LHCb 2001-054

COMP

27 April 2001

Revised C++ Coding Conventions

Olivier Callot

Laboratoire de l'Accélérateur Linéaire - Orsay

ABSTRACT

This document replaces the note LHCb 98-049 by Pavel Binko. After a few years of practice, some simplification and clarification of the rules was needed. As many more people have now some experience in writing C++ code, their opinion was also taken into account to get a commonly agreed set of conventions.

Table of contents

1 Introduction

Working in a collaboration implies that the "products" are usable by many people. In the case of software product, this means also that people who are not the authors will maintain them. A set of rules and conventions is necessary to insure a minimal coherence and consistency. The purpose of this document is to write such a set of rules.

One can discuss if all rules are at the same level, if some rules should be MANDATORY while other are only RECOMMENDED. This means that code is always unacceptable if a mandatory rule is violated. We all know that there are always exceptions, with good or less good reasons. Having a mandatory rule implies also that there is a tool to check the rules, which we don't have. I propose for the time being to have all the rules as is, making a difference by the phrasing of the sentence: "must be", "is forbidden" indicate clearly mandatory rules, "should", "can be" mean that this is recommended.

As we have no tool to check these rules, and as many people are able to commit their changes to the CVS repository, it is quite difficult to install a policeman to enforce the rules. As in many other areas in a collaboration, we rely on the goodwill of all contributors, who are working for the success of the overall project.

Some of the rules are subject to strong personal views, in particular the coding style, as this reflects aesthetic judgements and are based on personal experience. Common sense should be used, and everyone should try to follow the majority's views. Here, rules should be seen as guidelines, where exceptions are possible. Nobody will reject a source file with one line containing 81 characters. However, having most lines with over 100 characters is totally unacceptable, and shows that the meaning of collaboration, which means "working together", has not been accepted by the author.

For several rules, a good working environment would help. In particular, a common LHCb set-up for `emacs` and `Visual Studio` could help writing code which follows the rules. Experts of these tools are welcome to propose and develop such an environment.

2 Organisation

The LHCb software is organised in **PACKAGES,** which is a set of classes and the instructions to build and use them. Packages can be released independently, but they usually depend on other packages. The source files are maintained with **CVS** and released in the LHCb release area. A development area $LHCBDEV also exists. The documentation on the handling of packages (getting the package, committing changes, releasing a new version) is available on the LHCb web. This document concentrates on the content of a package, i.e. file organisation in the package, and (mainly) contents of the source files.

R1 Each package has an **unique name**, which should be written such that **each word starts with an initial capital letter.**

R2 Each package belongs to a **package group**, a short name like detector name, software product, etc. This is sometimes called "hat". The Gaudi packages are an exception, as they are in a different area and are not LHCb specific.

R3 Several directories are mandatory in the package:

- **cmt** for the requirement files and the makefiles.
- **src** for the private source files. A sub-directory level may exist in this directory, in case the package builds a static and a dynamic library.
- **doc** for the documentation on the package.
- **options** for Brunel option files, if the package contains algorithms.
- **<package>** for the public files of the package, typically header files
- **<binary>** for the compiled and possibly linked files. One directory per operating system and debug version, e.g. i386_linux22, Linuxdbx, Win32, Win32Debug.

R4 Each class should have a header file and an implementation file, the name of the files should be the name of the class. There should be no other `"."` in the filename, due to nmake limitations.

R5 C++ include (header) files should have the extension `".h"` and C++ implementation files should have the extension `".cpp"`. **No other file type is allowed**.

R6 The first line of every file should contain the CVS **`$Header:$`** macro, which CVS translates into the responsible and date of the most recent CVS commit.

```
Example: // $Header: $
```

R7 For including standard files, use : `#include <filename>`
For user files, use the syntax `#include "Package/FileName.h"`
Never use a relative path for include file.

For private header files which are in the same directory as the source file, you should use `#include "FileName.h"`

R8 The syntax of the `"cmt/requirements"` file has limitations due to nmake. In particular:

- Macro names must not contain the minus sign "-"
- Macro definitions must be enclosed in round brackets "()"
- The use of curly brackets "{}" is not allowed, except in `set` statements.

3 Naming conventions

Naming is something difficult to check by an automated tool, and everybody has his own feeling of what is a good name for an entity. However one should follow some guidelines, so that everyone understands what sort of entity is behind the name. Names should be meaningful, but not too long so that statements are not too long!

R9 Names are usually made of several words, written together **without underscore**, each first letter of a word being **uppercased**. The case of the first letter is specified by other rules, but is usually lowercase. Don't use special characters. **Only alphanumeric characters are allowed.**

```
Example : nextHighVoltage
```

R10 Names are case sensitive. However, do not create names that differ only by the case.

```
Example : track, Track, TRACK
```

R11 Avoid single character, or meaningless names like "`jjj`" except for local loop or array indices.

