ATLAS Quality Assurance Group

ATLAS C++ Coding Standard

Specification

Version: 1.2
Issue: 1
Status: FINAL
ID: ATL-SOFT-2002-001
Date: 2003-07-11

European Laboratory for Particle Physics
Laboratoire Européen pour la Physique des Particules
CH-1211 Genève 23 - Suisse

Abstract

This document defines the ATLAS C++ coding standard, that should be adhered to when writing C++ code.

It has been adapted from the original “PST Coding Standard” document (http://pst.cern.ch/HandBookWorkBook/Handbook/Programming/programming.html) CERN-UCO/1999/207.

The “ATLAS standard” comprises modifications, further justification and examples for some of the rules in the original PST document. All changes were discussed in the ATLAS Offline Software Quality Assurance Group (formerly known as the Quality Control Group) and feedback from the collaboration is taken into account in the final version.

Document Control Sheet

Document	**Title:**	ATLAS C++ Coding Standard Specification		
	Version:	1.2	**ID:**	ATL-SOFT-2002-001
	Issue:	1	**Status:**	FINAL
			Created:	2000-01-05
			Date:	2003-06-16
	Available at:	http://documents.cern.ch/		
	Keywords:	ATLAS coding standard, C++		
Tool	**Name:**	Adobe FrameMaker	**Version:**	6.0p405
	Template:	Software Doc Layout Templates	**Version:**	Vb1 - 16 November 1998
Authorship	**Written by:**	The former Spider group and ATLAS QC group		
	Contributors:	M. Asai, D. Barberis, M. Bosman, B. Jones, J-F. Laporte, M. Stavrianakou, C. Arnault, D. Candlin, R. Candlin, S.M. Fisher, E. Frank, T. Hansl-Kozanecka, D. Malon, S. Qian, D. Quarrie, RD Schaffer, P. Calafiura, M. Marino.		
	Coordinator	S. Albrand (from 2001-09-01)		
	Reviewed by:			
	Approved by:			

Document Status Sheet

Title:	ATLAS C++ Coding Standard Specification		
ID:	ATL-SOFT-2002-001		
Version	**Issue**	**Date**	**Reason for change**
0.1	0	2000-08-01	Release to QC group for review
0.2	0	2000-12-10	Release to a broader QC group for review
0.2	1	2001-01-19	Release to a broader QC group for review
1.0	0	2001-01-30	Release to public
1.1	0	2002-01-06	Release to original document authors
1.1	1	2002-03-05	Release to ATLAS developers
1.1	2	2002-03-27	Release to public
1.2	0	2003-06-16	Release to Atlas A-team and Atlas-SIT
1.2	1	2003-07-11	Release to ATLAS developers

Document Change Record

Title:	ATLAS C++ Coding Standard Specification	
ID:	ATL-SOFT-2002-001	
Version:	1.2	
Date:	2003_06_16	
Page	**Paragraph**	**Reason for Change**
all pages		Added label to rules which are checked by the rulechecker 2.0
page 2	1.3.1	Added this section
page 4	1.5	Added reference to Design Patterns, and Effective STL
page 8	NC1, NC2, NC6	Made clearer that rule NC6 applies to NC1 and NC2
page 13	CO2	Added reference to detailed explanation
page 14	CO5	New rule on the ordering of include files.
page 16	CL2	Some very common constants can be literal
page 18	CL5	Minor corrections to bring the text into line with the example.
page 22	CL11	Dropped this rule, because it is a question of design rather than code.
page 25	CI7	Wording changed to allow singleton pattern exception, and mention the case of iterators.
page 26	CI8	Wording changed to allow superclass exception. (so that Gaudi algorithms do not trigger rule violations)
page 31	CH1	Explanation expanded
page 38	CR3	Tightened rule on typedefs
page 40	CP10	Indexes for loops are excluded from the check
page 42	CT3	Testing templates
page 44	SG7	C++ keywords are exceptions to the rule.
page 45	SC1	Changed to be consistent with Doxygen use

Table of Contents

1 Introduction

1.1 Purpose

The purpose of this document is to define a C++ coding standard that ATLAS offline software developers should adhere to. The standard provides indications aimed at helping C++ programmers to meet the following requirements on a program:

- be free of common types of errors
- be maintainable by different programmers
- be portable to other operating systems
- be easy to read and understand
- have a consistent style

Questions of design are beyond the scope of this document. It is also assumed that the code is hand-written and not generated; otherwise a different standard would be needed for the input to the code generator.

Disclaimer: This document does not substitute in any way the study of a book on C++ programming.

1.2 Authors

This document has been adapted from the original “PST Coding Standard” document (http://pst.cern.ch/HandBookWorkBook/Handbook/Programming/programming.html).

The “ATLAS standard” comprises modifications, further justification and examples for some of the rules in the original PST document. All changes were discussed in the ATLAS Offline Software Quality Control Group and feedback from the collaboration was taken into account in the “final” version.

The original “PST Coding Standard” document originated in the context of the SPIDER project, where representatives from different experiments (ALICE, ATLAS, LHCb, CMS and COMPASS) led by the IT/IPT group established and propose a common standard across experiments/projects.

1.3 Evolution and updating responsibility

Changes to this standard will be implemented when necessary, following a procedure established by the ATLAS Offline Software Quality Assurance Group, formerly known as the

ATLAS Offline Software Quality Control Group, which is responsible for the maintenance of the document.

1.3.1 Motivations for the evolution of the document

The tool “rulechecker”[1] has been in use for Atlas code since December 2002. Also human review of Atlas code has started and is expected to increase.

Experience with applying the Atlas coding rules both with tools and by reviewers has lead us to rewrite several rules to make them clearer. In several cases the wording of the rule was not sufficiently unambiguous for the implementation of an automatic check.

Rules checked by rulechecker are marked with “**RC**” in the left hand margin.

1.4 Organization of this document and approach

This document is organized as follows:

- Chapter 1: Introduction - this chapter
- Chapter 2: Naming - list of all items on naming, with explanation and examples
- Chapter 3: Coding - list of all items on coding, with explanation and examples
- Chapter 4: Style - list of all items on style, with explanation and examples
- Appendix A: Terminology
- Appendix B: List of the items in the standard

1.4.1 Naming

This section contains indications on how to choose names for all entities over which the programmer has control, e.g. classes, typedefs, functions, variables, namespaces, files.

1.4.2 Coding

Indications in this section regard the syntax and related semantic of the code. Organization of code, control flow, object life cycle, conversions, object-oriented programming, error handling, parts of C++ to avoid, portability, are all examples of issues that are covered here. This section is organized in different paragraphs, each one grouping items addressing the same subject.

1. The tool is produced by the Istituto Trentino di Cultura, Trento, Italy http://www.itc.it/

1.4.3 Style

Code is always written in a particular style. This section contains indications aimed at defining one, that should allow a common and consistent look and feel of the code. Style relates to matters which do not affect the output of the compiler.

1.4.4 Item Numbering

Each item comprises at least two entities, an identifier and an item title. The identifier is formed by two letters and a number (e.g. NF3); the first letter (N, C or S) indicates to which section (Naming, Coding or Style) the item belongs, the second letter indicates the subsection, while the number represents the order within the subsection. This kind of identification should allow a minimal impact on the item numbering during the maintenance of the standard, that is in the cases where items are added or removed.

Table 1 List of Item Identifiers

Identifier	Type of rule
NF	File naming
NM	General naming
NI	Illegal naming
NC	Naming conventions
CO	Code organization
CF	Control flow
CL	Object life cycle
CC	Conversions
CI	Class interface
CN	`new` and `delete`
CS	Static and global objects
CB	Object oriented programming
CE	Assertions and errors
CH	Error handling
CA	Parts of C++ to avoid
CR	Readability and maintenance

Table 1 List of Item Identifiers

Identifier	Type of rule
CP	Portability
CT	Templates
SG	General style
SC	Comments

1.4.5 Item Information

Whenever possible and appropriate, a statement and an example have been added to the individual item; the statement is an explanation that expands the item and clarifies its meaning and scope.

Items have been marked with the attribute REQUIRED or RECOMMENDED. This is an attempt to indicate to developers and to reviewers, what their attitude should be to each item of the standard. An item marked “recommended” is not to be considered as obligatory. It is used to mark an item which is “desirable”, or “should be avoided”, for example, in the interests of a coherent style or of readability. An item which is marked as “required” is something that all developers MUST try to follow. This does not mean that a reviewer should never allow an exception, indeed possible exceptions are sometimes mentioned in the text, but that if a developer wishes to “transgress” one of these items, he or she must be prepared to justify to the reviewer, and to copiously comment within the code, that there was no other way both possible and acceptable, to obtain the desired result.

