PATNI COMPUTER SYSTEMS LTD

Coding Standard for C++

Compiled by Smart Programming CoP

DOCUMENT HISTORY

Reference no:		
Security classification:	**Patni internal confidential**	
Issue date:	**28-Mar-11**	
Contributed by	**PES**	
Date	**Version**	**Change Description**
24-Mar-11	**1.0**	**Baseline**

TABLE OF CONTENTS

1. INTRODUCTION

This document defines a set of *rules* and *guidelines* for the production of C++ code. Justification is provided for each rule or guideline. The guiding principles of this standard are safety, reliability, portability and readability. This involves defining a base language dialect and then further restricting use of this to a safer subset. Language designers tend to include as many features as they can think of in the language they are designing. It is useful to work within a subset that contains no features which are open to interpretation either by compiler writers (such features may vary from implementation to implementation) or by programmers (who may introduce subtle bugs that are hard to find, or may write code in such a way that the next programmer introduces a bug because he does not understand what the first programmer wrote). High quality code is portable, readable, clear and unambiguous.

1.1 PURPOSE

The C++ coding standard guides software-projects in order to achieve the following goals:

- High quality
- Uniform realization of software projects
- Reusable software-components

This C++ coding standard is a prerequisite for the uniform realization of software development projects for group products within Patni.

1.2 SCOPE

This coding standard is valid for all software development projects in Patni for group products using a C++ compiler.

1.3 FURTHER DEVELOPMENT

As the language is new to many programmers, great care has to be taken defining a safe subset of C++ so that also the novice C++-programmer will be guided to produce safe, readable and maintainable code.
Therefore this document will also address problems that are legal in C and the pre-processor.
There is a significant difference in stability and maturity of compilers on different platforms.
This standard attempts to restrict the risk of using the questionable parts of the language to allow the benefits to be realized.
This programming standard is an introduction to boosting the awareness of quality issues and potential risks when writing C++ code. It is also intended, as a database to collect project knows how and experiences with potential field bugs in the C++ development context. Stylistic issues and naming conventions are not covered in this document to prevent distraction from the main issues (see coding style standard).
This programming standard shall not be viewed as a fixed document but should benefit from continuous reviews in a maturing development environment. With growing experience of developers it might e.g. be sensible to promote some guidelines to rules or, in project environments new to a development team, demote some rules to guidelines. Further changes can result from growing experience in the field of developing in C++ and typical projects.

2. GENERAL

In general, all rules and guidelines defined in"C Coding Standards" apply. Rules and guidelines defined here are in addition to those defined in "C Coding Standards". In case of conflicts, the guru/software architect/expert team should be consulted.

Guideline 2.1	Do not declare a variable before you can initialize it.
Justification	A declaration can occur anywhere within a statement. To keep the scope as small as possible and to call constructors as late as possible (and thus only if really needed), variables can even be declared in **for** statements: e.g. **for (int i; ...)**.
Rule 2.1	Keyword **extern** and **goto** are forbidden.
Justification	This breaks the principle of object oriented software development.
Rule 2.2	**malloc, realloc** and **free** are forbidden.
Justification	They are replaced by **new** and **delete**.
Rule 2.3	Pointer arithmetic is forbidden.
Justification	It might be needed only deep inside the implementation of a class or function.
Rule 2.4	**void** pointer arguments are forbidden.
Justification	It might be needed only deep inside the implementation of a class or function.

2.1 COMMENTS

Code should have proper comments. The comments should explain the flow of the code as well as the usage of the variables. Comments should also be present on top of every module and top of every function definition. The Doxygen documentation standards may be used for writing comments in the code.

The Comments should be added at:

1. Top of every module
2. Top of every function definition
3. Top of every macro definition
4. At each major type definition
5. Before each logical group of defined constants
6. On each variable and parameter definition
7. Before logical blocks of code in each function

2.1.1 Descriptions

For each code item there are two types of descriptions, which together form the documentation: a brief description and detailed description, both are optional. Having more than one brief or detailed description however, is not allowed.

Brief description

As the name suggest, a brief description is a short one-liner.
Format:

```
/* brief description
 *        Brief description continued.
 *
 * Detailed description starts here.
 */
```

Detail description

The detailed description provides longer, more detailed documentation.

2.1.2 File headers

Format

```
/* File [<name>]
 * Brief <brief description of the file>.
 *
 * Author <author name>
 *
 * Version <version number>
 *
 * Date <Latest date of modification>
 *
 * <details of the file e.g functionality, dependencies etc.>
 */
```

Example
For Example the C header named structcmd.h can be documented as

```
/* File: structcmd.h
 * Brief: A Documented file.
 *
 * Author: Sam
 *
 * Version 1.0
 *
 * Date     01-03-2007
```

```
 *
 * Details.
 */
```

2.1.3 Function headers

Format

```
/* <function declaration>
 * Brief: <brief description of the function>
 *
 * Params: <parameter-name> {parameter description}
 * Return: {description of the return value}
 */
```

Example

For Example the function int read(int f_nFd,char *f_pcBuf,size_t f_tCount) can be documented as

```
/* int read(int f_nFd,char *f_pcBuf,size_t f_tCount)
 * Brief: Read bytes from a file descriptor.
 *
 * Params:  f_nFd The descriptor to read from.
 *          f_pcBuf The buffer to read into.
 *          f_tCount The number of bytes to read.
 * Return: returns 0 if fails and 1 for success.
*/
```

2.1.4 Comments in Code

1. All local / global / member variables should be commented and comments should be mentioned at right side of the declared variables.
2. Comments should be followed for logical section of code.
3. Comments should be there for each if/ else and for statement.
4. Each function definition should have function header mentioned in coding guidelines.
5. Each function declaration should have single line comment at the top of it.
6. #include should have comment at the top stating for what this file is included in code. Write group comment If #include in code are framework related / are standard one.
7. Local variables in code should be declared at the start of function even C++ allows to have it whenever required. This will increase the readability as local variables are expected with comments at right side.

Expressions

Expression semantics are very versatile in C++, however, not always unambiguous. The focus of this chapter is to avoid subtle bugs and ambiguities.

Rule 0.1 One fact in one place. Expressions shall not contain multiple side-effects due either to the same identifier being modified more than once, or due to the same identifier being modified and accessed.

