GROMACS - Task #1490

Usage of forward declarations vs typedef vs #include

04/30/2014 05:31 AM - Teemu Murtola

Status:	New
Priority:	Normal
Assignee:	Mark Abraham
Category:	core library
Target version:	
Difficulty:	uncategorized

Description

Moved discussion from https://gerrit.gromacs.org/#/c/3384/.

We should decide how we want to use opaque types. Currently, there are quite a few different approaches used, and now and then this gives rise to review comments that would be better discussed through in this general context.

Update Decision was: use option 3, preferring forward declarations in headers if possible, and converting to C++ only when planning to work on the code shortly afterwards.

For C code, there are basically three possibilities to declare an opaque type in a header (using naming conventions that seem to be currently used):

1. struct gmx_something_t; (a plain forward declaration)
2. typedef struct gmx_something *gmx_something_t; (forward declaration of gmx_something, plus a typedef for a pointer)
3. typedef struct gmx_something_t gmx_something_t; (forward declaration of a struct, and a typedef with the same name).

The first can occur any number of times in different header files, but the latter two can only occur once in a compilation unit.

Option 1

Plain use of this option looks like this in any header file that uses the type:

```
struct gmx_something_t;
int gmx_func_using_the_struct(struct gmx_something_t *p, …);
```

Headers do not need to include other headers to get the struct definition, and all headers look the same, but the struct keyword needs to be repeated throughout (in C++, this is no longer necessary, but for C it is).

Option 2
The header that declares the type can be like this (assume it is something.h):

```
typedef struct gmx_something *gmx_something_t;
int gmx_func_using_the_struct(gmx_something_t p, …);
```

Another header that uses the type can be either like this, introducing an #include dependency:

```
#include "something.h"
int gmx_another_func(gmx_something_t p, …);
```

or like this with a forward declaration, not introducing an #include, but using quite a different syntax:

```
struct gmx_something;
int gmx_another_func(struct gmx_something *p, …);
```

Option 3
The header that declares the type can be like this (assume it is something.h):

```
typedef struct gmx_something_t gmx_something_t;
int gmx_func_using_the_struct(gmx_something_t *p, …);
```

Another header that uses the type can be like in option 2:

```
#include "something.h"
int gmx_another_func(gmx_something_t *p, …);
```

or like this (as in option 2, but with syntax that is closer to the original header):

```
struct gmx_something_t;
int gmx_another_func(struct gmx_something_t *p, …);
```

Options 2 and 3 can also be used such that the typedef is in its own header.

Associated revisions

Revision 0d1494f4 - 06/15/2014 08:26 PM - Teemu Murtola

Convert gmx_residuetype_t to a non-pointer

This uses the agreed format for forward declarations in #1490.
This removes the need to include residuetypes.h in any headers, and can serve as a simple demonstration of the concept.

Change-Id: Ia82d4f96a1ea97e97e11e9840563f8beebe268c8

Revision cd16ffbb - 12/05/2014 09:19 PM - Mark Abraham

Clean up non-PME part of ewald module

ewald/ewald and ewald/ewald-util code had several different kinds of stuff in it. Separated code that does Ewald long-range and Ewald-family charge correction from the code specific to the group scheme.

Moved general-purpose routines that are used in a few other places to calculate Ewald splitting parameters to the math module.

Minimized header dependencies.

Removed unused things: FLBS FLBSZ

Removed use of typedef for existing opaque struct ewald_tab, per policy in #1490. Renamed to gmx_ewald_tab_t.

Change-Id: I1394bbd02aa92e6581d011e52c5bee12406a0144

Revision 7011af79 - 04/30/2015 10:43 AM - Mark Abraham

Move GPU implementation to new interface

This prepares for OpenCL implementation by updating the existing preprocessor-based interface for GPU functions (that has real implementations with a GPU build and null implementations without). Some related changes to identifier names, comments and docs.

