BART Coding Style Guide, Discussion and Suggestions (draft)

Weixiong Zheng

February 21, 2018

Contents

1 Introductions **4**

2 Coding style **5**

2.1 Briefs on what we follow with Google styles 5

2.2 Highlights 5

2.2.1 80-column-ish rule 5

2.2.2 Breaking up a parenthesis 6

2.2.3 Breaking up a general long line 6

2.3 Disagreement and modifications 7

2.3.1 Indent spaces for access specifiers 7

2.3.2 Function naming 7

2.3.3 Constant variable naming 7

2.3.4 Open curly brace position for classes, functions, conditionals 8

2.4 Further restrictions on horizontal spaces 9

2.4.1 Horizontal spaces 9

2.4.2 Use of parenthesis in conditionals 9

2.5 What's not covered 10

3 Doxygen documentation introductions **11**

3.1 General information 11

3.2 Documentation 11

3.2.1 Class method (aka member function) documentation 11

3.2.2 Class documentation 12
3.2.3 Class variable documentation 12
3.2.4 Hyper link usage 12
3.2.5 LaTeX usage 13

4 Unit testings **14**

4.1 Unit testing in BART 14
4.2 CTest based testings 14
4.2.1 Introductions 14
4.2.2 Produce a comparison file 14
4.2.3 dealii::LogStream 15
4.2.4 test_utilities.h 16
4.2.5 Testing with MPI 16

5 Other suggestions **18**

5.1 General suggestions 18
5.1.1 What about really long lines? 18
5.2 Useful things (maybe) for used-to-Python developers 19

Chapter 1

Introductions

After reading style guide from Google, I feel excited as well as upset at the same time. For most part, Google style gives a clear way to present the code to readers (essentially, the developers). Yet following Google style guide without thinking about it would also occasionally lead us to inconvenience or poor readability. Therefore, I would like to combine Google style guide with conventions seen in `deal.II` and `Libmesh` (they might follow some self-consistent style) so future coding would have a consistent style easy to maintain.

Nevertheless, coding style is not the only purpose of this document. It is rather written to provide demos about specifics in developing `BART`, such as how to write doxygen documentation, how to do unit testing, etc.

The rest of the file is arranged as follows: Chapter 2 introduces the coding style guide used in `BART` development; Chapter 3 introduces how doxygen is employed in documenting `BART`; Chapter 4 introduces how unit tests can be developed along with the code development; Chapter 5 gives other suggestions that are not covered by all other chapters.

Chapter 2

Coding style

2.1 Briefs on what we follow with Google styles

Most of the coding style follow Google styles. Some highlights are:

- Local variable naming: lower cases with underscores if needed.
- Class naming: mixed cases such as `class DemoClass`
- Class member variable naming: lower cases linked with and followed by an underscore `double a_demo_var_`
- Order of file inclusion: (corresponding header file→)STL header files→Third-party libraries (`deal.II`, `Boost`, etc.)→Other header files of current project.
- Indent. Two spaces for regular lines. No indent for macros.

2.2 Highlights

2.2.1 80-column-ish rule

Keep 80 columns ish as the maximum per row for and doxygen. You would have flexibility if the last word in the line starts before 80 but ends after 80. In such a case, you are the boss to determine what to do.

For coding, historically, 80 limitation was because of the machinery limit (rich people would afford 132-limitation machines). But it is kept so far because of its visual optimality. I would suggest avoiding coding too long for a line if possible, but 80 sometimes is a bit too tight.

2.2.2 Breaking up a parenthesis

In BART, we will have a lot of chances when we have to break up a parenthesis as there might just be too many parameters

Basically, if the first parameter of a function or conditional can be fitted in the first line, follow the example in https://google.github.io/styleguide/cppguide.html#Function_Declarations_and_Definitions. This actually happens quite often in BART.

So if you can fit at least the first parameter in the first line:

```
double func (double p1, double p2, double p3
             double p4, double p5);
```

That being said, after breaking up a line, the first character of the following line should match the first character following the parenthesis.