Justification For all but a few constructs in C++, the standard does not define the evaluation order even where there may be side. Reliance on them can create problems which are very expensive to track down.

```
x = i++ + a[i];                          // WRONG - 2 side effects
                                         // x assigned and i changed

x = i + a[i];                            // RIGHT
i++;
```

2.2 Unary Operators

2.2.1 Increment (++, --)

Rule 0.2 The postfix increment and decrement operators must not be mixed with prefix forms in the same expression.

Justification Expressions are particularly confusing to read when both postfix and prefix forms are used simultaneously. When using the postfix form the compiler has to keep a temporary copy of the object to be modified which might result in significant performance overhead when objects are non trivial.

```
c = x++ + --j;                           // poor style! And,
                                         // a temporary copy needs
                                         // to be kept for x!
```

2.3 Multiplicative Operators (*, /, %)

Rule 0.3 Both division (/) and remainder (%) operations should be guarded by a test on the right hand operand being non-zero.

Rule 0.4 The reminder operation should additionally be guarded to ensure that both arguments are non-negative.

Justification Defensive programming like this reduces the effects of implementation defined and undefined behaviour.

2.4 Shift Operators (<<, >>)

Rule 0.5 The right operand of a shift operator in a constant expression must not be negative or imply too large a shift.

Justification These Operations are not defined by the standard.

```
int array[ 32<<-3];                      //  undefined
int array[ 0] = 32<<66;                  //  undefined
```

Rule 0.6 The left operand of a shift operator must not be signed.

Justification A signed left operand of a right shift will produce an arithmetic shift on some platforms and a logical shift on others.

```
int             j;
unsigned int    x;
unsigned int    u;

x = j >> 8;                                 // wrong

x = u >> 8;                                 // right
```

2.5 RELATIONAL OPERATORS (<, >, <=, >=)

Rule 0.7 The operands of relational operators must be parenthesised unless both are simple values, identifiers, or function calls.

Justification The precedence of operators in C/C++ is not intuitive. Extra parentheses aid the reader establish the grouping of operands.

```
if (i <= 10)                                // ok. – comparison of
single value

if (i-k > 10 && k-i < 10)                   // bad: precedence not
obvious

if (i < j < k)                              // bad: meaning unclear
```

2.6 CONDITIONAL OPERATOR (... ? ... : ...)

Guideline 0.1 Avoid conditional operators.

Justification The convenience of the conditional operator should be weighed against the clarity of the code.

```
(a > b) ? i+=(a*b-a): j++;  // BAD

x = ((a > b) ? a : b);  // ALLOWED
```

3. TYPES AND DATA

3.1 DECLARATION AND DEFINITION

Guideline 3.1 Define Objects in the smallest possible scope. Exception: To prevent expensive stack operations and for debug purposes.

Justification It is a good practice to keep the scope of objects small for maintenance reasons as well as to avoid allocation of stack space for unneeded objects and unnecessary calls to constructors and destructors:

```
int function(int aParam)
{
   int returnValue = 0;

   if(aParam < 0)
   {
      returnValue = -1;// No need to construct an destroy an
                                  // instance of SomeLarge Class
   }
   else
   {
      SomeLargeClass myObject(aParam);
...
      returnValue = myObject.getInteger();
   }
   return returnValue;
}
```

Rule 3.1 Forbidden global Data.

Justification This breaks the principle of encapsulation and leads to pollution of the global namespace. This in turn raises risks and efforts in maintenance phases. Use Singelton Pattern to have only one instance.

Rule 3.2 Variables and constant objects defined in an inner block of a function must be uniquely named within the function.

Justification Not all compilers treat the scope of variables correctly that are e.g. defined in a for statement. This also increases readability and maintainability.

```
int someFunction(void)
{
   int i = 0;                          // first encounter of i

   for(int i = 0; i < 10; i++)
   {                                   // according to ISO C++ this i is
                                       // valid only inside the loop.
   }
                                       // outside the old i still
persists
}
```

3.2 FLOAT TYPES

Rule 3.3 floats must not be compared for equality or inequality.

Justification Only rarely will direct comparisons yield the desired result: e.g. sqrt(3) * sqrt(3) will usually not equal 3 because of rounding errors.

```
float f1, f2;
...
if (f1 == f2)                        // will not yield the desired
result
```

3.3 TYPE CONVERSIONS

Rule 3.4 If you do need explicit type conversions use the C++ cast operators.

Justification These cast operators have a more specialised purpose which is evaluated by the compiler and thus much safer than the old C-style cast (which remains a legal construct in C++). Also the new operators increase readability and are easier to maintain.

3.4 CONSTANTS

Rule 3.5 Use const whenever possible (whenever constness is meant).

Justification Declaring an item as const allows to specify a semantic restriction that is enforced by the compiler. Thus, if an object is not to be modified it should be declared const.

```
// different examples to illustrate the versatility of const
char* some_pointer;

// variable pointer to a constant string
const char* ptr_to_const_obj = ”xyz”;

ptr_to_const_obj[1] = ‘A’;                 // compiler error
ptr_to_const_obj = some_pointer;          // o.k.

// constant pointer to variable string
char* const const_ptr_to_obj = ”xyz”;

const_ptr_to_obj[1] = ’A’;                // o.k.
const_ptr_to_obj = some_pointer;          // compiler error

void find (const ULong ulBigNumber, UShort& rusSmallNumber);
AtmVclIndex getAtmVclIndex (const ULong ulId) const;
```

Rule 3.6 Constants are defined using const. Do not use #define.

Justification const objects have type, scope and linkage like variables that are being checked by the compiler. Constants created with #define lack these features since they are only textual substitutions! Very important also for debugging.

```
// using const and enum instead of #define
const int MAX_SIZE = 1000;
```

```
void function(void)
{
   int someArray[MAX_SIZE];
   const int lowerLimit = 0;
   const int upperLimit = MAX_SIZE;

   for (int i = lowerLimit; i < upperLimit; i++)
   {
...
   }
```

Rule 3.7 consts and enums must be declared inside a class definition.

Justification This prevents the pollution of the global namespace and explicitly specifies the origin of the data.

3.4.1 Integer and Float Constants

Rule 3.8 Code shall not contain explicit constant values - so called “magic numbers”.

Justification Readability demands to define constants with these values giving them a sensible name. This also enhances maintainability.

```
// BAD
for (int i = 1; i < 256; ++i)
{
   if (ucVal[i] != 0xff)
   {
      ulSum += ucVal[i];
   }
}

// GOOD
const UChar VAL_INVALID = 0xff;
const int   ARRAY_MIN   = 1;
const int   ARRAY_MAX   = 256;

for (int i = ARRAY_MIN; i < ARRAY_MAX; ++i)
{
  if (ucVal[i] != VAL_INVALID)
   {
      ulSum += ucVal[i];
   }
}
```

3.5 POINTERS

Rule 3.9 The following operations on pointers are forbidden: !, &&, ||, as well as the addition and subtraction of two pointers.

