SCOS-2000
Technical Note

C++ Naming and Coding Conventions
C++ Naming and Coding Convention

<table>
<thead>
<tr>
<th>Action</th>
<th>Name</th>
<th>Date</th>
<th>Signature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prepared by</td>
<td>Dan Søren Nielsen (Terma GmbH)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Verified by</td>
<td>Colin Haddow (OPS-GI)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Eduardo Gómez (OPS-GIC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Approved by</td>
<td>Eduardo Gómez (OPS-GIC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Authorised by</td>
<td>Mauro Pecchioli (H/OPS-GIC)</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Néstor Peccia (H/OPS-GI)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Distribution list

- ESA/TOS-GIC
- SCOS-2000 Industry Team
- SCOS-2000 Document Library

© COPYRIGHT EUROPEAN SPACE AGENCY, 2003
The copyright of this document is vested in European Space Agency. This document may only be reproduced in whole or in part, stored in a retrieval system, transmitted in any form, or by any means electronic, mechanical, photocopying, or otherwise, with the prior permission of the owners.
## DOCUMENT CHANGE LOG

<table>
<thead>
<tr>
<th>Issue/Revision</th>
<th>Date</th>
<th>Modification Nr</th>
<th>Modified pages</th>
<th>Observations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 rev 1</td>
<td>04.11.03</td>
<td>1</td>
<td>2</td>
<td>Colin Haddow added as verifier.</td>
</tr>
<tr>
<td>1 rev 1</td>
<td>04.11.03</td>
<td>2</td>
<td>6</td>
<td>Typos in first section</td>
</tr>
<tr>
<td>1 rev 1</td>
<td>04.11.03</td>
<td>3</td>
<td>6</td>
<td>References updated</td>
</tr>
<tr>
<td>1 rev 1</td>
<td>04.11.03</td>
<td>4</td>
<td>38</td>
<td>“ADD” replaced by text</td>
</tr>
<tr>
<td>1 rev 1</td>
<td>04.11.03</td>
<td>5</td>
<td>21</td>
<td>PC25 added to reflect RID85, following rules numbers consequently updated</td>
</tr>
<tr>
<td>1 rev 2</td>
<td>20.04.05</td>
<td>6</td>
<td>several</td>
<td>Some rules de-scoped, modified and added. Minor corrections (e.g. numbering)</td>
</tr>
</tbody>
</table>
## PAGE ISSUE RECORD

Issue of this document comprises the following pages at the issue shown

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Ref.: S2K-MCS-TN-9003-TOS-GIC
Issue: Issue 1.2
Date: 04.08.2005
Page: 4
Table of Contents

1 INTRODUCTION ............................................................................................................................................. 6
1.1 PURPOSE ...................................................................................................................................................... 6
1.2 SCOPE ......................................................................................................................................................... 6

2 REFERENCES .................................................................................................................................................. 6
2.1 APPLICABLE DOCUMENTS .......................................................................................................................... 6
2.2 REFERENCE DOCUMENTS .......................................................................................................................... 6

3 ABBREVIATIONS, TERMS AND DEFINITIONS .......................................................................................... 7
3.1 ABBREVIATIONS ......................................................................................................................................... 7
3.2 TERMS AND DEFINITIONS ........................................................................................................................ 7

4 C++ CONVENTIONS ..................................................................................................................................... 8
4.1 GENERAL C++ CONVENTIONS .................................................................................................................. 8
4.2 GENERAL LAYOUT OF CODE ....................................................................................................................... 9
4.2.1 The Overall Software Structure ........................................................................................................... 10
4.3 PROGRAM MODULARISATION .................................................................................................................. 13
4.4 PROGRAMMING CONSTRUCTS ................................................................................................................... 14
4.4.1 Declarations ........................................................................................................................................... 14
4.4.2 Declaration of variables ......................................................................................................................... 15
4.4.3 Declaration of Types .............................................................................................................................. 16
4.4.4 Exceptions ............................................................................................................................................. 16
4.4.5 Expressions ........................................................................................................................................... 18
4.4.6 Memory Allocation ............................................................................................................................... 19
4.4.7 Error Handling ..................................................................................................................................... 20
4.4.8 Portability ............................................................................................................................................... 21
4.4.9 Conditional Statements Layout ............................................................................................................ 24
4.4.10 Class ..................................................................................................................................................... 24
4.4.11 Generic Classes ................................................................................................................................... 33
4.4.12 Type conversion ................................................................................................................................... 34
4.4.13 Records/Structures .............................................................................................................................. 36
4.4.14 Specific OO facilities .......................................................................................................................... 37
4.5 NAMING CONVENTION .............................................................................................................................. 37
4.5.1 General Naming Convention ............................................................................................................... 37
4.5.2 Type and Class Names ........................................................................................................................... 38
4.5.3 Class member variable names ............................................................................................................... 38
4.5.4 Enumerated Types ............................................................................................................................... 39

APPENDIX 1 HEADER AND IMPLEMENTATION FILE OUTLINES ................................................................. 40
1 INTRODUCTION

1.1 PURPOSE

The purpose of this document is to define a dedicated C++ coding and naming convention.

1.2 SCOPE

The Scope of this technical note is to provide the dedicated coding and naming conventions for development projects.

The baseline for the work has been the [C++] combined with the established C++ convention [S2KC] and the language independent guideline [TN02]. This document is [TN02] instantiated at the C++ language.

For certain rules it has been necessary to expand the generic rules into more dedicated rules in these cases the ID have been extended with a small letter index at the end, such as NC.7.C++.a, and NC.7.C++.b. The used ID’s refer to the ID from [TN02].

For rule and guidelines where no additional information is added to the text from the generic conventions these are not replicated in this document. Hence the reader must look into the generic conventions [TN02] for further details.

2 REFERENCES

2.1 APPLICABLE DOCUMENTS

The documents, which are applicable for this document, are:

<table>
<thead>
<tr>
<th>Doc.</th>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>[C++]</td>
<td>BSSC(2000)1 Issue 1, 30 March 2000</td>
<td>C and C++ Coding Standards</td>
</tr>
</tbody>
</table>

2.2 REFERENCE DOCUMENTS

The documents, except for the applicable documents, which are referenced in this document, are:

<table>
<thead>
<tr>
<th>Doc.</th>
<th>Reference</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>[S2KC]</td>
<td>SCOS2K-CON-004, March 1999</td>
<td>C++ Conventions</td>
</tr>
</tbody>
</table>
3 ABBREVIATIONS, TERMS AND DEFINITIONS

3.1 ABBREVIATIONS

Abbreviations used in this document:

- SPR: Software Problem Report
- w.r.t: With respect to

3.2 TERMS AND DEFINITIONS

Terms and definitions specific for this document:

None
4 C++ CONVENTIONS

The C++ convention is split into two parts

- Coding convention
- Naming convention

The Coding convention will also address general topics such as file header and other general measures for increasing the quality of the produced code.

The document is basically a large set of rules, which have been, organised into categories in order to ease search for relevant rules. The structure is as follows:

- First major section is section 4.1 outlining the general conventions for C++ programming for projects under TOS-GI and TOS-GD.
- Second major section concerns the general code layout, covering subsystem, libraries, modules and files.
- Third section is on program modularisation
- Fourth key section is on programming constructs covering most aspects of the detailed programming approach such as declarations, expressions, error handling, portability etc.
- Fifth higher level section outlines the naming convention to be used

Some rules are given only as recommendations. They are always preceded by “[R]”

4.1 GENERAL C++ CONVENTIONS

This section covers the general project and coding conventions concerning the C++ language instantiation of the generic coding and naming guidelines [TN02]

GC.1.C++ Never break a rule without documenting it.

It is always reasonable to point out why deviations are made. Small rationales make it easier to remember the reason for deviating later on.

GC.2.C++ Maintain the source code and associated files under a configuration control system and document CM information in the files.

This rule is addressed further in the file header layout defined in rule GL.12.C++.

