

C++ PROGRAMMING TECHNIQUE

Technical Guide -- Standard

Authors: Alexander Kozlinski

Last updated: 25 August 2003

Copyright © 2003 Alexander Kozlinski.

Permission is granted to use, copy, modify, merge, distribute, sublicense, publicly display, publicly and/or digitally perform, publish and/or sell copies of this document under the terms of the UMP end—user license agreement (UMP—EULA). A copy of the UMP—EULA is included in the section entitled " UMP end—user license agreement (UMP—EULA)".

Contact information: Alexander V. Kozlinski, <mailto:alexkozlinski@uni-mod.com>

Contents

1. INTRODUCTION	3
1.1. Purpose	3
1.2. Scope	3
1.3. Terminology	3
2. Application Structuring	4
2.1. Application component	7
2.2. File layout	7
2.3. Component main *.cpp File Template	8
2.4. Class *.cpp File Template	9
2.5. Assistant *.cpp File Template	10
2.6. Project (application) wide Header File Template	11
2.7. Component wide Header File Template	12
2.8. Class Header File Template	13
2.9. Assistant Functions Header File Template	14
2.10. Version control directives	14
3. Project directories	15
4. Naming	15
4.1. General	15
4.2. Standard names	16
4.3. Prefixes	16
4.4. Suffixes	17
4.5. Formats	17
4.6. Resources Naming and Numbering	19
5. Comments	20
5.1. Comments' Content	20
5.2. Comments formatting	21

6. Code Layout	22
6.1. Braces and Parenthesis	22
6.2. Space	23
6.3. Wrapping	23
7. Program Structure	23
7.1. Types	23
7.2. Type Definitions	23
7.3. Declarations	23
7.4. Initialization	24
7.5. Constants	24
7.6. Pointers and references	25
7.7. Storage Class	26
7.8. Classes	26
7.8.1. Class Declaration	26
7.8.2. Class definition	29
7.8.3. Constructors	30
7.8.4. Inheritance	30
7.9. Member Functions	31
7.9.1. Function declaration	31
7.9.2. Function definition	32
7.9.3. Returns	33
7.9.4. Inline functions	34
7.10. Assistant functions	34
7.11. Templates and Template Functions	34
7.12. Statements	34
7.12.1. if Statement	35
7.12.2. switch Statement	37
7.12.3. Loop Constructs	37
7.12.4. while statement	38
7.12.5. for Statement	38
7.12.6. goto Statement	38
7.13. Expressions	38
7.13.1. Unary Expressions	38
7.13.2. Binary Expressions, Conditional Expressions and Assignments	39
7.13.3. Precedence	39
7.13.4. Type casting	39
7.14. Operations with Objects	39
8. Defensive Programming	40
9. Preprocessor	40
9.1. Preprocessor Macros	40
10. Preprocessor Directives	40
11. Exceptions	41
12. Library Functions	41
13. Miscellaneous	42
14. References and resources	42
UMP end—user license agreement (UMP—EULA)	43

1. INTRODUCTION

*The great code is readable. By you. By other members of the team.
And by those who will read your work in the future.*

1.1. Purpose

The purpose of this standard is to increase the ready understanding of the software code through phases of its development: design, coding, testing, quality, migration, and support. This increased understanding of the software code is directly related to the efficiency and productivity of the software engineers who support and reuse the software product.

1.2. Scope

This document describes the coding styles and formatting standards for software developed in the C/C++ language for the MS Windows 95 and NT.

The readability of the software is a requirement since, most of a software engineer's time is spent maintaining and enhancing existing software.

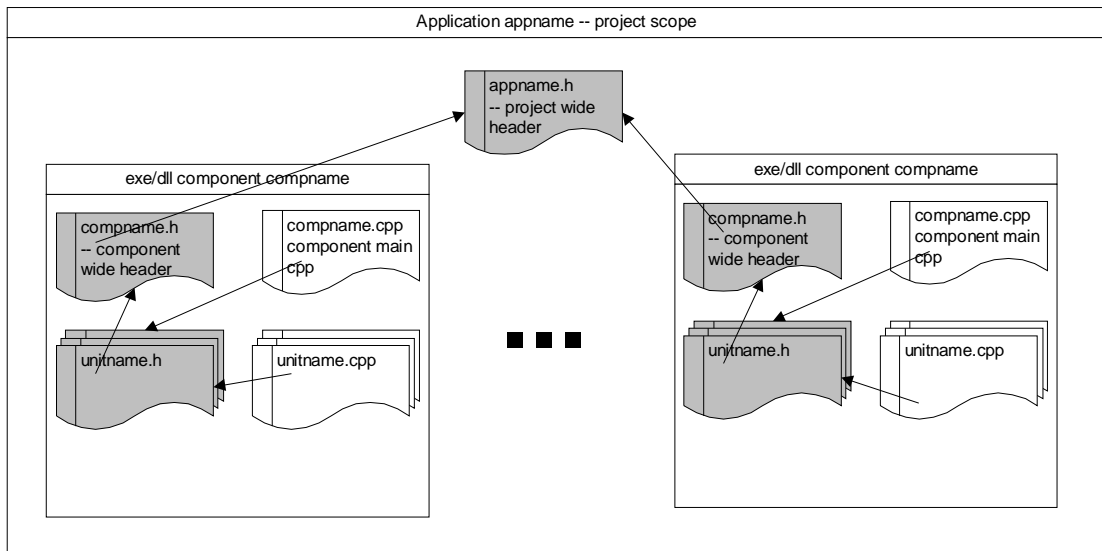
The reliability of the software is a requirement that is enhanced by the introduction of a standard to follow. Many of the rules to the standard protect the programmer from the problems that exist within the language.

1.3. Terminology

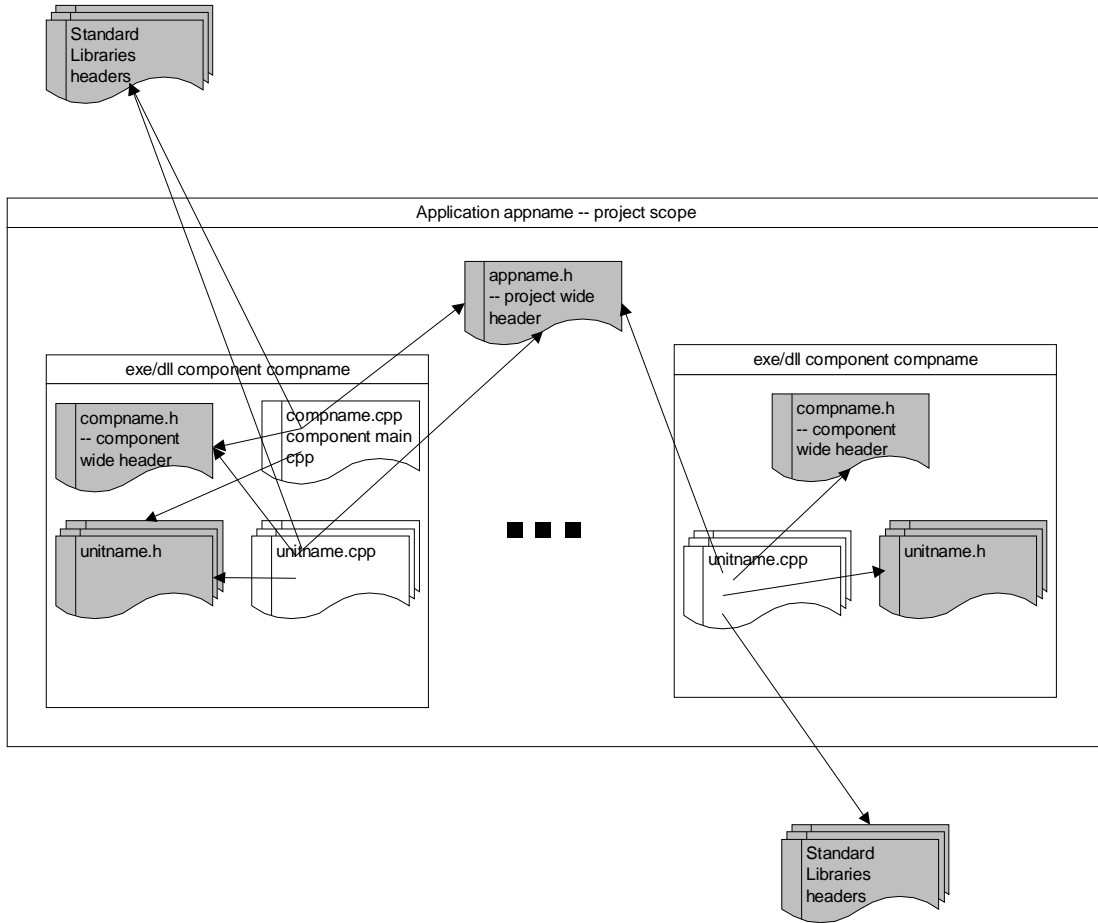
<i>Mandatory</i>	Refers to any action or language element whose use is mandatory. There can be no mitigating reasons for non-compliance.
<i>Forbidden</i>	Refers to any action or language element whose use is mandatory not allowed under any circumstance. There are no mitigating reasons for use. Used in defining standards.
<i>Recommended</i>	Refers to any action or language element whose use is expected within the parameters described. Sufficient rationale is required in cases of non-use. Used in defining style guidelines.
<i>Restricted</i>	Refers to any action or language element whose use is allowed if and only if the designer can present sufficient rationale. Sufficient rationale is considered to be cases of clarity or performance. Used in defining standards.

