Coding Style

by G.A.
(For internal Use Only. Do not distribute)

October 7, 2014

1 Organizational

1.1 Codes and Automation

Codes to which these rules apply are the following.
`DMRG++, PsimagLite, Lanczos++, FreeFermions, GpusDoneRight, SPFv7, and MPS++.`

The script `indentStrict.pl` can be used to automate the application of these rules, as shown in Listing 1, where the option `-fix` will fix most problems in place.

Listing 1: Use of the indentStrict.pl script.

```
PsimagLite/scripts/indentStrict.pl -file file.h [-fix]
```

1.2 Indentation

Indent using "hard" tabs and Kernighan and Ritchie indentation. This is the indentation style used by the Linux Kernel. Note that the tab character is octal 011, decimal 9, hex. 09, or "\t" (horizontal tab). You can represent tabs with any number of spaces you want. Regardless, a tab is saved to disk as a single character: hex. 09, and *not* as 8 spaces.

When a statement or other line of code is too long and follows on a separate line, do not add an indentation level with tab(s), but use spaces for aligment. For an example see Listing 2, where tabs (indentation levels) are indicated by arrows.

Listing 2: Proper use of continuation lines.

```
class␣A␣{
————→void␣thisIsAVeryVeryLongFunction(const␣VeryVeryLongType&␣x,
————→␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣const␣VeryVeryLongType&␣y)
————→{
————→————→//␣code␣here
————→}
```

I repeat the same thing more formally now: Indent following the Linux coding style, with exceptions due to the Linux style applying to C, but here we're talking about code written in C++.

1. Classes are indented like this:

```
class A {
};
```

That is, they are indented as for loops and not as functions. This is because classes can be nested, so this is the most accurate way of following K&R.

2. Constructors and destructors are indented as functions. This is kind of obvious but I mention it because `indent` gets confused if you try using, let's say indent -kr -i8 -ln 200 myFile.h with constructors and destructors.

3. Private:, Protected: and Public: labels do not create a new indentation level. (In C these would be just labels for goto statements, of course, and this is the way that `indent` interprets them.)

4. `namespace` must not add an indentation label.

The Linux Kernel indentation style is formally specified in many places, see, for example, a current version of 'man indent' and search for Linux coding style. Detailed description and `indent` flags are given below.

-nbad Do not force blank lines after declarations.

-bap Forces a blank line after every procedure body.

-nb c Do not force newlines after commas in declarations.

-bbo Prefer to break long lines before boolean operators.

-hnl Prefer to break long lines at the position of newlines in the input.

-br Put braces on line with `if`, etc.

-brs Put braces on struct declaration line.

-c33 Put comments to the right of code in column 33.

-cd33 Put comments to the right of the declarations in column 33.

-ncdb Do not put comment delimiters on blank lines.

-ce Cuddle else and preceding }.

-ci4 Continuation indent of 4 spaces.

-cli0 Case label indent of 0 spaces (do not indent case labels).

-d0 Set indentation of comments not to the right of code to 0 spaces. (i.e., do not unindent comments)

-di1 Put variables in column 1. (space between type and variable)

-nfc1 Do not format comments in the first column as normal.

-i8 Set indentation level to 8 spaces. Used in conjunction with -ts8

-ip0 (does not apply to C++)

-l80 Set maximum line length for non-comment lines to 80.

-lp Line up continued lines at parentheses.

-npcs Do not put space after the function in function calls.

-nprs Do not put a space after every '(' and before every ')'.

-npsl Put the type of a procedure on the same line as its name.

-sai Put a space after each `if`.

-saf Put a space after each `for`.

-saw Put a space after each `while`.

-ncs Do not put a space after cast operators.

-nsc Do not put the * character at the left of comments.

-sob Swallow optional blank lines.

-nfca Do not format any comments.

-cp33 Put comments to the right of `#else` and `#endif` statements in column 33

-ss On one-line `for` and `while` statements, force a blank before the semicolon.