If the function name is too long such that even the first parameter cannot be fitted in the first line, Google style guide says leave the open parenthesis in the first line and start parameters in the second line with 4-space indent counting from the first character of the return type.

```
void this_function_has_super_looooooooooooooooong_name (
    double p1,
    double p2);
```

2.2.3 Breaking up a general long line

Just use four extra spaces following the first letter of the previous line.

```
void some_func ()
{
  std::unique_ptr<SomeLongNameClass> =
      std::unique_ptr<SomeLongNameClass> (new SomeLongNameClass
      );
}
```

For braces, it follows the same rule. Just match the first characters of all the following lines.

```
// This example is valid since C++ 11
std::map<std::pair<int, int>, int> test_component_map = {
    ({0, 0}, 0), ({0, 1}, 1), ({0, 2}, 2), ({0, 3}, 3),
    ({1, 0}, 4), ({1, 1}, 5), ({1, 2}, 6), ({1, 3}, 7)};
```

or follow the parenthesis rule if a few item in the braces can be fitted in the first line

```
std::vector<std::vector<int>> demo_vec = {{0, 1, 2, 3, 4, 5, 6,
 7},
                                           {8, 9}};
```

2.3 Disagreement and modifications

2.3.1 Indent spaces for access specifiers

What I agree with Google is two spaces are used for a new line. Yet, Google style gives a suggestion that for access specifier with only one space for indent. This setting is actually anti text editor. Every time specifier with the colon are typed in, text editors (atom, XCode, Sublime Text) with knowing C++ syntax will automatically address the specifier to be with no indent. So what I suggest, which is also used in `deal.II`, `Libmeshand` `MOOSE`. It does not matters to vim and emacs users, but matters to text editor users.

The modification brings

```
class DemoClass
{
private:
  double p1;
};
```

2.3.2 Function naming

Google style about function naming (`https://google.github.io/styleguide/cppguide.html#Function_Names`) shows several examples by naming functions using mixed cases. For simple names, this works okay, but there are several potential cons. The first drawback is that it does not necessarily have readability. If the function name consists of multiple words, Google style is actually not easy to read. This is actually an issue as the name in `BART`is sometimes long and self-explaining. The second drawback is it is spelling prone if no auto-complete is enabled. Switching cases cause a potential issue on spelling if there's no proper auto-completion in IDE/text editor.

`deal.II`and `Libmesh`use lower cases with underscores linking different words in function names and keep it consistent. `BART`originally uses this and I think if consistency is kept, there's no reason to deny this style. For instance, the example function is then named as

```
void initialize_system_matrices_vectors ();
```

2.3.3 Constant variable naming

Google's style is weird that for normal variables, you use combination of lower cases and underscores with extra tailing underscore, which is clear. But for constant, it changes to leading k with mixed-case words. Still, I don't think mixing upper and lower cases is a good idea. So what I suggested and we agreed is:

```
const double k_this_is_a_constant_var;
```

The first rule is used by `MOOSE` and `Libmesh`.

2.3.4 Open curly brace position for classes, functions, conditionals

For open curly brace, Google suggests always not putting it in a new line. That being said, the open curly brace must always stays in the same line as the declaration. But this would look ugly if the parenthesis is long enough which has to to broken into pieces. Besides, the scope is therein not clear.

The modification is also widely used in `Libmesh`and `deal.II`to explicitly separate the scopes.

In summary, we have:

- If content is one line for `if`, `while`, `loop`, do not use braces but put the content in the new line.
- For functions, classes constructors and destructors, always use braces and open curly braces starts in the new line. If there's no content, the close curly brace stays in the same line as the open curly brace otherwise, put them different lines.
- Contents must not appear in the same line with either open or close curly braces.

For conditionals or loops, if the content is one line, do not use braces

```
// example 2 modified
if (looooooong_name_bool1 && looooooong_name_bool2 &&
    looooooong_name_bool3)
{
  demo_func1 ();
  demo_func2 ();
}

// example 3
void DemoClass::super_looooooooooooooong_name_func (
    double p1,
    double p2,
    double p3)
{
  demo_func1 ();
  demo_func1 ();
}

// example 4
class DemoClass ()
{}

// example 5 class constructor
DemoClass::DemoClass ()
{}

// example 6 class destructor
DemoClass::~DemoClass ()
{}

// example 7
```

```
DemoClass::DemoClass ()
{
  initialize_func ();
}
```

2.4 Further restrictions on horizontal spaces

Google style does not give strict restrictions for horizontal spaces. To increase the readability, we restrict ourselves on some rules proposed in this section.

2.4.1 Horizontal spaces

Places where we have horizontal spaces

Logical operators.

```
if (!x && y)
```

Places we don't use horizontal spaces

There are certain places we don't want horizontal spaces.

Adjacent close brackets. Certain old compilers with old C++standard required horizontal spaces. But modern compilers do not.

```
std::vector<std::vector<double>> vec_of_vec;
std::vector<std::shared_ptr<EquationBase<dim>>> equ_ptrs;
```

Adjacent parentheses.

```
int a = ((1+2)*(2+3));
```

2.4.2 Use of parenthesis in conditionals

Proper use of parenthesis would increase the readability. It's especially hard when multiple logical operations are involved. For instance,

```
if (a_expr && b_expr || c_expr && d_expr || e_expr)
{...}
```

would be more readable if it's changed to

```
if ((a_expr && b_expr) || (c_expr && d_expr) || e_expr)
{...}
```

2.5 What's not covered

Rest part that is not yet covered but being used in practice would follow the Google style guide `https://google.github.io/styleguide/cppguide.html`.

