



**RuleChecker & QualityChecker  
C++ Reference Manual**

---

*IBM Rational Logiscope*

*RuleChecker & QualityChecker C++ Reference Manual*

Before using this information, be sure to read the general information under “Notices” section, on page 139.

This edition applies to **VERSION 6.6, IBM Rational LOGISCOPE (product number 5724V81)** and to all subsequent releases and modifications until otherwise indicated in new editions.

© **Copyright IBM Corporation 1985, 2009**

US Government Users Restricted Rights—Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

---

# *About This Manual*

## **Audience**

This manual is intended for IBM® Rational® Logiscope™ *RuleChecker & QualityChecker* users for C++ source code verification.

## **Related Documents**

Reading first the following manual is highly recommended:

- *IBM Rational Logiscope - Basic Concepts.*
- *IBM Rational Logiscope - RuleChecker & QualityChecker - Getting Started.*

Creating new scripts to check specific / non standard programming rules is addressed in dedicated document:

- *IBM Rational Logiscope - Adding Java, Ada and C++ scriptable rules metrics and contexts.*

## **Overview**

### **C++ Project Settings**

Chapter 1 presents basic concepts of *Logiscope RuleChecker & QualityChecker C++*, its input and output data, its prerequisites and its limitations.

### **C++ Parsing Options**

Chapter 2 describes the way to adapt *Logiscope RuleChecker & QualityChecker C++* to the application. It also specifies the specifics of the C++ dialects supported by *Logiscope RuleChecker & QualityChecker C++*

### **Command Line Mode**

Chapter 3 specifies how to run *Logiscope RuleChecker & QualityChecker C++* using a command line interface.

---

## Standard Metrics

Chapter 4 specifies the metrics computed by *Logiscope QualityChecker C++*.

## Programming Rules

Chapter 5 specifies the programming rules checked by *Logiscope RuleChecker C++*.

## Customizing Standard Rules and Rule Sets

Chapter 6 describes the way to modify standard predefined rules and to create new ones with *Logiscope RuleChecker C++*.

# Conventions

The following typographical conventions are used:

<b>bold</b>	literals such as tool names ( <b>studio</b> ) and file extension ( <b>*.cpp</b> ),
<b><i>bold italics</i></b>	literals such as type names ( <b><i>integer</i></b> ),
<i>italics</i>	names that are user-defined such as directory names ( <i>log_installation_dir</i> ), notes and documentation titles,
typewriter	file printouts.

---

# Contacting IBM Rational Software Support

If the self-help resources have not provided a resolution to your problem, you can contact IBM® Rational® Software Support for assistance in resolving product issues.

**Note** If you are a heritage Telelogic customer, you can go to <http://support.telelogic.com/toolbar> and download the IBM Rational Telelogic Software Support browser toolbar. This toolbar helps simplify the transition to the IBM Rational Telelogic product online resources. Also, a single reference site for all IBM Rational Telelogic support resources is located at <http://www.ibm.com/software/rational/support/telelogic/>

## Prerequisites

To submit your problem to IBM Rational Software Support, you must have an active Passport Advantage® software maintenance agreement. Passport Advantage is the IBM comprehensive software licensing and software maintenance (product upgrades and technical support) offering. You can enroll online in Passport Advantage from <http://www.ibm.com/software/lotus/passportadvantage/howtoenroll.html>

- To learn more about Passport Advantage, visit the Passport Advantage FAQs at [http://www.ibm.com/software/lotus/passportadvantage/brochures\\_faqs\\_quickguides.html](http://www.ibm.com/software/lotus/passportadvantage/brochures_faqs_quickguides.html).
- For further assistance, contact your IBM representative

To submit your problem online (from the IBM Web site) to IBM Rational Software Support, you must additionally:

- Be a registered user on the IBM Rational Software Support Web site. For details about registering, go to <http://www-01.ibm.com/software/support/>.
- Be listed as an authorized caller in the service request tool

## Submitting problems

To submit your problem to IBM Rational Software Support:

1. Determine the business impact of your problem. When you report a problem to IBM, you are asked to supply a severity level. Therefore, you need to understand and assess the business impact of the problem that you are reporting.

Use the following table to determine the severity level.

Severity	Description
1	The problem has a <i>critical</i> business impact. You are unable to use the program, resulting in a critical impact on operation. This condition requires an immediate solution.
2	The problem has a <i>significant</i> business impact. The program is usable, but it is severely limited.
3	The problem has a <i>some</i> business impact. The program is usable, but less significant features (not critical to operation) are unavailable.
4	The problem has a <i>minimal</i> business impact. The problem causes little impact on operations or a reasonable circumvention to the problem was implemented.

2. Describe your problem and gather background information, When describing a problem to IBM, be as specific as possible. Include all relevant background information so that IBM Rational Software Support specialists can help you solve the problem efficiently. To save time, know the answers to these questions:
  - What software versions were you running when the problem occurred?  
To determine the exact product name and version, use the option applicable to you:
    - Start the IBM Installation Manager and select **File > View Installed Packages**. Expand a package group and select a package to see the package name and version number.
    - Start your product, and click **Help > About** to see the offering name and version number.
  - What is your operating system and version number (including any service packs or patches)?
  - Do you have logs, traces, and messages that are related to the problem symptoms?
  - Can you recreate the problem? If so, what steps do you perform to recreate the problem?
  - Did you make any changes to the system? For example, did you make changes to the hardware, operating system, networking software, or other system components?
  - Are you currently using a workaround for the problem? If so, be prepared to describe the workaround when you report the problem.

---

3. Submit your problem to IBM Rational Software Support. You can submit your problem to IBM Rational Software Support in the following ways:

- **Online:** Go to the IBM Rational Software Support Web site at <https://www.ibm.com/software/rational/support/> and in the Rational support task navigator, click **Open Service Request**. Select the electronic problem reporting tool, and open a Problem Management Record (PMR), describing the problem accurately in your own words.

For more information about opening a service request, go to <http://www.ibm.com/software/support/help.html>

You can also open an online service request using the IBM Support Assistant. For more information, go to <http://www-01.ibm.com/software/support/isa/faq.html>.

- **By phone:** For the phone number to call in your country or region, go to the IBM directory of worldwide contacts at <http://www.ibm.com/planetwide/> and click the name of your country or geographic region.
- **Through your IBM Representative:** If you cannot access IBM Rational Software Support online or by phone, contact your IBM Representative. If necessary, your IBM Representative can open a service request for you. You can find complete contact information for each country at <http://www.ibm.com/planetwide/>.

If the problem you submit is for a software defect or for missing or inaccurate documentation, IBM Rational Software Support creates an Authorized Program Analysis Report (APAR). The APAR describes the problem in detail. Whenever possible, IBM Rational Software Support provides a workaround that you can implement until the APAR is resolved and a fix is delivered. IBM publishes resolved APARs on the IBM Rational Software Support Web site daily, so that other users who experience the same problem can benefit from the same resolution.





---

# Table of Contents

Chapter 1	<b>C++ Project Settings</b>	
1.1	Starting a Logiscope Studio Session.....	1
1.2	Creating a Logiscope Project.....	2
1.3	Logiscope Repository .....	11
1.4	Relaxation Mechanism .....	12
Chapter 2	<b>C++ Parsing Options</b>	
2.1	Reserved Keywords.....	15
2.2	Choosing the Appropriate Dialect .....	17
2.2.1	Available C++ Dialects .....	17
2.2.2	Reference Documentation .....	18
2.2.3	Dialect Specifics.....	19
2.3	Configuring the Logiscope C++ Parser .....	23
2.3.1	Type - Syntax Item Association .....	23
2.3.2	Syntax of the Parser Configuration File .....	25
2.4	Managing pre-processing directives .....	26
2.4.1	Impact on Analysis Results .....	26
2.4.2	Restrictions.....	27
Chapter 3	<b>Command Line Mode</b>	
3.1	Logiscope create .....	29
3.1.1	Command Line Mode.....	29
3.1.2	Makefile mode.....	30
3.1.3	Options .....	31
3.2	Logiscope batch .....	33
3.2.1	Options .....	33
3.2.2	<i>Examples of Use</i> .....	34
Chapter 4	<b>Standard Metrics</b>	
4.1	Function Scope .....	36
4.1.1	Line Counting.....	36
4.1.2	Lexical and syntactic items .....	38
4.1.3	Data Flow .....	38
4.1.4	Halstead Metrics.....	41
4.1.5	Structured Programming .....	47
4.1.6	Control Flow .....	48
4.1.7	Relative Call Graph.....	49

4.2	Class Scope .....	52
4.2.1	Comments .....	52
4.2.2	Data Flow .....	52
4.2.3	Statistical Aggregates of Function Metrics .....	55
4.2.4	Inheritance Tree .....	58
4.2.5	Use Graph .....	59
4.3	Module Scope.....	60
4.3.1	Line Counting .....	60
4.3.2	Lexical and syntactic items.....	61
4.3.3	Data Flow .....	61
4.3.4	Halstead Metrics .....	62
4.4	Application Scope .....	64
4.4.1	Line Counting .....	64
4.4.2	Application Aggregates .....	65
4.4.3	Application Call Graph.....	66
4.4.4	Inheritance Tree .....	67
4.4.5	MOOD Metrics .....	69

## Chapter 5 **Programming Rules**

5.1	Rule Sets.....	77
5.2	Rule Scripts .....	78
5.2.1	Basic Rules .....	78
5.2.2	Customizable Rules .....	90
5.2.3	MISRA-C++ 2008 Programming Rules .....	105
5.3	Scott Meyers Rules .....	119

## Chapter 6 **Customizing Standard Rules and Rule Sets**

6.1	Modifying the Rule Set .....	125
6.2	Customizing Rules .....	126
6.3	Creating New Rules .....	142

## Chapter 7 **Notices**

# Chapter 1

---

## *C++ Project Settings*

A Logiscope project mainly consists in:

- the list of source files to be analysed,
- applicable source code parsing options according to the compilation environment,
- the verification modules to be activated on the source code files and the associated controls (e.g. metrics to be computed, rules to be checked).

A source file is a file containing C++ source code. This file is not necessarily compilable. It only has to conform to the C++ syntax.

Logiscope C++ projects can be created using:


- **Logiscope Studio Wizard:** a graphical interface requiring a user interaction, as described in the following sub-sections introducing the Logiscope C++ project settings,
- **Logiscope Create:** a tool to be used from a standalone command line or within makefiles, please refer to Chapter *Command Line Mode* to learn how to create a Logiscope project using **Logiscope Create**.

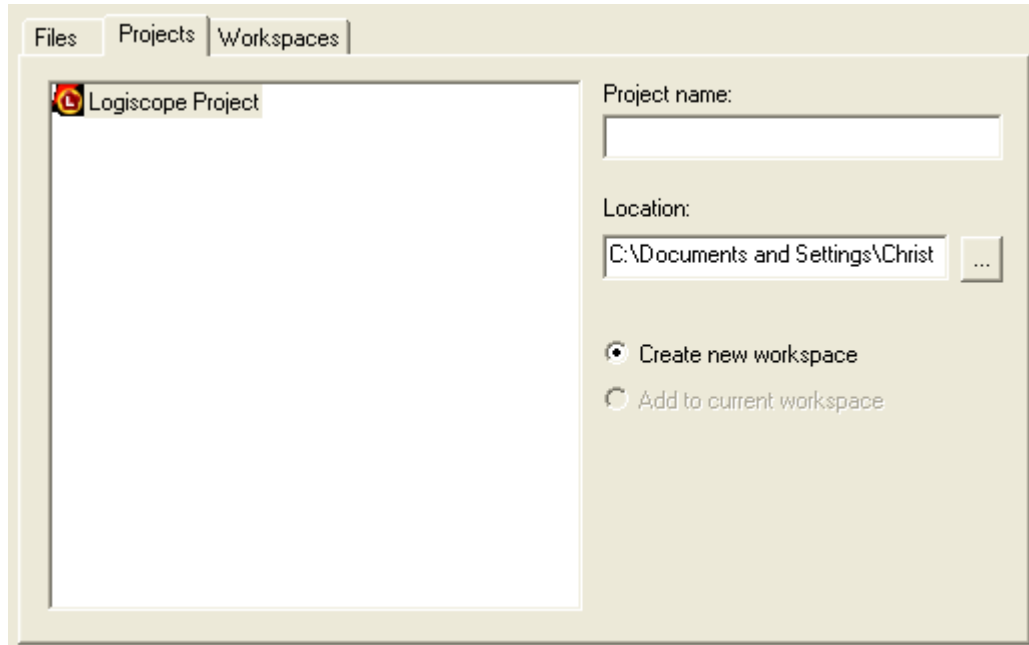
### 1.1 Starting a Logiscope Studio Session

To begin a Logiscope **Studio** session:

- On UNIX (i.e. Solaris or Linux):
  - launch the **vcs** binary .
- On Windows:
  - click the **Start** button and select the **IBM Rational Logiscope <version>** item in the **IBM Rational Programs Group**.

## 1.2 Creating a Logiscope Project

Once the Logiscope Studio main window is displayed, select the **New...** command in the File menu or click on the  icon, you get the following dialog box:



The **Project name:** pane allows to enter the name for the new Logiscope project to be created.

**Location:** allows to specify the directory where the Logiscope project and the associated Logiscope repository will be created. For more details, see the next section.

By default, the project name is automatically added to the specified location. This implies that a subdirectory named <ProjectName> is automatically created.

## Defining the type of the Logiscope project

The following **Logiscope Project Definition** dialog box appears:



The **Project Language:** is the programming language in which are written the source code files to be analysed. Of course, select C++.

Note: Only one language can be selected. If your application contains source code files written in several languages e.g. C and C++ source files, you should create several distinct Logiscope projects: one for each language.

The **Project Modules:** lists the verification modules to be activated on the source files of the project .

For instance, you can select both **QualityChecker** and **RuleChecker**.

Notes: At least one module should be selected. The **TestChecker** module cannot be selected with an other module.

For more details on *TestChecker* module, please refer to *IBM Rational Logiscope - TestChecker - Getting Started*.

For more details on *CodeReducer* module, please refer to *IBM Rational Logiscope - CodeReducer - Getting Started*.

## Specifying the source files to be analysed

The **Project Source Files** dialog box allows to specify what source files are to be analysed and where they are located.



**Source files root directory** shall specify the directory including all the source files to be analyzed.

If necessary, use the **Directories** choice to select the list of repertories covering the application source files.

- **Include all subdirectories** means that selected files will be searched for in every sub-directory of the source file root directory.
- **Do not include subdirectories** means that only files included in the application directory will be selected.
- **Customize subdirectories to include** allows the user to select the list of directories that include application files through a new page.

**Suffixes** choices allow to specify applicable source, header and inline file extensions needed in the above selected directories. Extensions shall be separated with a semi-colon.

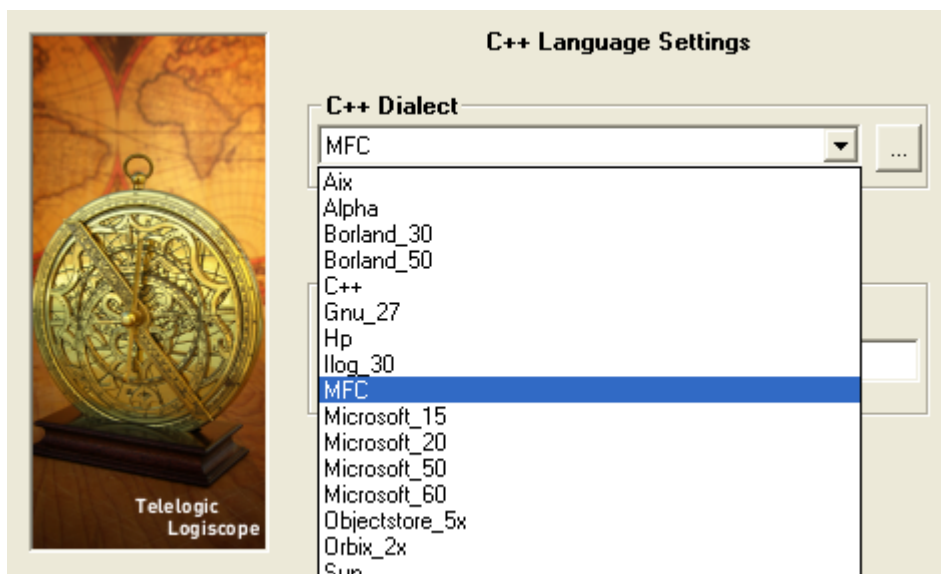
## Setting Parsing Options

The next dialog box allows to set up C++ source code parsing options:



**C++ Dialect:** A dialect is used to specify parsing actions associated to some types, “keywords” according to the source code compiler specifics.

For more details on available dialects, please refer to the next chapter *Parsing Options*.



**Preprocessor:** The source code files to be analyzed may contain some preprocessing directives (e.g. `#ifdef`). In some cases, these directives can lead to parsing errors and warnings by breaking up the code structure.

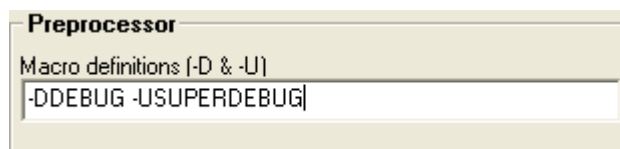
Logiscope allows to parse C++ files taking into account part of the preprocessing directives.

In the Macro definitions (-D & -U) pane, you can define and or undefine some preprocessing options by respectively using:

- **-D**<name>: defines <name> as if it were in a `#define` directive.
- **-U**<name>: considers <name> as undefined as if it were part of an `#undef` directive.

The number of occurrences of option **-D** and/or option **-U** is unlimited.

In the example below, the `DEBUG` option is defined, so the corresponding conditional code will be parsed. The `SUPER_DEBUG` option is considered as undefined so the corresponding conditional options will not be parsed.



For more details on the **Preprocessor**: settings, please refer to the next chapter *Parsing Options*.

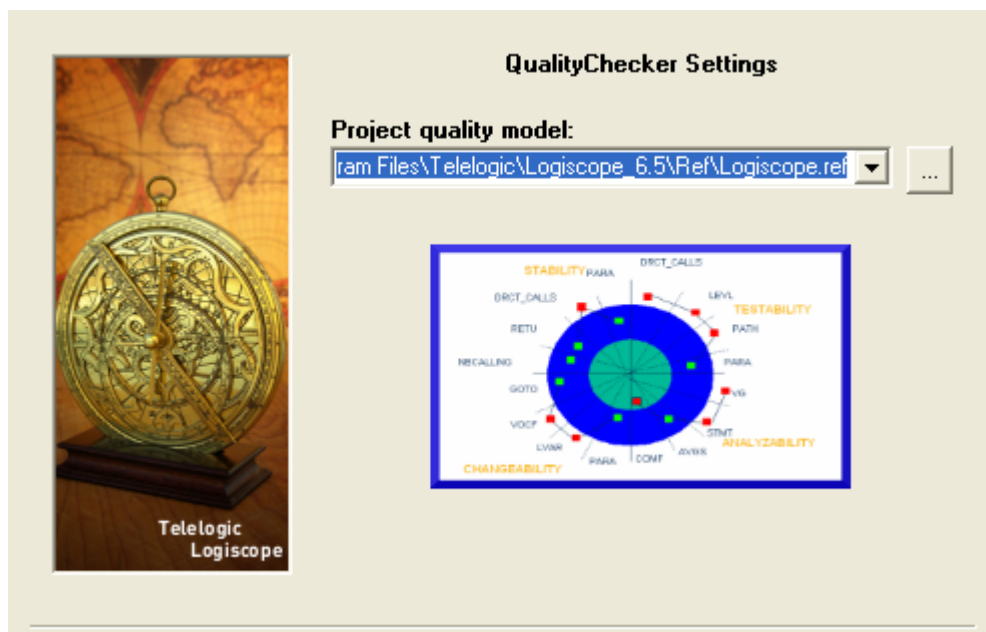


## Setting QualityChecker Parameters

The next dialog box allows to specify the applicable **Project quality model**: how the *QualityChecker* module evaluates software quality characteristics (e.g. Maintainability) based on a standard factors / criteria / metrics approach.

Note: Quality models are textual files (also called Reference files). Default quality models are provided with the standard Logiscope installation. They should be customized to take into account the verification objectives and contexts applicable to the project.

For more information, see the *IBM Rational Logiscope - Basic Concepts* manual.

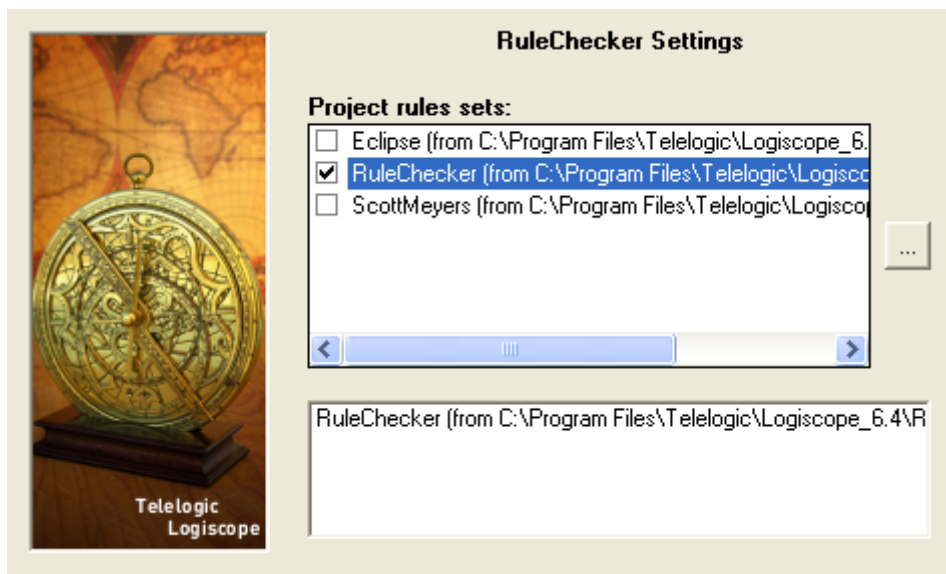


For your project verification, you should define and select your own applicable quality model.

## Setting RuleChecker Parameters

The **RuleChecker Settings** dialog box allows to specify the applicable **Project rule sets**: i.e. the rules / coding standards the *Logiscope RuleChecker* module shall verify on the project source files.

For more details on available rules and rule sets, please refer to the chapter *Standard Programming Rules*.

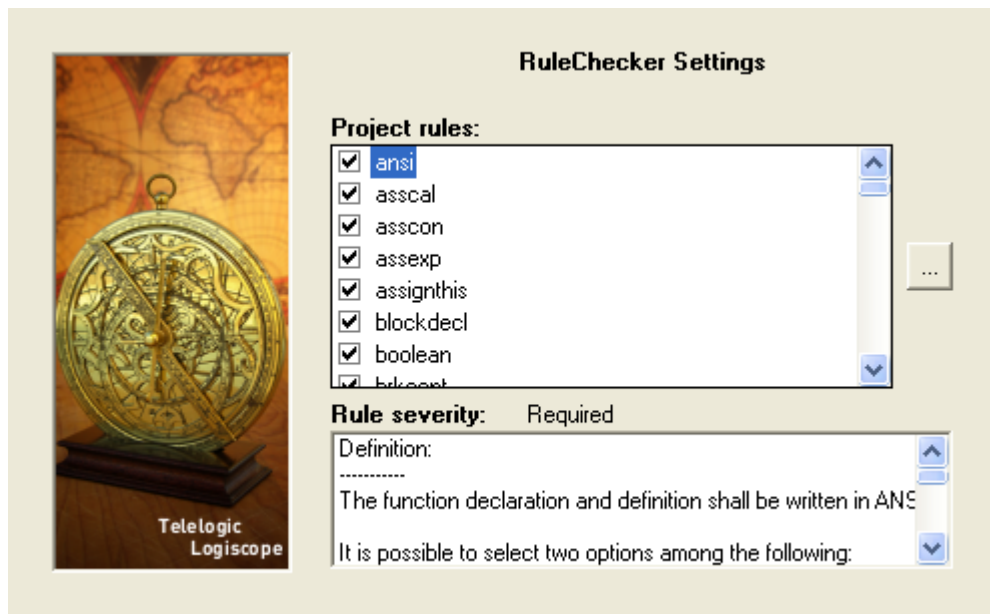


At least one rule set should be selected for the *Logiscope RuleChecker* projects.