1.5 References

1 *C++ Coding Standard - Specification, S. Paoli, P. Binko, D. Burckhart, S.M. Fisher, I. Hrivnacova, M. Lamanna, M. Stavrianakou, H.-P. Wellisch CERN-UCO/1999/207, 20 October 1999.*

2 *Standard for the Programming Language C++, ISO/IEC 14882.*

3 *Effective C++, 2nd Edition, S. Meyers, Addison-Wesley.*

4 *Design Patterns, Elements of Reusable Object-Oriented Software, Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Addison-Wesley.*

5 *Effective STL, 2nd Edition, S. Meyers, Addison-Wesley 2001 ISBN 0-201-74962-9*

1.6 Definitions and Acronyms

SPIDER Software Process Improvement for Documentation, Engineering, and Reuse of LHC and HEP Software Systems, Applications and Components

Item Single statement addressing a specific issue (other terms typically used for that are: rule, guideline, convention, recommendation). In this document items are either REQUIRED or RECOMMENDED.

Standard Collection of items addressing the same subject (in this case *coding of C++ software*)

2 Naming

This section contains a set of conventions on how to choose, write and administer names for all entities over which the programmer has control.

2.1 Naming of files (NF)

NF1 **The name of the header file must be the same as the name of the class it defines, with a suffix ".h" appended. *(REQUIRED)***

RC

Example:

The header file for the class CalorimeterCluster would have the name CalorimeterCluster.h

In the presence of name space, the file name should not include the name space prefix (e.g. for a class Emc::Track, the name of its header file should be Track.h).

NF2 **The name of the implementation file must be the same as the name of the class it implements, with a suffix ".cxx" appended. *(REQUIRED)***

RC

Example:

The implementation file for the class CalorimeterCluster would have the name CalorimeterCluster.cxx

2.2 Meaningful Names (NM)

NM1 **Use pronounceable English words or acronyms widely used in the experiment to compose all lexical entities except for local loop variables and array indices. *(RECOMMENDED)***

They aid discussion, and are helpful for newcomers.

Example:

Use `nameLength` instead of `nLn`.

Use names that are English and self-descriptive. If abbreviations and/or acronyms cannot be avoided, then they are better to be agreed upon by the communities concerned and be meaningful, uniform and documented.

This is very important for everyone to understand the meaning of the declared entities and to make the code easy to be read and used.

2.3 Illegal or Non-recommended Naming (NI)

NI1 **Do not create very similar names. *(RECOMMENDED)***

RC Very similar names might cause confusion in reading the code.

In particular do not create names that differ only by case.

Example:

```
track, Track, TRACK
cmlower, cslower
```

NI2 **Do not use identifiers that begin with an underscore. *(REQUIRED)***

RC Names starting with an underscore are used in C and C++ compiler implementation. To allow users to use such names introduces a needless confusion.

2.4 Naming Conventions (NC)

NC1 **Use prefix “m_” for private attributes (i.e. data members) in each class. *(REQUIRED)***

RC Start names of variables and functions with a lowercase letter after the prefix “m_”

NC2 **Use prefix “s_” for private static attributes (i.e. class data members). *(RECOMMENDED)***

RC Start names of variables and functions with a lowercase letter after the prefix “s_”

NC3 **The choice of prefixes for package names and the choice of name for namespaces should be agreed upon by the communities concerned and should, where appropriate, correspond to the Product Breakdown Structure (PBS). *(RECOMMENDED)***

NC4 **Use namespace to avoid name conflicts between classes. *(REQUIRED)*.**

A name clash occurs when a name is defined in more than one place. For example, two different class libraries could give two different classes the same name. If you try to use many class libraries at the same time, there is a fair chance that you will be unable to compile and link the program because of name clashes. To solve the problem you can use a namespace.

Example of the declaration of a namespace:

```
namespace Emc {
   class Track { ... };
   // ...
}
```

A namespace is a declarative region in which classes, functions, types and templates can be defined. A namespace should be defined at the package level to put in common what is necessary for classes of the package, and to render these common elements accessible for client packages. Thus a good name for the namespace is the name of the package to which it belongs. (see rule NC3)

A name qualified with a namespace name refers to a member of the namespace.

```
Emc::Track electronTrack;
```

A `using` declaration makes it possible to use a name from a namespace without the scope operator.

```
using Emc::Track;        // using declaration
Track electronTrack;
```

Recommendations for the use of namespaces based on proposals by L. Tuura

Using directives (e.g. '`using namespace std;`') can be used only in **function** scope or to import names from one named namespace to another named namespace.

Examples of allowed usage:

```
void MyClass::MyMethod()
{
using clhep::units::mm;    // function scope
// do something
}
namespace foo { using bar::foobar;  // import to named namespace }
```

"`using`" directives are forbidden everywhere else.

Examples of forbidden usage:

```
using std::operator!=;  // at file scope
namespace { using foo::bar;  // import to anonymous namespace }
```

Note: "`using`" declarations cannot be used in classes, where they have a completely different interpretation anyway (for example to access from a derived class a same name member of a base class or to restore public accessibility of a base class method when private inheritance is used -- for further details see References [1] and [2]).

Whatever the conventions adopted (depending on project policies and practices), it is essential that the style be consistent and documented (especially if it does not follow the "official" rules).

NC5 **Start class names, `typedefs` and `enum` types with an uppercase letter. *(REQUIRED)***

RC

Example:

```
class Track;
typedef vector<MCParticleKinematics*>  TrackVector;
enum State { green, yellow, red };
```

NC6 **Start names of variables and functions with a lowercase letter. *(REQUIRED)***

RC

Example:

```
double energy;
void extrapolate();
```

In the cases of NC1 and NC2 (i.e. for the private attributes), start names of variables and functions with a lowercase letter after the prefix ("s_" or "m_").

NC7 **In names that consist of more than one word, write the words together, and start each word that follows the first one with an upper case letter. *(RECOMMENDED)***

RC

Example:

```
class OuterTrackerDigit;
double depositedEnergy;
void findTrack();
```

This is a change from the previous Atlas coding rules which recommended a different convention (i.e. using the underscore between words). This new rule is based on the STL style. Existing code may not need to be changed; however, when a package has a serious overhaul, you should switch to this new rule. Clearly, when using other classes (including STL) one has to accept the name of the classes and methods which are provided. More suggestions are

(1) a class should (apart from STL) be written entirely in one convention or another;

(2) where classes in existing packages are mostly written with the old convention, new classes added to these packages should also use underscores;

(3) where an existing package contains a mixture, the package responsible should decide whether it is worth converting or whether to live with the mixture.

NC8 All package names in the release must be unique, independent of the package's location in the hierarchy. *(REQUIRED)*.

Here a "package" is a collection of codes which are related to each other.

Thus, if there is a package, say "offline/A/B/C", already existing, another package may not have the name "offline/D/E/C" because that "C" has already been used. This rule maintains our ability to flatten the structure in the future, if needed.

NC9 Underscores should be avoided in package names. *(RECOMMENDED)*

The old Atlas rule was that the "_" should be used in package names when they are composites of one or more acronyms, e.g. TRT_Tracker, AtlasDB_*.

Underscores should be avoided unless they really help with readability and help in avoiding spelling mistakes. TRTTracker looks odd because of the double "T". Using underscores in package names will also add to confusion in the multiple-inclusion protection lines (see rule CO1)

3 Coding

This section contains a set of items regarding the “content” of the code. Organization of the code, control flow, object life cycle, conversions, object-oriented programming, error handling, parts of C++ to avoid, portability, are all examples of issues that are covered here.

The purpose of the following items is to highlight some useful ways to exploit the features of the programming language, and to identify some common or potential errors to avoid.

3.1 Organizing the Code (CO)

CO1 **Header files must begin and end with multiple-inclusion protection. *(REQUIRED)***

RC Example of implementation:

```
#ifndef IDENTIFIER_H
#define IDENTIFIER_H
// The text of the header goes in here ...
#endif // IDENTIFIER_H
```

The actual value for the IDENTIFIER is the class name converted to upper case proceeded by the package name.

The current scheme in ATLAS is package name + “_” + class name (where the names are in upper case and for composite names “_” are inserted as separators e.g. FooBar => FOO_BAR).

Header files are often included many times in a program. Because C++ does not allow multiple definitions of a class, it is necessary to prevent the compiler from reading the definitions more than once.