-ts8 Set tab size to 8 spaces.

-il1 Set offset for labels to column 1.

1.3 Naming

1. Start file names with Uppercase, except for the driver program that starts lowercase.
2. Name classes the same as the basename of the file that contains them.
3. Name local variables short and start in lowercase, e.g., `i`, `tmp`, etc.
4. Start class data lowercase and end them in underscore, e.g., `mydata_`
5. Write enum data members and static const data members in all capitals.
6. Start function names in lowercase.
7. All names, whether they start lower- or uppercase, may use smallTalk notation, but avoid for local variables if possible. Never use under_score_notation, i.e., use thisIsAnExample or ThisIsAnExample (as determined above) but never this_is_an_example.

1.4 Compile cleanly at high warning levels

Use `-Werror -Wall` and make sure there are no warnings.

There is one exception and that is MPI, which being such a bad software won't compile cleanly with these switches. To work around this issue your code must not call raw MPI calls, but have a class that wraps them. The rest of the code should be shared among serial and MPI instances. Then turn off MPI and compile your serial code at high warning levels. In this way, the vast majority of your code complies with this rule.

1.5 Use *git* for software management

This is self explanatory, but one thing needs to be stressed: do not use *git* in a centralized way. In other words, have only one committer per repository. Push and pull among repositories to share and synchronize code, but do not share a single repository with others.

1.6 Libraries and Compliance

The use of the Boost library is not allowed.

C++ code must comply with the standard `http://www.csci.csusb.edu/dick/c++std/cd2/`

2 Design Style

1. Have only one `.cpp` file per executable. This is the driver program.
2. Have one `.h` file per class with the same name as the class itself.

2.1 The Driver Program

Compile as many cases as possible in the driver program. Use preprocessor "defines" sparingly for *compile-time* options, and only in the driver program. So, you ask, which options do I make compile-time and which run-time? The answer is simple: Make an option run-time unless you can't (i) because of performance reasons, or (ii) because changing that option is unlikely in a given project. Only then should you make the option compile-time.

So, how does the driver program look like? Like Listing 3.

Listing 3: Sample driver program.

```
#include␣"SimpleReader.h"
#include␣"ParametersEngine.h"
#include␣"Engine.h"
...

typedef␣double␣FieldType;
typedef␣Spf::ParametersEngine<FieldType>␣ParametersEngineType;
typedef␣Dmrg::ConcurrencySerial<FieldType>␣ConcurrencyType;
typedef␣Spf::GeometrySquare<FieldType>␣GeometryType;
...

int␣main(int␣argc,char␣*argv[])
{
————→ConcurrencyType␣concurrency(argc,argv);
```

```
————→ParametersModelType␣mp;
————→ParametersEngineType␣engineParams;
————→Dmrg::SimpleReader␣reader(argv[1]);
————→reader.load(engineParams);
————→reader.load(mp);

————→if␣(concurrency.root())␣std::cerr<<license;
————→GeometryType␣geometry(mp.linSize);
————→DynVarsType␣dynVars(geometry.volume());

————→ModelType␣model(engineParams,mp,geometry);
————→EngineType␣engine(engineParams,model,dynVars,concurrency);

————→engine.main();
}
```

2.2 The Classes

Have one `.h` per class with the declaration and definition of your class. Implement your class inline with its definition.

Have all classes in the correct named namespaces. The namespace should be either (i) the namespace of some toolkit that provides certain functionality (e.g., `PsimagLite`) or (ii) the namespace of your application (e.g., `Spf`, `Dmrg`). More on namespaces later.

Use the RAII principle.

See http://en.wikipedia.org/wiki/Resource_Acquisition_Is_Initialization.