2. Application Structuring

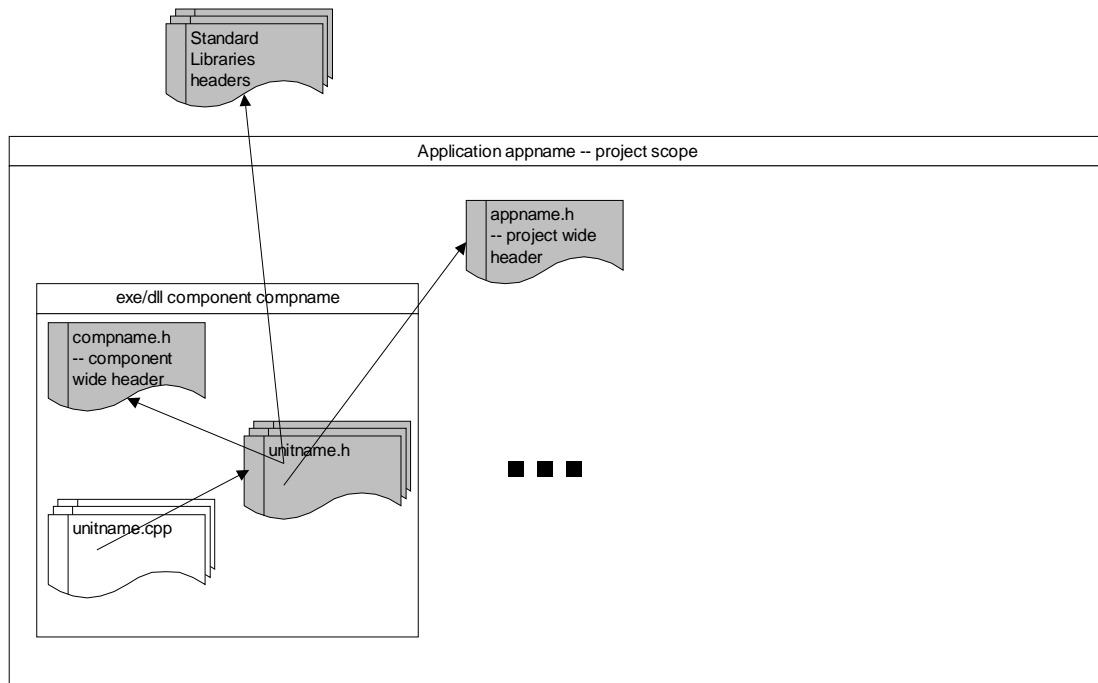
Variant #1: nesting headers



Variant #2: NO nesting headers



Variant #3: ONE nesting headers level



Unit is:

- Main function definition
- single class definition (or part of single class)
- assistant functions [Stroustrup 99] definitions functionally organized.

Unit is comprised of header (*.hpp) and definition files (*.cpp). Header files should be treated as interface definition files.

Project wide header file:

- Declare entities that are likely to be used together in a most units of all components

Special header files:

- Declarations is likely to change when code is ported from one machine to another
- Declarations are functionally related, but not related to specific class or assistant functions

Mandatory	There is no nesting in header files: variant #2.
Mandatory	Source of Class with a large number of functions should break the implementation into several .CPP files . Files should be functionally cohesive: should be split up along boundaries between class administrator, accessor, mutator and provider functions. A class shall have a single header file

2.1. Application component

Mandatory	Name files like variables, describing the functions they contain. Long file names are encouraged.
Mandatory	Executable .exe and library files (*.dll, *.lib, others) shall be named as component name, which they implement: <code><Component name>.xxx</code>
	<p>Unit naming:</p> <p>Main: <code><Component name>Main.cpp</code> <code><Component name>Main.hpp</code></p> <p>Class <code><ClassName>[N].cpp</code> <code><ClassName>.hpp</code> Where N is # of class source files.</p> <p>Assistant Functions: <code><Library/group name>.cpp</code> <code><Library/group name>.hpp</code> Class assistant functions are included in Class sources</p>

2.2. File layout

	Keep function length in code files to within one or two pages (100 lines).
1	Module file length is not longer than about 1000 lines.
2	<p>Line length in program files less than 78 characters</p> <p>Rationale: When the audience for the source and headers of a project may reach thousands, if not more, readability and continuity become prime factors for comprehension. Additionally, in a multi-developer environment, the potential for disjoint style is personified with no restrictions on column length.</p>

2.3. Component main *.cpp File Template

```
//*****
// DESCRIPTION: Main function of the application (component)
// PROJECT:      project name
// APPLICATION:  appname
// COMPONENT:    compname / type (exe, dll)
// CREATED BY:
// DATE CREATED:
// MODIFIED BY:
// DATE MODIFIED:
//*****
// Copyright © 1991-2001 by Alexander Kozlinski. All Rights Reserved
//*****

    //C++ standard library
#include <cstdlib.h> //Definitions for common types, variables, functions
#include <cstring.h> //Global string functions of C
    //MS Windows specific
#include <> //
    //!! Try to avoid compiler Borland C++ specific
    //External libraries, classes and assistant functions
#include ""
#include "compmain.h"

-----//Data Definitions
int a;

-----//Agregates Definitions
short table[]={1,2,3};

-----//Export Template Definitions
export template<class T> f(T t){/*-----*/};

<class definitions>

<function definitions>

void main()
{
}
}
```

2.4. Class *.cpp File Template

```
/* *****  
// CLASS NAME:   class name  
// PROJECT:     project name  
// APPLICATION: appname  
// COMPONENT:   compname / type (exe, dll)  
// CREATED BY:  
// DATE CREATED:  
// MODIFIED BY:  
// DATE MODIFIED:  
*****  
// Copyright © 1991-2001 by Alexander Kozlinski. All Rights Reserved  
*****  
*/  
  
    //C++ standard library  
#include <cstdlib.h> //Definitions for common types, variables, functions  
#include <cstring.h> //Global string functions of C  

```

2.5. Assistant *.cpp File Template

```
/* *****  
// NAME:          library or assistant functions group name  
// DESCRIPTION:  
// PROJECT:       project name  
// APPLICATION:   appname  
// COMPONENT:    compname / type (exe, dll)  
// CREATED BY:  
// DATE CREATED:  
// MODIFIED BY:  
// DATE MODIFIED:  
*****  
// Copyright © 1991-2001 by Alexander Kozlinski. All Rights Reserved  
*****  
*/  
  
    //C++ standard library  
#include <cstdlib.h> //Definitions for common types, variables, functions  
#include <cstring.h> //Global string functions of C  