CO2 **Use forward declaration instead of including a header file, if this is sufficient. *(RECOMMENDED)***

RC Example:

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

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

A detailed explanation of how the rulechecker tool implements the checking of this rule can be found at

`http://atlas.web.cern.ch/Atlas/GROUPS/SOFTWARE/OO/Development/qa/Tools/CO2.html`

CO3 **Each header file must contain one class (or embedded or very tightly coupled classes) declaration only. *(REQUIRED)***

RC

This makes your source code files easier to read. This also improves the version control of the files; for example the file containing a stable class declaration can be committed and not changed any more.

CO4 **Implementation files must hold the member function definitions for a single class (or embedded or very tightly coupled classes) as defined in the corresponding header file. *(REQUIRED)***

RC

This is for the same reason as for item CO3.

CO5 **Ordering of #include statements. *(RECOMMENDED)***

For ease of reading and to provide for a visual taxonomy of dependencies favor a grouping style of #include statements into three categories, in both the declaration and definition units for a class. The three categories are system, extra-package and intra-package dependencies. Note that this does not address the issue of reducing dependencies which should always be the first priority.

3.2 Control Flow (CF)

CF1 **Do not change a loop variable inside a `for` loop block. *(REQUIRED)***

When you write a `for` loop, it is highly confusing and error-prone to change the loop variable within the loop body rather than inside the expression executed after each iteration.

CF2 **All `switch` statements must have a `default` clause. *(REQUIRED)***

In some cases the default clause can never be reached because there are `case` labels for all possible `enum` values in the `switch` statement, but by having such an unreachable `default` clause you show a potential reader that you know what you are doing.

You also provide for future changes. If an additional `enum` value is added, the `switch` statement should not just silently ignore the new value. Instead, it should in some way notify the programmer that the `switch` statement must be changed; for example, you could throw an exception

CF3 **Each clause of a `switch` statement must end with "break". *(REQUIRED)***

Example:

CF2, CF3 joint example

```
// somewhere specified: enum Colors { GREEN, RED }

// semaphore of type Colors

switch(semaphore) {
   case GREEN:
      // statement
      break;
   case RED:
      // statement
      break;
   default:
      // unforseen color; it is a bug
      // do some action to signal it
}
```

CF4 **An "`if` statement" which does not fit in one line must have brackets around the conditional statement. *(REQUIRED)***

This makes code much more readable and reliable, by clearly showing the flow paths.

The addition of a final `else` is particularly important in the case where you have `if`/`else-if`. To be safe, even single statements should be explicitly blocked by {}.

Example:

```
if (val==thresholdMin) {
   statement;
} else if (val==thresholdMax) {
   statement;
} else {
   statement;       // handles all other (unforseen)cases
}
```

CF5 **Do not use `goto`. *(REQUIRED)***

RC Use `break` or `continue` instead.

This statement remains valid also in the case of nested loops, where the use of control variables can easily allow to break the loop, without using goto.

`goto` instructions decrease readability and maintainability and make testing difficult by increasing the complexity of the code.

`goto` instructions may also affect the optimisation by the compiler (loop-unrolling) and this sometimes results in severe performance penalties.

3.3 Object Life Cycle (CL)

In this paragraph it is suggested how objects are best declared, created, initialized, copied, assigned and destroyed.

3.3.1 Initialization of Variables and Constants

CL1 **Declare each variable with the smallest possible scope and initialise it at the same time. *(RECOMMENDED)***

It is best to declare variables close to where they are used. Otherwise you may have trouble finding out the type of a particular variable.

It is also very important to initialise the variable immediately, so that its value is well defined.

Example:

```
int value = -1;    // initial value clearly defined
int maxValue;      // initial value undefined, NOT recommended
```

CL2 **Do not use literals in expression. *(RECOMMENDED)***

Instead, declare and initialise a const variable and use that variable in subsequent expression. Exceptions to the rule are the constants {0,1, true, false ,““}.

Example:

CL1, CL2 joint example

```
#include <iostream>
#include <string>
...
// declare variables initialised to numeric values or strings
// in a highly visible position
// whenever possible collected in one place

const string myString = "GoodMorning";
const int numberOfRepetitions = 17;
...
// use symbolic names; do not use numeric values or strings ===>
// we use "numberOfRepetitions" and "myString" rather than the
// actual values AND
// declare each variable with the smallest possible scope
// and initialise it at the same time
for (int i=0; i<numberOfRepetitions; i++) {
      cout << myString << endl;
...
}

// do not recommend the following
for (int i=0; i<17; i++) { ... }
```

CL3 **Declare each type of variable in a separate declaration statement, and do not declare different types (e.g. int and int pointer) in one declaration statement. *(REQUIRED)***

Declaring multiple variables on the same line is not recommended. The code will be difficult to read and understand.

Some common mistakes are also avoided. Remember that when you declare a pointer, a unary pointer is bound only to the variable that immediately follows.

Example:

```
int i, *ip, ia[100], (*ifp)();   // Not recommended

// recommended way:
LoadModule* oldLm = 0;      // pointer to the old object
LoadModule* newLm = 0;      // pointer to the new object
```

Bad example: both ip and jp were intended to be pointers to integers but only ip is -- jp is just an integer!

```
int* ip, jp;
```

CL4 **Do not use the same variable name in outer and inner scope. *(REQUIRED)***

Otherwise the code would be very hard to understand; and it would certainly be very error prone.

3.3.2 Constructor Initializer Lists

CL5 **Let the order in the initializer list be the same as the order of the declarations in the header file: first base classes, then data members. *(RECOMMENDED)***

It is legal in C++ to list initializers in any order you wish, but you should list them in the same order as they will be called.

The order in the initializer list is irrelevant to the execution order of the initializers. Putting initializers for data members and base classes in any order other than their actual initialization order is therefore highly confusing and can lead to errors.

Class members are initialised in the order of their declaration in the class; the order in which they are listed in a member initialisation list makes no difference whatsoever! So if you hope to understand what is really going on when your objects are being initialised, list the members in the initialisation list in the order in which those members are declared in the class.

Here, in the bad example, "m_data" is initialised first (as it appears in the class) BEFORE you initialise "m_size" although you've put it first in the initialiser list ==> you don't know how much memory "m_data" points to!

Bad example:

```
class Array
{
public:
   Array(int lower, int upper);
private:
   int* m_data;
   unsigned m_size;
   int m_lowerBound;
   int m_upperBound;
};
Array::Array(int lower, int upper) : m_size(upper-lower+1),
m_lowerBound(lower), m_upperBound(upper),m_data(new int[size]) { ...
```

Correct example:

```
class Array
{
public:
   Array(int lower, int upper);
private:
   unsigned m_size;
   int m_lowerBound;
   int m_upperBound;
   int* m_data;
};
Array::Array(int lower, int upper) : m_size(upper-lower+1),
m_lowerBound(lower), m_upperBound(upper),m_data(new int[size]) { ...
```

Virtual base classes are always initialized first, then base classes, data members, and finally the constructor body for the derived class is run.

Example:

```
class Derived : public Base {     // Base is number 1
   public:
      explicit Derived(int i);
// The keyword explicit prevents the constructor being called implicitly
// Derived dNew = dOld;
// will not work
      Derived();
   private:
      int m_jM;         // m_jM is number 2
      Base m_bM;        // m_bM is number 3
};

Derived::Derived(int i) : Base(i), m_jM(i), m_bM(i) {
// Recommended order        1        2      3
   ...
}
```

3.3.3 Copying of Objects

CL6 **A function must never return, or in any other way give access to, references or pointers to local variables outside the scope in which they are declared. *(REQUIRED)***

RC

Returning a pointer or reference to a local variable is always wrong because it gives the user a pointer or reference to an object that no longer exists.

Example:

Bad example:

You are using a complex number class, Complex, and you write a method that looks like this:

```
Complex& calculateC1(const Complex& n1, const Complex& n2)
{
   double a = n1.getReal()-2*n2.getReal();
   double b = n1.getImaginary()*n2.getImaginary();

   // create local object
   Complex C1(a,b);

   // return reference to local object
   // the object is destroyed on exit from this function - trouble ahead!
   return C1;
}
```

In fact, most compilers will spot this and issue a warning.

CL7 **If objects of a class should never be copied, then the copy constructor and the copy assignment operator should be declared `private`. *(RECOMMENDED)***

Ideally the question whether the class has a reasonable copy semantic will naturally be a result of the design process. Do not define a copy method for a class that should not have it.

By declaring the copy constructor and copy assignment operator as `private`, you can make a class non-copyable. They do not have to be implemented, only declared.