GC.3.C++ Use a "Makefile" or its equivalent for building the application

The use of scripts for building ensures reproducibility.

GC.4.C++ Write the software to conform to the coding language international standards.

E.g. the SCOS-2000 C++ code shall be written to comply with the ANSI C++ standard.

GC.5.C++ Do not rely on compiler specific features.

Whenever it possible and known it shall be avoided to use compiler specific features in order to ensure compiler portability of the application.

GC.7.C++ Use independent tools to provide additional warnings and information about the code.

Tools shall be used for checking the C++ code for memory leakages. Possibly tools can be added for checking that the correct coding style is used and metrics are within limits.
GC.8.C++ Always identify the source of warnings and correct the code to remove them.

GC.9.C++ Do not attempt to optimize the code until it is proved to be necessary.

The early optimization might result in unreadable code, which in turn is difficult to maintain.

GC.10.C++ Provide meaningful, saying comments in the source code and make sure that they are kept up to date.

Source code is generally the only component of the overall software project, which is guaranteed to be up to date. Therefore the programmer should provide comments in the source code so that maintenance programmers will be able to understand the code without needing to refer to external documentation. Comment can and should be a substantial part of the source code. It is, in common metrics, expected that comments lines constitute between 30 – 50% of the total number of lines in a given source code file.

GC.11.C++ Minimize any debugging code.

Even debugging code can be source of bugs. Hence, the less the better.

GC.12.C++ Use appropriate tools to ensure that the code conforms to the rules and to catch potential problems as early as possible.

More and more of the current CASE tools such as TogetherSoft and Clearcase provides functionality for controlling the code quality.

GC.13.C++ Write the comments in the common language of the project.

E.g. for SCOS-2000 the common language is English, and all comments are to be provided in this language.

GC.14.C++ Avoid "fancy-layout" comments because they require time and effort to maintain.

Only plain comments are allowed.

GC.14.C++.a Comments must add to the code, not detract from it.

GC.14.C++.b The body of the code shall not contain comments that refer to SPRs, NCRs or SCRs

SPR references in the body of the code add little value to the understandability and are difficult to maintain. They usually lead to confusion when several SPRs touch the same code.

GC.15.C++ Comments should never be used for "commenting out" code.

4.2 GENERAL LAYOUT OF CODE

In the same way that comments affect other more specific areas of the guidelines, code layout is also entwined with other areas. This section provides some guidelines on code formatting and should provide the reader with a feel for the overall importance of the way that code is laid out. The use of white space to provide additional clues of the block structure via consistent indentation is presented, as well as the use of vertical white space. The use of opening and closing braces on lines of their own in a consistent way across all language constructs helps keep the structure of the code clear.

The following rules relating to layout shall be applied to all code unless:

- the code has been produced by a code-generator phase of another tool;
- the code has been provided by a third-party;
- the code is to be integrated with other tools for which a particular layout is necessary or useful
In relation to the first bullet, it shall be pointed out that for many code generators means are available for controlling the layout of generated code.

GL.1.C++ Each project shall define the maximum length in characters of the source code lines (e.g. Each line of source code shall be no more than 80 characters in length).

In this age of graphics workstations and resizable windows this may seem like an old-fashioned restriction harking back to dumb terminals, which were limited to 24 rows of 80 characters. This recommendation should be applied so that code can be printed or included in documentation without losing information or the user having to reformat it.

Note that most lines of source code included in this document are less than 65 characters in length even though a smaller font is used than for the ordinary text.

GL.1.C++.a The identifier of methods and functions must have between 4 and 25 characters, the identifier of types, variables, constants, macros and classes between 5 and 25, and any other identifiers between 1 and 25.

GL.1.C++.b There must not be more than one statement per line.

GL.2.C++ Each nested block of code, including one-line blocks, shall be identifiable through the code layout.

Each block shall be indented with the project specific indentation

GL.2.C++.a All aspects of indentation and formatting shall be consistent within a project.

GL.3.C++ Each project shall define the code block style to be used.

Each block is surrounded by indented brackets placed on their own line.

GL.4.C++ Each project shall define its own indentation rules (e.g. The contents of each nested block shall be indented by 3 spaces compared to the token which delimit the block).

GL.5.C++ Tabs are not allowed.

4.2.1 THE OVERALL SOFTWARE STRUCTURE

4.2.1.1 Subsystems

A subsystem consists of the source code and associated files, which are used to implement a free-standing part of the overall software system. A subsystem may use libraries and modules, which are also used in other subsystems.

4.2.1.2 Libraries

A library consists of a collection of modules, which define a set of inter-working abstract data types (ADTs) and associated functions. For software written using C++, a library is also likely to contain classes. It should be pointed out that the definitions of functions outside classes are discouraged as these are of global nature.

GL.6.C++ Use separate folders for each of the subsystems and libraries defined during the design phase.
For the standard libraries the some functions are provided, which are not allowed to be used in an ANSI C++ program.

GL.7.C++.a Never use goto, longjmp(), setjmp(), malloc(), free(), realloc()

Deviation from this rule is allowed in relation with the use of C-libraries where call to e.g. free() might be necessary.

[R]GL.7.C++.b 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.

GL.7.C++.c The usage of macro constants shall be limited

GL.7.C++.d Use inline functions instead of macro-functions

GL.7.C++.e The keyword union is not allowed

GL.7.C++.f Use the ptr->fld syntax instead of the (*ptr).fld syntax.

GL.7.C++.g The ternary conditional operator ? ... : ... must not be used.

4.2.1.3 Modules

A module is a conceptual unit, which consists of a set of files which together make up the interface and implementation of a particular abstract data type (ADT) or class.

GL.8.C++ Use a template to provide the starting point for all files within each particular subsystem, library or module.

It is recommended to use scripts for generation of the .H and .C files.

4.2.1.4 Files

GL.9.C++ Source code shall be separated into an interface and an implementation file

The include files or declaration files shall always have the extension “.H”. The implementation file shall have “.C” a extension. For each Class or template there are 2 distinct files:

- one interface definition file SYSmyClass.H, where the class is defined.
• one implementation file **SYSmyClass.C** containing at least the standard methods (destructor, constructors, operators).

File name must be unique over the whole project.

In case of writing code with relatively large functions, these shall be placed at the end of the file.

**GL.10.C++** The interface file must be the first included in its own corresponding implementation file.

This is actually the case with all current implementation.

**GL.11.C++** Each file shall contain a standard comment header block.

File header in both .C and .H files, shall have the following layout:

---

**Figure 2** SCOS-2000 file header

```
// (C) 2000 European Space Agency
// European Space Operations Centre
// Darmstadt, Germany
//
// System       : SCOS-2000 - the Satellite Control and Operations System
// Sub-System   : CMDA
// File Name    : CMDAfactory.H
// Author       : akowalc, hpederse
// Creation Date: 11-09-1998
// Description  : Implements factory design pattern for commanding
//                 applications subsystem (CMDA)
// LCC SPR programmer date
```

---

**GL.12.C++** Each file header block shall contain information about the functionality of the code, the author, the creation date, the system and subsystem where the code belongs and on the configuration management and tracking of changes (references, author of the changes and date)

Please refer to the example above where it is demonstrated that the following information shall be available in the file header:

• System name
• Sub-system name
• File name
• Author(s)
• Creation date
• Description of code functionality
• and possibly a section for configuration control information listing LCC (Library Control Change) number, SPR number, name of correcting person, and finally the date of changes being put back into the library.

The latter information is recommendable although redundant to the information available in the code library (CVS) where this information also shall be provided at time of put-back.

---

i. For templates, this must be the only implementation file, and must be in the include directory.
GL.13.C++ The public interface file shall be self-contained and self-consistent.

Definitions of a given class shall be fully done within the declaration file (.H file), and is not allowed to be distributed over more files.

GL.15.C++ The interface file shall contain declarations only.

The only exception to this rule is inline functions.

