C++ Design/Coding Guidelines

francesco.giacomini@cnaf.infn.it

Milano, 25th March 2004

Overview

- Basic assumptions
- Design Guidelines
- Coding Guidelines

Basic Assumptions

- These conventions are for C++ only
- Many (most) are applicable to C++ only
- Importance of C++ idioms
- Conventions inherited from other languages (typically Java and C) often not applicable/adequate/convenient or even harmful

Design Guidelines

- More objective (less subjective)
- Come from 20 years of experience in the community
- Not much disputable
- Sources
  - “Effective C++”, “More Effective C++”, “Effective STL”
  - www.aristeia.com, www.gotw.ca, www.boost.org, www.cuj.com, www.curbralan.com

Coding Guidelines

- More subjective (less objective)
- More disputable but important to have
- Sources:
  - www.boost.org
  - http://www.possibility.com/Cpp/CppCodingStandard.html
  - http://www.doc.ic.ac.uk/lab/cplus/c++.rules/
    - "Ellemtel Rules"

Effective C++

- Shifting From C to C++.
  - Prefer const and inline to #define.
  - Prefer iostream to stdio.h.
  - Prefer new and delete to malloc and free.
  - Prefer C++-style comments.
- Memory Management.
  - Use the same form in corresponding uses of new and delete.
  - Use delete on pointer members in destructors.
  - Be prepared for out-of-memory conditions.
  - Adhere to convention when writing operator new and operator delete.
  - Avoid hiding the "normal" form of new.
  - Write operator delete if you write operator new.

Effective C++

- Constructors, Destructors, and Assignment Operators.
  - Declare a copy constructor and an assignment operator for classes with dynamically allocated memory.
  - Prefer initialization to assignment in constructors.
  - List members in an initialization list in the order in which they are declared.
  - Make destructors virtual in base classes.
  - Have operator= return a reference to *this.
  - Assign to all data members in operator=.
  - Check for assignment to self in operator=.

Effective C++

- Classes and Functions: Design and Declaration.
  - Strive for class interfaces that are complete and minimal.
  - Differentiate among member functions, non-member functions, and friend functions.
  - Avoid data members in the public interface.
  - Use const whenever possible.
  - Prefer pass-by-reference to pass-by-value.
  - Don't try to return a reference when you must return an object.
  - Choose carefully between function overloading and parameter defaulting.
  - Avoid overloading on a pointer and a numerical type.
  - Guard against potential ambiguity.
  - Explicitly disallow use of implicitly generated member functions you don't want.
  - Partition the global namespace.

Effective C++

- Classes and Functions: Implementation.
  - Avoid returning "handles" to internal data.
  - Avoid member functions that return non-const pointers or references to members less accessible than themselves.
  - Never return a reference to a local object or to a dereferenced pointer initialized by new within the function.
  - Postpone variable definitions as long as possible.
  - Use inlining judiciously.
  - Minimize compilation dependencies between files.

Effective C++

- Inheritance and Object-Oriented Design.
  - Make sure public inheritance models "isa."
  - Differentiate between inheritance of interface and inheritance of implementation.
  - Never redefine an inherited nonvirtual function.
  - Never redefine an inherited default parameter value.
  - Avoid casts down the inheritance hierarchy.
  - Model "has-a" or "is-implemented-in-terms-of" through layering.
  - Differentiate between inheritance and templates.
  - Use private inheritance judiciously.
  - Use multiple inheritance judiciously.
  - Say what you mean; understand what you're saying.

Effective C++

- Miscellany
  - Know what functions C++ silently writes and calls.
  - Prefer compile-time and link-time errors to runtime errors.
  - Ensure that non-local static objects are initialized before they're used.
  - Pay attention to compiler warnings.
  - Familiarize yourself with the standard library.
  - Improve your understanding of C++.

More Effective C++

- Basics.
  - Distinguish Between Pointers and References.
  - Prefer C++-Style Casts.
  - Never Treat Arrays Polymorphically.
  - Avoid Gratuitous Default Constructors.
- Operators.
  - Be Wary of User-Defined Conversion Functions.
  - Distinguish Between Prefix and Postfix Forms of Increment and decrement operators.
  - Never Overload &&, ||, or,.
  - Understand the Different Meanings of New and Delete.

More Effective C++

- Exceptions.
  - Use Destructors to Prevent Resource Leaks.
  - Prevent Resource Leaks in Constructors.
  - Prevent Exceptions from Leaving Destructors.
  - Understand How Throwing an Exception Differs from Passing a Parameter or Calling a Virtual Function.
  - Catch Exceptions by Reference.
  - Use Exception Specifications Judiciously.
  - Understand the Costs of Exception Handling.

More Effective C++

- Efficiency.
  - Remember the 80-20 Rule.
  - Consider Using Lazy Evaluation.
  - Amortize the Cost of Expected Computations.
  - Understand the Origin of Temporary Objects.
  - Facilitate the Return Value Optimization.
  - Overload to Avoid Implicit Type Conversions.
  - Consider Using Op= Instead of Stand-Alone Op.
  - Consider Alternative Libraries.
  - Understand the Costs of Virtual Functions, Multiple Inheritance, Virtual Base Classes, and RTTI.

More Effective C++

- Techniques.
  - Virtualizing Constructors and Non-Member Functions.
  - Limiting the Number of Objects of a Class.
  - Requiring or Prohibiting Heap-Based Objects.
  - Smart Pointers.
  - Reference Counting.
  - Proxy Classes.
  - Making Functions Virtual With Respect to More Than One Object.