Chapter 3

Doxygen documentation introductions

3.1 General information

A default configuration file has been provided "doxygen_config". To produce documentation, run `doxygen doxygen_config`. The documentation is then generated as `doc/html/annotated.html`.

In BART, we use Qt style for doxygen documentation. All the documentation should live in header files. Note that intermediate asterisks are not used.

3.2 Documentation

3.2.1 Class method (aka member function) documentation

Generally it looks like

```
/*!
 One space for indent. Put necessary descriptions for the
 function.

 \param p1 Briefly describe what p1 is or what it is for.
 \return Void. If it's not void, describe the return briefly.

 \note This is optional. Describe something important you think
  developer should be aware of. If one line is not long enough,
  second line should
 match the slash on the first line.
 \todo This is optional. Describe non-trivial things that need
 to be completed
 in the future.
*/
```

```
void some_function(double p1);
```

3.2.2 Class documentation

For class documentation, an extra brief description about what this class provides should be provided. Furthermore, author and date, up to month, of completing the class should be added. If there exists authorship, add your name and date following the existing ones separated by commas. Here's example

```
//! This class provide a demo on doxygen documentation (one
line).
/*!
 This is a demo class. Put more detailed description here.

 \param p1 A demo parameter.

 \note Something to note optionally.
 \todo Something to do optionally.

 \author Weixiong Zheng, second_author_name
 \date 2017/11, some_other_date
*/
class DemoClass (double p1)
{
  ...
};
```

3.2.3 Class variable documentation

Keep the documentation for one-line. There are two options. The first one is for longer one-line:

```
//! This is a long one-line description for the following
variable
double demo_variable_;
```

If the description is short, you can also put the description following the variable declaration as

```
double demo_variable_; //!< This is a short one.
```

If one-line is not enough (for instance, you want to add a webpage for readers for further reading), use a detailed declaration.

3.2.4 Hyper link usage

Sometimes, hyper-link is super useful. In such a case, we create them obeying the following rules:

- Every hyper link should have a short name

- Color for that name should be blue and bold

Here is an example:

```
//! FEValues object.
/*!
 FEValues stands for finite element evaluated at quadrature
 points. For further reading, please go to <a href="https://www
 .dealii.org/8.5.0/doxygen/deal.II/classFEValuesBase.html"
 style="color:blue"><b>FEValuesBase</b></a>.
*/
dealii::FEValues<dim> fe_values;
```

`<b>...</b>` creates an environment setting bold mode for the page name represented by the dots.

3.2.5 LaTeX usage

This part is not really a style guide, but rather a introduction. Sometimes, it's rather nice to have mathematical symbols in sentences or equations in paragraphs. Here is how to use:

- `\f$symbol\f$` for symbols.
- `\f[equation\f]`

Here's an example

```
/*!
 This function calculate \f$k_\mathrm{eff}\f$ relative
 difference from from previous eigenvalue iteration.
 \f[
 \delta k_\mathrm{eff}=\frac{\left|k_\mathrm{eff}-k_\mathrm{eff
 , prev}\right|}{k_\mathrm{eff}}
 \f]

 \return Absolute difference of \f$k_\mathrm{eff}\f$ from
 previous iteration.
*/
double calculate_k_diff ();
```

Chapter 4

Unit testings

4.1 Unit testing in BART

Current BART implementation utilize CTest to establish test suites.

In addition, a GTest-based methodology is under development in parallel.

4.2 CTest based testings

4.2.1 Introductions

CTest based unit testing is to make testing executable generate output files and utilize the comparison file as a reference. Once matched, the testing passes.

4.2.2 Produce a comparison file

Naming convention

Generally speaking, for serial running, the comparison file has to be named as the executable name with ".output" as the appendix. For instance, if the source file is named demo.cc, the comparison file should be named as demo.output. In case many output files exist for the executable, the extension needs to be named in a fashion as "demo.output", "demo.output.2", "demo.output.3" etc. But this is recommended as the last resort with no alternatively viable approach.

If the test is run under some specific conditions, one adds conditions as middle names using "." for separation. In pseudo code, it writes as:

```
demo.[with_<string>(<=|>=|=|<|>)<on|off|version>.]*
[mpirun=<x>.][expect=<y>.][binary.][<debug|release>.]output
```

For instance, a comparison file running with 4 cores using MPI should be named as

```
demo.mpirun=4.output
```

This convention gives the flexibility of testing using the same source with different conditions such as adding a serial (just erase the mpirun=4) test for demo.cc besides its MPI test.