4.2.1.5 Metrics

Metrics can be used as another coding guideline, as it specifies the limits on size and complexity of the code written. Many metrics exist and all have their relevance in different perspectives. To keep it simple only a small set of simple metrics have been selected for this coding conventions, each project may define additional metrics to meet specific targets such eg. for certification purposes.

The table below gives a set of metrics and their recommended values:

<table>
<thead>
<tr>
<th>Metric/criticality</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>LoC per Module</td>
<td>100</td>
<td>200</td>
<td>300</td>
<td>400</td>
<td>500</td>
</tr>
<tr>
<td>Code comment freq.</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>30</td>
<td>20</td>
</tr>
<tr>
<td>Cyclomatic complexity</td>
<td>10</td>
<td>12</td>
<td>15</td>
<td>20</td>
<td>20</td>
</tr>
</tbody>
</table>

Where LoC per Module is the total number of lines minus the number of comment lines.

The code comment frequency is given in percentage and is the ratio between source statements and comment blocks.

Cyclomatic complexity is the McCabe complexity measure for each module, and are to be considered as max values.

4.3 PROGRAM MODULARISATION

In order to increase maintainability of a given software, it must be modularised in accordance with functional decomposition. Which means that the program code shall be subdivided into well defined functional areas, each as isolated as possible. This will ease the development as well as the long-term maintenance. Obviously the breakdown and organisation of the code is done during the architectural design phase of the project.

Each of these modules shall be organised properly around a project wide file naming convention. This convention must provide indication of module as well as functionality of a given file.

PM.1.C++ Each filename shall provide indication of modular relationship as well as functionality implemented.

As an example file naming convention for SCOS-2000 is as follows: First three to five letters are capital letters indicates the module or subsystem, followed by an abbreviated indication of what is implemented in the actual file. Some examples are given here:

- MSGhandler.C
- CMDseqElemDef.C
- PIFserverClass.C

When programming in C++ the best practice of class to file modularization shall be followed:
**PM.1.C++.a** For each class the implementation shall be distributed in two files, one for the interface specification and one for the actual implementation respectively with the .H and .C extensions.

Obviously the file names for the two files shall be the class name however without the class identifier (_c, classe or c).

In order to ensure easy navigation in the source code there must be a relation between the file name and the actual module or subsystem implemented.

**PM.1.C++.b** The first 3-5 letters of the filename shall indicate to which module the code belongs.

**PM.2.C++.a** In general, the use of namespaces should be avoided. However, if there is a potential for conflict (e.g. a third party product that would duplicate existing class names is integrated) namespaces shall be used to differentiate.

### 4.4 Programming Constructs

#### 4.4.1 Declarations

**PC.1.C++** Each declaration should start on its own line and have an explanatory comment.

Example given below

```cpp
// default constructor
MSGmsg();
```

```cpp
// copy constructor
MSGmsg(const MSGmsg&);
```

```cpp
// assignment operator
const MSGmsg& operator=(const MSGmsg&);
```

**PC.1.C++.a** Declarations must appear at the beginning of blocks

**PC.2.C++** Entities should be declared to have the shortest lifetime or most limited scope that is reasonable.

**PC.3.C++** Global entities must be declared in the interface file for the module.

**PC.4.C++** Declarations of "extern" variables and functions may only appear in interface files.

The above two rules are “exceptions”, as it generally is forbidden to use this way of programming. They should only be used when absolutely need, and in the way outlined by these rules.

**PC.5.C++.a** Declarations of static or variables and functions may only appear in classes.

**PC.5.C++.b** All global variables and constants shall be explicitly declared static.

And again this is only to be done if no other possibility solution can be found. All variables shall be declared inside classes.

```cpp
static const int MAX_BUFFER_LEN = 1024;
```

This prevents the possibility of conflicts with anything else with the same name in a different file.
PC.6. C++ Declarations should appear in predefined order (e.g. constants and macros; types, structs and classes; variables; and functions).

PC.7. C++ Symbolic constants shall be used in the code. "Magic" numbers and strings are expressly forbidden.

The use of magic numbers simply makes the debugging and maintenance work more difficult.

PC.7.C++.a Never use numbers in code, nor any 'hardcoded' string, except when the use of these numbers is obvious - for instance in mathematical expressions, loop counter initialisation and limit checking.

PC.8. C++ All symbolic constants shall be declared using an enumeration technique or the const keyword.

In general for the use of symbolic constants with wide scopes shall be avoided, if there are intended for local use.

4.4.2 DECLARATION OF VARIABLES

In the spirit of C++ all variables shall be declared inside classes, and the use of global or external variables shall only be used when no other possibilities are available.

PC.9. C++ Each variable shall have its own declaration, on its own line.

Each declaration shall contain a comment explaining the purpose of the declared variable. Examples of declarations in SCOS-2000:

```c++
// Total parameter length
unsigned short m_paramLen;

// Packet parameter slots. One for each packet parameter
d_vector<CMDpktSlotDef*> m_pktSlots;

// Packet definition unique ID
STLname<8> m_pktID;

PC.10. C++ All local variables shall be initialised in their declaration.

This avoids the possibility of using uninitialised values.

One good way of doing this is only to declare the variable at the point of first use, e.g.

```c++
if (i < length)
{
    Object * ptrI = array[i];
    // do stuff with ptrI
}

PC.10.C++.a Avoid global data if at all possible. Static class data provides a much better alternative.

PC.10.C++.b No direct access to neither public nor private class variables are allowed.
Methods like get and set must be defined and used for accessing member variables.

PC.10.C++.c Loop counters (in for loops) are to be initialized in the initialization statement within the loop.

PC.10.C++.d Global variables must be initialized when they are defined.

4.4.3 DECLARATION OF TYPES

Programmers should be encouraged to use specifically defined types and to create type names, which are meaningful in the problem domain. This promotes readability because it is immediately obvious what the "dimensions" of a variable should be.

An additional benefit to the readability is the extended use of the compilers type checking facilities.

4.4.4 EXCEPTIONS

Exceptions are a very useful tool for trapping not expected program circumstances or handling of erroneous program behaviour.

PC.11.C++.a A function which can issue exceptions (i.e. has a "throw") clause in its declaration) can only be called:
1. from within a "try-catch" construct catching all those exceptions;
2. from within a "try-catch" construct catching some of those exceptions where the other are declared in the "throw" clause of the function containing the "try-catch" construct;
3. from within another function which exports (via the "throw" clause int its declaration) exactly the same exceptions.

This rule forces to make very well clear in the code (and in the design) how exceptions flow from a portion of code to another. Example.

class ExBase
{
    private:
        string name;

    public:
        ExBase (string nm) { name = nm; }
        string name() { return name; }
        void announce() {
            cout << name() << "!" << endl;
        }
};

class Ex1553 : public ExBase
{
    public:
        Ex1553() : ExBase("1553 exception") {
            announce();
        }
};
class ExRs422 : public ExBase
{
public:
    ExRs422 : ExBase("RS422 exception");
    announce();
};

// Incorrect definition of Foo
void Foo()

    // …
    throw Ex1553();

    // …
    throw ExRs422();

    // …

// Correct definition of Foo
void Foo() throw (Ex1553, ExRs422)

    // …
    throw Ex1553();

    // …
    throw ExRs422();

    // …

// Incorrect use of Foo (Foo is called without controlling the exceptions
// it may throw
void Goo()

    // …
    Foo();

    // …

// Two correct ways of calling Foo
// All exceptions generated in Foo are caught in Goo
void Goo() { try
    // …
    Foo();
    } catch (Ex1553)
    { cout << "caught Ex1553" << endl;
    } catch (ExRs422)
    { cout << "caught ExRs422" << endl;
    }
catch (ExBase eb) {
    cout << "caught " << eb.name() << endl;
}
// ...

