OOPS C++ Coding Guidelines

Created by Yannick Tremolet, last modified on Aug 02, 2013

The following guidelines are based on a draft document from Baudouin Raoult, and were updated following the OOPS code review 11-15 July 2011.

Files Extensions

- Each class is defined in two files:
  - The header file contains the class definition and has the extension .h.
  - The implementation file defines the methods and has the extension .cc
- The file names should match the name of the class.
- .cc and .h files must be in the same directory.

Header Files

- Header files should follow the layout suggested in Documents/ExampleHeader.h
- The public, protected and private sections of a class should be declared in that order (i.e. public first, private last).
- Use #include guards to protect against multiple inclusions.
- The guard name should be constructed from the path to the .h file, as it appears in a #include statement (see below), but with lowercase characters converted to uppercase, special characters replaced by underscores, and with a trailing underscore.
- The #endif for the guard should be followed on the same line by // and then the name of the guard.
- All headers should be self-sufficient. A header should compile by itself.
- There should be one main class per file. Helper classes are allowed, as long as they are only used from within the file.

Code Documentation

- Documentation is generated using doxygen.
  - Instructions on building the documentation are in the README file.
  - Have a look at the Doxygen manual.
  - See instructions for instructions on including \f$\mbox{\LaTeX}\f$ mathematical formulae.
- Doxygen recognises special comments:
  - A brief comment is a single line like this:
    /// This one-line comment gives a brief description of a class or method.
  - A detailed comment looks like this:

/*!

```
 * This is a detailed comment. It has more than one line,
 * and provides more complete information about a class or method.
 */
```

- Class definitions should be preceded by a doxygen brief comment followed by a detailed comment.
- Method and member declarations should be preceded by a doxygen brief comment, unless the role of the method or member is completely obvious from its name. A detailed comment may also be provided.
- Normal C++ comments should be used in .cc files where necessary to explain the internal logic of a function.
- The code should be self-explanatory. But, add comments to explain complex algorithms.
- Do not comment-out code. Use the source-code management system!
- Don't include comments to indicate authorship or modification history. That is what git blame is for!
- HTML links to auxiliary documents (e.g. pdf files) can be made to appear on the Overview page by adding them to Documents/overview.h

Includes

- Do not include unnecessary headers. Use class forwarding. The compiler only needs to see a class definition when calling methods or establishing the size of an object. When referring to a pointer or a reference, the compiler does not need to know the detail of the class.
- #include<iosfwd> instead of #include<iostream>, if possible.
- Included ".h" files must use the full path from the build directory ("oops")
- Order of header files:
    - Own ".h" file.
    - C System headers (e.g. <unistd.h>).
    - C++ system headers (e.g. <iostream>).
    - Other library headers <boost/...>).
    - Oops/util/logger project headers.
- Within each of these categories,use alphabetical ordering.

Identifiers

- Identifiers should be in "camel case". That is, they should be mainly in lower case, with an upper case letter at the start of each internal word: e.g. changeResolution.
- Do not use underscores to separate words.
- Class names should start with an upper case letter.
- Method and member names should start with a lower case letter.
- Member names have an underscore at the end.
- Use short identifiers for local variables, loop indices, etc.
- Use longer, meaningful names for methods, members, classes, etc.

Use of Const

- Use const wherever possible.
- Avoid passing objects by const value. Pass by const reference instead.
- Remember. The rule when reading definitions is to work from right to left. So, for example, char * const test means that test is a const pointer to a char object:
  - const char * test = "xyz"; // non-const pointer to const data
  - char * const test = "xyz"; // const pointer to non-const data
  - const char * const test = "xyz"; // const pointer to const data
  - bool isEmpty() const; // the isEmpty method does not change its object.

How to Design Classes

- By default make classes noncopyable (using boost::noncopyable), unless copy is needed
- If you need copy, assignment or a destructor, then you probably need all three.
- Make interfaces non-virtual.
- Make virtual functions private.
- Check for assignment to self in operator=.
- A lot more guidelines needed here...

