LHCb

C++ Coding Conventions

LHCb Computing Note

Issue: 1
Revision: 1

Reference: LHCb 98-049 COMP
Created: 17th April, 1998
Last modified: 13th May, 1998

Prepared By: Pavel Binko

Abstract

The purpose of this document is to define one style of programming in C++. It contains LHCb naming conventions for all C++ entities, such as classes, data members, functions etc. It also contains practical and useful guidelines, including examples, which should be followed when writing C++ code. This will help to ensure understandable and good quality code. Support for training in the form of courses, recomended books and other documentation is also described. This document will be updated regularly as more experience is gained using the C++ language.

Document Status Sheet

Table 1 Document Status Sheet

1. Document Title: [Title qualification]			
2. Document Reference Number: LHCb 98-049 COMP			
3. Issue	**4. Revision**	**5. Date**	**6. Reason for change**
1	1	15 May 1998	First version

Table of Contents

1 Introduction

This document is under development. The rules and recommendations presented here are not final, but should serve as a basis for continued work with C++. This collection of rules should be seen as a dynamic document; suggestions for improvements are encouraged.

This draft has been compiled by studying the documents [1]-[8], and many useful examples have been added. LHCb often follows the BaBar C++ rules and recommendations, in some places in a slightly modified form.

The purpose of this document is to define one style of programming in C++. It contains practical and useful rules and recommendations including examples, which should be followed when writing C++ code. Good programming practices together with uniform coding style can significantly reduce the debugging phase. None of these rules and recommendations is absolute, but waiving of an important rule has to be agreed in advance with the project architect or package coordinator, and documented in detail.

The rules and recommendations should be applied before too much work developing C++ code has been done. It should improve productivity of programmers. Such C++ code should

- have a clear and consistent style
- be easy to read and understand
- be portable to other architectures
- be free of common types of errors
- be easy maintainable and changeable by different developers during the lifetime of the LHCb experiment

Special note should be taken of the following:

ANSI Standard

The ANSI C++ standard should be followed to help ensure portability between different platforms. Language extensions to the standard should be avoided. However at present there is only a draft standard [9]. Compiler options that force adherence to the ANSI standard should be used, as and when they become available.

Code Checking Tool

We will shortly start to evaluate a tool (Code Wizard) which will allow the coding conventions described in this document to be checked automatically. This document will be updated with information on how to use this tool as soon as it has been configured to check the LHCb conventions.

2 Organisation of the Software

Some of the naming conventions follow from the way code is organised and managed. The LHCb software will be divided into packages, each typically containing a set of classes which can be logically grouped together. Each package can be released independently of each other. Packages will be stored in the LHCb code repository, which is based on CVS [10]. The following rules apply when naming packages and their associated directories, files and libraries.

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

`Example : VertexTracking`

R2 The name of the directory containing the files corresponding to a package should be identical to the package name. It will appear in all statements referencing its files.

`Example : #include “PackageName/FileName.Extension”`

R3 Each pacakge must have an associated abbreviation, which will be subsequently used in naming classes belonging to the package. It is recomended to use 3 (maximum 4) letters for the abbreviation. The abbreviation should start with a capital letter and follow with lower case letters.

`Example : Vtr (for VertexTracking package)`

R4 C++ include files should have the extension `“.h”`

R5 C++ implementation files and programs have the extension `“.cpp”`

R6 Use a seperate `“.h”` file and corresponding `“.cpp”` file for each C++ class.

R7 The file name should correspond to the class name (see **R18**), including the case.

`Example : VtrTrackFitter.h`

R8 On UNIX library names should have the prefix `“lib”` followed by the package name. Archive libraries have the extension `“.a”` and shared libraries the extension `“.sl”`

`Example : libVertexTracking.a (for archive library)`

R9 On Windows NT the extensions are different. Archive libraries have the extension `“.lib”`. Shared libraries consist of two files; libraries themselves have the extension `“.dll”` and export files the extension `“.lib”`.

3 Naming Conventions

3.1 General Rules

R10 Names of C++ entities should be chosen with care, and should be meaningful. This is very important to make the code easy to read. The checking cannot be automated but relies upon the reviewer.