Justification Generally speaking these operations do not state the semantics of the code. Subtraction might be useful in time critical array operations. Hardware specific code might also rely on such constructs.

```
if(MyPtr > RefPtr)                          // forbidden
{
}

if(MyPtr != NULL)                    // better, now there is a logic
```

```
{                                       // expression
}
```

Rule 3.10 Pointers to functions/methods are forbidden.

Justification These does not naturally occur in C++. Although, it is still a legal construct but it is a C legacy that is not needed in C++ anymore. They are hard to understand and can make debugging very strenuous. The OO paradigm provides function derivation of functions from a virtual base class to achieve the same result in a more transparent way.

Guideline 3.2 Avoid pointers to pointers.

Justification They are hard to understand and tend to provoke errors during maintenance.

3.6 ARRAYS

Guideline 3.3 Container classes should be used instead of built-in arrays (except for arrays of integral types).

Justification Arrays and polymorphism do not mix. Since there is no bound checking you will run into problems when storing derived objects in arrays of the base class.

3.7 STRINGS

Guidline 3.4 Do not use C-style strings. Use String class library instead.

Justification Since native C strings are nothing but character arrays the same problems occur with strings. The use of almost any of the widely available String Classes relives the programmer of difficult and error prone memory manipulations and are safer and more comfortable in usage than the native C strings.

3.8 THE BOOL TYPE

Guidline 3.5 Use the C++ builtin 'bool' type for Boolean expressions whenever possible.

Justification Old style boolean macros and constants are deprecated, as there was no standard value for 'true'. The C++ now handles this as a builtin type .

```
TRUE_FALSE MyVariable; // deprecated
BOOLEAN    AnotherVar; // deprecated
bool       isOK;       // recommended
```

Rule 3.11 Variables of type 'bool' shall be assigned ONLY 'true', 'false', boolean results of comparisons, and results of boolean logic operations.

Justification This prevents accidentally assigning values other than 'true' or 'false' to bool types. C++ just happens to define a 'bool' equal to 0 as false, and all other values as true. Your software should be completely independent of that convention and never count on it. Therefore, all software using 'bool' should be written so that there is no place that any illegal value can ever be assigned to a bool except 'true' and 'false'.

```
bool      isOK;
bool isReady;
```

```
bool isHungry;
int A,B,C;

isOK = (A<B)&&(A<C); // OK
isOK = true;         // OK
isOK = false;        // OK
isOK = isReady && isHungry; // OK
isOK = isReady || (A<C);    // OK


bool isWrong;
isWrong = TRUE;      // NO! 'TRUE' and 'FALSE' are some other enum!
isWrong = 0;         // NO! Don't assume false is 0
isWrong = 1;         // NO! Don't assume true is 1
isWrong = VXT_STATE_1; // Don't assume you know the value of 'true'
isWrong = 7;         // Now we're really inviting disaster
isWrong = 0x01 | variable; // bitwise logic is NOT type bool!
```

Rule 3.12 Never compare directly against the boolean value 'true'

Justification Since there was traditionally no standard value for true (1 or !0); comparisons of non-zero expressions to true could fail. Use Boolean expressions instead. For maximum safety, non-bool expressions should always be compared against expressions of the same type whenever possible. Note also that most CPUs are most efficient performing comparisons == or !=0 (In many CPUs there is a 'Zero' bit that is automatically set whenever the most recent operation resulted in zero. Comparisons against other values require additional instructions.) Comparisons against nonzero values are generally more expensive.

```
// DON'T:
if(someNonzeroExpression == true) // NO!!
    // Might unexpectedly not be true is the value is >1

// DO:
if(someNonzeroExpression)
    // OK
```

Rule 3.13 Never depend on sizeof(bool)

Justification Since the introduction of 'bool', it has had sizes ranging from 1 to 4 bytes. 'bool' should therefore not be used as a member of structure that depends on the size of its elements being fixed. (ie, structures overlayed on a stream of bytes, or structures stored in nonvolatile memory)

3.9 QT FRAMEWORK DATA TYPES

Rule 3.14 QT datatypes to be used

Justification uchar,

ulong,

uint,

ushort

(Reference - http://doc.qt.nokia.com/4.4/qtglobal.html)

3.10 The Naming Convention

Name that shows the domain that is actual object or target. Add a prefix at the beginning showing the type for the basic type attribute. However, this limit does not exist for the objects used conventionally like the variables etc, used in counter for “for” loop. Also, related to pointer, character string etc, add a prefix which gives its meaning. The prefix of pointer should be at the beginning. In case of the pointer variable showing a specific attribute, “p” is added before the original attribute name. All pointers will have prefix ‘p’.

Note: All prefixes and data type should be in small letters.

Prefix	Separator	Type Specifier	Data Type	Name
Define Scope of variables See section 'Scope Specifier'	Underscore’_’	See section 'Type Specifier'	See section 'Data Type'	Variable name starting with Capital letter. If the Name consists of more than one word,the starting character of each word will be in Capital letters. Others will be strictly lowercase. **General Rule:** <scope>_<type specifier><data type>VariableName for e.g. 1. l_uiIndexValue - local unsigned integer variable 2. g_uiIndexValue - global unsigned integer variable 3. l_ccInputValue - local const character variable

Scope Specifier	Prefix (all in lower case)
Local Variable	l
Global Variable	g
Function Variable	f
class member Variable	m
Type Specifier	**Prefix (all in lower case)**
const variable (Not to be used for const reference variable)	c
static variable	s
register variable	r
volatile variable	v

Data Type	Prefix (all in lower case)
Note: Standard data types (as mentioned below) shall be used across all the project code files	
Char	c
Unsigned char	uc
Short	s
Unsigned short	us
Int	i
Unsigned int	ui
Long	l
Unsigned long	ul
Unsigned long long	ull
Float	f
Double	d
bool	B
Pointer	p
QList	lst
Queue	q
QVector	v
QMap	m
QString	c
Iterator	it
Function Pointer (Only for loading 'so')	fp

Naming Conventions for class name, object to class and pointer to class.