```

2.6. Project (application) wide Header File Template

```
#ifndef __APPNAME_HPP__
#define __APPNAME_HPP__
/*****
// PROJECT: project name
// PROJECT DESCRIPTION: <common>
// APPLICATION: appname
// APPLICATION DESCRIPTION: <common>
// HEADER DESCRIPTION: This header file describes the externally-visible
//                       facilities of the <application name>
// CREATED BY:
// DATE CREATED:
// MODIFIED BY:
// DATE MODIFIED:
*****/
// Copyright © 1991-2001 by Alexander Kozlinski. All Rights Reserved
*****/
*/

    //version definitions - check with the Borland version info
const int APP_MAJOR_VERSION = 1;
const int APP_MINOR_VERSION = 0;
enum AppVersion
{
    APP_MAJOR_VERSION = 1
    ,APP_MINOR_VERSION = 0
};

    //namespaces
namespace N{ /* */}

    //Types Definitions
struct ...
typedef ...
class ...

    //Templates Declarations
template <class T> class Z;

    //Templates Definitions
template <class T> class V{ /* */};

    //Functions Declarations
extern int strlen(const char*);

    -----//Inline Functions Definitions
inline char get(char* p) {return *p++};

    //Data Declarations
extern int a;

    //Constants & enums Definitions
const float pi = 3.141593
enum Light
{
    RED,
    YELLOW,
    Green
};
#endif /* __APPNAME_HPP__ */
```

2.7. Component wide Header File Template

```
#ifndef __COMPNAME_HPP
#define __COMPNAME_HPP
/*****
//          This header file describes the externally-visible
//          facilities of the <component name>
// PROJECT:   project name
// APPLICATION: appname
// COMPONENT: compname / type (exe, dll). Description
// CREATED BY:
// DATE CREATED:
// MODIFIED BY:
// DATE MODIFIED:
*****/
// Copyright © 1991-2001 by Alexander Kozlinski. All Rights Reserved
*****/
*/
    //namespaces
namespace N{ /* */}

    //Types Definitions
struct ...
typedef ...
class ...

    //Templates Declarations
template <class T> class Z;

    //Templates Definitions
template <class T> class V{ /* */};

    //Inline Functions Definitions
inline char get(char* p) {return *p++;}

    //External references
extern int strlen(const char*);
extern int a;

    //Constants & enums Definitions
const float pi = 3.141593
enum Light
{
    RED,
    YELLOW,
    Green
};
#endif /* __COMPNAME_HPP */
```

2.8. Class Header File Template

```
#ifndef __UNITNAME_HPP__
#define __UNITNAME_HPP__
/*****
// CLASS:      name and short description of the purpose of the entities
// PROJECT:    project name
// CREATED BY:
// DATE CREATED:
// MODIFIED BY:
// DATE MODIFIED:
*****/
// Copyright © 1991-2001 by Alexander Kozlinski. All Rights Reserved
*****/
*/

    //namespaces
namespace N{/// */}

    //Types Definitions
struct ...
typedef ...
class ...

    //Templates Declarations
template <class T> class Z;

    //Templates Definitions
template <class T> class V{ /* */};

    ———— //Inline Functions Definitions
inline char get(char* p) {return *p++};

    //External references
extern int strlen(const char*);
extern int a;

    //Constants & enums Definitions
const float pi = 3.141593
enum Light
{
    RED,
    YELLOW,
    Green
};

    //Class Declarations

    //Assistant functions declarations
#endif /* __UNITNAME_HPP__ */
```

2.9. Assistant Functions Header File Template

```

#ifndef __UNITNAME_HPP__
#define __UNITNAME_HPP__
/*****
// LIBRARY:      name, and description of the purpose of the entities
// PROJECT:      project name
// CREATED BY:
// DATE CREATED:
// MODIFIED BY:
// DATE MODIFIED:
*****/
// Copyright © 1991-2001 by Alexander Kozlinski. All Rights Reserved
*****/
*/

    //namespaces
namespace N{ /*    */}

    //Types Definitions
struct ...
typedef ...

    //Templates Declarations
template <class T> class Z;

    //Templates Definitions
template <class T> class V{ /*    */};

—————//Inline Functions Definitions
inline char get(char* p) {return *p++};

    //External references
extern int strlen(const char*);
extern int a;

    //Constants & enums Definitions
const float pi = 3.141593
enum Light
{
    RED,
    YELLOW,
    Green
};

    //Assistant functions declarations
#endif /* __UNITNAME_HPP__ */

```

2.10. Version control directives

```

$Header: <path>/<file_name>.p_v Revision Date Programmer_ID $
REVISION LOG
$Log: <path>/<file_name>.p_v $
Rev 1.1    <Date Time> Programmer_ID
The reasons of revision
Rev 1.0    <Date Time> Programmer_ID

```

3. Project directories

	<p>Project source directory structure:</p> <pre> \\<project name> ... \src \base -- contains the basic (core) functionality for the application. Base contains all the source code that cannot be categorized into a component \<component 1> \base \build -- contains any special makefiles for building the particular component \doc -- contains any documentation associated with this component AND not included in project wide SRS or SDD directories ... \<component N> \<subsystem name> </pre>
	<p>Every directory should have a README file that covers: the purpose of the directory and what it contains anything else that might help someone</p> <p>Rationale: Consider a new person coming in 6 months after every original person on a project has gone. That lone scared explorer should be able to piece together a picture of the whole project by traversing a source directory tree and reading README files, Makefiles, and source file headers.</p>

4. Naming

4.1. General

Mandatory	Use exactly the same names as in the object model.
Mandatory	Simple, descriptive, meaningful, exact names in English used
Mandatory	Name's meaning is preserved throughout the project
Mandatory	Similar names used for variables that perform similar functions
Mandatory	Most meanings have multiple words. Pick one word for one abstract function and stick with it. Rationale: For instance, it's confusing to have fetch, retrieve and get as same-acting methods of the different classes. Likewise, it's confusing to have a controller and a manager and a driver in the same process. The name leads you to expect two objects that have very different type as well as having different classes. [Ottinger 97]
Forbidden	Avoid words which already mean something else [Ottinger 97]
Forbidden	Renaming of any standard library or constants' names are forbidden
Forbidden	Use of the names of existing libraries functions constants is forbidden

Forbidden	Use names that differ only in case is forbidden. Don't use variable names that differ by only one or two characters. Use of l (lower-case L) or O (upper-case O) as names and all their derivatives such as ooO
Allowed	local or frequently used identifiers can be short; uncommon used should be long and meaningful. Use of short names (such as x, y, and i) is restricted by cases when: <p style="text-align: center;">their meaning is clear and when a longer name would not add information or clarity</p>
Restricted	The following names are also reserved by ANSI for its future expansion: <div style="border: 1px solid black; padding: 5px; margin: 5px 0;"> <pre> E[0-9A-Z][0-9A-Za-z] * errno values is[a-z][0-9A-Za-z] * Character classification to[a-z][0-9A-Za-z] * Character manipulation LC_[0-9A-Za-z_] * Locale SIG[_A-Z][0-9A-Za-z_] * Signals str[a-z][0-9A-Za-z_] * String manipulation mem[a-z][0-9A-Za-z_] * Memory manipulation wcs[a-z][0-9A-Za-z_] * Wide character manipulation </pre> </div> <p>Note that the first three namespaces are hard to avoid. In particular, many accessor methods naturally fall into the is* namespace, and error conditions map onto the E* namespace. Be aware of these conflicts and make sure that you are not re-defining existing identifiers [Gabryelski 97]</p>

4.2. Standard names

Mandatory	Project and problem domains standard names and abbreviations shall be <ul style="list-style-type: none"> • Defined and commented • used consistently through the project
Mandatory	abbreviations must be defined and

Abbreviation	Meaning
col	column index
row	row index
i, j	general counters
max	maximum
min	minimum
dsc	descriptor
msg	message string

4.3. Prefixes

Prefix	Description
ak	class
p	pointer
pp	general pointer to pointer
r	Reference

Allowed	В проекте могут быть введены специфические префиксы. Обязательное ус-
----------------	---