R11 Use multiple word names to describe the entity more precisely. The words are written together, don't separate them with underscores.

```
Example : nextHighVoltage
```

R12 Avoid the use of special characters within names as these are sometimes interpreted in a special way by compilers.

```
Example : #,$,&,@,...
```

R13 Do not create names that differ only by case

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

R14 Acronyms that are part of names should be all capitals

```
Example : anECALCluster
```

R15 Avoid non-standard acronyms and abbreviations.

```
Example : use "nameLength" instead of "nLn".
```

R16 Using the occasional article (`"aBMesonCandidate"`) or additional information (`"nextBMesonCandidate"`) can help where an identifier looks a bit odd (`"bMesonCandidate"`).

R17 Avoid single and simple character names (e.g. `"j", "iii"`) except for local loop and array indices.

R18 It is **not** mandatory to use implicit prefixes for built-in types , such as `"i"` for integers, `"p"` for pointers, `"s"` for strings, etc.

R19 Global variables should be avoided at all costs.

3.2 Class Names

R20 Class names should be **nouns or noun phrases** . The class name should start with the prefix corresponding to the abbreviation of the name of the package the class belongs to.

```
Example : "VtrPhiCluster" corresponds to the PhiCluster Class and
is found in the Vertex Tracking package ( Vtr )
```

3.3 Data Member Names

R21 We recommend that data members within a class should be decalared private. Public data members should be avoided. Protected data members can be used when necessary.

R22 Names of private and protected variables should start with a small letter.

```
Example : trackHits
```

R23 Names of constants should be written in capital letters.

```
Example : PI
```

3.4 Naming Accessor Functions

Accessor functions provide access to private data members and are used to either get or set the value.

R24 The “get” accessor function returns the current value(s) of the private data member(s)

```
Example : getTrackHits (NB the T in trackHits is now capitalised)
```

R25 The “set” accessor function sets the private data member(s) to the new current value(s)

```
Example : setTrackHits
```

3.5 Naming All Other Functions

R26 Function names should be **verb or verb phrases**. They should have initial lower case letters.

```
Example : drawTrack
```

R27 Functions that create a new object, but which are not responsible for deleting it, should start with “create”.

4 C++ Coding Guidelines

This chapter contains some general guidelines which, if followed, can be helpful in producing high quality code that can be easily understood by others. As a general rule, code for clarity rather than efficiency. Some general guidelines follow. More specific issues are treated in the sections that follow.

R28 Routines that do not have a void return type should always return a reasonable return code. Always check the return code, don't assume that the call succeeded. The exception mechanism should be used to deal with "unusual" circumstances rather than exploiting error codes. Check the input parameters, if your function operates only on a limited range of input values.

R29 Re-use existing and third party classes where possible,. The Standard Template Library (STL) is the primary source for simple classes for manipulation of strings, arrays and containers.

```
Example : use the STL string class rather than "const char*" .
```

R30 Do not use built-in arrays "[ ]". They do not offer any protection from going outside array bounds, and they are rarely the most suitable container. Choose one of the STL container classes instead.

R31 Classes can be used when passing arguments and return values rather than built-in arrays.

```
Example : use the CLHEP object "Hep3Vector" rather than passing
the array "double v[3]".
```

R32 Avoid overloading functions and operators unless there is a clear improvement in the clarity of the resulting code.

R33 Avoid implicit data-type conversions and mixed mode arithmetic. Use cast operators.

```
Example : x = (float)i + 0.5
```

R34 Avoid complicated implicit precedence rules (use parentheses).

R35 Use "`const`" for read-only arguements. In particular, it should be always used with input arguments to functions that are passed as pointers or via references. This allows the compiler to optimise, and clues the user that there will be no side effects.

R36 For true/false data items, use the "`bool`" type of C++ for booleans (C programmers may tend to use "`int`" instead of "`bool`").

R37 Do not use "`struct`" types. The "`class`" is identical to the "`struct`" except that by default its contents are private rather than public.

R38 Do not use "`union`" types.

R39 Use the "`iostream.h`" functions rather then those defined in "`stdio.h`".

4.1 Scope Considerations

R40 All C++ entities should be defined only in the smallest scope, where they will be used. Avoid public data members in your class definitions.

R41 Avoid global variables, encapsulate them in a class.

```
Example :
class MyGlobals {
public:
  static void myFunc( );
  static int getMyData( );
  static void setMyData( int value );
private:
  static int myData;
  virtual dummy( ) = 0; // Trick to make the class abstract
};
```

R42 For read and write access to data members, use:

```
int getMyData( ) const;
void setMyData( const int value );
```

R43 Avoid global functions and operators as much as possible. They may be used, for example, for symmetric binary operators and mathematical functions (where a global function seems more appropriate than using an object).

