Norsk Regnesentral
NORWEGIAN COMPUTING CENTER

Note

The C++ coding standard for SAND

Second Edition

Note no SAND/05/2010
Authors Per Røe
Ragnar Hauge
Date May 4, 2010

Norsk Regnesentral
NORWEGIAN COMPUTING CENTER

Norwegian Computing Center

Norsk Regnesentral (Norwegian Computing Center, NR) is a private, independent, non-profit foundation established in 1952. NR carries out contract research and development projects in the areas of information and communication technology and applied statistical modeling. The clients are a broad range of industrial, commercial and public service organizations in the national as well as the international market. Our scientific and technical capabilities are further developed in co-operation with The Research Council of Norway and key customers. The results of our projects may take the form of reports, software, prototypes, and short courses. A proof of the confidence and appreciation our clients have for us is given by the fact that most of our new contracts are signed with previous customers.

Norsk Regnesentral
Norwegian Computing Center
Postboks 114, Blindern
NO-0314 Oslo, Norway

Besøksadresse
Office address
Gaustadalléen 23
NO-0373 Oslo, Norway

Telefon · telephone
(+47) 22 85 25 00
Telefaks · telefax
(+47) 22 69 76 60

Internett · internet
www.nr.no
E-post · e-mail
nr@nr.no

Title	**The C++ coding standard for SAND**
Authors	**Per Røe , Ragnar Hauge**
Date	May 4, 2010
Publication number	SAND/05/2010

Abstract

This document describes the C++ coding standard for SAND. This standard shall be followed in all new coding projects. Existing projects should continue to use the already established standard for this project, but in cases where multiple coding styles are used or in association with major rewrites this standard should be used as a guide.

Keywords	standardization
Target group	SAND
Availability	
Project	
Project number	
Research field	
Number of pages	24

Contents

1 Naming conventions

Use descriptive names. Names with a short scope can have shorter names, but all names with global, template or class scopes should have longer descriptive names.

1.1 Files and directories

Files and directories shall always have names in lowercase letters which should reflect the name of the class it contains. The file extensions are `.cpp` and `.hpp` for source and header files, respectively.

```
/parser/fileparser.cpp
/parser/fileparser.hpp
```

1.2 Namespaces

Namespaces shall have names starting with an uppercase letter. If the name of the namespace consists of several words each word should begin with an uppercase letter. Do not use underscores in namespace names.

```
namespace NRLib {
    .
  double NormalPDF(double mean, double var);
}

void MyFunc() {
  double p = NRLib::NormalPDF(0, 2);
}
```

1.3 Classes

Classes should be named in the same way as namespaces.

1.4 Functions

Functions should be named in the same way as namespaces.

```
int Max(int a, int b);

void MyFunc() {
  int c = Max(2, 5);
}
```

1.5 Variables

Variables should be lowercase with words separated by underscores.

```
int count_iterations = 0;
```

Variable names should reflect what the variable is. For example `i`, `j` and `k` should only be used for integers. While `f` and `g` typically are floats.

1.6 Class member variables

Class member variables should end with an underscore.

```
class MyClass {
```

```
public:
  int GetSize() const;
protected:
  void SetSize(int size);
private:
  int size_;
};

MyClass::SetSize(int size) {
  size_ = size;
}
```

1.7 Enumerators

Enumerators should be uppercase with words separated by underscores.

```
enum {LOWERCASE_LETTER, UPPERCASE_LETTER};
```

1.8 Constants

Constants should be uppercase with words separated by underscores.

```
const double PI = 3.14159265358979323846;
```

1.9 Macros

Do not use preprocessor macros unless it is strictly necessary. In most cases templates, in-lined functions or constants can be used instead, giving type safety, and making it easier to debug.