// Goo cannot catch ExRs422 (it does not know what to do with it)
void Goo() throw (ExRs422) {
    // ...
    try {
        Foo();
    }
    catch (Ex1553) {
        cout << "caught Ex1553" << endl;
    }
    // ...
}

[R]PC.11.C++.b Every C++ program using exceptions must use the function set_unexpected() to specify which user defined function must be called in case a function throws an exception not listed in its exception specification.

[R]PC.11.C++.c Every C++ program using exceptions must use the function set_terminate() to specify which user defined function must be called if an handler for an exception cannot be found.

PC.11.C++.d Catch exceptions by reference

If they are passed by pointer it will not be clear who is responsible for deleting the allocated memory.

4.4.5  EXPRESSIONS

A general rule when creating expressions are

PC.13.C++.a The programmer must not override the comma, &&, || and ?: operators.

In C++ the programmer has the opportunity of overriding the comma, logical-AND, logical-OR and the conditional operators. This is not recommended. Not only would it be hard for a programmer to overload these operators to have a similar intuitive meaning. This means that the code using these new operators is unlikely to work in the same way as the rest of the code.

PC.13.C++.b Assignment operators (+=, -=, *=, /=, %=, >>=, <<=, &=, |=, ^=, ++, --) must not be used more than once in each statement (including declarations).

4.4.5.1 Conditional Expressions

PC.13.C++ Conditional expressions must always compare against an explicit value with Boolean expressions as an exception.

Furthermore the used expressions are not allow to use abbreviated notations such as (!x), the full expression such as (!x==0), must be written.
4.4.5.2 Order of Evaluation

PC.14.C++ The programmer shall make sure that the order of evaluation of the expression is defined by typing in the appropriate syntax, by using parenthesis.

4.4.5.3 Use of Parentheses

PC.15.C++ The programmer must use parentheses to make intentions clear, when it is needed for improving readability.

The following example gives an indication of the intention of this rule:

(a * b) + square((c - d))

although the operator preceding would calculate the result correctly it is still easier for the human to scan and understand the logical composition.

PC.16.C++ The programmer must always use parentheses around bitwise operators.

4.4.5.4 Use of White Space

PC.17.C++.a Do not use spaces around the "." and "->" operators or between a unary operators and their operands.

As the “.”, “->” and unary operators are ‘non’ algorithmic operators it is important that their position differs from the ‘normal’ operational operators which are surrounded by whitespace.

PC.17.C++.b Other operators should be surrounded by white space.

Write a * b rather than a*b.

4.4.6 MEMORY ALLOCATION

PC.18.C++.a Allocation using shall use new/delete.

The simple reason for this rule is that new and delete are part of the C++ environment and know about constructors and destructors. Objects, which are allocated using new, can be constructed and initialised automatically. Similarly the destructor for the object is called when delete is used.

PC.18.C++.b If the call to new uses [] then the corresponding call to delete must also use [].

If an object is allocated using new the pointer to the object provides the system with the indirect means of discovering information about the memory needed for that object. This information is used by delete when returning the memory to the system. A similar process is used for an array of objects. If an array of objects is allocated the programmer must use new with [] in order to tell the system about the size of an individual object and also information about the array of objects. The ordinary delete function (i.e. without []) only knows how to handle a single object and not an array of objects. The delete function using [] does know how to handle an array of objects and is able to return the memory for the whole array back to the system. To avoid problems with memory leaks or corruption, matching pairs of new/delete or new[]/delete[] should be used when allocating and de-allocating each area of memory.

It should here be emphasized, one more time that the use of the C memory allocation functions such as malloc(), realloc() etc. are not allowed.

PC.18.C++.c After calling delete set the pointer to NULL (or 0)
4.4.7 ERROR HANDLING

PC.18.C++ EMPTY

PC.19.C++ The programmer should validate function parameters where possible.

DESCOPED PC.20.C++ The return values of functions should be checked for errors.

PC.21.C++ Diagnostic code should be added to all areas of code which "should never be executed".

PC.22.C++ Error messages are not allowed to be hard coded, but shall be handled through some sort of central error message definition.

In order to create flexible and error free error messaging their definition shall be handled centrally, possibly on module basis if appropriate, else on application level. The definitions shall be done in such a way that the message itself is separated from the error location, but identified through encoding of the error id.

PC.22.C++.a When fallible functions fail, they shall indicate that they have failed by returning an error code or an out-of-bound value

Out-of-bound return values (such as -1 when returning an int) are generally discouraged. The main exception is that it is legitimate to return a null pointer to indicate failure
The error code should be either a class or sub-system specific enumeration, or a Boolean, with true indicating success and false indicating failure.

PC.22.C++.b Fallible functions should never return a reference when fallible functions fail, they shall indicate that they have failed by returning an error code or an out-of-bound value

There is no sensible value for them to return if they fail, as there is no such thing as a ‘NULL reference’.

PC.22.C++.c When fallible functions fail in ways that are fully described by their return value, they should not raise an error or warning.

Otherwise lots of errors and warnings would occur during normal system operations.

PC.22.C++.d When infallible functions fail, they should raise a fatal error.

Otherwise the system will be more difficult to test, as the consequence of the failure might not be immediately obvious.

PC.22.C++.e Whenever a fallible function is called, the return code shall always be checked.

This includes system and third-party library calls. Failure of a called function is treated in the same way as any other failure in the enclosing function - for example, if a fallible function is called from an infallible one, and fails, the infallible function should raise a fatal error.

Sometimes it is useful to have fallible and infallible variants of the same function - in this case the infallible one should just call the fallible one, and raise an error if it fails.

static String findFileSilent(String name,t_fileType type);
static String findFile (String name,t_fileType type)
{
  String foundFile = findFileSilent(name, type);

  if (foundFile.length() == 0)
  {
    MSG_SYS_FATAL(MISC_NO_FILE, “Cannot find file “ << name);
  }

  return foundFile;
}

PC.22.C++.f Calls to new should be assumed to succeed.

There is rarely anything that can be done at the call site if no more virtual memory is available. Instead, a new_handler callback should be installed, that takes the appropriate action when virtual memory is exhausted.

PC.22.C++.g C++ exceptions shall never be used without an appropriate design concept

Most existing code is not exception-safe. Consequently, any decision to use exceptions should be centrally coordinated, not introduced ad-hoc.

An exception for the above has to be made however, when using CORBA/omniORB, since CORBA/omniORB will raise exceptions. Appropriate try and catch blocks have to be placed around CORBA/omniORB calls.

4.4.8 PORTABILITY

PC.23.C++ The programmer should use "problem domain" types rather than implementation types.

The programmer is encouraged to add a level of data abstraction to the code so that variables are expressed in terms which relate directly to the problem domain rather than to the underlying "computer science" or hardware implementation.

typedef unsigned short AgeInYears_t;
typedef float KmPerHour_t

It shall be stressed that the type definition process shall be coordinated within a project to ensure that the same types are not reinvented.

PC.24.C++ Use non-portable code shall be minimized.

When necessary conditional compilation shall be used. It prohibited to conditional compilation for diagnostic code (use command line options).

4.4.8.1 Sizes

PC.24.C++.a The programmer may only assume range(char) < range(short) <= range(int) <= range(long).

ISO/IEC C standard defines a standard header file, limits.h, in which standard macro names are defined to denote the minimum and maximum values of the different integer types on that system.
The standard also provides the minimum range of values for a particular integer type, which is guaranteed to be portable between different implementations. Note that the exact range allowed for a variable of type char depends on whether the implementation uses a signed char or an unsigned char implementation.

**PC.24.C++.b** The programmer may only assume that range(float) \(\leq\) range(double) \(\leq\) range(long double)

ISO/IEC C Standard also defines a standard header file, float.h, in which standard macro names are defined to denote the minimum and maximum values of the different floating point types on that system. As for integer types, the default ANSI values provide the ranges which are guaranteed to be portable between implementations.

4.4.8.2 Representation

**PC.24.C++.c** The programmer may not assume knowledge of the representation of data types in memory, which implies that the use of memory dumps are forbidden.