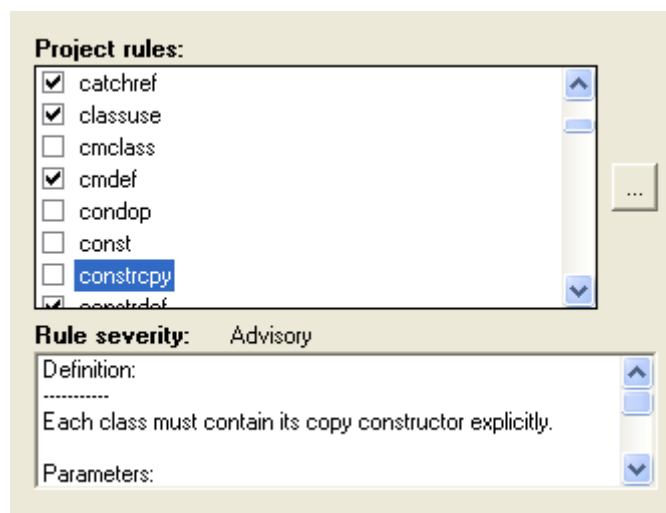
Several rule sets can be selected. If so, *Logiscope RuleChecker* will check the union of the rules specified in all selected rule sets.

The next **RuleChecker Settings** dialog box allows to fine tune the list of **Project rules**. It is possible to select or unselect some of the rules available.

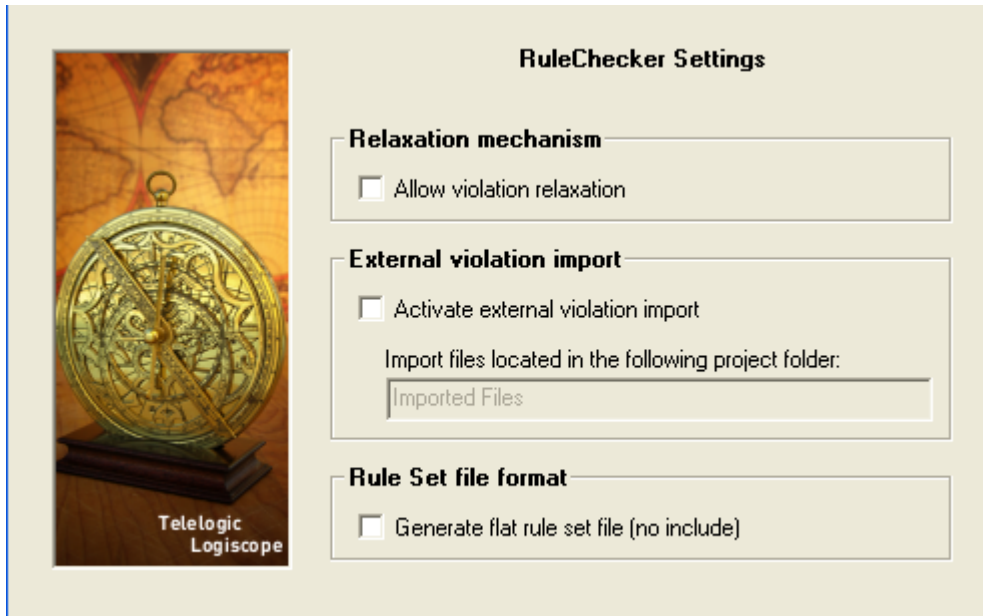
The rules that are selected are those listed in the Project rule sets selected in the previous **RuleChecker Settings** dialog box



You can check / uncheck the rules. The description of the selected rule and the rule severity are displayed in the bottom pane.



The last **RuleChecker Settings** dialog box allows to use some advanced features of the *Logiscope RuleChecker* module.



**Relaxation mechanism:** when the box is checked, rule violations can be relaxed using special comments in the code. For more details, please refer to the next section.

**External violation import:** when the box is checked, the files in the specified project folder can be used to import violations generated by an external tool. For more details, please refer to the *IBM Rational Logiscope - RuleChecker & QualityChecker - Getting Started* document.

**Rule set file format:** when the box is checked, the project rule set file (i.e. with a “.rst”) extension) that is generated for the project doesn’t contain any includes of other rule set files. It will contain an expanded copy of the contents of any rule sets that were used for the project.

For more details, please refer to the Chapter *Customizing Rules and Rule Sets*.

## 1.3 Logiscope Repository

The Logiscope Repository is the directory where Logiscope will create and maintain all internal files storing the necessary information. The Logiscope Repository is specified using the **location** pane in the Project Creation window (see previous section).

At the end of the of a Logiscope project creation process, the following files are generated in the Logiscope Repository:

- <ProjectName>.**ttw** for Logiscope workspace,
- <ProjectName>.**ttp** for Logiscope project,
- <ProjectName>.**rst** for Logiscope Rule Set.

Once a Logiscope project has been “built”: i.e. the source files of the project have been parsed to extract all necessary information for code verification, a Logiscope folder is created containing several Logiscope internal ASCII format files among which:

- a file named **standards.chk** containing all the violations found for the source code file of the project under analysis.
- a control graph file (suffixed by **.cgr**) for each source code file,
- global analysis result files (suffixed by **.dat**, **.tab** and **.graph**).

All files stored in the Logiscope Repository are internal data files to be used by Logiscope **Studio**, **Viewer** and **Batch**. They are not intended to be directly used by Logiscope users. The format of these files is clearly subject to changes.

## 1.4 Relaxation Mechanism

When **Relaxation mechanism** is activated for a Logiscope RuleChecker project, rule violations that have been checked and that you have decided are acceptable exceptions to the rule, can be relaxed for future builds: they will no longer appear in the list of rule violations. This can be very useful when checking violations in a context where multiple reviews are performed.

The violations that have been relaxed will remain accessible for future reference in the Relaxed Violations folder.

The relaxation mechanism is based on comments inserted into the code where the tolerated violations are. There are two ways to do this, depending on whether there is a single rule violation to relax on the line, or multiple ones to relax on the given line.

### Relaxing a single rule violation

If there is a single violation to relax, it can be done as a comment on the same line as the code, using the following syntax:

```
some code // %RELAX<rule_mnemonic> justification
```

where:

- `rule_mnemonic`: is the mnemonic of the rule that you want to ignore violations of on the current line.
- `justification`: is free text, allowing to justify the relaxation of the rule violation.

If justification carries over several lines, they will not be included as part of the justification of the relaxation. In order for the justification to be written on several lines, the second syntax which is presented in the next section should be used.

### Relaxing several violations and/or adding a longer justification

If there are several violations to relax for a same line (several violations occurring in different places in the code at the same time cannot be relaxed), or if the justification of the violation should have several lines, the following syntax should be used.

```
// >RELAX<rule_mnemonic> justification
```

followed by any number of empty lines, comment lines, or relaxations of other rules relating to the same code line, then by the code line of the violation.

## Relaxing all violations in pieces of code

If all the violations of one or more rules are to be relaxed in a given piece of code (e.g. reused code included in a newly developed file), the piece of code should be surrounded by:

```
//  {{RELAX<list_of_rule_mnemonics> justification  
the piece of code  
//  }}RELAX<list of rule mnemonics>
```

where:

- `list_of_rule_mnemonics`: is the list of all mnemonics of the rules that you want to ignore violations of on the piece of code.

The rule mnemonics shall be separated by a comma.





# Chapter 2

---

## *C++ Parsing Options*

Logiscope uses source code parsers to extract all necessary information from the source code files specified in the project under analysis.

As the source code under analysis may contain compiler specifics, this chapter first details the available options to adapt the default behavior of the Logiscope C++ parsing to such specifics. They involve:

- choosing the appropriate dialect (cf.2.2),
- configuring the Logiscope C++ parser (cf. 2.3),
- managing pre-processing directives (cf. 2.4).

### 2.1 Reserved Keywords

The source code shall respect the C++ syntax defined in the reference document:

*"Working Paper for Draft Proposed International Standard for Information Systems - Programming Language C++"*, by Andrew Koenig, referenced X3J16/96-0225 WG21/N1043, dated: December, 1996.

Because of the use of a parser configuration file to define type specifiers, type qualifiers and access specifiers (cf. section 2.4), the list of keywords for *Logiscope C++* is smaller than the list of keywords of the language.

The list of C++ reserved words is the following:

```
asm
break
case
catch
class
const_cast
continue
default
delete
do
dynamic_cast
extern
else
enum
false
for
goto
```

```
if  
namespace  
new  
operator  
reinterpret_cast  
return  
sizeof  
static_cast  
struct  
switch  
template  
this  
throw  
true  
try  
typeid  
typename  
typeof  
union  
using  
while
```

The \$ character is not authorized in identifiers.

## 2.2 Choosing the Appropriate Dialect

### 2.2.1 Available C++ Dialects

Compilers allow specifics that may not be correctly handled by the default Logiscope C++ parsing. To consider those specifics when parsing the source code and thus avoid parsing errors and warnings, the user shall choose the appropriate C++ dialect when setting up the project.

The list of available C++ dialects is the following:

- **Aix** for the IBM C++ 3.1 dialect
- **Alpha** for the DIGITAL C++ 6.0 dialect.
- **Borland\_30** for the BORLAND C++ 3.0 dialect,
- **Borland\_50** for the BORLAND C++ 5.0 dialect,
- **C++** for standard ISO C++,
- **Gnu\_27** for the GNU 2.7 dialect,
- **Hp** for the HP C++ dialect,
- **Ilog\_30** for the Ilog 3.0 dialect,
- **MFC** for the Microsoft Foundations Classes dialect,
- **Microsoft\_15** for the MICROSOFT C++ 1.5 dialect,
- **Microsoft\_20** for the MICROSOFT C++ 2.0 dialect,
- **Microsoft\_50** for the MICROSOFT C++ 5.0 dialect,
- **Microsoft\_60** for the MICROSOFT C++ 6.0 dialect,
- **Rhapsody** for source code file generated by *IBM Rational Rhapsody*,
- **Object\_5x** for the ObjectStore 5 dialect,
- **Orbix\_2x** for the Orbix 5 dialect,
- **sun** for the SUN C++ dialect,

The specifics of each dialect are specified in the following sections.

## 2.2.2 Reference Documentation

### AIX

IBM C++ Compiler User's Guide  
5/2/96 xIC 3.1 1.69

### Alpha

DIGITAL C++ Version 6.0 for DIGITAL  
C++ Programming Language, Third Edition, by Bjarne Stroustrup

### Borland C++ 3.0

Borland TURBO C++ 3.0  
User's Guide

### Borland C++ 5.0

Borland C++ 5.0 Development Suite  
CD User's Guide

### GNU 2.7

Info file ``gcc'` made from the Texinfo source file `gcc.texinfo`.

### HP C++

The C++ Programming Language  
Bjarne Stroustrup  
Second Edition  
Addison-Wesley Publishing Company, 1991.

### Microsoft C++ 1.5

Extract related to MICROSOFT C++ 1.5 language of the compact disk  
Microsoft Visual C++  
Development System and Tools for Windows

### Microsoft C++ 2.0, 5.0, and 6.0

Microsoft Visual C++  
Development System and Tools for Windows

### SUN C++

SPARCCompiler (Extract on the compact disk)  
C++ 4.0 Language System

## 2.2.3 Dialect Specifics

### Aix

The following keywords are recognized:

**`__offsetof`**  
**`__System`**

### Alpha

The following keywords are recognized:

**`__builtin_sizeof`**  
**`__builtin_isfloat`**

### Borland C++ 3.0

The following keywords are recognized:

<b><code>__cdecl</code></b>	<b><code>cdecl</code></b>
<b><code>__far</code></b>	<b><code>far</code></b>
<b><code>__fastcall</code></b>	<b><code>fastcall</code></b>
<b><code>__huge</code></b>	<b><code>huge</code></b>
<b><code>__interrupt</code></b>	<b><code>interrupt</code></b>
<b><code>__loadds</code></b>	
<b><code>__near</code></b>	<b><code>near</code></b>
<b><code>__pascal</code></b>	<b><code>pascal</code></b>
<b><code>__saveregs</code></b>	
<b><code>__seg</code></b>	

The following keywords are not recognized: **`__asm`**

### Borland C++ 5.0

The following keywords are recognized:

<b><code>__cdecl</code></b>	<b><code>cdecl</code></b>	
<b><code>__cs</code></b>		
<b><code>__declspec</code></b>		
<b><code>__ds</code></b>	<b><code>__ds</code></b>	
<b><code>__es</code></b>	<b><code>__es</code></b>	
<b><code>__except</code></b>		
<b><code>__export</code></b>	<b><code>__export</code></b>	
<b><code>__far</code></b>	<b><code>__far</code></b>	<b><code>far</code></b>
<b><code>__fastcall</code></b>	<b><code>__fastcall</code></b>	
<b><code>__huge</code></b>	<b><code>__huge</code></b>	<b><code>huge</code></b>
<b><code>__import</code></b>	<b><code>__import</code></b>	
<b><code>__interrupt</code></b>	<b><code>__interrupt</code></b>	<b><code>interrupt</code></b>

<b>__loadds</b>	<b>_loadds</b>	
<b>__near</b>	<b>_near</b>	<b>near</b>
<b>__pascal</b>	<b>_pascal</b>	<b>pascal</b>
<b>__rtti</b>		
<b>__saveregs</b>	<b>_saveregs</b>	
<b>__seg</b>	<b>_seg</b>	
<b>__ss</b>		
<b>__thread</b>		

The following keywords are not recognized:

<b>__asm</b>	<b>_asm</b>
<b>__finally</b>	<b>__try</b>

## Gnu 2.7

The following keywords are recognized:

<b>bool</b>	
<b>true</b>	<b>false</b>
<b>explicit</b>	
<b>mutable</b>	
<b>__alignof</b>	<b>__alignof__</b>
<b>__asm</b>	<b>__asm__</b>
<b>__attribute</b>	<b>__attribute__</b>
<b>__const</b>	<b>__const__</b>
<b>__extension__</b>	
<b>__inline</b>	<b>__inline__</b>
<b>__label__</b>	
<b>__signed</b>	<b>__signed__</b>
<b>__typeof</b>	<b>__typeof__</b> <b>typeof</b>
<b>__volatile</b>	<b>__volatile__</b>
<b>__wchar_t</b>	

The following keywords are not recognized:

**<?=>?=**  
**<?>?**  
**signature**

Overmore, the Gnu specific construction:

```
#define AAAA(prefix, string, args...) fprintf(stderr, prefix string, ##args)
```

is not supported.

## Microsoft C++ 1.5

The following keywords are recognized:

<b>__based</b>	<b>_based</b>	
<b>__cdecl</b>	<b>_cdecl</b>	<b>cdecl</b>

<b>__export</b>	<b>_export</b>	
<b>__far</b>	<b>_far</b>	<b>far</b>
<b>__fastcall</b>	<b>_fastcall</b>	
<b>__fortran</b>	<b>_fortran</b>	
<b>__huge</b>	<b>_huge</b>	<b>huge</b>
<b>__inline</b>	<b>_inline</b>	
<b>__interrupt</b>	<b>_interrupt</b>	
<b>__loadds</b>	<b>_loadds</b>	
<b>__near</b>	<b>_near</b>	<b>near</b>
<b>__pascal</b>	<b>_pascal</b>	
<b>__saveregs</b>	<b>_saveregs</b>	
<b>__segment</b>	<b>_segment</b>	
<b>__segname</b>	<b>_segname</b>	

## Microsoft C++ 2.0

The following keywords are recognized:

<b>__based</b>	<b>_based</b>	
<b>__cdecl</b>	<b>_cdecl</b>	<b>cdecl</b>
<b>__cdeclspec</b>	<b>_cdeclspec</b>	
<b>__except</b>		
<b>__fastcall</b>	<b>_fastcall</b>	
<b>__inline</b>	<b>_inline</b>	
<b>__int8</b>	<b>_int8</b>	
<b>__int16</b>	<b>_int16</b>	
<b>__int32</b>	<b>_int32</b>	
<b>__int64</b>	<b>_int64</b>	
<b>__leave</b>		
<b>__stdcall</b>	<b>_stdcall</b>	

The following keywords are not recognized:

**\_\_finally**  
**\_\_try**

## Microsoft C++ 5.0 and 6.0

The following keywords are recognized:

**\_\_based**  
**\_\_cdecl**  
**\_\_cdeclspec**  
**\_\_dllexport**  
**\_\_dllimport**  
**\_\_except**  
**\_\_fastcall**  
**\_\_inline**  
**\_\_int16**  
**\_\_int32**

**\_\_int64**  
**\_\_int8**  
**\_\_leave**  
**\_\_multiple\_inheritance**  
**naked**  
**\_\_single\_inheritance**  
**\_\_stdcall**  
**thread**  
**\_\_virtual\_inheritance**

The following keywords are not recognized:

**\_\_asm**  
**\_\_finally**  
**\_\_try**

## Rhapsody

A source code file is considered as a generated file if it contains one of the following:

- a comment starting by #[,
- a comment starting by ##,
- a comment containing `///  
Generated Date .:`

Otherwise, the source code file is considered as “hand written”.

In generated file, user-written code is detected using the following rules:

- a line following a line starting by a comment starting with ## is a user code line,
- a line ending with comment starting with ## is a user code line,
- a line starting with a comment starting with #[ open a block of user code ending on a line starting with a comment starting with #].

Exception: if a comment ## is followed by « auto\_generated » it does not introduce user code.



## 2.3 Configuring the Logiscope C++ Parser

In fact, each dialect is associated to a textual file that specifies the dialect specifics: the parser configuration file.

The default parser configuration files are located in the directory **data/env\_c++/** in the Logiscope installation directory.

If the C++ dialect or the C++ library used is not supported by one of the standard C++ dialects, the user can customize an existing Logiscope C++ parser configuration file to better suit the application source code syntax specifics.

### 2.3.1 Type - Syntax Item Association

The parser configuration file allows the description of specific types, keywords and macros in order to improve the source code parsing. For each identifier, a type may be associated. This type corresponds to an item of the C++ syntax.

The following table details the relation between the type of identifiers and the C++ syntax.

Type	Syntax item
IDENTIFIER	A simple identifier. Allows to mask predefined keywords.
STORAGE_CLASS	static, extern, register, ...
TYPE_SPECIFIER	int, char, float, unsigned, ...
TYPE_NAME	Allows to specify an identifier is a type name.
TYPE_QUALIFIER	const, volatile, ...
ACCESS_SPECIFIER	private, public, protected, ...
STRING_MACRO	A macro defined as a character string "..."
EXPRESSION_MACRO	A macro defined as an expression 3, t[i], f(a, b), ...
STATEMENT_MACRO	A macro defined as a statement a = 3; , f(a, b); , ...
DECLARATION_MACRO	A macro defined as a declaration int a; , myclass obj(a,b); , ...
TYPE_MACRO	A macro defined as a typename mytempl<int> , x##_ptr, ...
OPEN_BLOCK_MACRO	A macro that replaces {
CLOSE_BLOCK_MACRO	A macro that replaces }

OPEN_LOOP_MACRO	A macro that replaces for(;;) {
CLOSE_LOOP_MACRO	A macro that replaces } corresponding to for(;;) {
COMMENT_MACRO	A macro that replaces a comment
STRING_FUNC_MACRO	A macro function defined as a character string " ... "
EXPRESSION_FUNC_MACRO	A macro function defined as an expression 3, t[i], f(a, b), ...
STATEMENT_FUNC_MACRO	A macro function defined as a statement a = 3; , f(a, b); , ...
DECLARATION_FUNC_MACRO	A macro function defined as a declaration int a; , myclass obj(a,b); , ...
TYPE_FUNC_MACRO	A macro function defined as a typename mytempl<int> , x##_ptr, ...
OPEN_BLOCK_FUNC_MACRO	A macro function that replaces {
CLOSE_BLOCK_FUNC_MACRO	A macro function that replaces }
OPEN_LOOP_FUNC_MACRO	A macro function that replaces for(;;) {
CLOSE_LOOP_FUNC_MACRO	A macro function that replaces } corresponding to for(;;) {
COMMENT_FUNC_MACRO	A macro function that replaces a comment
SQL_MACRO_START	Starts an SQL embedded statement. EXEC
SQL_MACRO_TYPE	Type of SQL embedded statement. SQL, ORACLE, IAF, ...

The definition of **...\_FUNC\_MACRO** types allows to pass parameters to these macros that should not be allowed for function calls.

Examples:

```
LIST_MAP(mylist, char *, str)
OPER_NAME(struct)
```

## 2.3.2 Syntax of the Parser Configuration File

The EBNF notation is used to describe the syntax of the C++ parser configuration file.

```

<conf> ::= "%START_C_CONFIGURATION" <head> <defs>
"%END_C_CONFIGURATION"
<head> ::= "=" <init> ";"
<init> ::= {"copy" <predefined>} | {"add" <predefined>}
<predefined> ::= "EMPTY" | "C++"
<defs> ::=
    | <defs> <def>
<def> ::= <type> ":" <idents> ";"
<type> ::= "%IDENTIFIER"
    | "%STORAGE_CLASS"
    | "%TYPE_SPECIFIER"
    | "%TYPE_QUALIFIER"
    | "%TYPE_NAME"
    | "%ACCESS_SPECIFIER"
    | "%STRING_MACRO"
    | "%EXPRESSION_MACRO"
    | "%STATEMENT_MACRO"
    | "%DECLARATION_MACRO"
    | "%TYPE_MACRO"
    | "%OPEN_BLOCK_MACRO"
    | "%CLOSE_BLOCK_MACRO"
    | "%OPEN_LOOP_MACRO"
    | "%CLOSE_LOOP_MACRO"
    | "%COMMENT_MACRO"
    | "%STRING_FUNC_MACRO"
    | "%EXPRESSION_FUNC_MACRO"
    | "%STATEMENT_FUNC_MACRO"
    | "%DECLARATION_FUNC_MACRO"
    | "%TYPE_FUNC_MACRO"
    | "%OPEN_BLOCK_FUNC_MACRO"
    | "%CLOSE_BLOCK_FUNC_MACRO"
    | "%OPEN_LOOP_FUNC_MACRO"
    | "%COMMENT_FUNC_MACRO"
    | "%CLOSE_LOOP_FUNC_MACRO"
    | "%SQL_MACRO_START"
    | "%SQL_MACRO_TYPE"
<idents> ::=
    | <idents> <ident>
<ident> ::= [a-zA-Z0-9_][a-zA-Z0-9_]*

```

Comments begin with `/*` and end with `*/`. They cannot be nested.

Separators are blanks, tabulations, ends of lines, and comments.

## 2.4 Managing pre-processing directives

C++ source code can be analyzed either expanded or not, this means after or before use of the preprocessor. The user shall supply Logiscope with the more suitable source code according to the analysis goals. The explanations below are intended to help choosing between these two solutions.

### 2.4.1 Impact on Analysis Results

As a general rule, if the purpose is to assess the maintainability of the software, the non-expanded source code suits better as it is near the developer point of view. For example, a piece of source code with a low complexity but using a lot of macro calls, can have a very high complexity after the preprocessing. Analyzing preprocessed code can generate unjustified alarms. In the same way, a piece of source code with a high complexity because of the use of a lot of `#if` statements, can be very simple after preprocessing. Analyzing preprocessed code can omit to raise important alarms.

More detailed considerations have to be taken into account. Within non-expanded code, conditional statements (`#if`, `#ifdef`, ...) are considered as `if` statements. Macro calls are considered as function calls.

Analyzing non preprocessed code has an influence on measurements when the same variable, the same type or the same function is declared in both branches of a `#if ... #else ... #endif`.

In the following example, the number of declared variables is equal to 2 instead of 1.

```
...
#ifdef POSIX
void *ptr;
#else
char *ptr;
#endif
...
```

In the same way, the following source code has a number of functions equal to 2.

```
...
#ifdef POSIX
int fn()
{ ... }
#else
int fn()
{ ... }
#endif
...
fn();
```

Choosing between both solutions shall be done according to the analysis goals and programming styles (macro often used or not, for example).

## 2.4.2 Restrictions

The Logiscope way of parsing source code imposes restrictions on the use of preprocessing statements in C++ programs. A file which does not follow the restrictions may be incompletely parsed by Logiscope (this yields *syntax error...* messages).

The main limitations are:

- Only the following macro types are allowed:
  - Macros used in place of an identifier or an expression.

Example:

```
#define ZERO 0
...
a=ZERO;
```

- Macros used in place of a statement or a declaration.

Example:

```
#define PERROR(errno) .....
...
if (ret_code < 0)
    PERROR (7);
```

- Macros used in place of the beginning or the end of a block.

Example:

```
#define WHEN(x) if (x) {
#define END }
...
WHEN(ret_code < 0)
...
END
```

Among the above three types of macros, only part of the first one can be parsed without using the configuration file. The following example shows the use of invalid macros.

:

```
#define IS_NEG < 0
#define STRUCT(x) struct x

foo () {

    STRUCT(point) pt1; /* should be defined as TYPE_FUNC_MACRO */

    if (i IS_NEG)      /* invalid */
        i=0;

}
```

- The preprocessing directives can only be located in a place where an instruction, a declaration or an expression can be found. A preprocessing directive must not "cut" a declaration or an instruction.