R12 **Class names** must be nouns, or **noun phrases**. The first letter is capital.

R13 **Data Member** names should be <u>private variables</u> and start with "**m_**" followed by a **lowercase lette**r.

- Protected variables may be used for base classes.
- Never use public data members. Instead you should use accessors functions.

R14 **Static variables** should start by `"s_"` to distinguish them clearly.

R15 **Avoid global variables**. They cause problems with shared images. Their function is better replaced by a class with mainly accessor methods. Avoid global functions and operators. The only case where you can use them is symmetric binary operators and mathematical functions.

R16 **Accessor functions** are named from the variable they control. The "get" accessor is mainly the variable name, the set accessor has a "set" prefix.

```
Ex: trackHits( )         // Get the value of m_TrackHits
    setTrackHits( nnn )  // Set the value.
```

R17 Other functions must be **verb** or **verb phrases**. Like all member functions (excluding creators) they start with a lowercase letter.

R18 Functions that create or make a new object should start with the verb `"create" or "make".`

4 Header files

Header files with extension "`.h`" contain the description of the class. They may contain some implementation, but only in the case this is inline code. The rest of the code is in the implementation file, with extension "`.cpp`".

R19 A header file should contain **the definition of a single class**. If this class defines and uses internally another class, they can be both defined in the same file.

R20 Every header file contains **a mechanism to prevent multiple inclusion**. The file "Package/File.h" starts and ends by the following lines:

```
#ifndef PACKAGE_FILE_H
#define PACKAGE_FILE_H 1

   … body of the header file …

#endif   // PACKAGE_FILE_H
```

R21 One should minimise the number of header files included, to avoid too complex dependencies. In particular, if one uses only a pointer or reference to a class, one can just do a **forward declaration**, without including the class header file :

```
class Line;          // forward declaration, this is enough
class Point {
public:
  Number distance(const Line& line) const; // distance
};
```

R22 The class declaration starts with the "`public`" members first, this includes the constructors and destructors. Then the "`protected`" and the "`private`" sections. One may use `typedef` to clarify the typing, they should be at the beginning of the public section of the class declaration.

R23 Header files should not have any method bodies inside the class definition. **The "`inline`" functions should go at the end of the file, after the class definition.** The other functions should all be in the implementation file. Inline function should NOT call something external to the class. One could however **put a very simple accessor on the declaration line**, to make the header file more readable.

```
class Point {
public:
  Number x()                     const {return m_x; }
  void   setX( const Number x )        {m_x = x;    }
Private
  Number m_x;
};
```

5 Implementation file

The implementation file contains the non-trivial member functions. They should hold the definition of a single class, they may contain auxiliary (private) classes if needed.

R24 **A constructor should initialise all variables and internal objects** which may be used in the class. Variables may also be initialised by their declaration in the header file.

R25 **A copy constructor is mandatory, together with an assignment operator, if a class has built-in pointer member data.**

```
class Line {
public:
  Line (const Line & );             // copy constructor
  Line & operator= (const Line &); // Assignment operator
```

Note that the arguments should be `const reference`. If you want to prevent copy and/or assignment, you should provide a `private` declaration of the copy and assignment constructors. The object can no longer be copied.

R26 **Declare a "virtual" destructor for every class**

```
class Line {
public:
  virtual ~Line( );     // virtual destructor
};
```

R27 Virtual functions should be re-declared "virtual" in derived classes, just for clarity. And also to avoid mistakes when deriving a class from this derived class…

6 Member functions and arguments

R28 Functions without side effects are by far better. **The use of the `"const"` specifier is strongly encouraged**, to make clear that a function doesn't change the object, or that the arguments are not changed.

```
class Point {
public:
  Number distance( const Line& line ) const; // distance
  void   translate( const Vector& vector );  // translate
};
```

The 'distance' function has a `const` argument, as the line is not modified, and is "`const`" as the point is not affected. The 'translate' method is also not affecting its argument, but the object is modified and the method can not be "`const`".

R29 **Pass objects by constant reference**. Passing by value is also acceptable for small objects.

R30 The use of default arguments is strongly discouraged. This reduces the risk of forgetting an argument…

R31 **Do not declare functions with unspecified** (`"..."`) **arguments.**

R32 **The returned value of a function should always be tested.** A function that does not have a `void` type should always return a meaningful value.
"`StatusCode::SUCCESS`" is a good return value, this is used in the Algorithm class.

R33 The exception mechanism should be used only to trap "unusual" problems.

7 Coding style

This area is usually the most debated, as some aesthetic considerations are involved. There are sometimes also 'religious' issues, because there is no way to convince each other that this is better than that. It's only something one can believe, but can't prove!