	ловие - их согласованное использование во всем проекте
--	--

4.4. Suffixes

Suffix	Description
<code>_ST</code>	static
<code>_FUNC</code>	function address

4.5. Formats

Forbidden	Developer-created names with leading and trailing underscores -- reserved for system purposes: <code>_something</code> <code>_something_</code> <code>something_</code>
------------------	--

Object	Format	Comment
Function	[p]FunctionName	<ul style="list-style-type: none"> • Verb-noun shall be used to describe void function: <pre style="margin-left: 40px;">void GetData(data variable)</pre> <p style="margin-left: 40px;">Accessor methods start with the word 'get' and should be const</p> • Mutator procedures start with the word 'set' and don't return values. <div style="border: 1px solid black; padding: 5px; margin: 10px 0;"> <pre>class Foo { public: // // Accessor // StringCref getName(void) const; bool isEmpty(void) const; // // Mutator // void setName(StringCref aName); protected: String theName; };</pre> </div> • Factory instantiation functions start with the word 'create'. <pre style="margin-left: 40px;">pThread createThread(void);</pre> • Factory destruction functions start with the word 'destroy'. <pre style="margin-left: 40px;">void destroyThread(ThreadPtr);</pre> • Functions which return value shall be named following returned value name: <pre style="margin-left: 40px;">string CustomerID()</pre> <ul style="list-style-type: none"> • Adjectives shall be used to describe Boolean functions: <pre style="margin-left: 40px;">bool IsValid()</pre>

Object	Format	Comment
typedef	TypeName	<p>Avoid predefined types:</p> <pre> // Bad double x = 0.0 double y = 0.0 // Good typedef Coordinate double; ... Coordinate x = 0.0 Coordinate y = 0.0 </pre>
variable	[p]variableName[s]	<p>P - <pointer> s - <suffix> variable_name - lower case with : nounAbstractNoun, e.g.: pWallHeight Noun shall be functionally oriented:</p> <pre> longString //Bad safetyInstruction //Good </pre>
class	akClassName	
enum	<pre> enum EnumNname { VALUE1 VALUE2 ... }; </pre>	<p>named constants in upper and identifier ???:</p>
Constant	CONST_NAME	upper-case

4.6. Resources Naming and Numbering

Resource ID naming should follow the [TN020: ID Naming and Numbering Conventions] format:

R<TN020>UPPERCASE_NAME	Company-wide IDs
a<TN020>UPPERCASE_NAME	Application specific IDs

for example RIDP_MESSAGE_NAME for messages.

Numbering conventions:

Prefix	Resource Type	Company Wide	Application Specific
IDR_	multiple	0x5001 -> 0x6FFF	0x1001 -> 0x4FFF
IDD_	dialog templates	0x5001 -> 0x6FFF	0x1001 -> 0x4FFF
IDC_, IDI_, IDB_	cursors, icons, bit-maps	0x5001 -> 0x6FFF	0x1001 -> 0x4FFF

IDS_, IDP_	general strings	0x5001 -> 0x7FFF	0x1001 -> 0x4FFF
ID_	commands	0xC001 -> 0xDFFF	0x8001 -> 0xBFFF
IDC_	controls	0xC001 -> 0xDFFF	0x1001 -> 0xDFFF

5. Comments

5.1. Comments' Content

Mandatory	Pseudocode from the SDD is incorporated into the source code file
Mandatory	are commented: <ul style="list-style-type: none"> • Any restrictions • Necessary complexity
Forbidden	There are no comments that are clear from the code
Forbidden	There are no unnecessary comments, requiring maintenance
Recommended	<p>Frequently there is a need to leave reminders in the code about uncompleted work or special cases that are not handled correctly. These comments should be of the form:</p> <pre style="border: 1px solid black; padding: 5px;"> // !<gotcha>! <author> <date>: When we can, replace this // code with a wombat // </pre> <p>Rationale: This gives maintainers some idea of whom to contact. It also allows one to easily <code>grep</code> the source looking for unfinished areas [<i>Gabryelski 97</i>]</p> <p>Gotcha Keywords</p> <p>!TODO! topic -- Means there's more to do here, don't forget.</p> <p>!BUG! [bugid] topic -- means there's a Known bug here, explain it and optionally give a bug ID.</p> <p>!KLUDGE! When you've done something ugly say so and explain how you would do it differently next time if you had more time.</p> <p>!TRICKY! -- Tells somebody that the following code is very tricky so don't go changing it without thinking.</p> <p>!WARNING! -- Beware of something.</p> <p>!COMPILER! -- Sometimes you need to work around a compiler problem. Document it. The problem may go away eventually.</p> <p>!ATTRIBUTE! value -- The general form of an attribute embedded in a comment. You can make up your own attributes and they'll be extracted.</p> <p><i>[Todd Hoff OO]</i></p>

Recommended	<p>For class, functions block headers – consider to put the following information:</p> <pre> // PRECONDITION: Document what must have happened for // the object to be in a state where the // method can be called // WARNING: Document anything unusual users should // know about this method. // LOCK REQUIRED: Some methods require a semaphore // be acquired before using the method. // When this is the case use lock required // and specify the name of the lock. // EXAMPLES: Include examples of how to use a method. // A picture says a 1000 words, a good // example answers a 1000 questions. </pre>
--------------------	---

5.2. Comments formatting

Mandatory	C++ style comments only -- //
Mandatory	<p>Comments alone on a lines are indented on 2SI of the code that follows:</p> <pre> //This comment describes the following code SomeUsefulCode(); </pre>
Mandatory	<p>Use multi-line comments in aligned blocks, interchanging with the code:</p> <pre> //This comment describes the following code //this is the second line of the comment //this is the third line of the comment SomeUsefulCode(); </pre>
Mandatory	<p>Use column (or “tabled”) comments:</p> <pre> int x; //describe what x does const double precision; //describe what precision does int (*pfi)(); //describe what pfi does int z; //describe what z does const char* variable; //describe what variable does x = 10; //put here comment the_variable = x; //put here 2nd comment z = x; //put here 2nd comment </pre>
Mandatory	Use blank lines before and after block comments.

6. Code Layout

6.1. Braces and Parenthesis

Mandatory	Braces shall be aligned using Ulman style where the braces are at the same scope as the statement that proceeds them and the code within the braces is indented one level. Braces are always on a line by themselves.:		
	<table border="1"><tr><td data-bbox="477 501 895 904"><p><i>OK</i></p><pre>void DoSomething(void) { if(x != y) { y = x; } else { ; // do nothing } }</pre></td><td data-bbox="932 501 1482 1845"><p><i>NOT</i></p><pre>void DoSomething(void) { if(x != y) { y = x; // should be indented } else { ; // do nothing } }</pre><p>OR</p><pre>void DoSomething(void) { if(x != y) { // brace is not // the same scope y = x; } else { ; // do nothing } }</pre><p>OR</p><pre>void doSomething(void) { if(x != y){ // brace is not // on a line by // itself y = x; } else{ ; // do nothing } }</pre></td></tr></table>	<p><i>OK</i></p> <pre>void DoSomething(void) { if(x != y) { y = x; } else { ; // do nothing } }</pre>	<p><i>NOT</i></p> <pre>void DoSomething(void) { if(x != y) { y = x; // should be indented } else { ; // do nothing } }</pre> <p>OR</p> <pre>void DoSomething(void) { if(x != y) { // brace is not // the same scope y = x; } else { ; // do nothing } }</pre> <p>OR</p> <pre>void doSomething(void) { if(x != y){ // brace is not // on a line by // itself y = x; } else{ ; // do nothing } }</pre>
<p><i>OK</i></p> <pre>void DoSomething(void) { if(x != y) { y = x; } else { ; // do nothing } }</pre>	<p><i>NOT</i></p> <pre>void DoSomething(void) { if(x != y) { y = x; // should be indented } else { ; // do nothing } }</pre> <p>OR</p> <pre>void DoSomething(void) { if(x != y) { // brace is not // the same scope y = x; } else { ; // do nothing } }</pre> <p>OR</p> <pre>void doSomething(void) { if(x != y){ // brace is not // on a line by // itself y = x; } else{ ; // do nothing } }</pre>		