Different systems may make use of different representations of data types in memory. Such differences may include differences in word ordering, byte ordering within a word. The programmer should therefore avoid any specific knowledge of an underlying representation when manipulating the data because what may be valid on one system may not hold true on another system.

However in cases where data is being encoded/decoded to a particular data format (e.g. Big Endian/Little Endian) this rule does not apply, and the programmer is responsible for conversion between representations.

**PC.24.C++.d** The programmer may not assume that different data types have equivalent representations in memory.

This rule follows from the previous section and the rule above. If the different types specify different ranges of values it is likely that they are represented differently in memory. The programmer may not assume that different types share a common representation in memory.

4.4.8.3 Pointers

**PC.24.C++.e** The programmer may not assume knowledge of how different data types are aligned in memory.

Different hardware platforms impose different restrictions on the alignment of data types within memory.

**PC.24.C++.f** The programmer may not assume that pointers to different data types are equivalent.

Some hardware platforms actually use different pointer representations for different data types. Therefore assigning one pointer value for one type to a pointer variable for another type is not always guaranteed.

The only exception is the equivalence of void* pointers to pointers of other types. The use of void pointers is generally not allowed in order to preserve maintainability and readability of the program.

**PC.24.C++.g** The programmer may not mix pointer and integer arithmetic.

Pointers are not integers. The programmer must not treat pointers as integers. The programmer is prohibited from assigning integers (or integer values) to pointers and vice versa. If we assume the following declarations:
```
int intValue;
Thing_t thingArray[10];
Thing_t *thingPointer = thingArray; // i.e. &thingArray[0];
intValue = thingPointer; // DO NOT DO THIS IN REAL CODE!
intValue += 1; // increase integer value by 1
thingPointer += 1; // adjust pointer value to point to
// next thing, i.e. &thingArray[1]
```

At this point it is unlikely that intValue and thingPointer contain the same value because intValue has been incremented by 1 whereas thingPointer has been adjusted by the size of a Thing_t object (including possible adjustments for the alignment of Thing_t objects in memory).

**PC.24.C++.h** It is not allow to use void pointers and pointers to functions.

4.4.8.4 Underflow and Overflow

**PC.24.C++.i** The programmer must use a wider type or unsigned values when testing for underflow or overflow.

This addresses the case when the programmer is dealing with two integer variables, with a relatively narrow possible range (such as chars and shorts), and is worried about overflow or underflow. Then it is possible to convert the values into variables with a wider range in order to detect whether overflow or underflow occurs during the calculation.

```
if ( (long)shortValue + (long)shortValue > SHRT_MAX) // overflow
```

This is not possible if the variables in question are of type `long` or have the same range as a `long`. However it is possible for the programmer to code around potential problems of overflow or underflow when dealing with two unsigned integer values because this can be detected without relying on the underlying hardware implementation.

The following practices should encouraged in order to deal with overflows/underflows

Provide flags inside conversion functions (or throw exceptions)

Use overflow safe operations within classes (or overload the relevant operator)

Make use of the division operator (or the right shift operator for unsigned integer) to test possible overflow e.g. something like:

```
if (((HugeFloat/10+AnotherHugeFloat/10 +1)> MAX_FLOAT/10)
    return ERROR;
else return OK;
```

4.4.8.5 Conversion

**PC.24.C++.j** The programmer must be careful when assigning "long" data values to "short" ones.

As mentioned in rule PC.24.C++.a-b the ranges of the different integer types and floating point types can be different. Therefore the programmer must ensure that a value does not lie outside the permitted range of a particular type before assigning the value to a variable of that type.
4.4.9 **CONDITIONAL STATEMENTS LAYOUT**

For case statements in order to ensure well-defined flow and status a default catch statement must be programmed.

```
switch (expression)
{
    case 0:
        statement(s);
        break;

    case 1:
        statement(s);

    // no break so FALLS THROUGH

    case 2:
        case 3:
            statement(s);
            break;

        default:
            statement(s);
            break;
}
```

**PC.25.C++.a Each case within a switch statement must contain a break statement or a "fall-through" comment.**

Where one case within a switch statement performs some pre-processing before falling through to the code of the following case this must be commented in the code. See case 1 in the example above.

Where a case shares all of its code with the following case this does not need to be commented because this is immediately apparent. See cases 2 and 3 in the example. All other cases, including the last, must have an explicit break statement. Explicit break statements and fall-through comments indicate those places in the code where it is safe, or unsafe, for a programmer to add an additional case statement without interfering with the existing code.

**PC.25.C++.b All switch statements shall have a default clause.**

All switch statements shall have a default case even if the programmer believes that the default case can never be reached. This allows for possible errors in the code, or corruption of data, to be detected more easily.

4.4.10 **CLASS**

**PC.26.C++ In Class declarations shall the declaration of public, protected and private data and functions be clearly separated and only one section for each type.**

In order to increase readability and maintainability the three types of declarations shall be well structured and separated. Furthermore each type shall only be declared once, to avoid cluttered up class definitions.
class CExample
{
    public:
    // Members

    // Methods

    protected:
    // Members

    // Methods

    private:
    // Members

    // Methods
};

PC.26.C++.a Any collection of data that does not warrant the work of writing a full class (e.g. defining accessor functions) should be defined as a struct.

Classes always have their own files according to rule PM1.C++.a, structs can be defined in the same header file as a class. This rule avoids the potential pitfall of having a multiplicity of small classes with no significant functionality.

PC.26.C++.b Classes for which it is not intended to instantiate any objects should be abstract - i.e. they should contain at least one pure virtual function.

This ensures it is not possible to create any objects of this class.

class abstractClass
{
    void pureVirtualFunction() = 0;
};

PC.26.C++.c Use of inheritance from non-abstract classes shall be minimised.

Do not use private inheritance for implementing a ‘has a’ relationship.

PC.26.C++.d Diamond-shaped inheritance hierarchies are not allowed.

class base {};
class derived1 : public base {};
class derived2 : public base {};
class diamondDerived : public derived1, public derived2 {};

Diamond-shaped inheritance hierarchies cause many special problems, such as defining the meaning of copying, resolving virtual function ambiguities, etc.

The way to avoid this problem is to use virtual classes which of course is possible if the design and implementation is in control, and own hands. However in case of a tool provider it most be expected
that virtual attribute is implemented by the tool provider, as this is common practice of solving this issue.

**PC.26.C++ Inline functions should be defined within the class declaration.**

Functions too large to conveniently fit within the class declaration (i.e. longer than 3 lines) should not normally be inlined. On the other hand, very simple, stereotypical functions such as accessors should normally be inlined, as this is both simpler and faster than the non-inline version.

class MyClass
{
    public:
        int getValue() const
        {
            return m_value;
        }
} // end of class definition

**PC.27.C++ Class member variables must not be declared "public".**

One of the frequently quoted benefits of Object Oriented systems is the ability to encapsulate data within an object and hide the implementation details behind a strictly enforced interface.

If the class member variables are hidden behind more widely available accessor functions then the underlying implementation of the class can be hidden from the user of the class. The implementation may change without affecting the interface itself. This means that other areas of code do not need to change simply because the internal details of a class change. If member variables were public, and the programmer used the directly elsewhere in the code then all this code must be modified and recompiled if the member variable change.

If the member variables are hidden, any interaction between the members variables can be strictly controlled within the class itself so they are guaranteed to be in a consistent state, thereby reducing the amount of checking code which is needed before their values are used elsewhere.

class CRange
{
    public:
        // ...
        void SetLimits(int lowerLimit, int upperLimit);
    private:
        int m_lowerLimit; // m_lowerLimit <= m_upperLimit
        int m_upperLimit;
    };  
    void CRange::SetLimits(
        int lowerLimit,
        int upperLimit
    )
    {
        MSG_ASSERT(lowerLimit <= upperLimit);
        if (lowerLimit <= upperLimit)
        {
            m_lowerLimit = lowerLimit;
            m_upperLimit = upperLimit;
        }
else
    {
        m_lowerLimit = upperLimit;
        m_upperLimit = lowerLimit;
    }
}

In the example above the programmer who uses this class does not have access to the individual member variables as it is obliged to use the `SetLimits()` function which guarantees that the limits are consistent.