2.3 The *export* Problem (Informational)

The setup just mentioned with only one `.cpp` is rooted in the *export* problem, which I briefly address in the following. The problem: Templates need to be in the same compilation unit as the function using them. This is not strictly true, because one could use the *export* keyword to tell the compiler that the template is in a different compilation unit. However, *few* compilers implement *export*. Why? Short answer: Because it's difficult to do.
Long Answer: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1426.pdf

This is better handled in D because D has modules, and better templates, and compiles faster, but I digress. The point is that the best solution we've come up with is to have *only one* compilation unit. That solves the problem the best way we can but it has (at least) two drawbacks.

First, we cannot compile separate `.o` files, and need to recompile the whole thing each time we make a tiny change somewhere. This is a bad thing^{TM}.

Solution: We need to switch to D, but that's for another write-up.

2.4 Types of Files and Their Rules

Four types of files are described in this specification, ClassDeclFiles, StructDeclFiles, InterfaceDeclFiles, DriverFiles, and the ConfigScript.

2.4.1 ClassDeclFiles

ClassDeclFiles must start with a `normal header`, see §2.5.

ClassDecl files must have, after the `normal header`, a named namespace, like this:

```
namespace␣Name␣{
```

Namespaces' names must conform to §FIXME.

Inside the namespace the first declaration is the class declaration, whose name must coincide with the basename of the file in question. The class may or may not be templated. If templated, the template line must open a new indentation level. If not templated, the class declaration proper must open the indentation level. Follow the rules under section 1.2 for how to open indentation levels.

The template line, if present, must be on one or more lines of its own. Follow the rules to write, and particularly, to break template lines in section FIXME. The class declaration proper must be on a line of its own, which must include the opening brace.

ClassDecl may inherit from another single ClassDecl. ClassDecl may inherit from multiple InterfaceDecl. Inheritance must conform to the rules laid out in §FIXME.

The next code block is optional, and consists of all private typedefs for this class.

The next code line is mandatory, containing the word `public:`. Open a new indentation level after it. None of `public:`, `protected:` or `private:` do ever gain an indentation level, but code writen after them always does.

```
namespace␣Name␣{
class␣SomeClass␣{
————→typedef␣SomeType␣SomeOtherType;
public:
————→//␣more␣code␣here
private:
————→//␣more␣code␣here
};␣//␣class␣SomeClass
}␣//␣Name
```

The next code block is optional, and consists of all public typdefs for this class.

The next code block is optional, and consists of all public enums for this class.

The next code block is optional, and consists of all public static integer constants for this class.

The next code block is optional, and consists of all public static non-integer constants for this class.

The next code block is mandatory, and contains either the constructor(s) for the class, or a comment stating that this class's default constructor is OK. In the latter case, the suggested (but not normative) format is the following:

```
//␣No␣explicit␣constructor␣because
//␣this␣class'␣default␣constructor␣is␣OK.
```

The next code block is (i) if the copy-constructor(s) will be declared private, empty; (ii) if the copy-constructor(s) will not be declared private, and there are pointers or references data members, mandatory; (iii) in all other cases, optional. If present, it contains either the copy-constructor(s) for the class, or a comment stating that this class's default copy-constructor(s) is OK. In the latter case, the suggested (but not normative) format is the following:

```
//␣No␣explicit␣copy-constructor(s)␣because
//␣this␣class'␣default␣copy-constructor(s)␣is␣OK.
```

The next code block is mandatory if there is at least one virtual member function in this class, and optional otherwise. If present, it contains the destructor for the class.

The next code block is (i) if the assignment operator will be declared private, empty; (ii) if the assignment operator will not be declared private, and there are pointers or references data members, mandatory; (iii) in all other cases, optional. If present, it contains either the assignment operator for the class, or a comment stating that this class's default assignment operator is OK. In the latter case, the suggested (but not normative) format is the following:

```
//␣No␣explicit␣assignment␣operator␣because
//␣this␣class'␣default␣assignment␣operator␣is␣OK.
```

Following this, come the public member functions. A ClassDecl must contain at least one public member function. Declare and implement all member functions inline and do not use the inline keyword qualifier. This applies whether the function is public or private, const or non-const, except as permitted in section §7.1.

