | **struct** SMyStruct {...}; |
| *Enum* | **enum** EMyEnum {...}; |
| *Union* | **union** UMyUnion {...}; |
| *Typedef'd type* | **typedef char** TMyType; |

## 2.2 Types

The Hungarian Notation applies also to user defined types. In Table 3 we report some examples.

The only exception to the stated rules are the following types[1]: *UInt8*, *UInt16*, *UInt32*, *UInt64*, *SInt8*, *SInt16*, *SInt32*, *SInt64*, *Real*. These types are wrappers of the C++ primitive types typedef'd to ensure portability. Since these types are used very often, they do not follow the convention to save characters.

The *U* or *S* at the beginning of the name stand for *unsigned* or *signed*; while the number at the end of them indicates their size in bits. The *Real* type corresponds to either the *float* or *double* primitive types, depending on the platform on which the code is compiled.

> **!** **Never** use the unwrapped C++ primitive types, such as *int*, *unsigned int*, *float* and the likes. **Always** use the wrapped types, as this ensures portability.

Furthermore, for all the classes in the *control interface*, we follow a slightly different convention—all the classes must be preceded by *CCI_* instead of just *C*, for example:

```
class CCI_MyClass {
  ...
};
```

[1] Defined in file `argos2/common/utility/datatypes/datatypes.h`.

## 2.3 Constants

Constants do no follow the Hungarian Notation. They are defined in all capitals and the words are separated by underscores. For example:

```
class CMyClass {
  public:
    static const UInt32 MY_FIRST_CONSTANT;
    static const Real MY_SECOND_CONSTANT;
};
```

## 2.4 Functions and Class Members

Functions and class members follow the same convention. Their names start with a capital letter, and words are separated by capital letters. Class get/set methods are prepended by the *Get/Set* sequence and take the name of the variable their refer to. If a get method refers to a boolean flag, it is prepended by the *Is* sequence instead. Examples:

```
class CMyClass {
  public:
    void ThisIsAMethod();
    inline UInt8 GetVar() const
    {
      return m_unVar;
    }
    inline void SetVar(UInt8 un_var)
    {
      m_unVar = un_var;
    }
    inline bool IsFlag() const
    {
      return m_bFlag;
    }
    inline bool SetFlag(bool b_flag)
    {
      m_bFlag = b_flag;
    }
  private:
    UInt8 m_unVar;
    bool m_bFlag;
};

Real Normalize(Real f_min,
               Real f_max,
               Real f_value)
{
  ...
}
```

## 2.5 Files and Directories

File names and directories are all in lower case and words are separated by underscores. Header files have extension `.h` and implementation files have extension `.cpp`. For example, `file_name.h` is right, while `fileName.h`, `File_Name.h`, `File_name.h` and `FileName.h` are wrong.

# 3 Formatting Conventions

## 3.1 Generalities

When writing code, it is important to comply with also the formatting guidelines that follow.

> **!** Indentation is done with **spaces**. Each level is **four** spaces long.

> **!** The code formatting style is **Stroustroup**. Namespaces are indented.

All files must include at the very beginning the GPL license information, typeset **exactly** as follows:

```
/*
 * 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; version 2.
 *
 * 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, write to the Free Software
 * Foundation, Inc., 59 Temple Place, Suite 330, Boston, MA  02111-1307  USA
 */
```

## 3.2 Header Files

After the licensing information, a header file must contain a macro to avoid multiple inclusion. The name of the defined macro must match the file name, but written all capital:

```
#ifndef FILE_NAME_H
#define FILE_NAME_H
```

If the file defines classes, there may be problems in inclusion. To avoid them once and for all, include the following statement after the macro:

```
namespace argos {
   class CMyClass;
}
```

and only **after** this line start including other stuff:

```
#include <argos2/common/utility/datatypes/datatypes.h>
#include <string>
```

About inclusions, there is an important thing to bear in mind:

> **!** When including ARGoS headers from a `.h` file, **always** specify the complete path and use the `<...>` syntax, as shown in the example above.