Interface Principle

- Guideline to be written...

Templates versus Inheritance

- Guideline to be written...

Proper Inheritance

- Remember the Liskov substitution principle: don't derive a "square" from a "rectangle"!
- More guidelines required here...

Constructors and Destructors

- Always declare copy and assignment constructors. Make them private unless you need to copy.
- If the default copy constructor is sufficient, include a comment to this effect in the class definition.
- Write the constructors and the destructors at the same time.
- If the class has a virtual table, its destructor must be virtual.

- All resources allocated by an object must be deallocated in the destructor.
- Beware of partially constructed objects.
- Except for the copy constructor, single-argument constructors should be declared explicit to prohibit implicit type conversions.
- Base class destructors must be either public and virtual, or protected and not virtual.
- The copy constructor should copy all the data. However, you may wish to give it a second, default argument to allow this behaviour to be over-ridden.
- Do not code MyClass a = b;
    - This looks like assignment, but in fact calls the copy constructor.
    - Code MyClass a(b); instead.

Member Variables

- Member variables should always be private.
- Use accessors if you need to access a member variable from outside a class.
- Don't use the accessors from within the class. Use the member itself.

Accessors

- Accessors (a.k.a. getters and setters) should only be implemented if necessary. They break the encapsulation.
- Accessors should be inline.
- Accessors must have the same name as the member (but without the underscore) for example, the accessors for a member Foo foo_; should be:
    - const Foo& foo() const {return foo_; }
    - void foo(const Foo& f) { foo_ = f; }

Methods

- A method is a request to an object to do something or to provide something. The name of the method should reflect this. - E.g. changeResolution is preferable to resolutionChanger.
- If a method is virtual in a base class, declare it as virtual in all derived classes that override it.

Operator Overloading

- Don't do it unless it is meaningful.
- Don't subvert the mathematical properties (associativity, etc.).
- Don't use an operator for conversion. Implement an "asDouble" method rather than "operator double()".

Pointers and References

- Prefer references to pointers. If an object is guaranteed to exist, use a reference.

- Passing or returning a non-const pointer means passing ownership of the pointed object.
- Passing or returning a const pointer means keeping ownership of the pointed object, and that the pointed object can be null
- In any other case, pass a reference to the object. Use const whenever the object will not be modified.

Use of Static

- Avoid static if possible.
- Be aware there are different types of static (function-local, file-scope).
- Be aware that static variables cause problems in multi-threaded applications.

Use of Casts

- Use c++ style casts.
- Avoid downcasting. It is a symptom of bad inheritance, or not enough functionality in the base class
- Write "double(expression)", not "(double) expression"
- Guideline for the use of const cast to be written....

C Code

- Don't use C functions (e.g. printf) if C++ provides the same functionality.
- If you must use a C function, prefix it with a double colon (e.g. "::sleep(10)")
- When possible, wrap any C function in a C++ object (e.g. Sleeper)
- Never use C style casts.
- For unsigned value, use a typedef: typedef unsigned long ulong;

Preprocessor

- The preprocessor should only be used to define \#include guards in .h files and for variables specified via the -D flag at compile time.
- The preprocessor should not be used to define macros or constants.
    - The only permitted macros are ABORT, ASSERT and LOG, and macros defined in the boost library (e.g. BOOST_AUTO_TEST_CASE).
- Don't pepper the code with ifdef's for machine/compiler dependent conditional compilation. Put any such code in a header file that can be included wherever needed.

Namespaces

- Model-independent code should be defined in the oops namespace.
- Model-specific code should be defined in a separate, model-specific namespace.
- Do not use an entire namespace (i.e. using directive).

- By preference, use explicit namespace qualifications (e.g. std::string). However, using std::string etc. is acceptable.
- Using statements must never be used at global scope in a header file.
- Use anonymous namespaces to restrict classes (e.g. Factories) to file scope.