If used, they should be named using uppercase letters. To avoid name clashes with constants and enumerators, all macro names should be prefixed with `MAC_`.

```
#define MAC_MY_DEBUG 1
  .
  .
  .
#ifdef MAC_MY_DEBUG
  assert(i > 0);
#endif
```

2 Formatting

2.1 Indents

Each scope should be indented with two spaces. The editor should be configured in such a way that it only inserts spaces, since tabs can be visualized with variable width in different editors.

```
int main() {
  for (int i = 0; i < 100; ++i) {
    if (i % 2 != 0) {
      std::cout << i << ‘‘is odd\n’’;
    }
  }
```

```
}
```

2.2 Spacing

Use spacing in such a way that the code becomes easy to read.

The following rules should be obeyed:

- No space before a ';':

```
std::cout << i << ``is odd\n'';
```

- No space around the '->' and '.' operators.

```
MyClass *my_ptr = new MyClass();
MyClass my_obj();

my_ptr->DoSomething();
my_obj.DoSomething();
```

- Space before and after mathematical and logical operators except the not operator `!`, and the increment and decrement operators '++' and '--':

```
if (!(i % 2 == 0)) {
  ++i;
  j = 0.5 * i;
}
```

- But the space before and after operators can be let out to visualize precedence:

```
Example:
  a = b + c*d;
Instead of:
  a = b + c * d;
```

- No space after '(' or before ')':

```
if ((i == 1) && (j == 0))
```

- Make two empty lines after a function definition.

```
void MyClass::SetVal(int val) {
  val_ = val;
}



int MyClass::GetVal() const {
  return val_;
}
```

2.3 Variable alignment

To improve readability, variable declarations should be aligned. This includes both within normal code, when declaring class variables and in function headers.

Initializations, function declarations etc. should also be aligned whenever appropriate. To enhance readability, the * and & tokens denoting pointer and reference types should also be aligned.

```
Do:
```

```
class MyClass {
  ...
private:
  int                size;
  NRLib::Vector    * mean;
  NRLib::SymMatrix & covariance;
  double             signal_noice_ratio;
}
```

Instead of:

```
class MyClass {
  ...
private:
  int size;
  NRLib::Vector* mean;
  NRLib::SymMatrix& covariance;
  double signal_noice_ratio;
}
```

2.4 Brackets

Rules for placing of brackets:

- Always put '}' on a separate line, while '{' shall be on the end of a line:

  ```
  if (i == 1) {
     :
  }
  else {
     :
  }
  ```

  An exception to this rule is starting brackets for namespaces, classes and functions, which can be set on a separate line if it improves readability.

  ```
  Grid::Grid(int nx, int ny, int nz)
    : nx_(nx),
      ny_(ny),
      nz_(nz)
  {
    ...
  }
  ```

- Always enclose nested statements in brackets:

  ```
  for (int i = 0; i < len_; ++i) {
    if (val_[i] > max)
      max = val_[i];
  }
  ```

3 Classes

3.1 Member variables

Member variables and classes should generally either be defined as `protected`, or preferably `private`.

Convenience class variables should be avoided. Convenience class variables are variables that are easily deduced from other class variables or are copies of data held by other classes.

3.2 Accessors

- Use functions whose names start with `Get` and `Set` for access of class members:

```
class MyClass {
public:
    .
    .
  int  GetValue() const    { return val_; }
  void SetValue(int val)   { val_ = val; }

private:
  int val_;
}

void MyFunc() {
  MyClass object;
     :
  if (object.GetValue() == 0) {
    object.SetValue(42);
  }
}
```

  These functions should generally be inline.

  Never use Get in the beginning of functions that are not simple accessors. For variables easily derived from accessors, use `Find` as prefix. For more complex computations, use `Calculate`. This gives an indication of the time used in these functions, and the utility of temporary storage.