Non-const public member functions must be declared first. If there is a load from file or load from memory type of function(s), list it (them) first.

Const public member functions must be declared next. If a function does not change the class data it must be explicitly qualified as const. All arguments that the function does not change must be explicitly declared const.

The next code block is optional, and, if present, contains the static public member functions of the class.

The next code block is optional, and, if present, contains the friend prototypes for this class.

The next code line is mandatory, and contains the word `private:` in the same indentation level as `public:`.

The next code block is optional, and contains the copy-constructor(s)'s declaration without implementation, if these are to be declared private.

The next code block is optional, contains the assignment operator declaration without implementation, if these are to be declared private.

List then the private functions of the class, if any. If a function does not change the class data it must be explicitly qualified as const. All arguments that the function does not change must be explicitly declared const. Make private all functions that are not used outside of your class.

The next code block is optional. If present, it contains all private member variables for this ClassDecl.

Close the class declaration by decreasing the indentation level appropriately, and with closing brace on a line of its own:

```
};␣//class␣ClassNameHere
```

The next code block is optional, and contain the companion functions for this ClassDecl. These are non-member non-friend functions that benefit from Koenig lookup, or friend functions. No member function can be listed here, except as permitted in section §7.1.

The file closes with the format

```
}␣//␣namespace␣Dmrg
/*@}*/
#endif
```

Make sure an empty line is included after the `#endif`.

2.4.2 StructDeclFiles

StructDeclFiles must start with a `normal header`, see §2.5.

StructDeclFiles files must have, after the `normal header`, a named namespace, like this:

```
namespace␣Name␣{
```

Namespaces' names must conform to §FIXME.

Inside the namespace the first declaration is the struct declaration, whose name must coincide with the basename of the file in question. The struct may or may not be templated. If templated, the template line must open a new indentation level. If not templated, the struct declaration proper must open the indentation level. Follow the rules under section 1.2, for how to open indentation levels.

The template line, if present, must be on one or more lines of its own. Follow the rules to write, and particularly, to break template lines in section FIXME. The struct declaration proper must be on a line of its own, which must include the opening brace.

StructDecl may not inherit functions.

The next code block is optional, and consists of all public typdefs for this struct.

The next code block is optional, and consists of all public enums for this struct.

The next code block is optional, and consists of all public static integer constants for this struct.

The next code block is optional, and consists of all public static non-integer constants for this struct.

The next code block is optional, and contains the constructor(s) for the struct.

The next code block is optional. If present, it contains the destructor for the struct.

The next code block is mandatory and non-empty. It contains all private member variables for this StructDecl.

Close the struct declaration by decreasing the indentation level appropriately, and with closing brace on a line of its own:

```
};␣//␣struct␣StructNameHere
```

The next code block is optional, and contains the companion functions for this StructDecl. These are non-member non-friend functions that benefit from Koenig lookup. Member functions are not allowed in a StructDecl.

The file closes with the format

```
}␣//␣namespace␣Name
/*@}*/
#endif
```

Make sure an empty line is included after the `#endif`.

2.4.3 InterfaceDecl Files

InterfaceDecl files must start with a `normal header` conforming to §2.5, and then start a named namespace, like this:

```
namespace␣Name␣{
```

Namespaces' names must conform to §FIXME.

Inside the namespace the first declaration is the class declaration, whose name must coincide with the basename of the file in question. The class may or may not be templated. If templated, the template line must open a new indentation level. If not templated, the class declaration proper must open the indentation level. Follow the rules under section 1.2, for how to open indentation levels.

The template line, if present, must be on one or more lines of its own. Follow the rules to write, and particularly, to break template lines in section FIXME. The class declaration proper must be on a line of its own, which must include the opening brace.

InterfaceDecl may inherit, but only from other InterfaceDecl. Multiple inheritance is permitted. Inheritance must conform to the rules laid out in §FIXME.

