The N Commandments

Sven Mikael Persson, ReaK library

September 1, 2011

In this document ReaK's N Commandments are presented.
The layout of the Commandments tables is as follows:

n. **Short description.**
`Code example` Motivation, background and additional information. Reference

1 Commandments

1.1 Robustness

This section is on requirements that directly affect the robustness of the ReaK library and are thus the requirements with the highest level of importance. Not obeying them is a serious offense (figuratively). There are very good justifications behind each of these rules, so breaking those rules for reasons of personal preferrence or sloppiness is not acceptable, but they can be broken if strong justification can be provided in a particular case.

1. Provide unit tests with your code.
Any new code in ReaK has to be tested and validated with a unit-test program. In concrete terms, it means that every new chunk of code (one or more source files) which defines new classes or functions has to be tested within a program that remains essentially quiet (no output) as long as all the results are correct and returns a value of 0 if all successful, non-zero otherwise. In order to best guarantee that all functions and classes are not forgotten, when creating a new function, mark the doxygen comment with a `test` doxygen keyword, and replace it with `TEST PASSED` when the unit-test has been created and run successfully.

2. Header files must contain an include guard.

```
#ifndef HEADER_NAME_HPP
#define HEADER_NAME_HPP
#endif // HEADER_NAME_HPP
```

This construction is to avoid compilation errors about the C/C++ One-Definition-Rule (ODR). If the name of the header file is fairly common, prepend `REAK_` to the header-guard's name.

3. Keep the `#include` statements in the header files to a bare minimum.

Only #include, in a header file, the bare minimum number of header-files that you need in order to complete the declarations of the classes or functions in it, nothing more. Then, #include, in the source files, the other headers that you need to complete the definitions of the classes and functions. Forward-declarations can also be used if it is enough to complete the declarations in a header-file, but it is not required to make this additional effort to minimize header-file dependencies. The main motivation for this guideline is to minimize the dependency between header-files, which reduces compilation time, maintenance cost, and also promotes software design that also has fewer inter-dependencies.

4. Do not rely on second-hand inclusions.

This is not to contradict the previous commandment. Including a bare minimum of headers in a header file does not imply that you should rely on second-hand inclusions, that is, include one header and expect that that header also includes another one that you need. It is tempting to do so because it seems reasonable, but you should not. Include all the headers you need to complete your declarations (no more, no less) and don't rely on the assumption that some of the headers included certainly include this other header that you need. In other words, your header files should be self-sufficient (not making assumption about what is in other header files). One exception to this rule is, of course, the case where the particular header included is meant to include a few other headers (i.e. it is specified that it does so).

5. Compile with highest warning levels and treat them as errors.

```
GCC: -Wall -Woverloaded-virtual -Wold-style-cast
-Wnon-virtual-dtor
MSVC: /W3
```

By default, the ReaK library build is setup to compile with the highest possible warning level on whatever compiler is used. If you see warnings originating from ReaK library code, treat them as errors. If it is in your code, fix them. If not, report them. Most warning messages are there for a good reason, and they often hide a coding mistake, typo, or design flaw, in any case, they should be address with the same rigor as compilation errors.

6. Using-statements are prohibited at global or namespace scope.

```
using namespace std;
using std::vector;
```

This prevents name collisions from occurring. It is very important not to pollute the ReaK namespace or any of its sub-namespaces or the global scope by importing symboles for an alien namespace with a using-statement. The only place where using-statements are acceptable is in a function scope, that is, within the function body, because the names imported will be limited to the function scope. Nonetheless, even in a function body, it is mandatory to only use statements of the form `using std::vector;` and not a using-namespace-statement.

7. Do not include C heritage libraries in a header-file.

```
#include <cmath>
```

C++ has included, as part of its standard libraries, the C standard libraries (with a pre-pended "c" and no ".h"). Although the C++ standard requires that all the functions in the C++ version of those C libraries be within the "std" namespace, it doesn't mean that the functions only appear in the "std" namespace, most implementations also provides the functions in the global namespace. Thus, to avoid namespace pollution, it is highly recommended to avoid including these headers in a header-file, limit their use to the source files (cpp files).