6.2. Space

Mandatory	Use spaces, not tabs. Rationale: Tab sizes vary between developers. When spaces are used, the alignment is maintained no matter where the <code>_le</code> is edited.
Mandatory	Do not space before separators (semicolon, argument comma separator) but do space the other side.
Mandatory	Commas and semicolons have one space (or new line) after them
Mandatory	Keywords (<i>if, while, for, switch, return</i>) are followed by one space

6.3. Wrapping

Mandatory	When wrapping lines, indent the continuation line past the current indent column. <pre> Int val(2); cout << "This is an example where I wrap " << val << " lines of code" << endl; </pre>
------------------	---

7. Program Structure

Forbidden	There is no complex nesting (normal maximum is of three levels, deeper excursion should be short and infrequent)
Mandatory	Each new level of logic is indented

7.1. Types

Forbidden	<code>char*</code> - use standard type <code>string</code> instead [Stroustrup 99]
Forbidden	Unions
Forbidden	Bit structures
Mandatory	types: <code>short int</code> , <code>long int</code> : use <code>int</code> instead [Stroustrup 99]
Mandatory	types: <code>float</code> , <code>long double</code> : use <code>double</code> instead [Stroustrup 99]
Mandatory	types: <code>signed char</code> , <code>unsigned char</code> : use <code>char</code> instead [Stroustrup 99]
Mandatory	arrays; try to use standard <code>vector<T></code> , <code>valarray<T></code> , <code>list<T></code> , <code>map<T></code> instead <code>T []</code> [Stroustrup 99]

7.2. Type Definitions

Mandatory	<i>Use of typedef over</i> <code>struct Tag {...};</code>
Mandatory	<i>Typedef'ed</i> structures have the same struct and the typedef name

7.3. Declarations

Mandatory	All data should be defined as close as possible to where it is needed.
------------------	--

Mandatory	Declarations shall be written with only one variable per source line. Unrelated declarations, even of the same type, are on separate lines
Mandatory	Use vertical alignment to ease scanning of declarations. <pre>String aStringToUse; Int anInt; Real aRealNumberToUse;</pre> <p>instead of</p> <pre>String aStringToUse; Int anInt; Real aRealNumberToUse;</pre>
Mandatory	All names are explicitly declared / There are no implicit type declarations
Mandatory	Do not declare variable until there is a value for it initialization [Stroustrup, 99]
Forbidden	There are no declarations that override declarations at higher levels. In particular, local variables are not redeclared in nested blocks, names are not re-defined in inner blocks
Forbidden	Type definitions and declarations inside different declarations are not used
Forbidden	Nesting level of declarations is not used
Mandatory	After each declaration a comment that describes it is included
Mandatory	Each <i>data item</i> is commented <ul style="list-style-type: none"> - meaning (definition) - definition range -- units of measure
Mandatory	Declarations that are likely to change when code is ported are grouped together in a separate header file and commented

7.4. Initialization

Mandatory	Any <i>variables</i> are <i>explicitly</i> initialized:
Mandatory	Initialize all variables at the time they are declared to the appropriate value. If the value is not yet known, initialize pointers to NULLPTR and simple types to zero.
Forbidden	Forbidden: Initialization <code>char</code> , <code>string</code> with 0: <pre>char symbol = 0; string name = 0;</pre>
Mandatory	Initializers of structures and arrays are formatted with braces around each subaggregate, one row per line: <pre>static int x[2][5] = { {1, 2, 3, 4, 5}, {6, 7, 8, 9, 10}. };</pre>

7.5. Constants

Mandatory	Use <code>const</code> to define constants.
------------------	---

Recommended	<p>The enum data type is the preferred way to handle situations where a variable takes on only a discrete set of values because of the added type checking done by the Compiler [Gabryelski 97]:</p> <pre> const int Red = 0; // Bad Form const int Blue = 1; const int Green = 2; enum ColorComponent // Much Better, Explicit values can be given { RED = 0x10, // to each item as well... BLUE = 0x20, GREEN = 0x40 }; </pre>
Forbidden	<p>Use of constants that are coded directly (e.g. if (index < 100)) is forbidden: Only const declaration shall be used. (In certain places, 0 and 1 may appear as themselves.)</p>
Allowed	<p>Well recognized constants, such as 0, 1, and -1, can often be used directly. For example if a for loop iterates over an array, then it is reasonable to code:</p> <pre> for (i = 0; i < size; i++) { // statements using array[i]; } </pre>
Mandatory	<p>Masks (bit-and) are mandatory used to produce a specific number of bits. A program not relies on data size (or casts) to truncate expressions to a specific number of bits</p>
Mandatory	<p>Hexadecimal constants start with 0x. (lower-case x only). Upper-case A through F is used to construct hexadecimal constants:</p> <p style="text-align: center;">0x00F, 0x1A3</p>
Mandatory	<p>double constants have at least one digit on either side of the decimal point</p>
Mandatory	<p>The exponent in a double constant is a lower-case e. This is followed by a sign:</p> <p style="text-align: center;">1.2e+10</p>
Forbidden	<p>There are not dependencies upon the order of bytes within an integral or floating constant</p>
Forbidden	<p>NULL usage: 0 shall be used [Stroustrup, 99]</p>
Mandatory	<p>If the value of a constant is other than a single number, it is enclosed in parentheses</p>
	<p>Wherever possible, sizes should be expressed in terms of the sizeof operator. For example, if an array's size is determined by its initializers, the proper construct for determining the number of elements it has is:</p> <pre> double factors[] = { 0.1345, 123.23451, 0.0 }; const int FactorsSize = sizeof factors / sizeof factors[0]; </pre>

7.6. Pointers and references

Forbidden	<p>Do not use spaces in object de-references.</p> <pre> val = *pFoo; // ok val = * pFoo; // wrong </pre>
------------------	--

Mandatory	The "pointer" qualifiers, "*", are with the variable name <pre>char *s; // ok char* s; // wrong</pre>
Mandatory	There are no pointer conversions
Forbidden	<code>void*</code> [Stroustrup, 99]
Forbidden	Assignments to <i>this</i> pointer are not allowed.
Forbidden	Implicit conversion of pointer to derived class member into pointer to its base class member.
Restricted	Pointer math. Exception: trivial [Stroustrup, 99]
Mandatory	Array indexing is used over pointers into arrays
Forbidden	Beware of making assumptions about the size of pointers. They are not always the same size as <code>int</code> . Nor are all pointers always the same size, or freely interchangeable [Gabryelski 97]

7.7. Storage Class

Restricted	Use of <i>global</i> data is restricted: Exceptions: math entities (matrixes, complex numbers, or low-level types, such as linked lists, etc). Use class members instead. Consider putting global information in the context of a static class.
Restricted	usage of <code>static</code> is allowed inside the functions or classes [Stroustrup 99]

7.8. Classes

7.8.1. Class Declaration

Mandatory	The programmer should only have to look at the <code>.hpp</code> file to use a class.
------------------	---

Mandatory

Base class declaration templates -- in *.HPP:

```

//*****
// CLASS: < Name of a class >
// DESCRIPTION: <description>
// SRS REQUIREMENTS:
//*****
Class Base
{
public:    // Public method declarations
    virtual
    Base ( void );
    Base ( const Base &r );           //copy constructor
    const Base &operator= ( const Base &r ); //copy assignment
    ~Base ( void );
    // operator overloads
    // accessors
    // mutators

protected: // Protected method declarations

private:    // Private method declarations

protected: // Protected class data members

private:    // Private class data members

};
//*** END of CLASS Base *****

