Note

The C++ coding standard of SAND

Note no SAND/07/05

Authors Per Røe
Harald H. Soleng

Date November 17, 2005

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 of SAND**
Authors	**Per Røe , Harald H. Soleng**
Date	November 17, 2005
Publication number	SAND/07/05

Abstract

We have now the honor to submit to the consideration of the SAND group in Σ assembled, that C++ coding standard which has appeared to us the most advisable.

Keywords	standardization
Target group	
Availability	
Project	
Project number	
Research field	
Number of pages	22

Contents

1 Naming conventions

Use descriptive names. Names with a short scope can have shorter names, but all names with global or template 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 `.h` for source and header files, respectively.

```
/parser/fileparser.cpp
/parser/fileparser.h
```

1.2 Namespaces

Namespaces shall have names starting with an uppercase letter. If the name of the namespace contains of several words each should word should begin with an uppercase letter. Don't 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

Please don't use preprocessor macros unless strictly it is nessessary. In most cases templates, inlined 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 shold be configured in such a way that it only inserts spaces, since tabs can be visualised with variable width on different platforms.

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

```
    }
  }
}
```

2.2 Line length

Line length should not exceed 79 characters, and overflowing lines should be alligned accordingly.

```
int MyClass::MyFunction(const MyType&
                        this_is_a_very_long_variable_name,
                        int n_iter,
                        MyType& another_variable) { }
```

2.3 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'';
  ```

- Space before and after operators except the not operator !:

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

- But the space before and after operators can be let out to visualise 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_;
  }
  ```

2.4 Brackets

Rules for placing of brackets:

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

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

- 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 should either be defined as `protected`, or preferably `private`. If for some reason most or all member variables should be public, the `struct` keyword should be used instead of the `class` keyword.

3.2 Accessors

- Use functions whose names start with get and set for access of class members:

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

private:
  int val_;
}

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

- Use an overloaded `[]`-operator for access of data from an one-dimensional datastructure. 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 does 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 the how the function works.

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

```
    :
}
```

3.4 Lazy evaluation of class members

In some cases a class can have a member which takes long time to compute. In this case it might be feasible to wait with computing the member until it is really needed. In such cases we can get a accessor function that is logically const, but that calculates the member value the first time it is called. In such cases the accessor function should be `const`, and member variable should be declared `mutable`.

```
class MyClass {
    :
  public: double GetVar() const;
    :
private:
  // Only evaluated when needed.
  mutable double var_;
  bool var_evaluated_;
}

MyClass::getVar() {
  if (var_evaluated_) {
    return var_;
  } else {
    var_ = CalculeVar();
  }
}
```

The same effect can also be achieved by casting away the const-ness of the object, but this is not guaranteed to give a predictable result in cases where the object was originally defined as a const.

3.5 Rule of 3

When a class needs a special copy constructor, an copy assignment operator or a destructor it usually needs all three. This is typically the case if the class handles some resources like allocated memory, files, etc. 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 assignemnt operator.

```
class MyClass {
public:
  MyClass(int size);
  ~MyClass();
private:
  double* data;

  // Not implemented.
  MyClass(const MyClass& obj);
  MyClass& operator=(cont MyClass& obj);
```

```
}

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

MyClass::~MyClass() {
  delete [] data;
}
```

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, thereafer 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 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();
  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 initialisation in constructors

Variables should be initialised 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 Integer types

`int` should be used as the default integer type.

For specific uses as for example size of an object or time representation the built in types for example size_t, time_t should be used.

```
vector<int> vec(100);
size_t vec_size = vec.size();
time_t now = time(NULL);
```

4.2 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        PI_F     = 3.14159265F;
const long double  PI_L     = 3.14159265358979323846264338327950 29L;
const unsigned int len      = 192U;
const long         long_len = 1024L;
```

4.3 Placement of * and &

Place the * or & after right after type when declaring reference or pointer types. Place right in front of variable name when dereferencing or taking reference of a variable.

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

4.4 Function parameters

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();
}
```

5 General coding style

5.1 Declare local variables when used

Local variables should be declared when they are first used.

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

5.2 Importing the std namespace into global namespace

Be careful with importing the std namespace using `using namespace std` directives. Often it is better to use the complete name, including the namespace. `using` directives if used should only be used in cpp-files, **never** in h-files.

