Open CASCADE Technology

Coding Rules

CONTENTS

1. INTRODUCTION

1.1. Preamble

The purpose of this document is to define and formalize one style of programming for developers working with Open CASCADE Technology (OCCT). The intended audience is developers who contribute to OCCT.

The establishment of a common style facilitates understanding and maintaining code developed by more than one programmer as well as making it easier for several people to co-operate in the development of the same software.

For historical reasons, OCCT code currently represents a mixture of several different programming styles. This document defines common style to be followed in new developments, aimed to make comprehension of OCCT source code easy and smooth, and to facilitate co-operative development of OCCT by wide community of developers.

The recommendations and rules presented here are based on existing de-facto standards and practices. They are not final, but should serve as a basis for continued work with OCCT.

1.2. Format

This document defines a set of rules. For each rule the following attributes are defined:

- **Unique rule number** in format **OCR####** (for possible references)
- **Severity**: one of
  - **Mandatory** - for critical rules that must be strictly followed by anybody who contributes to OCCT
  - **Recommended** - for non-critical but recommended rules which improve quality of OCCT sources but may be ignored in some cases
  - **Advise** - for advises that are aimed to make programmer's work more convenient and effective
- **Short Description**: To ease rule identification
- **Description**: Exact description of the rule (body)
- **Explanations**: Additional comments explaining why the rule is necessary can be found in the plain text around the rule.

The rules are numbered in the order of their creation and these numbers are fixed from that moment for the whole life of this document. For easy navigation the index of rules is provided in Appendix A.

2. NAMING CONVENTIONS

2.1. General naming rules

The names considered in this section are mainly those which compound the interface to OCCT libraries and thus are visible through the OCCT header files.

Rule OCR1013 Mandatory International language in names

All names in OCCT are composed of English words and their abbreviations.

Reason: OCCT is an open source available for international community.

Rule OCR1012 Recommended Suggestive names

Names should be suggestive or, at least, contain a suggestive part.

Currently, there is no exact rule accepted at OCCT that would define how to generate suggestive names. However, usually names given to toolkits, packages, classes and methods are suggestive. Here are several examples from Open Source part of OCCT:

- Packages containing words Geom or Geom2d in their names are related to geometrical data and operations.
- Packages containing words TopoDS or BRep in their names are related to topological data and operations.
- Packages that define storage schema tools for data classes contained in some package have the same names prefixed by 'P' for persistent classes (e.g. PGeom for Geom).
- In OCAF, packages that define transient, persistent data classes and drivers to map between them, have similar names prefixed by 'T', 'P', and 'M' correspondingly (e.g. TDocStd, PDocStd, MDocStd)
- Packages starting from XSDRAW… define resources and executables for Data Exchange DRAW applications, etc.
- Methods starting with Get… and Set… are usually responsible for (accordingly) retrieving/storing some data.

Rule OCR1031 Recommended Related names

Names that define logically connected functionality should have the same prefix (start with the same letters) or, at least, have any other common part in them.

As an example the method GetCoord can be given. It returns a triple of real values and is defined for directions, vectors and points. The logical connection is obvious.

Rule OCR1085 Recommended Camel Case style

Camel Case style is preferred for names.

For example:

```
Standard_Integer awidthofbox;  // This is bad
Standard_Integer width_of_box; // This is bad
Standard_Integer aWidthOfBox;  // This is OK
```

2.2. Names of development units

Usually unit (e.g. package) is a set of classes, methods, enumerations or any other sources implementing certain common functionality which, to the certain extent, is self contained and independent from other parts of library.

Rule OCR1032 Mandatory Underscores in units names

Names of units should not contain underscores, except cases where usage of underscores is allowed explicitly.

Usually names of files in OCCT are constructed according to the rules defined in the appropriate sections of this document.

Rule OCR1015 Mandatory File names extensions

The following extensions should be used for source files, depending on their type:

- **.cdl** - CDL declaration files
- **.c** / **.cxx** - C/C++ source files
- **.h** / **.hxx** - C/C++ header files
- **.lxx** - headers with definitions of inline methods (CDL packages)
- **.gxx** - sources of generic (template) CDL classes

2.3. Names of toolkits

The following rules are usually observed by OCCT developers in naming of toolkits:

Rule OCR1011 Mandatory Prefix for toolkits names

Toolkits names are prefixed by **TK**, followed by suggestive part of name explaining the domain of functionality covered by the toolkit (e.g. TKOpenGl).