But it is important to try to get a similar look, for an easier maintenance, as most code writers will be replaced during the lifetime of the experiment.

7.1 General lay-out

R34 The length of any line should be **limited to 80 characters**.

R35 Each block is **indented by two spaces**. It starts by an opening brace **on the line** of the control statement, and end by a closing brace alone on its line, with optionally a comment indicating which block is closed. In case of `"if ( ) {"` statement, the closing brace may be followed on the same line by an `"else {"` or `"else if ( ) {"` clause.

```
if ( 1 < x ) {
  log << MSG::INFO << "x>1" << endreq;
} // x greater than 1
```

One can also put a short single line statement on the same line, like:

```
if ( 1 < x ) { x = 1; }
```

R36 When declaring a function, try to put one argument per line, this allows inline comments, and helps to stay in the 80 column limit

```
int myFunction( int          intVAlue,
                std::string  aString,
                MyClass*     myClassPointerValue );
```

7.2 Comments

Comments should be abundant, and must follow the following rules to allow an automatic documentation by the DOXYGEN tool:

R37 **Every header file should have a comment block to describe the class**, in the format appropriate for the tool, just before the class declaration:

```
/** @class ClassName ClassName.h Package/ClassName.h
 *
 * The first sentence will be used as a summary. The rest of
 * the text is a complete, free format description. It
 * should end with author and date in the format:
 *
 *   @author Your Name
 *   @author A colleague helped also
 *   @date  08/03/2001
 */
class ClassName {
```

R38 **Every method should have a description before it**, either in a single line with `"///"`:

```
/// single line description of the method
StatusCode Method();
```

Or with a block, again starting with `"/**"` and ending with `"*/"`. Arguments and parameters can be indicated by special tags:

```
/** A method with arguments to be documented on
 *   several lines if needed.
 *   @return  status code
 *   @param arg1 this is the meaning of this argument
 *   @param blabla another reason
 */
virtual int method2( Type1 arg1, Type2 blabla );
```

R39 For comments that you don't want in the documentation, use a simple `"//"`. The C-like syntax is discouraged, except in the DOXYGEN formatted blocks. Use blank lines to separate blocks of statements, but don't use blank comment lines, i.e. a line which contains only `"//"` and nothing else is more a nuisance for the code clarity.

7.3 How to write a safer code

R40 **Comparison between a variable and a constant should have the constant first**, the compiler will then spot it if you forgot one of the equal signs in

```
if ( 0 == value ) {
```

By comparison,

```
if ( value = 0 ) {
```

is a valid statement, but is not what you usually want !
If you are doing an assignment is a comparison, make it explicit by using parenthesis:

```
While ( 0 != (ptr = iterator() ) ) {
```

R41 Comparison between floating point values should not test for equality. In case this is what you want, test that the difference is smaller than a small number.

R42 **To test that a pointer is valid, compare it to the value zero**, do not treat it as having a Boolean value:

```
if ( 0 != ptr ) {
```

R43 In "`switch`" statements, **each choice must have a closing `break`**, or a clear comment indicating that the fall-through is the desired behaviour.

R44 **Constants should NOT be defined by `#define` pre-processor statements.** One should use `enum` for integer, or `const` declaration. They are best put inside a **`namespace`** block to avoid naming conflicts.

```
namespace CaloName {
  static const unsigned int nBits = 6;
}
```

and use them like that :

```
if ( CaloName::nBits == value ) {
```

R45 Re-use existing classes. STL should be exploited. Old C habits should be changed…

- Use `std::string` instead of `char*`.
- Don't use built-in arrays. Use one of the STL containers like `std::vector`.
- Use standard classed like `"Hep3Vector"` rather than `"double v[3]"`
- Do not use `struct,` use a class to hold your data.
- Do not use `"union"` types
- Use `"bool"` type for logical values.
- Use the "`MsgStream`" for all your outputs
- Explicitly put "`std::`", this is mandatory for Windows, even if it works without it on Linux.

R46 **Avoid overloading operators**, unless there is a clear improvement in the clarity of the code.

R47 **Avoid complicated implicit precedence rules**, use parentheses to clarify your wishes.

R48 **Use cast operators** for data-type conversion . You should use `static_cast` or `dynamic_cast`, but not `reinterpret_cast` or `const_cast`. The best is to provide the appropriate cast operator inside the class.