```

<p>Mandatory</p>	<p>Derived class declaration template – in *.hpp:</p> <pre> //***** // CLASS: < Name of a class > // DESCRIPTION: <description> // SRS REQUIREMENTS: //***** Class Derived : public Base { public: // Public method declarations virtual Derived (void); Derived (const Base &r); //copy constructor const Derived &operator= (const Derived &r); //copy assignment ~Derived (void); // operator overloads // accessors // mutators protected: // Protected method declarations private: // Private method declarations protected: // Protected class data members private: // Private class data members }; //*** END of CLASS Derived ***** </pre>
<p>Mandatory</p>	<p>In each access control group for methods the order:</p> <pre> // constructors // destructor // operator overloads // accessors // mutators </pre>
<p>Forbidden</p>	<p>Do not put data members in the public interface.[Stroustrup 99]</p>
<p>Mandatory</p>	<p>Make default, copy constructor as well as an assignment operator for a class private if they should not be used.</p>
<p>Mandatory</p>	<p>Class-members has to be declared explicitly.</p>
<p>Mandatory</p>	<p>Data members cannot have same name as the classes in whose scope they are declared.</p>
<p>Mandatory</p>	<p>Variable, that is part of the class, but is not part of the class' object, must be declared as static; [Stroustrup 99].</p>
<p>Forbidden</p>	<p>Forbidden: class “field types”, use virtual functions [Stroustrup 99]</p>
<p>Mandatory</p>	<p>Use struct for classes with all data opened [Stroustrup 99] and no member functions.</p>

7.8.2. Class definition

Mandatory	<p>Base class definition templates -- in *.CPP:</p> <pre> /** constructor ***** Base::Base (void) { } /** copy constructor ***** Base::Base (const Base &r) { } /** copy assignment ***** const Base&::operator= (const Base &r) { if (this != &r) //do not assign to yourself { // do the assignment } return *this; } /** virtual destructor ***** ~Base::Base (void) { } /** ***** member functions ***** </pre>
Mandatory	<p>Derived class definition template – in *.cpp:</p> <pre> /** constructor ***** Derived::Derived (void) : Base (value) { } /** copy constructor ***** Derived::Derived (const Derived &r) : Base (value) { } /** copy assignment ***** const Derived&::operator= (const Derived &r) { if (this != &r) //do not assign to yourself { // do the assignment } return *this; } /** virtual destructor ***** ~Derived::Derived (void) { } // other functions </pre>
Mandatory	<p>In each access control group for methods the order:</p> <pre> // constructors // destructor // operator overloads // accessors // mutators </pre>

7.8.3. Constructors

Forbidden	Default constructor is forbidden.
Mandatory	Class, which has pointer as a member or reference member, must has copy constructor and copy assignment [Stroustrup 99, 10.4.4.1., 10.4.6.3.].
Mandatory	List members in a constructor initialization list in order in which they are declared in the class header.
Mandatory	Use initialization to assignment in constructors. <pre> MyClass::MyClass(void) : theAlpha(0), theBeta(0), theGamma(0) { ; // do nothing } INSTEAD OF MyClass::MyClass(void) { theAlpha = 0; theBeta = 0; theGamma = 0; } </pre>
Mandatory	A constructor should put its object in a well-defined state. At the end of the constructor, the object must satisfy its class invariant. [Stroustrup 99].
Mandatory	If constructor requires some resource, the destructor, releasing this resource, must be defined [Stroustrup 99].
Forbidden	Call class constructor from other constructor is forbidden
Mandatory	A constructor that fails shall throw an exception.

7.8.4. Inheritance

Forbidden	Never redefine an inherited non-virtual function.[Corelinux 00]
Forbidden	Never redefine an inherited default parameter value.[Corelinux 00]
Forbidden	All inheritance must be public. private and protected inheritance is not allowed.[Gabryelski 97]

7.9. Member Functions

7.9.1. Function declaration

<p>Mandatory</p>	<p>Function declaration template - *.hpp:</p> <pre> //***** ** tttt ffff (<Type> <inputName> //Input Description ,<Type> <inputOutputName> //I/O Description ,<Type> <OutputName> //Output Description); // DESCRIPTION: // RETURNS: // SIDE EFFECTS: // END of FUNCTION < Function Name > </pre> <p>or for very simple functions:</p> <pre>tttt ffff(<Type> <inputName>); // Description</pre> <p>Rationale: We usually use a one-line-per-declaration form for several reasons.</p> <ol style="list-style-type: none"> 1. It is easy to comment the individual parameters, 2. It makes it easier to read when there many parameters. 3. It is easy to reorder the parameters, or to add one. The closing); is on a line by itself to make it easier to add a new 4. parameter at the end of the parameter list. 5. It is designed to be visually similar to the other declaration statements. 6. It works well with long identifier names. <p>However, with simple declarations the weight seems too great for the benefit [Gabryelski 97]</p>
<p>Mandatory</p>	<p>Do not space between function name and parenthesis, but space after open parents and before close parents with the exception of no arguments. e.g.</p> <pre> void doFunction(void) // ok void doFunction(ObjectRef aRef) // ok void doFunction(ObjectRef aRef) // wrong void doFunction(ObjectRef aRef) // wrong void doFunction() // ok </pre>
<p>Mandatory</p>	<p>Function prototypes are placed in .hpp files with the argument names:</p> <pre>SomeFunction(int a, int b, int diff)</pre>
<p>Mandatory</p>	<p>Function, that requires access to class' members, but is not called for class object instance, must be declared as <code>static</code>; [Stroustrup 99].</p>
	<p>Member functions should be declared <code>const</code> whenever possible [Gabryelski 97]</p>
<p>Mandatory</p>	<p>Class' function-member shall be declared as <code>const</code>, if it should not modify object value [Stroustrup 99]:</p> <pre>int month() const {...};</pre>
<p>Mandatory</p>	<p>All accessor functions should be <code>const</code>.</p>

Mandatory	Write member functions for all combinations of input conditions. Avoid using modes to distinguish between conditions. Use member function overloading instead.
Forbidden	Avoid case statements on object type; use member functions instead.
Recommended	Function has no more than five <i>arguments</i>
Mandatory	Each <i>parameter</i> is declared and the role in the function described
Mandatory	Parameters to a function and return values are commented at the point of declaration indicating the content
Restricted	Restricted: functions with undefined number of arguments: <code>f(<type>...)</code> [Stroustrup 99]
Restricted	Use of friend functions is restricted; exception: to avoid global data or open class members [Stroustrup 99]

7.9.2. Function definition

Mandatory	Function definition template - *.cpp: <pre> //***** tttt ffff::CCCC (<Type> <inputName> ,<Type> <inputOutputName> ,<Type> <OutputName>) { // BRIEF DESCRIPTION: <local variables> <function code> } //*** END of FUNCTION < Function Name > ***** </pre>
Mandatory	The whole function body shall be seen in the editor screen
Forbidden	Forbidden: Function call by reference – bad readability. Example: <pre> void increment(int& a) {a++} void f() { int x = 1; increment(x); // x = 2 } </pre> [Stroustrup, 99]
Mandatory	If function cannot be written in < 50 lines, the structuring using small <sub>function or assistant functions shall be used: Instead of: <pre> type VeryLongFuncyion() { << more than 50 lines> }; </pre> use: <pre> type VeryLongFuncyion() { // small functions are hidden in long function </pre>

	<pre> type SmallFunction1() { << ~ 20 lines >> }; type SmallFunction2() { << ~ 20 lines >> }; << less than 50 lines, small functions used >> }; </pre> <p>Use assistant functions, if small function could be used more than 1 member functions.</p>
--	--

7.9.3. Returns

Forbidden	Avoid member functions that return pointers or references to members less accessible than themselves. Use const ! [Corelinux 00]
Mandatory	Functions shall have a single exit point. Rationale. Multiple exit points usually add to the complexity of a function. Post conditions and invariant checks must also be performed prior to each return.
Forbidden	Do not return handles to internal data from const member functions. If a handle must be returned, make it const [Corelinux 00]
	Never return a reference to a local object or a de-referenced pointer initialized by new within the function. Rationale: Obviously, when a functions ends, the local object goes out of scope, and the reference is no longer valid. One might attempt to NEW the object in the function instead, but then who would call the corresponding DELETE? [Corelinux 00]
Mandatory	Function's return value <i>type</i> is explicitly declared or void
Forbidden	pointers or references to local variable return. Examples: <pre> int* f() { int local = 1; //... return &local //Forbidden } int& f() { int local = 1; //... return local //Forbidden } </pre> <p>[Stroustrup 99]</p>
Mandatory	The function return type is alone on a line
Allowed	The optional expression after return is omitted if and only if the return type of the function containing the return statement is declared as of type void