The next code block is optional, and consists of all private typedefs for this class.

The next code line is mandatory, containing the word `public:`. Open a new indentation level after it. None of `public:`, `protected:` or `private:` do ever gain an indentation level, but code writen after them always does.

The next code block is optional, and consists of all public typdefs for this class.

The next code block is optional, and consists of all public enums for this class.

The next code block is optional, and consists of all public static integer constants for this class.

The next code block is optional, and consists of all public static non-integer constants for this class.

The next code block is optional, and contains the public constructor(s).

The next code block is optional, and, if present, contains the public member functions. No public member function can be virtual. These functions may provide a default implementation for the interface.

Declare and implement all member functions inline and do not use the inline keyword qualifier. This applies whether the function is public or private, const or non-const, except as permitted in section FIXME.

Non-const public member functions must be declared first. If there is a load-from-file or load-from-memory type of function(s), list it (them) first.

Const public member functions must be declared next. If a function does not change the class data it must be explicitly qualified as const. All arguments that the function do not change must be explicitly declared const.

The next code block is optional, and, if present, contains the static public member functions of the class.

The next code block is optional, and, if present, contains the friend prototypes for this class.

The next code line is mandatory, and contains the word `protected:` in the same indentation level as `public:`. None of `public:`, `protected:` or `private:` do ever gain an indentation level, but code writen after them always does.

List then the protected functions of the class, if any. These can be virtual. If a function does not change the class data it must be explicitly qualified as const. All arguments that the function does not change must be explicitly declared const.

The next code line is optional, and contains the the word `private:` in the same indentation level as `public:`.

List then the private functions of this interface, if any. These may be part of the default implementation of the interface. If a function does not change the class data it must be explicitly qualified as const. All arguments that the function does not change must be explicitly declared const. Make private all functions that are not used outside of your InterfaceDecl.

The next code block is optional. If present, it contains all private member variables for this InterfaceDecl.

Close the class declaration by decreasing the indentation level appropriately, and with closing brace on a line of its own:

```
};␣␣␣␣␣//class␣InterfaceNameHere
```

The next code block is optional, and contain the companion functions for this ClassDecl. These are non-member non-friend functions that benefit from Koenig lookup, or friend functions. No member function can be listed here, except as permitted in section FIXME.

The file closes with the format

```
}␣//␣namespace␣Dmrg
/*@}*/
#endif
```

Make sure an empty line is included after the `#endif`.

2.5 Normal Header

The first line of the normal header is a file type line. This line is optional, and, if included, it can only contain the text:

```
//C++
```

Following it, is an optional license block. If present, the block must conform to the following form:

```
/*
any␣number␣of␣lines␣containing␣license␣information
*/
```

Following these optional parts, there is the first mandatory part, the `top level documentation`. This block is mandatory, and has the form:

```
/**␣\ingroup␣NameOfTheGroup␣*/
/*@{*/

/*!␣\file␣Filename.h
␣*
␣*␣␣At␣least␣one␣linee
␣*␣describing␣what␣this␣file␣does␣(but␣not␣how␣it␣does␣it)
␣*
␣*/
```

Following are the mandatory include guards.

```
#ifndef␣LABEL
#define␣LABEL
```

The rules for LABEL is as follows. FIXME.

This ends the `normal header`.

3 Coding Style

3.1 Prefer compile- and link- time errors to run-time errors

3.2 Use const

Use const for all function arguments that the function in question does not change.
Use const when you build a reference to an object that need not change.
Make const all member functions that don't change the class data.

3.3 Avoid MACROS

Do not use preprocessor macros, except as permitted in 2.1.

3.4 Declare variables as locally as possible

See Listing 4 for an example.