4.2 Comments

R44 Every include file should have its own comment that contains the author's name, the modification history, and describes the purpose of contained class in detail. It is not necessary to repeat the whole history in the corresponding implementation files. See the examples in Chapter 6.

R45 Before every function declaration, there must be a short description of its functionality, the input and output arguments and its side-effects.

R46 Don't bother to comment individual source lines

R47 Use "//" for comments. If the comment extends over multiple lines, each line must begin with "//".

R48 Use the C-like syntax `"/* Comment inside a command line */"` only if you need more code on the same source line (beyond the comment).

R49 All "#else" and "#endif" directives should carry a comment that tells what the corresponding "#if" was about if the conditional section is longer than five lines.

```
#ifndef POINT_H
#define POINT_H
// Two-dimensional point definition
class Point {
public:
  Point(Number x, Number y);                // Create from (x,y)
  Number distance(Point point) const;       // Distance to a
point
  Number distance(const Line & line) const; // Distance from line
  void translate(const Vector & vector);    // Shift a point
private:
  Number x;              // X-coordinate
  Number y;              // Y-coordinate
};
#endif // POINT_H
```

4.3 Include Files

R50 Include files should hold the definition of a single class. It is possible to define more classes in one include file, if these classes are embedded or if they are very tightly coupled.

R51 All data members of a class should be declared "`private`" (or "`protected`"). This increases the data security and ensures that data members are only accessed from the member functions of that class. Hiding data makes it easier to change implementation and provides a more uniform interface to the object.

R52 Classes should be declared with "`public`" first, then "`protected`" and "`private`" sections. This is the order one wants to understand the definition - first the public interface, and then the protected and private entities. Within each section embedded types (e.g. "`enum`" or "`class`") should appear at the top of that section.

`Example : see example in section 6.1`

R53 Class definition starts by default with a "`private`" section. As "`friend`" declarations are not affected by the "`private`", "`protected`" and "`public`" keywords, placing them at the beginning of the class definition seems most natural (use friend declarations sparingly).

R54 Use the directive `#include <FileName.Extension>` in order to include from standard directories, such as `"/usr/include/CC"`. To include user defined include files, use names enclosed in double quotes e.g. `#include "FileName.Extension".`

R55 All LHCb include directives should specify a path, which identifies the package name.

```
Example : #include "PackageName/FileName.Extension"
```

Note that external software may use more complicated package hierarchies. The following example comes from the external package `"HepODBMS"`, which has a sub-package `"tagdb"`, and the include file name itself is `"HepEvent.h"`.

```
Example : #include "HepODBMS/tagdb/HepEvent.h"
```

R56 There will be a package LHCb, which will contain definitions common for the whole LHCb software. The first line of code in each include file should include the main LHCb collaboration wide definition file:

```
Example : #include "LHCb/LHCb.h"
```

R57 The number of files included should be minimised. If a file is included in an include file, then every implementation file that includes the second include file must be re-compiled whenever the first file is modified. A simple modification in one include file can make it necessary to re-compile a large number of files.

R58 When only referring to pointers or references to types defined in a file, it is often not necessary to include that file. It may be sufficient to use a forward declaration to inform the compiler that the class exists

```
class Line;                // Forward declaration
class Point {
public:
  Number distance(const Line & line) const; // Distance from line
};
```

Here it is sufficient to say that Line is a class, without giving all the details of the class, which are inside its header. This saves time in compilation and avoids an apparent dependency upon the Line header file.

R59 Avoid relative search paths when including files. Your code should be portable and independent of the underlying operating system. Instead, search paths should be provided in "make" files as options for the compiler.

4.4 Inline Functions

R60 Header files should not have any method bodies inside the class definitions. Any bodies of "`inline`" functions should go at the end of the header file after the class definition. Any bodies of other functions should go to the implementation file. This improves the readability of the interface. We recommend use of "`inline`" functions to return values stored as "`private`" (or "`protected`") member data.

```
class Point {
public:
  Number getX() const; // Return the x coordinate
private
  Number x;
};
inline Number Point::getX() const { return x; }
```

R61 Normally it is advisable to avoid long, complex "`inline`" member functions. They reduce clarity, make it take longer to compile and harder to debug, and the performance gain from making them "`inline`" is small for large routines. To avoid problems, do not inline :

- functions calling member functions of other classes
- functions call virtual member functions of this class
- functions calling templated functions

In effect, do not inline anything that calls something else, or involves significant control structures.

