Project Support Team - IT Division

C++ Coding Standard

Specification

Version: 1.1
Issue: 5
Status: FINAL
ID: CERN-UCO / 1999 / 207
Date: 5 January 2000

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

This document has been prepared using the Software Documentation Layout Templates that have been prepared by the IPT Group (Information, Process and Technology), IT Division, CERN (The European Laboratory for Particle Physics). For more information please contact *docsys@ptsun00.cern.ch.*

Abstract

This document defines a C++ coding standard, that should be adhered to when writing C++ code. It is the result of a work started in a Working Group, in the context of the SPIDER project, formed by representatives from different LHC experiments, with the goal to bring together their existing coding standards.

Document Control Sheet

Table 1 Document Control Sheet

Document	**Title:**	C++ Coding Standard Specification		
	Version:	1.1	**ID:**	CERN-UCO / 1999 / 207
	Issue:	5	**Status:**	FINAL
	Edition:	[Document Edition]	**Created:**	
			Date:	5 January 2000
	Available at:	http: / / consult.cern.ch / writeup / cppstd /		
	Keywords:	coding standard, C++		
Tool	**Name:**	Adobe FrameMaker	**Version:**	5.5
	Template:	Software Doc Layout Templates	**Version:**	Vb1
Authorship	**Written by:**	S.Paoli		
	Contributors:	P.Binko (LHCb), D.Burckhart (ATLAS), S.M.Fisher (ATLAS), I.Hrivnacova (ALICE), M.Lamanna (COMPASS), M.Stavrianakou (ATLAS), H.-P.Wellisch (CMS)		
	Reviewed by:	S.Giani, A.Khodabandeh		
	Approved by:	G.H.Pawlitzek		

Document Status Sheet

Table 2 Document Status Sheet

Title:	C++ Coding Standard Specification		
ID:	CERN-UCO / 1999 / 207		
Version	**Issue**	**Date**	**Reason for change**
1.0	0	5.3.1999	Release to the Review Board for review
1.1	1	4.8.1999	First public release
1.1	2	13.8.1999	Changed the item identifiers in paragraph 3.8
1.1	3	17.8.1999	Corrected title of item CB1
1.1	4	20.10.1999	Added CERN write-up reference on the front page
1.1	5	5.1.2000	Added missing “int” in item CA5. Changed SPIDER to Project Support Team. New e-mail address: Pst@cern.ch

Table of Contents

1 Introduction

This chapter describes the approach adopted for the definition of this document.

1.1 Purpose

The purpose of this document is to define a C++ coding standard that should be adhered to when writing C++ code. ISO 9000 and the Capability Maturity Model (CMM) state that coding standards are mandatory for any organization with quality goals. 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, such as how to design a class or a class hierarchy, 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.

This document does not substitute in any way the study of a book on C++ programming. To learn the C++ language we refer you to the classical books: [9] for getting started, and [10] for a complete and definitive guide. For more advanced readings on C++, we strongly recommend the books [11], [12] and [13].

1.2 Intended Audience

This document is addressed to all people involved in the production of C++ code for the experiments / projects at CERN.

1.3 Authors

This document originated in the context of the SPIDER project, where during summer 1998 a working group was set up, formed by representatives from different experiments / projects: ALICE, ATLAS, LHCb, CMS and COMPASS, and led by the IT / IPT group. Its goal was to propose a common standard across experiments / projects in order to foster common solutions, homogeneity of the C++ code produced in different experiments / projects, and save resources for implementation and maintenance of products and services (coding standard,

code check utilities, help-desk, tutorials, etc.). The work started from the C++ coding standards that were already in place in the experiments / projects participating to the Working Group [1], [2], [3], [4], [5]; the working group did an important work of identification of all the commonalities, and subsequent clarification, consolidation and agreement of the items to include in the common standard.

The work group was interrupted in spring 1999 by the suspension of the SPIDER project. The definition of the standard was completed by the IT / IPT group, taking into account feedback received from various experiments and individuals. Its audience was extended to all people involved in the production of C++ code at CERN.

1.4 Evolution and updating responsibility

Changes to this standard will be implemented according to a change management procedure, defined in the context of the Project Support Team, the follow up to the SPIDER project. Feedbacks and suggestion on how to improve this document are encouraged; they should be sent to Pst@cern.ch.

A continuation related to this standard was the evaluation of available static analysis tools, to support automatic checking of code against this standard. This evaluation has been performed; the detailed evaluation report is available from [6].