More Effective C++

- Miscellany.
  - Program in the Future Tense.
  - Make Non-Leaf Classes Abstract.
  - Understand How to Combine C++ and C in the Same Program.
  - Familiarize Yourself With the Language Standard.

Effective STL

- Containers.
  - Choose your containers with care.
  - Beware the illusion of container-independent code.
  - Make copying cheap and correct for objects in containers.
  - Call empty instead of checking size against zero.
  - Prefer range member functions to their single-element counterparts.
  - Be alert for C++'s most vexing parse.
  - When using containers of newed pointers, remember to delete the pointers before the container is destroyed.
  - Never create containers of auto_ptrs.
  - Choose carefully among erasing options.
  - Be aware of allocator conventions and restrictions.
  - Understand the legitimate uses of custom allocators.
  - Have realistic expectations about the thread safety of STL containers.

Effective STL

- Vector and string.
  - Prefer vector and string to dynamically allocated arrays.
  - Use reserve to avoid unnecessary reallocations.
  - Be aware of variations in string implementations.
  - Know how to pass vector and string data to legacy APIs.
  - Use "the swap trick" to trim excess capacity.
  - Avoid using vector<BOOL>.

Effective STL

- Associative Containers.
  - Understand the difference between equality and equivalence.
  - Specify comparison types for associative containers of pointers.
  - Always have comparison functions return false for equal values.
  - Avoid in-place key modification in set and multiset.
  - Consider replacing associative containers with sorted vectors.
  - Prefer map::insert to map::operator[] when efficiency is a concern.
  - Familiarize yourself with the nonstandard hashed containers.

Effective STL

- Iterators.
  - Prefer iterator to const_iterator, reverse_iterator, and const_reverse_iterator.
  - Use distance and advance to convert const_iterators to iterators.
  - Understand how to use a reverse_iterator's base iterator.
  - Consider istreambuf_iterators for character by character input.

Effective STL

- Algorithms.
  - Make sure destination ranges are big enough.
  - Know your sorting options.
  - Follow remove-like algorithms by erase if you really want to remove something.
  - Be wary of remove-like algorithms on containers of pointers.
  - Note which algorithms expect sorted ranges.
  - Implement simple case-insensitive string comparisons via mismatch or lexicographical_compare.
  - Use not1 and remove_copy_if to perform a copy_if.
  - Use accumulate or for_each to summarize sequences.

Effective STL

- Functors, Functor Classes, Functions, etc.
  - Design functor classes for pass-by-value.
  - Make predicates pure functions.
  - Make functor classes adaptable.
  - Understand the reasons for ptr_fun, mem_fun, and mem_fun_ref.
  - Make sure less means operator<.

Effective STL

- Programming with the STL.
  - Prefer algorithm calls to hand-written loops.
  - Prefer member functions to algorithms with the same names.
  - Distinguish among count, find, binary_search, lower_bound, upper_bound, and equal_range.
  - Consider function objects instead of functions as algorithm parameters.
  - Avoid producing write-only code.
  - Always #include the proper headers.
  - Learn to decipher STL-related compiler diagnostics.
  - Familiarize yourself with STL-related web sites

Other Design Guidelines

- Open/close principle
- Prefer free to member functions
- Consider moving private members to the implementation file
  - pimpl idiom (handle/body, compilation firewall)
- Namespace names after content (not e.g. reverse domain names) and not too deep

Other Design Guidelines (cont.)

- "Early optimization is the root of all evil"
  - optimize only if it makes sense (e.g. after profiling)
  - consider the disadvantages in terms of maintainability/readability
- Consider the pimpl idiom for interface classes
- Code should be reentrant/thread safe
- Prefer code to comments
- Use third-party libs if good, in particular the standard library and boost

Other Design Guidelines (cont.)

- Header files
  - Appropriate include guards
  - Be independently compilable
  - Don't impose choices on including TUs
    - Useless includes in place of forward declarations
    - Using directives and declarations
    - Namespace aliases

Other Design Guidelines (cont.)

- Exploit RAII as much as possible
- Exceptions
  - "Do I want/can afford stack unwinding here?"
  - Don't use exception specifications, just document them
  - Derive from std::exception
  - constructor/copy constructor/operator= should be throw()
  - Don't throw in destructors
  - Code should be exception safe
    - at least basic guarantee
    - Possibly, if worth, strong guarantee or nothrow

Coding Guidelines

- File naming
  - *.cpp, *.hpp
- Variable and function naming
  - **Meaningful**, even if lengthy
  - Lowercase, words separated by '_'
  - (for variables) distinguish between member data (m_*), static class data (s_*), file scope data (f_*) and global data (g_*)

Coding Guidelines (cont.)

- Class names are capitalized
- Acronyms are preserved
- Class members:
  - public, protected, private
- Always use braces for if, while, do, for
  - Possible exception: one-liners on the same line
- Avoid reserved names
  - beginning with '_' followed by a capital letter
  - containing “__”

Coding Guidelines (cont.)

- Formatting
  - No tabs, only spaces
  - No spaces at end of line
  - No empty lines at end of file
  - 2-space indentation
  - 80-column lines
  - No parens next to keywords
  - Parens next to function declarations/definitions/calls
  - No space after the open paren and before the close paren
  - No parens in return statements, unless necessary

Coding Guidelines (cont.)

- One statement per line
- One variable declaration/definition per line
- * and & are part of the type
- const between type and */&
- Avoid embedded assignments

Conclusions

- Preserve and exploit language-specific characteristics
- Design guidelines should be accepted almost “as-is”
- Coding guidelines are more debatable but some conventions are needed