Readability

- As far as possible, adhere to the rules listed in the Google C++ Style Guide. Note however that, contrary to the Google rules:
  - We do use streams.
  - We allow non-const references as arguments.
- Use cpplintto check your code. You may wish to turn off the following cpplint filters:
  - build/include_alpha (Because our idea of alphabetic order is different from Google's.)
  - build/include_order (Because cpplint wrongly thinks boost header files are c-system header files.)
  - readability/streams (Because we use streams.)
- Keep lines below 80 characters.
- Tab characters are not allowed.
- Indent class and function bodies, if and for blocks, etc.
  - public, private and protected labels in a class definition should be indented one space with respect to the start of the class definition.
  - Use a two space indent for everything else.
  - It is preferable to indent code inside a namespace block. (However, we have many examples where this is not done.)
- Split long lines in a way that makes it obvious that the code continues on the next line.
- Continuation lines should be indented.
- If you split an argument list, align the arguments with those on the previous line.
- The opening brace should appear on the same line as the argument list, initialisation list, loop expression. etc.
- The closing brace should appear on its own line, and aligned with the start of the statement it closes.
- Braces should be used for all control structures (if, for, switch, etc.), even for "one-liners".
- The else statement should be on the same line as the closing brace of the preceding block, and the opening brace of the following block.
- Don't declare more than one member per line.
- Don't initialise on the same line as you declare: (e.g. int i=3;).
- Only one statement per line.
- Remove whitespace at the end of a line.
- Add a space after a comma in an argument list
- All operators, except "!" should be surrounded by spaces.
- Separate inline comments from code by at least two spaces
- There should be a space after // (or after /// in the case of a Doxygen inline comment).
- If the initialisation list in a class definition is too long to be on the same line, put in on the next line with the colon indented by 4 spaces.

Optimization

- Pass objects by reference, not by value.
- Prefer initialization over assignment.
- Use ++i, not i++ when incrementing iterators.
- Use the initialization list to initialize member objects.

Logging

- All logging messages should use the Logger class. Do not write to cout or cerr.
- The logger adds a newline at the end of each message, so you don't need to.
- endl forces an unnecessary buffer flush. Use \n instead.
- Use the appropriate logging category:
    - Info is for normal output
    - Trace is for more verbose output that could help the user understand the logical flow of the program.
    - Warning is for non-fatal error messages.
    - Error is for fatal error messages.
    - Configs is for echoing configuration data.
    - Debug is for debugging. Code in the shared repository should not output to this category.

ABORT, ASSERT and Error Handling

- Errors should abort
- Use the ABORT macro to exit after an error. Do not call exit directly.
- Use the ASSERT macro liberally. It compiles to nothing unless the CHECK_ASSERTS macro variable is set, and it helps the reader to understand the code.
- Remember, asserts can be disabled. Your code should not change behaviour if you disable the asserts. Use "if" and ABORT if you want something to be always checked.
- Do not use exceptions (try/catch/throw)

Pointers and Smart Pointers

- Use references instead of pointers as much as possible.
- Use smart pointers in preference to c pointers.
- Use boost::scoped_ptr if possible, otherwise boost::shared_ptr.
- Do not use auto_ptr.

Return Values

- Use return values instead of argument where possible.
- However, do not assume that the compiler will perform return value optimization.

Interfacing Fortran and C++

- use ISO_C_BINDING
- Only pass pointers and scalar variables between Fortran and C++
- naming convention (to be written...)
- functions (to be written...)
- parameters (to be written...)
- const (to be written...)
- order (input first, output last) (to be written...)
- c prototype of Fortran function - generate automatically? to be written...)

Private, Public and Protected Access

- Do not use "protected".
- More rules please

Build

- Tomas to write this!

Directory Structure

- Code for each library should be in its own directory.
- Models use OOPS, they are not part of OOPS thus source code for models should be kept outside the OOPS directories.

No labels