Renamed
s/ncuda_dev/n_dev/g
s/cuda_dev/dev/g
s/nb_cuda/nb_gpu/g
s/cu_nb/nb/g
s/cu_nbv/gpu_nbv/g
s/cuda_dev_info/gmx_device_info/g
so they were more generic, for when an OpenCL implementation wants to share the same identifiers. Related, some *gpu* had to become *cuda_gpu* because it will only have a CUDA implementation, other *cuda_gpu* had to become just *gpu*, some *cuda* had to become *gpu* or *cuda_gpu*. Some CUDA became GPU.

Several CUDA header files are moved from mdlib/nbnxn_cuda to files of more generic names in mdlib. This is not great either, but reorganizing the whole nbnxn code into a proper module, perhaps with submodules is not within the scope of this change.

Updated naming of some data types to be struct gmx_name_t, per style in Redmine #1490. Used some explicit forward declarations instead of including files to get them. Removed typedefs for opaque pointers.

Moved gpu_timing struct from legacyheaders/types/nbnxn_cuda_types_ext.h to timing/gpu_timing.h, and the remaining content to mdlib/nbnxn_gpu_types.h. So we no longer install this internal-use-only header.

The last part of init_interaction_const() in forcrec.cpp is split off, so that the construction phase can be moved to occur before init_nb_verlet(), so that the "constants" are known before any JIT compilation of GPU kernels takes place. Future work will address the question of handling JIT compilation more flexibly, or in more than one place (e.g. when things become compile-time constants after PME tuning).

Converted some Doxygen style to new guidelines, added basic file-level documentation, updated include guards.

Introduced gpu_set_host_malloc_and_free so the implementation-specific details can be handled in the implementations.

Change-Id: I888722c92daeccc7f32987d9b6cb15544351b68d

History

#1 - 05/01/2014 05:59 AM - Teemu Murtola

- Description updated

Now, with proper examples of the alternatives.

Option 1, and the latter examples from options 2 and 3 allow reducing #include dependencies between headers. The main benefits of this are:

- Reduced recompilation time if a function in something.h is changed, since only those source files that actually use those functions need to be recompiled, opposed to all files that transitively include it because they use a function that takes the type as a parameter.
- Reduced amount of transitive dependencies: source files are forced to #include most of the headers they actually need, instead of automatically getting them through some unrelated header. This makes it easier to understand the dependencies, and makes it easier to maintain the code, when changes in a header #include dependencies are less likely to break completely unrelated code.

The latter point also makes the #include dependencies more representative of the real dependencies in the code. But there is also a related drawback: use of forward declarations can hide dependencies from the #include graph in some cases. But for me, the advantage of the two bullets above would outweigh this. The #include dependency graph is just a convenient tool to analyze the dependencies, and review can (maybe) spot cases where an inappropriate dependency is introduced without actually causing an #include dependency.

#2 - 05/01/2014 06:00 AM - Teemu Murtola

- Description updated

#3 - 05/01/2014 08:28 AM - Roland Schulz

We could convert most files to C++ (before changing the typdedefs) and then go with Option 1. Then it doesn't has the disadvantage. Or we could go with Option 2 or 3 with the typedef in a separate include file (unless the dependencies are no problem). In that case we should think about how those extra include files should be named and whether/how they should be combined.

#4 - 05/01/2014 02:30 PM - Mark Abraham

Thanks for clarifying the distinction with and without a typedef.

Converting most files to C++ would make option 1 most palatable - introducing a bunch of "struct" prefixes only to take them out later is ugly. However, that conversion comes with some other issues to resolve (e.g. "%"GMX_PRId64 now provokes some warning unless you insert a space, min/max need std:: and #include <algorithm>, and there's probably others that I haven't seen yet). I'm open to that option, particularly if it makes managing cleaning up dependencies more palatable. Bondeds, PME, do_md and update code were already high on my priority list for post-5.0 cleanup to move to C++ so I can start throwing code around for task parallelism.

Doing the C++ conversion would solidify the general understanding that code in gromacs/gmxlib|mdlib|gmxana that just happens to be in a file that has a .cpp extension is still legacy code. (For that matter, many things that have a "module" (e.g. fileio, swap) are pretty much legacy, whether or not they have a .cpp extension. Sigh.)

