Namespaces:

Prefix namespace names with “NS”: NSSampleNamespace

Type names:

Prefix type names (typedefs, structs, etc...) with “T”.

Prefix enums with “E”.

Postfix exceptions with “Exception”: InvalidOperationException, StackOverflowException.

Prefix class names with “C”, even if class has only static members (or do not prefix them at all), except for "java interface analog" where class name is prefixed with "I".

Examples:

```
typedef int TInt;
typedef short int TInt16;
struct TPoint {
  TInt m_x;
  TInt m_y;
}
class CSampleClass;
class CStaticClass;
class ISampleInterface;
enum EBasicColor {RED, GREEN, BLUE};
```

SEALED

Data names:

All variable names should begin in lowercase; if a var name contains multiple words, the first letter of each word will be capitalized.

All constants and enumerated constants should be uppercase

Examples:

```
string fileName;
int xCoord, yCoord;
bool myFlag;

#define ADD(_x, _y) (_x + _y)
#define PI 3.14

enum TBasicColor {RED, GREEN, BLUE};
```

SEALED.

Prefix all function arguments with "_" or with "a_" or don't use a prefix at all.

Examples:

```
void myFunction(int _x, int _y, string _fileName, bool _flag) {}
void myFunction(int a_x, int a_y, string a_fileName, bool a_flag) {}
void myFunction(int x, int y, string fileName, bool flag) {}
```

The 3rd style is recommended.

SEALED.

Use "m_" prefix for member variables (class or struct): m_coordX, m_myString

Automatic variables are not to be prefixed; the same holds for heap vars defined inside functions.

Examples:

```
void f() {
  int x;
  float startPoint;
 double myLocalVariable_1;
  int* y;
  float* endPoint;
 double* myLocalVariable_2;
}
```

SEALED.

Global variables, static variables (class, function etc) are not to be prefixed.

SEALED.

Function names begin in lowercase and respect the same convention as var names: length(), handleCommand(...), setSize(...), getSize(), getText()

SEALED.

For acronyms: only the first acronym letter should be uppercase

```
string myUrlDesc; // compared to myURLDesc which is harder to read
string url;
void makeUrlMap() {}
```

```
void urlTest() {}
```

and for member vars:

```
string m_myUrlDesc;
string m_url;
```

SEALED.

Ideal OOP should not expose its internal variables... in other words all member vars should be private or protected.

A class should expose accessors to manipulate its internal state.

```
class CTest {
  public:
    CTest();
    ~CTest();
                                    // place empty lines to group related
functions (here, this should be an empty line)
    int getCoord();
    void setCoord(int _coord);
                                   // here, this should be an empty line
  private:
    int m_coord;
};

...
CTest test;
test.setCoord(10);
cout << test.getCoord() << endl;
```

Anyway the programmer should use OOP (or ideal OOP) when it makes life easier. In some cases it may be better for project to break some OOP rules, or even to use functional style.

Anyway there are some cases in which breaking this OOP rule is justified. This OOP rule should be interpreted only as a hint.

HINT. SEALED.

Other conventions:

> Not to use private inheritance in classes

sometimes it is necessary: when you want to benefit from the super class behavior internally but not to expose that behavior directly outside

In this case it is recommended to instantiate that class as a private variable. But anyway it is not strict requirement, if we see that private inheritance dramatically simplifies that task then will use it.

HINT. SEALED.

> Try to avoid using multiple inheritance, use interfaces instead

HINT. SEALED.

> Identifiers that start with uppercase are often used as template parameters: template <class T> class point { }. Also template parameters should be one letter in length, capitalized.

HINT. SEALED.

> Basic coding style:

Place an empty line between different sections of a class: private, public, protected

SEALED.

Place an empty line between function definition section and variable definition section inside each private, public, protected section

Also place var defs before function defs inside each section. Separate var defs and function defs in each section by an empty line.

```
class CTest {
    public:
        void someFunc();
                    // empty line
    private:
        int m_data1;
        int m_data2;
                    // empty line
        void f1();
        void f2();
}
```

SEALED.