2.4. Names of classes

Usually source files located in the unit have names that start from the name of the unit, separated from remaining part of file name (if any) by underscore "_". For instance, names of files containing sources of C++ classes are constructed according to the following template:

Rule OCR1014 Recommended Naming of C++ class files

The following template should be used for names of files containing sources of C++ classes:

```
<unit-name>_<class-name>.cxx (.hxx, .cdl etc.)
```

Files that contain sources related to whole unit are called by the name of unit with appropriate extension.

If source is split into several files, extra files should have names with additional number before extension, separated by underscore from basic name, for instance:

```
ShapeFix_Wire_1.cxx
```

2.5. Names of functions

The term “function” here is defined as

- Any class method
- Any package method
- Any non-member procedure or function

Currently there are no additional rules for names of functions

2.6. Names of variables

There are several rules that describe currently accepted practice used for naming variables in OCCT developments. While these rules are not mandatory, it is very desirable to follow them in order to avoid conflicts with names already used by OCCT libraries, and to have the same style of code.

There are some practices that are commonly accepted in OCCT regarding how names of variables are constructed.

Rule OCR1018 Recommended Naming of variables

Name of variable should not conflict with the OCCT global names (packages, macros, functions, global variables etc.), either existing or possible.

The name of variable should not start with underscore(s).

See the following examples:

```
Standard_Integer Elapsed_Time = 0; // This is bad – possible class name
Standard_Integer gp = 0;           // This is bad – existing package name
Standard_Integer gP = 0;           // This is OK
Standard_Integer _KERNEL; // This is bad
Standard_Integer  KERNEL; // This is OK
```

Rule OCR1019 Recommended Names of function parameters

The name of a function (procedure, class method) parameter should start with ‘**the**’ followed by the rest of the name starting with capital letter.

See the following examples:

```
void MyClass::MyFunction (const gp_Pnt& p);        // This is bad
void MyClass::MyFunction (const gp_Pnt& theP);     // This is OK
void MyClass::MyFunction (const gp_Pnt& thePoint); // This is preferred
```

Rule OCR1020 Recommended Names of class member variables

The name of a class member variable should start with ‘**my**’ followed by the rest of the name (rule for suggestive names applies) starting with capital letter.

See the following examples:

```
Standard_Integer counter;   // This is bad
Standard_Integer myC;       // This is OK
Standard_Integer myCounter; // This is preferred
```

It is strongly recommended to avoid defining any global variables (see Portability chapter). However, as soon as global variable is necessary, the following rule applies.

Rule OCR1021 Recommended Names of global variables

Global variable name should be prefixed by the name of a class or a package where it is defined followed with ‘**_my**’.

See the following examples:

```
Standard_Integer MyPackage_myGlobalVariable = 0;
Standard_Integer MyPackage_MyClass_myGlobalVariable = 0;
```

Rule OCR1022 Recommended Names of local variables

Local variable name should be constructed in such way that it can be distinguished from the name of a function parameter, a class member variable and a global variable. It is preferred to prefix local variable names with '**a**' and '**an**' (also '**is**' and '**has**' for Boolean variables).

See the following examples:

```
Standard_Integer theI;    // This is bad
Standard_Integer i;       // This is bad
Standard_Integer index;   // This is bad
Standard_Integer anIndex; // This is OK
```

Rule OCR1086 Recommended Avoid dummy names

Avoid dummy names like I, j, k. Such names are meaningless and easy to mix up.

One-char names commonly parasitized as cycle variables. Such names hide actual meaning and even cycle depth. Code becomes more and more complicated when such dummy names used multiple times in code for different cycles with different iteration ranges and meaning.