Examples of invalid source code:

```
main () {
  int
  #ifdef OK      /* #ifdef inside a declaration */
  i;
  #else
  j;
  #endif

  #ifdef OK
  if (i == 1)    /* invalid */
  #else
  if (i == 0)
  #endif
  printf("OK\n") ;
  else
  printf("ERROR\n") ;
  }
  char *day_name (int n) {
  static char *name[] = {
  #ifdef FRENCH  /* invalid */
  "jour inconnu",
  "lundi", "mardi", "mercredi", "jeudi",
  "vendredi", "samedi", "dimanche"
  #else
  "unknown day",
  "Monday", "Tuesday", "Wednesday", "Thursday",
  "Friday", "Saturday", "Sunday"
  #endif
  } ;
  return (n < 1 || n > 7) ? name[0] : name[n];
  }
```

# Chapter 3

---

## *Command Line Mode*

### 3.1 Logiscope create

Logiscope projects: i.e. “.ttp” file are usually built using Logiscope **Studio** as described in chapter *Project Settings* or in the *Logiscope RuleChecker & QualityChecker Getting Started* documentation.

The logiscope **create** tool builds Logiscope projects from a standalone command line or within makefiles (replacing the compiler command) .

#### 3.1.1 Command Line Mode

When started from a standard command line, The **create** tool creates a new project file with the information provided on the command line.

For a complete description of the command line options, please refer to the Command Line Options paragraph.

When used in this mode, there are two different ways for providing the files to be included into the project:

##### **Automatic search**

This is the default mode where the tool automatically searches the files in the directories. Key options having effect on this modes are:

**-root <root\_dir>** : the root directory where the tool will start the search for source files. This option is not mandatory, and if omitted the default is to start the search in the current directory.

**-recurse** : if present indicates to the tool that the search for source files has to be recursive, meaning that the tool will also search the subdirectories of the root directory.

##### **File list**

In this mode, the tool will look for the **-list** option which has to be followed by a file name. This provided file contains a list of files to be included into the project. The file shall contain one filename per line.

**Example:** Assuming a file named `filelist.lst` containing the 3 following lines:

```
/users/logiscope/samples/C++/Hangman/GenericDlg.cpp
/users/logiscope/samples/C++/Hangman/Hangman32.cpp
/users/logiscope/samples/C++/Hangman/Hangman.cpp
```

Using the command line:

```
create -audit -lang c++ aProject.ttp -list filelist.lst
```

will create a new Logiscope C++ project file `aProject.ttp` containing 3 files: `GenericDlg.cpp`, `Hangman32.cpp` and `Hangman.cpp` on which *QualityChecker* and *RuleChecker* verification modules will be activated.

### 3.1.2 Makefile mode

When launched from makefiles, **create** is designed to intercept the command line usually passed to the compiler and uses the arguments to build the Logiscope project.

The project makefiles must be modified in order to launch **create** instead of the compiler. In this mode, the name of the project file (“`.ttp`” file) has to be an absolute path, otherwise the process will stop.

When used inside a Makefile, **create** uses the same options as in command line mode, except for:

- root, -recurse, -list : which are not available in this mode
- : which introduces the compiler command.

The following lines can be introduced in a Makefile to build a Logiscope project file :

```
CREATE=create /users/projects/my.ttp -audit -rule -lang c++
CC=$(CREATE) -- gcc
CPP=$(CC) -E
...
```

In this mode, the project file building process is as follows:

1. **create** is invoked for each file by the make utility, instead of the compiler.
2. When **create** is invoked for a file it adds the file to the project, with appropriate preprocessor options if any, then **create** starts the normal compilation command which will ensure that the normal build process will continue.
3. At the end of the make process, the Logiscope project is completed and can be used either using Logiscope **Studio** or with the **batch** tool (see next section).

***Note:** Before executing the makefile, first clean the environment in order to force a full rebuild and to ensure that the **create** will catch all files.*



### 3.1.3 Options

The **create** options are the following:

<code>create -lang cpp</code>	
<code>&lt;ttp_file&gt;</code>	name of a Logiscope project to be created (with the .ttp extension). Path has to be absolute if the option -- is used.
<code>[-root &lt;directory&gt;]</code>	where <directory> is the starting point of the source search. Default is the current directory. This option is exclusive with -list option.
<code>[-recurse]</code>	if present the source file search is done recursively in subfolders.
<code>[-list &lt;list_file&gt;]</code>	where <list_file> is the name of a file containing the list of filenames to add to the project (one file per line). This option is exclusive with -root option.
<code>[-repository &lt;directory&gt;]</code>	where <directory> is the name of the directory where Logiscope internal files will be stored.
<code>[-no_compilation]</code>	avoid compiling the files if the -- option is used
<code>[-]</code>	when used in a makefile, this option introduces the compilation command with its arguments.
<code>[-audit]</code>	to activate the <i>QualityChecker</i> verification module
<code>[-ref &lt;Quality_model&gt;]</code>	where <Quality_model> is the name of the Quality Model file (“ref”) to add to the project. Default is <install_dir>/Ref/Logiscope.ref
<code>[-rule]</code>	to select the RuleChecker verification module
<code>[-rules &lt;rules_file&gt;]</code>	where <rule_file> is the name of the rule set file (.rst) to be included into the project. Default is the RuleChecker.rst file located in the /Ref/RuleSets/<lang>/ will be used.
<code>[-relax]</code>	to activate the violation relaxation mechanism for the project.

<code>[-import &lt;folder_name&gt;]</code>	where <code>&lt;folder_name&gt;</code> is the name of the project folder which will contain the external violation files to be imported. When this option is used the external violation importation mechanism is activated.
<code>[-external &lt;file_name&gt;]*</code>	where <code>&lt;file_name&gt;</code> is the name of a file to be added into the import project folder. This option can be repeated as many times as needed. Only applicable if the <code>-import</code> option is activated.
<code>[-dial &lt;dialect_name&gt;]</code>	where <code>&lt;dialect_name&gt;</code> is one of the available C++ dialects.
<code>[-source &lt;suffixes&gt;]</code>	where <code>&lt;suffixes&gt;</code> is the list of accepted suffixes for the source files. Default is <code>"*.cpp;*.cc;*.cxx"</code> .
<code>[-header &lt;suffixes&gt;]</code>	where <code>&lt;suffixes&gt;</code> is the list of accepted suffixes for header files. Default is <code>"*.h;*.hxx;*.hh"</code>
<code>[-inline &lt;suffixes&gt;]</code>	where <code>&lt;suffixes&gt;</code> is the list of accepted suffixes for inline files . Default is <code>"*.inl"</code> .
<code>[-D&lt;macro_name&gt;]*</code>	same syntax as a preprocessor. When used, this option activates the <code>unifdef</code> tool when parsing the code.
<code>[-U&lt;macro_name&gt;]*</code>	same syntax as a preprocessor. When used, this option activates the <code>unifdef</code> tool when parsing the code.

## 3.2 Logiscope batch

Logiscope **batch** is a tool designed to work with Logiscope in command line to:

- parse the source code files specified in a Logiscope project: i.e. “.ttp” file,
- generate reports in HTML and/or CSV format automatically.

Note that before using **batch**, a Logiscope project shall have been created:

- using Logiscope **Studio**, refer refer to Section 1 or *IBM Rational Logiscope RuleChecker & QualityChecker Getting Started* documentation,
- or using Logiscope **create**, refer to the previous section.

Once the Logiscope project is created, **batch** is ready to use.

### 3.2.1 Options

The **batch** command line options are the following:

batch

<ttp_file>	name of a Logiscope project.
[-tcl <tcl_file>]	name of a <b>Tcl</b> script to be used to generate the reports instead of the default <b>Tcl</b> scripts.
[-o <output_directory>]	directory where the all reports are generated.
[-external <violation_file>]*	name of the file to be added into the import project folder. This option can be repeated as many times as needed. This option is only significant for <i>RuleChecker</i> module for which the external violation importation mechanism is activated
[-nobuild]	generate reports without rebuilding the project. The project must have been built at least once previously.
[-clean]	before starting the build, the Logiscope build mechanism removes all intermediate files and empties the import project folder when the external violation importation mechanism is activated.
[-addin <addin> options]	where <i>addin</i> nis the name of the addin to be activated and <i>options</i> the associated options generating the reports.

<code>[-table]</code>	generate tables in predefined html reports instead of slices or charts. By default, slices or charts are generated (depending on the project type). This option is available only on Windows as on Unix there are no slices or charts, only tables are generated.
<code>[-noframe]</code>	generate reports with no left frame.
<code>[-v]</code>	display the version of the <b>batch</b> tool.
<code>[-h]</code>	display help and options for <b>batch</b> .
<code>[-err &lt;log_err_folder&gt;]</code>	directory where troubleshooting files <b>batch.err</b> and <b>batch.out</b> should be put. By default, messages are directed to standard output and error.

### 3.2.2 Examples of Use

Considering a previously created Logiscope project named **MyProject.ttp** where:

- *RuleChecker* and *QualityChecker* verification modules have been activated,
- the Logiscope Repository is located in the folder **MyProject/Logiscope**,

(Refer to the previous section or to the *RuleChecker & QualityChecker Getting Started* documentation to learn how creating a Logiscope project).

Executing the command on a command line or in a script:

```
batch MyProject.ttp
```

will:

- perform the parsing of all source files specified in the Logiscope project **MyProject.ttp**,
- run the standard TCL script **QualityReport.tcl** located in `<log_install_dir>/Scripts` to generate the standard *QualityChecker* HTML report named **MyProjectquality.html** in the default **MyProject/Logiscope/reports.dir** folder.
- run the standard TCL script **RuleReport.tcl** located in `<log_install_dir>/Scripts` to generate the standard *RuleChecker* HTML report named **MyProjectrule.html** in the default **MyProject/Logiscope/reports.dir** folder.

# Chapter 4

---

## *Standard Metrics*

*Logiscope QualityChecker C++* proposes a set of standard source code metrics. Source code metrics are static measurements (i.e. obtained without executing the program) to be used to assess software attributes (e.g. complexity, self-descriptiveness) or characteristics (e.g. Maintainability, Reliability) of the C++ functions, classes, modules, application under evaluation.

The metrics can be combined to define new metrics more closely adapted to the quality evaluation of the source code. For example, the “comments frequency” metric, well suited to evaluate quality criteria such as self-descriptiveness or analyzability, can be defined by combining two basic metrics: “number of comments” and “number of statements”.

The user can associate threshold values with each of the quality model metrics, indicating minimum and maximum reference values accepted for the metric.

For more details on Source Code Metrics, please refer to:

- *IBM Rational Logiscope - Basic Concepts*.

Source code metrics apply to different domains (e.g. line counting, control flow, data flow, calling relationship) and the range of their scope varies.

The scope of a metric designates the element of the source code the metric will apply to. The following scopes are available for *Logiscope QualityChecker C++*.

- The *Function scope*: the metrics are available for each member and non-member function defined in the source files specified in the Logiscope Project under analysis.
- The *Class scope*: the metrics are available for each C++ class defined in the header and source files specified in the Logiscope Project under analysis. Classes contain member functions and member data.
- The *Module scope*: the metrics are available for each C++ header or source file specified in the Logiscope Project under analysis.
- The *Application scope*: the metrics are available for the set of C++ header and source files specified in the Logiscope Project .

## 4.1 Function Scope

### 4.1.1 Line Counting

For more details on Line Counting Metrics, please refer to:

- *IBM Rational Logiscope - Basic Concepts.*

<b>Ic_cline</b>	<b>Total number of lines</b>
<b>Definition</b>	Total number of lines in the function.
<b>Ic_cloc</b>	<b>Number of lines of code</b>
<b>Definition</b>	Total number of lines containing executable code in the function.
<b>Ic_cblank</b>	<b>Number of empty lines</b>
<b>Definition</b>	Number of lines containing only non printable characters in the function.
<b>Ic_ccomm</b>	<b>Number of lines of comments</b>
<b>Definition</b>	Number of lines of comment in the function.
<b>Ic_ccpp</b>	<b>Number of preprocessor statements</b>
<b>Definition</b>	Number of preprocessor directives (e.g. <code>#include</code> , <code>#define</code> , <code>#ifdef</code> ) in the function.
<b>Ic_csbra</b>	<b>Number of lines with lone braces</b>
<b>Definition</b>	Number of lines containing only a single brace character : i.e. “{“ or “}” in the function.
<b>Ic_pro_c</b>	<b>Number of lines in Pro*C</b>
<b>Definition</b>	Number of lines written in Pro*C in the function.

**lc\_bcom Number of comment blocks.**

**Definition** Number of comment blocks used between a function's header and the closing curly bracket (Blocks of COMments). Several consecutive comments are counted as a single comment block.

**Example**

```

func() ;
{
    /* this is a comment */
    printf ("-----") ;
    /* this is a second */
    /* comment          */
    printf ("-----") ;
    /* this is a third
       comment          */
}
lc_bcom= 3

```

**Alias**

BCOM

**lc\_bcob Number of comment blocks before**

**Definition** 1 if there is a block of comments used just before a function (Blocks of COMments Before). 0 either.

**Example**

```

/* this comment is not counted */
/* as a comment before the function */
int i;
/* this one is counted
   as a comment          */
/* before the function   */
func() ;
{
    printf ("-----") ;
    printf ("-----") ;
}
lc_bcob = 1

```

**Alias**

BCOB

**lc\_parse Number of lines not parsed**

**Definition** Number of lines which cannot be parsed in a function because of syntax errors or of some particular uses of macros.

## 4.1.2 Lexical and syntactic items

### **lc\_algo**      **Number of syntactic entities in algorithms**

**Definition**      Number of syntactic entities inside statements of a function that are not counted as declarations.

### **lc\_decl**      **Number of syntactic entities in declarations**

**Definition**      Number of syntactic entities in the declaration part of a function.

### **lc\_stat** **Number of Statements**

*Also called STMT.*

**Definition**      Number of statements in the function body

Without an optional parameter, following statements are counted:

- Control statements: *break*, statement block, *continue*, *do*, *for*, *goto*, *if*, labels, *return*, *switch*, *while*, *case*, *default*,
- Statements followed by ; ,
- Empty statement.

This metric can be parametrized to count the statements in a familiar way:

- if no parameter is provided, all statements listed above are counted,
- if the parameter "**no\_null\_stat**" is provided, block statements, empty statements and labeled statements (including *case* and *default* labels in *switch* statements) are omitted.

### **lc\_synt**      **Number of syntactic entities**

**Definition**      Number of syntactic entities used in the function.

**Note**              **lc\_synt** is the sum of **lc\_decl** and **lc\_algo**.

## 4.1.3 Data Flow

### **dc\_consts**      **Numbers of declared constants**

**Definition**      Number of constants in a function declared by:

- the *#define* statement,



- variables having a simple type declared as *const*,
- *enum* elements.

### **dc\_types**    **Number of declared types**

**Definition**    Number of types declared in a function with the *typedef*, *struct*, *class* or *enum* statement.

### **dc\_vars**    **Number of declared variables**

**Definition**    Number of variables declared in a function.

### **dc\_lvars**    **Number of local variables**

*Also called LVAR.*

**Definition**    Total number of variables declared in a function (Local VARIables).

### **dc\_clas\_var** **Number of class-type local variables**

*Also called LVARop.*

**Definition**    Number of class type variables which are local to a function. This metric shows a specific type of coupling between classes.

### **dc\_other\_clas\_var** **Number of other class-type local variables**

**Definition**    Number of class type variables which are local to a function, where the class is different from the current class.  
If the function being analyzed is a non-member function, the value is 0.

### **ic\_param**    **Number of parameters**

*Also called PARA.*

**Definition**    Number of parameters of a function.

### **ic\_parvar**    **Variable number of parameters**

**Definition**    Equals 1 if the function has a variable number of parameters, 0 otherwise.

### **ic\_paradd**    **Number of parameters passed by reference**

*Also called PARAadd.*

**Definition** Number of parameters passed by reference of a function. If the function returns a value, then the returned value is considered as a passed by reference parameter.

**ic\_parcl Number of class-type parameters**

*Also called PARAc.*

**Definition** Number of class-type parameters of a function. If the function returns a class-type value then the returned value is considered as a class-type parameter. This metric shows a specific type of coupling between classes.

**ic\_par\_othercl Number of other class-type parameters**

**Definition** Number of class-type parameters of a function, where the class is different from the current class. If the function being analyzed is a non-member function, then the value is 0.

**ic\_parval Number of parameters passed by value**

*Also called PARAval.*

**Definition** Number of parameters passed by value of a function.

**ic\_usedp Number of parameters used**

*Also called U\_PARA.*

**Definition** Number of function parameters used in a function body. A parameter is said to be used wherever it appears in the function code. Combined with the number of function parameters, this metric is a good indicator of the consistency of the function's interface.

**ic\_vare Number of uses of external attributes**

*Also called VARe*

**Definition** Number of uses of attributes defined outside the class. An attribute is said to be "external" if it belongs to another class.  
All attribute occurrences are counted.

**ic\_vari Number of uses of internal attributes**

*Also called VARi.*

**Definition** Number of uses of attributes defined in the class. An attribute is said to be "local" if it belongs to the class of the function being analyzed.  
All attribute occurrences are counted.

### **ic\_varpe Number of distinct uses of external Aattributes**

*Also called VAR\_PATHSe.*

**Definition** Number of distinct times attributes defined outside the class are used. An attribute is said to be "external" if it belongs to another class.  
Different uses of the same attribute count for one.

### **ic\_varpi Number of distinct uses of local attributes**

*Also called VAR\_PATHSi.*

**Definition** Number of times the distinct class attributes are used. An attribute is said to be "local" if it belongs to the class of the function being analyzed.  
Different uses of the same attribute count for one.

## **4.1.4 Halstead Metrics**

For more details on Halstead Metrics, please refer to:

- *IBM Rational Logiscope - Basic Concepts.*

### **n1 Number of distinct operators**

*Also called ha\_dopt.*

**Definition** Number of different operators used in a function.  
This metric can be parametrized to count the operators in a familiar way:

- if no parameter is provided, operators are counted between the beginning of the function's definition and its closing curly bracket,
- if the parameter "**in\_body**" is provided, operators are only counted in the function body (that is between the function's opening and closing curly brackets).

For the use of this parameter, see Chapter Customizing Metrics & Rules.

The following are operators:

- Expressions:
  - Unary operators:

<b>+ -</b>	unary plus or minus
<b>++ --</b>	pre-/post- increment or decrement
<b>!</b>	negation
<b>~</b>	complement of 1 or destructor
<b>*</b>	indirection
<b>&amp;</b>	address
<b>sizeof</b>	sizeof
<b>throw</b>	throw
<b>new</b>	new
<b>::new</b>	global scope new
<b>delete</b>	delete
<b>::delete</b>	global scope delete
<b>delete []</b>	array delete
<b>::delete []</b>	global scope array delete
<b>.</b>	dot
<b>-&gt;</b>	arrow
<b>()</b>	expression in parenthesis

- Binary Operators:

<b>+ - * / %</b>	arithmetic operators
<b>&lt;&lt; &gt;&gt; &amp;   ^</b>	bitwise operators
<b>&gt; &lt; &lt;= &gt;= == !=</b>	comparison operators
<b>&amp;&amp;   </b>	logical operators
<b>-&gt;* .*</b>	pointer to member operators

- Ternary conditional operator: **?:**
- Assignment operators: **= \*= /= %= += -= >>= <<= &= ^= |=**
- Other operators:

<b>(...)</b>	cast	(ex: <b>(float)1</b> )
<b>dynamic_cast</b>	cast	(ex: <b>dynamic_cast&lt;T&gt;(v)</b> )
<b>static_cast</b>	cast	(ex: <b>static_cast&lt;T&gt;(v)</b> )
<b>reinterpret_cast</b>	cast	(ex: <b>reinterpret_cast&lt;T&gt;(v)</b> )
<b>const_cast</b>	cast	(ex: <b>const_cast&lt;T&gt;(v)</b> )
<b>[]</b>	subscripting	(ex: <b>a[i]</b> )

<b>::</b>	(global) scope	(ex: X:: <i>i</i> , :: <i>i</i> )
<b>...()</b>	function call	(ex: func( <i>1</i> ))
<b>(., ., .)</b>	expressions list	(ex: func( <i>1,2,3</i> ))
<b>this</b>		

- Statements:

**IF**                    **ELSE**                    **WHILE()**            **DO WHILE()**  
**RETURN**            **FOR(;;)**                    **SWITCH**            **BREAK**  
**CONTINUE**        **GOTO label**                    **CASE**              **DEFAULT**  
**LABEL**  
**{ }**                    (compound)  
**;**                    (empty statement)

- Declarations:

<b>ASM</b>	(ex: <b>asm</b> ("foo"))
<b>EXTERN</b>	(ex: <b>extern</b> "C" { ... } )
<b>;</b> (empty declaration)	
<b>(member) declaration</b>	(ex: <b>int i; int i = 1;</b> )
<b>type specifier</b>	(ex: <b>int</b> )
<b>storage class</b>	(ex: <b>auto, register, static, extern, mutable</b> )
<b>enumerator specifier</b>	(ex: <b>enum X { ... };</b> )
<b>enumerator-list</b>	(ex: <b>enum X {a, b, c};</b> )
<b>enumerator-definition</b>	(ex: <b>enum X {a=1, b=2};</b> )
<b>typename</b>	(ex: typedef <b>typename X::a b;</b> )
<b>namespace definition</b>	(ex: <b>namespace N { ... } )</b>
<b>using declaration</b>	(ex: <b>using A::x;</b> )
<b>using directive</b>	(ex: <b>using namespace M;</b> )

- Declarators:

	function declarator	(ex: <b>int func();</b> )
<b>[ ]</b>	array declarator	(ex: <b>int tab[5];</b> )
<b>*</b>	pointer declarator	(ex: <b>int *i;</b> )
<b>&amp;</b>	reference declarator	(ex: <b>int&amp; i;</b> )
<b>::*</b>	pointer to member declarator	(ex: <b>int X::* i;</b> )
<b>(., ., .)</b>	parameter-declaration-list	(ex: <b>int func(int i, char *j);</b> )

{..., ..., ..}	initializer-list	(ex: int tab[] = {1, 3, 5};)
	type qualifier	(ex: <b>const</b> , <b>volatile</b> )
	type identifier	(ex: sizeof( <b>int</b> ), new ( <b>int</b> ))

- Classes:

<b>class keys</b>	class struct union
<b>access specifiers</b>	private public protected

- Derived classes:

<b>base classes</b>	(ex : class Z : public X , public Y)
---------------------	--------------------------------------

- Special member functions

:	constructor initializer	(ex: C::C(): A() { ...} try : i(f(ii)), d(id))
.., .., ..	member initializer list	(ex: i(f(ii)), d(id))
	member initializer id	(ex: i(f(ii)))

- Overloading: **operator ...**

- Templates:

<b>template parameters</b>	(ex: template<class K, class V>)
<b>type parameter</b>	(ex: template<class K = int> template<template<class T> class K = int>)
<b>template name</b>	(ex: T1<T2>)
<b>template argument list</b>	(ex: T<T1,T2,T3>)
<b>explicit instantiation</b>	(ex: template A::operator void*());
<b>explicit specialization</b>	(ex: template <> A::operator char*() { return 0; })

- Exceptions:

<b>throw (.., ..)</b>	exception specification (ex: int func() throw(X,Y)
<b>try { ... }</b>	try block
<b>catch (..) { ... }</b>	handler

- Preprocessing directives:

```

#define                #undef
#if                    #ifdef                #ifndef
#elif                 #else                #endif
    
```

<b>#line</b>	<b>#error</b>	<b>#pragma</b>
<b>#</b>	<b>#include</b>	
<b>#define func(.., .., ..)</b>	macro arguments	

## N1 Total number of operators

*Also called ha\_topt.*

**Definition** Total number of operators used in a function.

**Note** The function area where operators are counted depends on the parameter of the **n1** metric (see above).

## n2 Number of distinct operands

*Also called ha\_dopd.*

**Definition** Number of different operands used in a function.

This metric can be parameterized to count the operands in a familiar way:

- if no parameter is provided, operands are counted between the beginning of the function's definition and its closing curly bracket,
- if the parameter "**in\_body**" is provided, operands are only counted in the function's body (that is between the function's opening and closing curly brackets).

For the use of this parameter, see Chapter Customizing Metrics & Rules.