Place {, preceded by a space, on the same line as class defs, function defs etc

Place } on its own line. Use indentation of 2 or 4 characters:

```
if (a < b) {
```

```
  doSomething();
  ...
}

class CSample {
  public:
    int m_publicVariable;

    CSample();
    ~CSample();

  private:
    int m_privateVariable;
}

int test() {
  int a, b, i, r;
  bool flag;
  ...
  while (a < b) {
    if (flag) {
      ...
    }
    else {
      for(i = 0; i < N; i++) {
        ...
      }
    }
  }
  return r;
}
```

SEALED.

> Use spaces in expression: a + b , not a+b

> Do not use spaces in unary operators: ++i, not ++ i

> When passing arguments by reference or by pointer use this style for formal parameters:

```
void swap(int& a, int& b)  // do not use int &a
void display(int* a, int* b)  // do not use int *a
```

SEALED.

For variable declaration there may be a different situation. You may declare multiple variables:

```
int* a, b, c;
```

and the int* will only be applied to "a"; "b" and "c" will have int type. In this case it is better to declare them in following way:

```
int *a, b, c;
```

But, we can agree not to place pointer or reference declarations in multiple variable declaration statement, and rewrite this code like:

```
int* a;
int b, c;
```

In this case everything is OK.

HINT. SEALED.

Every variable should be a shared_ptr except for the built in types like int, double, char etc or a stream type(shared_ptr does not have << and >> operators overloaded).

Do not use ever raw pointers(not even *this* pointer, except it is a visitor).

If it is not a built in type use shared pointers.

Every type of shared_ptr should have a typedef over it.

For shared_ptr over ANSI c++ types the typedefs should be added in the support/typedefs.h

For CSLOM the typedefs are in CSLOM_Declarations.h

Maintainable Code.

The following set of rules will create code which is easy to maintain, high performance and easy to debug. Furthermore, following these conventions will help you write code faster and write maintainable, debuggable and easy-to-understand code.

1. Write factored code:

Not factored code	Factored Code
...	void fAB() {
if (cond) {	stmtA;
stmtA;	stmtB;
stmtB;	}
}	...
else {	if (cond) {
stmtA;	fAB();
stmtB;	}
stmtC;	else {
}	fAB();
...	stmtC;
	}
	...

2. Align your code, and use rectangular selections in your editor

Not aligned code	Aligned code
enum Example {	enum Example {
TYPE_A,	TYPE_A ,
TYPE_B,	TYPE_B ,

```
  TYBE_AB,                        TYBE_AB,
};                              };
Example ex;                     Example ex;
...                             ...
switch (ex) {                   switch (ex) {
case TYPE_A:                    case TYPE_A : doSomething();     break;
  doSomething();                case TYPE_B : doSomethingElse(); break;
  break;                        case TYPE_AB: doSomething();     break;
case TYPE_B:                    }
  doSomethingElse();
  break;
case TYPE_AB:
  doSomething();
  doSomethingElse();
  break;
}
```

3. Do not write function calls in parameter list. Instead assign the function return values to temporary variables. (this will help you a lot in debug mode)

How not to do it	How to do it

`int f( ... ) { ... }`
`inf g( ... ) { ... }`
`...`
`doSomething(f(...), g(...));`
`...`
	`int f( ... ) { ... }`
	`inf g( ... ) { ... }`
	`...`
	`int x = f(...);`
	`int y = g(...);`
	`doSomething(x, y);`
	`...`

4. Use else if in cases that the conditions are mutualy excusive

How not to do it	How to do it
`int x;` `...`	`int x;` `...`

```
if (x == 1) {
   ...
}
if (x == 2) {
   ...
}
if (x == 3) {
   ...
}
```

```
if (x == 1) {
   ...
}
else if (x == 2) {
   ...
}
else if (x == 3) {
   ...
}
```

5. Learn how to use the power features in your editor

   (a) regular expression search

   (b) auto indent, highlight, search & replace

   (c) rectangle operation

   (d) macros

6. Use #ifdef for Development and Release versioning

Example

```
inline RefCSLOmBase CSLOmSignal::getCSLOmBaseChild(int i) const {
 #ifdef DEVELOP
   return m_children->at(i);
 #else RELEASE
   return m_children[i];
 #endif
}
```