4.5 Multiple inclusion

R62 Every header file has to contain a mechanism to prevent multiple inclusion of it. Assuming the header file "Package/File.h", the LHCb convention is:

```
#ifndef PACKAGE_FILE_H
#define PACKAGE_FILE_H
...
#endif // PACKAGE_FILE_H
```

4.6 Enumerations

R63 Use "`enum`" for small named integer constants within classes rather than "`const`". Unfortunately, you could meet some problems defining "`const`" within classes

`Example : use of "const int HALTED=15;" inside a class is not supported by some compilers).`

The "`enum`" construct allows a new type to be defined and hides the numerical values of the enumeration constants.

```
enum State {HALTED, STARTING, RUNNING, HALTING};
enum State {HALTED=15, STARTING=1, RUNNING=2, HALTING=3};
```

R64 Do not use "`#define`" to define constants, because it can result in name collisions in other people's code.

4.7 Implementation Files

R65 Implementation files should hold the member function definitions for a single class (or embedded or very tightly coupled classes) as defined in the corresponding header file.

`Example : See the example in section 6.2`

R66 It is always a good practice to design functions without any side effects (no-one would expect `sin(x)` to modify `x`). Systematic use of the "`const`" qualifier can help a lot. Declare as "`const`" all those member functions that are not meant to alter member data. It makes the code safer, and is useful in understanding what a class is supposed to do.

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

The first function is "`const`", it calculates the distance of "`Point`" from "`Line`". Translate must not be "`const`", because it changes "`Point`" (but is not allowed to change "`Vector`").

R67 Re-declare virtual functions as "`virtual`" also in derived classes. This is just for clarity of code. The compiler will know it is "`virtual`", but the human reader may not.

4.8 Argument passing

R68 Pass small objects by value and large objects by constant reference. Passing an object by value creates a new copy of the object, which is fine if the object is small. Passing by constant reference passes the address of the object. The compiler will not allow the arguments to be modified.

```
class Point {
public:
  Number distance(Point point) const;       // Distance to point
  Number distance(const Line & line) const; // Distance from line
 };
```

Here a "`Line`" may be large so it is passed by constant reference.

If the previous example had been written:

```
class Point {
public:
  Number distance(Point point) const;       // Distance to point
  Number distance(Line & line) const;       // Distance from line
 };
```

the member function, `Number distance (Line & line) const`, would be allowed to modify the Line passed to it. This fact should be apparent from the function name (which is not the case here).

R69 Use references rather than pointers as arguments whenever possible.

R70 Avoid use of default arguments. It reduces the risk that you miss off one argument by mistake (it will be detected at compile time).

R71 Do not declare functions with unspecified arguments (i.e. avoid using "`...`"). The compiler cannot check them.

R72 Pre-conditions and post-conditions should be checked with "`assert(condition)`". These should be at the beginning and end of a routine (and commented).

```
#include <assert.h>
[....]
void
MyClass::myFunction( char* name ) {
  assert ( 0 != name ); // Check that the name is not NULL
}
```

The assertion can be removed from the code, once it has been tested, by compiling with the symbol NDEBUG defined. If checks for internal logic errors are outside an assert, they should call abort(), not assert(0) when they fail. This way the action and check will have the same behaviour in test and production versions.

4.9 Initialisation

R73 Variables and objects should always be created in a valid, ready-to-use state. Don't expect the user to call an "`open`" function, nor a "`close`" (use the destructor).

R74 Initialise objects at the time of declaration.

```
Number n(0);          // initialises n to 0
Point p(20,10);      // initialises the two values of p to 20,10
```

4.10 Constructors

R75 Single argument non-converting constructors should use the keyword "`explicit`". This is to avoid possible confusing behaviour with implicit conversions during assignments. A non-converting constructor constructs objects just as converting constructors, but does so only where a constructor call is explicitly indicated by the syntax.

```
Example : For the definition of constructors for Z:
class Z {
public:
  explicit Z(int);
};
```

the following assignment would be illegal (implicit conversion is forbidden):

```
Z a1 = 1;
```

however

```
Z a1(1);
```

would work, as would:

```
Z a1 = Z(1);
Z* p = new Z(1);
```

4.11 Copy Constructors

R76 A copy constructor is recommended when an object is initialised using an object of the same type.

R77 A class that has built-in pointer member data should have a copy constructor and an assignment operator.