1.5 Approach

The sources of this standard are the original experiments / projects' documents [1], [2], [3], [4], [5], and the well known ELLEMTEL standard (last edition) [7], the de-facto C++ coding standard in the software industry; also the famous book by S.Meyers [11] has provided useful inputs for this standard. The present document contains, no more or different items (see par. 1.8) than those contained in the above mentioned documents. A selection and, in some cases, a rewording of the items have been necessary in order to achieve a coherent and comprehensive coding standard (set of items).

The experiments / projects' standards [1], [2], [3], [4], [5] can be found on the web, while the books [7] and [11] can be consulted in the Reference Section of the IPT library (CERN, Meyrin, building 1 R-017).

In any case, the standard cannot cover every issue of C++ programming, and cannot always match the different choices that different experiments / projects have made on certain issues. Therefore the different experiments / projects should, if necessary, tailor this standard to their specific quality requirements; this could mean to suppress an item or to add additional items.

The items contained in this standard have been organized in three sections: Naming, Coding and Style. The content of each section is described below.

Though usually items in coding standards are characterized with different levels of importance (rules and guidelines / recommendations), the items of this standard have not

been characterized in this way. The reason is that the different experiments / projects have different quality criteria, which determine whether a certain item is a “rule” or a “guideline”, as well as different implications of the importance levels (how “rules” or “guidelines” are differently enforced). Avoiding to propose such a characterization in this standard allows experiments / projects to adopt their own criteria.

This approach causes one problem: it seems that all the defined items have the same level of importance. The reader should be aware that some items are very important, as they strongly impact the quality of the produced code. On the other hand, some items are rather arbitrary conventions whose importance is simply in fostering a common style and idiom across a wide community of programmers; the benefit is clearly an increase in the readability and maintainability of the produced code.

1.5.1 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.

1.5.2 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.

1.5.3 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 anyway do not affect the output of the compiler.

1.5.4 Information provided for the items

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 simply represent the order within the subsection. This kind of identification should allow a minimal impact on the items numbering during the maintenance of the standard, that is in the possible cases in which items are added or removed.

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 title and clarifies its meaning and scope.

For most items two other keywords, source and status, are also present; this information was maintained until version 0.8 of the document, that is as long as the document was defined in the SPIDER working group. For the items subsequently introduced, and therefore not discussed in the working group, the status information is not present. The meaning of the two keywords is the following:

- Source: provides the identifier of the items from which the item was derived (See Table 1)
- Status: indicates whether the item was agreed by all experiments / project in the working group (Status = Common), or by the majority of them (Status = Majority)

The two keywords source and status are temporary; they will stay for the time necessary to help a possible migration to this standard, but will be removed as soon as they become historical information.

Table 1 mapping between Identifier and the Source document

Identifier (n=number)	Source document
n.RN (or RC, RS, GN, GC, GS)	CMS
CXX-n	ATLAS
Rn	LHCb
COMPn	COMPASS
ARNn	ALICE

1.6 Organization of this document

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 of the standard
- Appendix C: Correspondence of item numbers, from this version to version 0.8

1.7 References

1 *C++ coding standards for ATLAS*, S.M.Fisher, L.A.Tuura.
Document on the WWW at the URL:
http://www.cern.ch/Atlas/GROUPS/SOFTWARE/OO/asp/cxx-rules/

2 *C++ Coding Conventions*, P.Binko. LHCb Computing Note: LHCb 98-049 COMP

3 *The CMS coding and design guidelines*, J.P.Wellisch. CMS-NOTE 1998/070, CMS-NOTE 1998/071, and CMS-NOTE 1998/072

4 *ALICE C++ Coding Conventions*, I.Hrivnacova, http://www1.cern.ch/ALICE/Projects/offline/CodingConv.html

5 *COMPASS C++ Coding Conventions*, M.Lamanna, http://wwwcompass.cern.ch/compass/software/offline/coffee/codingRules.html

6 *C++ Coding Standard - Check Tools Evaluation Report*, S.Paoli, E.Arderiu-Ribera, G.Cosmo, S.M.Fisher, A.Khodabandeh, G.H.Pawlitzek, M.Stavrianakou, Restricted access, for availability please contact CERN IT-PST Pst@cern.ch

7 *Rules and Recommendations, Industrial Strength C++*, M.Henricson, E.Nyquist. Prentice Hall, 1996

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

9 *C++ Primer*, S.B.Lippman, Addison-Wesley, 1998

10 *The C++ Programming Language, Third Edition*, B.Stroustrup, Addison-Wesley, 1997

11 *Effective C++, Second Edition: 50 Specific Ways to Improve Your Programs and Designs*, S.Meyers, Addison-Wesley

12 *More Effective C++ : 35 New Ways to Improve Your Programs and Designs*, S.Meyers, Addison-Wesley

13 *Advanced C++ Programming Styles and Idioms*, J.O.Coplien, Addison-Wesley

1.8 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; those are not used in this document)

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. This would guarantee that programs are easier to understand, read and maintain.