7.9.4. Inline functions

Restricted	Use inlining judiciously. <i>Rationale.</i> Inlines cause code bloat, slow down compile times, eat up name space, and not all compilers handle them the same way. [Corelinux 00] exceptions – considerable optimization. Inlining is not a panacea and in general should be done after the program is written, debugged and instrumented. [Stroustrup 99]
Mandatory	Inline class member functions has to be described at class declaration.
Forbidden	Never use <code>inline</code> functions in public interface definitions. <i>Rationale:</i> Since a client using your inlined interface actually compiles your code into their executable, you can never change this part of your implementation. And no one else can provide an alternate implementation. [Gabryelski 97]

7.10. Assistant functions

Forbidden	global functions [Stroustrup 99]
------------------	----------------------------------

7.11. Templates and Template Functions

Mandatory	<p>Template function indentation should follow the form</p> <pre>template<class T> void doSomethingFunction(args) { // }</pre> <p>-NOT-</p> <pre>template<class T> void template_function(args) {};</pre> <p><i>Rationale:</i> In class definitions, without indentation whitespace is needed both above and below the declaration to distinguish it visually from other members.</p>
Mandatory	<p>Template class indentation should follow the form</p> <pre>template<class T> class Base { public: // Types: };</pre> <p>-NOT-</p> <pre>template<class T> class Base { public: // Types: };</pre>

7.12. Statements

Mandatory	Put one statement per line
Mandatory	While, for and if statements shall always use braces, even if they are not syntactically required.

Mandatory	<i>Null statements</i> are commented, even if it is only <code>/* Do Nothing */</code> .
Forbidden	Continue statement is not used
Forbidden	do-while construction: [Stroustrup, 99].
Mandatory	The ends of the long compound statement must be commented: <pre> while(a < b) { while(something_else()) { for(i = 10; --i >= 0;) { for(j =10; --j >=0;) { //the massive code follows }//for(j =10; --j >=0;) }//for(i = 10; --i >= 0;) } //while(something_else()) } //while(a < b) </pre>
Mandatory	Long, complex statements are changed into many smaller, simpler ones
Mandatory	Only 1 function call is used in the condition or a structured statement
Mandatory	In a statement that consists of two or more lines, every line except the first is indented an extra level to show that they are continuations of the first line
Mandatory	In multi-line statements the <i>arithmetic and logical operators</i> are put at the beginning of each continuation line
Mandatory	Each <i>statement</i> put on a line by itself

7.12.1. if Statement

Recommended	Most if statements should be followed by an else statement. [Corelinux 00]
Mandatory	<pre> if(by_land) { major_action(); } else { minor_action(); } </pre>
Mandatory	<pre> if(by_land) { major_action(); } else if(by_air) { minor_action(); } else(by_sea); { stay_at_home(); } </pre>

Mandatory	<p>Empty statements shall have a semicolon with a <code>// do nothing</code> comment.</p> <pre> if(x != y) { y = x; } else { ; // do nothing } </pre> <p>Rationale: The statement clearly shows that the developer has thought about the condition.</p>
Forbidden	<code>break</code> statement is not used in the body of if statement
Mandatory	Boolean value is not checked for equality with 1; instead test for inequality with 0 is used
	<p>Use predicate form for the long conditions:</p> <pre> if(TodaySomethingIsGoingOn() ItIsFridayNight() EverythingIsAlreadyBotheredMe()) { GoToTheNextStore(); } </pre>
1	<p>The <i>else-if</i> is used for multiple-choice constructs whenever: the conditions are not mutually exclusive their order of evaluation is important they test different variables. Otherwise, <i>switch</i> is used</p>
Mandatory	Use <code>if (cond) . . . else</code> rather than conditional expressions <code>(cond) ? :)</code> if only to clarify the intended operation.

7.12.2. switch Statement

Mandatory	<p>Indent cases one level from the switch and indent the code one level beyond the case. The break statement is at the same indentation level as the code. All switch statements shall have a default case. If all cases have been handled then the default code shall be kind of assertion (see [Corelinux 00]. If not then it shall be an empty statement.</p> <pre> switch(variable) { case 1: break; case 2: break; default: NEVER_GET_HERE; // See Assertion.hpp break; } </pre> <p>OR</p> <pre> switch(variable) { case 1: break; case 2: break; default: ; // do nothing break; } </pre> <p>Rationale: The last <code>break</code> in the <code>switch</code> is, strictly speaking, redundant, but it is required nonetheless. This prevents a fall-through error if another case is added after the last one.[Gabryelski 97]</p>
Mandatory	Every case in a <code>switch</code> is ended with a <code>break</code> or the comment
Mandatory	Default case in every switch is included, even if it consists of nothing but a null statement
Mandatory	Cases are short and independent of one another

7.12.3. Loop Constructs

Mandatory	<p><code>for</code> expressions is used only for control of the loop:</p> <pre> for (; ;) //loop comment { <loop body> } </pre> <p>Make its usage clear with comments.</p> <p>Rationale: This form is better than the functionally equivalent <code>while (TRUE)</code> or <code>while (1)</code> since they imply a test against <code>TRUE</code> (or <code>1</code>), which is neither necessary nor meaningful (if <code>TRUE</code> ever is not true, then we are all in real trouble).[Gabryelski 97]</p>
Mandatory	Use <code>for</code> loops when the loop control needs initializing or recalculating; otherwise, use <code>while</code>
Mandatory	Minimize use of <code>break</code> in loops. Only use it for abnormal escape.

Mandatory	The null body of a <i>for</i> or <i>while</i> loop is alone on a line and commented so that it is clear that the null body is intentional and not missing code
Mandatory	Count for loops in ascending.

7.12.4. while statement

Mandatory	<pre>while(some_condition) { //internal code }</pre>
Mandatory	In control loops, the controlling equality-expression remains true for most of the loop

7.12.5. for Statement

Mandatory	<p>Long <i>for</i> statements are splitted along statement boundaries, e.g.</p> <pre>for (index = the_first_element; index <= last_element; inc_pointer(index)) { statement1(); }</pre>
------------------	--

7.12.6. goto Statement

Forbidden	
------------------	--

7.13. Expressions

Forbidden	Expressions with nested ternary ?: operators are not used
Forbidden	Ternary operator is not used
Forbidden	Comma operator is not used
Forbidden	None of the arithmetic expressions is a logical expression
Forbidden	None of the logical expressions is an arithmetic expression
Forbidden	unsigned arithmetic [Stroustrup 99]
Forbidden	complex expressions [Stroustrup 99]
Mandatory	brackets should be used [Stroustrup 99]
Mandatory	Parentheses are used to emphasize chunks in expressions
Mandatory	Spaces before and after each <i>arithmetic operator</i> are put. The primary operators (“arrow” (->), “dot” (.), subscript ([]), and parentheses after function names) are written with no space around them
Mandatory	The assignment operators and the conditional operator have space around them

7.13.1. Unary Expressions

Mandatory	++ and -- operators are used only in assignment statements/not considered to be operators/are not used inside other statements
Mandatory	Do not space between an unary operator and its operand, but do space the other side. <pre>a++ ; --b;</pre>
Mandatory	Operators ++ and -- are put on lines by themselves

Mandatory	Default to pre-increment and pre-decrement unless the post-increment/decrement operators are logically necessary.
------------------	---

7.13.2. Binary Expressions, Conditional Expressions and Assignments

Mandatory	One space spacing used around binary operators
Forbidden	There is no comparison floating point numbers for equality (use <code>></code> , <code><</code> , <code><=</code> , or <code>>=</code>)
Forbidden	There are no assignment expressions inside statements
Mandatory	In the test expression of <i>while</i> , <i>for</i> or <i>if</i> , the comparison is written explicitly, rather than relying upon the default comparison to zero
Mandatory	Only a logical expression is used in the condition of structured statement
Mandatory	Logical expressions assignment only to a variables of <i>Boolean</i> type are used
Forbidden	Do not use comparisons in mathematical expressions. <pre>numberOfDays = (isLeapYear() == TRUE) + 28; // not OK</pre>
Mandatory	All non-boolean comparison expressions should use a comparison operator. Do not use implicit <code>!= 0</code> . <pre>REQUIRE(aObjectPtr != NULLPTR); // good REQUIRE(aObjectPtr); // bad</pre>
Recommended	Minimize negative comparisons.