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

Without these the default copy constructor and assignment operator would perform shallow copies and so produce two references to the same object. This is one of the easier ways to make a program crash. If you do not think it likely that anyone will want to copy an object of the class you are working on, then just provide private declarations of the copy constructor and assignment operator and the object will be uncopyable.

R78 The argument to a copy constructor and to an assignment operator should be a `const reference`. This ensures that the copying or assigning didn't alter the object.

4.12 Destructors

R79 Declare a "virtual" destructor in every virtual base class (i.e. one having at least one virtual function). This ensures that objects can be properly deleted when referenced by base class pointers.

A virtual base class should always have a virtual destructor.

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

If a class, having virtual functions but without virtual destructors, is used as a base class, there may be a surprise if pointers to the class are used. If such a pointer is assigned to an instance of a derived class and if delete is then used on this pointer, only the base class' destructor will be invoked. If the program depends on the derived class' destructor being invoked, the program will fail.

4.13 Memory Management

R80 Match every invocation of "`new`" with exactly one invocation of "`delete`". This is a minimum requirement to keep control of memory.

R81 Any pointers to automatic objects must have the same, or a smaller scope than the object they point to. This makes sure that when the object goes out of scope and is destroyed, the pointer is not still trying to point to it.

R82 A function must not use the "delete" operator on any pointer passed to it as an argument. This is also to avoid dangling pointers, i.e. pointers to memory, which has been given back. Such code will often continue to work until the memory is re-allocated for another object.

R83 Use "`new`" and "`delete`" instead of "`malloc( )`" and "`free( )`".

4.14 User Defined Operators

R84 To behave as expected, the assignment operator functions should return a reference to their left operand.

```
Example : a = b = c;
```

will assign `c` to `b` and then `b` to `a` as is the case with built in objects.

R85 Give operators conventional definitions. For example if you define the operator "+", then it should do something like an addition.

R86 Declare symmetric binary operators as global functions.

R87 Define asymmetric binary operators and unary operators that modify their operand as member functions.

4.15 Use of the Pre-processor

R88 Do not use "#define" to define constants. Use rather "`enum`" to define integer constants or sets of constants, or "`const`" for non-integer constants.

R89 Use templates for parameterised types rather than the pre-processor.

R90 Use functions (maybe inline) rather than pre-processor macros.

4.16 Common Pitfalls

R91 In a comparison specify the const item first, since this will give rise to a compile-time error if an assignment is specified by mistake

```
Example: if( 0 == value )
```

The following will not be trapped by the compiler

```
if( value = 0) ...
```

when what you meant was

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

R92 Take care when testing floating point values for equality. It is better to use:

```
#include <math.h>
if ( fabs(value1 - value2) < 0.001 ) ...
```

than

```
if ( value1 == value2 ) ...
```

R93 In "switch" statements each choice must have a closing "break", or there should be a comment to indicate that the fall-through is the desired action.

```
switch( expression ) {
  case constant:
    statements;
    break;
  default:
    statements;
    break;
}
```

R94 Avoid declarations of variables inside statements. Rather use

```
int i;
while( i = ... ) {...}
```

than

```
while( int i = ... ) {...}
```

R95 Use the integer constant 0 (zero) for the null pointer. Never implicitly compare pointers to non-zero (i.e. do not treat them as having a boolean value). Use

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

R96 If you are doing an assignment in a comparison expression, make the comparison explicit:

```
while ( 0 != (ptr=iterator() ) )
```

5 Page Layout

The rules in this chapter exist in order to improve readability of the C++ code.

R97 Use short lines (maximum 80 characters long). Long lines can always be arranged on multiple lines. Use spaces and parenthesis in expressions.

R98 Indent (2 spaces are recommended) each nested block in order to highlight the start and end of each of them. Avoid here the tab character (other people are unlikely to use "your" tab definition).

R99 Avoid complicated "if" constructs, rather use several simpler nested "if" constructs. The same logic can be applied to a switch statement

R100 Use braces even if the body of a control statement contains only one statement

```
if( 1 > x ) {
  cout << "single statement" << endl;
}
```

or

```
if( 1 > x ) { cout << "single statement" << endl; }
```

R101 We recommended the following style for a function declaration, since it allows space for a comment describing each argument:

```
int myFunction( int intValue,
                char* charPointerVaue,
                int* intPointerValue,
                MyClass* myClassPointerValue );
```

6 Examples

The following examples can be obtained from the directory: /afs/cern.ch/...

or from the WWW: http://wwwcn.cern.ch/~binko/...