- Use an overloaded `[]`-operator for access of data from an one-dimensional data structure. Both a const-version, and a version that can be used to change data should be provided.

```
class Well {
public:
     :
  const double   operator[](int i) const;
  double       & operator[](int i);

private:
  std::vector<double> log_;
}
```

```
void MyFunc() {
  Well my_well;
    :
  if (my_well[i] == 0) {
    my_well[i] = my_well[i - 1];
  }
    :
}
```

- Use an overloaded `()`-operator for access of multi-dimensional data. Both a const-version, and a version that can be used to change data should be provided.

```
class Grid {
public:
    :
  const double   operator()(int i, int j, int k) const;
  double       & operator()(int i, int j, int k);
    :
}

void MyFunc() {
  Grid grid;
    :
  if (grid(i, j, k) < 0) {
    grid(i, j, k) = grid(i, j, k - 1);
  }
    :
}
```

3.3 Const functions

All member functions that do not alter member variables should be labeled `const`. This is important so that the functions can be used on const objects, and it also helps documenting how the function works.

```
class MyClass {
    :
  int GetValue() const;
    :
}
```

3.4 Lazy evaluation of class members

A constant function should never be allowed to change member variables.

C++ allows declaring `mutable` variables that can be changed by constant functions, however using this may lead to unclear code and should be avoided. Casting away the const-ness is an alternative approach, but this can lead to logical errors, and should never be done.

3.5 Rule of 3

When a class needs a non-empty destructor it usually also needs a non-standard copy constructor and assignment operator too. This is typically the case if the class handles some resources like allocated memory, files, etc.

If no copy constructor and assignment operator is given for a class, the compiler will make default ones that copy all member data, but these will not do the correct thing in the cases given above.

```
class MyClass {
public:
  MyClass(int size);
  ~MyClass();
  MyClass(const MyClass& rhs);
  MyClass& operator=(cont MyClass& rhs);

private:
  int     size_;
  double* data_;
}

MyClass::MyClass(int size)
 : size_(size)
{
  data_ = new double[size];
}

MyClass::~MyClass() {
  delete [] data_;
}

MyClass(const MyClass& rhs) {
  data_ = new double[rhs.size_];
  for (int i = 0; i < size_; ++i) {
    data_[i] = rhs.data_[i];
  }
}

MyClass& operator=(cont MyClass& rhs) {
  // Check for self-assignment
  if (this != &rhs) {
    delete data_;
    data_ = new double[rhs.size_];
    for (int i = 0; i < size_; ++i) {
      data_[i] = rhs.data_[i];
    }
  }
}
```

If the copy assignment operator or the copy constructor is not implemented, meaning that it should not be possible to make a copy of a object of the class, a copy constructor and/or copy assignment operator definition should be declared as private for the class to prevent use of the default copy constructor and/or copy assignment operator.

```
class FileReader {
public:
  FileReader(const std::string& filename);
```

```
  ~FileReader();
private:
  fd file_;

  // Not allowed to make copy of FileReader.
  FileReader(const Filereader& rhs);
  FileReader& operator=(cont FileReader& rhs);
}

FileReader::FileReader(const std::string& filename) {
  fd = open(filename);
}

FileReader::~FileReader() {
  close(fd);
}
```

3.6 Ordering of class members

The access levels should be ordered in the following way:

1. `public` class members.
2. `protected` class members.
3. `private` class members.

The `public` interface should come first in a class definition, thereafter the `protected` interface, that is needed to develop sub-classes. The `private` section containing implementation details should come last.

For each access level the functions and member data should be given in the following order:

1. Friend classes.
2. Constructors.
3. Copy constructor.
4. Destructor.
5. Overloaded operators.
6. Accessor functions. (Get and set functions)
7. General functions.
8. Friend functions.
9. Member variables.

Friend classes and functions should in most cases be avoided.

The order of the class members in the implementation (`.cpp`-file) shall be the same as in the header file.

```
class Well {
public:
  friend WellTransform;
  Well(int len);
```

```
  Well(const Well& well);
  ~Well();
  Well         & operator=(const Well& well);
  const double   operator[](int index) const;
  double       & operator[](int index);
  int            GetLength() const;
  void           DoSomethingWithWell();
  friend void    SmoothWell(Well& well);

protected:
  Well();
  void resize(int new_len);

private:
  void DoSomeVectorTrick();
  int                 len_;
  std::vector<double> log_;
};
```

3.7 Variable initialization in constructors

All member variables of a class must be explicitly initialized after a constructor has been called. C++ implements implicit initialization, but for explicit initialization should be used to make the code clearer.