Type	Prefix
Name of the class	CClassName Example : Video class will have name Cvideo
Name of class object	objectObjectName Example: object ProcessedVideo of class CVideo will have name objProcessedVideo
Name of class pointer	pPointerName Example: pointer class CVideo with name ProcessedVideo will have name pProcessedVideo

Naming Conventions for nonbasic data types

In case of non-basic types, add a prefix showing the type at the beginning as follows.

Type	Suffix
Enumeration type(enum)	E
Structure(struct)	T

Tags:

This includes typedefs, structs, enums and unions.

Names for user-defined data-types should be in Upper case with suffix _Type

If the name consists of more than one word, they shall be separated by underscore '_'.

Structures and Enums should always be defined as typedefs

Member variable of the structure will have the prefix ‘m’.

For Example:

• Structure for coordinates can be defined as

```
typedef struct COORD_T
 {
   int m_x;
int m_y;
    } COORD_Type;
```

• Enum for different IO operations can be defined as

```
typedef enum IO_TYPE_E
{
 READ_TEXT = 0,
 WRITE_TEST,
 READ_BIN,
 WRITE_TEXT,
} IO_TYPE_Type;
```

3.11 SIGNAL SLOT NOMENCLATURE

3.11.1 Signal Slot nomenclature

Follow the standard function nomenclature along with addition of ‘Signal’ / ‘Slot’ for all the signal slots QT based classes. Example – PlaybackActionSignal – PlaybackActionSlot

3.11.2 Passing param and return type for signal slots

Coding guideline states that function should only return the error code / status as return value and hence function type should be int. If there is no error condition then function type should be void. However, this is not possible for signal slot functions.

Also as per coding guideline reference variables should be use for passing parameter. However, its not possible for signal slots functions.

STATEMENTS

3.12 LABELED STATEMENT

Guideline 0.1 Statements must not be labelled.

Justification The only reason for using a labelled statement is to jump to it. Such jumps are forbidden (this excludes inline assembler of course). Exceptions to this rule are case and default labels, which are structured elements in C/C++. (these labels are keywords of the language and not self defined labels in the above sense).

3.13 CONTROLLING STATEMENTS

Rule 0.1 A controlling expression shall not be an assignment.

Justification Isolating side-effects in expression statements helps the reader by making changes in state explicit, separating them from tests on that state. Implicit actions on behalf of an implementation are to be avoided for the sake of clarity. Modern compilers will optimise a comparison against zero in any case, defeating any arguments of efficiency.

```
int i;

if (i = 1)                                                    // WRONG -
assignment with implicit test

if (i)                                                        // WRONG -
compare should be explicit

if (i == 1)                                                   // RIGHT
```

Rule 0.2 All if, else, for, while statements must be followed by a block {..} construct even when only one statement, (null (;) or otherwise), is used.

Justification If only indentation is used to indicate that a single statement belongs to an if, it is a frequently occurring mistake to enhance this construct by adding another statement without supplying the now necessary {..} to indicate that both statements belong to the if. The practice of using {..} also aids readability.

```
while ( ... );                              // WRONG - no {}
while ( ... )                               // RIGHT
{
   // EMPTY
}

if ( ... ) i++;                             // WRONG - no {}
if ( ... )                                  // RIGHT
{
   i++;
}
```

3.13.1 if statement

Guideline 0.2 Multiple choice constructs programmed using if…else if... shall have a “catch-all” else clause.

Justification Defensive programming requires the presence of else in a multiple choice construct, to guard against unwanted program status. Use empty else part for error handling.

```
if (temp < 0)
{
...
}
else if (temp > 0)
{
...
}
else                                          // RIGHT - safeguard
{
   error_message( .... );
}
```

Guideline 0.3 Avoid negations in logic expression if possible.

Justification Negations are hard to trace at times and are easily misunderstood.

```
if (activate(*pNvRamParam) != NO_SUCCESS ) // avoid it
{
   printf (“\n\rEverything successful!”);
}

if (activate(*pNvRamParam) == SUCCESS ) // better
{
   printf (“\n\rEverything successful!”);
}
```

3.13.2 switch statement

Rule 0.3 All switch statements must have a default clause, even if that clause is empty.

Justification In analogy to multiple choice if .. else if .., the same justification applies for switch.

Rule 0.4 The switch expression must not contain any logical expression (one or more of the ‘>‘, ‘>=‘, ‘<‘, ‘<=‘, ‘==‘, ‘!=‘, ‘&&’, ‘||’ or ‘!’ operators) .

Justification Using relational or logical conjunctive operators in a switch expression is very likely to be wrong and is at best confusing.

Rule 0.5 The default clause shall be the last entry in the switch statement.

Justification This increases clarity.

Rule 0.6 Each code segment has break statement at the end.

Justification This increases clarity.

```
switch (i)
{
```

```
case '1':                                   // RIGHT - empty case
case '2':
   k = i
   break;
case '3':
   k = i + 2;                               // WRONG - no break
default:                                    // RIGHT - default available
   assert( … );
   break;
}
```

3.13.3 for statement

Rule 0.7 A control variable must not be altered by the body of a for statement.

Justification Modifying a control variable is a very confusing practice - bugs can easily be introduced when the code is enhanced later on in the software cycle.

```
for (int i = 0; i < 10 ; i++ )
{
   i--;                                     // Bad: alters the usual
behaviour
}
```

3.13.4 return statement

Rule 0.8 Functions shall have exactly one entry and one exit.

Justification For the sake of clarity functions should be consequently structured: Functions returning void should not have any return statement; all other functions should preferably have one sensible return statement. Functions may have more than one return if this increases clarity substantially.

4. FUNCTIONS AND METHODS

4.1 FUNCTION ARGUMENTS / PARAMETERS

Rule 4.1 It is forbidden to define functions with an unspecified number of arguments (ellipsis Notation).

Justification The best known function which uses unspecified arguments is printf(). The use of such functions is not advised since the strong type checking provided by C++ is thereby avoided. Some of the possibilities provided by unspecified function arguments can be attained by overloading functions and by using default arguments. The use of functions of the standard library that make use of this feature can be considered an exception to this rule since their use should be sufficiently practised.

Guideline 4.1 Prefer references as function arguments. Only if a function needs to work with the pointer to an object let the argument be a pointer type.

Justification By using references instead of pointers as function arguments, the code can be made more readable, especially within the function. Another advantage is that there are no such things as NULL references, this ensures that there is an instance of an object to be worked with.

```
// too complex use of pointers
void addOneComplicated(int* integerPointer)
{
   *integerPointer += 1;
}

addOneComplicated(&j);                    // call

// better
void addOneEasy(int& integerReference)
{
   integerReference += 1;
}

addOneEasy(i);                            // call
```

Rule 4.2 Use constant references (const &) instead of call-by-value, unless using a pre-defined data type or a pointer.