The reason for this is that ARGoS can be also installed system-wide in locations that cannot be foreseen. If you do not follow this rule, user code won't compile correctly because some headers won't be found.

After the includes, all ARGoS code should be included in the `argos` namespace:

```
namespace argos {
  class CMyClass {
    ...
  };
}
```

End the file by closing the macro declaration:

```
#endif
```

## 3.3 Implementation Files

After the licensing information, implementation files should include the needed headers. In implementation files, the rule to include headers is as follows: if a file is contained in the current directory or in a subdirectory, use the **#include** "..." syntax; otherwise use the **#include** <argos2/...> syntax.

After the includes, all code should be declared in a **namespace** argos {...} block.

> **!** **Never** use the clause **using namespace** argos.

The individual elements should be separated by a comment line containing forty *. An example to show all this:

```
#include "my_header_file.h"
#include "subdir/another_header.h"
#include <argos2/common/utility/string_utilities.h>

namespace argos {

  /****************************************/
  /****************************************/

  const UInt32 CMyClass::MY_FIRST_CONSTANT  = 56;
  const Real   CMyClass::MY_SECOND_CONSTANT = 37.472;

  /****************************************/
  /****************************************/

  void CMyClass::AMethod(const std::string& str_param)
  {
    ...
  }

  /****************************************/
  /****************************************/

  UInt32 CMyClass::AnotherMethod()
  {
    ...
  }

  /****************************************/
  /****************************************/

}
```

# 4 Coding Tips

## 4.1 Using Namespaces

> **!** **Never** employ the clause **using namespace** ....

There is a number of reasons for this to be enforced. First of all, clarity: specifying std::string is more informative than just saying string — it says where the definition comes from.

Furthermore, it makes name clashes less probable. For instance, it happens often that libraries wrap the primitive types for portability and often the used names are colliding. If the programmers of the library have been smart enough to define their own namespaces, there would no problem using identical symbols. Let us see the problem with an example:

```
#include <library1.h> // defines a type uint8
#include <library2.h> // defines a type uint8 too

using namespace lib1;
using namespace lib2;

...

void MyFunction()
{
  uint8 unVar = 0; // which uint8 are we using here? -> ERROR!
}
```

With the above stated rule, the ambiguity disappears:

```
#include <library1.h> // defines a type uint8
#include <library2.h> // defines a type uint8 too

...

void MyFunction()
{
  lib1::uint8 unVar1 = 0; // no ambiguity now -> OK!
  lib2::uint8 unVar2 = 0; // no ambiguity now -> OK!
}
```

## 4.2 Copy, Reference or Pointer?

The choice among copy, reference or pointer depends on a few factors.

### 4.2.1 Parameter Passing and Returning

Let us consider the case of parameter passing. The main choice factor in this case is optimization. Strings, vectors, classes and structs are usually large constructs, and passing them by copy can be very time consuming. Thus, for them, passing by copy should be avoided.

> **!** Pass and return by copy **only** the primitive types: *UInt8*, *UInt16*, *UInt32*, *UInt64*, *SInt8*, *SInt16*, *SInt32*, *SInt64*, *Real* and *bool*.

If not by copy, should the choice be pointers or references? The differences between the two are that pointers can have the *NULL* value, whereas references cannot, and pointers are usually more likely to confuse people due to the necessity to dereferenciate them to obtain their value (the * operator). Therefore, unless you know what you are doing, follow this rule:

> **!** For structured types, **always** pass and return by reference.