8. Do not define your own general mathematical and physical constants locally.

~~`pi = 3.1415`~~

Mathematical or physical constants should be regrouped into header-files with sensible names and groupings, and which contain essentially nothing but those constants (or only a few highly related functions (e.g. a degree-to-radian conversions in the same header as the definition of "pi"). NOTE: Mathematical constants, such as `M_PI` and `M_E`, that are sometimes included in some versions of the cmath library, are not standard C/C++ constants and should not be used.

9. Uninitialized variables or data members are not allowed to exist.

~~`double temp;`~~, `double temp = 0.0;`

A variable or data member that was not given an initial value is, at best, useless, at worse, a horrible bug. All local variables should be initialized upon creation, and thus, they should be created where they are first needed (if that is not possible, initialize it to a harmless value (often zero) at the point of creation). All data members which are POD-types or do not have a default constructor should appear either in the initialization list of the constructor of the class or in its body (but the initialization list is preferred and often mandatory).

10. All C-style or plain C++ casts are essentially forbidden.

~~`(T)a`~~, ~~`reinterpret_cast<T>(a)`~~, ~~`const_cast<T>(a)`~~, ~~`static_cast<T>(a)`~~, ~~`dynamic_cast<T>(a)`~~

Both the C-style cast and the `reinterpret_cast` have undefined behavior and are thus forbidden. The `const_cast` operator can always be avoided, use the `mutable` qualifier on an exceptional data member that needs to change even within a const member function of a class. The `static_cast` operator is almost always avoidable by design, but it can be used to avoid warning messages like "comparing signed and unsigned type". For conversions, use the explicit conversion operator, i.e. `T(a)`, instead of a static cast. For doing static casting of pointers, use the Boost casting operator for the smart-pointers, that is, `boost::static_pointer_cast`. The `dynamic_cast` operator must not be used in ReaK. First, dynamic casting is usually a sign of bad design and is most often avoidable by design. If not avoidable and since ReaK should use smart-pointers almost exclusively (especially for polymorphic classes, for which a dynamic cast could be required), a plain C++ dynamic cast is not appropriate. Finally, ReaK uses its own RTTI system (Run-time Type Identification) and not that of C++, and thus, the `boost::static_pointer_cast` operator is not appropriate either because it uses the C++ RTTI. So, for dynamic casting in ReaK, use the `ReaK::rtti::rk_dynamic_ptr_cast` operator, which has the exact same effect as the Boost or plain C++ versions, but using ReaK's RTTI and smart-pointers.

11. Use `const` pro-actively.

```
| class foo_bar {
|   public:
|     bool compare_to(const foo_bar& rhs) const;
| }
```

The `const` qualifier is an integral part of the interfaces of functions and variable types. ReaK promotes a proactive use of this qualifier and since it has a infectious quality (thanks to the nice and strict type rules in C++), if you want to program in the ReaK library (or any other decent library), learn to program with const-correctness in mind.

1.2 Portability

This section is on portability issues. These are rules and guidelines to make sure that ReaK can stay portable and clean. To maximize the ease of use of ReaK and its applicability, it is very much desired that it has very few, if any, external dependencies (and only portable dependencies), and that it obeys the

C++ standard strictly.

12. Specify include directories of files from the second-level sub-directories.

~~`#include <defs.hpp>`~~, `#include <base/defs.hpp>`

The ReaK library has a top-level source directory named "ReaK" and then second-level sub-directories like "core", "ctrl" and "examples". The intended inclusion path (added to the build system) are those second-level sub-directories, and thus, all header files that are included from a source or header file and that are not in the same folder as that source or header file should include the relative path from the relevant second-level sub-directory, e.g. the defs.hpp header file is in the "ReaK/core/base" folder and should thus be included as "base/defs.hpp".

13. Know the standard C++ libraries and use them.

The C++ standard libraries are your best bet to find useful generic constructs to implement code, know them well and prefer using them over any kind of hand-rolled code of your own.

14. Restrict external dependencies to well-known and portable libraries.

The ReaK library has, currently, a single external dependency, that is, the Boost libraries which are very well-known, respected, actively maintained and portable to any platform. This is the ideal case for an external dependency used in ReaK. There are very few libraries that have such qualities, and this is why few external dependencies are expected in the future of ReaK. Any potential inclusion of an external library should be discussed extensively.