2.1 Naming of files

NF1 **The name of the header file should be the same as the name of the class it defines, with a suffix "`.h`" appended.**

Example:

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

Source 1.RN, 1.GN, CXX-8, R4, ARN4, 4.RN, 4.GN, CXX-7, R7, R6
Status Common

NF2 **The name of the implementation file should be the same as the name of the class it implements, with a project dependent suffix appended.**

Example:

> The implementation file for the class CalorimeterCluster would have the name CalorimeterCluster.cxx if it were part of a project which had chosen the "cxx" suffix.
>
> The different LHC experiments / projects have chosen the following suffixes:
> ALICE, ATLAS: .cxx
> LHCb, COMPASS: .cpp
> CMS: .cc

Source 2.RN, 2.GN, CXX-19, R5, ARN5, 5.RN, 5.GN, CXX-18
Status Common

NF3 **If the implementation of `inline` functions is put in a separate file, this should have the same name of the class it implements, with a project dependent suffix appended.**

Typical choices for the suffix are "`.icc`" and "`.inl`".

Example:

> If the class CalorimeterCluster contains inline methods, and those are implemented in a separated file, this would have the name CalorimeterCluster.icc, or CalorimeterCluster.inl depending on the choice made in the project.

2.2 Meaningful Names

NM1 **Use pronounceable names, or acronyms used in the experiment.**

They have big merits in discussion, and for newcomers.

Example:

Use `nameLength` instead of `nLn`.

Source 6.RN, 7.GN, 31.RC, 3.GS, R15
Status Common

NM2 **Use names that are English and self-descriptive.**

This would help anybody else to understand the meaning of the declared entities.

Source 7.RN, 8.GN, COMP16
Status Common

NM3 **Names of classes, methods and important variables should be chosen with care, and should be meaningful. Abbreviations are to be avoided, except where they are widely accepted.**

This is very important to make the code easy to read and use.

Source CXX-50, R10, ARN1
Status Common

2.3 Illegal Naming

NI1 **Do not create very similar names.**

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
```

Source R13, 26.RS, 5.GS
Status Common

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

Many identifiers of this kind are reserved C key words.

Source 35.RC, 1.GS
Status Common

NI3 **Avoid single and simple character names (e.g. "`j`", "`iii`") except for local loop and array indices.**

Source R17, 27. RS, 6. GS
Status Common

2.4 Naming Conventions

NC1 **Class names start with the prefix "XYZ".**

The actual value for the prefix is a project/experiment convention. Of course it must not be unique all over the complete project; for example it could rather be unique within each component.

This is a way to improve the readability of the code; particularly when browsing over a large set of classes from different components. But to avoid name conflicts it is preferable to use namespaces, see item NC2.

Example:

A class Track could be present in different contexts, hence in different SW components. The class will be easier to identify if the name has as prefix the component identifier:

MCTrack	Monte Carlo
RecTrack	Reconstruction
AnalTrack	Analysis

NC2 **Use namespaces to avoid name conflicts.**

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. You can avoid that by declaring and defining names (that would otherwise be global) inside namespaces.

Example:

A namespace is a declarative region in which classes, functions, types and templates can be defined.

```
namespace Emc {

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

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;
```

It is possible to make all names from a namespace accessible with a `using` directive.

```
using namespace Emc;     // using directive
Track electronTrack;     // Emc::Track electronTrack;
Array<Track> allTracks;    // Emc::Array<Emc::Track> allTracks;
```

The following items could appear rather arbitrary. The importance of these conventions is simply in fostering a common style and naming across a wide community of programmers. The benefit is an increase in the readability and maintainability of the produced code, especially when compared to a situation were each programmer adopts an own naming convention.

NC3 Start class names, `typedefs` and `enum` types with an uppercase letter.

Example:

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

Source 9.RN, 11.GN, R11, ARN6
Status Common

NC4 **Start names of variables and functions with a lowercase letter.**

Example:

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

NC5 **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.**

Example:

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

Source 10.RN, 13.GN, R11, ARN2

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

CO1 Each header file should be self-contained.

If a header file is self-contained, nothing more than the inclusion of the single header file is needed to use the full interface of the class defined.

One way to test your header file is to always include it first in the corresponding implementation file.

Source 8.RC, 4.GC
Status Common

CO2 Avoid unnecessary inclusion.

This is necessary to guarantee that the dependencies present in the implementations are only those foreseen in the design.

Example A: unnecessary inclusion in the header file

```
file A.h:   #include “B.h”

file C.h:   #include “B.h”      // NOT necessary, avoid
            #include “A.h”
```

Example B: unnecessary inclusion in the implementation file

```
file A.h:   #include “B.h”

file A.cc:  #include “B.h”      // NOT necessary, avoid
            #include “A.h”
```

Source R57
Status Common

CO3 Header files should begin and end with multiple-inclusion protection.

Here below is showed how this is implemented:

```
#ifndef IDENTIFIER_H
```

```
#define IDENTIFIER_H

// The text of the header goes in here ...