See the following examples for preferred style:

```
void average (const Standard_Real** theArray,
              Standard_Integer theRowsNb, Standard_Integer theRowLen,
              Standard_Real & theResult)
{
  theResult = 0.0;
  for (Standard_Integer aRow = 0; aRow < aRowsNb; ++aRow)
     for (Standard_Integer aCol = 0; aCol < aRowLen; ++aCol)
      theResult += theArray[aRow][aCol];
  theResult /= Standard_Real(aRowsNb * aRowLen);
}
```

3. FORMATTING RULES

In order to improve the open source readability and, consequently, maintainability, the following set of rules is applied.

Rule OCR1044 Mandatory International language in comments

All comments in all sources must be in English.

Reason: OCCT is an open source available for international community.

Rule OCR1046 Recommended Line length

In all sources try not to exceed 80 characters limit of line length.

Rule OCR1047 Recommended C++ style comments

Prefer C++ style comments in C++ sources.

Rule OCR1056 Recommended Get rid of unused code

Prefer commenting out or deleting unused pieces of code instead of using #define for that purpose.

Rule OCR1061 Mandatory Indentation in sources

Indentation in all sources should be set to two space characters. Using tabulation characters for indentation is disallowed.

Rule OCR1062 Recommended Using spaces

C/C++ reserved words, commas, colons and semicolons should be followed by a space character if they are not at the end of line. There should be no space characters after ‘**(**‘ and before ‘**)**’. Closing and opening brackets should be separated by a space character.

For better readability it is also recommended to surround conventional operators by a space character. See the following examples:

```
while (true)                // NOT:   while( true ) ...
{
  doSomething (a, b, c, d); // NOT:   doSomething (a,b,c,d);
}
for (i = 0; i < 10; ++i)    // NOT:   for (i=0;i<10;++i){
{
  a = (b + c) * d;          // NOT:   a=(b+c)*d
}
```

Rule OCR1063 Recommended Separate logical blocks

Separate logical blocks of code with one blank line and comments.

See the following example:

```
// Check arguments
Standard_Integer argc = argCount();
if (argc < 3 || smth_invalid) return ARG_INVALID;

// Read and check header
...
...

// Do our job
...
...
```

Notice that multiple blank lines should be avoided.

Rule OCR1064 Mandatory Separate function bodies

Use function descriptive blocks to separate function bodies from each other.

See the following example:

```
// ----------------------------------------------
// Function: TellMeSmthGood
// Purpose:  Gives me good news
// ----------------------------------------------
void TellMeSmthGood()
{
  ...
}

// ----------------------------------------------
// Function: TellMeSmthBad
// Purpose:  Gives me bad news
// ----------------------------------------------
void TellMeSmthBad()
{
  ...
}
```

Rule OCR1065 Mandatory Block layout

General block should have layout similarly to the following:

```
while (expression)
{
  ...
}
```

Entering block increases and leaving block decreases indentation to one tabulation.

Rule OCR1066 Recommended Single-line operators

Single-line conditional operator (if, while, for etc.) can be written without brackets on the following line.

```
if (expression)
  DoSomething();
```

Code on the same line as if() is less convenient for debugging.

Rule OCR1067 Recommended Use alignment

Use alignment wherever it enhances readability.

See the following example:

```
MyPackage_MyClass anObject;
Standard_Real     aMin;
Standard_Integer  aVal;

if      (aVal == lowValue)    computeSomething();
else if (aVal == mediumValue) computeSomethingElse();
else if (aVal == highValue)   computeSomethingElseYet();
```

Rule OCR1068 Recommended Indentation of comments

Comments should be indented similar to the code which they refer to or can be on the same line if they are short.

See the following example:

```
while (expression)
{
  // this is a long multi-line comment
  // which is really required
  DoSomething();     // maybe, enough
  DoSomethingMore(); // again
}
```

Rule OCR1082 Recommended Early return statement

Prefer early return condition rather than collecting indentations

Better write like this

```
int computeSumm (const int* theArray, unsigned int theSize)
{
  if (theArray == NULL || theSize == 0)
    return 0;

  ... Computing summ ...
  return aSumm;
}
```

rather than

```
int computeSumm (const int* theArray, unsigned int theSize)
{
  int aSumm = 0;
  if (theArray != NULL && theSize != 0)
  {
    ... Computing summ ...
  }
  return aSumm;
}
```

This could improve readability and reduce indentation depth.