Listing 4: Declare variables as locally as possible.

```
// Prefer this...
for (size_t i=0;i<n;i++) {
————→FieldType x = function(i);
————→sum += x * sqrt(3.5);
}
/*... to this:
size_t i;
FieldType x;
for (i=0;i<n;i++) {
————→x = function(i);
————→sum += x * sqrt(3.5);
}
*/
```

3.5 Always initialize variables

3.6 Avoid long functions and avoid deep nesting

3.7 Make header files self-sufficient

If you use `PsimagLite::Vector` in a header file, make sure `Vector.h` is included in that same file.

3.8 Always write internal #include guards

3.9 A Class File

Follow the example provided in Listing 5.

Listing 5: Sample class file. Note that all the implementation is inline.

```
/* Optional license block here */
```

```
/**␣\ingroup␣SPF␣*/
/*@{*/

/*!␣\file␣Engine.h
␣*
␣*␣Short␣description␣of␣what␣this␣file␣is␣for
␣*␣or␣what␣it␣does␣(but␣not␣how!)
␣*/
#ifndef␣SPF_ENGINE_H
#define␣SPF_ENGINE_H
#include␣"Utils.h"
#include␣"ProgressIndicator.h"

namespace␣Spf␣{

template<typename␣ParametersType,typename␣ModelType>
class␣Engine␣{

————→typedef␣typename␣ModelType::DynVarsType␣DynVarsType;

public:

————→Engine(ParametersType&␣params,ModelType&␣model)
————→:␣params_(params),model_(model)
————→{
————→}

————→void␣main()
————→{
————→————→thermalize();
————→————→//␣announce␣thermalization␣done
————→————→measure();
————→————→//␣announce␣measurements␣done
————→————→finalize();
————→}

private:

————→void␣thermalize()
————→{
————→————→size_t␣acc␣=␣0;
————→————→for␣(size_t␣iter=0;iter<iterTherm;iter++)␣{
————→————→————→acc␣+=␣model_.monteCarlo(dynVars_);
————→————→}
————→}

————→ConcurrencyType&␣concurrency_;
————→std::ofstream␣fout_;
————→ProgressIndicatorType␣progress_;
};␣//␣Engine
}␣//␣namespace␣Spf

/*@}*/
#endif␣//␣SPF_ENGINE_H
```

3.10 The Inline Disease

First, a must-read:

> *There appears to be a common misperception that gcc has a magic "make me faster" speedup option called "inline". While the use of inlines can be appropriate (for example as a means of replacing macros, see Chapter 12), it very often is not. Abundant use of the inline keyword leads to a much bigger kernel, which in turn slows the system as a whole down, due to a bigger icache footprint for the CPU and simply because there is less memory available for the pagecache. Just think about it; a pagecache miss causes a disk seek, which easily takes 5 miliseconds. There are a LOT of cpu cycles that can go into these 5 miliseconds.*
>
> *A reasonable rule of thumb is to not put inline at functions that have more than 3 lines of code in them. An exception to this rule are the cases where a parameter is known to be a compiletime constant, and as a result of this constantness you know the compiler will be able to optimize most of your function away at compile time. For a good example of this later case, see the kmalloc() inline function.*
>
> *Often people argue that adding inline to functions that are static and used only once is always a win since there is no space tradeoff. While this is technically correct, gcc is capable of inlining these automatically without help, and the maintenance issue of removing the inline when a second user appears outweighs the potential value of the hint that tells gcc to do something it would have done anyway.*

from "Chapter 15: The inline disease", *Linux kernel coding style*.

Now, for our style, where we write the implementation of member functions inline with the class declaration, gcc already treats those functions as if they had the `inline` keyword. Please don't write this:

Listing 6: Avoid redundant inlines

```
class␣A␣{
————→inline␣int␣someFunction()␣//␣inline␣is␣redundant␣here
————→{
————→————→//␣impl.␣here
————→}
};
```

4 Inheritance and Class Relations

4.1 Use public Inheritance only to derive from a Interface

4.2 Inheritance

Use public Inheritance only to derive from a Interface, where here interface means an abstract class, with or without implementation, virtual or non-virtual.