Variables should normally be initialized in the constructor instead of being assigned in the constructor body. If a special version of the constructor for the parent class should be called, this is done in a similar fashion:

```
class Grid {
public:
  Grid(int nx, int ny, int nz);
private:
  int                 nx_;
  int                 ny_;
  int                 nz_;
  std::vector<double> values_;
};

Grid::Grid(int nx, int ny, int nz)
  : nx_(nx),
    ny_(ny),
    nz_(nz) {}

class StormGrid : public Grid {
public:
  StormGrid(int nx, int ny, int nz, double missing_val);
private:
  double missing_val_;
};
```

```
StormGrid::StormGrid(int nx, int ny, int nz, double missing_val)
  : Grid(nx, ny, nz),
    missing_val_(missing_val) {}
```

4 Types

4.1 Default types

If there is no special requirements on the variable types, the following types should be used as default:

- `int` should be used for integers.
- `double` should be used for floating point numbers.
- `std::string` should be used for textual data.
- `std::vector` should be used for tabular data.

4.2 Standard types

The standard library defines some special types for quantities that may be represented differently on different platforms. These types should in simple cases be used directly instead of being casted to a generic type. Examples of such types are `size_t` that is used for size of standard containers, and `time_t` that is used for measuring time in seconds:

```
time_t      now = time(NULL);
vector<int> vec(100);
size_t      vec_size = vec.size();
for (size_t i = 0; i < vec_size; ++i) {
  vec[i] = 5 * i;
}
```

However, if you do computations involving these types, all variables involved in the computation will be casted to unsigned, which may cause strange results. Also note that unsigned variables wrap at 0, so the expression

```
for(size_t i = vec.size(); i >= 0; i--)
```

will never terminate. If in doubt, use casting to generic signed type.

4.3 Literals

Use the upper-case suffixes 'U', 'UL' and 'L' for `unsigned int`, `unsigned long` and `long` integer literals. Use the upper-case suffixes 'F' and 'L' for `float` and `long double` floating-point literals.

```
const int          MY_CONST = 42;
const double       PI       = 3.14159265358979323846;
const float        FLOAT_PI = 3.14159265F;
const long double  LONG_PI  = 3.14159265358979323846264338327950291;
const unsigned int len      = 192U;
const long         long_len = 1024L;
```

4.4 Placement of * and &

Place the * or & right in front of variable name when dereferencing or taking reference of a variable. A space between these and the variable indicates pointer and reference declarations.

```
void MyFunc(const int& my_int) {
  int * my_int_pointer = &my_int;
  int   another_int    = *my_int_pointer;
}
```

4.5 Function parameters

4.5.1 Parameter types

Big objects should be given as constant reference arguments. All reference or pointer arguments shall be declared `const` if they are not modified.

```
int CalculateSize(const BigObject& obj) {
  return obj.GetSize();
}
```

instead of

```
int CalculateSize(BigObject obj) {
  return obj.GetSize();
}
```

In the last case the object will be copied when the function is called.

4.5.2 Parameter ordering

Function parameters should be order so that input parameters come first, and output parameters last.

```
void Grid::GetIJK(size_t index, size_t &i, size_t &j, size_t &k) const;
```

4.5.3 Default parameters

Try to avoid using long lists of default parameters. It will often be clearer if multiple overloaded versions of the function are given.

5 General coding style

5.1 Declare local variables when used

Local variables should be declared when they are first used, and should have the smallest possible scope.

```
void MyFunc {
     .
     .
  int i = 0;
  while (well[i] == 0) {
    ++i;
  }
    .
    .
```

```
}
```

5.2 Importing namespaces into global namespace

Importing of namespaces should generally be avoided, since the namespace information makes it easier to see where a symbol is defined. This also prevents collisions if the same symbol is defined in more than one namespace.

The preferred syntax is to use the complete symbol name including the namespace:

```
int main() {
  std::cout << "Hello World!\n";
}
```

5.2.1 Never import a namespace in a header file

Namespaces should **never** be imported into header files. The reason for this is that this will import the namespace in all the source files that include this header file, something that can lead to obscure name collisions.