Rule OCR1084 Mandatory No trailing spaces

Trailing spaces should be avoided.

Spaces at end of the line are useless.

Rule OCR1083 Recommended Headers order

Split headers into groups: system headers, headers of external libraries, headers of other modules, locally defined headers.

In each group, sort #include statements alphabetically.

This rule can improve readability, allows detection of useless header's multiple inclusions and makes 3rd-party dependencies clearly visible.

```
// system headers
#include <iostream>
#include <windows.h>

// Qt headers
#include <QDataStream>
#include <QString>

// OCCT headers
#include <gp_Pnt.hxx>
#include <gp_Vec.hxx>
#include <NCollection_List.hxx>
```

4. DOCUMENTATION RULES

The source code is one of the most important references for documentation. The comments in the source code should be complete enough to allow understanding of that code, and to serve as basis for other documents. The main reasons why comments are regarded as documentation and should be maintained are:

- The comments are easy to reach - they are always together with source code
- It s easy to update description in the comment when source is modified
- The source itself represents a good context to describe various details that would require much more explanations in separate document
- As a summary, this is the most cost-effective documentation

The comments should be compatible with Doxygen tool for automatic documentation generation (thus should use compatible tags).

Rule OCR1026 Mandatory Documenting classes

Each class should be documented in its header file (.hxx or .cdl). The comment should give enough details for the reader to understand the purpose of the class and main way of work with it.

Rule OCR1027 Mandatory Documenting class methods

Each class or package method should be documented in the header file (.hxx or .cdl). The comment should explain the purpose of the method, its parameters, and returned value(s).

Accepted style is:

```
//! Method computes the square value.
//! @param theValue – the input value;
//! @return squared value.
Standard_Export Standard_Real Square (Standard_Real theValue);
```

Rule OCR1030 Recommended Documenting C/C++ sources

It is very desirable to put comments in the C/C++ sources of the package/class.

They should be detailed enough to allow any person to understand what does each part of code, and get familiar with it. It is recommended to comment all static functions (like methods in headers), and at least each 10-100 lines of the function bodies.

There are also some rules that define how comments should be formatted, see section "Formatting Rules". Following these rules is important for good comprehension of the comments; moreover it makes possible to automatically generate user-oriented documentation directly from commented sources.

5. APPLICATION DESIGN

The following set of rules defines the common style of designing an OCCT application which should be applied by any developer contributing to the open source.

Rule OCR1037 Recommended Allow for possible inheritance

Try to design general classes (objects) keeping possible inheritance in mind.

This rule means that making possible extensions of your class the user should not encounter with problems of private implementations. Try to use protected members and virtual methods wherever you expect extensions in the future.

Rule OCR1039 Recommended Avoid friend declarations

Avoid using ‘friend’ classes or functions except some specific cases (ex., iteration)

‘Friend’ declarations increase coupling.

Rule OCR1041 Recommended Set/get methods

Avoid providing set/get methods for all fields of the class.

Intensive set/get functions break down encapsulation.

Rule OCR1042 Mandatory Hiding virtual functions

Avoid hiding a base class virtual function by a redefined function with a different signature.

Reason: Most of the compilers issue warning on this.

Rule OCR1043 Recommended Avoid mixing error reporting strategies

Try not to mix different error indication/handling strategies (exceptions or returned values) on the same level of an application.

Rule OCR1050 Mandatory Minimize compiler warnings

When compiling the source pay attention to and try to minimise compiler warnings.

Rule OCR1053 Recommended Avoid unnecessary inclusion

Try to minimise compilation dependencies by removing unnecessary inclusion.

6. GENERAL C/C++ RULES

This section defines rules for writing portable and maintainable OCCT C/C++ source code.

Rule OCR1036 Mandatory Wrapping of global variables

Use package or class methods returning reference to wrap global variables.

Reason: Following this rule reduces possible name space conflicts.

Rule OCR1035 Recommended Avoid private members

Use 'protected' members instead of 'private' wherever reasonable to enable future extensions. Use 'private' fields if future extensions should be disabled.