The following are operands:

- Literals:
  - Decimal literals (ex: 45, 45u, 45U, 45l, 45L, 45uL)
  - Octal literals (ex: 0177, 0177u, 0177l)
  - Hexadecimal literals (ex: 0x5f, 0X5f, 0x5fu, 0x5fl)
  - Floating literals (ex: 1.2e-3, 1e+4f, 3.4l)
  - Character literals (ex: 'c', L'c', 'cd', 'a', '\177', '\x5f')
  - String literals (ex: "hello", L" world\n")
  - Boolean literals (true or false)
- Identifiers (variable names, type names, function names, etc.)
- File names in #include clauses (ex: #include <stdlib.h>, #include "foo.h")
- Operator names:

<b>new</b>	<b>delete</b>	<b>new[]</b>	<b>delete[]</b>					
+	-	*	/	%	^	&		~
!	=	<	>	+=	-=	*=	/=	%=

^=	&=	=	<<	>>	>>=	<<=	==	!=
<=	>=	&&		++	--	,	->*	->
()	[]							

## **N2 Total number of operands**

*Also called ha\_topd.*

**Definition** Total number of operands used in a function.

**Note** The function area where operands are counted depends on the parameter of the **n2** metric (see above).

## **n Halstead vocabulary**

**Definition** Halstead vocabulary of the function:  $n = n1 + n2$

**Alias** ha\_voc

## **N Halstead length**

**Definition** Halstead length of the function:  $N = N1 + N2$

**Alias** ha\_olg

## **CN Halstead estimated length**

**Definition** Halstead estimated length of the function:  
 $CN = n1 * \log_2(n1) + n2 * \log_2(n2)$

**Alias** ha\_elg

## **V Halstead volume**

**Definition** Halstead volume of the function:  $V = N * \log_2(n)$

**Alias** ha\_vol

## **L Halstead level**

**Definition** Halstead level of the function:  $L = (2 * n2) / (n1 * N2)$

**Alias** ha\_lev



<b>D</b>	<b>Halstead difficulty</b>
<b>Definition</b>	Halstead difficulty of the function: $D = 1/L$
<b>Alias</b>	ha_dif
<b>I</b>	<b>Halstead intelligent content</b>
<b>Definition</b>	Halstead intelligent content of the function: $I = L * V$
<b>Alias</b>	ha_int
<b>E</b>	<b>Halstead mental effort</b>
<b>Definition</b>	Halstead mental effort of the function: $E = V / L$
<b>Alias</b>	ha_eff

## 4.1.5 Structured Programming

In structured programming:

- a function shall have a single entry point and a single exit point,
- each iterative of selective structures shall have a single exit point: i.e. no `goto`, `break`, `continue` or `return` statement in the structure.

Structured programming improves source code maintainability.

<b>ct_bran</b>	<b>Number of destructuring statements</b>
<b>Definition</b>	Number of destructuring statements in a function ( <code>break</code> and <code>continue</code> in loops, and <code>goto</code> statements).
<b>ct_break</b>	<b>Number of break and continue branchings</b>
<b>Definition</b>	Number of <code>break</code> or <code>continue</code> statements used to exit from loop structures in the function. <code>break</code> statements in <code>switch</code> structures are not counted.
<b>ct_exit</b>	<b>Number of out statements</b>
<b>Definition</b>	Number of nodes associated with an explicit exit from a function ( <code>return</code> , <code>exit</code> ).
<b>Alias</b>	N_OUT

**ct\_goto**      **Number of gotos****Definition**      Number of *goto* statements.**Alias**              GOTO**ESS\_CPX**      **Essentiel complexity****Definition**      Cyclomatic number of the “reduced” control graph of the function.  
The “reduced” control graph is obtained by removing all structured constructs from the control graph of the function.  
A structured construct is a selective or iterative structure that does not contains auxiliary exit statements: *goto*, *break*, *continue* or *return*.**Justification**      When the Essentiel Complexity is equal to 1, the function complies with the structured programming rules.  
Note that the **ct\_exit** and **ct\_bran** metrics already provide such an information on the structuring of the function with more details.

## 4.1.6 Control Flow

For more details on Control Graph Metrics, please refer to:

- *IBM Rational Logiscope - Basic Concepts.*

**ct\_decis**      **Number of decisions****Definition**      Number of selective statements in a function : *if*, *switch***Alias**              N\_STRUCT**ct\_degree**      **Maximum degree****Definition**      Maximum number of edges departing from a node.**ct\_edge**        **Number of edges****Definition**      Number of edges of the control graph of a function.**ct\_nest**        **Maximum nesting level****Definition**      Maximum nesting level of control structures in a function.**ct\_node**        **Number of nodes****Definition**      Number of nodes of the control graph of a function.

<b>ct_loop</b>	<b>Number of loops</b>
<b>Definition</b>	Number of iterative statements in a function (pre- and post- tested loops): for, while, do while,
<b>ct_path</b>	<b>Number of paths</b>
<b>Definition</b>	Number of non-cyclic execution paths of the control graph of the function.
<b>Alias</b>	PATH
<b>ct_raise</b>	<b>Number of exception raises</b>
<b>Definition</b>	Number of occurrences of the <i>throw</i> clause within a function body.
<b>Alias</b>	N_RAISE
<b>ct_try</b>	<b>Number of exceptions handlers</b>
<b>Definition</b>	Number of <i>try</i> blocks in a function.
<b>Alias</b>	N_EXCEPT
<b>ct_vg</b>	<b>Cyclomatic number (VG)</b>
<b>Definition</b>	Cyclomatic number of the control graph of the function.
<b>Alias</b>	VG, ct_cyclo
<b>DES_CPX</b>	<b>Design complexity</b>
<b>Definition</b>	Cyclomatic number of the “design” control graph of the function. The “design” control graph is obtained by removing all constructs that do not contain calls from the control graph of the function.

## 4.1.7 Relative Call Graph

For more details on Call Graph Metrics, please refer to:

- *IBM Rational Logiscope - Basic Concepts.*

### **cg\_entropy** Relative call graph entropy

<b>Definition</b>	SCHUTT entropy of the relative call graph of the function.
<b>Alias</b>	ENTROPY

### **cg\_hiercpx** Relative call graph hierarchical complexity

<b>Definition</b>	Average number of components per level( i.e. number of components divided by number of levels) of the relative call graph of the function..
-------------------	---

<b>Alias</b>	HIER_CPX
<b>cg_levels</b>	<b>Relative call graph levels</b>
<b>Definition</b>	Depth of the relative call graph of the function..
<b>Alias</b>	LEVELS
<b>cg_strucpx</b>	<b>Relative call graph structural complexity</b>
<b>Definition</b>	Average number of calls per component: i.e. number of calling relations between components divided by the number of components) of the relative call graph of the function..
<b>Alias</b>	STRU_CPX
<b>cg_testab</b>	<b>Relative call graph testability</b>
<b>Definition</b>	Mohanty system testability of the relative call graph of the function.
<b>Alias</b>	TESTBTY
<b>dc_calls</b>	<b>Number of direct calls</b>
<b>Definition</b>	Number of direct calls in a function. Different calls to the same function count for one call.
<b>Alias</b>	DRCT_CALLS
<b>dc_calle</b>	<b>Number of external calls</b>
<b>Definition</b>	Number of Calls to Functions Defined outside the Class. A function is said to be "defined outside" the class if the function does not belong to the same class as the function being analyzed. If the function being analyzed is a non-member function, then all functions called by the function being analyzed are considered as "defined outside" the class. All call occurrences are counted.
<b>Alias</b>	CALLe
<b>dc_calli</b>	<b>Number of internal calls</b>
<b>Definition</b>	Number of Calls to Functions Defined in the Class. A function is said to be "defined in" the class if the function belongs to the same class as the function being analyzed. If the function being analyzed is a non-member function, then there is no function "defined in" the class (the value is 0). All call occurrences are counted.

<b>Alias</b>	CALLi
<b>dc_calling</b>	<b>Number of callers</b>
<b>Definition</b>	Number of functions calling the designated function.
<b>Alias</b>	NBCALLING
<b>dc_callpe</b>	<b>Number of external direct calls</b>
<b>Definition</b>	Number of distinct calls to functions defined outside the class of the function being analyzed (see <b>dc_calle</b> above). Different calls to the same function count for one call.
<b>Alias</b>	CALL_PATHSe
<b>dc_callpi</b>	<b>Number of internal direct calls</b>
<b>Definition</b>	Number of distinct calls to functions defined in the class of the function being analyzed (see <b>dc_calli</b> above). Different calls to the same function count for one call.
<b>Alias</b>	CALL_PATHSi
<b>dc_stat_call</b>	<b>Number of calls to static members</b>
<b>Definition</b>	Number of calls to static member functions in a function.
<b>IND_CALLS</b>	<b>Relative call graph call-paths</b>
<b>Definition</b>	Number of call paths in the relative call graph of the function.

## 4.2 Class Scope

### 4.2.1 Comments

#### **cl\_bcob**      **Number of comment blocks before**

*Also called BCOBc.*

**Definition**      Number of blocks of comments located between a class header and the curly bracket of the previous class or between a class header and the beginning of the file.

#### **cl\_bcom**      **Number of comment blocks**

*Also called BCOMc.*

**Definition**      Number of comment blocks in a class. Consecutive comments are counted as belonging to the same block. Comments located outside the class are not counted.

### 4.2.2 Data Flow

#### **cl\_base\_priv** **Number of private base classes**

**Definition**      Number of declared classes from which a class inherits, whose names appear after the `private` keyword.

#### **cl\_base\_prot** **Number of protected base classes**

**Definition**      Number of declared classes from which a class inherits, whose names appear after the `protected` keyword.

#### **cl\_base\_publ** **Number of public base classes**

**Definition**      Number of declared classes from which a class inherits, whose names appear after the `public` keyword.

**cl\_base\_virt Number of virtual base classes**

**Definition** Number of declared classes from which a class inherits, whose names appear after the `virtual` keyword.

**cl\_clas\_frnd Number of friend classes**

**Definition** Number of classes declared in a class definition, whose names appear after the `friend` keyword.

**cl\_cobc Coupling between classes**

*Also called COBC, cl\_dep\_deg*

**Definition** Coupling between classes is the sum of:

- the number of inherited classes (see in **in\_data\_class** Number of Direct Base Classes),
- the number of class type attributes for the class (see **cl\_data\_class** below),
- two times the number of calls to static member functions for class methods (see in **dc\_stat\_call** Number of Calls to Static Member Functions).
- two times the number of class-type parameters for the class methods,
- three times the number of class-type local variables for the class methods (see in **dc\_clas\_var** Number of Class Type Local Variables).

$$\text{cl\_cobc} = \text{in\_dbases} + \text{cl\_data\_class} + \sum_{\text{methods}} (2 \times \text{dc\_stat\_call} + 2 \times \text{ic\_parcl} + 3 \times \text{dc\_clas\_var})$$

**cl\_data\_class Sum of class-type attributes**

**Definition** Number of class-type attributes for the class.

**Alias** LACT

**cl\_data\_priv Number of private attributes**

**Definition** Number of data members declared in the `private` section of a class.

**Alias** LAPI, cl\_field\_priv

**cl\_data\_prot Number of protected attributes**

**Definition** Number of data members declared in the `protected` section of a class.

**Alias** LAPO, cl\_field\_prot

### **cl\_data\_publ Number of public attributes**

**Definition** Number of data members declared in the `public` section of a class.

**Alias** LAPU, cl\_field\_publ

### **cl\_data\_stat Number of static data members**

**Definition** Number of data members declared after the `static` keyword in a class.

### **cl\_data\_inh Number of inherited attributes**

**Definition** Number of public or protected attributes in the base classes of a class, which are not overridden in that class.

### **cl\_dep\_meth Number of dependent methods**

**Definition** Number of methods within the class depending on other classes. A method is said to be dependent if:

- it calls a non-member function or other class methods (see in **dc\_calle** Number of Calls to functions Defined outside the Class),
- it uses an attribute which belongs to a different class (see in **ic\_vare** Number of Times External Attributes are used),
- it has a class instance parameter which belongs to a different class (see in **ic\_par\_othercl** Number of Other Class Type Parameters),
- it declares a class instance variable which belongs to a different class (see in **dc\_other\_clas\_var** Number of other Class Type Local Variables).

$$cl\_dep\_meth = \sum_{\text{methods}} \begin{cases} 1 & \Leftrightarrow dc\_calle + ic\_vare + ic\_par\_othercl + dc\_other\_clas\_var > 0 \\ 0 & \text{otherwise} \end{cases}$$

**Alias** NMD

### **cl\_rfc Response for a class**

**Definition** Number of methods that can be invoked in response to a message to an object of the class or by some method in the class.  
This includes all methods accessible within the class hierarchy.

### **cl\_type Number of local types**

**Definition** Number of types declared in a class.



**cl\_const Number of local constants**

**Definition** Number of constants declared in a class. Constants are data members declared with the keyword `const`, like `const type name ...`, or `type * const name ...` (constant pointer), or `type C::* const name` (constant pointer to member) for instance (but not pointers to constant).

**cl\_genp Number of of parameters for templates**

**Definition** Number of parameters declared in a class for classes that are templates. If `cl_genp` has the value 0 the class is not a template.

**cl\_oper\_conv Number of conversion operators**

**Definition** Number of conversion operators declared in a class declaration.

**cl\_oper\_std Number of standard operators**

**Definition** Number of operators declared in a class, whose names belong to a certain list being a parameter of the metric (by default, this list is empty).

**cl\_oper\_affc Number of assignment operators**

**Definition** Number of operators declared in a class, whose names belong to a certain list which is a parameter of the metric (by default, this list contains "=", "+=", "-=", "\*=", "/=", "%=", "^=", "&=", "|=", "<<=", ">>=", "+", "-", "\*", "/" and "[]").

**cl\_oper\_spec Number of special operators**

**Definition** Number of operators declared in a class, whose names belong to a certain list which is a parameter of the metric (by default, this list contains "->", "()", ",", "->\*", "new", "delete", "new[]", and "delete[]").

## 4.2.3 Statistical Aggregates of Function Metrics

**cl\_func\_priv Number of private methods**

**Definition** Number of methods declared in the `private` section of a class.

**Alias** LMPL, `cl_meth_priv`

**cl\_func\_prot Number of protected methods**

**Definition** Number of methods declared in the `protected` section of a class.

**Alias** LMPO, `cl_meth_prot`

### **cl\_func\_publ Number of public methods**

**Definition** Number of methods declared in the `public` section of a class.

**Alias** LMPU, cl\_meth\_publ

### **cl\_func\_virt Number of virtual methods**

**Definition** Number of methods declared after the `virtual` keyword in a class.

### **cl\_func\_pure Number of abstract methods**

**Definition** Number of methods declared after the `virtual` keyword and followed by `=0` in a class.

**Alias** LMABS

### **cl\_func\_cons Number of constant methods**

**Definition** Number of methods declared after the `const` keyword in a class.

### **cl\_func\_inln Number of inline methods**

**Definition** Number of methods declared after the `inline` keyword in a class.

### **cl\_func\_excp Number of methods handling or raising exceptions**

**Definition** Number of methods declared in a class declaration in which:

- the body of the function is a `try` block, or
- the function body contains a `try` block, or
- exceptions are specified using the `throw` keyword.

### **cl\_func\_frnd Number of friend functions**

**Definition** Number of methods declared after the `friend` keyword in a class.

### **cl\_func\_inh Number of inherited methods**

**Definition** Number of public or protected methods in the base classes of a class, which are not overridden in that class.

### **cl\_func\_over Number of overridden methods**

**Definition** Number of inherited methods which a class overrides.

**Justification** High values for `cl_func_over` tend to indicate design problems. Sub-classes should generally add to and extend the functionality of the parent classes rather than overriding them.

**Alias** LMRE

**cl\_data\_vare Sum of uses of external attributes**

<b>Definition</b>	Total number of times attributes defined in other classes (ic_varpe) are used by the class methods.
<b>Alias</b>	LMVAR_PATHSe

**cl\_data\_vari Sum of uses of internal attributes**

<b>Definition</b>	Total number of times the class's attributes (ic_varpi) are used by the class methods.
<b>Alias</b>	LMVAR_PATHSi

**cl\_fpriv\_path Sum of paths of private methods**

<b>Definition</b>	Sum of non-cyclic execution paths (cl_path) of the private methods of the class.
<b>Alias</b>	LMPIPATH

**cl\_fprot\_path Sum of paths of protected methods**

<b>Definition</b>	Sum of non-cyclic execution paths (cl_path) of the protected methods of the class.
<b>Alias</b>	LMPOPATH

**cl\_fpubl\_path Sum of paths of public methods**

<b>Definition</b>	Sum of non-cyclic execution paths (cl_path) of the public methods of the class.
<b>Alias</b>	LMPUPATH

**cl\_func\_calle Sum of external calls**

<b>Definition</b>	Total number of calls from the class methods to non-member functions or member functions of other classes (dc_callpi).
<b>Alias</b>	LMCALL_PATHSe

**cl\_func\_calli Sum of internal calls**

<b>Definition</b>	Total number of calls from class methods to member functions of the same class (dc_callpi).
<b>Alias</b>	LMCALL_PATHSi

**cl\_usedp Sum of parameters**

<b>Definition</b>	Total number of parameters (ic_usedp) used in the class methods.
<b>Alias</b>	LMU_PARA

The two following metrics have been introduced by Shyam R. Chidamber and Chris F. Kemerer in "*A Metrics Suite for Object Oriented Design*" (IEEE Transactions on Software Engineering, vol 20, pp. 476-493, June 1994).

### **cl\_wmc      Weighted Methods per Class**

**Definition**      Sum of static complexities of class methods.  
Static complexity is represented in this calculation by the cyclomatic numbers (VG).

**Alias**              LMVG, cl\_cyclo

### **cl\_locm      Lack of cohesion of methods**

**Definition**      Percentage of methods that do not access a specific attribute of a class averaged over all attributes in that class.

$$cl\_locm = \frac{\sum_{i=1}^{TA} (1 - Ac(A_i))}{TA}$$

where:

$$Ac(A_i) = \frac{\sum_{j=1}^{TM} is\_accessed(A_i, M_j)}{TM}$$

and:

$$is\_accessed(A_i, M_j) = \begin{cases} 1 & \text{if } M_j \text{ accesses } A_i \\ 0 & \text{otherwise} \end{cases}$$

## 4.2.4 Inheritance Tree

### **in\_bases      Number of base classes**

**Definition**      Number of classes from which a class inherits directly or not  
If multiple inheritance is not used, the value of **in\_bases** is equal to the value of **in\_depth**.

**Alias**              in\_inherits

### **in\_dbases      Number of direct base classes**

**Definition**      Number of classes from which a class directly inherits.

<b>Note</b>	A value of <b>in_dbases</b> upper than 1 denotes multiple inheritance.
<b>Alias</b>	MII, in_dinherits
<b>in_depth</b>	<b>Depth of the inheritance tree</b>
<b>Definition</b>	Maximum length of an inheritance chain starting from a class.
<b>in_derived</b>	<b>Number of derived classes</b>
<b>Definition</b>	Total number of classes which inherit from a class directly or indirectly.
<b>in_noc</b>	<b>Number of children</b>
<b>Definition</b>	Number of classes which inherit directly from a class.
<b>Alias</b>	NOC, in_dderived
<b>in_reinh</b>	<b>Number of classes inherited several times</b>
<b>Definition</b>	Number of classes which directly inherit from a class.

## 4.2.5 Use Graph

<b>cu_level</b>	<b>Depth of use</b>
<b>Definition</b>	Maximum length of a chain of use starting from a class (not counting use loop).
<b>cu_cdused</b>	<b>Number of direct used classes</b>
<b>Definition</b>	Number of classes used directly by a class.
<b>cu_cused</b>	<b>Number of used classes</b>
<b>Definition</b>	Number of classes used by the current class directly or not.
<b>cu_cdusers</b>	<b>Number of direct user classes</b>
<b>Definition</b>	Number of classes which use directly a class.
<b>cu_cusers</b>	<b>Number of user classes</b>
<b>Definition</b>	Total number of classes which use directly or not a class.

## 4.3 Module Scope

### 4.3.1 Line Counting

For more details on Line Counting Metrics, please refer to:

- *IBM Rational Logiscope - Basic Concepts.*

#### **md\_blank      Number of empty lines**

**Definition**      Number of lines containing only non printable characters in the module.

#### **md\_comm      Number of lines of comments**

**Definition**      Number of lines of comments in the module.

**Alias**              LCOM

#### **md\_cpp        Number of preprocessor statements**

**Definition**      Number of preprocessor directives (e.g. *#include*, *#define*, *#ifdef*) in the module.

#### **md\_line      Total number of lines**

**Definition**      Total number of lines in the module.

#### **md\_loc        Number of lines of code**

**Definition**      Total number of lines containing executable code in the module.

#### **md\_sbraz     Number of lines with lone braces**

**Definition**      Number of lines containing only a single brace character : i.e. “{“ or “}” in the module.

#### **md\_pro\_c     Number of lines in Pro\*C**

**Definition**      Total number of lines of PRO\*C in the module.

## 4.3.2 Lexical and syntactic items

<b>md_algo</b>	<b>Number of syntactic entities in algorithms</b>
<b>Definition</b>	Number of syntactic entities inside statements that are not counted as declaration in the file.
<b>md_decl</b>	<b>Number of syntactic entities in declarations</b>
<b>Definition</b>	Number of syntactic entities in the declaration part of the module (function headers and declaration).
<b>md_synt</b>	<b>Number of syntactic entities</b>
<b>Definition</b>	Total number of syntactic entities in the module.
<b>md_stat</b>	<b>Number of statements</b>
<b>Definition</b>	Total number of executable statements in the functions defined in the module.

## 4.3.3 Data Flow

<b>md_consts</b>	<b>Number of declared constants</b>
<b>Definition</b>	Number of constants declared in the module.
<b>md_expfn</b>	<b>Number of exported functions</b>
<b>Definition</b>	Number of non-static global functions defined in the module.
<b>md_expva</b>	<b>Number of exported variables</b>
<b>Definition</b>	Number of non-static global variables defined in the module.
<b>md_impmo</b>	<b>Number of imported modules</b>
<b>Definition</b>	Number of modules included inside a module.
<b>md_types</b>	<b>Number of declared types</b>
<b>Definition</b>	Number of types declared in the module.
<b>md_vars</b>	<b>Number of declared variables</b>
<b>Definition</b>	Number of variables declared in the module.

## 4.3.4 Halstead Metrics

For more details on Halstead Metrics, please refer to:

- *IBM Rational Logiscope - Basic Concepts.*

### **md\_n1      Number of distinct operators**

**Definition**      Number of distinct operators referenced in the module.  
See metric n1 in Function Scope section for the specification of operators.

### **md\_n2      Number of distinct operands**

**Definition**      Number of distinct operands referenced in the module.  
See metric n2 in Function Scope section for the specification of operands.

### **md\_N1      Total number of operators**

**Definition**      Total number of operators referenced in the module.

### **md\_N2      Total number of operands**

**Definition**      Total number of operands referenced in the module.

### **md\_n      Halstead vocabulary**

**Definition**      Halstead vocabulary of the module.  
 $n = n1 + n2$

### **md\_N      Halstead length**

**Definition**      Halstead observed length of the module.  
 $N = N1 + N2$

### **md\_CN      Halstead estimated length**

**Definition**      Halstead estimated length of the module.  
 $\hat{N} = n1 * \log_2(n1) + n2 * \log_2(n2).$

### **md\_V      Halstead volume**

**Definition**      Halstead Program Volume  
 $V = N * \log_2(n)$



<b>md_L</b>	<b>Halstead level</b>
<b>Definition</b>	Halstead Program Level $L = (2 * n2) / (n1 * N2)$
<b>md_D</b>	<b>Halstead difficulty</b>
<b>Definition</b>	Halstead Program Difficulty $D = 1/L$
<b>md_I</b>	<b>Halstead intelligent content</b>
<b>Definition</b>	Halstead Intelligent Content $I = L * V$
<b>md_E</b>	<b>Halstead mental effort</b>
<b>Definition</b>	Halstead Intelligent Content $E = V / L$

## 4.4 Application Scope

Metrics presented in this section are based on the set of C++ header and source files specified in Logiscope Project under analysis. It is therefore recommended to use these metrics values exclusively for a complete application or for a coherent subsystem.

### 4.4.1 Line Counting

For more details on Line Counting Metrics, please refer to:

- *IBM Rational Logiscope - Basic Concepts.*

#### **ap\_sline**      **Total number of lines**

**Definition**      Total number of lines in the application source files.

#### **ap\_sloc**      **Number of lines of code**

**Definition**      Total number of lines containing executable in the application source files.

#### **ap\_sblank**      **Number of empty lines**

**Definition**      Total number of lines containing only non printable characters in the application source files.

#### **ap\_scomm**      **Number of lines of comments**

**Definition**      Total number of lines of comments in the application source files.

#### **ap\_scpp**      **Number of preprocessor statements**

**Definition**      Number of preprocessor directives (e.g. *#include*, *#define*, *#ifdef*) in the application source files.

#### **ap\_ssbra**      **Number of “brace” lines**

**Definition**      Number of lines containing only a single brace character (“{” or “}”) in the application source files.