Justification One difference between references and pointers is that there is no null-reference in the language, whereas there is a null-pointer. This means that an object must have been allocated before passing it to a function. The advantage with this is that it is not necessary to test the existence of the object within the function.

Note C++ invokes functions according to call-by-value. This means that the function arguments are copied to the stack via invocations of copy constructors, which, for large objects, reduces performance. In addition, destructors will be invoked when exiting the function. Const & arguments mean that only a reference to the object in question is placed on the stack (call-by-reference) and that the object's state (its

instance variables) cannot be modified. (At least some const member functions are necessary for such objects to be at all useful).

```
// inefficient: a copy of the argument is created on the stack.
void foo1( String s );
String a;
foo1( a );     // call-by-value

// may lead to messy syntax when the function uses the argument.
void foo4( const String* s );
String d;
foo4( &d );    // call-by-constant-pointer

// the actual argument can be modified by the function.
void foo2( String& s );
String b;
foo2( b );    // call-by-reference

// the actual argument cannot be modified by the function.
void foo3( const String& s );
String c;
foo3( c );    // call-by-constant-reference
```

4.2 FUNCTION OVERLOADING

Overloading of functions can be a powerful tool for creating a family of related functions that only differ as to the type of data provided as arguments; however care must be taken since this causes considerable confusion!

Guideline 4.2 When overloading functions, all variations should have the same semantics (be used for the same purpose).

Justification If not used properly (such as using functions with the same name for different purposes) this leads to wrong use of these functions and generates hard to find bugs.

```
// proper usage of function overloading
class String
{
public:                                 // Used like this:
   //                                   // String x="abc123";
   bool contains( const char c );       // bool i=x.contains('b');
   bool contains( const char* cs);      // bool j=x.contains("bc1");
   bool contains( const String& s);     // bool k=x.contains( x );
   // ...
};

// This is the price you pay for the comfort:
if ( my_str.contains(5) ){   //now the compiler has many more
                             // possibilities to find a conversion:
                             // int -> char, int -> char*, int ->
                             // String these ambiguities will cause
                             // compiler errors.
```

4.3 FORMAL PARAMETER

Rule 4.3 The names of formal arguments to functions have to be specified and are to be the same both in the function declaration and in the function definition.

Justification Providing names for function arguments is a part of the function documentation. The name of an argument may clarify how the argument is used, reducing the need to include comments in, for example, a class definition. It is also easier to refer to an argument in the documentation of a class if it has a name.

```
int setPoint(int, int);                    // bad!
int setPoint(int x, int y);                // ok.

int setPoint(int x, int y)
{
    // ...
}
```

4.4 RETURN TYPES AND VALUES

Rule 4.4 A function must never return a reference or a pointer to a local (non static) variable (that is placed on the stack).

Justification If a function returns a reference or a pointer to a local variable, the memory to which it refers will already have been de-allocated, when this reference or pointer is used, and might thus be reused otherwise by then.

```
char* strangeFunction(void)
{
   char localBuff[Max];

   return localBuff;          // BAD: return pointer to stack!
}

a = strangeFunction();        // This might work for a few tests, but
                              // sooner or later this will lead to
                              // memory exceptions!
```

4.5 INLINE FUNCTIONS

Inline functions have the advantage of often being faster to execute than ordinary functions. The disadvantage in their use is that the implementation becomes more exposed, since the definition of an inline function must be placed in an include file for the class, while the definition of an ordinary function may be placed in its own separate file.

The compiler is not compelled to actually make a function inline. The decision criteria for this differ from one compiler to another. It is often possible to set a compiler flag so that the compiler gives a warning each time it does not make a function inline (contrary to the declaration). “Outlined inlines” can result in programs that are both unnecessarily large and slow.

Guideline 4.3 Avoid long inline functions (more than 3 Statements).

Justification As mentioned above this implements compiler dependencies, breaks modularity, and thus compromises maintainability.

4.6 NOMENCLATURE FOR FUNCTION

Guideline 4.4 Function Names

Justification Use following style of nomenclatures for function names- GetListHandle. The function name should have first letter as capital letter and words in function name should be differentiated by capital letter. Get and List these two words are differentiated by 'L' of word list.

4.7 FUNCTION DECLARATION

Functions should be delceared along with the passing variables. Example –

Correct - > int CQuickPlayList::CreateList(CListCreationInfo *& f_pListCreationInfo,
uint & f_ulListHandle,
uint & f_ulListSize,
uint f_cuiSourceAppID,
uint f_cuiRegisterID)

InCorrect -> int CreateList (CListCreationInfo *&,
uint &,
uint &,
uint f_cuiSourceAppID = 0,
uint f_cuiRegisterID = 0);

5. CLASSES

Classes introduce the paradigm of object orientation in C++, They are objects or abstract data types. They do not only contain data but also procedural code. They encapsulate groups of data and the methods to work on the data. Apart from that, by deriving or inheriting certain aspects of a class from a base class, the so called polymorphism is being introduced. Working with these complex language features is not always intuitive and the following rules will be guidance.

5.1 ACCESS RIGHTS

Guideline 5.1 The data of a class should be **private** and set get APIs should be provided for accessing the member variables. The member variables should be public for the class that is used as structure. (class containing only member vairables and not the functions)

Guideline 5.2 Following order should be used for the scope respresentation of the class.

```
Class A
{
    private:
            List of member vairables
            List of member functions
    protected:
            List of member vairables
            List of member functions
    public:
            List of member vairables
            List of member functions
    signals:
            List of member functions
     public slots:
            List of member functions
     Q_SLOTS:
             Only for Dbus APIs
};
```

Note: Having scope resolution private, protected and public are mandatory even if there no variables function defined. Scope resloution – signals, public slots and Q_SLOTS are optional based requirements.

5.2 RETURN IN MEMBER FUNCTIONS

Rule 5.1 Do not return a non const pointer or reference to data outside of the class from a public member function, except to share the data with other objects.