Example:

There is only one ATLASExperimentalHall that should not be copied and evidently cannot be compared to another one!

```
class ATLASExperimentalHall
{
public:
   ATLASExperimentalHall();
   ~ATLASExperimentalHall();
private:
   // private copy constructor - never allow copying
   ATLASExperimentalHall(const ATLASExperimentalHall& ) {};
   // private assignment operator
   ATLASExperimentalHall operator=(const ATLAS_ExperimentalHall&) {};
};
```

CL8 **If a class has built-in pointer member data then the copy constructor, the copy assignment operator and the destructor should all be implemented. *(RECOMMENDED)***

RC

The compiler will generate a copy constructor, a copy assignment operator, and a destructor if these member functions have not been declared. A compiler-generated copy constructor does memberwise initialization and a compiler-generated copy assignment operator does memberwise assignment of data members and base classes. For classes that manage resources (examples: memory (new), files, sockets) the generated member functions have probably the wrong behaviour and must be implemented by the developer. You have to decide if the resources pointed to must be copied as well (deep copy), and implement the correct behaviour in the operators. Of course, the constructor and destructor must be implemented as well.

Example:

Bad Example:

```
class String
{
public:
   String(const char *value=0);
   ~String(); // destructor but no copy constructor or assignment operator
private:
   char *m_data;
}
String::String(const char *value)
{ // assume correct behaviour implemented in constructor, e.g.
     m_data = new char[strlen(value)]; // fill m_data }
inline String::~String()
{ // assume correct behaviour implemented in destructor, e.g.
     delete m_data; }
```

In your program:

```
// declare and construct a ==> m_data of “a” points to "Hello"
String a("Hello");
// open new scope
{ // declare and construct b ==> m_data of “b” points to "World"
  String b("World");
  b=a;
  // execute default op= as synthesised by compiler ==> member-wise assignment
  // i.e. for pointers  (m_data) bitwise copy
  // ==> m_data of “a” and “b” now point to the same string "Hello"
  // ==> 1) memory b used to point to never deleted ==> possible memory leak
  //     2) when either a or b goes out of scope, its destructor
  //        will delete the memory still pointed to by the other
}
// close scope: b’s destructor called; memory still pointed to by a deleted!
String c=a;    // but m_data of “a” is undefined!!
```

CL10 Assignment member functions must work correctly when the left and right operands are the same object. *(REQUIRED)*

This requires some care when writing assignment code, as the case (when left and right operands are the same) may require that most of the code is bypassed.

Example:

```
A& A::operator=(const A& a) {
  if (this != &a) {    // beware of s=s - "this" and "a" are the same object
    // ... implementation of operator= }
}
```

3.4 Conversions (CC)

CC1 **Use explicit rather than implicit type conversion. *(REQUIRED)***

Most conversions are bad in some way. They can make the code less portable, less robust, and less readable. It is therefore important to use only explicit conversions. Implicit conversions are almost always bad. (but see item **CS2)**

In general, casts should be strongly discouraged, especially the old style C casts must be forbidden.

CC2 **Use the new cast operators (`dynamic_cast` and `static_cast`) instead of the C-style casts. *(REQUIRED)***

RC

The new cast operators give the user a way to distinguish between different types of casts. Their behaviour is well-defined in situations where the behaviour of an ordinary cast is undefined, or at least ambiguous.

The C++ static_cast operator offers the same functionality and restrictions (e.g. you still cannot cast a double into a pointer) as the old C-style casts but it respects constness and is quite visible.

The C++ dynamic_cast operator is used to perform safe casts down or across an inheritance hierarchy. One can actually determine whether the cast succeeded because failed casts are indicated either by a bad_cast exception or a null pointer. The use of this type of information at run time is called Run-Time Type Identification (RTTI).

Example:

```
static_cast
double r = static_cast<double>(n) * a;

dynamic_cast
class Base { };
class Derived : Base { };
void f(Derived* d_ptr)
{
 // if the following cast is inappropriate a null pointer will be returned!
 Base* b_ptr = dynamic_cast<Base*>(d_ptr);
 // ...
}
```

CC3 **Do not convert `const` objects to non-`const`. *(REQUIRED)***

In general you should never cast away the `const`ness of objects.

Exception:

The only rare case when you have to do it is in the case where you need to invoke a function that has incorrectly specified a parameter as non-`const` even if it does not modify it. If the correction of this function is really impossible, then use the cast operator `const_cast`.

Example:

You are using an external package that provides a Polygon class with a positionPolygon method defined like this:

```
void positionPolygon(Polygon* p);
```

You are now using Polygons in your code:

```
const Polygon p;
positionPolygon(&p);
// wrong - positionPolygon incorrectly takes a non-const Polygon*
// (you don't expect the positioning method to change the object it positions)

positionPolygon(const_cast<Polygon*>(&p));
// correct - constness explicitly cast away BUT now Polygon p can be
// inadvertently modified by the positioning...
```

The keyword `mutable` allows data members of an object that have been declared `const` to remain modifiable, thus reducing the need to cast away constness. The mutable should only be used for variables which are used for caching information. In other words the object appears not to have changed but it has stored something to save time on subsequent use.

CC4 Do not use `reinterpret_cast`. ***(REQUIRED)***

RC `reinterpret_cast` is machine-, compiler- and compile-options-dependent. It is a way of forcing a compiler to accept a type conversion which is dependent on implementation. It blows away type-safety, violates encapsulation and more importantly, can lead to unpredictable results.

Exception:

`reinterpret_cast` is required in some cases if one is not using old-style casts. It is required for example if you wish to convert a callback function signature (X11, expat, unix signal handlers are common causes). Some external libraries (X11 in particular) depend on casting function pointers. If you absolutely have to work around limitations in external libraries, you may of course use it.

3.5 The Class Interface (CI)

3.5.1 Inline Functions

CI1 **Header files must contain no implementation except for small functions to be inlined. These inlines must appear at the end of the header after "};" of the class definition. *(REQUIRED)***

RC

Inline functions can improve the performance of your program; but they can increase the overall size of the program and then, in some cases, have the opposite result. It can be hard to know exactly when inlining is appropriate. In general, inline only very simple functions and only those functions called frequently.

Use of inlining makes debugging hard and, even worse, can force a complete release rebuild or large scale recompilation if the inline needs to be changed.

3.5.2 Argument Passing and Return Values

CI2 **Functions which have a return value should not modify their arguments nor have any side effects. (*RECOMMENDED)***

RC

Example:

No one would expect the x to be modified when calling sin(x).

CI3 **(1)Pass an unmodifiable argument by value only if it is of built-in type or small;**
(2)otherwise, pass the argument by const reference or by pointer to const. *(RECOMMENDED)*

RC

An object is considered small if it is a built-in type or if it contains at most one small object. Built-in types such as `char`, `int`, and `double` can by value because it is cheap to copy such variables. If an object is larger than the size of its reference (typically 32 bits), it is not efficient to pass it by value. Of course a built-in type **can** be passed by reference when appropriate.

Example:

```
void func(char c);              // OK
void func(int i);               // OK
void func(double d);            // OK
void func(complex<float> c);    // OK

void func(Track t);             // not good, since Track is large, thus
                                // there is an overhead in copying t.
```

(2) Arguments of class type are often costly to copy, so it is convenient to pass a reference (or in some cases a pointer), preferably declared `const`, to such objects; in this way the argument is not copied. `Const` access guarantees that the function will not change the argument.

Example:

```
void func(const LongString& s);   // const reference
```

However, besides the speed consideration, pass by reference, pass by value and pass by pointer represent different design issues. There are times when speed is NOT the key issue, but clarity or other aspects of design are. For example, to some people, a call by reference vs. pointer connotes aspects of ownership designs.

CI4 **Have `operator=` return a reference to `*this`. *(REQUIRED)***

RC This ensures that

```
a = b = c;
```

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

3.5.3 `const` Correctness

CI5 **Declare a pointer or reference argument, passed to a function, as `const` if the function does not change the object bound to it. *(RECOMMENDED)***

RC

An advantage of `const`-declared parameters is that the compiler will actually give you an error if you modify such a parameter by mistake, thus helping you to avoid bugs in the implementation.

Example:

```
// operator<< does not modify the String parameter
ostream& operator<<(ostream& out, const String& s);
```

CI6 **The argument to a copy constructor and to an assignment operator must be a `const` reference. *(REQUIRED)***

RC

This ensures that the object being copied is not altered by the copy or assign.

CI7 **In a class method, do not return pointers or non-`const` references to private data members. *(REQUIRED)***

RC

Otherwise you break the principle of encapsulation.