I'm happy to take the possible hit of some undesirable non-#include dependencies if it makes it more reasonable to get #include dependencies straightened out. Perhaps some tool exists / could exist that could also graph the forward-declaration dependencies so that information can be used / combined. After all, the forward declarations are mostly one-liners, so grep does a lot of the job.

I'm not keen to introduce a large wad of *_fwd.h files and to teach people to use them if we'll then end up in C++-style option 1 territory later on.

Given a C++ conversion, there's not much to choose between options 1-3; we just choose not to rename things gratuitously, and use forward declarations rather than #includes wherever possible.

#5 - 05/01/2014 07:30 PM - Teemu Murtola

In addition to what Mark mentioned, the C++ conversion also needs to fix a lot of cppcheck and clang static analysis issues, as well as compiler warnings, since all of these are stricter for C++ code than for C code. So that is a significant amount of effort. Needs to be done at some point, but doing it gradually as someone actually plans to work on a part of the code is likely more productive and less error-prone.

Like Mark, I don't like having a large number of .h files that only declare one or a few typedefs. My personal preference is for option 3, with forward declarations instead of the typedef wherever necessary:

- This looks exactly like option 1, except that the extra struct keywords only appear in headers that use the forward declaration, which is significantly fewer places than in option 1.
- All types have an _t prefix, unlike in option 2.
- The only difference between a function that uses the typedef and a function that uses the forward declaration is one additional struct keyword, unlike in option 2 where both the type name and the pointer star are different. I think this improves readability.
- Conversion to C++/option 1 is trivial: once all users of the code are C++, the typedef and the extra struct keywords can be mechanically removed.

Some additional limitations on how the different options can look in source files is placed by Doxygen, which unfortunately is not very intelligent in associating functions from header and source files as the same entity if the parameter types are not "stringwise" identical. I don't remember if I've tested all these cases, but it may mean that the source file that defines the function needs to use the same form as the header that declares the function. But I've used option 3 with the forward declarations in the selection code for a long time before it became C++ code (you can still see remnants in the inner parts), so that should work.

Some types of forward declarations could indeed be parsed automatically with some heuristics, but that can be quite a lot of work to get working in the general case. Using Doxygen or libclang parsing could be feasible. But I don't think this is a priority, as in a majority of cases, the source file will still include the header, even if the header is content with a forward declaration. So most module-level dependencies will still be represented by #include dependencies directly, which are "easy" to parse.

#6 - 05/02/2014 01:10 AM - Mark Abraham

OK, unless/until there's further insight, let's run with option 3, preferring forward declarations in headers if possible, and converting to C++ only when planning to work on the code shortly afterwards.

#7 - 06/03/2014 01:07 PM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #1490.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: I2b5748b2b96c599531c04109e4f657d2e4c6281e
Gerrit URL: https://gerrit.gromacs.org/3539

#8 - 06/03/2014 01:07 PM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #1490.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: Icc3e55e6d8425788136c31270fcdb7f1e43c0963
Gerrit URL: https://gerrit.gromacs.org/3540

#9 - 06/05/2014 08:46 PM - Gerrit Code Review Bot

Gerrit received a related patchset '1' for Issue #1490.
Uploader: Teemu Murtola (teemu.murtola@gmail.com)
Change-Id: Ia82d4f96a1ea97e97e11e9840563f8beebe268c8
Gerrit URL: https://gerrit.gromacs.org/3552

#10 - 12/13/2014 09:43 PM - Gerrit Code Review Bot

Gerrit received a related DRAFT patchset '1' for Issue #1490.
Uploader: Mark Abraham (mark.j.abraham@gmail.com)
Change-Id: I888722c92daeccc7f32987d9b6cb15544351b68d
Gerrit URL: https://gerrit.gromacs.org/4306

#11 - 01/21/2015 10:23 AM - Mark Abraham

- Description updated