**PC.28.C++ It shall be ensured that an object of a class is created in a controlled manner**

The "default" constructor, i.e. the one with no arguments, is provided if no other constructors are explicitly declared. As it is not specified how the compiler initializes the member attributes with its explicitly defined constructor, the programmer shall always provide a default constructor which initializes the member attributes correctly.

**PC.28.C++.a Avoid the use of global objects with constructors. The order in which global objects are initialised is not defined and can lead to ‘chicken and egg’ problems.**

Instead, use a pointer to the object, initialised when first used.

```cpp
static const MSGhandler &MSGhandler::instance()
{
    static MSGhandler * s_handler = 0;

    if (!s_handler)
    {
        s_handler = new MSGhandler();
    }

    return *s_handler;
}
```

**PC.29.C++ It shall be ensured that an object of a class is deleted in a controlled manner.**

The programmer should always declare a destructor in order to guarantee that memory is reclaimed correctly when an object is destroyed. Consider a class, which represents a name.

```cpp
class CName
{
    // …
    string m_name;
};
```

The member variable points to dynamic memory. Assume that the member variable is initialized to NULL in the default constructor and to a particular name by providing a constructor with a string argument:

```cpp
CName::CName()
{
    // default constructor
    m_name = NULL;
};
```
As a means for allocating dynamic memory has been provided, a means for returning that memory to the system when it is no longer required and thus avoiding a memory "leak", must be provided too. This is obviously a task for the destructor:

```c++
CName::~CName()
{
    delete[] m_name; // also handles m_name == NULL
}
```

**PC.30.C++.a** Base classes (capable of being derived) should have virtual destructors. But there is no need to have virtual destructor when the class may not be derived from.

**PC.31.C++** It shall be ensured that an object of a class is copied in a controlled manner.

Consider the following non-member function, which takes a CName object as parameter:

```c++
Bool_t IsNameInDataBase(CName name)
{
    // …
}
```

A copy constructor shall always be defined if an object has attributes.

**PC.31.C++.a** Stick to established conventions for overloaded operators.

Overloaded operators can make code much more difficult to understand, unless the reader is aware of their presence and definition. A few idioms, such as + for string concatenation, << and >> for stream operations, etc. have become common. Use these, but do not attempt to create new ones, unless you are sure the benefits will outweigh the costs.

**PC.31.C++.b** Any non-member, non-global function shall be explicitly declared static

For instance,

```c++
static void initialise();
```

will not conflict with any other function called initialise in another file.

**PC.31.C++.c** In a function declaration, the names of formal arguments shall be specified and should be meaningful. If the function definition uses the parameters, the names should match the declaration.
int SIGxyPlot::addPoint(int x, int y);

The only exception is when the type of the parameter provides enough information, e.g.

void addExecutionRequest (CMDexecRequest &);

**PC.32. C++ Member functions shall have a standard layout.**

An example of the layout is given below:

class CExample
{
    public:
        CExample(int x, int y);
    protected:
        private:
            int m_x;
            int m_y;
    }

CExample::CExample(
    int x,
    int y
) : m_x(x), m_y(y)
{
    // body of the constructor, if any
}

**PC.33. C++ Constructor functions which explicitly initialize any base class or member variable should not rely on a particular order of evaluation.**

When using an initialisation list as in the previous example, the programmer must not initialise a data member in terms of another data member, which has been previously initialised using a parameter. If the previous example is extended to include a constructor, which initialises both member variables to the same value, will look like this:

CExample::CExample(int z) : m_y(z), m_x(m_y)
{
    // body of the constructor, if any
}

This will not give the desired result `m_x` and `m_y` both being equal to `z` because the initialisation takes place in the order of declaration of the member variables. Therefore the `m_x(m_y)` initialisation is performed before the `m_y(z)` initialization resulting in `m_x` being equal to zero and `m_y` equal to `z` instead of both being equal to `z`. Generally this kind of initialisation shall be done in the constructor body if needed.

**PC.34. C++ Objects should be constructed and initialized immediately if possible rather than be assigned after construction.**

It is more efficient to create an object and initialise it immediately using a copy constructor than it is to construct and object and then assign to it later. This becomes more important as the class of the object becomes larger or more complex because the effort needed to construct one object only to tear it all down in order to assign the contents of another object may be significant.
CComplicatedClass newOne(oldOne);

// are preferable to
CComplicatedClass newOne;
newOne = oldOne;

ANSI compilers will anyway generate warnings if the initialisation is not performed immediately.

[R]PC.35.C++ The class should always declare an assignment operator.

Remember that the compiler-supplied functions provide “bitwise” operations on the objects. Consider the following fragment of code:

CName name("Fred Smith"); // name.m_name points to “Fred Smith”
void SomeFunction(void)
{
    // new scope
    CName copy; // copy.m_name points to NULL;
    copy = name; // copy.m_name points to “Fred Smith”
    // end of scope – copy destroyed
}

The variable “name” is constructed using the string provided and memory is allocated within the object for a copy of the string argument. The variable “copy” is constructed but it contains a pointer to NULL. In the absence of an explicit assignment operator when “name” is assigned to “copy” the compiler-supplied assigned operator is called. This makes a “bitwise” copy of “name” into “copy”. This means that both “name” and “copy” contain a pointer to the same memory area (with value “Fred Smith”). At the end of its scope, the destructor is called for the “copy” variable. The memory which pointed by “copy” (and therefore also by “name”) is returned to the system leaving “name” containing a “dangling” pointer, i.e. one which points to an undefined area of memory. What happens after this depends on the rest of the program, but it is unlikely to be what the programmer expects and could prove very difficult to debug. To avoid this problem the programmer needs to explicitly declare an assignment operator function for the class. An initial version of this might look like the example below (although this will be defined in a later rule).

CName & CName::operator=(const CName & sourceName)
{
    // return the current memory to the system
    delete[] m_name;
    // allocate memory for the copy
    int length = strlen(sourceName.m_name);
    m_name = new char[length + 1];

    // copy from the source to this
    strcpy(m_name, sourceName.m_name);

    return(*this);
}

PC.36.C++ The assignment operator(s) must check for assigning an object to itself.

The assignment operator given in the previous rule plugs one set of memory leaks compared to the compiler-supplied bitwise version when dealing with classes which contain pointers to dynamic
memory. However it introduces a potential problem with data loss if the programmer assigns an object to itself because the dynamic memory pointed to by the target object is returned to the system before the memory of the source is copied. If both source and destination are the same this means that the information in the dynamic memory will be lost.

It is important that the assignment operator(s) check whether the source and the destination refer to the same object before irrevocable changes are made to the data contained, as in the example below:

```cpp
cname & Cname::operator=(const CName & sourceName)
{
    // check whether assigning sourceName to itself
    if (this == &sourceName)
    {
        return (*this);
    }

    // return the current memory to the system
    delete[] m_name;

    // allocate memory for the copy
    int length = strlen(sourceName.m_name);
    m_name = new char[length + 1];
    // copy from the source to this
    strcpy(m_name, sourceName.m_name);

    return(*this);
}
```

**PC.37.C++.a** When possible, always use initialisation and initialisation list instead of assignment. This means the copy constructor is called, rather than the default constructor followed by an assignment operator.