## 4.4.2 Application Aggregates

### **ap\_clas**      **Number of application classes**

**Definition**      Number of classes defined in the header and source files.

**Alias**              LCA

### **ap\_func**      **Number of application functions**

**Definition**      Number of member and non-member functions defined in the header and source files.

**Alias**              LMA

### **ap\_stat**      **Number of statements**

**Definition**      Number of executable statements (i.e. lc\_stat) of all the functions defined in the application functions.

### **ap\_cbo**      **Coupling between objects**

**Definition**      Sum of the relationships from class to class other than inheritance relationships.

$$ap\_cbo = \sum_{classes} (cl\_func\_calls + cl\_data\_class)$$

**Alias**              CBO

### **ap\_mdf**      **Number of defined methods**

**Definition**      Number of defined member functions in the application.

**Alias**              MDF

### **ap\_nmm**      **Number of member functions**

**Definition**      Number of member functions in the application.

**Alias**              NMM

### **ap\_npm**      **Number of public methods**

**Definition**      Number of public methods in the application.

**Alias**              NPM

**ap\_vg Sum of cyclomatic numbers**

**Definition** Sum of cyclomatic numbers (i.e. `ct_vg`) for all the functions defined in the application.

**Alias** VGA, `ap_cyclo`

### 4.4.3 Application Call Graph

For more details on Call Graph Metrics, please refer to:

- *IBM Rational Logiscope - Basic Concepts.*

**ap\_cg\_cycle Call graph recursions**

**Definition** Number of recursive paths in the call graph for the application's functions. A recursive path can be for one or more functions.

**Alias** GA\_CYCLE

**ap\_cg\_edge Call graph edges**

**Definition** Number of edges in the call graph of application functions.

**Alias** GA\_EDGE

**ap\_cg\_leaf Call graph leaves**

**Definition** Number of functions executing no call.  
In other words, number of leaves nodes in the application call graph.

**Alias** GA\_NSS

**ap\_cg\_level Call graph depth**

**Definition** Depth of the Call Graph: number of call graph levels.

**Alias** GA\_LEVEL

**ap\_cg\_maxdeg Maximum callers/called**

**Definition** Maximum number of calling/called for nodes in the call graph of application functions.

**Alias** GA\_MAXDEG

**ap\_cg\_maxin Maximum callers**

**Definition** Maximum number of "callings" for nodes in the call graph of Application functions.

**Alias** GA\_MAX\_IN

**ap\_cg\_maxout Maximum called**

<b>Definition</b>	Maximum number of called functions for nodes in the call graph of Application functions.
<b>Alias</b>	GA_MAX_OUT

**ap\_cg\_node Call graph nodes**

<b>Definition</b>	Number of nodes in the call graph of Application functions. This metric cumulates Application's member and non-member functions as well as called but not analyzed functions.
<b>Alias</b>	GA_NODE

**ap\_cg\_root Call graph roots**

<b>Definition</b>	Number of roots functions in the application call graph.
<b>Alias</b>	GA_NSP

## 4.4.4 Inheritance Tree

**ap\_inhg\_cpx Inheritance tree complexity**

<b>Definition</b>	The complexity of the inheritance tree is defined as a ratio between: <ul style="list-style-type: none"> <li>• the sum for all of the graph levels of the number of nodes on the level times the level weight index,</li> <li>• the number of graph nodes.</li> <li>• Basic classes are on the top level and leaf classes on the lower levels</li> </ul>
<b>Alias</b>	GH_CPX

**ap\_inhg\_edge Inheritance graph edges**

<b>Definition</b>	Number of inheritance relationships in the application.
<b>Alias</b>	GH_EDGE

**ap\_inhg\_leaf Number of final class**

<b>Definition</b>	Number of final classes in the inheritance tree of the application. A class is said to be a final class if it has no child class.
<b>Alias</b>	GH_NSP

**ap\_inhg\_level Depth of inheritance tree**

<b>Definition</b>	The Depth of the Inheritance Tree (DIT) is the number of classes in the longest inheritance link.
-------------------	---

**Alias** GH\_LEVEL

**ap\_inhg\_maxdeg Maximum Number of derived/inherited classes**

**Definition** Maximum number of inheritance relationships for a given class. This metric applies to the Application's inheritance graph.

**Alias** GH\_MAX\_DEG

**ap\_inhg\_maxin Maximum Number of derived classes.**

**Definition** Maximum number of derived classes for a given class in the inheritance graph.

**Alias** GH\_MAX\_IN

**ap\_inhg\_maxout Maximum Number of inherited classes.**

**Definition** Maximum number of inherited classes for a given class in the inheritance graph.

**Alias** GH\_MAX\_OUT

**ap\_inhg\_node Inheritance tree classes**

**Definition** Number of classes present in the inheritance tree of the application.

**Alias** GH\_NODE

**ap\_inhg\_pc Protocol complexity**

**Definition** Depth of the Inheritance Tree times the maximum number of functions in a class of the inheritance tree over the total number of functions in the inheritance tree

$$ap\_inhg\_pc = ap\_inhg\_levl \times \frac{MAX (LMPI + LMPO + LMPU)}{SUM (LMPI + LMPO + LMPU)}$$

**Alias** GH\_PC

**ap\_inhg\_root Number of basic classes**

**Definition** Number of basic classes in the application. A class is said to be basic if it does not inherit from any other class.

**Alias** GH\_NSS

## ap\_inhg\_uri Number of repeated inheritances

<b>Definition</b>	Repeated inheritances consist in inheriting twice from the same class. The number of repeated inheritances is the number of inherited class couples leading to a repeated inheritance.
<b>Alias</b>	GH_URI

### 4.4.5 MOOD Metrics

The MOOD (Metrics for Object Oriented Design) set of metrics described in this chapter has been introduced by Fernando Brito e Abreu in "*Object-Oriented Software Engineering: Measuring and Controlling the Development Process*" (Proceedings of the 4th International Conference on Software Quality, ASQC, McLean, VA, USA, October 1994).

Their definitions have been refined since their first introduction. The MOOD metrics computed by *Logiscope C++ QualityChecker* conform to the latest definitions and the corresponding C++ bindings described in "*Evaluating the Impact of Object-Oriented Design on Software Quality*" (Proceedings of the Third International Software Metrics Symposium, IEEE, Berlin, Germany, March 1996).

## ap\_mhf Method Hiding Factor (MHF)

### Definition

$$ap\_mhf = \frac{\sum_{i=1}^{TC} \left[ \sum_{m=1}^{M_d(C_i)} (1 - V(M_{mi})) \right]}{\sum_{i=1}^{TC} M_d(C_i)}$$

where:

$$V(M_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(M_{mi}, C_j)}{TC - 1}$$

and:

$$is\_visible(M_{mi}, C_j) = \begin{cases} 1 & \Leftrightarrow \begin{cases} j \neq i \\ C_j \text{ may call } M_{mi} \end{cases} \\ 0 & \text{otherwise} \end{cases}$$

The MHF numerator is the sum of the invisibilities of all methods defined in all classes. The invisibility of a method is the percentage of the total classes from which this method is not visible.

The MHF denominator is the total number of methods defined in the project.

The following C++ bindings are used to compute this metric:

<b>MOOD</b>		<b>C++</b>
TC	total classes	total number of classes
	methods	constructors; destructors; function members; operator definitions
$M_d(C_i)$	methods defined (not inherited)	all methods declared in the class including virtual (deferred) ones
$V(M_{mi})$	visibility - percentage of the total classes from which the method $M_{mi}$ is visible	= 1 for methods in public clauses; = 0 for methods in private clauses; = $DC(C_i)/(TC-1)$ for methods in protected clauses ( $DC(C_i)$ = descendants of $C_i$ )



**ap\_ahf      Attribute Hiding Factor (AHF)**

**Definition**

$$ap\_ahf = \frac{\sum_{i=1}^{TC} \left[ \sum_{m=1}^{A_d(C_i)} (1 - V(A_{mi})) \right]}{\sum_{i=1}^{TC} A_d(C_i)}$$

where:

$$V(A_{mi}) = \frac{\sum_{j=1}^{TC} is\_visible(A_{mi}, C_j)}{TC - 1}$$

and:

$$is\_visible(A_{mi}, C_j) = \begin{cases} 1 & \Leftrightarrow \begin{cases} j \neq i \\ C_j \text{ may reference } A_{mi} \end{cases} \\ 0 & \text{otherwise} \end{cases}$$

The AHF numerator is the sum of the invisibilities of all attributes defined in all classes. The invisibility of an attribute is the percentage of the total classes from which this attribute is not visible.

The AHF denominator is the total number of attributes defined in the project.

The following C++ bindings are used to compute this metric:

<b>MOOD</b>	<b>C++</b>
$A_d(C_i)$	attributes defined (not inherited)      data members
$V(A_{mi})$	visibility - percentage of the total classes from which the attribute $A_{mi}$ is visible      = 1 for attributes in public clauses; = 0 for attributes in private clauses; = $DC(C_i)/(TC-1)$ for attributes in protected clauses ( $DC(C_i)$ = descendants of $C_i$ )

**ap\_mif      Method Inheritance Factor (MIF)****Definition**

$$ap\_mif = \frac{\sum_{i=1}^{TC} M_i(C_i)}{\sum_{i=1}^{TC} M_a(C_i)}$$

where:

$$M_a(C_i) = M_d(C_i) + M_i(C_i)$$

The MIF numerator is the sum of inherited methods in all classes of the project.

The MIF denominator is the total number of available methods (locally defined plus inherited) for all classes.

The following C++ bindings are used to compute this metric:

<b>MOOD</b>		<b>C++</b>
$M_a(C_i)$	available methods	function members that can be invoked in association with $C_i$
$M_d(C_i)$	methods defined	function members declared within $C_i$
$M_i(C_i)$	inherited methods	function members inherited (and not overridden) in $C_i$

**ap\_aif      Attribute Inheritance Factor (AIF)****Definition**

$$ap\_aif = \frac{\sum_{i=1}^{TC} A_i(C_i)}{\sum_{i=1}^{TC} A_a(C_i)}$$

where:

$$A_a(C_i) = A_d(C_i) + A_i(C_i)$$

The AIF numerator is the sum of inherited attributes in all classes of the project.

The AIF denominator is the total number of available attributes (locally defined plus inherited) for all classes.

The following C++ bindings are used to compute this metric:

<b>MOOD</b>		<b>C++</b>
$A_a(C_i)$	available attributes	data members that can be invoked associated with $C_i$
$A_d(C_i)$	attributes defined	data members declared within $C_i$
$A_i(C_i)$	inherited attributes	data members inherited (and not overridden) in $C_i$

## ap\_pof Polymorphism Factor (POF)

### Definition

$$ap\_pof = \frac{\sum_{i=1}^{TC} M_d(C_i)}{\sum_{i=1}^{TC} [M_n(C_i) \times DC(C_i)]}$$

where:

$$M_d(C_i) = M_n(C_i) + M_o(C_i)$$

The POF numerator is the sum of overriding methods in all classes. This is the *actual number of possible different polymorphic situations*. Indeed, a given message sent to a class can be bound, statically or dynamically, to a named method implementation. The latter can have as many shapes (morphos) as the number of times this same method is overridden (in that class's descendants).

The POF denominator represents the *maximum number of possible distinct polymorphic situations* for that class as the sum for each class of the number of new methods multiplied by the number of descendants. This value would be maximum if all new methods defined in each class would be overridden in all of their derived classes.

The following C++ bindings are used to compute this metric:

<b>MOOD</b>		<b>C++</b>
DC(C <sub>i</sub> )	descendants count	number of classes descending from C <sub>i</sub>
M <sub>n</sub> (C <sub>i</sub> )	new methods	function members declared within C <sub>i</sub> that do not override inherited ones
M <sub>o</sub> (C <sub>i</sub> )	overriding methods	function members declared within C <sub>i</sub> that override (redefine) inherited ones

## ap\_cof      Coupling Factor (COF)

### Definition

$$ap\_cof = \frac{\sum_{i=1}^{TC} \left[ \sum_{j=1}^{TC} is\_client(C_i, C_j) \right]}{TC^2 - TC}$$

where:

$$is\_client(C_i, C_j) = \begin{cases} 1 & \Leftrightarrow C_i \Rightarrow C_j \wedge C_i \neq C_j \\ 0 & \text{otherwise} \end{cases}$$

The COF denominator stands for the *maximum possible number of couplings* in a system with TC classes.

The client-supplier relation (represented by  $C_c \Rightarrow C_s$ ) means that  $C_c$  (*client* class) contains *at least one* non-inheritance reference to a feature (method or attribute) of class  $C_s$  (*supplier* class). The COF numerator then represents the *actual number of couplings not imputable to inheritance*.

Client-supplier relations can have several shapes:

Client-supplier shapes	C++
regular message passing	call to the interface of a function member in another class
"forced" message passing	call to a visible or hidden function member in another class by means of a friend clause
object allocation and deallocation	call to a class constructor or destructor
semantic associations among classes with a certain arity (e.g. 1:1, 1:n or n:m)	reference to a supplier class as a data member or as a formal parameter in a function member interface



# Chapter 5

---

## *Programming Rules*

This chapter describes the default programming rules that can be checked using Logiscope C++ *RuleChecker*.

They are made available through various standard Rule Sets provided with the Logiscope distribution. See §5.1.

Rule Sets and rules can be tailored to better fit to the effective coding standards and quality requirements applicable to the project or organisation: e.g. naming rules. For more details, see Chapter Customizing Metrics & Rules.

### 5.1 Rule Sets

Logiscope C++ *RuleChecker* comes with several standard Rule Sets

They are made available through various Rules Set

The rules specified in section 5.2 can be customized by modifying parameters in the project Rule Set file (see Chapter Customizing Metrics & Rules).

The **Motor Industry Software Reliability Association** has published a list of rules for the use of the C++ programming language for critical systems :

- *MISRA-C++:2008 Guidelines for the use of the C++ language critical systems* - June 2008 [MISRA-C++:2008].

*Logiscope RuleChecker* C++ provides the MISRAC++2008 Rule Set including some of the key MISRA C++ programming rules:

- Please refer to [MISRA-C++:2008] for a complete specification of the rules.

## 5.2 Rule Scripts

### 5.2.1 Basic Rules

#### asscal Assignment inside function calls

**Description** Assignment operators (=, +=, -=, \*=, /=, %=, >>=, <<=, &=, |=, ^=, ++, --) shall not be used inside function calls.

**Justification** Removes ambiguity about the evaluation order.

#### asscon Assignment inside conditions

**Description** Assignment operators (=, +=, -=, \*=, /=, %=, >>=, <<=, &=, |=, ^=, ++, --) shall not be used inside conditional expression in control statements `if`, `while`, `for` and `switch`.

**Justification** An instruction such as  
`if (x=y) { ...`  
 is ambiguous and unclear. One might think the author wanted to write `if (x==y) { ...`

Example:

```
// do not write
if (x -= dx) { ...
for (i=j=n; --i > 0; j--) {
..

// write
x -= dx;
if (x) { ...
for (i=j=n; i > 0; i--, j--)
{ ...
```

#### assexp Assignment inside expressions

**Description** Inside an expression:

- an *lvalue* has to be assigned only once,
- with multiple assignments, an assigned *lvalue* can appear only where it has been assigned.

**Justification** Removes ambiguity about the evaluation order.



Example:

```
// do not write
i = t[i++];
a=b+c+a;
i=t[i]=15;
```

## blockdecl      **Declarations in Blocks**

**Description**    Declarations must appear at the beginning of blocks.

**Justification**    Makes the code easier to read.

## boolean      **Use Proper Boolean Expressions**

**Description**    The tests in control structures must contain proper boolean expressions.

**Justification**    Makes the code easier to understand.

Example:

```
// do no write
while (1) {
  if (test) {
    for (i=1; function_call(i); i++) {

// write
AlwaysTrue = true;
while (AlwaysTrue == true) {
  if (test == true) {
    for (i=1; function_call(i); i++) {
```

## brkcont      **Break and Continue Forbidden**

**Description**    `break` and `continue` instructions are forbidden inside conditional expressions in control statements (`for`, `do`, `while`).

Nevertheless, the `break` instruction is allowed in the block instruction of the `switch` statement.

**Justification**    Like a `goto`, these instructions break down code structure. Prohibiting them in loops makes the code easier to understand.

## classuse      **Hidden class uses**

**Description**    Following expressions are not allowed: `u.v.a`, `u.v.f()`, `u.g().a`, `u.g().f()`, as well as expressions using the `->` operator.

**Justification**    Prevents from calling a class method not known in the user class (hidden use), through calls in series.

Example:

```
// do not write
myWindow.itsButton.push();
```

Manipulate the *myWindow* object from the *Window* class; access to the *itsButton* attribute; directly call the *push* method on it. But only the *Window* class and its interface, containing *itsButton*, are normally known, and not the *itsButton* attribute class, neither its public methods (including *push*).

Example:

```
// do not write
Error->pos.line;
```

There is a hidden use of *line*, which is not known from *Error*.

## condop      No ternary operator

**Description** The ternary conditional operator `? ... : ...` must not be used.

**Justification** Makes the code easier to read.

## constrdef      Default constructor

**Description** Each class must contain its default constructor explicitly.

**Justification** Makes sure the author has thought about the way to initialize an object of the class.

Example:

```
// write
class aClass {
...
aClass();
...
};
```

## ctrlblock      Blocks in Control Statements

**Description** Block statements shall always be used in control statements (*if*, *for*, *while*, *do*).

**Justification** Removes ambiguity about the scope of instructions and makes the code easier to read and to modify.

Example:

```
// do not write
if (x == 0) return;
else
    while (x > min)
        x--;

// write
if (x == 0) {
    return;
} else {
    while (x > min) {
        x--;
    }
}
```

## delarray      Use Delete [] For Array

**Description** Empty brackets must be used for delete when de-allocating arrays.

**Justification** Reliability: Ensures that the appropriate amount of memory is freed.

Example:

```
int *table = new int[7];
delete table;            // violation
delete [10] table; // violation
delete [] table;        // ok
```

**Limitations** There are some limitations to this rule when delete is used followed by a variable name.

These limitations do not apply in the case where delete is followed by a number in brackets.

This rule is not violated in the case of "complex" types:

Example 1:

```
int ** myarray = new int[2];
myarray[0] = new int[10];

delete myarray;        // violation
delete myarray[0];    // no violation
```

**Example 2:**

```
class A
{
    public:
        int *tab;
    ...
};
A var;
var.tab = new int[10];
delete var.tab;           // no violation
```

The rule is also not violated when the new operation is hidden:

```
int * create_array(int nb)
{
    return (new int[nb]);
}
...
int * myarray = create_array(10);
delete myarray;       // no violation
```

**destr            Destructor**

**Description** Each class must contain its destructor explicitly.

**Justification** Reliability: being sure that the author has thought about the way to destroy an object of the class.

Example:

```
// write
class aClass {
    ...
    ~aClass(aClass &object);
    ...
};
```

**fntype            Function Types**

**Description** Each function has to declare its type. If nothing is returned, it must be declared of void type.

**Justification** Portability.

## forinit Initialize For Loop Counter In For Head

**Description** Loop counters (in for loops) are to be initialized in the initialization statement within the loop. The loop counter is determined by the third element of the loop head, which is most frequently used to increment the loop counter.

In all the following examples, *i* is the loop counter.

**Justification** This way the loop counter is certain to have been initialized, and with a value that is visible alongside with the loop condition and increment. The loop is easier to understand and to control.

Example:

```
for (int i = 0; i < 10; i++) ... // ok
for (int i; i < 10; i++) ... // violation
for (int j = 0; j < 10; i++) ... // violation
for (int j = 10; i < j; i++) ... // violation
for (int j = 1; i < funct(j); i+=j) ... // violation
```

## frndclass Friend Classes

**Description** If friend classes are used, they must be declared at the beginning of the class (before member declaration).

## funcptr No Function Pointers

**Description** Do not use function pointers.

## globinit Global Variable Initialization

**Description** Global variables must be initialized when they are defined.

**Justification** Not all compilers give the same default values. Unexpected behaviour can be avoided with better control over variable values. Initializing global variables when they are declared ensures that they are initialized before being used.

## imptype Do Not Use Implicit Typing

**Description** Function, parameter, attribute or variable types must be declared explicitly.

This rule applies to non-ANSI compliant C++ code and should be turned off when using an ANSI compliant C++ compiler.

**Justification** Improves code portability.

Example:

```
// write
void aFunction(int value);

// do not write
aFunction(value);
```

## macroparenth Parenthesis in Macro Definitions

**Description** Each occurrence of the macro parameters shall be enclosed in parenthesis (or braces) inside the macro definition.

**Justification** Makes the code easier to read.

Example:

```
// do not write
#define GET_NAME(obj,ind) obj->name[ind]

// write
#define GET_NAME(obj,ind) (obj)->name[ind]
```

## mfunc Inline Functions instead of Macro-functions

**Description** Use inline functions instead of macro-functions.

**Justification** In comparison with macro-functions, inline functions enable the checking of their parameters types and do not allow side effects (such as `MIN(++i, j)` with the below example).

Example:

```
// write
inline char *GetName(aClass &object) {
return(object.name); }
inline min (int i, int j) { return (i<j)?i:j; }

// do not write
#define GetName(s) ((s)->name)
#define MIN(i,j) ((i)<(j)) ? (i) : (j)
```

## multiass No Multiple Assignment

**Description** Assignment operators (`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `>>=`, `<<=`, `&=`, `|=`, `^=`, `++`, `--`) must not be used more than once in each statement (declarations are also checked).

**Justification** Removes ambiguity about the evaluation order.

Example:

```

// do not write
b = c = 5;
a = (b++ * c) + 5;

// write
c = 5;
b = c;
b++;
a = (b * c) + 5;

```

## **nostruct      Keyword Struct Not Allowed**

**Description**    The keyword struct may not be used. If the parameter is specified, only C-style structs may be used.

**Parameters**    An optional string may be used (cstruct) to enable C-style structs to be used. When the cstruct rule is used, the possibilities allowed in C++ in a struct (such as access specifiers: private for example, or methods) are not to be used.

## **notemplate    Avoid Using Templates**

**Description**    Do not use templates.

**Justification**    Efficiency.

## **nothrow      No Throw Instructions**

**Description**    No exceptions may be raised by the user (the keyword throw may not be used).

## **nounion      No Union**

**Description**    The keyword union is not allowed.

## **parse        Parse Error**

**Description**    This rule identifies module parts that could not be parsed.

**Justification**    Enables to determine which portions of code have been analyzed and which portions of code have been rejected by Logiscope C++ *RuleChecker*.

**pmfrtn      Do Not Return Pointer To Member Data**

**Description** Member functions must not return a pointer or a non-const reference to member data.

**Justification** Helps to ensure that data encapsulation is respected.

**ptraccess      Pointer Access**

**Description** Use the ptr->fld syntax instead of the (\*ptr).fld syntax.

**ptrinit      Pointers Initialization**

**Description** Each auto variable that is explicitly declared as a pointer (using "\*"), must be initialized when declared.

**Justification** Makes sure pointer variables are correctly initialized before being used.

Example:

```
// write
int* y=&x;
...

// do not write
int *y ;
*y=&x ;
...
```

**rtnloctr      Do Not Return Pointer To Local Variable**

**Description** Functions must not return a pointer to a non-static local variable.

**Justification** This avoids dangling references of the pointer to the variable after its lifetime.

**sgdecl      A Single Variable per Declaration**

**Description** Variable declarations have the following formalism:  
type variable\_name;

It is forbidden to have more than one variable for the same type declarator.

**Justification** Makes the code easier to read.



Example:

```
// write
int width;
int length;

// do not write
int width, length;
```

## sglreturn      **A Single Return per Function**

**Description** Only one return instruction is allowed in a function.

**Justification** Maintainability : a basic rule for structured programming.

## slcom          **Use // Comments**

**Description** /\* \*/ comments are forbidden. Use only // comments.

**Justification** Makes the code easier to read.

## slstat         **One Statement per Line**

**Description** There must not be more than one statement per line.

A statement followed by a curly bracket (`instr {}`) or a curly bracket followed by a statement (`{ instr}`) is allowed in the same line, but not both of them (`instr { instr}`).

**Justification** Makes the code easier to read.

Example:

```
// write
x = x0;
y = y0;
while (IsOk(x)) {
    x++;
}

// do not write
x = x0; y = y0;
while (IsOk(x)) {x++;}
while (IsOk(x)) {x++;}
}
```

## typeinher      **Inheritance Type**

**Description** The inheritance type (public, protected, private) must be specified.

**Justification** Analysability

Example:

```
class inherclass : public Base1, private Base2
{...
```

## **vararg**      **Variable Number of Arguments**

**Description** Functions with a variable number of arguments are not allowed. Parameters of `va_list` type and `...` are forbidden in function declarations.