An example:

```
class CMyClass { ... };

class CAnotherClass {
  public:
    // Primitive type -> return by copy
    inline UInt64 GetCounter() const
    {
      return m_unCounter;
    }
    // Primitive type -> pass by copy
    inline void SetCounter(UInt64 un_counter)
    {
      m_unCounter = un_counter;
    }
    // Complex type -> return by const reference
    inline const std::string& GetId() const
    {
      return m_strId;
    }
    // Complex type -> pass by const reference
    inline void SetId(const std::string& str_id)
    {
      m_strId = str_id;
    }
    // Complex type -> return by const reference
    inline const CMyClass& GetMyClass() const
    {
      return *m_pcMyClass;
    }
    // Complex type -> pass by const reference
    inline void SetMyClass(const CMyClass& c_my_class)
    {
      m_pcMyClass = &c_my_class; // this works because
                                 // a reference to a class
                                 // _is_ the original class!
    }
  private:
    UInt64 m_unCounter;
    std::string m_strId;
    CMyClass* m_pcMyClass;
};
```

### 4.2.2 Storing

Let's say you have a class like the above example and you have to choose among storing a variable ("by copy") or referencing it (reference or pointer). What do you choose?

The main question to ask yourself is whether the variable references to something that is integral part of your class or not. If the answer is 'yes', then prefer

storing by copy. See, in the example above, the variables m_unCounter and m_strId. With primitive types this is easy. Unfortunately, with structured types, the thing gets a little more complex. Let's say you have a class *C* and a class *D* that derives from it. If you store by copy with *C*, you lose polymorphism. Therefore, the choice should fall on a reference or a pointer. Which of the two?

Furthermore, if the answer to the previous question was 'no', the choice falls again on references or pointers, but which one?

The discriminant here is that references cannot be *NULL*. If you declare an attribute to be a reference to something, you must have that something when the class is created. To be clearer, see this example:

```
class CMyClass { ... };

class CAnotherClass {
  public:
    // Bad constructor: the references is not initialized -> ERROR
    CAnotherClass()
    {
    }
    // Good constructor: the references is initialized -> OK
    CAnotherClass(CMyClass& c_my_class) :
      m_cMyClass(c_my_class)
    {
    }
  private:
    CMyClass& m_cMyClass;
};
```

With a pointer you don't have this problem:

```
class CMyClass { ... };

class CAnotherClass {
  public:
    // Good constructor: the pointer is initialized to NULL
    CAnotherClass() :
      m_pcMyClass(NULL)
    {
    }
    // Good constructor: the pointer is initialized to something
    CAnotherClass(const CMyClass& c_my_class) :
      m_pcMyClass(&c_my_class)
    {
    }
  private:
    CMyClass* m_pcMyClass;
};
```

In ARGoS, we generally opted for pointers:

> **!** When an attribute refers to an external structured type, **always** prefer pointers.

## 4.3 Is *const* Really Necessary?

The short answer is yes. The long answer is yeeeeeeeeees. Now for the reasons.

A function can be seen as a service. Its declaration is a sort of contract between the user and the provider. If the two respect the contract, both get

the right result. A fundamental part of this contract is what to do with the exchanged objects: the user passes some of his objects and needs to know whether or not they are going to be changed. The same applies to the function's return values: can the user change it, once received, or not?

As a consequence of such contract, in a class, methods can be roughly divided in two categories: *inspectors* and *modifiers*. Inspectors just view the content of a class, but promise not to change it. On the contrary, modifiers are meant to change it.

The *const* keyword exists to make all this possible, with the added value that you can spot contract breaches at compile time. Unfortunately, the definition of *const* in C++ is one of the most shamelessly confusing a human mind has ever imagined. The first thing to remember is that *const* refers to what *precedes* it. Some examples:

```
SInt8  const  nVar;  // const signed integer
SInt8* const  pnVar; // const pointer to signed integer (1)
SInt8  const* pnVar; // pointer to a const signed integer (2)
```

The examples marked with (1) and (2) are tricky. The first says: you can change the value pointed to by pnVar, but you cannot change the pointer itself. In other words:

```
SInt8 nVar1, nVar2;
SInt8* const pnVar(&nVar1); // creation of the variable -> OK
*pnVar = 10;                // setting the value -> OK
pnVar = &nVar2;             // setting the pointed address -> ERROR!
```