Do not inherit data from a public class. Remember that all data in all classes is private, unless the class is a `struct`, that is, a class without functions. (For `structs` (a class without functions) data can be inherited if needed.)

Prefer containment to inheritance when getting functionality from one class to another.

Provide a clear public interface for your classes, either implicitly or explicitly via a parent abstract class. Prefer providing abstract interfaces, instead of an implicit interface in the same place as the implementation.

A ClassDecl can inherit from at most one ClassDecl. A ClassDecl can inherit from multiple InterfaceDecl. A StructDecl may not inherit functions.

4.3 Make data members private

I can't emphasize this enough, make *all* data members private. There is only one exception and that is when the class contains *only* data, and no functions, in which case declare it `struct`, not `class`.

Also: make private all functions that are not used outside of your class.

4.4 Don't give away your internals

Example in Listing 7.

Listing 7: Don't give away your internals.

```
class␣A␣{
public:
————→...
————→/*␣Avoid␣this:␣*/
————→PsimagLite::Vector<FieldType>::Type&␣data()␣{␣return␣data_;␣}
————→/*␣Prefer␣this:␣*/
————→FieldType␣data(size_t␣i)␣const␣{␣return␣data_[i];␣}

private:
————→PsimagLite::Vector<FieldType>::Type␣data_;
};
```

4.5 Prefer the weakest coupling between function and class

Do not make members those functions that should be friends instead. Do not make friends those functions that should be companions instead. Place companion functions in the same `.h` file and namespace as the class itself, following the class declaration.

Prefer companions (no this pointer, no scope, no private access) to friends (no this pointer, no scope) to statics (no this pointer) to members, in this order. See Table 1.

Function	`this` pointer	Scope	Private Access	Koenig lookup
Companion	No	No	No	Yes
Friend	No	No	Yes	Yes
Static Member	No	Yes	Yes	Yes
Member	Yes	Yes	Yes	Yes

Table 1: A function's relation to a class. For Koenig lookup see http://en.wikipedia.org/wiki/Argument_dependent_name_lookup

5 Construction, Destruction, and Copying

5.1 Define and initialize member variables in the same order

The `-Werror -Wall` should catch this. See Listing 8.

Listing 8: Define and initialize member variables in the same order.

```
class␣Engine␣{
————→//␣Pay␣attention␣to␣the␣order␣of
————→//␣definition␣vs.␣initialization
public:

————→Engine(ParametersType&␣params,ModelType&␣model)
————→:␣params_(params),␣//␣first
————→␣␣model_(model)␣␣//␣second
————→{
————→}
private:
————→const␣ParamsType&␣params_;␣//␣first
————→const␣ModelType&␣model_;␣␣␣//␣second
————→...
```

5.2 Prefer initialization to assignment in constructors

See Listing 9.

Listing 9: Prefer initialization to assignment in constructors.

```
class␣Engine␣{
public:
————→//␣prefer␣this...:
————→Engine(ParametersType&␣params,ModelType&␣model)
————→:␣params_(params),model_(model)
————→{
————→}
————→/*␣...to␣this:
————→Engine(ParametersType&␣params,ModelType&␣model)
————→{
————→————→params_=params;
————→————→model_=model;
————→}
————→*/
}
```

5.3 Explicity enable or disable copying

There are 3 cases here:

1. If the default (compiler-provided) copy constructor and assignment operator are correct for your class then don't declare them yourself. In this case prefer to comment that the default behavior is correct.

2. If copying doesn't make sense for your type, disable both copy construction and copy assignment by declaring them as private unimplemented functions (see Listing 10).

3. If copy and copy assignment are warranted for your type, but correct copying behavior differs from what the default (compiler-generated) versions will do, then write the functions yourself. See for example, http://www.parashift.com/c++-faq-lite/assignment-operators.html. Note that there are all sort of nuances related to exception handling.