**Justification** Makes the code easier to understand.

## **voidptr**      **No Void Pointer**

**Description** The void pointer (`void *`) should not be used.

## **varinit**      **All Variables Must Be Initialized Before Being Used**

**Description** All variables must be initialized before they are used, without counting on the default value attributed by the compiler. Global variables, parameters of a function in the function body, and data fields of a class in its methods are considered to be initialized.

**Justification** Not all compilers give the same default values. Unexpected behaviour can be avoided with better control over variable values.

**Limitations** This rule is not violated in the following cases:

- If an array, a struct or a class are used, they will be considered initialized as soon as a part of them has been initialized.

For example:

```
int a[2];
int b[2] = {6, 7};
int h;

a[0] = b[0];    // no violation
h = a[1];      // no violation

struct {
    int i;
    int j;
} e, f;
e.i = 0;
g = e;         // no violation
```

This rule is violated in the following cases where initialization is uncertain:

- Using a variable in a function call is considered as "being used": if it is not initialized, the rule will be violated. This will occur whatever the use of the function, even initializing the variable.
- In cases including a conditional initialization, the rule is violated even though the variable may well be initialized.

```
int i, j, k;
j = func();
if (j)
    i = 0;
k = i;           // violation
```

- This applies even when there is an else branch:

```
int i, j, k;
j = func();
if (j)
    i = 0;
else
    i = 5;
k = i;           // violation
```

where initialization is certain.

- In the case of a loop, for example:

```
int j, k;
for (int i=0; i<glob; i++)
{
    j=func(i);
}
k = j;           // violation
```

where `glob` is a global variable, depending on the value of `glob`, `j` will have been initialized or not: the rule is violated, whether the loop condition occurs or not.

## 5.2.2 Customizable Rules

The rules specified in this section have parameters that can be modified to better fit to the effective programming rules applicable to the project or organisation: e.g. naming rules.

Please, refer to the Chapter *Customizing Metrics & Rules* for more details on how tailoring rule checking using Rule Set file.

### ansi                      **Function Declarations in ANSI Syntax**

<b>Description</b>	Function declaration and definition shall be written in ANSI syntax. It is possible to select two options among the following: <ul style="list-style-type: none"> <li>• <b>name</b>: parameters shall be named and their type indicated in function declaration,</li> <li>• <b>void</b>: empty parameter lists are forbidden.</li> </ul> By default, both options are selected.
<b>Parameters</b>	A list of character strings composed of chosen options listed above.
<b>Justification</b>	Makes the code easier to read and improves its portability.

Example:

```

// do not write
f(int, char*);
f();

// write
f(int a, char *b);
f(void);
```

### cmclass                      **A Single Class per Code File**

<b>Description</b>	In a code file, every function must belong to the same class. A C function is considered to belong to the main class. The first function encountered in the file sets the class for that file. By default, a code file has one of the suffixes *.cc, *.cxx, *.cpp, *.C or *.c.
<b>Parameters</b>	A string representing the types of modules (metric type) that should be considered as code files.
<b>Justification</b>	Makes the code easier to read.
<b>Limitation</b>	Friend functions of a class that don't have a scope are considered to belong to the main class.

## cmdef      **Classes in Code File**

- Description** A code file must not contain any class declaration.  
A C function is considered to belong to the main class.  
By default, a code file has one of the suffixes \*.cc, \*.cxx, \*.cpp, \*.C or \*.c.
- Parameters** A string representing the types of modules (metric type) that should be considered as code files.
- Justification** Makes the code easier to read.

## const      **Literal Constants**

- Description** Numbers and strings have to be declared as constants instead of being used as literals inside a program.  
Specify allowed literal constants. By default allowed literal constants are "", " ", "0" and "1".
- Parameters** A list of character strings representing allowed literal constants.  
A special parameter can be used: LOG\_SWITCH\_CONST. If present, it must be the first parameter of the list. When activated it allows constants to be used in switch cases.
- Justification** Makes maintenance easier by avoiding the scattering of constants among the code, often with the same value.
- Note** In the case of constants used in initializing lists (concerning array and struct structures), only the first five violations are detected.

Example:

```
// do not write
char tab[100];
int i;
...
if (i == 7) {
    p = "Hello World.\n";
}

// write
#define TAB_SIZE 100
enum i_val { ok =7; ko =11};
const char HelloWorld[] = "Hello World.\n";
char tab[TAB_SIZE];
i_val i;
...
if (i == ok) {
    p = HelloWorld;
}
```

**constrcpy      Copy Constructor**

- Description** Each class must contain its copy constructor explicitly.
- Parameters** The string "dynalloc" which, if used, indicates that the rule has to be checked only if there is a class member which is a pointer
- Justification** Makes sure the author has thought about the way to copy an object of the class.

Example:

```
// write
class aClass {
...
aClass(const aClass &object); // "const" is optional
...
};
```

**dmaccess      Access to Data Members**

- Description** The class interface must be purely functional: data members definitions can be limited.  
By default, only the data members definition in the public part of a class are forbidden.
- Parameters** A list of character strings corresponding to the forbidden access specifiers for the data members.
- Justification** The good way to modify the state of an object is via its methods, not its data members. The data members of a class should be private or at least protected.

**exprcplx      Expressions Complexity**

- Description** Expressions complexity must be smaller than a limit given as a parameter. This complexity is calculated with the associated syntactic tree, and its number of nodes.  
By default, the maximum authorized complexity level is 13.
- Parameters** A number representing the maximum authorized complexity level.
- Justification** Makes the code easier to read.

Example:

For instance, this expression:

$$(b+c*d) + (b*f(c)*d)$$

is composed of 8 operators and 7 operands.

The associated syntactic tree has 16 nodes, so if the limit is under 16, there will be a rule violation.

## exprparenth Parentheses in Expressions

**Description** In expressions, every binary and ternary operator shall be put between parentheses.

It is possible to limit this rule by using the **partpar** option. The following rule is then applied: when the right operand of a "+" or "\*" operator uses the same operator, omit parentheses for it. In the same way, omit parentheses in the case of the right operand of an assignment operator. Moreover, omit parentheses at the first level of the expression.

By default, the **partpar** option is selected.

**Parameters** The character string "**partpar**", which, if used, allows programmers not to put systematically parentheses, according to the rule above.

**Justification** Reliability, Maintainability: Removes ambiguity about the evaluation priorities.

Example:

```
// do not write
result = fact / 100 + rem; // Violation
// write
result = ((fact / 100) + rem); // Ok
// or write, with the partpar option
result = (fact / 100) + rem;
// with the partpar option, write
result = (fact * ind * 100) + rem + 10 + power(coeff,c);
// instead of
result = ((fact * (ind * 100)) + (rem + (10 + power(coeff,c))));
```

## funcres Reserved Functions

**Description** Certain names cannot be used for the declaration or definition of functions, and for function calls.

By default, no function names are forbidden.

**Parameters** A list of character strings representing the function names considered as reserved.

**Justification** Portability: Prevents from the use of system functions that are non portable or dangerous.

Example:

```
// if the system function is forbidden, do not write
int
system(char *command);
int
system(char *command)
{
...
}
system("cp file /tmp");
```

## goto Goto Statement

**Description** The `goto` statement must not be used.

By default, all `goto` statements are forbidden.

**Parameters** A list of strings specifying labels which are authorized with the `goto` statement.

**Justification** Maintainability Insures that structured programming rules are respected, so the code is easier to understand. The `goto` statement often reveals an analysis error and its systematic rejection improves the code structure.

## Headercom Module Header Comment

**Description** Modules must be preceded by a header comment.

It is possible to define a format for this comment depending on the type of the module as it is defined in metric type.

By default, a header comment with the name of the file, its author, its date and possible remarks is required for header and code files (see below example).

**Parameters** Two lists of character strings: the first one for the header files, and the second for the code files. Each list begins with the string "HEADER" or "CODE", followed by strings representing the associated regular expressions.

**Justification** Makes the code easier to read.



Example of the default required header comment:

```

////////////////////////////////////
////
// Name: program
// Author: Andrieu
// Date: 08/07/96
// Remarks: example of comments
////////////////////////////////////
////

```

## headercom    **Function and Class Header Comments**

- Description**    Functions and classes must be preceded by a comment.
- It is possible to define a format for this comment depending on the type of the function definition or declaration, or class definition (func\_glob\_def, func\_glob\_decl, func\_stat\_def, func\_stat\_decl, class).
- By default, only a comment beginning with "/\*" is required for functions or classes.
- Parameters**    Five lists of character strings concerning the five cases listed above. Each list begins with one of the five strings (func\_glob\_def for instance), followed by a string representing the regular expression.
- Justification**    Makes the code easier to read.

## hmclass        **A Single Class Definition per Header File**

- Description**    A header file must not contain more than one class definition.
- Nested classes are tolerated.
- By default, a header file corresponds to the filter \*.{h,hh,H,hxx,hpp}.
- Parameters**    A string representing types of modules (metric type) that should be considered as header files.
- Justification**    Makes the code easier to read.

## hmdef           **Header File Contents**

- Description**    Header files may not contain some of language statements (data and function definitions).
- The forbidden language items are function definitions (func-stat-def, func-glob-def) and data definitions (var-stat, var-glob).

By default, a header file corresponds to the filter `*.{h, hh, H, hxx, hpp}`.

**Parameters** A string representing types of modules (metric type) that should be considered as header files.

**Justification** The implementation of a class should not be found in header files.

## hmstruct Header File Structure

**Description** The main structure of header files should be:

```
#ifndef <IDENT>
#define <IDENT>
...
#endif
```

or

```
#if !defined (<IDENT>)
#define <IDENT>
...
#endif
```

where `<IDENT>` is an identifier built from the name of the header file.

The comparison is made only on alphanumeric characters and is not case sensitive.

The part of the filename taken into account is between the MINth and the MAXth characters (including them). This character string should be found in the identifier according to the above comparison rules.

By default, the MIN value is 1 and the MAX value is 999 and a header file corresponds to the filter `*.{h, hh, H, hxx, hpp}`.

**Parameters** A MINMAX couple of values giving the part of the filename to take into account, and a list of character strings giving the list of file types to be considered as header files for this rule. The types are those defined by the metric type.

**Justification** Prevents multiple inclusions of header files.

Example:

```
// if the parameter is MINMAX 4 9, the following contents
// of file div_audit_env.h is correct
#ifndef AUDIT_H
#define AUDIT_H
...
#endif
```

**identfmt      Identifier Format**

- Description** The identifier of a function, type or variable declared in a module must have a format corresponding to the category of the declaration.
- By default, the only restrictions concern the constants and the macros, which must have no lower case letter.
- Parameters** A list of couples of character strings; the first string of the couple represents the declaration category name, the second one the regular expression associated to that category.
- Justification** Makes the code easier to understand.

**identl      Identifier Length**

- Description** The length of a function, type or variable identifier has to be between a minimum and a maximum value.
- By default, the methods and functions must have between 4 and 25 characters, the types, variables, constants, macros and classes between 5 and 25, and the other identifiers between 1 and 25.
- Parameters** A list of couples of character strings; the first string of the couple represents the declaration category name, the second one the MINMAX expression associated.
- Justification** Makes the code easier to read.

**identres      Reserved Identifiers**

- Description** Some identifiers may be forbidden in declarations. For instance, names used in compilation directives or in libraries.
- By default, there are no reserved identifiers.
- Parameters** A list of character strings representing reserved identifiers.
- Justification** Improves code portability.

**incltype      Included Modules Type**

- Description** Only some types of modules are allowed to be included in other modules.
- By default, header modules can be included in header and code modules.

**Parameters** Lists of lists of character strings, each list being comprised of a string representing a type of module (metric type), followed by strings representing the types of modules that may be included in it.

**Justification** Improves code structuring.

## **inldef**      **Inline Functions Declaration and Definition**

**Description** Inline functions must be declared in their class and defined outside of it.

**Parameters** The string “private” which is an optional parameter. When the parameter is used, private inline functions must be defined in the class definition file (.cpp file), other inline functions must be defined in the class declaration file (.h file).

**Justification** Makes the code easier to read.

## **macrocharset** **Characters Used in Macros**

**Description** Some characters may be forbidden in the writing of the definitions of macro-functions and macro-constants (not in their name).

The two cases are treated separately.

By default, no characters are forbidden in macros.

**Parameters** A list of two couples of character strings; the first string of the couple is "constant" or "function", and the second one a string composed by the associated forbidden characters.

**Justification** Improves code portability.

## **mconst**      **Macro Constant Usage**

**Description** The usage of macro constants shall be limited.

It is possible to choose between three options:

- **var**: global or static variables are used for string constants, other constants could be defined by macros (this is the default option),

Example:

```
// write
const char *string = "Hello world!\n";
#define value 3

// do not write
#define string "Hello world!\n"
```

- **const**: `const` data are always used instead of macros,

Example:

```
// write
const char *string = "Hello world!\n";
const int value = 3;

// do not write
#define string "Hello world!\n"
#define value 3
```

- **nodefine**: only compilation flags and macro functions are allowed.

Example:

```
// write
#define VERBOSE
#define min(x,y) ((x)<(y)?(x):(y))

// do not write
#define value 3
#define current_value f(tab[0])
```

**Parameters** One of the three character strings explained above.

**Justification** Limits the use of macro-constants.

## **mname**      **File Names**

**Description** A file name and the name of the class declared or defined in this file must be closely related.

The comparison is made only on alphanumeric characters and is not case sensitive.

The extension of the file name is not taken into account.

The part of the file name taken into account to correspond to the name of the class is between the MIN and the MAX characters (these included). This character string should be found in the identifier according to the above comparison rules.

By default, the part of the file name taken into account is between the characters 1 and 5.

**Parameters** A MINMAX couple of values giving the part of the file name to take into account.

**Justification** Makes the application easier to understand.

**Example:**

```

if the MINMAX parameters are 4 and 10, and the file name
is
    My_Graph_Node.h
then the part of the file name that should be found in
the class name is:
    GRAPHN
(the first 10 characters: My_Graph_N,
minus the first 3: Graph_N,
minus non alphanumeric characters: GraphN)

Then, the class name that the file is based upon could
be one of the following declarations
    class CLA_Graph_Node { ...}
    class Graph_Node { ...}
    class Graph_Node_Def { ...}
    class graphnode { ...}
But not the following ones
    class Graph { ...}
    class NodeGraph { ...}

```

**nopreproc      No Pre-processing Instructions**

**Description** No pre-processing instructions may be used, except for those specified in the parameter list.

**Parameters** A list of strings defining the exceptions to this rule. The list can be empty. By default, only #line and # alone may not be used.

“define”: #define may be used

“include”: #include may be used

“if”: #if, #ifdef and #ifndef may be used

“pragma”: #pragma may be used

“undef”: #undef may be used

“line”: #line may be used

“error”: #error may be used

“none”: # may be used alone

**Justification** Makes the code easier to read and understand.

**operass      Assignment Operator**

**Description** Each class must explicitly contain at least one assignment operator.

**Parameters** The string "dynalloc" which, if used, indicates that the rule has to be checked only if there is a class member which is a pointer

**Justification** Makes sure the author has thought about the way to assign an object of the class.

Example:

```
// write
class aClass {
...
operator = (const aClass &object); // "const" is optional
...
};
```

## parammode Parameters Mode

**Description** In function definitions, the parameters mode used (IN, OUT or INOUT) must be indicated.

By default, the three modes "IN", "OUT" and "INOUT" are authorized.

**Parameters** A list of character strings representing the authorized keywords (their order does not matter).

**Justification** Enables to control parameter passing.

Example:

```
// write
int Multiply(IN Matrix *m, IN Vector *v, OUT Matrix *result);
```

## sectord "public", "private" and "protected" Sections Order

**Description** In a class declaration, sections defined by the access specifiers must follow a particular order, given in the parameters of the rule.

An empty string can be used (in the first position), representing the first section without any specifier.

**Note** Class definitions have not to contain all the access specifiers defined in the standard.

By default no particular order is given.

**Parameters** A list of character strings representing the access specifiers in the wanted order.

**Justification** Makes the code easier to read.

Example:

```

// if the standard has the following strings in this order:
// "", "private", "protected" and "public",
// following declarations are allowed
class aClass {
    int i ;
    protected:
    void p();
};
class aClass {
    protected:
    int i ;
    public:
    void p();
};

// and not the following ones:
class aClass {
    protected:
    ...;
    private:
    ... ;
};
class aClass {
    protected:
    ...;
    protected:
    ... ;
};

```

## **sgancstr**      **Single Ancestor**

**Description** All classes must have a same direct or indirect ancestor. The ancestor can be specified as a parameter.

**Parameters** A string representing the name of the ancestor. The parameter is optional.

## **swdef**      **default within switch**

**Description** A default case is mandatory within a `switch` in order to cover unexpected cases.

By default, the `default` case has to be the last one.

**Parameters** The character string "last", which, if used, specifies that the default case has to be the last one.

**Justification** All cases must be provided for in a `switch`.

## **swend**      **End of Cases in a "switch"**

**Description** Each case in a `switch` shall end with `break`, `continue`, `goto`, `return` or `exit`. Several consecutive case labels are allowed.



By default, such instructions are not mandatory for the last case.

**Parameters** The character string "nolast", which, if used, allows not to have one of these instructions in the last case.

**Justification** Makes the code easier to understand and reduces the risk of errors.

## **varstruct**      **Struct and Union Variables**

**Description** Variables must not be directly declared using a `struct` or an `union` structure.

An intermediate type must be automatically used.

**Parameters** The string "nostruct" which, if used, prevents from declaring a `struct` or `union` variable except in a `typedef` structure.

This option has no meaning in C++ programs, where class declarations are always allowed outside a `typedef` structure.

**Justification** Makes the code easier to understand.

Example:

```
// write
typedef struct {
    ...
} typeName;
typeName varName;
struct structName;
typedef struct structName {
    ...
    struct structName *ptr;
} typeName;
typeName varName;

// do not write
struct {
    ...
} varName;
// do not write, if the "nostruct" option is used
struct structName {
    ...
};
struct structName varName;
```

## **typeres**      **Reserved Types**

**Description** Some types may be forbidden for variables or functions.

It is possible to define the list of types that are forbidden for variables (`extern`, `static`, and `automatic` variables) and the list of types that are forbidden for functions.

The type specifiers and qualifiers are forbidden in any order and even if they are merged with other specifiers or qualifiers.

These types are allowed in `typedef` definition.

**Parameters** Two lists of strings beginning by the keywords "data" or "function". The other items of the list are strings containing the forbidden groups of type specifiers or type qualifiers separated by spaces ( ' ').

**Justification** Not relying on predefined types improves code portability.

## 5.2.3 MISRA-C++ 2008 Programming Rules

The Motor Industry Software Reliability Association has published a list of rules for the use of the C++ programming language for critical systems :

- *MISRA-C++:2008 Guidelines for the use of the C++ language critical systems* - June 2008 [MISRA-C++:2008].

*Logiscope RuleChecker C++* provides the MISRAC++2008 Rule Set including some of the key MISRA C++ programming rules:

- Please refer to [MISRA-C++:2008] for a complete specification of the rules.

### **MISRA\_Rule0\_1\_1:**

A project shall not contain unreachable code.

### **MISRA\_Rule0\_1\_10**

Every defined function shall be called at least once.

### **MISRA\_Rule0\_1\_3**

A project shall not contain unused variables.

### **MISRA\_Rule0\_1\_4**

A project shall not contain non-volatile POD variables having only one use.

### **MISRA\_Rule0\_1\_5**

A project shall not contain unused type declarations.

### **MISRA\_Rule2\_10\_1**

Different identifiers shall be typographically unambiguous.

### **MISRA\_Rule2\_10\_3**

A typedef name (including qualification, if any) shall be a unique identifier.

### **MISRA\_Rule2\_10\_4**

A class, union or enum name (including qualification, if any) shall be a unique identifier.

**MISRA\_Rule2\_10\_5**

The identifier name of a non\_member object or function with static storage duration should not be reused.

**MISRA\_Rule2\_13\_1**

Only those escape sequences that are defined in ISO/IEC 14882:2003 shall be used.

**MISRA\_Rule2\_13\_2**

Octal constants (other than zero) and octal escape sequences (other than "\0") shall not be used.

**MISRA\_Rule2\_13\_4**

Literal suffixes shall be upper case.

**MISRA\_Rule2\_3\_1**

Trigraphs shall not be used.

**MISRA\_Rule2\_7\_1**

The character sequence /\* shall not be used within a C-style comment.

**MISRA\_Rule2\_7\_2**

Sections of code shall not be "commented out" using C-style comments.

**MISRA\_Rule2\_7\_3**

Sections of code should not be "commented out" using C++ comments.

**MISRA\_Rule3\_1\_1**

It shall be possible to include any header file in multiple translation units without violating the One Definition Rule.

**MISRA\_Rule3\_1\_3**

When an array is declared, its size shall either be stated explicitly or defined implicitly

by initialization.

### **MISRA\_Rule3\_9\_2**

typedefs that indicate size and signedness should be used in place of the basic numerical types.

**Note:** Renaming of the standard **typeres** rule with the following parameters:

```
LIST "data" "char" "int" "short" "long" "float" "double" "long double" END LIST
```

```
LIST "function" "char" "int" "short" "long" "float" "double" "long double" END LIST
```

### **MISRA\_Rule5\_0\_2**

Limited dependence should be placed on C++ operator precedence rules in expressions.

**Note:** Renaming of the standard **exprparenth** rule with the parameter “partpar”:

### **MISRA\_Rule5\_14\_1**

The right hand operand of a logical **&&** or **||** operator shall not contain side effects.

### **MISRA\_Rule5\_18\_1**

The comma operator shall not be used.

### **MISRA\_Rule5\_2\_1**

Each operand of a logical **&&** or **||** shall be a postfix-expression.

### **MISRA\_Rule5\_2\_10**

The increment (**++**) and decrement (**--**) operators should not be mixed with other operators in an expression.

### **MISRA\_Rule5\_2\_11**

The comma operator, **&&** operator and the **||** operator shall not be overloaded.

### **MISRA\_Rule5\_2\_4**

C\_style casts (other than void casts) and functional notation casts (other than explicit constructor calls) shall not be used.

**MISRA\_Rule5\_2\_5**

A cast shall not remove any const or volatile qualification from the type of a pointer or reference.

**MISRA\_Rule5\_3\_3**

The unary & operator shall not be overloaded.

**MISRA\_Rule6\_2\_2**

Floating\_point expressions shall not be directly or indirectly tested for equality or inequality.

**MISRA\_Rule6\_2\_3**

Before preprocessing, a null statement shall only occur on a line by itself; it may be followed by a comment, provided that the first character following the null statement is a white\_space character.

**MISRA\_Rule6\_3\_1**

The statement forming the body of a switch, while, do ... while or for statement shall be a compound statement.

**Note:** Renaming of the standard **ctrlblock** rule.

**MISRA\_Rule6\_4\_1**

An if ( condition ) construct shall be followed by a compound statement. The else keyword shall be followed by either a compound statement, or another if statement.

**MISRA\_Rule6\_4\_2**

All if ? else if constructs shall be terminated with an else clause.

**MISRA\_Rule6\_4\_4**

A switch label shall only be used when the most closely-enclosing compound statement is the body of a switch statement.

**MISRA\_Rule6\_4\_5**

An unconditional throw or break statement shall terminate every non-empty switch-

clause.

**MISRA\_Rule6\_4\_6**

The final clause of a switch statement shall be the default-clause.

**MISRA\_Rule6\_4\_8**

Every switch statement shall have at least one case- clause.

**MISRA\_Rule6\_5\_1**

A for loop shall contain a single loop-counter which shall not have floating type.

**Limitation:** Loop-counters modified within the for body are not accounted for when evaluating this rule

**MISRA\_Rule6\_5\_2**

If loop-counter is not modified by -- or ++, then, within condition, the loop-counter shall only be used as an operand to <=, <, > or >=.

**MISRA\_Rule6\_5\_3**

The loop-counter shall not be modified within condition or statement.

**MISRA\_Rule6\_5\_4**

The loop-counter shall be modified by one of: --, ++, -=n, or +=n; where n remains constant for the duration of the loop.

**MISRA\_Rule6\_6\_1**

Any label referenced by a goto statement shall be declared in the same block, or in a block enclosing the goto statement.

**MISRA\_Rule6\_6\_2**

The goto statement shall jump to a label declared later in the same function body.

**MISRA\_Rule6\_6\_4**

For any iteration statement there shall be no more than one break or goto statement used

for loop termination.

### **MISRA\_Rule6\_6\_5**

A function shall have a single point of exit at the end of the function.

### **MISRA\_Rule7\_3\_2**

The identifier main shall not be used for a function other than the global function main.