5.2.2 Namespace aliasing

In some cases the namespace names can be very long, especially in association with nested namespaces. In these cases namespace aliasing can be used:

```
namespace Sim = Com::Statoil::Find::PCube::Simulator;

int main() {
  std::cout << "Running PCube!\n";
  Sim::PCube();
}
```

5.2.3 Use of namespaces

Be conservative with the number of namespaces you create. In particular, try to avoid namespace nesting, unless the inner namespace is not intended to be accessed from outside the outer namespace.

5.3 Derived classes

Be conservative with the use of derived classes. More than one level quickly leads to problems with reading the class.

5.4 Memory management

- `new` and `delete` should always be used for memory management instead of C-style `malloc` and `free`.
- If possible, avoid allocating memory on the free store using `new`. This makes it easier to prevent memory leaks, for example when an exception is thrown.

  Use:

  ```
  void ModifyGrid(Grid& grid);

  void MyFunc() {
    Grid grid(nx, ny, nz);
    ModifyGrid(grid);
  ```

```
}

Instead of:
void ModifyGrid(Grid* grid);

void MyFunc() {
  Grid* grid = new Grid(nx, ny, nz);
  ModifyGrid(grid);
  delete grid;
}
```

If `ModifyGrid` in the previous examples throws an exception, this would result in a memory leak in the last case where `new` and `delete` was used.

- C-style arrays should never be used, except as private member variables inside container classes.

```
Use:
  std::string      name = "Test";
  std::vector<int> data(5);

Instead of:
  char * name = "Test";
  int  * data = new int[5];
```

- Variables should whenever possible be passed as references instead of pointers in function headers.

```
Use:
  int MyFunc(MyClass            & object_that_will_be_changed
             const AnotherClass & input_object);

Instead of:
  int MyFunc(MyClass            * object_that_will_be_changed
             const AnotherClass * input_object);
```

- If data is transferred to a class member function as a pointer or if a member function returns a pointer it should be clearly documented if it is the class or the caller of the function that is responsible for deleting the pointer.

5.5 Increment operator

Do not use ++i or i++ in complex statements, since it makes it much more difficult to see what a statement does. If the increment operator is used in a complex statement the intended behavior should be documented.

```
Use:
  ++i;
  my_list[i] = 0;

Instead of:
  my_list[++i] = 0;
```

5.6 Expressions inside function calls

Nested function calls should not be used, with exception of simple get-functions.

Generally, one should be careful with expressions inside function calls to functions that take more than one argument. The order of evaluation of the parameters is dependent on the compiler.

```
DO NOT DO THIS:
int F(int i, int j) {
  return i % j;
}

void MyFunc() {
  i = 1;
  F(i++, i++);
}
```

In the example above it depends on the compiler if `F(2, 3)`, `F(3, 2)` or `F(3, 3)` is evaluated.

For this reason nested function calls like `F(G(), H())`, should be avoided.

```
Do:
  double g = G();
  double h = H();
  double f = F(g, h);

Instead of:
  double f = F(G(), H());
```

5.7 Comment out unused parameters

Unused function parameters should be commented out. Functions with unused parameters sometimes appears in association with inheritance.

In the following case the function `GetZ` takes the x and y positions as parameters. However, these parameters are not needed for constant surfaces, and are commented out.

```
class Surface {
public:
  virtual double GetZ(double x, double y) const;
}

class ConstantSurface : public class Surface {
public:
  virtual double GetZ(double /*x*/, double /*y*/) const {
    return z_;
  }

private:
  double z_;
}
```

6 Comments

Good documentation of the code is important, but self-documenting code should not be commented.

C++ style comments should be used; Use //... instead of /* ... */ for commenting the rest of the line.

6.1 Standard header

All files should start with the following line stating version and modification date of the file:

```
// $Id: fileio.hpp 159 2008-10-09 05:45:44Z anner $
```

The `$Id: ... $` is part is filled out by Subversion every time the file is checked in provided that the `svn:keyword` property is set for the file.