7.13.3. Precedence

Mandatory	Parentheses are put around everything (precedence rules not used)
Forbidden	Implicit evaluation order in equality-expressions is not used (separate statements used instead)
Mandatory	Bitwise operators (<code>&</code> <code>>></code> <code><<</code>) are explicitly parenthesized when combined with other medium-precedence operators (arithmetic, bitwise, relational, logical)

7.13.4. Type casting

Restricted	explicit type casting; if it is unavoidable, use explicit operators; avoid $T(e)$ form. [Stroustrup 99]
Forbidden	There are no data casts to truncate expressions to a specific number of bits
Mandatory	Use explicit casting, instead of the compiler default. <pre>Dword aUnsignedValue(0); Real aRealValue(3.7); aBigValue = Dword(aRealValue);</pre>

7.14. Operations with Objects

Mandatory	Pass and return objects by reference instead of value whenever possible.
Mandatory	If the passed object is not going to be modified then pass it as a const reference.
Mandatory	Explicit call of destructor is restricted
Mandatory	Ensure that objects (both simple and class) are initialized before they are used
Mandatory	Scope resolution operator (<code>::</code>) is put on line by himself

Mandatory	Dereference pointer to class (->*) is put on line himself
Mandatory	Don't assume you know how a <code>struct</code> or <code>class</code> is laid out in memory, or that it can be written to a data file as is. [Gabryelski 97]

8. Defensive Programming

Mandatory	Input variables and intermediate variables with physical significance are checked for plausibility
Mandatory	Variables' range is checked
Mandatory	Values' plausibility is checked (where possible)
Mandatory	Parameters to procedures are range checked at procedure entry
Mandatory	Functions treat all parameters as read-only
Mandatory	The code is fault tolerant
Mandatory	Heavy use of assertions and exceptions are encouraged.

9. Preprocessor

9.1. Preprocessor Macros

Forbidden	macros usage
Forbidden	<p><code>#define</code> shall not be used for manifest constants. Use <code>const Type name = value;</code> or <code>enum</code> instead:</p> <pre>#define SOME_MAGIC_NUMBER 5 // wrong const Int magicNumber(5); // ok</pre>

10. Preprocessor Directives

Forbidden	<p>No <code>#pragma</code> directive should be used</p> <p>Rationale. <code>#pragma</code> directives are, by definition, non-standard, and can cause unexpected behavior when compiled on other systems. On another system, a <code>#pragma</code> might even have the opposite meaning of the intended one. In some cases <code>#pragma</code> is a necessary evil. Some compilers use <code>#pragma</code> directives to control template instantiations. In these rare cases the <code>#pragma</code> usage should be documented and, if possible, <code>#if-def</code> directives should be to ensure other copilers don't trip over the usage [Gabryelski 97].</p>
Restricted	<p>In general, avoid using <code>#ifdef</code>. Conditional compilation is used sparingly and Don't obscures the code.</p> <p>Rationale. Modularize your code so that machine dependencies are isolated to different files and beware of hard coding assumptions into your implementation.[Gabryelski 97]</p>
Mandatory	<code>#else</code> and <code>#endif</code> directives are commented with the symbol used in the initial <code>#ifdef</code> , <code>#ifndef</code> , or <code>#if</code> directive
Forbidden	There is no commented out code. Conditional compilation (<code>#ifdef UNDEF</code>) is used to get rid of unwanted code

Mandatory	<p>If you use <code>#ifdef</code> to select among a set of configuration options, you need to add a final <code>#else</code> clause containing a <code>#error</code> directive so that the compiler will generate an error message if none of the options has been defined:</p> <pre data-bbox="416 331 1465 667"> #ifdef sun #define USE_MOTIF #define RPC_ONC #elif hpux #define USE_OPENLOOK #define RPC_OSF #else #error unknown machine type #endif </pre> <p><i>[Gabryelski 97]</i></p>
Forbidden	<p>Never change the language's syntax via macro substitution. For example, do not do the following:</p> <pre data-bbox="416 808 1465 936"> #define BEGIN { // EXTREMELY BAD STYLE!!! #define when break;case // EXTREMELY BAD STYLE!!! </pre> <p>Rationale: This makes the program unintelligible to all but the perpetrator. C++ is hard enough to read as it is <i>[Gabryelski 97]</i></p>

11. Exceptions

Mandatory	Use exceptions rather than returning a failure status.?????
Mandatory	<p>Exception format:</p> <pre data-bbox="416 1245 683 1630"> try { statements; } catch (type) { statements; } catch (...) { statements; } </pre>

12. Library Functions

Mandatory	Project Allowed function libraries only shall be used
Mandatory	For each allowed function libraries the list of forbidden functions shall be established; the use of functions from such lists is forbidden

13. Miscellaneous

Mandatory	Don't assume you know the memory layout of a data type. <i>[Gabryelski 97]</i>
Mandatory	Eradicate all compiler warnings. Set the compiler to the highest warning level. Any unavoidable warnings must be explicitly commented in the code. Unavoidable compiler warnings should be extremely rare.
Forbidden	Hardcodings (Absolute file path names for files, literals (except constant definition))
Mandatory	Only the 95 standard ANSI characters are used

14. References and resources

- [Coad 93]* Coad Peter and Jill Nicola. Object-oriented programming. - (Yourdon Press computing series), 1993.
- [Maguire 93]* Maguire Steve. Writing solid code. Microsoft's Techniques for Developing Bug-Free C Programs. - Microsoft Press, 1993
- [Pohl 93]* Pohl Ira. Object-Oriented Programming Using C++. Addison-Wesley Publishing Company, 1993.
- [STD-WinApDesign Rev. 1.0 96]* WINDOWS APPLICATION DESIGN TECHNIQUE / TCK COMPANY INTERNAL STANDARD
- [Stroustrup 99]* Stroustrup Bjarne. The C++ Programming Language /. - AT&T Labs, 1999.
- [Holub 95]* Allen I. Holub. Rules for C and C++. - М.: БИНОМ, 1996
- [Corelinux 00]* C++ Coding Standards / The Corelinux Consortium, Revision 1.6, 2000
- [Gabryelski 97]* Keith Gabryelski. Wildfire C++ Programming Style / With Rationale / Wildfire Communications, Inc., Email: ag@wildfire.com
- [Todd Hoff 00]* Todd Hoff. C++ Coding Standard. -- 2000-04-14, tmh@possibility.com
- [Ottinger 97]* Tim Ottinger. Ottinger's Rules for Variable and Class Naming

UMP end—user license agreement (UMP—EULA)

Copyright © 2003 Alexander Kozlinski, www.uni-mod.com

"UniModelers Process" and UMP are trademarks of Alexander Kozlinski

1. Permission is hereby granted, free of charge, to any individual or entity obtaining a copy of this software and associated documentation files (the **Toolkit**), to deal in the **Toolkit** and to engage in any actions related to same without any restrictions whatsoever.
2. If you distribute, sublicense, publicly display, publicly and/or digitally perform, publish and/or sell copies of the **Toolkit** or any Derivative Works or Collective Works, you must include in all copies or substantial portions of the **Toolkit** the above copyright notice and this permission notice.
3. **The Toolkit is provided "as is", without warranty of any kind, express or implied, including, but not limited to, the warranties of merchantability, fitness for a particular purpose and non-infringement. In no event shall the authors and/or copyright holders be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of, or in connection with, the Toolkit or the use or other dealings in the Toolkit.**

***** end of UMP—EULA *****

The UMP—EULA is based on the MIT License and Creative Commons License.