#endif // IDENTIFIER_H
```

The actual value for the IDENTIFIER is a project/experiment convention.

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.

Source CXX-9, 7.RC, COMP8, R62, 1. GS, ARC3
Status Common

CO4 Use forward declaration instead of including a header file, if this is sufficient.

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.

Source CXX-22, R58, 9. RC, 2. GC
Status Common

CO5 Each header file should contain one class (or embedded or very tightly coupled classes) declaration only.

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

CO6 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.

This is for the same reason as for item CO5.

3.2 Control Flow

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

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 **Follow all flow control primitives (`if`, `else`, `while`, `for`, `do`, `switch`, and `case`) by a block, even if it is empty.**

This make code much more reliable and easy to read.

Example:

```
while (condition) {
   statement;
}
```

Avoid the following error-prone form:

```
if (condition) // avoid! this omits the braces {}!
    statement;
```

CF3 **All `switch` statements should have a `default` clause.**

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.

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 **All `if` statements should have an `else` clause.**

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`.

Example:

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

CF5 **Do not use `goto`.**

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.

Source	CXX-67, 9. GC, 24. RC
Status	Common

CF6 **Do not have overly complex functions.**

The number of possible paths through a function, which depends on the number of control flow primitives, is the main source of function complexity. Therefore you should be aware that heavy use of control flow primitives will make your code more difficult to maintain.

As a rule of thumb, remember the 7±2 rule: typically methods should not be longer than 7±2 statements.

3.3 Object Life Cycle

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 variables initialised to numeric values or strings in a highly visible position; whenever possible collected them in one place.**

It would be very hard to maintain a code in which numeric values or strings are spread over a big file. If declaration and initialization of variable to numeric values or strings is put on the most visible position, it will be easy to locate them, and maintain.

CL2 Declare each variable with the smallest possible scope and initialise it at the same time.

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 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
```

Source 2.RC, R40, CXX-31, R74, 1.GC
Status Common

CL3 In the function implementation, do not use numeric values or strings; use symbolic values instead.

For the definition of symbolic values see item CL1.

Source CXX-58, COMP6, R64, R88, 14. RC, 7.GC, 14. GC, 5.GS, 7.GS
Status Common

CL4 Do not use the same variable name in outer and inner scope.

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

Source 32.RC, 1.GC
Status Common

CL5 Declare each variable in a separate declaration statement.

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
```

Source CXX-32, R74
Status Majority
Not Common for CMS

3.3.2 Constructor Initializer Lists

CL6 Initialise in the class constructors all data members.

And if you add a new data member, don't forget to update accordingly all constructors, operators and the destructor.

CL7 Let the order in the initializer list be the same as the order of declaration in the header file: first base classes, then data members.

It is legal C++ to list initializer 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. A data member could be accessed before it is initialized if the order in the initializer list is incorrect.

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

Example:

```
class Derived : public Base {     // Base is number 1

   public:
      explicit Derived(int i);
      Derived();

   private:
      int jM;         // jM is number 2
      Base bM;        // bM is number 3
};

Derived::Derived(int i) : Base(i), jM(i), bM(i) {
// Recommended order        1        2      3

   // Empty
}
```

Source CXX-35, CXX-36
Status Majority
Not Common for CMS (will be OK in the future)

3.3.3 Copying of Objects

CL8 Avoid unnecessary copying of objects that are costly to copy.

Because a class could have other objects as data members or inherit from other classes, many member function calls would be needed to copy the object. To improve performance, you should not copy an object unless it is necessary.

It is possible to avoid copying by using pointers and references to objects, but then you will instead have to worry about the lifetime of objects. You must understand when it is necessary to copy an object and when it is not.

CL9 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.

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.

CL10 If objects of a class should never be copied, then the copy constructor and the copy assignment operator should be declared `private` and not implemented.

Ideally the question whether the class has a reasonable copy semantic will naturally come out of the design process. Do not push copy semantics on a class that should not have it.

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

CL11 If objects of a class should be copied, then the copy constructor and the copy assignment operator should be implemented, with the desired behaviour.

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 behavior and must be implemented. You have to decide if the resources pointed to must be copied as well (deep copy), and write the right behaviour in the operators.

Of course, constructor and destructor must be implemented as well, see item CB2.

Source	CXX-38, R77, 6.RC, 6.GC, 7.GC, COMP2
Status	Common

CL12 Assignment member functions should work correctly when the left and right operands are the same object.

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

    // ... implementation of operator=
  }
}
```

Source CXX-41, 10GC
Status Common

3.4 Conversions

CC1 Use explicit rather then implicit type conversion.