5.2.1 Using full names

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

5.2.2 Importing only a single function

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

5.2.3 Importing the complete std namespace

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

5.3 Import namespaces locally when needed

```
// Get access to NRlib utilities throughout
using namespace NRlib::Util;
int main(unsigned int argc, char* argv[]) {

  { // Begin scoping bracket

    // Get access to NRlib User Interface namespace
    using namespace NRlib::UI;

    Program& program =
    Singleton<Program>::Instance();

    program.Initialize("Demo program", // Name
                       0, // Major version
                       1, // Minor version
                       0, // Patch number
                       "Testing", // Purpose.
                       "Demo of NRlib", // Program description
                       2005, // First year copyright
                       "Norwegian Computing Center" // Copyright holder
                       );

    Options& options =
      Singleton<Options>::Instance();
    options.Initialize();
    options.Read(argc, argv);
```

```
  } // end scoping

  // Other code
}
```

5.4 Memory management

- `new` and `delete` should always be used for memory management instead of C-style `malloc` and `free`.

- Whenever possible allocate memory on the stack instead of on the heap. 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.

5.5 Definition of index in for-loops

The iterator index for a `for`-loop should always be defined before the start of the `for`-loop.

The reason for this is that although the scope of the iterator index is clearly defined in the C++-standard to be inside the `for`-loop, some compilers among them Visual C++ puts the iterator index in the parent scope.

```
Use:
int i;
for (i = 0; i < len; ++i) {
  values[i] = 0;
}

Instead of:
for (int i = 0; i < len; ++i) {
  values[i] = 0;
}
```

5.6 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 used in the statement the behaviour should be documented.

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

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

For complex datatypes the prefix operator `++i` should be used instead of `i++`, since it can be implemented more effeciently since the previous value is not stored.

5.7 Iterators

For containers like vectors and lists iterators should be used instead of indexes, since they often give more efficient code that indexes, and since it makes it easier to replace the container.

```
void Init(std::vector<int> values) {
  std::vector<int>::iterator it;
  for (it = values.begin(); it != values.end(); ++it) {
    *it = 0;
  }
}
```

5.8 Expressions inside function calls

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 compilator.

```
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 compilator 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());
```

6 Comments

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

6.1 Standard header

All files should start with a standard header giving the author, date, and giving a short description of the file.

```
// $Id: crava.cpp,v 1.69 2005/10/04 15:58:41 ok Exp $
/** @file
  Implementation of main routines for CRAVA.
  @author  Odd Kolbjørnsen, Norsk Regnesentral
  @date    7/9 2004
*/
```

The `$Id: ... $` is part is filled out by CVS or Subversion every time the file is checked in. The rest of the comment is formatted so that it can be used to generate documentation by doxygen.

6.2 Documentation of headerfiles

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 parametres and return values of functions, and don't forget to put your name in the author field. Bugs are bad, but anonymous bugs are even worse!

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.
   @author Harald H. Soleng, Norwegian Computing Center
*/
class Polynomial {
public:
  /**
     Constructor.
     @param[in] coefficients an array of polynomial coefficients.
     @param[in] order the order of the polynomial.
   */
  Polynomial(double* coffecients, 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 be made up of catalog name and filename to make sure that it is unique.

```
#ifndef NRLIB_POLYGON_H
#define NRLIB_POLYGON_H

// Header-file code

#endif // NRLIB_POLYGON_H
```

In Visual Studio `#pragma once` does the same thing, but this is not portable.

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 enter a forward declaration, e.g., `class Polynomial` in the header file and use the include statement in the source file. This being said, don't try do declare any part of the Standard Template Library (STL). For streams use the forward declaration file `iosfwd`.

```
class Point;

class Line {
public:
  Line(Point from, Point to);
private:
  Point from_;
  Point to_;
}
```

8 Error handling

Exceptions should be used for errorhandling 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 occured when opening file: " << e.what()
              << std::endln;
    std::abort();
  } catch (std::exception e) {
    std::cerr << "A non-nrlib exception occured: " << e.what()
              << std::endln;
    std::abort();
  }
}
```

9 Standard tools

9.1 Version control

A version control system should be used in all larger projects. In internal projects either CVS or subversion should be used.

A file checked out with a CVS or subversion client for Windows should not be edited from UNIX and visa versa. UNIX and Windows has different standards for line shifts, and the clients are able to translate between different version so that files checked out on Windows has Windows line-shifts, while files checked out on UNIX has UNIX line-shifts.

9.2 Tools for detection of memory bugs

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

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

Index