Instead of

```cpp
int x;
x = 2;
```

use

```cpp
int x = 2;  // more efficient and clearer to read.
X x(2);    // or allocate using the ‘right’ constructor
```

**PC.37.C++ The assignment operator(s) must also assign base class member data.**

Care should be taken that assignment operator(s) for a derived class must also handle the member data of the base class. This member data should not be copied on member by member basis because this means that any modifications to the base class must be explicitly handled in the derived classes. In any case, there may be private data members of the base class to which the derived class does not have access. To work around this problem the assignment operator of the derived class should make use of the assignment operator of the base class. Remember that the base class assignment operator can be a protected member function of the base class so that it is available to derived classes but not to the rest of the world.

ii. But be careful of which declares a function called x returning an X.
class CDerived : public CBase
{
    // …
    CDerived & CDerived::operator=(const CDerived & sourceDerived);
};

CDerived & CDerived::operator=(const CDerived & sourceDerived)
{
    // check whether assigning sourceDerived to itself
    if (this == &sourceDerived)
    {
        return (*this);
    }

    // assign the data members of the CBase part of this
    // using the operator= function defined in CBase, i.e.
    // either CBase::operator(sourceDerived) ; or ((CBase) * this) = sourceDerived;

    // handle CDerived data memebers here
    // …

    return (*this);
}

PC.38.C++ The assignment operator(s) should return a reference to the object.

In the previous rules the assignment operator returned a reference to the object but without any explanation. The reason is quite simple: it allows for a series of assignments in one statement:
sonsName = fathersName = grandFathersName;

PC.39.C++ Symmetric operators, with the exception of assignment operator, should be defined as friend functions. All asymmetric operators (i.e. (), [], unary * and unary ->) must be defined as member functions.

Operators can be defined either as member functions or as friend function. Because member functions are inherently asymmetric (i.e. they treat the Object in the argument list and this in different ways), it is recommended to use friend functions to implement symmetric operators and member function to implement the others.

PC.40.C++ Member functions, which do not alter the state of an object, shall be declared “const”.

If a member function does not affect the internal state of an object, i.e. provides information about an object or calculates something derived from the contents of the object, then the member function should be declared as “const”. This allows the compiler to issue error messages if the programmer subsequently tries to modify the objet from within the function.

int CName::Length() const
{
    int length = strlen (m_name);

    return m_name;
}
PC.41.C++ Public member functions must not return non-const references or pointers to member variables of an object.

If a member function returns a non-const reference or pointer to a member variable the programmer may inadvertently change the value of a member variable by using this reference or pointer. As there are other routines to limit the access to the member variables this is probably not what the class designer intended. Such indirect access to member variables can result in corruption if other member variables are interdependent on the one being changed.

DESCOPED PC.42.C++ A function may not return a reference to memory, which it has allocated.

To avoid the problem of returning a pointer or reference to a local variable (see Rule PC.50.C++) the programmer may be tempted to allocate dynamic memory within a function and return details of this memory to the caller. If the function returns a pointer to the memory the programmer is able to see that the caller must take care of returning this memory to the system.

PC.43.C++ Member functions shall only be declared as “inline” if the need for optimization has been identified or if used instead of macros.

Small accessor and modifier functions shall be implemented as inline functions.

On systems where memory is limited, a significant increase in size may result in a drop of performance as the executable code is swapped in and out of memory.

Note that some debugging tools may not be able to handle inline functions.

PC.44.C++ A function shall not be declared as “inline” within the class definition itself.

4.4.11 GENERIC CLASSES

In C++ Template classes appear to be the answer to many programmers’ prayers but in reality are not as straightforward as they might at first appear, especially when trying to provide a truly general template class.

The design of a template needs some forethought. The designer must determine whether the parameter for the template is a pointer to an object, a reference to an object or the object itself. This can make a big difference to the implementation and efficiency of the template such as when copying objects which are passed "by value", which also requires the class to have a public copy constructor. Any class, which is used as a template parameter, must also satisfy all of the requirements of the template. For example a general purpose template may declare a function which uses the operator<< member function of the parameter class which means that the parameter class must have declared an operator<< member function itself even if this feature of the template class is not actually used.

PC.45.C++ Generic units should be encouraged as a convenient way of reusing code.

(Given the templates were properly designed, as discussed in the previous paragraph).

PC.45.C++.a Templates should only be used if all instantiations of the template will use the same algorithms.

Otherwise, use some other technique such as inheritance

PC.45.C++.b There should be no functions in a template that do not depend on the type the template is instantiated for.
With current compilers, this causes unnecessary code bloat. Instead, use a base class from which the actual template derives.

PC.45.C++.e If templates are used then Auto_ptr pointers should be preferred to normal pointers.

PC.45.C++.f All generic code shall work without modification when passed a valid subclass of a class it is expecting as an argument.

This implies no enums listing all known classes, hard-coded class names, downcasting, etc.

PC.45.C++.g All functions should use references or pointers to base classes wherever possible.

4.4.12 Type Conversion

PC.46.C++ The use of type conversion shall as widely as possible be avoided in order to maintain compiler specific type checking.

Type conversion in C++ is called casting and the traditional C method of casting simple variables and pointers from one type to another is still available in C++ but its use generally not allowed. The traditional C cast is an instruction to the compiler to override any type information, which it may hold about an expression.

Note that this can lead to portability problems because a cast, which may be valid on one system, may not be valid on another but the compiler assumes that the programmer knows best.

The C++ standard defines four new cast operators which are more specific in their operation and therefore the intention of the cast is clearer:

- static_cast<type>(expression)
- const_cast<type>(expression)
- dynamic_cast<type>(expression)
- reinterpret_cast<type>(expression)

The first is equivalent to a traditional C cast but is more obvious in the code. The second allows the programmer to remove the const or volatile nature of the expression.

The third provides a means of testing whether a class instance object belongs to a particular class within the class hierarchy.

The fourth is intended for particularly complicated and non-portable casting such as it concerns pointers to functions.

PC.46.C++.a Do not write code that force people to use explicit casts.

Avoid the use of explicit casts. If you have a pointer to a base class and you want to do a cast to get a pointer to a derived then you may not be using C++ in an object-oriented manner. Virtual functions should be used to go direct to the derived class code. Never cast away constness - never change a const.

Two exceptions:

- interfacing with C or old C++ code that does not correctly use const where it should.
• emulating the ‘mutable’ keyword in ANSI C++, whereby some attributes of an object can legitimately change even when the object itself is const

4.4.12.1 Function and procedures

The function and procedures sections known from the language independent guidelines are located here in relation to the member functions rules, due to the fact that it is generally forbidden to write global functions and procedures, they all have to be members of a class. The rule numbering has been kept to preserve trace ability with respect to [TN02]

PC.47.C++ A full function prototype shall be declared in the interface file(s) for each globally available function.

The ANSI C++ standard provides function prototypes as a means of "forward-declarations" of function, which are defined elsewhere

PC.48.C++ Each function shall have an explanatory header comment.

The standard header is shown below:

```c
// =========================================================================
// =========================================================================
// Function Name : CMDAcmdSheet::setColumnLabels
// Description : Set titles on the spreadsheet columns
// =========================================================================
```

As can be seen above the function comment field contains only two types of information, the name of the function and a description of the functionality implemented.

PC.48.C++.a Each function shall have an explicit return type. A function, which returns no value, shall be declared as returning "void".

A function, which does not have an explicit return type, is assumed to return an integer. This can lead to problems if there is a mismatch between what the caller of the function expects and what the function actually returns. If the return type is not specified explicitly the compiler cannot issue appropriate warnings if the function is called incorrectly or if the return value is invalid.

PC.49.C++ A parameter, which is not changed by the function, should be declared "const".

PC.50.C++ The layout of a function shall be well defined and used throughout the project.