Listing 10: How to disable copy construction and assignment.

```
class␣T␣{␣//...
private:␣␣/␣make␣T␣non-copyable
————→T(const␣T&);␣//␣not␣implemented
————→T&␣operator=(const␣T&);␣//␣not␣implemented
};
```

6 Namespaces and Modules

6.1 Don't write namespace usings in a header file or before an #include

For example, use `std::cout` instead of just `cout`, and `PsimagLite::Matrix` instead of just `Matrix`.

6.2 Don't define entities with linkage in a header file

Seriously. Have classes in a header file. But if you add non-templated nonmember functions write an `inline` there. The implementation has to be there since we don't have multiple `.cpp` files. See Listing 11

Listing 11: Kill linkage in a header file with the inline keyword.

```
//␣This␣is␣sample.h
class␣Sample␣{
————→//␣implementation␣here
};␣//␣Sample

//!␣Mandatory␣inline␣keyword␣here:
inline␣Sample␣operator+(const␣Sample&␣a,␣const␣Sample&␣b)
{
————→//implementation␣here
}
```

7 Templates and Genericity

7.1 Don't specialize function templates

You'll have a hard time if you do, anyway. Use overloads instead.

7.2 Template Constraints

Use `PsimagLite::EnableIf` in combination with `IsVectorLike` or similar (all available in the `PsimagLite` namespace) to enforce template constraints when necessary. For example, let's say there is a function `f` overloaded to take both `int`s and vectors, then code can be written as in Listing 12.

Listing 12: Sample code showing use of template constraints.

```
void␣f(int␣x)
{
————→//␣Do␣something␣here
}

template<typename␣SomeVectorType>
PsimagLite::EnableIf<PsimagLite::IsVectorLike<SomeVectorType>::True,
␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣␣void>::Type␣f(const␣SomeVectorType&␣v)
{
————→//␣Do␣something␣here
}
```

8 Error handling and Exceptions

8.1 Prefer to use exceptions to report errors

Don't return an error status the C way. The caller might ignore it.

8.2 Throw by value, catch by reference

See Listing 13.

Listing 13: Catching and throwing again.

```
catch(MyExceptionType&␣e)␣{
————→e.AppendContext("Passed␣through␣here");
————→throw;␣//␣rethrow␣modified␣object
}
```

9 Type Conventions

9.1 USE_FLOAT

Use the preprocessor define `USE_FLOAT` to specify float (single precission) in the driver. If `USE_FLOAT` is not defined then the code should use double (double precission).

9.2 SizeType

Use the type `SizeType` anywhere an unsigned integer type is needed. Use `SizeType` instead of `size_t`. The code must print `sizeof(SizeType)` at least once for information purposes. `SizeType` should be *typedefed* to be `unsigned int` unless `USE_LONG` has been defined, in which case `SizeType` should be *typedefed* to be `unsigned int long`.

9.3 Vector, Map and Stack

Whenever a vector is needed use `PsimagLite::Vector<T>::Type`. For example, do not use `std::vector<double>`, but use `PsimagLite :: Vector <double>::Type` instead. Note that sometimes a `typename` is needed. `Vector <T>::Type` is *typedefed* to `std::vector` with a custom allocator, which is the standard allocator, unless `USE_CUSTOM_ALLOCATOR` has been defined.

Whenever a map is needed use `PsimagLite::Map<T>::Type`. Whenever a stack is needed use `PsimagLite::Stack<T>::Type`. These are *typedefed* to `std::map` and `std::stack` respectively, with a custom allocator, which is the standard allocator, unless `USE_CUSTOM_ALLOCATOR` has been defined.

9.4 String

Whenever a string is needed use `PsimagLite::String`. This is a `basic_string` with a custom allocator, which is the standard allocator, unless `USE_CUSTOM_ALLOCATOR` has been defined.

10 Bibliography

H. Sutter and A. Alexandrescu, "C++ Coding Standards: 101 Rules, Guidelines, and Best Practices".