If necessary, you can return a pointer to a `const` or `const` reference.

Remember that an iterator is NOT a general pointer, but an object (see reference [1] §19.2). It is acceptable and even recommended to prefer iterators to const_iterators, because some container member functions have iterators as arguments. see (Item 26 of reference [5])

An allowed exception to this rule if the use of the singleton pattern. The example of a singleton pattern given in [3] shows a non const reference to a static object being returned, whereas the example in [4] shows a pointer to a static object returned. If you use a singleton pattern be sure to add a clear explanation in a comment so that other developers will understand what you are doing.

CI8 **Declare as `const` a member function that does not affect the state of the object. *(REQUIRED)***

RC Declaring a member function as `const` has two important implications:

- Only `const` member functions can be called for `const` objects
- A `const` member function will not change data members

It is a common mistake to forget to `const` declare member functions that should be `const`.

This rule does not apply to the case where a member function which does not affect the state of the object overrides a non-const member function inherited from some super class.

CI9 **Do not let `const` member functions change the state of the program. *(RECOMMENDED)***

A `const` member function promises not to change any of the data members of the object. Usually this is not enough. It should be possible to call a `const` member function any number of times without affecting the state of the complete program. It is therefore important that a `const` member function refrains from changing static data members or other objects to which the object has a pointer or reference.

3.5.4 Overloading and Default Arguments

CI10 **Use function overloading only when methods differ in their argument list, but the task performed is the same. *(RECOMMENDED)***

Using function name overloading for any other purpose than to group closely related member functions is very confusing and is not recommended.

3.6 `new` and `delete` (CN)

CN1 **Match every invocation of `new` with one invocation of `delete` in all possible control flows from `new`. *(REQUIRED)***

A missing delete would cause a memory leak.

Example:

Every item created by “new” must, at any time, be associated with an entity responsible for its deletion.

Note: However, in the Gaudi / Athena framework, an object created with “new” and registered in the Transient Data Store (TDS) is under control of the TDS and must not be deleted.

CN2 **A function must not use the `delete` operator on any pointer passed to it as an argument. *(REQUIRED)***

RC

Note: 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.

This is essentially a design issue. This is illustrated in the following example: Suppose that a track finding algorithm finds some track candidates. These candidates are passed by pointer to the track matching routine. If a candidate does not match to any calorimeter hit and it is recognised as fake, who will delete this candidate? The track matching? There are various solutions but evidently this is a design issue.

CN3 **Do not access a pointer or reference to a deleted object. *(REQUIRED)***

A pointer that has been used as argument to a `delete` expression should not be used again unless you have given it a new value, because the language does not define what should happen if you access a deleted object. This includes trying to delete an already deleted object. You should assign the pointer to `0` or a new valid object after the "delete" is called; otherwise you get a "dangling" pointer.

CN4 **After deleting a pointer, assign it to zero. *(REQUIRED)***

RC C++ guarantees that deletion of zero pointers is safe, so this gives some safety against double deletes.

Example:

```
X* myX = makeAnX();
delete myX;
myX = 0;
```

3.7 Static and Global Objects (CS)

CS1 **Do not declare global variables. *(REQUIRED)***

RC If necessary, encapsulate those variables in a class or in a namespace. Global variables violate encapsulation and can cause global scope name clashes. Global variables make classes that use them context-dependent, hard to manage and difficult to reuse.

CS2 **Do not use global functions except to implement symmetric binary operators. *(REQUIRED)***

RC This is the only way to get conversions of the left operand of binary operations to work. It is common in implementing the symmetric operator to call the corresponding asymmetric binary operator.

Example:

```
Complex operator* (const Complex & lhs, const Complex & rhs) {
   Complex result(lhs);
   return result *= rhs;
}
```

Here the * operator has been defined for Complex numbers in terms of the *= operator.

Note:

If you implement a binary operator as a member function, say

```
Rational operator*(const Rational& rhs) const;
```

you'll get an error if the left hand operand is not a Rational:

```
Rational a(3,4);
Rational result;
result = a*2;   // fine
// this translates in functional form as result = a.operator*(2)
// and works since "a" is an instance of the class that contains
// an operator* and an implicit type conversion done by the
// compiler takes the integer 2 and creates the corresponding
// Rational by invoking a Rational(int) constructor

result = 2*a;   // error
// this translates in functional form as result = 2.operator*(a)
// and fails since the integer 2 has no associated class with
// an operator* member function and the implicit conversion seen
// above cannot be performed here because 2 is not in the
// parameter list of the member function!
```

If instead you implement: class Rational with no operator* and

```
Rational operator*(const Rational& lhs, const Rational& rhs)
```

you restore commutativity i.e. correct math!

Exception:

Sometimes global functions are required for other reasons, e.g. X11 callbacks, unix signal handlers. Moreover, there are non-symmetric operators that need to be overloaded globally (e.g. operator<< and operator>> for iostreams). Other examples include overloading common pure functions like "abs" or "min" for custom types.

3.8 Object-Oriented Programming (CB)

CB1 **Do not declare data members to be public. *(REQUIRED)***

RC This ensures that data members are only accessed from within member functions. Hiding data makes it easier to change implementation and provides a uniform interface to the object.

Example:

```
class Point {
   public:
     Number x() const; // Return the x coordinate
   private:
     Number m_x;       // The x coordinate (safely hidden)
};
```

The fact that the class Point has a data member `m_x` which holds the `x` coordinate is hidden.

CB2 **If a class has at least one virtual method then it must have a public virtual destructor, or (exceptionally) a protected destructor *(REQUIRED)***

RC

The destructor of a base class is a member function that in most cases should be declared virtual. It is necessary to declare it virtual in a base class if derived class objects are deleted through a base class pointer. If the destructor is not declared virtual, only the base class destructor will be called when an object is deleted that way.

Exception:

There is a case where it is not appropriate to use a virtual destructor: a mix-in class. Such a class is used to define a small part of an interface, which is inherited (mixed in) by subclasses. In these cases the destructor, and hence the possibility of a user deleting a pointer to such a mix-in base class, should normally not to be part of the interface offered by the base class. It is best in these cases to have a nonvirtual, nonpublic destructor because that will prevent a user of a pointer to such a base class from claiming ownership of the object and deciding to simply delete it. In such cases it is appropriate to make the destructor protected. This will stop users from accidentally deleting an object through a pointer to the mix-in base-class, so it is no longer necessary to require the destructor to be virtual.

CB3 **Always re-declare virtual functions as virtual in derived classes. *(REQUIRED)***

RC This is just for clarity of code. The compiler will know it is virtual, but the human reader may not. This, of course, also includes the destructor, as stated in item CB2.

CB4 **Avoid multiple inheritance, except for abstract interfaces. *(RECOMMENDED)***

RC Multiple inheritance is seldom necessary, and it is rather complex and error prone.

The only valid exception is for inheriting interfaces or when the inherited behaviour is completely decoupled from the class's responsibility.

Example:

> For a detailed example of a reasonable application of multiple inheritance, see Ref. [3] item 43.

CB5 **Avoid the use of `friend` declarations. *(RECOMMENDED)***

RC Friend declarations are almost always symptoms of bad design and they break encapsulation. When you can avoid them, you should.

Possible exceptions are operators "<<" and ">>", and binary operators on classes.

3.9 Assertions and error conditions (CE)

CE1 **Pre-conditions and post-conditions should be checked for validity. *(RECOMMENDED)***

You should validate your input and output data, whenever an invalid input can cause an invalid output.

CE2 **Make sure assertions are not compiled in the production releases. *(REQUIRED)***

Assertions should be used for the testing phase. They should not be present in production releases. This means that if you use assertions you must remember to remove them at the end of the chain of development releases. (Assertions, when assert(expression) is used, are automatically discarded if the macro NDEBUG is defined.)

The program will also run faster if unnecessary checks are compiled out.

Assertions which seem to be necessary even in production code, are really exceptions, and should be treated as such. In other words, you should not use assertions to check conditions that should always result in throwing an exception if the check fails. Such exceptions are part of the production code and should not be removable.

3.10 Error Handling (CH)

CH1 **Use the standard error printing facility for informational messages. Do not use cerr and cout. *(RECOMMENDED)***

The "standard error printing facility" in Athena / Gaudi is "MsgStream". No production code should use cout. Classes which are not Athena-aware could use cerr before throwing an exception, but all Athena-aware classes should use "MSG::FATAL" and / or throw a Gaudi Exception.

CH2 **Check for all errors reported from functions. *(REQUIRED)***

It is important to always check error conditions, regardless of how they are reported. If a function throws exceptions, it is important to catch all of them.