Sample programs illustrating recommended coding styles are also located in: /afs/cern.ch/..

6.1 Example of an Include File

```
#ifndef XYZPATH_H
#define XYZPATH_H
//////////////////////////////////////////////////////////////
//
// XyzPath.h
//
// Project: LHCb Detector System Xyz
// Description: The class XyzPath defines ...
//
// Author : Pavel Binko, 07/05/98
// Changes: Pavel Binko, 12/05/98
//          New functionality "abc" added, bug "klm" removed
//
//////////////////////////////////////////////////////////////

#include "LHCb/LHCb.h"
//--------------------------------------------------------------
class XyzPath {
public:
  XyzPath();
  virtual ~XyzPath();

protected:
  void draw();
private:
  class Internal {
  // XyzPath::Internal declarations Ö
  };
};
//--------------------------------------------------------------

#endif  // XYZPATH_H
```

6.2 Example of an Implementation File

```
//--------------------------------------------------------------
//
// XyzPath.cpp
//
// Package: Package Name / Package Abbreviation
//
// Author : Pavel Binko, 07/05/98
//
//
#include "Xyz/XyzPath.h"
//
// Function definitions of the class Internal
//
//--------------------------------------------------------------
// Description of the function internalFunction
//--------------------------------------------------------------
XyzPath::Internal::InternalFuction( ) {
}


//
// Function definitions of the class XyzPath
//
//--------------------------------------------------------------
// Description of the function draw
//--------------------------------------------------------------
XyzPath::InternalFuction( ) {
}
//--------------------------------------------------------------
```

7 Training

7.1 Courses at CERN

CERN offers many courses on object-oriented analysis and design, and on object-oriented programming languages. The list of courses (in the recomended order in which they should be followed) is as follows:

- Introduction to Software Engineering
- Object Oriented Analysis and Design
- C++ for Particle Physicists (This is the very popular course given by Paul Kunz)
- Hands-on Object Oriented Design and Programming for Software Developers
- C++ Programming for Software Developers
- Objectivity / DB for C++ Developers
- Overview of the ESA Software Engineering Standards

There are numerous other courses on particular technologies, such as CORBA, on the Java programming language etc. The complete list can be found via the web :

`http://www.cern.ch/Training/ENSTEC/P9798/Software/content.htm`

C++ for Particle Physicists is the very popular course by Paul Kunz. Copy of his transparencies can be obtained from :

`/afs/slac.stanford.edu/public/users/pfkeb/c++class/session0n.ps.Z`
where "n" ranges from 1-6.

7.2 Books

Some books on the C++ language and object-oriented analysis and design are listed in this section. Many of them are available in UCO. Some of them are available in our LHCb computing library in Bat. 2, R-008.

7.2.1 Introductory Books on C++

- S.B. Lippman, C++ Primer, Addison Wesley
- J.J. Barton, L.R. Nackman, Scientific and Engineering C++, Addison Wesley
- I. Pohl, Object-Oriented Programming Using C++, Benjamin Cummings

7.2.2 Intermediate Books on C++

- R.B.Murray, C++ Strategies and Tactics, Addison Wesley
- Scott Meyers, Effective C++, Addison Wesley
- Scott Meyers, More Effective C++, Addison Wesley

7.2.3 Books on Object Oriented Programming

- G.Booch, Object-Oriented Analysis and Design, Addison Wesley
- Bjarne Stroustrup, The C++ Programming Language, Addison Wesley
- D.R. Musser, A. Saini, STL Tutorial and Reference Guide, Addison Wesley
- E. Gamma, et al., Design Patterns, Elements of reuseable object-oriented software, Addison Wesley
- Taligent, Taligent's Guide to Designing Programs, Prentice Hall
- J.O.Coplien, Advanced C++ Programming Style and Idioms, Addison Wesley

7.2.4 Software Engineering Books

- G.Booch, Object Solutions, Addison Wesley
- R. Martin, Designing OO C++ Applications using the Booch Method, Prentice Hall
- I. White, Using the Booch method: a Rational approach, Benjamin Cummings
- C.Mazza, J.Fairclough et al., Software Engineering Standards, Prentice Hall