Case (2) works in the opposite way: you can change the pointed address, but you cannot change the value you find there:

```
SInt8 nVar1, nVar2;
SInt8 const* pnVar(&nVar1); // creation of the variable -> OK
pnVar = &nVar2;             // setting the pointed address -> OK
*pnVar = 10;                // setting the value -> ERROR!
```

Clearly, you can define a constant pointer to a constant variable like this:

```
SInt8 const * const pnVar;
```

For what concerns references, you have to remember that a reference *is* the referenced object. Therefore:

```
SInt8  const& nVar; // reference to a const signed integer -> OK
SInt8& const  nVar; // const reference to signed integer -> MEANINGLESS
                    // because it corresponds to this:
SInt8  const  nVar;
```

To make it simpler for programmers coming from other languages and more confusing for everybody else, *const* can also be used in the following *degenerate* form:

```
const SInt8  nVar; // const signed integer
const SInt8& nVar; // reference to a const signed integer
```

In ARGoS, we prefer this latter, degenerate form.

Back to the uses of *const*, class inspectors and modifiers are declared as follows:

```
class CMyClass {
  public:
    void Inspector() const
```

```
    {
      ...
    }
    void Modifier()
    {
      ...
    }
};
```

Recalling the discussion and about copies, references and pointers (Section 4.2), now we can fully appreciate this example:

```
class CARGoSEntity {
  public:
    // inspector, return const reference
    inline const CVector3& GetPosition() const
    {
      return m_cPosition;
    }
    // modifier, pass const reference
    inline void SetPosition(const CVector3& c_position)
    {
      m_cPosition = c_position;
    }
    // inspector, return copy
    inline Real GetOrientation() const
    {
      return m_fOrientation;
    }
    // modifier, pass copy
    inline void SetOrientation(Real f_orientation)
    {
      m_fOrientation = f_orientation;
    }
  private:
    CVector3 m_cPosition;
    Real m_fOrientation;
};
```

## 4.4 Why and When to Inline

In a nutshell, inlining a function means to copy its content in the place where it's called. This saves the cost to perform a function call, which is quite expensive. For this reason, inlining may be a pretty attractive optimization tool.

The decision whether or not to inline a function is ultimately performed by the compiler, but the compiler doesn't do it if you don't mark the possible candidates. Not all methods are good candidates. Short functions of few, simple lines are the best, especially if these functions are called often.

> **!** **Always** inline a class' getters and setters.

Too much inlining is harmful. A method, to be inlineable, must be declared in the `.h` file. If you do it for too many methods, you break the *information hiding* principle and render the header file difficult to read.

Furthermore, too much inlining has a performance hit. In fact, when a method is inlined, the code of the caller grows and may exceed the size of a memory page, thus obliging the OS to load and unload memory pages to fetch the necessary code. For this reason, once again, inline only few small functions.

! **Never** inline methods that include loops.

## 4.5 You Have a New Friend: *typedef*

Sometimes functions exchange complex structures, such as the following:

```
std::map<std::string, Real> mapMyFancyStructure;
```

Remembering the definition of that map is though, especially if there are other similar definitions for other things. And what if you find a better way to store your data, and that map changes definition? You'd have to change all the code that refers to it—a nightmare.

Compare this example:

```
void DoSomething(std::map<std::string, Real>& map_my_fancy_structure)
{
  ...
}
```

with this one:

```
typedef std::map<std::string, Real> TMyFancyStructure;

void DoSomething(TMyFancyStructure& t_my_fancy_structure)
{
  ...
}
```

The second is much more readable and easier to use and maintain.

! **Always** typedef complex type definitions, such as maps and vectors.

Notice that unlikely to what happens in C, when you declare a union, an enum or a struct, you automatically typedef it.