Gromacs - Feature #988
Definition of "public API"
08/03/2012 01:03 PM - Teemu Murtola

Status: New
Priority: Normal
Assignee: 
Category: core library
Target version: future

Description
One of the goals stated on the wiki for 5.0 is for libgromacs to behave more like a real library. To this end, we should think about what we want to expose as the "public API" of libgromacs and how fixed are we ready to keep those parts between releases.

There are at least the following aspects to consider:

- Which symbols (classes/functions/types) we want to export from our shared library/libraries? (see #701)
- Which symbols do we document as being part of our public API? (see Doxygen documentation instructions in the Doxygen-generated documentation for the current approach)
- Which symbols are declared in installed headers?
  - Should symbols that are not part of a public API, but are declared in installed headers, be within a separate namespace such as internal or detail?
- Which symbols are used in our executables?
- What is the division between the library and the executables?
- Which symbols are used in our unit tests?

This issue is mostly about discussing the above aspects. The outcome should be a clear definition of how the above-mentioned aspects should relate to each other in our code.

A few points to support the discussion:

- Symbols used in installed binaries must be exported.
- Symbols used in unit tests should be exported to avoid a lot of headaches and build system complications.
- If we have our library internally divided into modules, it may also make a lot of sense to unit test also some of the interfaces between modules, even if they are not public.
- Currently, with the exception of mdrun and gmx view, essentially all the code for the programs is within libgromacs, and the executable is only a shell. If we move more content into the executables, can we support things like Python bindings to the analysis tools?

Related issues:
Related to Gromacs - Task #701: Add symbol visibility macros
Related to Gromacs - Bug #999: add MYLIB_EXPORT for public API
Related to Gromacs - Task #1013: Library division for tools and generic Groma...

History
#1 - 08/05/2012 08:56 PM - Teemu Murtola

One slightly related issue for Doxygen documentation is that we should also consider how can one find the documentation for the public API. Currently the Doxygen setup builds a separate "public API" documentation that only includes a subset of functions and classes. There is also a custom \inpublicapi tag that groups all public API functions and classes under one page. For details, see http://www.gromacs.org/Developer_Zone/Programming_Guide/Doxygen.

The main issue with this setup is that \inpublicapi conflicts with tags like \ingroup module_utility for most other items except classes and structs. So a function can only be specified either as being part of a public API, or as belonging to a module (i.e., a subdirectory under src/gromacs/). I think that in many cases, it may be more useful to be able to see an overview of everything that is part of a module than the whole public API. It could be enough that we have that separate "public API" documentation: if a function appears there and does not have an explicit comment that it does not belong to the public API, then it is in public API.

So for this part, I think there are at least these three alternatives:

- Follow the current guideline, which suggests using \inpublicapi everywhere, and leave the \ingroup away if it is not allowed. It also suggests to document things such that everything that appears in installed headers also appears in the public API documentation. The original rationale for this approach was that if someone is browsing through the installed headers, it could be nice to explain everything that is there. But I now think that it may be better to not include this in the public API documentation; the Doxygen comments are still there in the headers if someone wants...
to read them.

- Prefer \ingroup over \inpublicapi. Still make everything from installed headers appear in the public API docs, with explicit comments for those members that are in installed headers only for implementation purposes.
- Prefer \ingroup over \inpublicapi. Only make those members of installed headers that are actually in the public API to appear in the public API documentation.

For the last two, we may want to remove \inpublicapi completely as it may not be that useful. The main benefit that it adds in these cases (for classes only) is that there is a nice “Public API” tag just under the class name. This may be useful when viewing more complete documentation, where also other functions than just public API is included.

Whatever is decided here for \inpublicapi, a similar \inlibraryapi construct should probably follow suit.

---

#2 - 08/11/2012 07:50 PM - Teemu Murtola

- Description updated

Updated the description with one more point to consider.

#3 - 02/12/2014 08:26 PM - Teemu Murtola

- Tracker changed from Task to Feature
- Project changed from Source code reorganization to Gromacs
- Description updated
- Category set to core library
- Target version changed from 5.0 to future

The discussion did not happen for 5.0, but the points in the description are still valid. Added one more issue to consider into the list.

#4 - 09/07/2014 08:21 PM - Peter Kasson

Not sure if this is the best place to bump the discussion, but...

What features do we want for a public API?
Would it make sense to differentiate into a high-level API and a low-level API?
If so, it might be natural to have different stability standards for these two e.g. the high-level API is more stable.

If we want just a single-level public API, any thoughts on which pieces we should prioritize?
The idea of porting analysis tools to use the API is attractive.
Another possibility that would be useful but maybe involve more low-level functionality would be external programs that interface with Gromacs.

#5 - 09/08/2014 02:14 PM - Mark Abraham

IMO, an external API is what people call when they want to re-use existing code. That could take the form of things like

- something like mdrun that they write that calls our (parallel) neighbour-search, force, update or constraint routines (and likely paying some memory-copy costs if they want human-convenient data structures)
- something like an analysis tool that wants access to things like reading trajectories, making selections (but IMO we should target having a Python-extension API sitting on our C++ analysis-framework API)
- installing some plug-in for mdrun to call back, which might either react to or modify a trajectory in progress (e.g. like NAMD's scripting thing, which people ask for occasionally)
The first point has the most internal benefit, because we can use our own API when testing modules, so one or other implementation can develop from the other. Interest from others is probably divided mostly over the first two points.

For example, the PLUMED implementation of (at least) REST would have been straightforward to implement within mdrun if there was a cleaner separation of responsibilities so that mdrun could maintain n distinct Hamiltonians to apply to the coordinates to suit the algorithm in use. (As it is, they do things like gather the coordinates to send them to another replica where the Hamiltonian exists, re-do domain decomposition, then get the new energy/force.) Conversely, if they’d been able to call API functions to load a .tpr and set up a Hamiltonian, then they’d have been able to organize doing that n times on each replica (or just applying the REST scaling operation to one .tpr loaded), and then just call do_force however they’d like to do it, and then send whatever forces they want into the update stage. Ideally, we’d make available a handle-style implementation so that the client doesn’t have to pay anything for data that they don’t actually extract (e.g. a REST client needs to get potentials, but would be quite happy for the forces that might be used later for an update to live out on the compute resources where the update might happen).

Such Hamiltonian flexibility probably has other applications too, e.g. less ad-hoc implementation of whatever free-energy methods use foreign-lambda energy evaluations now.

So, to answer Peter’s questions I’d say one high-level API that stays highly stable, but I consider the feature sets of mdrun and analysis tools so distinct that I would treat their APIs as separate entities.

And none of this is likely to happen in the near term unless we get people with dedicated support to do it! :-)