15. Isolate any external dependencies that are not portable or well-maintained.

The ReaK library has, currently, and has had in the past some external dependencies that are less reputable but offer interesting features and avoid having to re-invent the wheel (e.g. a GUI tool or image processing library or HAL). Such external dependencies must be isolated from the rest of the library. In most cases, these external dependencies can be needed to implement examples and applications of the library, in which case, it is easy to restrict the use of that external library to the application-specific code and programs, this way, the core library and its functionalities are not dependent on that external library. If an external library is used to implement a functionality of or an extension to the ReaK library, then isolate the code that uses the external library via a compilation firewall, that is, restrict its use to source files and do not include the external headers in their header files (a useful idiom to achieve this is called the "PImpl" idiom or "Cheshire Cat" idiom).

16. Use cmake as a build system and nothing else.

The ReaK library uses cmake as a build system. This build system is very flexible and powerful as well as supported on many if not all IDE or editors out there. It is also entirely cross-platform and cross-compiler. Do not try to make your own build configuration with some other system or any clever scripts to generate one. If you find that cmake is missing a feature you really would like to have, then contact the cmake developers and see with them what can be done. It is important that all developers of a library use the same build system, to make sure that what builds correctly for one, builds correctly for another (and that there is no discrepancies in the build configurations).

17. Watch out for compiler extensions.

~~`uint a;`~~, ~~`M_PI`~~

There are many things that can easily look like standard functions, constants or types but in reality are extensions provided by specific compilers. Most compilers do not strictly obey the C++ ISO standard and often provide a number of "extensions" which are often in the form of additional built-in types (like `uint`), additional functions, additional STL containers (like `hash_map`), additional constants (like `M_PI`), or successful compilation of things that technically are either unspecified by the standard or not permitted by it (so-called permissive extensions). However, most modern compilers (2008 and newer) are able to pretty much guarantee (a few bugs aside) that any strictly standard C++98 or C++03 code will compile successfully (putting linking issues aside). The only reliable document of reference to know if something is standard or not, is the C++ ISO standard document itself. But, to know with pretty high confidence if something is standard or not, simply compile with GCC giving it the command-line options `-std=c++98 -pedantic-errors`.

1.3 Programming rules

This section is on programming rules. Most of these rules must be obeyed very strictly as well. They are mostly a collection of ReaK-specific decisions (and restrictions made) and general good programming guidelines to be observed when programming in ReaK. Additionally, one could pretty much say that rules 32 to 63 in the C++ Coding Standards book (Sutter & Alexandrescu, 2004) are all to be obeyed as well.

18. Use ReaK's preferred channels for debug prints.

~~`std::cout << ''the code reached this point successfully'' << std::endl;`~~, `RK_NOTICE(5,''the code reached this point successfully'');`

In order to make sure the debugging outputs are controlled to output only during the debugging or are outputted differently for a release build, the ReaK library has a number of preferred channels for outputting debugging messages to the terminal. The "base/defs.hpp" header includes a few MACROs to accomplish this. First, for printing information that is relevant to debugging (like printing out values of variables or simply marking that a particular point in the execution is reached), ReaK has the MACRO `RK_NOTICE(X,Y)` which first takes an integer value describing the verbosity level of the debug print, where 0 means it will always print, 1-5 means it will print at the default verbosity level (`-DRK_VERBOSITY=5`), and 6 or more is not printed at the default level but may be activated by increasing the verbosity. The second argument is exactly of the same format as if it was between a `std::cout <<` and a `<< std::endl` (note that a semi-colon should appear after the MACRO call). Then, ReaK has the MACROs `RK_ERROR` and `RK_WARNING` that take a single argument (with same spec as the second argument of `RK_NOTICE(X,Y)`), and these MACROs can be used to report errors or warnings that are relevant to a debugger. All those MACROs will also print the source file-name and line-number from which the print originates.

19. Throw exceptions, don't return error-codes.

~~`return SOME_ERROR_CODE`~~, `throw some_exception;`

The ReaK library uses the exception mechanisms provided by C++. Do not generate error-code mechanism when coding in the ReaK library, always prefer exceptions. However, do not specify exceptions for functions, that is, `void some_function()` ~~`throw(std::bad_alloc, some_exception)`~~. Nevertheless, if a function can be guaranteed not to throw, then provide the no-throw specification (with `throw()` or `noexcept`).