Rule OCR1051 Mandatory Constants and inlines over defines

Use constant variables (const) and inline functions instead of defines (#define).

Reason: Better control of types during development phase.

Rule OCR1052 Mandatory Avoid explicit numerical values

Avoid usage of explicit numeric values. Use named constants and enumerations instead.

Magic numbers are difficult to understand and troublesome in maintainance.

Rule OCR1058 Recommended Three mandatory methods

A class with any of (destructor, assignment operator, copy constructor) usually needs all of them.

Rule OCR1059 Recommended Virtual destructor

A class with virtual function(s) ought to have a virtual destructor.

Rule OCR1060 Recommended Default parameter value

Do not redefine a default parameter value in an inherited function.

Rule OCR1072 Recommended Use const modifier

Use *const* modifier wherever possible (functions parameters, return values etc.)

Rule OCR1079 Mandatory Usage of goto

Avoid *goto* statement except the cases of where it seems to improve code readability.

Rule OCR1025 Recommended Declaring variable in *for()* header

Declaring cycle variable in the header of the *for()* statement if not used out of cycle.

Rule OCR1081 Mandatory Condition statements within zero

Avoid usage of C-style comparison for non-boolean variables.

So you should write

```
void function (Standard_Integer theValue)
{
  if (theValue == 0)
    doSome();
}
```

rather than

```
void function (Standard_Integer theValue)
{
  if (!theValue)
    doSome();
}
```

The same rule for checking pointers:

```
void function (Standard_Real* thePointer)
{
  if (thePointer != NULL)
    doSome();
}
```

This improves code readability and simplifies understanding which condition code author was really checked.

7. PORTABILITY ISSUES

This chapter contains rules that are critical for cross-platform portability of OCCT.

Rule OCR1080 Mandatory Ensure code portability

It is required that OCCT source code must be portable to all platforms listed in the official 'Technical Requirements'. The term 'portable' here means 'able to be built from source'.

Reason: Breaking this rule may cause problems that are specific for particular platform, compiler, environment or circumstances; therefore such problems are very difficult to detect.

Rule OCR1023 Mandatory Avoid usage of global variables

Avoid usage of global variables.

Reason: Usage of global variables may cause problems of accessing them from another shared library.

Instead of global variables, use global (package or class) functions that return reference to static variable local to this function.

Another possible problem is the order of initialization of global variables defined in various libraries that may differ depending on platform, compiler and environment.

Rule OCR1054 Recommended Avoid explicit basic types

Avoid explicit usage of basic types (int, float, double etc.), use OCCT types (from package Standard) or specific *typedef* instead.

Rule OCR1055 Mandatory Use sizeof to calculate sizes

Do not assume sizes of types. Use sizeof instead to calculate sizes.

8. STABILITY ISSUES

The rules listed in this chapter are important for stability of the programs that use OCCT libraries.

Rule OCR1024 **Recommended** ***OSD::SetSignal()* to catch exceptions**

When using OCCT in an application, make sure to call OSD::SetSignal() function when the application is initialised.

Reason: This will install OCCT-specific C handlers for run-time interrupt signals and exceptions, so that low-level exceptions (such as access violation, division by zero etc.) will be properly caught by OCCT functions (that use **try {…} catch (Standard_Failure) {…}** blocks).

The above rule is especially important for robustness of modeling algorithms.

Rule OCR1034 **Recommended** **Cross-referenced handles**

Take care about cycling of handled references to avoid chains which will never be freed. For that purpose, use a pointer at one (subordinate) side.