In order to increase readability and maintainability each function shall have the same layout, to ease the navigation in the code

The following layout is used at SCOS-2000:

```c
// FunctionName() returns the XYZ coefficient using parameter1 and parameter2 for the boundary conditions. Assumes that both parameter1 and parameter2 are valid.

ReturnType FunctionName(
    ParamType1 parameter1,  // comment on parameter1
    ParamType2 parameter2   // comment on parameter2
)
```


```cpp
{ 
    // local declarations
    MSG_ASSERT( IsValidParam1(parameter1) == True );
    MSG_ASSERT( IsValidParam2(parameter2) == True );

    // body of function
}

PC.51.C++ A function may not return a reference or pointer to one of its own local automatic variables.

With the notable exception of static data variables within a function, all local variables are created on entry into the function and returned to the system on exit from the function. Therefore the caller has no guarantee that the memory corresponding to what was once the local data variable still contains anything useful or whether it has already been overwritten by something else.

The programmer should be careful not to introduce other potential memory problems in an attempt to work around the one described here. Although it is unlikely to occur in an ordinary function which return simple types.

PC.51.C++.a In order to pass an object of type T as a function argument, use type:

T if the object is small, value-like and not modified in the function.

- const T & if the object is large and not modified in the function.
- const T * if the object is large, not modified in the function, and may not be present.
- T & if the object is modified in the function. T* is an acceptable alternative that emphasizes the possibility of change to the object.

4.4.13 RECORDS/STRUCTURES

PC.52.C++ Records/Structs should be converted to explicit tagged types/classes where possible.

In C++ a struct is the equivalent of a class whose members are all public. Because it is effectively a class, it can have constructors, a destructor, assignment operators and other member functions.

We can give the following general layout for a union and structure:

```cpp
enum
{
    ShapeUnknown, // zero corresponds to invalid shape
    ShapeSquare,
    ShapeCircle
} ShapeId_t;
struct
{
    ShapeId_t id; // which shape is held in union
    union
    {
        Square_t square;
        Circle_t circle;
    } shape;
};
```
4.4.14 SPECIFIC OO FACILITIES

In order to increase the quality of the written programs as well as the maintainability the use of “Design Patterns” shall be done. Design patterns help simplifying complex language constructions and handling of complex situations where a normal programming approach would have cause problems in maintainability.

PC.53.C++ The use of Design Patterns shall be done as much as possible

Design Patterns frequently used is the Factory pattern concerned with the creations of new objects. The use of this pattern will standardise the way new objects are created. Several other patterns exist and a look into [DeP] is recommend before designing new software or software modules.

4.5 NAMING CONVENTION

4.5.1 GENERAL NAMING CONVENTION

As some compilers and systems have their internal material such as libraries identifiers, use multiple underscore in their names it is generally forbidden to use underscore in normal program variable name.

NC.1.C++ Names shall in general not start with an underscore character (_).

NC.2.C++ Names shall be meaningful and consistent.

Descriptive names are preferable to "random" names. However the descriptive name must reflect the use of the entity over its lifetime. For example "count" is better than "xyz" for the name of a variable which contains the total number of something but not if it is used to hold an error code.

NC.3.C++ Names containing abbreviations should be considered carefully to avoid ambiguity.

In the absence of a list of standard abbreviations different programmers are likely to choose different abbreviations to represent a particular word or concept. This is especially true in the multi-lingual environment found in many ESA projects. Abbreviations should be used with care. For example does "opts" refer to "options" or the "old points" or something else?

NC.4.C++ Avoid using similar names, which may be easily confused or mistyped.

Names which differ by only one character, especially if the difference is between "1" (the digit) and "l" (the letter) or "0" (the digit) and "O" (the letter). Avoid names, which consist of similar meaning words because it is easy to confuse them. Consider the differences between "x1" and "xl" and between "countX" and "numberX".

NC.5.C++ Names for system global entities shall contain a prefix, which denotes which subsystem or library contains the definition of that entity.

The subsystem or library prefix provides additional information about the origin of an entity, which can be useful to the programmer. The basic name (without the prefix) can be reused elsewhere in the source code without causing a name clash.

NC.6.C++ Names, which have wide scope and long lifetimes, should be longer than names with narrow scope and short lifetimes.

Names for entities which are local to a block or function may be short, e.g. count, result. Names for entities which are available throughout a particular module should be longer, e.g. maxNumber, startPoint. Globally visible entities should have long descriptive names, including library prefix, so that their origin and use are clear, e.g. uiWindowIndex, uiMousePosition. Especially for variables with a large scope, attention shall be made to their naming, allowing programmers to identify their purpose.
NC.6.C++.B Variables with large scopes are not allowed to have generic names.

NC.7.C++ Each project shall define its own specific rules for naming conventions.

As this note is concerned with the a general guidelines are no further rules or examples given as the different programming languages all have their own de facto naming convention. Hence it is for this area recommended to look in to the language specific notes.

NC.7.C++.a Any operation that matches one of the descriptions below should use the corresponding term as the first part of its name.

- **get** - access attribute (either direct member data, or something directly derived from it). Not allowed to do anything ‘clever’ or time consuming, or fail.
- **calculate** - access attribute, with clever, time-consuming and fallible operations permitted.
- **set** - change attribute (either direct member data, or something directly related to it). Not allowed to do anything ‘clever’, time consuming or fallible.
- **change** - set an attribute, and do any associated non-local processing.
- **update** - bring an object up to date.
- **is** - boolean test on an object.
- **add** - put something into a collection.
- **find** - look for something inside a collection.
- **remove** - take something out of a collection.
- **move** - place in a different position in a collection.
- **instance** - access singleton object.
- **clone** - virtual deep copy.
- **selfCheck** - class invariant. Returns false if there is something wrong.
- **make** - create an object.

The intent of the above list is to make it obvious from the name what an operation is doing, and in particular to distinguish between simple operations that cannot fail, and complex, time-consuming ones that can.

NC.7.C++.b Accessor functions for an attribute shall always be based on the attribute name without the m_ prefix.

Attributes that are not mentioned in the high level architectural documentation should not be publicly visible, even through accessor functions.

4.5.2 TYPE AND CLASS NAMES

NC.8.C++ User defined type and class names consist of one or more words where each word is capitalised plus an appropriate prefix or suffix.

NC.9.C++ User defined type names shall begin with "T" or end with "/_t" or "/_type".

NC.10.C++ Class names shall begin with "C" or end with "/_c" or "/_class".

4.5.3 CLASS MEMBER VARIABLE NAMES

For Class member variables the naming convention recommended is the one used for normal variables as the variable will only be used without its class reference inside its own class scope. Hence, the referencing to the origin class would be unnecessary redundancy.

iii. i.e. it is not useful for the caller to test for failure.
4.5.4 ENUMERATED TYPES

NC.11 C++ Each enumeration within an enumerated type shall have a consistent prefix.

It is not uncommon for different enumerated types to be used within a program which have similar characteristics, such as those representing the "status" of different parts of the system. Although they are distinct, enumerations within one type may have similar roles to those of another type. To reduce the risk of name clashes for such enumerations, the name of each enumeration should include a prefix which denotes the exact enumerated type to which it belongs. This also reduces possible confusion when converting to or from integer variables.
APPENDIX 1 HEADER AND IMPLEMENTATION FILE OUTLINES

Below the standard structure for the declarations files .H

// -------------------------------------------------------------------------------------
//
//
// Mission Control System
//
// Sub-System  :
//
// File Name   :
//
// Author      :
//
// Creation Date:
//
// Description :
//
// -------------------------------------------------------------------------------------
//
#ifndef
#define

// -- Includes
#include <*.h>
#include "*.H"

// -- Forward Declarations
class .....
{

public:

// -- Members

// -- Methods
protected:

// -- Members

// -- Methods

private:

// -- Members

// -- Methods

};

#endif
Below the standard structure of an implementation file .C file.

/****************************************************************************
// (C) 2000 European Space Agency
// European Space Operations Centre
// Darmstadt, Germany
//
/****************************************************************************
// System: SCOS-2000 - the Satellite Control and Operations System
// Sub-System:
//
// Function Name: CMDAcmdSheet::CMDAcmdSheet
// Description: Constructor
/****************************************************************************
#include <.h files>
#include ".H files"

// -- Forward Declarations

// -- Constants, Enums, Typedefs, Statics

/****************************************************************************
// Function Name: CMDAcmdSheet::CMDAcmdSheet
// Description: Constructor
/****************************************************************************