20. Provide an output operator `<<` for std::ostream for value-classes.

This overloaded operator is helpful when testing and debugging code.

1.3.1 Object-Oriented Programming rules

21. All classes should be RAII classes.

The *Resource Acquisition Is Initialization* (RAII) idiom is a fundmental building block for robust C++ coding and management of resources. All classes in ReaK should obey this idiom, period. Follow the link below for a tutorial on the creation of a RAII class in modern C++.
Beginning C++0x: Making a RAII Class

22. Protect class invariants with encapsulation.

Class invariants are the relationships that should always be held between the variables it holds and the constraints on these variables (like ranges of values). The class invariants should be hold from the construction (by any constructor) to the destruction of an object of that class. It is not acceptable to allow an undefined or uninitialized state for an object, but a "zombie-state" can be used if the object is required to exist in an invalid state (a "zombie-state" means that one or more data members have some specific value (like NULL) to indicate that the state is invalid). If data members could be bound by some relationship or constraint, they should not be public, i.e., they should be either private or protected depending on the case. If data members are not bound to any invariants and will never be for sure, then they can be made public (no need to make them private and then provide trivial set-get functions in the public interface, but that can still be done, it makes no real difference).

23. Design the ownership relationships between objects.

Ownership design is another fundmental building block for robust C++ coding and management of resources and memory. Follow the link below for a tutorial on the design of ownership relations in modern C++, using smart-pointers.
Beginning C++0x: Design of Ownership

1.3.2 Generic Programming rules

24. Preserve the original semantics of operators when overloading.

Generic programming often involves overloading operators (to be polymorphic with built-in types). However, this feature should not be abused, always make sure that the operator overloads have the same semantics as the built-in operators (with the exception of the << and >> operators and a few others that have conventional semantics that are different from built-in semantics).

25. Overload function templates, don't specialize them.

Generic programming often involves function templates and very often you want to create special versions of these functions for some specific types. In these cases, use normal function overloading rules of C++ to provide these special implementations, do not make template-specializations of those function templates. This is mainly because the rules that apply in this case are cumbersome and it becomes hard to predict which version will be called. If the overloaded versions of the function templates are ambiguous (they can often be when they have the equal number of non-templated parameters), then use "Sfinae-switches" to disable the inappropriate overloads (Sfinae: "Substitution Failure Is Not An Error").

26. Provide concept-check and traits class templates.

27. Obey the principle of minimum requirement.

1.4 Coding style

This section is about coding style. For experienced programmers, these are mere recommendations that probably won't be a surprise, and the requirement

to follow them is more in the order of obeying the principles of them and respecting the uniformity of the style in the ReaK library. For beginners, these are guidelines to help you and that you should respect pretty strictly.

28. All C++ header files should be have the extension .hpp and all C++ source files should be have the extension .cpp.

~~vect_alg.c~~, ~~vect_alg.h~~, vect_alg.cpp, vect_alg.hpp

This is to create consistency in the file naming.

29. Use the ReaK naming standard.

```
frame_2D<double> anchor_point;
class kte_map
void setTimeStep(double aTime) or void set_time_step(double
aTime)
namespace rtti
revolute_joint.cpp
template <typename ForwardIterator>
class StateVectorConcept
```

Basically, lower-case underscore-separated names are the convention for just about everything in non-templated code, and CamelCase is the convention for template arguments and concept-check class templates. This is in harmony with the C++ STL and Boost libraries. ReaK was developed over many years and conventions have evolved so don't be surprised to see some other conventions for some classes. One such example is the name for the functions, in ReaK, both the camelCase (with leading lower-case) and the lower-case underscore-separated names are common and accepted. To avoid conflicts between data members and parameter values, the convention is to use a leading-lower case camelCase for parameter names, with the addition of an "a" at the front. File names also follow the underscore-separation and lower-case, the names of the files do not have to be exactly the name of the class declared in that file, but it is preferred to do so if the file in question does only contain that one class. Finally, as usual, names with all upper-case letters are reserved for MACROs and #defines, nothing else, and should at least be pre-pended with `RK_` to avoid name-clashes.