See the following example:

```
In MyPackage.cdl:

class MyFirstHandle;
class MySecondHandle;
pointer MySecondPointer to MySecondHandle;
...

In MyPackage_MyFirstHandle.cdl:

class MyFirstHandle from MyPackage
...
is
...
  SetSecondHandleA (me: mutable; theSecond: MySecondHandle from MyPackage);
  SetSecondHandleB (me: mutable; theSecond: MySecondHandle from MyPackage);
...
fields
...
  mySecondHandle : MySecondHandle from MyPackage;
  mySecondPointer : MySecondPointer from MyPackage;
...
end MyFirstHandle from MyPackage;

In MyPackage_MySecondHandle.cdl:

class MySecondHandle from MyPackage
...
is
...
  SetFirstHandle (me: mutable; theFirst: MyFirstHandle from MyPackage);
...
fields
...
  myFirstHandle : MyFirstHandle from MyPackage;
...
end MySecondHandle from MyPackage;

In C++ code:

void MyFunction ()
{
  Handle(MyPackage_MyFirstHandle)  anObj1 = new MyPackage_MyFirstHandle;
  Handle(MyPackage_MySecondHandle) anObj2 = new MyPackage_MySecondHandle;
  Handle(MyPackage_MySecondHandle) anObj3 = new MyPackage_MySecondHandle;

  anObj1->SetSecondHandleA(anObj2);
  anObj1->SetSecondHandleB(anObj3);
  anObj2->SetFirstHandle(anObj1);
  anObj3->SetFirstHandle(anObj1);

  // Memory is not freed here !!!
  anObj1.Nullify();
  anObj2.Nullify();

  // Memory is freed here
  anObj3.Nullify();
}
```

Rule OCR1069 Recommended C++ memory allocation

In C++ use *new* and *delete* operators instead of *malloc* and *free*. Try not to mix different memory allocation techniques.

Rule OCR1070 Mandatory Match new and delete

Use the same form of new and delete:

```
aPtr1 = new TypeA[n]; ... ; delete[] aPtr1;
aPtr2 = new TypeB; ... ; delete aPtr2;
```

Rule OCR1071 Mandatory Methods managing dynamical allocation

Define a destructor, a copy constructor and an assignment operator for classes with dynamically allocated memory.

Rule OCR1073 Recommended Do not hide global new

Avoid hiding the global *new* operator.

Rule OCR1074 Recommended Assignment operator

In *operator*= assign to all data members and check for assignment to self.

Rule OCR1075 Recommended Float comparison

Don't check floats for equality or non-equality; use comparison operators with precision instead.

```
If (Abs (theFloat1 – theFloat2) < theTolerance) doSome();
```

Rule OCR1076 Recommended Non-indexed iteration

Avoid usage of iteration over non-indexed collections of objects. If such iteration is used, make sure that the result of the algorithm does not depend on order.

Reason: Since the order of iteration is unpredictable in this case, it frequently leads to different behavior of the application from one run to another, thus embarrassing the debugging process.

It mostly concerns mapped objects for which pointers are involved in calculating the hash function. For example, the hash function of TopoDS_Shape involves the address of TopoDS_TShape object. Thus the order of the same shape in the TopTools_MapOfShape will vary in different sessions of the application.

Rule OCR1077	Recommended	Do not throw in destructors
Do not throw exceptions from within destructor.		

Rule OCR1078	Mandatory	Assigning to reference
Avoid possible assignments of the temporary object to a reference.		

Reason: Different behaviour for different compiler of different platforms.

Such code (being OK for the modern C++) will produce problems on SGI and MS Visual 5.0 since these compilers will call destructor of the temporary object immediately at the end of the statement, regardless of the fact that reference to this object remains.

9. PERFORMANCE ISSUES

These rules define the ways of avoiding possible loss of performance caused by ineffective programming.

Rule OCR1045 Recommended Class fields alignment

In a class, declare its fields in the decreasing order of their size for better alignment.

Generally, try to reduce misaligned accesses since they impact the performance (for example, on Intel machines).

Rule OCR1048 Mandatory Fields initialization order

List class data members in the constructor's initialisation list in the order they are declared.

Rule OCR1049 Recommended Initialization over assignment

In class constructors prefer initialisation over assignment.

Rule OCR1057 Recommended Optimize caching

When programming procedures with extensive memory access, try to optimise them in terms of cache behaviour.

Here is an example of how cache behaviour can be impact:

On x86 this code

```
Standard_Real anArray[4096][2];
Standard_Integer i;
for (i = 0; i < 4096; i++) anArray[i][0] = anArray[i][1];
```

is more efficient than

```
Standard_Real anArray[2][4096];
Standard_Integer i;
for (i = 0; i < 4096; i++) anArray[0][i] = anArray[1][i];
```

since linear access (above) does not invalidate cache too often.