**Parameter:** A list of non usable function names .

For instance, on a Windows platform,

LIST "main" "WinMain" "DllMain" END LIST

**Default:** Only “main” is considered.

### **MISRA\_Rule7\_3\_3**

There shall be no unnamed namespaces in header files.

**Parameter:** A list of accepted header files extensions.

ex. LIST ".h" ".hpp" END LIST

**Default:** Only files with extensions “.h”, “.hxx” and “.hh” are considered as header files.

### **MISRA\_Rule7\_3\_4**

Using-directives shall not be used.

### **MISRA\_Rule7\_3\_6**

Using-directives and using-declarations (excluding class scope or function scope using-declarations) shall not be used in header files.

### **MISRA\_Rule7\_4\_3**

Assembly language shall be encapsulated and isolated.

### **MISRA\_Rule7\_5\_1**

A function shall not return a reference or a pointer to an automatic variable (including parameters), defined within the function.



**MISRA\_Rule7\_5\_2**

The address of an object with automatic storage shall not be assigned to another object that may persist after the first object has ceased to exist.

**MISRA\_Rule8\_0\_1**

An init-declarator-list or a member-declarator-list shall consist of a single init-declarator or member-declarator respectively.

**Note:** Renaming of the standard **sgdecl** rule.

**MISRA\_Rule8\_3\_1**

Parameters in an overriding virtual function shall either use the same default arguments as the function they override, or else shall not specify any default arguments.

**MISRA\_Rule8\_4\_1**

Functions shall not be defined using the ellipsis notation.

**Note:** Renaming of the standard **vararg** rule.

**MISRA\_Rule8\_4\_3**

All exit paths from a function with non-void return type shall have an explicit return statement with an expression.

**Limitations:** The rule only checks that all non-void functions have at least one return statement, and that all of its return statements return an expression

**MISRA\_Rule8\_5\_1**

All shall have a defined value before they are used.

**MISRA\_Rule8\_5\_3**

In an enumerator list, the = construct shall not be used to explicitly initialize members other than the first, unless all items are explicitly initialized.

**MISRA\_Rule9\_5\_1**

Unions shall not be used.

**Note:** Renaming of the standard **nounion** rule.

**MISRA\_Rule9\_6\_2**

Bit-fields shall be either bool type or an explicitly unsigned or signed integral type.

**Note:** In addition of wchar\_t, forbidden types (because potentially signed or unsigned) are int, char, short, long.

**MISRA\_Rule9\_6\_4**

Named bit-fields with signed integer type shall have a length of more than one bit.

**MISRA\_Rule10\_1\_1**

Classes should not be derived from virtual bases.

**MISRA\_Rule10\_2\_1**

All accessible entity names within a multiple inheritance hierarchy should be unique.

**MISRA\_Rule10\_3\_1**

There shall be no more than one definition of each virtual function on each path through the inheritance hierarchy.

**MISRA\_Rule10\_3\_2**

Each overriding virtual function shall be declared with the virtual keyword.

**MISRA\_Rule10\_3\_3**

A virtual function shall only be overridden by a pure virtual function if it is itself declared as pure virtual.

**MISRA\_Rule11\_0\_1**

Member data in non-POD class types shall be private.

**MISRA\_Rule12\_1\_2**

All constructors of a class should explicitly call a constructor for all of its immediate base classes and all virtual base classes.

**Limitation:** The rule checks that base class constructors are called in the initialization list, separated by colons. It does not check that base class constructors are called within the constructor body.

**MISRA\_Rule12\_1\_3**

All constructors that are callable with a single argument of fundamental type shall be declared explicit.

The fundamental types are:

- bool
- char "unsigned char" "signed char"
- short "short int" "signed short" "unsigned short"
- int "signed int" "unsigned int"
- \_\_int8 \_\_int16 \_\_int32 \_\_int64
- long "signed long" "unsigned long" "long long" "signed long long" "unsigned long long"
- float double "long double"
- \_\_wchar\_t

**Parameter:** A list of additional types.

ex. LIST "\_UI8" "\_UI16" "\_UI32" "\_UI64" END LIST

**MISRA\_Rule14\_7\_1**

All class templates, function templates, class template member functions and class template static members shall be instantiated at least once.

**MISRA\_Rule15\_0\_2**

An exception object should not have pointer type.

**MISRA\_Rule15\_0\_3**

Control shall not be transferred into a try or catch block using a goto or a switch statement.

**MISRA\_Rule15\_1\_1**

The assignment-expression of a throw statement shall not itself cause an exception to be thrown.

**MISRA\_Rule15\_1\_2**

NULL shall not be thrown explicitly.

**MISRA\_Rule15\_1\_3**

An empty throw (throw;) shall only be used in the compound-statement of a catch handler.

**MISRA\_Rule15\_3\_1**

Exceptions shall be raised only after start-up and before termination of the program.

**MISRA\_Rule15\_3\_7**

Where multiple handlers are provided in a single try-catch statement or function-try-block, any ellipsis (catch-all) handler shall occur last.

**MISRA\_Rule15\_5\_1**

A class destructor shall not exit with an exception.

**MISRA\_Rule15\_5\_3**

The terminate() function shall not be called implicitly.

**MISRA\_Rule16\_0\_1**

#include directives in a file shall only be preceded by other preprocessor directives or comments.

**MISRA\_Rule16\_0\_2**

Macros shall only be #define'd or #undef'd in the global namespace.

**MISRA\_Rule16\_0\_3**

#undef shall not be used.

**MISRA\_Rule16\_0\_4**

Function-like macros shall not be defined.

**Note:** Renaming of the standard **mfunc** rule.

**MISRA\_Rule16\_0\_5**

Arguments to a function-like macro shall not contain tokens that look like preprocessing directives.

**MISRA\_Rule16\_0\_6**

In the definition of a function-like macro, each instance of a parameter shall be enclosed in parentheses, unless it is used as the operand of # or ##.

**MISRA\_Rule16\_0\_8**

If the # token appears as the first token on a line, then it shall be immediately followed by a preprocessing token.

**MISRA\_Rule16\_1\_1**

The defined preprocessor operator shall only be used in one of the two standard forms.

**MISRA\_Rule16\_1\_2**

All #else, #elif and #endif preprocessor directives shall reside in the same file as the #if or #ifdef directive to which they are related.

**MISRA\_Rule16\_2\_1**

The pre-processor shall only be used for file inclusion and include guards.

**MISRA\_Rule16\_2\_2**

C++ macros shall only be used for include guards, type qualifiers, or storage class specifiers.

**MISRA\_Rule16\_2\_3**

Include guards shall be provided.

**MISRA\_Rule16\_2\_4**

The ', ", /\* or // characters shall not occur in a header file name.

**MISRA\_Rule16\_2\_5**

The \ character should not occur in a header file name.

### **MISRA\_Rule16\_2\_6**

The #include directive shall be followed by either a <filename> or "filename" sequence.

### **MISRA\_Rule16\_3\_1**

There shall be at most one occurrence of the # or ## operators in a single macro definition.

### **MISRA\_Rule16\_3\_2**

The # and ## operators should not be used.

### **MISRA\_Rule17\_0\_2**

The names of standard library macros and objects shall not be reused.

**Parameter:** The list of non reusable names.

**Default:**

LIST

```
"NULL" "EXIT_SUCCESS" "EXIT_FAILURE" "RAND_MAX" "__max" "__min"
"_MAX_PATH" "_MAX_DRIVE" "_MAX_DIR" "_MAX_FNAME" "_MAX_EXT"
"_OUT_TO_DEFAULT" "_OUT_TO_STDERR" "_OUT_TO_MSGBOX"
"_REPORT_ERRMODE" "errno" "_doserrno" "__argc" "__argv" "__wargv" "_environ"
"_wenviron" "_pgmptr" "_wpgmptr" "size_t" "wchar_t" "_onexit_t" "div_t" "ldiv_t"
```

END LIST

### **MISRA\_Rule17\_0\_3**

The names of standard library functions shall not be overridden.

**Parameter:** The list of non overridden names.

### **MISRA\_Rule17\_0\_5**

The setjmp macro and the longjmp function shall not be used.

**Note:** Renaming of the standard **funres** rule with the following parameter:

```
LIST "setjmp" "longjmp" END LIST
```

### **MISRA\_Rule18\_0\_1**

The C library shall not be used.

**Parameter:** The list of non reusable names.

**MISRA\_Rule18\_0\_2**

The library functions `atof`, `atoi` and `atol` from library `<cstdlib>` shall not be used.

**Note:** Renaming of the standard **funres** rule with the following parameter:

```
LIST "atof" "atoi" "atol" END LIST
```

**MISRA\_Rule18\_0\_3**

The library functions `abort`, `exit`, `getenv` and `system` from library `<cstdlib>` shall not be used.

**Note:** Renaming of the standard **funres** rule with the following parameter:

```
LIST "abort" "exit" "getenv" "system" END LIST
```

**MISRA\_Rule18\_0\_4**

The time handling functions of library `<ctime>` shall not be used.

**MISRA\_Rule18\_0\_5**

The unbounded functions of library `<cstring>` shall not be used.

**Note:** Renaming of the standard **funres** rule with the following parameter:

```
LIST "strcpy" "strcmp" "strcat" "strchr" "strspn" "strcspn" "strpbrk" "strrchr" "strstr"
"strtok" "strlen" END LIST
```

**MISRA\_Rule18\_2\_1**

The macro `offsetof` shall not be used.

**Note:** Renaming of the standard **funres** rule with the following parameter:

```
LIST "offsetof" END LIST
```

**MISRA\_Rule18\_4\_1**

Dynamic heap memory allocation shall not be used.

**MISRA\_Rule18\_7\_1**

The signal handling facilities of `<csignal>` shall not be used.

**MISRA\_Rule19\_3\_1**

The error indicator errno shall not be used.

**MISRA\_Rule27\_0\_1**

The stream input/output library <cstdio> shall not be used.



## 5.3 Scott Meyers Rules

The following rules come from two books written by Scott Meyers: "*Effective C++: 50 Specific Ways to Improve Your Programs and Designs*" (Addison-Wesley, second edition, 1997, ISBN: 0-201-92488-9) and "*More Effective C++: 35 New Ways To Improve Your Programs And Designs*" (Addison-Wesley, first edition, 1996, ISBN: 0-201-63371-X).

### **assignthis**      **Check for Assignment to "self" in Operator "="**

*This rule relates to Item 17 in "Effective C++".*

**Description**    Inside the definition of an assignment operator:  
 - the equality between the parameter and `this` or `*this` shall be checked;  
 - in case of equality, `*this` must be returned..

**Justification**   Ensures that self-assignment will work.

### **cast**              **Prefer C++-style Casts**

*This rule relates to Item 2 in "More Effective C++"*

**Description**    Use the C++-style casts (`static_cast`, `const_cast`, `dynamic_cast` and `reinterpret_cast`) instead of the general-purpose C-style cast.

**Justification**    The C-style cast does not allow to make a distinction between the different types of casts and it is not easy to detect.

### **catchref**        **Catch Exceptions by Reference**

*This rule relates to Item 13 in "More Effective C++"*

**Description**    In `catch` clauses references to exceptions must be indicated.

**Justification**    Improves code efficiency.

### **constrinit**      **Prefer Initialization to Assignment in Constructors**

*This rule relates to Item 12 in "Effective C++".*

**Description**    Non static data members must be initialized inside the member initialization list of the constructor(s) of the class.

**Justification**    Improves code efficiency.

## **convnewdel Adhere to Convention when Writing "new" and "delete" Operators**

*This rule relates to Item 8 in "Effective C++".*

- Description** Convention for writing operator `new`:
- the type of the return value shall be `void *`;
  - the type of the first parameter shall be `size_t`.
- Convention for writing operator `delete`:
- the type of the return value shall be `void`;
  - the type of the first parameter shall be `void *`;
  - in case of a second parameter, its type shall be `size_t`.
- Parameters** The string "static" which, if used, indicates that operator `new` and operator `delete` shall be declared `static`.
- Justification** Keeps the consistency with the default `new` and `delete` operators.

## **dataptr Data of Pointer Type**

*This rule relates to Item 10 in "More Effective C++".*

- Description** Class members which are pointers to objects are not allowed.
- Justification** Prevents resource leaks in constructors and simplifies destructors definitions.

## **delifnew Write Operator "delete" if you Write Operator "new"**

*This rule relates to Item 10 in "Effective C++".*

- Description** If operator `new` is declared inside a class, then operator `delete` shall be also declared inside the same class.
- Justification** `new` and `delete` operators work together.

## **excepspec Exception Specifications**

*This rule relates to Item 14 in "More Effective C++"*

- Description** Do not use exception specifications.
- Justification** Prevents violations of exception specifications, which are dangerous.

## **inlinevirt Inline Virtual Functions**

*This rule relates to Item 24 in "More Effective C++"*

**Description** Virtual functions shall not be declared `inline`.

**Justification** Improves code efficiency.

## **multinher      Multiple Inheritance Only Allowed for Inheriting Abstract Classes**

*This rule relates to Item 43 in "More Effective C++".*

**Description** If multiple inheritance is used, the classes inherited must be abstract, that is to say that they must contain at least one pure virtual method.

**Justification** Makes the overall design less complicated and the code easier to understand.

Example:

```

1st case:
    A and B are not abstract classes (they contain no pure
    virtual methods). C inherits A and B: the rule is vio-
    lated.

2nd case:
    A and B are abstract classes (they contain at least one
    pure virtual method each). C inherits A and B: the rule
    is not violated.

    Current limitation of this case:
    If class C remains abstract (A and/or B's pure virtual
    methods are not redefined in C) and if a class D inherits
    C and another abstract class, the rule will be violated
    for D, although it inherits only abstract classes.

3rd case:
    A is abstract, B is not, C is (has a pure virtual func-
    tion), and inherits A and B. C violates the rule, but is
    abstract for inheriting classes.

```

## **nonleafabs      Make non-leaf classes abstract**

*This rule relates to Item 33 in "More Effective C++".*

**Description** Non-leaf classes shall be abstract.

**Justification** Helps assignment do what most programmers expect and improves the design of classes.

## **normalnew      Avoid Hiding the "Normal" Form of "new"**

*This rule relates to Item 9 in "Effective C++".*

**Description** If operator `new` is declared one or several times inside a class, at least one of these declarations shall follow the "normal" form:  
 - the type of the first parameter shall be `size_t`;  
 - all other parameters, if any, shall have a default value.

**Justification** Lets the usual invocation form of `new` available.

## **overload      Never overload "&&", "||" and "," operators**

*This rule relates to Item 7 in "More Effective C++".*

**Description** "&&", "||" and "," operators must not be overloaded.

**Justification** Makes the code do what most programmers expect.

## **prepost      Distinguish between Prefix and Postfix Forms of Increment and Decrement Operators**

*This rule relates to Item 6 in "More Effective C++"*

**Description** Increment and decrement operators must be declared in the same manner as in the following example:

```
class Example {
public:
    Example& operator++();           // prefix ++
    const Example operator++(int);  // postfix ++
    Example& operator--();          // prefix --
    const Example operator--(int);  // postfix --
}
```

**Justification** Keeps the consistency with built-in types.

## **refclass      References of Classes**

*This rule relates to Item 22 in "Effective C++".*

**Description** Every parameters of class type shall be passed by reference.

**Justification** Improves the efficiency of the code.

## **returnthis      Return "\*this" in Assignment Operators**

*This rule relates to Item 15 in "Effective C++"*

**Description** Inside the definition of an assignment operator, the return value shall be `*this`.

**Justification** Allows chains of assignments and type conversions.

## **tryblock Try Blocks**

*This rule relates to Item 15 in "More Effective C++".*

**Description** Do not use try blocks.

**Justification** Efficiency.

## **trydestr Try Blocks in Destructors**

*This rule relates to Item 11 in "More Effective C++".*

**Description** If it is explicit, the definition of a destructor must contain a `try` and `catch` block.

**Justification** Prevents the call of `terminate` in case of exception propagation, and helps ensure that destructors do everything they are supposed to do.

## **virtdestr Virtual destructors**

*This rule relates to Item 14 in "Effective C++".*

**Description** Destructors of base classes must be declared `virtual`.

**Justification** Ensures that base and derived destructors are called before memory deallocation.



# Chapter 6

---

## *Customizing Standard Rules and Rule Sets*

### 6.1 Modifying the Rule Set

A Rule Set is user-accessible textual file containing the specification of the programming rules to be checked by Logiscope *RuleChecker*.

Specifying one or more Rule Set files is mandatory when setting up a Logiscope *RuleChecker* project.

The Rule Sets allow to adapt Logiscope *RuleChecker* verification to a specific context taking into the applicable coding standard.

- Rule checking can be activated or de-activated.
- Some rules have parameters that allow to customize the verification. Changing the parameters changes the behaviour of the rule checking.
- The default name of a standard rule can be changed to match the name and/or identifier specified in the applicable coding standard.  
The same standard rule can even be used twice with different names and different parameters.
- The default severity level of a rule can be modified.
- A new set of severity levels with a specific ordering: e.g. “Mandatory”, “Highly recommended”, “Recommended”. can be specified.

All these actions can be done by editing the Logiscope Rule Set(s) and changing the corresponding specifications. We highly recommend to make copies of the default Rule Set files provided with Logiscope *RuleChecker C++* before making changes.

Modifying Rule Set files is documented in *Logiscope IBM Rational - Basic Concepts* manual.

## 6.2 Customizing Rules

*The precise definition of these rules has been given in previous chapter.*

### ansi Function Declarations in ANSI Syntax

By default, the parameters **name** and **void** are both put:

```
STANDARD ansi ON LIST "name" "void" END LIST END STANDARD
```

To check that the parameters are named and their type indicated, just put the **name** parameter:

```
STANDARD ansi ON LIST "name" END LIST END STANDARD
```

To forbid the empty parameter lists, just put the **void** parameter:

```
STANDARD ansi ON LIST "void" END LIST END STANDARD
```

### cmclass A Single Class per Code File

By default, the type of modules considered as code files is **CODE**, which corresponds to the suffixes **\*.cc**, **\*.cxx**, **\*.cpp**, **\*.C** or **\*.c**, as defined by the metric **type**:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h, hh, H, hxx}" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD cmclass ON LIST "CODE" END LIST END STANDARD
```

Change the definition of the **CODE** module type if it does not suit the application:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h, hh, H, hxx}" END LIST
LIST "CODE" "*.CC" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD cmclass ON LIST "CODE" END LIST END STANDARD
```

Or choose to add a new module type (**MY\_CODE**, for example):

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h, hh, H, hxx}" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "MY_CODE" "*.CC" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD cmclass ON LIST "MY_CODE" END LIST END STANDARD
```

### cmdef Classes in Code File

By default, the type of modules considered as code files is **CODE**, which corresponds to the suffixes **\*.cc**, **\*.cxx**, **\*.cpp**, **\*.C** or **\*.c**, as defined by the metric **type**:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h, hh, H, hxx}" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD cmdef ON LIST "CODE" END LIST END STANDARD
```



Change the definition of the **CODE** module type if it does not suit the application:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*. {h, hh, H, hxx}" END LIST
LIST "CODE" "*.CC" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD cmdef ON LIST "CODE" END LIST END STANDARD
```

Choose to add a new module type (**MY\_CODE**, for example):

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*. {h, hh, H, hxx}" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "MY_CODE" "*.CC" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD cmdef ON LIST "MY_CODE" END LIST END STANDARD
```

## const Literal Constants

By default, the allowed literal constants are "", " ", "0" and "1":

```
STANDARD const ON LIST " "" "" "0" "1" END LIST END STANDARD
```

To allow the literal constant **MY\_CST**, but forbid the constant **1**:

```
STANDARD const ON LIST " "" "" "0" "MY_CST" END LIST END STANDARD
```

## constrcpy Copy Constructor

By default the "**dynalloc**" parameter is not put:

```
STANDARD constrcpy ON END STANDARD
```

To look for the copy constructor only if there is a class member which is a pointer:

```
STANDARD constrcpy ON "dynalloc" END STANDARD
```

## convnewdel Adhere to Convention when Writing "new" and "delete" Operators

By default the "**static**" parameter is not put:

```
STANDARD convnewdel ON END STANDARD
```

To declare **static new** and **delete** operators :

```
STANDARD convnewdel ON "static" END STANDARD
```

## dmaccess Access to Data Members

By default, only the data members in the public part of a class are forbidden:

```
STANDARD dmaccess ON LIST "public" END LIST END STANDARD
```

To forbid the data members in the public and protected part of a class:

```
STANDARD dmaccess ON LIST "public" "protected" END LIST END STANDARD
```

## exprcplx Expressions Complexity

By default, the maximum authorized complexity level is 13:

```
STANDARD exprcplx ON MINMAX 0 13 END STANDARD
```

To change this value to 16, for example:

```
STANDARD exprcplx ON MINMAX 0 16 END STANDARD
```

## exprparenth Parenthesis in Expressions

By default, the **partpar** parameter is put:

```
STANDARD exprparenth ON "partpar" END STANDARD
```

For a stricter rule, remove this parameter:

```
STANDARD exprparenth ON END STANDARD
```

## funcres Reserved Functions

By default, no function names are forbidden:

```
STANDARD funcres ON LIST END LIST END STANDARD
```

To forbid the functions **system** and **malloc**, for example:

```
STANDARD funcres ON LIST "system" "malloc" END LIST END STANDARD
```

## goto Goto Statement

By default, all `goto` statements are forbidden:

```
STANDARD goto ON LIST END LIST END STANDARD
```

To authorize the statements `goto ok;` and `goto error;`:

```
STANDARD goto ON LIST "ok" "error" END LIST END STANDARD
```

## Headercom Module Header Comments

It is possible to define a format for the header comment depending on the type of the module as it is defined in metric **type**.

The format of the comment is defined as a list of regular expressions that shall be found in the header comment in the order of declaration.

Formats are defined by regular expressions. The regular expression language is a subset of the one defined by the Posix 1003.2 standard (Copyright 1994, the Regents of the University of California).

A regular expression is comprised of one or more non-empty branches, separated by the "|" character.

A branch is one or more atomic expressions, concatenated.

Each atom can be followed by the following characters:

- \* - the expression matches a sequence of 0 or more matches of the atom,
- + - the expression matches a sequence of 1 or more matches of the atom,
- ? - the expression matches a sequence of 0 or 1 match of the atom,
- {i} - the expression matches a sequence of i or more matches of the atom,
- {i,j} - the expression matches a sequence of i through j (inclusive) matches of the atom.

An atomic expression can be either a regular expression enclosed in "()", or:

- [...] - a brace expression, that matches any single character from the list enclosed in "[]",
- [^...] - a brace expression that matches any single character not from the rest of the list enclosed in "[]",
- . - it matches any single character,
- ^ - it indicates the beginning of a string (alone it matches the null string at the beginning of a line),
- \$ - it indicates the end of a string (alone it matches the null string at the end of a line).

For more details, please refer to the related documentation.

Example:

```
"._Ptr" matches strings like "abc_Ptr", "hh_Ptr", but not "_Ptr",
"T[a-z]*" matches strings like "Ta", "Tb", "Tz",
"[A-Z][a-z0-9_]*" matches strings like "B1", "Z0", "Pp", "P_1_a".
```

By default, a header comment with the name of the file, its author, its date and possible remarks is required for files of the **HEADER** and **CODE** type (for the signification of these types, see in [Paragraph , \*cmclass A Single Class per Code File\*](#)):

```
STANDARD Headercom ON
LIST "HEADER"          "Name: [a-z]*" "Author: [A-Z][a-z]*"
                        "Date: [0-9][0-9]/[0-9][0-9]/[0-9][0-9]"
                        "Remarks:" END LIST
LIST "CODE"            "Name: [a-z]*" "Author: [A-Z][a-z]*"
                        "Date: [0-9][0-9]/[0-9][0-9]/[0-9][0-9]"
                        "Remarks:" END LIST
END STANDARD
```

Example of required header:

```
////////////////////////////////////
// Name: program
// Author: Andrieu
// Date: 08/07/96
// Remarks: example of comments
////////////////////////////////////
```

## headercom Function and Class Header Comments

It is possible to define a format for the comment preceding a function or a class, depending on the type of the function definition or declaration, or class definition (**func\_glob\_def**, **func\_glob\_decl**, **func\_stat\_def**, **func\_stat\_decl**, **class**).

The format of the comment is defined as a list of regular expressions (see in [Paragraph , \*Headercom Module Header Comments\*](#)) that shall be found in the comment in the order of declaration.