CH3 **Use exception handling instead of status values and error codes. *(RECOMMENDED)***

Exceptions in C++ are a means of separating error reporting from error handling. They should be used for reporting errors that the calling code should not be expected to handle. An exception is "thrown" to an error handler, so the treatment becomes non-local.

Because an exception is an object, an arbitrary amount of error information can be stored in an exception object; the more information that is available, the greater the chance that the correct decision is made for how to handle the error.

Note that when you throw an exception, control DOES NOT return to the throw site, and you usually have to do some cleaning up in order to leave, or continue, the program execution gracefully.

In certain cases, exception handling can be localized to one function along a call chain; this implies that less code needs to be written, and it is more legible.

Example:

```
try {
   // ordinary flow of control
   f();
   g();
}
catch(...) {    // handler for any kind of exception
// but as you do not know what you might catch with this syntax
// probably the only thing to do is to stop as cleanly as you can
   // error handling
}
```

Example:

Examples for CH2 and CH3:

Catch all exceptions:

```
try
{
   // code that may throw various exceptions
}
catch (std::ios_base::failure  { // handle any stream io error }
catch (std::exception& e)     { // handle any standard library exception }
catch (...)                                  { // catch any other exception }
```

The order of handlers matters: if the catch-all handler (see below) came first, for instance, the standard library exception handler would never be considered.

On some occasions it may be necessary to re-throw an exception. This happens for example when the handler may decide that it cannot completely handle the error or when the information needed to handle it is not available in a single handle.

Re-throw if necessary:

```
try
{
   // code that may throw an error, say of type SizeError
}
catch (SizeError)
{
   if(Error_Can_Be_Handled)
   {
      // do something
      return;
   }
   else
   {
      // do something else and
      // re-throw the exception: throw() without an operand
      throw;
   }
}
```

One way of catching every exception:

```
try
{
   // code that may throw an error, say of type SizeError
}
catch (...)  // the ellipsis here means "catch any exception"
{
   // cleanup
   throw;
}
```

CH4 **Do not throw exceptions as a way of reporting uncommon values from a function. *(RECOMMENDED)***

If an error CAN be handled locally, then it should be. Exceptions should not be used to signal events which can be expected to occur in a regular program execution. It is up to programmers to decide what it means to be exceptional in each context.

Example:

Take the case of a function `find()`. It is quite common that the object looked for is not found, and it is certainly not a failure; it is therefore not reasonable in this case to throw an exception. It is clearer if you return a well defined value.

CH5 **Use exception specifications to declare which exceptions might be thrown from a function. *(RECOMMENDED)***

RC

If a function does not have an exception specification, that function is allowed to throw any type of exception; that makes code unreliable and difficult to understand.

It is recommended to use exception specification as much as possible. The compiler will check that the exception classes exist and are available to the user. Compilers are also sometimes able to detect inconsistent exception specification during compilation

Syntactically, an exception specification is part of a function declaration or function definition, and has the form:

```
function header throw (type list)
e.g. void foo() throw(int, over_flow);
```

The type list is the list of types that a throw expression within the function can have. If the list is empty (e.g. `void noex(int i) throw();`) the compiler may assume that no throw will be executed by the function, either directly or indirectly.

The function definition and the function declaration must write out the exception specification identically: .

Example:

Wrong:

```
void foo() throw(int, over_flow);

void foo()  // error: exception-specification missing!
{
   // do something
}
```

Correct:

```
void foo() throw(int, over_flow);

void foo() throw(int, over_flow)
{
  // do something
}
```

Warnings:

Without a clear policy on exceptions, specifications can cause a great deal of problems.

Exception specifications may have unexpected run-time overheads. It has to be understood whether these overheads are significant and whether compilers are likely to optimise them in the future.

Class or method documentation should mention what exceptions the class

1. throws
2. passes through
3. swallows
4. if the code is exception safe and what kind of exception semantics it has (e.g. if the class implements full commit-or-rollback, or whether it just makes it safe to delete the object later, or something in between)
5. what exception semantics and behaviour it expects from other classes it uses, especially if abstract interfaces are involved.

All this is more than what can be deduced from exception specifications.

3.11 Parts of C++ to Avoid (CA)

Here a set of different items are collected. They highlight parts of the language which should be avoided, because there are better ways to achieve the desired results. In particular, programmers should avoid using the old standard C functions, where C++ has introduced new and safer possibilities.

CA1 **Do not use `malloc`, `calloc`, `realloc` and `free`. Use `new` and `delete` instead. *(REQUIRED)***

RC You must avoid all memory-handling functions from the standard C-library (`malloc`, `calloc`, `realloc` and `free`) because they do not call constructors for new objects or destructors for deleted objects.

CA2 **Do not use functions defined in `stdio`. Use the `iostream` functions in their place *(RECOMMENDED)***

RC

`scanf` and `printf` are not type-safe and they are not extensible. Use `operator>>` and `operator<<` associated with C++ streams instead. `iostream` and `stdio` functions should never be mixed.

Example :

```
// type safety
char* aString("Hello Atlas");
printf("This works: %s \n", aString);
cout <<"This also works:"<<aString<<endl;
char aChar('!');
printf("This does not %s \n", aChar);// and you get a core dump
cout <<"But this is still OK :"<<aChar<<endl;

//extensibility
std::string aCPPString("Hello Atlas");
printf("This does not work: %s \n", aCPPString); //Core dump again
```

Example : Taken from offline / Control / StoreGateSvc.cxx

```
#include <iostream>
..............
#ifndef HAVE_NEW_IOSTREAMS
#include <strstream>  /*gnu-specific*/
typedef strstream t_sstream;
#else
#include <sstream>
typedef std::ostringstream t_sstream;
#endif
..............
t_sstream ost;
..............
// loop over each type:
    SG::ConstProxyIterator p_iter = (s_iter->second).begin();
    SG::ConstProxyIterator p_end =  (s_iter->second).end();
while (p_iter != p_end) {
      const DataProxy& dp(*p_iter->second);
      ost << " flags: ("
          << setw(7) << (dp.isValid() ? "valid" : "INVALID") << ", "
          << setw(8) << (dp.isConst() ? "locked" : "UNLOCKED") << ", "
          << setw(6) << (dp.isResetOnly() ? "reset" : "DELETE")
          << ") --- data: " << hex << setw(10) << dp.object() << dec
          << " --- key: " << p_iter->first << '\n';
      ++p_iter;
    }
```

CA3 **Do not use the ellipsis notation for function arguments. *(REQUIRED)***

RC Functions with an unspecified number of arguments should not be used because they are a common cause of bugs that are hard to find. But catch(...) to catch any exception is acceptable.

Example:

```
// avoid to define functions like:

void error(int severity ...) // “severity” followed by a
                          // zero-terminated list of char*s
```

CA4 **Do not use preprocessor macros, except for system provided macros. *(RECOMMENDED)***

RC Use templates or inline functions rather than the pre-processor macros.

Example:

```
// NOT recommended to have function-like macro
#define SQUARE(x) x*x
```

Better to define an inline function:

```
inline int square(int x) {
   return x*x;
};
```

CA5 **The only permissible use of #ifdef is to precede a block of non-portable code or to allow the template declaration as CT1. *(REQUIRED)***

RC

CA6 **Do not declare related numerical values as `const`. Use `enum` declarations. *(REQUIRED)***

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

Example:

```
enum State {halted, starting, running, paused};
```

CA7 **Do not use `NULL` to indicate a null pointer; use the integer constant `0`. *(REQUIRED)***

RC No object is allocated with the address `0`. Consequently, `0` acts as a pointer literal, indicating that a pointer doesn’t refer to an object. In C, it has been popular to define a macro NULL to represent the zero pointer. Because of C++’s tighter type checking, the use of plain `0`, rather than any suggested NULL macro, leads to fewer problems.

CA8 **Do not use `const char*` or built-in arrays "`[]`"; use std::string instead. *(REQUIRED)***

RC The standard library, STL is the accepted standard for ATLAS. Use STL whenever it has the desired functionality.

Note: some exceptions might be necessary for online software.

CA9 **Do not use `union` types. *(REQUIRED)***

RC Unions can be an indication of a non-object-oriented design that is hard to extend. The usual alternative to unions is inheritance and dynamic binding. The advantage of having a derived class representing each type of value stored is that the set of derived class can be extended without rewriting any code. Because code with unions is only slightly more efficient, but much more difficult to maintain, you should avoid it.

Note: some exceptions might be necessary for online software.