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.

CC2 When the new casts are supported by the compiler, use the new cast operators (`dynamic_cast` and `static_cast`) instead of the C-style casts.

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 behavior of an ordinary cast is undefined, or at least ambiguous.

CC3 Do not convert `const` objects to non-`const`.

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

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`.

3.5 The Class Interface

The class interface is the most important part of the class. Sophisticated algorithms will not help if the class interface is wrong.

3.5.1 Inline Functions

CI1 Inline access functions and forwarding functions.

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 function.

Source 2.GC, 3.GC, COMP3
Status Common

3.5.2 Argument Passing and Return Values

CI2 Adopt the good practice of design functions without any side effects.

Example:

No-one would expect `sin(x)` to modify `x`.

Source R66
Status Common

CI3 Pass arguments of built-in types by value unless the function should modify them.

A good practice is to pass built-in types such as `char`, `int`, and `double` by value because it is cheap to copy such variables. This recommendation is also valid for some objects of classes that are cheap to copy, such as simple aggregates of very small number of built-in types.

Example:

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

CI4 Pass arguments of class types by reference or pointer.

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
```

CI5 Have `operator=` return a reference to `*this`.

This ensures that:

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

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

Source CXX-40, R84, 15.RC, 4.GC
Status Common

3.5.3 const Correctness

CI6 Declare a pointer or reference argument, passed to a function, as const if the function does not change the object bound to it.

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);
```

CI7 The argument to a copy constructor and to an assignment operator should be a const reference.

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

Source CXX-39, R78
Status Common

CI8 In a class method do not return pointer or non-const reference to private data members.

Otherwise you break the principle of encapsulation.

If necessary you can return pointer to const or const reference.

Source 4.GC, 23.GC
Status Common

CI9 Declare as post const a member function that does not affect the state of the object.

Declaring a member function as const has two important implications:

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

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

Source 25.RC, 10.GC, R42, R66, CXX-26, 5.GC, 3. GC
Status Common

CI10 **Do not let `const` member functions change the state of the program.**

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, global data, or other objects to which the object has a pointer or reference.

3.5.4 Overloading and Default Arguments

CI11 **Use function overloading only when methods differ in their argument list, but the task performed is the same.**

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`

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

A missing `delete` would cause a memory leak.

Example:

If you allocates memory in the constructor, you should take care of deallocate it in the destructor.

Source CXX-42, R80, 13. GC
Status Common

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

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.

Source CXX-44, R82
Status Common

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

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. You could assign the pointer to `0` or a new valid object. Otherwise you get a “dangling” pointer.

3.7 Static and Global Objects

CS1 **Do not declare global variables.**

If necessary, encapsulate those variables in a class or in a namespace.

Source R19, R41, COMP15, 17.GC, 11.RC, 5.GC, ARC7
Status Majority
Not Common for ALICE

CS2 **Use global functions only for symmetric binary operators.**

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.

Source R43, CXX-17, CXX-46, R86, 5.GC
Status Common

3.8 Object-Oriented Programming

CB1 **Declare data members `private` or `protected`.**

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.

Source 3. RC, R40, 4. GC, COMP3, 9.GC, R21, CXX-12, R51
Status Common

CB2 Always declare and implement constructor and destructor.

This is important to avoid possible memory leak problems that one would not expect.

Example:

```
class Track {
   public:
      Track();
      virtual ~Track();
   private:
      class1 c1;
      class2 c2;
      int i3;
};

Track::Track() : c1(), c2(), i3(0) {

   // Empty
}
Track::~Track() {

   // Empty
}
```

CB3 A public base class must have either a public virtual destructor or a protected destructor.

The destructor 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.

However, there is a case where it is not appropriate to use virtual destructor: mix-in classes. 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.

Source CXX-48, R79
Status Majority
Not Common for CMS

CB4 Always redeclare virtual functions as virtual in derived classes.

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 CB3.

Source CXX-49, R67

CB5 Avoid multiple inheritance.

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 classes responsibility.

Example:

> For a detailed example of a reasonable application of multiple inheritance see [11] item 43.

CB6 Use public inheritance.

Private and protected inheritance is useful in rather specific cases only. As a rule of thumb use aggregation instead.

CB7 Avoid the use of `friend` declarations.