6.2 Doxygen documentation of header files

The header file should contain all information needed by users of your class. Use /// or ///< to make comments available for doxygen. The text up to the first period is a brief description of the object. The rest is for a more detailed description. Please document the input and output parameters and return values of functions. Possible exceptions thrown by the function should also be documented.

For more information about doxygen see http://www.doxygen.org.

```
/// Root finder.
/// This class is used to find
/// exact solutions of quartic (biquadratic), cubic,
/// and quadratic polynomials with real coefficients.
/// The class is tested in a test program in \c polynomial_test.c.

class Polynomial {
public:
  /// Constructor.
  /// @param[in] coefficients an array of polynomial coefficients.
  /// @param[in] order the order of the polynomial.

  Polynomial(double* coefficients, unsigned int  order);

  /// Solver. This command solves the equation.
  /// @param[out] real components (must be preallocated).
  /// @param[out] imaginary components (must be preallocated if used).
  /// By default the imaginary array is a null pointer,
  /// in which case only real solutions are found.
  /// @return number of solutions.

  unsigned int Solve(double* real,
                     double* imaginary = 0);
```

7 The header file

7.1 Include guards

Include-guards **shall** be used in all header-files. The include-guards should consist of the directory name and file name to make sure that it is unique.

```
#ifndef NRLIB_GEOMETRY_POLYGON_HPP
#define NRLIB_GEOMETRY_POLYGON_HPP

// Header-file code

#endif // NRLIB_GEOMETRY_POLYGON_HPP
```

7.2 Forward declarations and includes

To avoid needless compile-time dependencies avoid the use of includes in header files whenever possible. It is often sufficient to put a forward declaration in the header file and use the include statement in the source file.

This being said, do not try do forward declare any part of the Standard Template Library (STL). For streams use the forward declaration file `<iosfwd>`.

```
#include <iosfwd>
#include <string>

namespace NRLib{
  class Point;

  class Line {
  public:
    Line(const Point& from, const Point& to);
    WriteToFile(const std::string& filename);
    WriteToFile(std::ofstream& fout);
  private:
    Point from_;
    Point to_;
  }
}
```

8 Error handling

Exceptions should be used for error handling inside library code. All exceptions should implement a function what() that describes the exception. All exception should directly or indirectly be a subclass of std::exception.

```
namespace NRlib {
  namespace Util {
    class Exception : public std::exception {
    public:
```

```
      Exception(const std::string& msg = "")
        : msg_(msg) {}

      std::string what() const {
        return msg_;
      }
    private:
      std::string msg_;
    };

    class FileIOError : public Exception {
    public:
      FileIOError(const std::string& msg = "") : NRlib::Exception(msg);
    };
  }
}


void FileParser::OpenFile(std::string filename) {
  fin_ = new std::ifstream(filename.c_str());
  if (!fin) throw NRlib::Util::FileIOError("Error opening file " + filename);
}

void MyFunc() {
  try {
    FileParser parser;
    parser.OpenFile(filename);
  } catch (NRlib::Util::Exception& e) {
    std::cerr << "Exception occurred when opening file: " << e.what() << "\n";
    std::abort();
  } catch (std::exception& e) {
    std::cerr << "A non-nrlib exception occurred: " << e.what() << "\n";
    std::abort();
  }
}
```

9 Standard tools

9.1 Version control

A version control system should be used in all projects. The default version control system is Subversion.

9.2 Tools for detection of memory bugs

All production code should routinely be run through a tool checking for memory bugs and memory leaks.

Two such programs that are available at SAND are Purify and Valgrind.

9.3 Issue tracker

Large projects should use a issue tracker to track bugs, suggested improvements and software versions.

In SAND https://jira.nr.no is the standard issue tracker.

It is also possible to integrate JIRA and Subversion, so that JIRA tasks are linked to the corresponding Subversion revisions. To obtain this all comments for Subversion commits must be tagged with a JIRA task ID and JIRA must be configured to parse the Subversion project.

Index