CA10 **Do not use asm (the assembler macro facility of C++). *(REQUIRED)***

RC Note: some exceptions might be necessary for online software.

CA11 **Do not use the keyword `struct`. *(RECOMMENDED)***

RC The `class` is identical to the `struct` except that by default its contents are private rather than public.

`struct` may be allowed for writing non-object-oriented PODs (plain old data, i.e. C structs) on purpose. It is a good indication that the code is on purpose not object oriented, e.g. in some hidden internals of complicated classes or in bottom layers of interfacing to the host system.

CA12 **Do not use file scope objects; use class scope instead. *(REQUIRED)***

RC File scope is a useless complication that is better to be avoided.

CA13 **Do not declare your own typedef for booleans. Use the `bool` type of C++ for booleans. *(REQUIRED)***

RC

The `bool` type was not implemented in C. Programmers usually got around the problem by typedefs and / or const declarations. This is no longer needed, and must not be used in ATLAS code.

CA14 **Avoid pointer arithmetic. *(REQUIRED)***

RC Pointer arithmetic reduces readability, and is extremely error prone.

3.12 Readability and maintainability (CR)

CR1 **Avoid duplicated code. *(RECOMMENDED)***

This statement has a twofold meaning.

The first and most evident is that one must avoid simply cutting and pasting pieces of code. When similar functionalities are necessary in different places, they should be collected in methods, and reused.

The second meaning is at the design level, and is the concept of code reuse.

Note:

Reuse of code has the benefit of making a program easier to understand and to maintain. An additional benefit is better quality because code that is reused gets tested much better.

Code reuse, however, is not the end-all goal, and in particular, it is less important than encapsulation. One should not use inheritance to reuse a bit of code from another class.

CR2 **Document in the code any cases where clarity has been sacrificed for performance. *(REQUIRED)***

Optimise code only when you know you have a performance problem. This means that during the implementation phase you should write code that is easy to read, understand and maintain. Do not write cryptic code, just to improve its performance.

Very often bad performance is due to bad design. Unnecessary copying of objects, creation of large numbers of temporary objects and improper inheritance, poor choice of algorithms, for example, can be rather costly and are best addressed at the architecture and design level.

CR3 **If you use a Typedef it should be declared private or protected. *(REQUIRED)***

RC Typedefs are a serious impediment in large systems. While they simplify code for the original author, a system filled with typedefs can be difficult to understand. If the reader encounters a class A, he can find an #include with “A.h” in it to locate a description of A, but typedefs carry no context that tell a reader where to find a definition. Moreover, most of the generic characteristics obtained with typedefs are better handled by object oriented techniques, like polymorphism.

A typedef statement, which is declared within a class as private or protected, is used within a limited scope and is therefore acceptable.

CR4 **Code should be written to use standard Atlas units for time, distance, energy, etc. *(RECOMMENDED)***

This rule will not be effective until ATLAS has a standard way of dealing with units.

3.13 Portability (CP)

CP1 **All code must be adherent to the ANSI C++ standard. *(REQUIRED)***

Note: Adhesion to the standard must be done to the extent that the selected compilers allow it.

CP2 **Make non-portable code easy to find and replace. *(REQUIRED)***

Isolate non-portable code as much as possible so that it is easy to find and replace. For that you can use the directive `#ifdef`.

However, #ifdefs can make a program completely unreadable. In addition, if the problems being solved by the ifdef are not solved centrally by the release tool, then you resolve the problem over and over. Therefore. the using of #ifdef should be limited.

CP3 **Headers supplied by the implementation (system or standard libraries header files) must go in <> brackets; all other headers must go in "" quotes. *(REQUIRED)***

Example:

```
// Include only standard header with <>
#include <iostream>   // OK: standard header
#include <MyFyle.hh>  // NO: nonstandard header

// Include any header with ""
#include "stdlib.h"   // NO: better to use <>
#include "MyPackage/MyFyle.h"  // OK
```

CP4 **Do not specify absolute directory names in `include` directives. Instead, specify only the terminal package name and the file name. *(REQUIRED)***

RC

It is better to specify to the build environment where files may be located because then you do not need to change any `include` directives if you switch to a different platform.

Example:

```
#include "/afs/cern.ch/atlas/software/dist/1.2.1/Foo/Bar/Qux.h"  // Wrong
#include "Bar/Qux.h"                                             // Right
```

CP5 **Always treat `include` file names as case-sensitive. *(REQUIRED)***

Some operating systems, e.g. Windows NT, do not have case-sensitive file names. You should always include a file as if it were case-sensitive. Otherwise your code could be difficult to port to an environment with case-sensitive file names.

Example:

```
// Includes the same file on Windows NT, but not on UNIX
#include <Iostream>   //not correct
#include <iostream>   //OK
```

CP6 **Do not make assumptions about the size or layout in memory of an object. *(RECOMMENDED)***

The sizes of built-in types are different in different environment. For example, an `int` may be 16, 32 or even 64 bits long. The layout of objects is also different in different environments, so it is unwise to make any kind of assumption about the layout in memory of objects.

CP7 **Take machine precision into account in your conditional statements. Do not compare floats or doubles for equality. *(REQUIRED)***

Have a look at the numeric_limits<T> class, and make sure your code is not platform dependent. In particular, take care when testing floating point values for equality.

Example:

it is better to use:

```
const double tolerance = 0.001;
...
#include <math.h>
if ( fabs(value1 - value2) < tolerance ) ...
```

than

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

CP8 **Do not depend on the order of evaluation of arguments to a function, in particular never use ++ and -- operators on method arguments in function calls. *(REQUIRED)***

The order of evaluation of function arguments is strongly compiler dependent. The behaviour of `foo(a++, vec(a));` is platform dependent.

Example:

```
func(f1(), f2(), f3());
// f1 may be evaluated before f2 and f3,
// but don't depend on it!
```

CP9 **Do not use system calls if there is another possibility (e.g. the C++ run time library). *(REQUIRED)***

For example, do not forget about non-unix platforms.

CP10 **Use long (not int) and double (not float). *(RECOMMENDED)***

RC This ensures sufficient storage without rollover but, more importantly, reduces incompatibility between clients/consumers that use integral and floating point data.

Indexes of "for" loops are excluded from the rule.

CP11 **Do not call any code that is not in the release or is not in the list of allowed external software. *(REQUIRED)***

CP12 **Use the capabilities of the release tools system rather than writing or extending the makefiles. *(REQUIRED)***

As the system grows, per-package customizing of makefiles would make control of the build process impossible.

3.14 C++ Templates (CT)

Templates are powerful and should be used when needed. But it should be recognized that templates can increase the size of the executable, increase the compilation time, and can create code that can not be repaired by simple re-linking, forcing release builds.

CT1 **Declare a template in a.h file, as for any other class, but conditionally include the definition at the end. *(REQUIRED)***

RC

Different compilers handle class and function templates in different ways. Some want the template definition to be included into the source code of consumers, others do not. To give systems level control to this process, the following way of organizing templates is recommended.

Example:

```
Declare the template in a .h file, as for any other class, but conditionally
include the definition at the end:

  X.h:
        template <class T>
        class X {
           ... template class declaration
        };

        #ifdef ATLAS_COMP_INST
        #include “Package/X.cxx”
        #endif

In the implementation file:

  X.cxx:
        #include “Package/X.h”
        ... implement the class
```

CT2 **Exclude template definition (.cxx) files from the library list. *(REQUIRED)***

The .cxx files with template definitions must be excluded from the “list of things to compile into the package’s library”.

CT3 Templates must be tested by a program which instantiates them. Exclude the test program from the library list. *(REQUIRED)*

If you provide class or function templates in a package, then you must write a test application that instantiates these templates. Exclude this test from your library, but you must compile it as a pre-release test to insure that the template code can compile. Templates are only compiled when instantiated, so if they are not tested there may be an error which will be revealed only when a client uses your code.

CT4 Class and function templates must not depend upon the context of the translation file instantiating them. *(REQUIRED)*

For example, you should not have a template that depends upon a typedef, enum or other global quantity that is provided by the caller. Otherwise, template instantiations made in A.cxx may differ from those in B.cxx.

4 Style

Code is always written in a particular style. Discussing style is highly controversial. This section contains indications aimed at defining one style that should allow a common and consistent “style of the code”, i.e. a common look. Style relates to matters which do not affect the output of the compiler.

4.1 General aspects of style (SG)

SG1 **The `public`, `protected` and `private` sections of a class must be declared in that order. Within each section, nested types (e.g. `enum` or `class`) must appear at the top. *(REQUIRED)***

RC

The public part should be most interesting to the user of the class, and should therefore come first. The private part should be of no interest to the user and should therefore be listed last in the class declaration.