By default, only a comment beginning with "/\*" is required for functions or classes:

```
STANDARD headercom ON
LIST "class"           "/*" END LIST
LIST "func_glob_def"   "/*" END LIST
LIST "func_glob_decl" "/*" END LIST
LIST "func_stat_def"   "/*" END LIST
LIST "func_stat_decl" "/*" END LIST
```

```
END STANDARD
```

Here is another example, with different required comments depending on the item type:

```
STANDARD headercom ON
LIST "class" "Name of the class:"
"Filename:"

END LIST
LIST "func_glob_def" "Definition of the extern function:"
"Author: [A-Z][a-z]*"

END LIST
LIST "func_glob_decl" "Declaration of the extern funciton:"
"Date: [0-9][0-9]/[0-9][0-9]/[0-9][0-9]"

END LIST
LIST "func_stat_def" "Definition of the static function:"
"Remarks:"

END LIST
LIST "func_stat_decl" "Declaration of the static function:"
"Purpose:"

END LIST
END STANDARD
```

## hmclass A Single Class Definition per Header File

By default, the type of modules considered as header files is **HEADER**, which corresponds to the filter **\*.{h, hh, H, hxx, hpp}**, as defined by the metric **type**:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*. {h, hh, H, hxx}" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmclass ON LIST "HEADER" END LIST END STANDARD
```

Change the definition of the **HEADER** module type if it does not suit the application:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.HH" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmclass ON LIST "HEADER" END LIST END STANDARD
```

Or choose to add a new module type (**MY\_HEADER**, for example):

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*. {h, hh, H, hxx}" END LIST
LIST "MY HEADER" "*.HH" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmclass ON LIST "MY_HEADER" END LIST END STANDARD
```

## hmdef Header File Contents

By default, the type of modules considered as header files is **HEADER**, which corresponds to the filter **\*.{h, hh, H, hxx, hpp}**, as defined by the metric **type**:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*. {h, hh, H, hxx}" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
```

```
STANDARD hmdef ON LIST "HEADER" END LIST END STANDARD
```

Change the definition of the **HEADER** module type if it does not suit the application:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.HH" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmdef ON LIST "HEADER" END LIST END STANDARD
```

Or choose to add a new module type (**MY\_HEADER**, for example):

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h, hh, H, hxx}" END LIST
LIST "MY_HEADER" "*.HH" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmdef ON LIST "MY_HEADER" END LIST END STANDARD
```

## hmstruct Header File Structure

By default, the MIN value is 1 and the MAX value is 999 and the type of modules considered as header files is **HEADER**, which corresponds to the filter **\*.{h, hh, H, hxx, hpp}**, as defined by the metric type:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h, hh, H, hxx}" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmstruct ON MINMAX 1 999 LIST "HEADER" END LIST END STANDARD
```

Change the definition of the **HEADER** module type if it does not suit the application:

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.HH" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmstruct ON MINMAX 1 999 LIST "HEADER" END LIST END STANDARD
```

Or choose to add a new module type (**MY\_HEADER**, for example):

```
METRIC module type OFF FORMAT "30"
LIST "HEADER" "*.{h, hh, H, hxx}" END LIST
LIST "MY_HEADER" "*.HH" END LIST
LIST "CODE" "*.cc" "*.cxx" "*.cpp" "*.C" "*.c" END LIST
LIST "INTERFACE" "*.i" END LIST
LIST "YACC" "*_y.c" END LIST
END METRIC
STANDARD hmstruct ON MINMAX 1 999 LIST "MY_HEADER" END LIST END STANDARD
```

Change the MINMAX values:

```
STANDARD hmstruct ON MINMAX 4 9 LIST "HEADER" END LIST END STANDARD
```

## identfmt Identifier Format

It is possible to define a format for each of the categories listed below:

NAME	DESCRIPTION	DEFAULT
type	type name	any

type_obj	object type name	type, any
type_array	array type name	type, any
type_array_obj	object array type name	type_array, type_obj, type, any
type_ptr	pointer type name	type, any
type_ptr_obj	object pointer type name	type_obj, type_ptr, type, any
type_ref	reference type	type_ptr, type, any
type_ref_obj	object reference type	type_obj, type_ref, type_ptr, type, any
variable	variable name	any
variable_obj	object variable name	variable, any
variable_array	array variable name	variable, any
variable_array_obj	object array variable name	variable_obj, variable_array, variable, any
variable_ptr	pointer variable name	variable, any
variable_ptr_obj	object pointer variable name	variable_obj, variable_ptr, variable, any
variable_ref	reference variable name	variable_ptr, variable, any
variable_ref_obj	object reference variable name	variable_obj, variable_ref, variable_ptr, variable, any
type_func	function type name	function, type, any
type_struct	structured type name	type, any
type_struct_item	structure item name	variable, any
type_struct_item_obj	object structure item name	type_struct_item, variable_obj, variable, any
type_struct_item_array	array structure item name	type_struct_item, variable_array, variable, any
type_struct_item_array_obj	object array structure item name	type_struct_item_obj, type_struct_item_array, type_struct_item, variable_array, variable_obj, variable, any
type_struct_item_ptr	pointer structure item name	type_struct_item, variable_ptr, variable, any
type_struct_item_ptr_obj	object pointer structure item name	type_struct_item_obj, type_struct_item, variable_ptr, variable_obj, variable, any

type_struct_item_ref	reference structure item name	type_struct_item_ptr, type_struct_item, variable_ptr, variable_ref, variable, any
type_struct_item_ref_obj	object reference structure item name	type_struct_item_ptr_obj, type_struct_item_obj, type_struct_item_ref, type_struct_item_ptr, type_struct_item, variable_obj, variable_ptr, variable_ref, variable, any
type_union	union type name	type, any
type_union_item	union item name	variable, any
type_union_item_obj	object union item name	type_union_item, variable_obj, variable, any
type_union_item_array	array union item name	type_union_item, variable_array, variable, any
type_union_item_array_obj	object array union item name	type_union_item_obj, type_union_item_array, type_union_item, variable_obj, variable_array, variable, any
type_union_item_ptr	pointer union item name	type_union_item, variable_ptr, variable, any
type_union_item_ptr_obj	object pointer union item name	type_union_item_obj, type_union_item, variable_obj, variable_ptr, variable, any
type_union_item_ref	reference union item name	type_union_item_ptr, type_union_item, variable_ref, variable_ptr, variable, any
type_union_item_ref_obj	object reference pointer union item name	type_union_item_ptr_obj, type_union_item_obj, type_union_item_ref, type_union_item_ptr, type_union_item, variable_obj, variable_ref, variable_ptr, variable, any
enum	enumerated type name	type, any
const_enum_item	enumerated type item name	const, any
class	class name	type, any

class_attr	class attribute name	variable, any
class_attr_obj	class object attribute name	class_attr, variable_obj, variable, any
class_attr_array	class array attribute name	class_attr, variable_array, variable, any
class_attr_array_obj	class object array attribute name	class_attr_obj, class_attr, variable_obj, variable_array, variable, any
class_attr_ptr	class pointer attribute name	variable_ptr, class_attr, variable, any
class_attr_ptr_obj	class object pointer attribute name	class_attr_obj, class_attr_ptr, class_attr, variable_obj, variable_ptr, variable, any
class_attr_ref	class reference attribute name	class_attr_ptr, class_attr, variable_ref, variable_ptr, variable, any
class_attr_ref_obj	class object reference attribute name	class_attr_ptr_obj, class_attr_obj, class_attr_ref, class_attr_ptr, class_attr, variable_obj, variable_ref, variable_ptr, variable, any
method	class method name	function, class_attr, any
namespace	name space name	any
function	function name	any
const	constant name	any
const_obj	constant object name	const, any
const_array	constant array name	const, any
const_array_obj	constant object array name	const_obj, const, any
const_ptr	constant pointer name	const, any
const_ptr_obj	constant object pointer name	const_obj, const, any
const_ref	constant reference name	const_ptr, const, any
const_ref_obj	constant object reference name	const_ptr_obj, const_obj, const_ref, const_ptr, const, any
var_stat	static variable name	variable, any
var_stat_obj	static object variable name	variable_obj, var_stat, variable, any
var_stat_array	static array variable name	variable_array, var_stat, variable, any



var_stat_array_obj	static object array variable name	variable_obj, variable_array, var_stat, variable, any
var_stat_ptr	static pointer variable name	var_stat, variable_ptr, variable, any
var_stat_ptr_obj	static object pointer variable name	var_stat_obj, var_stat_ptr, var_stat, variable_obj, variable_ptr, variable, any
var_stat_ref	static reference variable name	var_stat_ptr, var_stat, variable_ref, variable_ptr, variable, any
var_stat_ref_obj	static object reference variable name	var_stat_ptr_obj, var_stat_obj, var_stat_ref, var_stat_ptr, var_stat, variable_obj, variable_ref, variable_ptr, variable, any
var_glob	global variable name	variable, any
var_glob_obj	global object variable name	variable_obj, var_glob, variable, any
var_glob_array	global array variable name	variable_array, var_glob, variable, any
var_glob_array_obj	global object array variable name	variable_obj, variable_array, var_glob, variable, any
var_glob_ptr	global pointer variable name	var_glob, variable_ptr, variable, any
var_glob_ptr_obj	global object pointer variable name	var_glob_obj, var_glob_ptr, var_glob, variable_obj, variable_ptr, variable, any
var_glob_ref	global reference variable name	var_glob_ptr, var_glob, variable_ref, variable_ptr, variable, any
var_glob_ref_obj	global object reference variable name	var_glob_ptr_obj, var_glob_obj, var_glob_ref, var_glob_ptr, var_glob, variable_obj, variable_ref, variable_ptr, variable, any
var_auto	automatic variable name	variable, any
var_auto_obj	automatic object variable name	var_auto, variable_obj, variable, any
var_auto_array	automatic array variable name	var_auto, variable_array, variable, any

var_auto_array_obj	automatic object array variable name	var_auto, variable_obj, variable_array, variable, any
var_auto_ptr	automatic pointer variable name	var_auto, variable_ptr, variable, any
var_auto_ptr_obj	automatic object pointer variable name	var_auto_obj, var_auto_ptr, var_auto, variable_obj, variable_ptr, variable, any
var_auto_ref	automatic reference variable name	var_auto_ptr, var_auto, variable_ref, variable_ptr, variable, any
var_auto_ref_obj	automatic object reference variable name	var_auto_ptr_obj, var_auto_obj, var_auto_ref, var_auto_ptr, var_auto, variable_obj, variable_ref, variable_ptr, variable, any
macro	macro name	any
macro_func	function macro name	macro, function, any
macro_const	macro constant name	macro, const, any
macro_flag	macro flag name	macro, any
parameter	parameter name	variable, any
parameter_obj	object parameter name	parameter, variable_obj, variable, any
parameter_array	array parameter name	parameter, variable_obj, variable_array, variable, any
parameter_array_obj	object array parameter name	parameter_obj, parameter_array, parameter, variable_array_obj, variable_obj, variable_array, variable, any
parameter_ptr	pointer parameter name	parameter, variable_ptr, variable, any
parameter_ptr_obj	object pointer parameter name	parameter_obj, parameter_ptr, parameter, variable_ptr_obj, variable_obj, variable_ptr, variable, any

parameter_ref	reference parameter name	parameter_ptr, parameter, variable_ref, variable_ptr, variable, any
parameter_ref_obj	object reference parameter name	parameter_ptr_obj, parameter_obj, parameter_ref, parameter_ptr, parameter, variable_obj, variable_ref, variable_ptr, variable, any

The third column represents inherited categories: for instance, for no distinction between the **macro-func**, the **macro-const** and the **macro-flag** categories, just define a particular format for the **macro** categories, which is inherited by the previous ones.

A special keyword **any** is used to define the default value for all identifier categories not explicitly defined.

The format of the identifier is defined by a regular expression (see in [Paragraph , Headercom Module Header Comments](#)).

By default, the only restrictions concern the constants and the macros, which must have no lower case letter:

```

STANDARD identfmt ON
LIST "any"                ".*"
    "type"                ".*"
    "type_obj"            ".*"
    "type_array"          ".*"
    "type_array_obj"      ".*"
    "type_ptr"            ".*"
    "type_ptr_obj"        ".*"
    "type_ref"            ".*"
    "type_ref_obj"        ".*"
    "variable"            ".*"
    "variable_obj"        ".*"
    "variable_array"      ".*"
    "variable_array_obj"  ".*"
    "variable_ptr"        ".*"
    "variable_ptr_obj"    ".*"
    "variable_ref"        ".*"
    "variable_ref_obj"    ".*"
    "type_func"           ".*"
    "type_struct"         ".*"
    "type_struct_item"    ".*"
    "type_struct_item_obj" ".*"
    "type_struct_item_array" ".*"
    "type_struct_item_array_obj" ".*"
    "type_struct_item_ptr" ".*"
    "type_struct_item_ptr_obj" ".*"
    "type_struct_item_ref" ".*"
    "type_struct_item_ref_obj" ".*"
    "type_union"          ".*"
    "type_union_item"     ".*"
    "type_union_item_obj" ".*"
    "type_union_item_array" ".*"
    "type_union_item_array_obj" ".*"
    "type_union_item_ptr" ".*"
    "type_union_item_ptr_obj" ".*"
    "type_union_item_ref" ".*"
    "type_union_item_ref_obj" ".*"
    "enum"                ".*"

```

```

"const_enum_item"      ".*"
"class"                ".*"
"class_attr"          ".*"
"class_attr_obj"      ".*"
"class_attr_array"    ".*"
"class_attr_array_obj" ".*"
"class_attr_ptr"      ".*"
"class_attr_ptr_obj"  ".*"
"class_attr_ref"      ".*"
"class_attr_ref_obj"  ".*"
"method"              ".*"
"namespace"           ".*"
"function"            ".*"
"const"               "[A-Z0-9_]*"
"const_obj"           "[A-Z0-9_]*"
"const_array"         "[A-Z0-9_]*"
"const_array_obj"     "[A-Z0-9_]*"
"const_ptr"           "[A-Z0-9_]*"
"const_ptr_obj"       "[A-Z0-9_]*"
"const_ref"           "[A-Z0-9_]*"
"const_ref_obj"       "[A-Z0-9_]*"
"var_stat"            ".*"
"var_stat_obj"        ".*"
"var_stat_array"      ".*"
"var_stat_array_obj"  ".*"
"var_stat_ptr"        ".*"
"var_stat_ptr_obj"    ".*"
"var_stat_ref"        ".*"
"var_stat_ref_obj"    ".*"
"var_glob"            ".*"
"var_glob_obj"        ".*"
"var_glob_array"      ".*"
"var_glob_array_obj"  ".*"
"var_glob_ptr"        ".*"
"var_glob_ptr_obj"    ".*"
"var_glob_ref"        ".*"
"var_glob_ref_obj"    ".*"
"var_auto"            ".*"
"var_auto_obj"        ".*"
"var_auto_array"      ".*"
"var_auto_array_obj"  ".*"
"var_auto_ptr"        ".*"
"var_auto_ptr_obj"    ".*"
"var_auto_ref"        ".*"
"var_auto_ref_obj"    ".*"
"macro"               "[^a-z]*"
"macro_const"         "[^a-z]*"
"macro_flag"          "[^a-z]*"
"macro_func"          "[^a-z]*"
"parameter"           ".*"
"parameter_obj"       ".*"
"parameter_array"     ".*"
"parameter_array_obj" ".*"
"parameter_ptr"       ".*"
"parameter_ptr_obj"   ".*"
"parameter_ref"       ".*"
"parameter_ref_obj"   ".*"
END LIST END STANDARD

```

For the class attributes to begin with "m\_", the class pointer attributes to begin with "m\_p", the constants and the macros to have no lower case letter and no underscore at the beginning and the end, the global variables to begin with "g\_", the global pointer variables to begin with "g\_p" and all other identifiers not to begin or end with an underscore:

```

STANDARD identfmt ON
LIST "any"                "[^_](.*[^_])?$"
    "class_attr"         "m_.*[^_]"
    "class_attr_ptr"    "m_p.*[^_]"
    "const"              "[A-Z0-9]([A-Z0-9_]*[A-Z0-9])?$"
    "var_glob"          "g_.*[^_]"
    "var_glob_ptr"      "g_p.*[^_]"
    "macro"              "[A-Z0-9]([A-Z0-9_]*[A-Z0-9])?$"
END LIST END STANDARD

```

## identl Identifier Length

The possible categories of identifiers are the same as for the **identfmt** rule (see in [Paragraph , \*identfmt Identifier Format\*](#)).

By default, the methods and functions must have between 4 and 25 characters, the types, variables, constants, macros and classes between 5 and 25, and the other identifiers between 1 and 25:

```

STANDARD identl ON
LIST "any"                MINMAX 1 25
    "type"                MINMAX 5 25
    "type_ptr"            MINMAX 5 25
    "variable"            MINMAX 5 25
    "variable_ptr"        MINMAX 5 25
    "type_func"           MINMAX 5 25
    "type_struct"         MINMAX 5 25
    "type_struct_item"    MINMAX 5 25
    "type_union"          MINMAX 5 25
    "type_union_item"     MINMAX 5 25
    "enum"                MINMAX 5 25
    "const_enum_item"     MINMAX 5 25
    "class"               MINMAX 5 25
    "class_attr"          MINMAX 5 25
    "class_attr_ptr"      MINMAX 5 25
    "method"              MINMAX 4 25
    "namespace"           MINMAX 5 25
    "function"            MINMAX 4 25
    "const"               MINMAX 5 25
    "const_ptr"           MINMAX 5 25
    "var_stat"            MINMAX 1 25
    "var_stat_ptr"        MINMAX 1 25
    "var_glob"            MINMAX 5 25
    "var_glob_ptr"        MINMAX 5 25
    "var_auto"            MINMAX 1 25
    "var_auto_ptr"        MINMAX 1 25
    "macro"               MINMAX 5 25
END LIST END STANDARD

```

## identres Reserved Identifiers

By default, there are no reserved identifiers:

```
STANDARD identres ON LIST END LIST END STANDARD
```

To forbid the identifiers "true" and "false":

```
STANDARD identres ON LIST "true" "false" END LIST END STANDARD
```

## incltype Included Modules Type

By default, **HEADER** modules can be included in **HEADER** and **CODE** modules:

```
STANDARD incltype ON
LIST "HEADER"          "HEADER" END LIST
```

```
LIST "CODE" "HEADER" END LIST
END STANDARD
```

To also allow **CODE** modules to be included in **CODE** modules:

```
STANDARD incltype ON
LIST "HEADER" "HEADER" END LIST
LIST "CODE" "HEADER" "CODE" END LIST
END STANDARD
```

For the signification of the **CODE**, **HEADER**, ... types, see in Paragraph , *cmclass A Single Class per Code File*).

## inldef Inline Functions Declaration and Definition

By default, the "private" parameter is not active:

```
STANDARD inldef ON END STANDARD
```

To indicate that private inline functions must be defined in the class definition file (.cpp file) and other inline functions in the class declaration file (.h file):

```
STANDARD inldef ON "private" END STANDARD
```

## macrocharset Characters Used in Macros

By default, no characters are forbidden in macros:

```
STANDARD macrocharset ON LIST "constant" "" "function" "" END LIST END
STANDARD
```

To forbid the characters `@#!&/[]{}~`'`` in macro-constants and `#@%.\`` in macro-functions:

```
STANDARD macrocharset ON LIST "constant" "@#!&/[]{}~`'" "function"
"#@%.\`" END LIST END STANDARD
```

## mconst Macro Constant Usage

By default, the **var** option is selected:

```
STANDARD mconst ON "var" END STANDARD
```

To have the **const** option instead:

```
STANDARD mconst ON "const" END STANDARD
```

To have the **nodefine** option instead:

```
STANDARD mconst ON "nodefine" END STANDARD
```

## mname File Names

By default, the part of the class name taken into account is between the characters 1 and 5:

```
STANDARD mname ON MINMAX 1 5 END STANDARD
```

To have instead the characters 4 and 10:

```
STANDARD mname ON MINMAX 4 10 END STANDARD
```

## nopreproc No Pre-processing Instructions

By default, only `#line` and `#` alone may not be used:

```
STANDARD nopreproc ON LIST "define" "include" "if" "pragma" "undef"
"error" END LIST END STANDARD
```

To allow only `#define`, `#line` and `#` alone:

```
STANDARD nopreproc ON LIST "define" "line" "none" END LIST END STANDARD
```

## **nostruct Keyword Struct Not Allowed**

By default, C-style structs are forbidden:

```
STANDARD nostruct ON END STANDARD
```

To allow C-style structs and then forbid C++-style structs (such as access specifiers: `private` for example, or methods):

```
STANDARD nostruct ON "cstruct" END STANDARD
```

## **operass Assignment Operator**

By default the `"dynalloc"` parameter is not put:

```
STANDARD operass ON END STANDARD
```

To look for the assignment operator only if there is a class member which is a pointer:

```
STANDARD operass ON "dynalloc" END STANDARD
```

## **parammode Parameters Mode**

By default, the three modes `"IN"`, `"OUT"` and `"INOUT"` are authorized:

```
STANDARD parammode ON LIST "OUT" "INOUT" "IN" END LIST END STANDARD
```

To authorize only the mode `"IN"`:

```
STANDARD parammode ON LIST "IN" END LIST END STANDARD
```

## **sectord "public", "private" and "protected" Sections Order**

By default no particular order is given:

```
STANDARD sectord ON LIST END LIST END STANDARD
```

To authorize the first section to be without any specifier, and then the specifiers to be in the order `private`, `protected` and `public`:

```
STANDARD sectord ON LIST "" "private" "protected" "public" END LIST END STANDARD
```

## **sgancstr Single Ancestor**

By default no ancestor is specified:

```
STANDARD sgancstr ON END STANDARD
```

To indicate a particular ancestor, name it:

```
STANDARD sgancstr ON "father" END STANDARD
```

## **swdef "default" within "switch"**

By default, the `default` case has to be the last one:

```
STANDARD swdef ON "last" END STANDARD
```

To have only a `default` case, whatever its position:

```
STANDARD swdef ON END STANDARD
```

## swend End of Cases in a "switch"

By default, an instruction `break`, `continue`, `goto`, `return` or `exit` is not mandatory for the last switch of a case:

```
STANDARD swend ON "nolast" END STANDARD
```

To impose such an instruction at the end of all the cases of a switch:

```
STANDARD swend ON END STANDARD
```

## typeres Reserved Types

By default, there are no reserved types:

```
STANDARD typeres ON  
LIST END LIST  
LIST END LIST END STANDARD
```

To forbid the types `int`, `char` and `register double` for variables and the types `unsigned int` and `double` for functions:

```
STANDARD typeres ON  
LIST "data" "int" "char" "register double" END LIST  
LIST "function" "unsigned int" "double" END LIST  
END STANDARD
```

## varstruct Struct and Union Variables

By default, the `nostruct` option is not selected:

```
STANDARD varstruct ON END STANDARD
```

To have the `nostruct` option:

```
STANDARD varstruct ON "nostruct" END STANDARD
```

# 6.3 Creating New Rules

New rules can also be created entirely using Tcl scripts.

More about this can be found in the dedicated *IBM Rational Logiscope - Adding Java, Ada and C++ scriptable rules, metrics and contexts* manual.



---

# Notices

I © Copyright 1985, 2009

U.S. Government Users Restricted Rights - Use, duplication, or disclosure restricted by GSA ADP Schedule Contract with IBM corp.

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send written license inquiries to:

IBM Director of Licensing  
IBM Corporation  
North Castle Drive  
Armonk, NY 10504-1785  
U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send written inquiries to:

IBM World Trade Asia Corporation  
Licensing  
2-31 Roppongi 3-chome, Minato-ku  
Tokyo 106-0032, Japan

**The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:** INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions. Therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

Intellectual Property Dept. for Rational Software  
IBM Corporation  
1 Rogers Street  
Cambridge, Massachusetts 02142  
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

## Trademarks

IBM, the IBM logo, ibm.com are trademarks or registered trademarks of International Business Machine Corp., registered in many jurisdictions worldwide. Other product and services names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at:

[www.ibm.com/legal/copytrade.html](http://www.ibm.com/legal/copytrade.html).

Adobe, the Adobe logo, Acrobat, the Acrobat logo, FrameMaker, and PostScript are trademarks of Adobe Systems Incorporated or its subsidiaries and may be registered in certain jurisdictions.

AIX and Informix are trademarks or registered trademarks of International Business Machines Corporation in the United States, other countries, or both.

HP and HP-UX are registered trademarks of Hewlett-Packard Corporation.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Macrovision and FLEXnet are registered trademarks or trademarks of Macrovision Corporation.

Microsoft, Windows, Windows 2003, Windows XP, Windows Vista and/or other Microsoft products referenced herein are either trademarks or registered trademarks of Microsoft Corporation.

Netscape and Netscape Enterprise Server are registered trademarks of Netscape Communications Corporation in the United States and other countries.

Sun, Sun Microsystems, Solaris, and Java are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries.

Pentium is a trademark of Intel Corporation.

ITIL is a registered trademark, and a registered community trademark of the Office of Government Commerce, and is registered in the U.S Patent and Trademark Office.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product or service names may be trademarks or service marks of others.