10. APPENDIX A. INDEX OF RULES

Page	Rule No,	Severity	Short description
4	OCR1011	Mandatory	Prefix for toolkits names
3	OCR1012	Recommended	Suggestive names
3	OCR1013	Mandatory	International language
4	OCR1014	Recommended	Naming of C++ class files
4	OCR1015	Mandatory	File names extensions
5	OCR1018	Recommended	Naming of variables
5	OCR1019	Recommended	Names of function parameters
5	OCR1020	Recommended	Names of class member variables
5	OCR1021	Recommended	Names of global variables
6	OCR1022	Recommended	Names of local variables
15	OCR1023	Mandatory	Avoid usage of global variables
16	OCR1024	Recommended	*OSD::SetSignal()* to catch exceptions
14	OCR1025	Recommended	Declaring variable in *for()* header
11	OCR1026	Mandatory	Documenting classes
11	OCR1027	Mandatory	Documenting class methods
11	OCR1030	Recommended	Documenting C/C++ sources
3	OCR1031	Recommended	Related names
4	OCR1032	Mandatory	Underscores in units names
16	OCR1034	Recommended	Cross-referenced handles
13	OCR1035	Recommended	Avoid private members
13	OCR1036	Mandatory	Wrapping of global variables
12	OCR1037	Recommended	Allow for possible inheritance
12	OCR1039	Recommended	Avoid friend declarations
12	OCR1041	Recommended	Set/get methods
12	OCR1042	Mandatory	Hiding virtual functions
12	OCR1043	Recommended	Avoid mixing error reporting strategies
7	OCR1044	Mandatory	International language
19	OCR1045	Recommended	Class fields alignment
7	OCR1046	Recommended	Line length
7	OCR1047	Recommended	C++ style comments
19	OCR1048	Mandatory	Fields initialization order
19	OCR1049	Recommended	Initialization over assignment
12	OCR1050	Mandatory	Minimize compiler warnings
13	OCR1051	Mandatory	Constants and inlines over defines
13	OCR1052	Mandatory	Avoid explicit numerical values
12	OCR1053	Recommended	Avoid unnecessary inclusion
15	OCR1054	Recommended	Avoid explicit basic types
15	OCR1055	Mandatory	Use sizeof to calculate sizes
7	OCR1056	Recommended	Get rid of unused code
19	OCR1057	Recommended	Optimize caching
13	OCR1058	Recommended	Three mandatory methods
13	OCR1059	Recommended	Virtual destructor
13	OCR1060	Recommended	Default parameter value
7	OCR1061	Mandatory	Indentation in sources
7	OCR1062	Recommended	Using spaces
8	OCR1063	Recommended	Separate logical blocks
8	OCR1064	Mandatory	Separate function bodies
8	OCR1065	Mandatory	Block layout
8	OCR1066	Recommended	Single-line operators
9	OCR1067	Recommended	Use alignment
9	OCR1068	Recommended	Indentation of comments
17	OCR1069	Recommended	C++ memory allocation
17	OCR1070	Mandatory	Match new and delete
17	OCR1071	Mandatory	Methods managing dynamical allocation
13	OCR1072	Recommended	Use const modifier

17	**OCR1073**	**Recommended**	**Do not hide global new**
17	**OCR1074**	**Recommended**	**Assignment operator**
17	**OCR1075**	**Recommended**	**Float comparison**
17	**OCR1076**	**Recommended**	**Non-indexed iteration**
18	**OCR1077**	**Recommended**	**Do not throw in destructors**
18	**OCR1078**	**Mandatory**	**Assigning to reference**
14	**OCR1079**	**Mandatory**	**Usage of goto**
15	**OCR1080**	**Mandatory**	**Ensure code portability**
14	**OCR1081**	**Mandatory**	**Condition statements within zero**
9	**OCR1082**	**Recommended**	**Early return statement**
9	**OCR1083**	**Recommended**	**Headers order**
9	**OCR1084**	**Mandatory**	**No trailing spaces**
3	**OCR1085**	**Recommended**	**Camel Case style**
6	**OCR1086**	**Recommended**	**Avoid dummy names**