7. Use function predicates (for self-documenting code)

How not to do it	How to do it
`enum EType {`	`enum EType {`

```
  TYPE_NUM,                                  TYPE_NUM,
  ...                                        ...
};                                         };
class A {                                  class A {
   EType m_type;                              EType m_type;
 public:                                    public:
   ...                                        ...
   EType getType() {                          EType getType() {
     return m_type;                             return m_type;
   }                                          }
   ...                                        bool isNumber() {
};                                             return e.getType() == TYPE_NUM;
...                                            }
if (e.getType() == TYPE_NUM) {                 ...
   ...                                     };
}                                          ...
...                                        if (isNumber()) {
                                              ...
                                           }
                                           ...
```

8. Use defensive programing. Use ASSERT

Example

```
void f(string *p) {
   ASSERT(p != NULL, „Pointer received is NULL”);
   ASSERT(p->length() != 0, „String received is empty”);
   ...
}
```

9. Every class that uses RefCountshould have a cast function to cast a base object to the type of the class and a getThis() function that returns a RefCount object to itself

Example

```
class A : public B{
 public:
   static RefA cast(B object) {
    ...
    // Test the type of the object to be sure that the
    // object can be cast
    ...
    return NSRefCout::ref_down_cast(object);
   }
 private:
   RefA getThis() { return RefA::cast(m_weakRef.getStrongRef());}
 protected:
   A() { ... }
 public:
   static RefA build() { ... }
   ...
};
```

10. Use debuger for finding a segmentation fault, print a value

Walkthrough

Compile your code with -ggdb (for g++) or debug(for ant)option:

g++

```
# g++ test.cpp -ggdb -o a.bin
```

ant (source files in build.xml)

```
# ant debug
```

Run gdb with the binary file:

```
# gdb a.bin
```

Run the program:

```
(gdb) run
Starting program: /home/bogdanz/temp/a.bin
Program received signal SIGSEGV, Segmentation fault.
0x0000000000400aa6 in class::function (this=0x0) at test.cpp:11
11          int function() {return x;}
```

This line represents the line that the segmentation fault appeared on.

Print all variables on this line to see which one triggered the segmentation fault

```
(gdb) print x
Cannot access memory at address 0x0
```

If no variable from this line triggered the segmentation fault then print the stack with command „bt" go to previous stack frame by executing „up" command:

```
(gdb) bt
#0  0x0000000000400aa6 in A::getX (this=0x0) at a.cpp:11
#1  0x0000000000400a0d in main () at a.cpp:18
(gdb) up
#1  0x0000000000400a0d in main () at a.cpp:18
18        cerr<<y->getX()<<endl;
```

Print again all the variables to see which one triggered the segmentation fault:

```
(gdb) print y
$1 = (A *) 0x0
```

```
Repeat last 2 steps until you find the pointer with 0x0 value.
```

11. On assert failure print the values that trigger the assertion or use gdb to see the values being tested.

Walkthrough

Compile your code with -ggdb (for g++) or debug(for ant)option:

g++	ant (source files in build.xml)
`# g++ test.cpp -ggdb -o a.bin`	`# ant debug`

Run gdb with the binary file:

```
# gdb a.bin
```

Run the program:

```
(gdb) run
Starting program: /home/bogdanz/temp/a.bin
****
a.bin: a.cpp:18: int main(): Assertion `y' failed.

Program received signal SIGABRT, Aborted.
0x000000389852e21d in raise () from /lib64/tls/libc.so.6
```

Assertion failure prints on the screen the file, the line number and the function where the assert has been triggered.

Run the next command to terminate the current execution.

```
(gdb) next
Single stepping until exit from function raise,
which has no line number information.
```

```
Program terminated with signal SIGABRT, Aborted.
The program no longer exists.
```

Toggle a break point on the line number in the file printed by the assertand run again.

```
(gdb) break a.cpp:18
Breakpoint 1 at 0x400a04: file a.cpp, line 18.
(gdb) run
Starting program: /home/bogdanz/temp/a.bin
****

Breakpoint 1, main () at a.cpp:18
18        assert(y);
```

Print all the variablesto see which one triggered the assert:

```
(gdb) print y
$1 = (A *) 0x0
```

Use bt and up commands to backtrace where the value has been change to the undisered value. See 10.

12. Do NOT use tabs in code. Use only spaces.
13. warnings/errors : No strings in code, use the warning error class interface and error/warning codes.
14. const int: No numbers in code, use const int variables and for drivers use random numbers(to test more cases).
15. Learn how to use Valgrind memory leak checker