Example:

```
class Path {
   public:
      Path();
      ~Path();
   protected:
      void draw();
   private:
      class Internal {
      // Path::Internal declarations go here ...
      };
};
```

SG2 **Keep the ordering of methods in the header file and in the source files identical. *(REQUIRED)***

RC This facilitates the readability of the class implementation.

SG3 **Statements should not exceed 100 characters (excluding leading spaces). If possible, break long statements up into multiple ones. *(RECOMMENDED)***

RC

SG4 **Limit line length to 120 character positions (including white space and expanded tabs). *(REQUIRED)***

RC

SG5 **Include meaningful dummy argument names in function declarations. Any dummy argument names used in function declarations must be the same as in the definition. *(REQUIRED)***

RC

Although they are not compulsory, dummy arguments improve a lot the understanding and use of the class interface.

Example:

The constructor below takes 2 Numbers, but what are they?

```
class Point {
public:
    Point (Number, Number);
}
```

the following is clearer

```
class Point {
public:
    Point (Number x, Number y);
}
```

because the meaning of the parameters is explicitly indicated.

SG6 **The code should be properly indented for readability reasons. *(RECOMMENDED)***

Amount of indentation is hard to regulate. If a recommendation were to be given then four spaces seem reasonable since it guides the eye well, without running out of space in a line too soon. The important thing is that if one is modifying someone else's code, the indentation style of the original code should be adopted.

SG7 **Do not use spaces in front of [], (), and to either side of . and ->, in references to functions or arrays. *(REQUIRED)***

RC

This rule does not apply to spaces which follow C++ keywords such as `if`, `for`, and `switch`.

Example:

```
a->foo();    // Good
b.bar();     // Good
b . bar();   // Not good
```

SG8 **Keep the style of each file consistent within itself. *(RECOMMENDED)***

Although standard appearance amongst ATLAS source files is desirable, when you modify a file, code in the style that already exists in that file. This means, leave things as you find them. Do not take a non-compliant file and adjust a portion of it that you work on. Either fix the whole thing, or code to match.

4.2 Comments (SC)

SC1 **Use javadoc style comments BEFORE class/method/field declarations. Use "`//`" for comments in method bodies. *(RECOMMENDED)***

RC

Atlas has adopted the Doxygen code documentation tool, which requires a specific format for comments; also the recommended CASE tool (Together) produces javadoc style comments, which are recognized by Doxygen. But this style of comment "`/**/`" does not nest; therefore you would have problems if you were to nest them by accident.

You should be careful to use "`#if 0`" and `#endif` rather than /**/ comments to temporarily kill blocks of code.

For help on Doxygen style comments see the Atlas sit "How_to" page at

http://atlas-sw.cern.ch/cgi-bin/cvsweb.cgi/~checkout~/offline/AtlasDoc/doc/sit/UserDev_HowTo.html

SC2 **All comments should be written in complete (short and expressive) English sentences. *(RECOMMENDED)***

The quality of the comments is an important factor for the understanding of the code.

SC3 **In the header file, provide a comment describing the use of a declared function and attributes, if this is not completely obvious from its name, also a comment describing what a method does *(REQUIRED)***

Example:

```
class Point {
   public:
      // Perpendicular distance of Point from Line
      Number distance (Line);
}
```

the comment includes the fact that it is the perpendicular distance.

SC4 **In the implementation file, above each method implementation, provide a comment describing what the method does (if not obvious), and the preconditions and postconditions. *(REQUIRED)***

The code of a method will be much easier to understand and maintain, if it is well explained in an initial comment.

A Terminology

The terminology used by this book is as defined by the "Standard for the Programming Language C++" [2], with some additions presented below.

Abstract interface An **abstract interface** is a class in which all methods are virtual and which has no member data.

Built-in type A **built-in type** is one of the types defined by the language, such as `int`, `short`, `char`, and `bool`.

`const` correct A program is **`const` correct** if it has correctly declared functions, parameters, return values, variables, and member functions as `const`.

Copy assignment operator The **copy assignment operator** of a class is the assignment operator that takes a reference to an object of the same class as a parameter.

Copy constructor The **copy constructor** of a class is the constructor that takes a reference to an object of the same class as a parameter.

Dangling pointer A **dangling pointer** points at an object that has been deleted.

Direct base class The **direct base class** of a class is the class explicitly mentioned as a base class in its definition. All other base classes are **indirect base classes**.

Dynamic binding A member function call is **dynamically bound** if different functions will be called depending on the type of the object operated on.

Encapsulation **Encapsulation** allows a user to depend only on the class interface, and not upon its implementation.

Exception safe A class is **exception safe** if its objects do not lose any resources, and do not invalidate their class invariant or terminate the application when they end their lifetimes because of an exception.

Explicit type conversion An **explicit type conversion** is the conversion of an object from one type to another where you explicitly write the resulting type.

File scope An object with **file scope** is accessible only to functions within the same translation unit.

Forwarding function A **forwarding function** is a function that does nothing more than call another function.

Global object A **global object** is an object in global scope.

Global scope An object or type is in **global scope** if it can be accessed from within any function of a program.

Implementation-defined behaviour Code with **implementation-defined behaviour** is completely legal C++, but compilers may differ. Compiler vendors are required to describe what their particular compiler does with such code.

Implicit type conversion	An **implicit type conversion** occurs when an object is converted from one type to another and when you do not explicitly write the resulting type.
Inheritance	A derived class **inherits** state and behaviour from a base class.
Inline definition file	An **inline definition file** is a file that contains only definitions of inline functions.
Iterator	An **iterator** is an object used to traverse through collections of objects.
Literal	A **literal** is a sequence of digits or characters that represents a constant value.
Member object	The **member objects** of a class are its base classes and data members.
Modifying function (modifier)	A **modifying function (modifier)** is a member function that changes the value of at least one data member.
Non-copyable class	A class is **non-copyable** if its objects cannot be copied.
Object-Oriented programming	A language supports **object-oriented programming** if it provides encapsulation, inheritance, and polymorphism.
Polymorphism	**Polymorphism** means that an expression can have many different interpretations depending on the context. This means that the same piece of code can be used to operate on many types of objects, as provided by dynamic binding and parameterization, for example.
Postcondition	A **postcondition** is a condition that must be true on exit from a member function if the precondition was valid on entry to that function. A class is implemented correctly if postconditions are never false.
Precondition	A **precondition** is a condition that must be true on entry to a member function. A class is used correctly if preconditions arc never false.
Resource	A **resource** is something that more than one program needs, but of which there is limited availability. Resources can be acquired and released.
State	The **state** of an object is the data members of the object, and possibly also other data to which the object has access, which affects the observable behaviour of the object.
Template definition file	A **template definition file** is a file containing only definitions of non-inline template functions.
Translation unit	A **translation unit** is the result of merging an implementation file with all its headers and header files.
Undefined behaviour	Code with **undefined behaviour** is not correct C++. The standard does not specify what a compiler should do with such code. It may ignore the problem completely, issue an error, or do something else.
Unspecified behaviour	Code with **unspecified behaviour** is completely legal C++, but compilers may differ. Compiler vendors are not required to describe what their particular compiler does with such code.

User-defined conversion A **user-defined conversion** is a conversion from one type to another introduced by a programmer; that is, it is not one of the conversions defined by the language. Such user-defined conversions are either non explicit constructors taking only one parameter, or conversion operators.

B List of the items of the standard

2.1 Naming of files (NF)

2.2 Meaningful Names (NM)

2.3 Illegal or Non-recommended Naming (NI)

2.4 Naming Conventions (NC)

3.1 Organizing the Code (CO)

3.2 Control Flow (CF)

3.3 Object Life Cycle (CL)

3.3.1 Initialization of Variables and Constants

3.3.2 Constructor Initializer Lists

3.3.3 Copying of Objects

3.4 Conversions (CC)

3.5 The Class Interface (CI)

3.5.1 Inline Functions

3.5.2 Argument Passing and Return Values

3.5.3 `const` Correctness

3.5.4 Overloading and Default Arguments

3.6 new and `delete` (CN)

3.7 Static and Global Objects (CS)

3.8 Object-Oriented Programming (CB)

3.9 Assertions and error conditions (CE)

3.10 Error Handling (CH)

3.11 Parts of C++ to Avoid (CA)

3.12 Readability and maintainability (CR)

3.13 Portability (CP)

3.14 C++ Templates (CT)

4.1 General aspects of style (SG)

4.2 Comments (SC)