Friends declarations are syntoms of bad design and they break encapsulation. Tipically you can solve your problem in a different way.

Source CXX-D5, 26.RC, COMP4, 20. GC, 9. GC

3.9 Assertions and error conditions

CE1 Pre-conditions and post-conditions should be checked for validity.

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

Example:

```
EmcString::EmcString(const char* cp) throw(bad_alloc)
   : lengthM(strlen(cp)) {

   // check of preconditions: cp != 0
   // ...

   // check of postconditions:
   // operator new() will throw bad_alloc
   // if allocation fails

   cpM = new char[lenthM + 1];
   strcpy(cpM, cp);
}
```

CE2 Remove all assertions from production code.

Assertions should be used for the testing phase. The program will also run faster if unnecessary checks are removed.

Some conditions are not checked by assertions. 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

CH1 Check for all errors reported from functions.

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.

CH2 Use exception handling instead of status values and error codes.

For error reporting, exception handling is a more powerful technique than returning status values and error codes. It allows to separate code that handles errors from the ordinary flow of control.
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.
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
   // error handling
}
```

CH3 Do not throw exceptions as a way of reporting uncommon values from a function.

Your code can be difficult to understand if you throw exceptions in many different situations, ranging from a way to report unusual threads in your code to reporting fatal runtime problems.

Example:

Take the case of a function `find()`. 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.

CH4 Use exception specifications to declare which exceptions might be thrown from a function.

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 recommendable 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.

Example:

```
char& EmcString::at(size_t index) throw(EmcIndexOutOfRange) {

   if (index > lengthM) {
      throw EmcIndexOutOfRange(index);
    }
   return cpM[index];
}
```

3.11 Parts of C++ to Avoid

Here below a set of different items are collected. They highlight parts of the language that should be avoided, because there are better ways to achieve the desired results.

CA1 Use `new` and `delete` instead of `malloc`, `calloc`, `realloc` and `free`.

You should 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.

Source CXX-71, R83, 12. GC
Status Common

CA2 Use the `iostream` functions rather than those defined in `stdio`.

`scanf` and `printf` are not type-safe and they are not extensible. Use `operator>>` and `operator<<` instead.

Source CXX-69, R39
Status Common

CA3 Do not use the ellipsis notation.

Functions with an unspecified number of arguments should be avoided because they are a common cause of bugs that are hard to find.

Example:

```
// avoid to define functions like:

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

Source CXX-28, R71, 8.GC
Status Common

CA4 Do not use preprocessor macros, except for system provided macros.

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;
};
```

Source 29.RC, 4.GC, CXX-59, R89, CXX-61, R90
Status Common

CA5 Do not use `#define` to define symbolic constants or enums.

Example:

```
#define levels 5    // NOT recommended
```

If you need to define a symbolic constant, use:

```
const int levels = 5;
```

Source CXX-58, COMP6, R64, R88, 14. RC, 7.GC, 14. GC
Status Common

CA6 Use `enum` for related constants rather than `const`.

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};
```

Source CXX-78, R63
Status Majority
Not Common for CMS

CA7 Use the integer constant `0` for the null pointer; don't use `NULL`.

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.

Source 4.GC, CXX-70, 13.RC
Status Common

CA8 Use the standard library (STL) whenever it has the desired functionality.

In particular, do not use `const char*` or built-in arrays "`[]`".

Source 36.RC, 1.GC, CXX-75, 10.GC, R29, CXX-79, R30, 22.RS, 1.GS, 1.GC, 22.RC, 15.GC
Status Majority
Not Common for ATLAS Online

CA9 Do not use `union` types.

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.

Source CXX-77, R38
Status Majority
Not Common for CMS, ATLAS Online

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

NOTE: some exceptions might be necessary in Online.

Source 28.RC, 3.GC
Status Common

CA11 Do not use the keyword `struct`.

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

Source CXX-73, R37
Status Common

CA12 Do not use file scope objects; use class scope instead.

File scope is a useless complication that is better to avoid.

Source CXX-34
Status Majority
Not Common for CMS

CA13 Use the `bool` type of C++ for booleans.

Programmers may tend to use `int` instead of `bool` as this is a relatively new feature.

Source CXX-74, R36, 8. GC
Status Common

CA14 Avoid pointer arithmetic.

Pointer arithmetic makes readability very difficult and it is certainly one of the most error prone parts.

3.12 Readability and maintainability

CR1 Avoid duplicated code and data.

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, those should be collected in class methods, and reused.

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

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.

CR2 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.

Performance problems are more likely solved at an architecture and design level.

3.13 Portability

CP1 All code must be adherent to the ANSI C++ standard.