Justification This is another way to break encapsulation. If an object relies on external data that can be modified by means not inherently belonging to itself, inconsistencies can easily be the result.

```
// Never return non const references of a class by public member
// functions.
class Account
{
public:
   Account(int aMoney) : mMoneyAmount(aMoney) {};
   const int& getSafeMoney() const { return mMoneyAmount; }
   int& getRiskyMoney() const { return mMoneyAmount; } // No!

private:
   int mMoneyAmount;
};

// I'll try and raise my salary...
Account myAcc(10);

// Error! Cannot assign to const. Too bad - no raise.
myAcc.getSafeMoney() += 1000000;

// myAcc::moneyAmount = 1000010 !
myAcc.getRiskyMoney() += 1000000; // constnes is gone! Kling!
```

5.3 FRIENDS

Operations on an object are sometimes provided by a collection of classes and functions.

A **friend** is a non-member of a class that has access to the non-public members of the class. Friends offer an orderly way of getting around data encapsulation for a class, without totally breaking it. A friend class can be advantageously used to provide functions which require data that is not normally needed by the class.

Iterators of container classes are an example of the use of **friend**s. An iterator provides access to the data in the container class without being member of neither the container nor the stored object, but a **friend** of the container class. This does not break encapsulation since all data members of the container class can be kept **private**, and still there is access to the data inside the container by the **friend** iterator.

Guideline 5.2 Use friend sparingly and provide a detailed design when using it.

Justification friends provide a hole into the strong encapsulation which can useful. However, they add extra complexity which needs to be documented in detail.

Guideline 5.3 Friends of a class should only be used to provide additional functionality that is best kept outside the class.

Rule 5.2 An object must not be a friend of more than one class.

Justification Even if friends are an orderly way to get around strong encapsulation they contradict this elementary principle. Unreflected use of this feature will inevitably lead to confusion an dependencies that - once implemented - can hardly be resolved.

5.4 CONST MEMBER-FUNCTIONS

Rule 5.3 A member function that does not affect the state of an object (its instance variables) is to be declared const.

Justification Member functions declared as const may not modify member data and are the only functions which may be invoked on a const object. (Such an object is inherently unusable without const methods). A const declaration is an excellent insurance that objects will not be modified (mutated) when they should not be. A great advantage that is provided by C++ is the ability to overload functions with respect to their constness.

Rule 5.4 If the behaviour of an object is dependent on data outside the object, this data must not be modified by const member functions.

Justification Actually such data should never occur. Where this might be the exceptional case the data should be treated as if it belonged to the class and thus should not be modified by const member functions.

```
//const-declared access functions to internal data in a class
class SpecialAccount : public Account
{
public:
   void setAmountOfMoney(int aMoney);
   int getAmountOfMoney1(void); // No! Forbids ANY constant object
                                // to access the amount of money.

   int getAmountOfMoney2(void) const;     // Better!
   // ...
private:
   int mMoneyAmount;
};
```

5.5 CONSTRUCTOR AND DESTRUCTOR

Guideline 5.4 Data members of a class should be initialised by using an initialisation list that initialises the data in the order in which they are declared.

Justification The initialisation list is usually the most performant way to initialise data. Keeping the declaration order explicitly document the order that the compiler uses anyway, which prevents the inexperienced programmer to try otherwise or even introduce dependencies in-between the data.

```
Class MyClass
{
public:
   // constructor
   MyClass(int aStatusInfo);
```

```
private:
   int mStatus;
};

// initialisation list
MyClass::MyClass(int aStatusInfo) : mStatus(aStatusInfo)
{
};
```

Rule 5.5 All classes which have virtual functions, must define a virtual destructor.

Justification If a class, having virtual functions but without virtual destructors, is used as a base class, there may be a surprise if pointers to the class are used. If such a pointer is assigned to an instance of a derived class and if delete is then used on this pointer, only the base class' destructor will be invoked. If the program depends on the derived class' destructor being invoked, the program will fail.

```
// Problems, if destructor of polymorphic class is non virtual.
class Base
{
public:
   Base();                    // Default constructor
   ~Base();                   // no virtual destructor!
   // virtual ~Base();        // <- this would have been right.
   virtual void foo() { cout << "Base::foo" << endl; }
};

// Derived overrides foo()
class Derived : public Base
{
public:
   Derived();                 // constructor for private data
   //foo is being overridden
   virtual void foo() { cout << "Derived::foo" << endl; }
private:
   bool IsUpToDate;           // own data
};

Derived d;                    // Derived constructed

void func( Base* pb )
{
   delete pb;                 // explicit delete:
}                             // destructor ~Base() called

func(&d);                     // ~Base() called, but ~Derived()
                              // would have been needed =>
                              // own data of Derived
// are not destroyed!
```

Rule 5.6 A class which uses new to allocate instances managed by the class, must define a copy constructor and an assignment operator. A class that does not need to provide these semantically should declare them private to prevent their use.

Justification A copy constructor is recommended to avoid surprises when an object is initialised using an object of the same type. If an object manages the allocation and de-allocation of an object on the heap (the managing object has a pointer to the object to be created by the class' constructor), only the value of the pointer will be copied. This can lead to two invocations of the destructor for the same object (on the heap), probably resulting in a run-time error.

Assignment operations are not inherited like other operators, causing a very similar situation. The compiler will automatically define an assignment by calling the assignment of each data member which leads to pointers to the same physical address. This will again lead to more than one deallocation if the objects are destroyed or de-rerencing of deallocated memory and thus cause runtime errors.

If a copy or an assignment of such a class does not make sense (e.g. there is all ways only one instance of it during the hole application) it is sensible to declare these operators private. This prevents that someone later tries to call a compiler generated copy-constructor or assignment operator by provoking a compiler error.

```
//Definition of a "dangerous" class not having a copy //constructor
#include <string.h>

class String
{
public:
   String(const char* cp = "");           // Constructor
   ~String();                             // Destructor
private:
   char* sp;
};

// Constructor
String::String(const char* cp) : sp( new char[strlen(cp)] )
{
   strcpy(sp,cp);
}

String::~String()    // Destructor
{
   delete sp;
}

// "Dangerous" String class
void main(void)
{
   String w1;
   String w2 = w1;
   // WARNING: IN A BITWISE COPY OF w1::sp,
   // THE DESTRUCTOR FOR W1::SP WILL BE CALLED TWICE:
   // FIRST, WHEN w1 IS DESTROYED; AGAIN, WHEN w2 IS DESTROYED.
}
```

```
// "Safe" class having copy constructor and default constructor
#include <string.h>
class String
{
public:
   String(const char* cp = "");           // Constructor
   String(const String& sp);              // Copy constructor
   ~String();                             // Destructor
private:
   char* sp;
};

// Constructor
String::String(const char* cp) : sp( new char[strlen(cp)])
{
   strcpy(sp, cp);
}

String::String(const String& aString) :
   sp(new char[strlen(aString.sp)])
{
   strcpy(sp, aString.sp);
}

String::~String()                         // Destructor
{
   delete sp;
}

// "Safe" String class
void main(void)
{
   String w1;
   // SAFE COPY: String::String(const String&) CALLED.
   String w2 = w1;
}
```

Guideline 5.5 Avoid the use of global objects in constructors and destructors..