30. Everything has to be written in (Canadian) English.

In an international environment English is the preferred language, Canadian English (which allows for both the British and American spelling).

31. Provide clear and complete doxygen comments on interfaces.

```
/** this is a ``long'' doxygen comment */, /// this is a short
doxygen comment, ///< this is a short doxygen comment for a
preceeding element (variables only).
```

The code has to be perfectly understandable for an outside developer. Practically this means full Doxygen documentation of the header file (with tags: "file", "author", "date", and possibly others like "todo" and "test"), of the function declarations (with tags: "param", "return" and "tparam"; and, if applicable: "test", "pre", "post", "throw", and "note"; the "brief" tag is desired but optional), and class declarations (with tags: "author", "tparam", "test" and "todo", whenever applicable).

32. Provide reasonable comments in the source code.

```
// Some comment.
```

Again, the code, including the source code, should be understandable to a programmer unfamiliar with the implementation. So, provide a reasonable amount of comments within the code, e.g., a good guideline is to comment each block (like if, for, while, etc.) with a brief and clear description of what that block does.

33. "Inline" short definitions and but not long ones, keep interfaces readable.

The header files should declare an interface that is readable and understandable, as well as not pessimizing (opposing of optimizing). Short functions with less than a 5 lines or so can be defined inline with the declaration (e.g. within a class declaration or in a header-file with the `inline` keyword). This promotes an understandable interface because a glimpse at a trivial function definition is more quickly understandable than any kind of documentation. This also allows for function inlining (formally speaking) which is an important optimization in C++. However, do not define the functions "inline" (i.e. put them in a source file (.cpp) instead) in the following cases: if it breaks readability of the interface (due to function definitions being too long); if it requires including additional header files in the header file in question; or is a virtual member function (which should not be allowed to be inlined in ReaK).

34. Prefer long meaningful names to accronyms or abreviations or meaningless names.

`angularVelocity, ~~omega~~`

This applies mostly to names that affect the interface of a class or library, i.e. parameter names, data members, global constants, function names, class names, etc. Most programmers in this world use code completion tools and large screens, so, the prehistoric age of very short and cryptic names for variables and types is revolute. The code should read like prose, names should be as complete and meaningful as possible, within reason. Very well known accronyms or symbols are acceptable. Very local variables, like local to a small block of code (like temporary variables in a function), can have meaningless names like "t" or "i", but it is highly recommended that their scope be very limited and that they be declared as close as possible to their first point of use. Finally, for naming functions, use action verbs, like `setTimeStep` or `execute` or the like.

35. Keep the code well spaced out.

```
if( fabs( u - v ) < std::numeric_limits< double >::epsilon() )
```

Code needs to be easy to read, it doesn't have to occupy the minimum number of lines or jam as much instructions in the least amount of space. Try to leave an empty space between operators and operands, between parentheses and their content, between parameters, and so on. Also, in the view that code should be readable like prose, well, prose is usually separated into paragraphs, it's the same thing in programming, i.e., leave empty lines to logically separate groups of instructions (and ideally add a comment at the start of each such group of instructions).

36. File content must be kept within 100 columns.

It improves readability when you do not have to scroll sideways.

37. For code indentation, use 2 hard spaces.

```
| class foo_bar {
|   public:
|     void some_function();
| }
```

Tabs should not be used, as they can give different behavior in different editors. Most editors can be set such that each tab is replaced by 4 or 8 spaces. Everything in ReaK should be indented properly, with 2 spaces for each level of indentation, except for namespace scopes that don't require indentation.

38. Use angle brackets (< and >) for external and standard library headers. Use quotes (") for all ReaK headers.

```
#include <iostream>
#include "kte_map.hpp"
```

It is important to distinguish the two forms of `#include` directive not only for documentation purposes but also for portability. The quote-version tells the compilers to first look in the ReaK library's folders for the given header file, while the bracket-version tells the compiler not to look in the ReaK library's folders but only in the external include paths and the system include directories (with standard and system libraries).

39. Put the notation for pointer (*) and reference (&) right after the type.

```
double* pointerToVariable, double& variable
```

In this way, it is more clear which type it points/references to. This is a pretty basic guideline but it is included here because it is a good habit to have.