R49 All C++ entities should be **defined only in the smallest scope** they are needed. Unfortunately there is a problem of inconsistency between Linux and Windows for the scope of a loop variable.

```
for ( int j = 0; 5 < j; ++j ) {
  ..
}
```

Windows extends the scope of "j" outside the loop! <u>It is better to declare "j" explicitly</u> before the loop:

```
int j;
for ( j = 0; 5 < j; ++j ) {
  ..
}
```

R50 Every invocation of **`"new"` should be matched with exactly one invocation of `"delete"`**, and they should be clearly related if they are more than 5 lines apart. This **doesn’t apply for objects created on the Transient Store:** This is the case you want the object to persist longer that the code which created it !

R51 A function must not use the `"delete"` operator to any pointer passed to it as argument.

R52 **Use `"new"` and `"delete"`** in place of `"malloc()"` and `"free()"`.

R53 Any pointer to automatic objects should have the same or a smaller scope than the object it points to.

7.4 Readability and maintainability

R54 **Functors are discouraged**. They hide the real work in another source file. They may be needed, in particular to use the `std::sort` algorithms. For simpler operation, an explicit loop is by far clearer for the reader of the code.

R55 **User defined operators are discouraged.** If used, they should behave 'naturally', i.e. as they behave for usual numbers and objects:

- The assignment operator function should return a reference to their left operand.
- The `"+"` operator should do something like an addition.
- The `"[]"` operator should be an access by a sort of index.

The only purpose of an operator is to save typing. But their use tends to make the code cryptic for anyone else than the author. Member functions are as good, and carry a description of their purpose in their name.

R56 **Macro are discouraged** for producing code. They make the code more difficult to understand and to maintain, and are impossible to debug.

R57 **Use spaces** to separate the operators from their operands:

`m_x = x;` is more readable than `m_x=x;`

R58 A function should have a **single return statement**. One may use return statements when checking the arguments as the beginning of a function, but one should avoid a return statement in the middle of nested loop and if blocks.

R59 One should avoid using `goto` statements. They make the code structure more confused.

R60 The conditional operator "`condtion ? true : false`" is discouraged. It is acceptable only for very simple statements and never nested. A simple "if" block is preferred, it is by far more readable, and as efficient.

8 Examples

8.1 Header file

```
//$Header: $
#ifndef PACKAGENAME_CLASSNAME_H
#define PACKAGENAME_CLASSNAME_H 1

// Include files
#include <string>
#include <list>
#include <map>
#include "Package/AnotherClassName.h"

// Forward declarations
class NeededClass;

/** @class ClassName ClassName.h Package/ClassName.h
 *
 *  The first sentence is used as a summary description by the
 *  documentation tool. The description of a class is expected
 *  to be found before the class declaration in the .h file.
 *  The code is documented following the JavaDoc style. Using
 *  the multi-line documentation block or the simple line
 *  documentation as is shown in this example.
 *  The cvs keyword in the first line of the file will be
 *  expanded by the code management tool to include the
 *  file path name, revision number, the author and the state.
 *
 *  @author  Author Name
 *  @author  Another Name
 *  @date    20/11/2000
 */

class ClassName : virtual public AnotherClass,
                  virtual public OtherClasses {
public:
  // typedefs and local class declarations
  typedef std::list<IService*> ListSvc;
  typedef std::map<const std::string, const ISvcFactory*> MapFactory;

  /// default creator
  ClassName(Type1 argument1, Type2 argument2);
  /// virtual destructor
  virtual ~ClassName();

  /** Description of the method. Again the first sentence
   * is used as a summary by the documentation tool. It is
   * also expected to find comments before the method
   * declaration.
   */
  virtual void method1();

  /** Another method with some arguments. The arguments can be
   *  documented as normal inline comments
   *
   *  @return status code
   *  @param argument1 this is the first argument
   *  @param argument2 this is the second argument which goes
  *                    after the first argument
  */
  virtual int method2( Type1 argument1, Type2 argument2 );

  /// Yet another method of this class.
  StatusCode method3();
protected:

private:
  int     m_refcount;  ///< Reference counter
  Type1*  m_member2;   ///< Pointer to an object of type Type1
  Type2&  m_member3;   ///< Reference to an object of type Type2
};

#endif  // PACKAGENAME_CLASSNAME_H
```

8.2 Implementation file

```
//$Header: $

// Include files
#include <string>
#include <list>
#include <map>
#include "Package/ClassName.h"


// method1 (the description will be taken from the .h file)
void ClassName::method1() {
  // Implementation of method1. In order to not take unnecessary
  // width we should use 2 spaces for each identation.
  if ( bla & bla ) {
    int i;
    for ( i = 0; i < n; i++ ) {
      // do something useful here
      ...
    }
  }  // if
}

// method2
int ClassName::method2( Type1 argument1, Type2 argument2 ) {
  int i;
  float f;
  for ( i = 0; i < 1000; i++ ) {
    // something here
  }
  return i;
}

// method3
StatusCode ClassName::method3() {
  return StatusCode::SUCCESS;
}
```