Justification In connection with the initialisation of statically allocated objects, it is not secure that other static objects will be initialised (for example, global objects). This is because the order of initialisation of static objects which is defined in various compilation units, is not defined in the language definition. There are ways of avoiding this problem, all of which require some extra work (See pg. 255).

```
// The following example shows, why calls to static
// objects within a constructor may fail!
class MyClassA
{
public:
    MyClassA (const int aValue = 0);
    ~MyClassA () {}
public:
```

```
    inline int getValue(void) const { return mValue; }
protected:
    int mValue;
};

class MyClassB
{
public:
    MyClassB ();
    ~MyClassB () {}
public:
    inline int getValue(void) const { return mValue; }
protected:
    int mValue;
};

// It is not clear, wheter rootValue or virtualValue is
// instantiated first.
static const MyClassA rootValue(100);

static MyClassB virtualValue;

MyClassA::MyClassA (const int aValue) : mValue(aValue)
{
}

MyClassB::MyClassB ()
{
    // Dangerous! rootValue migth not be instantiated yet!!
    mValue = rootValue.getValue() + 1000;
}
```

Guideline 5.6 Constructors taking a single argument should carry the explicit keyword.

Justification This type of constructors will be used by the compiler to perform implicit conversions although they are rarely written for this purpose.

Note However see chapter for the use of explicit.

```
// constructor without explicit
String::String(int aSize)
{
   this = new....            // get "size" amount of memory
   current_size = size;      // set size
}
// now the compiler knows a way to
// convert int to myString!

String Name = "Agent" + 007; // forgotten quotes
                             // the compiler uses the converting
                             // constructor to generate my String of
                             // size 007 which was not the intention!
```

```
// better to use explicit
explicit String::String(int aSize)... // see above
{
                                  // now the compiler may not make implicit
                                  // use of the constructor preventing the
                                  // above.
```

5.6 ASSIGNMENT OPERATOR

Rule 5.7 An assignment operator which performs a destructive action must be protected from performing this action on the object upon which it is operating.

Justification A common error is assigning an object to itself (a = a). Normally, destructors for instances which are allocated on the heap are invoked before assignment takes place. If an object is assigned to itself, the values of the instance variables will be lost before they are assigned. This may well lead to strange run-time errors. If a = a is detected, the assigned object should not be changed.

```
// assignment operator checks for self assignment:
MyObj& MyObj::operator=(const MyObj& o)    // assignment operator
{
  if( this != &o)                           // check for self assignment
  {
    delete....                             // destructive action now ok.
    ...
  }
  return *this;
}
```

Rule 5.8 An assignment operator must return a const reference to the assigning object.

Justification If an assignment operator returns “void”, then it is not possible to write a = b = c. It may then be tempting to program the assignment operator so that it returns a reference to the assigning object. Unfortunately, this kind of design can be difficult to understand. The statement (a = b) = c can mean that a or b is assigned the value of c before or after a is assigned the value of b. This type of code can be avoided by having the assignment operator return a const reference to the assigned object or to the assigning object. Since the returned object cannot be placed on the left side of an assignment, it makes no difference which of them is returned (that is, the code in the above example is no longer correct).

```
//Incorrect and correct return values from an assignment operator void
MySpecialClass::operator=
   ( const MySpecialClass& msp ); // Well...?

MySpecialClass& MySpecialClass::operator=
   ( const MySpecialClass& msp ); // No!

const MySpecialClass& MySpecialClass::operator=
   ( const MySpecialClass& msp ); // Recommended

//Definition of a class with an overloaded assignment operator
```

```
class DangerousBlob
{
public:
   const DangerousBlob& operator=( const DangerousBlob& dbr );
   // ...
private:
   char* cp;
};

// Definition of assignment operator
const DangerousBlob&
DangerousBlob::operator=( const DangerousBlob& dbr )
{
   // Guard against assigning to the "this" pointer
   if ( this != &dbr )
   {
      delete cp;          // Disastrous if this == &dbr
   }
}
```

5.7 OPERATOR OVERLOADING

For operator overloading the same things as for function overloading can be applied (see section).

Guideline 5.7 Use operator overloading sparingly and in a uniform manner.

Justification Operator overloading has both advantages and disadvantages. One advantage is that code which uses a class with overloaded operators can be written more compactly (more readably). Another advantage is that the semantics can be both simple and natural. One disadvantage in overloading operators is that it is easy to misunderstand the meaning of an overloaded operator (if the programmer has not used natural semantics). The extreme case, where the plus-operator is re-defined to mean minus and the minus-operator is re-defined to mean plus, probably will not occur very often, but more subtle cases are conceivable.

Rule 5.9 When two operators are opposites (such as == and !=), it is appropriate to define both.

Justification Designing a class library is like designing a language! If you use operator overloading, use it in a uniform manner; do not use it if it can easily give rise to misunderstanding. If the operator != has been designed for a class, then a user may well be surprised if the operator == is not defined as well.

5.8 INHERITANCE

Rule 5.10 Avoid inheritance for “parts-of” relations.

Justification A common mistake is to use multiple inheritance for parts-of relations when an object consists of several other objects, these are inherited instead of using instance variables. This can result in strange class hierarchies and less flexible code. In C++ there may be an arbitrary number of instances of a given type. If inheritance is used, direct inheritance from a class may only be used once.

Guideline 5.8	Give derived classes access to class type member data by declaring protected access functions.
Justification	A derived class often requires access to base class member data in order to create useful member functions. The advantage in using protected member functions is that the names of base class member data are not visible in the derived classes as protection against accidental change. Such access functions should only return the values of member data (read-only access). This is best done by simply invoking const functions for the member data.
Note	The guiding assumption is that those who use inheritance know enough about the base class to be able to use the private member data correctly, while not referring to this data by name. This reduces the coupling between base classes and derived classes.