Appendix A: Terminology

1. **Identifier** is a name, which is used to refer to a variable, constant, function or type in C++.
2. **Class** is a user-defined data type, which consists of data elements and functions, which operate on that data. Data defined in a class are called member data and functions defined in a class are called member functions.
3. **Abstract class** is a class, which does not have any public or protected member data.
4. **Public members** of a class are member data and member functions, which are everywhere accessible by specifying an instance of the class and their name.
5. **Protected members** of a class are member data and member functions, which are accessible by specifying their name within member functions of derived classes.
6. **Private members** of a class are member data and member functions, which are accessible by specifying their name only within the classes they are defined in.
7. **Enumeration type** is an explicitly declared set of symbolic integral constants. In C++ it is declared as an "enum".
8. **Typedef** is another name for a data type, specified in C++ using a "typedef" declaration.
9. **Reference** is another name (alias) for a given variable. In C++, the "address of" ("&") operator is used immediately after the data type to indicate that the declared variable, constant, or function argument is a reference.
10. **Macro** is a name for a text string, which is defined in a "#define" statement. When this name appears in source code, the compiler replaces it with the defined text string.
11. **Constructor** is a function which initialises an object.
12. **Copy constructor** is a constructor in which the first argument is a reference to an object that has the same type as the object to be initialised.
13. **Default constructor** is a constructor, which needs no arguments.
14. **Overloaded function name** is a name, which is used for two or more functions or member functions having different types. (The type of a function is given by its return type and the type of its arguments.)
15. **Overridden member function** is a member function in a base class, which is re-defined in a derived class. Such a member function is declared virtual.
16. **Pre-defined data type** is a type, which is defined in the language itself, such as int.
17. **User-defined data type** is a type, which is defined by a programmer in a class, enum, or typedef definition or as an instantiation of a class template.
18. **Pure virtual function** is a member function, for which no definition is provided. Pure virtual functions are specified in abstract base classes and must be defined (overridden) in derived classes.
19. **Accessor** is a function, which either returns the value of a data member, or sets its value.
20. **Forwarding function** is a function, which does nothing more than call another

function.

21. **Constant member function** is a function, which may not modify any data members.
22. **Exception** is a run-time program anomaly that is detected in a function or member function. Exception handling provides for the uniform management of exceptions. When an exception is detected, it is thrown (using a throw expression) to the exception handler.
23. **Catch clause** is code that is executed when an exception of a given type is raised. The definition of an exception handler begins with the keyword "catch".
24. **Abstract base class** is a class from which no objects may be created; it is only used as a base class for the derivation of other classes. A base class is abstract either if it includes at least one member function that is declared as pure virtual, or its default constructor is private or protected.
25. **Iterator** is an object which, when invoked, returns the next object from a collection of objects.
26. **Scope of a name** refers to the context in which it is visible. (Context, here, means the functions or blocks in which a given name can be used.)
27. **Compilation unit** is the source code (after pre-processing) that is submitted to a compiler for compilation (including syntax checking).
28. An **Include file** contains declarations of the class and, when appropriate, the declarations of in-line functions.
29. An **implementation file** contains the implementation of a class. This includes the implementation of all non in-line functions.

References

[1] David R. Quarrie, Guidelines for Writing or Modifying BaBar Software, 1996 http://www.slac.stanford.edu/BFROOT/doc/Programming/Guidelines/Guidelines.html

[2] Neil Geddes, The BaBar User Guide, 1998 http://www.slac.stanford.edu/BFROOT/doc/Computing/NewUser/htmlbug/

[3] S.M.Fisher, C++ Coding standards for ATLAS, http://atlasinfo.cern.ch/Atlas/GROUPS/SOFTWARE/NOTES/note29/cxx-rules.html

[4] CMS : A suggested "training path" in C++ and Object-Orientation, http://hpl3sn02.cern.ch/homepages/innocent/cmsoo/training.html

[5] Erik Nyquist, Mats Henricson, Programming in C++ Rules and Recommendations, Ellemtel, 1990-92

[6] Barton and Nackman, Scientific and Engineering C++, Addison-Wesley, 1994

[7] Gamma et al., Design Patterns, Addison-Wesley, 1995

[8] Taligent, Well-mannered object-oriented design in C++, http://hpsalo.cern.ch/TaligentDocs/TaligentOnline/DocumentRoot/1.0/Docs/index.html

[9] C++ ANSI(draft) standard public review document : http://www.maths.warwick.ac.uk/cpp/pub/wp/html/cd2/index.html

[10] LHCb Configuration Management Plan (in preparation)