Current edition: [8]

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

Source CXX-62, ARC1
Status Majority
Not Common for CMS (OK on the principle)

CP2 Make nonportable code easy to find and replace.

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

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

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 "MyFyle.hh"  // OK
```

CP4 Do not specify absolute directory names in `include` directives.

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.

CP5 Always treat `include` file names as case-sensitive.

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>
#include <iostream>
```

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

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, such as when lumping together different data in a `struct`.

CP7 **Do not cast a pointer to a shorter quantity to a pointer to a longer quantity.**

Certain types have alignment requirements, which are requirements about the address of objects. For example, some architectures require that objects of a certain size start at an even address. It is a fatal error if a pointer to an object of that size points to an odd address. For example, you might have a `char` pointer and want to convert to an `int` pointer. If the pointer points to an address that it is illegal for an `int`, dereferencing the `int` pointer creates a runtime error.

CP8 **Take machine precision into account in your conditional statements. 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 ) ...
```

CP9 **Do not depend on the order of evaluation of arguments to a function.**

The order of evaluation of function arguments is strongly compiler dependent.

In particular never use ++, -- operators on method arguments in function calls. 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!
```

CP10 **Avoid using system calls if there is another possibility (e.g. the C++ run time library).**

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

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

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

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 ...
      };
};
```

Source CXX-16, ARC6, R52, R53, COMP10
Status Majority
Not Common for CMS

SG2 Keep the ordering of methods in the header file and in the source files identical.

This facilitates the readability of the class implementation.

Source 20.RS, 8.GS
Status Common

SG3 Arrange long statements on multiple lines in a way which maximises readability. If possible, break long statements up into multiple ones.

Source CXX-3, 24.RS, 3.GS
Status Common

SG4 Do not have any method bodies inside the class definitions (in header files).

The class definition will be more compact and comprehensible if no implementation can be seen in the class interface.

This also applies to inline functions. You can either put them in a separate file, or at the end of the header file, below the class definition.

Example:

```
class X
{
   public:
      // Not recommended: function definition in class
      bool insideClass() const { return false; }
      bool outsideClass() const;
};
// Recommended: function definition outside class
inline bool X::outsideClass() const
{
   return true;
}
```

Source CXX-13, R60, CXX-15
Status Majority
Not Common for CMS, ALICE (both experiments allow 2 lines of method body)

SG5 Include meaningful dummy argument names in function declarations.

Although they are not compulsory, dummy arguments improves 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 it is explicitly indicated the meaning of the parameters.

Source CXX-11, 3.GS
Status Common

SG6 **Any dummy argument names used in function declarations should be the same as in the definition.**

Source 4.GS, 25.RS, 7.GS
Status Common

SG7 **The code must be properly indented for readability reasons.**

SG8 **Do not use spaces in front of `[]`, `()`, and to either side of `.` and `->`.**

Example:

```
a->foo();    // Recommended
b.bar();     // Recommended
```

4.2 Comments

SC1 **Use "`//`" for comments.**

The C-like comments "`/**/`" do not nest; therefore you would have problems if by accident you nest them.

A special situation is if you adopt a code documentation tool, with specific conventions for comments; in this case you could be forced to violate this item. In this case be careful to use "`#if 0`" and `#endif` rather than /**/ comments to temporarily kill blocks of code.

Source CXX-72, R47, 5.GS, R48
Status Common

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

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

Source 7.RN, 15. GC, 22. GS
Status Common

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.**

Example:

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

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

Source CXX-10
Status Common

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

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

SC5 **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.**

The number five is obviously a reasonable arbitrary convention, in order to make the item objective and checkable.

Example:

```
#ifndef GEOMETRY_POINT_H
#define GEOMETRY_POINT_H

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 a line
    void translate(const Vector & vector);  // Shift a point
};
#endif // GEOMETRY_POINT_H
```

Source CXX-60, R49
Status Common

A Terminology

The terminology used by this book is as defined by the "Standard for the Programming Language C++" [8], with some additions presented below.

Abstract base class An **abstract base class** is a class with at least one pure virtual member function.

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

Class invariant A **class invariant** is a condition that defines all valid states for an object. An class invariant is both a precondition and post condition to a member function of the class.

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.

Declarative region A **declarative region** is the largest part of a program where a name declared can be used with its unqualified name.

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.

Flow control primitive The **flow control primitives** are `if-else`, `switch`, `do-while`, `while`, and `for`.

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

Free store An object on the **free store** is an object allocated with `new`.

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 behavior** 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 behavior 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.
Noncopyable class	A class is **noncopyable** 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.
Self-contained	A header file is **self-contained** if nothing more than its inclusion is needed to use the full interface of a class.
Signature	The **signature** of a function is defined by its return type, its parameter types and their order, and whether it has been declared const or volatile.

Slicing	**Slicing** means that the data added by a subclass are discarded when an object of the subclass is passed or returned by value to or from a function expecting a base class object.
Stack unwinding	**Stack unwinding** is the process during exception handling when the destructor is called for all local objects between the place where the exception was thrown and where it is caught.
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 behavior of the object.
Substitutability	**Substitutability** means that it is possible to use a pointer or reference to an object of a derived class wherever a pointer or reference to an object of a public base class is used.
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 behavior** 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 behavior** 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	AA **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 nonexplicit constructors taking only one parameter, or conversion operators.
Virtual table	A **virtual table** is an array of pointers to all virtual member functions of a class. Many compilers generate such tables to implement dynamic binding of virtual functions.

B List of the items of the standard

2.1 Naming of files

2.2 Meaningful Names

2.3 Illegal Naming

2.4 Naming Conventions

3.1 Organizing the Code

3.2 Control Flow

3.3 Object Life Cycle

3.3.1 Initialization of Variables and Constants

3.3.2 Constructor Initializer Lists

3.3.3 Copying of Objects

3.4 Conversions

3.5 The Class Interface

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`

3.7 Static and Global Objects

3.8 Object-Oriented Programming

3.9 Assertions and error conditions

3.10 Error Handling

3.11 Parts of C++ to Avoid

3.12 Readability and maintainability

3.13 Portability

4.1 General aspects of style

4.2 Comments

C Correspondence of item numbers

This appendix contains a matrix of correspondence between the item numbers in this version of the document and the version 0.8, that has been used by some projects. This correspondence matrix should facilitate the migration to this new version; it will remain part of the document until the migration has been completed.

Correspondance matrix

Current item number	Item number in version 0.8
NF1	RN1
NF2	RN2
NF3	3.GS, 4.GS
NM1	RN5
NM2	RN6
NM3	RN18
NI1	RN29
NI2	RN30
NI3	GN3
NC1	RN9, ARN6, R20
NC2	-
NC3	RN9, RN10
NC4	ARN13
NC5	RN11
CO1	RC27
CO2	RC30B
CO3	RC2
CO4	RC30
CO5	COMP12, AEC4, 14.RS, 1.GS, CXX-20, R50
CO6	R65, ARC5, COMP13, CXX-20, R50, 2.GS, 15.RS
CF1	-
CF2	-
CF3	6.GC
CF4	-
CF5	RC19
CF6	-

Correspondance matrix

Current item number	**Item number in version 0.8**
CL1	-
CL2	RC3
CL3	RC15
CL4	RC29
CL5	RS7
CL6	9.GC
CL7	RS8, RS9
CL8	-
CL9	-
CL10	-
CL11	RC23
CL12	RC22
CC1	4.RC, 6.GC, CXX-76, COMP5, R33
CC2	12.GC
CC3	7.GC, 30.RC
CI1	RC24B
CI2	GC4
CI3	RC8
CI4	RC8
CI5	RC21
CI6	RC22B, RC8
CI7	RC22B
CI8	RC10
CI9	RC6
CI10	-
CI11	21.GC
CN1	RC34
CN2	RC25
CN3	-
CS1	RC4
CS2	RC5
CB1	RC9

Correspondance matrix

Current item number	**Item number in version 0.8**
CB2	-
CB3	RC11
CB4	CXX-49, R67
CB5	8.GC, 18.GC, 11.GC
CB6	7.GC
CB7	CXX-D5, 26.RC, COMP4, 20.GC, 9.GC
CE1	R72
CE2	-
CH1	-
CH2	-
CH3	-
CH4	-
CA1	RC37
CA2	RC36
CA3	RC31
CA4	RC12
CA5	RC15
CA6	RC17
CA7	RC13
CA8	RC18
CA9	RC39
CA10	RC28
CA11	RC41
CA12	RC32
CA13	RC38
CA14	9.GC
CR1	-
CR2	3.GC, 41.RC
CP1	RC1
CP2	-
CP3	-
CP4	-

Correspondance matrix

Current item number	**Item number in version 0.8**
CP5	-
CP6	3.GC
CP7	-
CP8	6.GC, R92
CP9	9.GC, 38.RC
CP10	14.GC
SG1	RS1
SG2	RS2
SG3	RS4
SG4	RS6
SG5	RS6B
SG6	RS6C
SG7	5.GS, 29.RS, COMP11, R98, CXX-4
SG8	39.RS, 17.GS
SC1	RS6D
SC2	7.RN, 15.GC, 22.GS, 16.GC
SC3	RS6E
SC4	-
SC5	RS10