Rule 5.11	Avoid multiple inheritance from classe with state (member variables). Inheritance from classes (interfaces) with only pure virtual methods (and pure virtual destructor) should be allowed without limits.
Justification	multiple inheritance creates subtle class dependencies tat cause difficulties in the design. When enhancing such code the programmer often finds himself in a dead end which makes reimplementation necessary. For very few exceptional cases multiple inheritance can be an elegant solution, but take into account the difficulties of enhancing such code!
Note	The derivation of a class from more than one base class is called multiple inheritance.

6. EXCEPTIONS

Exception handling is another recent addition to the ISO C++ standard, which is not properly handled in detail by many compilers. Exception handling is a global concept for fault handling, which spreads throughout an entire application and thus requires a thorough design. Therefore it is fatal to start introducing exception handling locally in some classes. The trade off for using the elegant global approach of exception handling (that saves the fuzzy work of setting errors and handing back return codes) is that a fault will also have much more global implications and can be very hard to locate. When faced with the decision whether or not to introduce exception handling for an "isolated" subproject it is essential to investigate whether the isolation in respect to exceptions can indeed be guaranteed. Should an uncaught exception occur it would inevitably lead to termination of the whole application! Before introducing exception handling there should always be a detailed redesign (or design review) of the project in question.

Rule 6.1 Constructors must be examined for their "exception safety" when introducing exceptions.

Justification Clumsy memory allocation in constructors can lead to memory leaks if an exception is thrown.

```
// insecure class
class Y
{
   int* p;
   void init();
public:
   Y(int s){ p = new int[s]; init(); }
   ~Y { delete [] p; }
...
}
// when while calling the constructor init() throws an exception
// the memory allocated for 'p' will not be freed, since the
// constructor was not fully executed => memory leak.

// A safe variant of the above:
class Z
{
   vector<int> p;
   void init();
public:
   Z(int s) : p(s) { init(); }
...
}
// The memory used by 'p' is now handled by 'vector.
// If init() throws an exception, the memory acquired will be freed
// when the destructor for 'p' is (implicitly) invoked.
```

7. DYNAMIC MEMORY

The concept of dynamic memory management in C++ differs from that in C. Objects manage the storage of their data by allocating and releasing memory with the operators new and delete, which they invoke in their constructor or destructor. This encapsulation keeps the user of a class from having to worry about memory allocation since it's already built into the class (or at least it should be). This chapter mentions possible traps in this context.

Rule 7.1 Only use **new** and **delete,** forbidden **malloc**, **realloc** or **free**.

Justification In C malloc, realloc and free are used to allocate memory dynamically on the heap. This may lead to conflicts with the use of the new and delete operators in C++.

It is dangerous to:

1. invoke delete for a pointer obtained via malloc/realloc,
2. invoke malloc/realloc for objects having constructors,
3. invoke free for anything allocated using new.

Thus, avoid whenever possible the use of malloc, realloc and free.

Rule 7.2 Always provide empty brackets ("[]") for **delete** when deallocating arrays.

Justification If an array a having a type T is allocated, it is important to invoke delete in the correct way. Only writing delete a; will result in the destructor being invoked only for the first object of type T (resource leak). By writing delete [m] a; where m is an integer which is greater than the number of objects allocated earlier, the destructor for T will be invoked for memory that does not represent objects of type T. The easiest way to do this correctly is to write delete [] a; since the destructor will then be invoked only for those objects which have been allocated earlier.

```
//  Right and wrong ways to invoke delete for arrays with
//  destructors
int n = 7;

// T is a type with defined constructors and destructors
T* myT = new T[n];

// ...
delete myT;                  // No! Destructor only called for first
                             // object in array a

// No! Destructor called on memory out of bounds in array a
delete [10] myT;

delete [] myT;               // OK, and it is always safe!
```

Guideline 7.1 Whoever allocates memory is responsible for its deallocation or do it by convention.

Justification One major advantage of C++ over C is the possibility to encapsulate the memory management into the object that needs it. This means that allocation should take place in the constructor of a class and deallocation is done in the destructor. In this way one can be sure that resources are freed after the lifetime of an object expires.

```
// resource leak resulting from bad memory management
String myFunc(const char* aArgument)
```

```
{
   String* temp = new String(aArgument);
   return *temp;              // temp is never deallocated.
                              // A user of myFunc cannot free the
                              // storage because he only gets a copy.
                              // DON'T do this!
}
```

Rule 7.3 Always assign a NULL value to a pointer that points to deallocated memory.

Justification Pointers that point to deallocated memory should either be set to NULL or be given a new value to prevent access to the released memory. This can be a very difficult problem to solve when there are several pointers which point to the same memory, since C++ has no garbage collection.

Rule 7.4 Regarding Memory allocation.

To confirm that memory has been allocated properly use NEW_M macro (new (nothrow)) for memory allocation and check the return value against NULL. If return error code is NULL is returned. (Refer and use di_projects\linked\ai_mv\infotainment\Common\inc\common.h). In the same way use DEL_M and DELARR_M for delete operations. OSAL_NEW should only be used for Service and client handler classes. Rest all other classes should use Linux new (NEW_M) .

Rule 7.5 Regarding Dynamic / static Memory allocation.

Use dynamic allocation than the static allocation especially for class object. (i.e. declare class pointer and then allocate memory dynamically than static allocation of the memory). Ensure that dynamically allocated memory is freed once its not required.

8. PRE-PROCESSOR

The pre-processor handles text substitutions. It is the same pre-processor that is known from C. Among the typical tasks are cutting away comments, substituting constants defined with the **#define** directive, and selecting text to be parsed by evaluating **#if**, **#else**, and **#endif** directives. There is no type checking. Implementing complex pre-processor constructs often leads to misunderstandings of errors.

All rules and guidelines pertaining to use of the preprocessor are described in [PF91 C] and apply to C++ as well.

9. TEMPLATES FOR HEADER AND SOURCE FILES

9.1 FOR HEADER FILE

9.2 FOR SOURCE FILE

To be made available

10. SUMMARY

10.1 LIST OF RULES

10.2 LIST OF GUIDELINES

11. LITERATURE

[Lit 1] Stroustrup, B.
The C++ programming language, third edition
Addison-Wesley, 1998

[Lit 2] Ellemtel
Programming in C++ Rules and Recommendations
Ellemtel, 1992

[Lit 3] Meyers, S.
Effective C++, second edition
Addison-Wesley, 1997

[Lit 4] Meyers, S.
More Effective C++
Addison-Wesley, 1995

[Lit 5] Goldsmith, D.
Unofficial C++ style guide.
Magazine "Develop", April 1990