For more details, check `https://www.dealii.org/8.4.1/developers/testsuite.html#layout`.

Accordingly, naming of output file from the source file has to be fixed as "output" s.t. deal.II will be successfully finding the output file for comparison.

4.2.3 dealii::LogStream

There are numerous ways to output the result for unit testing. In case of involving stream, one could choose to use stream in STL or `dealii::LogStream` or a combination of both. Specifically, when using dealii::LogStream, one is essentially using a dealii wrapper of STL stream with a user-defined prefix.

In logstream.h, there's an instance, dealii::deallog, of LogStream ready to be used. Similar to std::fstream, one needs to initialize the deallog as

```
std::ofstream deallogfile;
std::string deallogname = "output";
deallogfile.open (deallogname.c_str());
dealii::deallog.attach (deallogfile, false);
```

One of the perks of using dealii::deallog is s.t. one can prepend any string only once and use it until one popes it out:

```
dealii::deallog.push ("Demo␣prefix");
dealii::deallog << "This␣is␣a␣demo␣with␣string" << "\n"
                << "this␣is␣the␣second␣line" << std::endl;
dealii::deallog.pop();
```

correspondingly, in the output file, you will see

```
DEAL:Demo prefix:: This is a demo with string
this is the second line
```

Note that when `std::endl` is used, current line is ended. Next time deallog is invoked, prefix will be added. For new line without prefix, "\n" has to be used instead of `std::endl` without ending deallog.

Here's a complete demo main function in unit test:

```
int main ()
{
  // Initialize stream
  std::ofstream deallogfile;
  std::string deallogname = "output";
  deallogfile.open (deallogname.c_str());
```

```
  dealii::deallog.attach (deallogfile, false);

  // Prepare the prefix
  dealii::deallog.push ("Demo");

  // Testing section
  int a = 3;
  AssertThrow (a==3, dealii::ExcInternalError());

  // Throw the prefix. For this demo, this step is not
  // necessary. But if there are different tests in one source
  // file and different prefices are preferred in such a case,
  // we need to pop the old prefix out s.t. new prefix can be
  // pushed in.
  dealii::deallog.pop ();
}
```

4.2.4 test_utilities.h

Actually a bunch of functionalities has been provided to simplify the usage of `deallog`. Those functionalities are in the `testing` namespace. One of the functionalities is to simplify the stream initialization by invoking `testing::init_log()`:

```
int main ()
{
  // Initialize stream using init_log()
  /* the old code
  std::ofstream deallogfile;
  std::string deallogname = "output";
  deallogfile.open (deallogname.c_str());
  dealii::deallog.attach (deallogfile, false);
  */

  testing::init_log ();

  /*
  The rest are the same as last demo
  */
}
```

That being said, the stream is implicitly initialized by `testing::init_log()`.

Actually, testing_utilities.h provides another important functionality, parallel logging, to enable us to do unit testing as easy as in serial case. This will be introduced in section subsection.

4.2.5 Testing with MPI

Sometimes, MPI is necessary for testing. The good news is we have provided the deallog initialization in MPI, the `struct testing::MPILogInit` so the whole process will be explained by comments in

the following demo:

```
int main (int argc, char *argv[])
{
  // Step 1: initialize MPI. MPI will be finalized when the
  // program is done.
  dealii::MPI::MPI_InitFinalize mpi_initialization (argc, argv,
   1);

  // Step 2: initialize deallog in parallel. By default deallog
  // will be prefixed with "Process ID#"
  testing::MPILogInit init ();

  // Step 3: testing on current processor. This is exactly the
  // same as serial testing
  int mpi_rank;
  MPI_Comm_rank (MPI_COMM_WORLD, &mpi_rank);
  dealii::deallog << "On␣Processor:␣" << mpi_rank << std::endl;

  // return 0
  return 0;
}
```

If running the executable with 2 processors, the output will be like:

```
DEAL:Process 0::On Processor: 0

DEAL:Process 1::On Processor: 1
```

As seen above, the MPI unit testing has been simplified as easy as serial testing. For details about how we deal with deallog in MPI, please refer to `test_utilities.h`.

Chapter 5

Other suggestions

5.1 General suggestions

- Use caution when using auto. If you know the type, spell it out; if you don't, get to know it and spell it out. At least it would not correctly refer to `active_cell_iterator`.
- Never use `size_t` unless the intension is for retrieving memory usage. For loops, use `int`.
- Maybe it's time to migrate from `std_cxx11` to `std`.
- Use `std::unique_ptr` instead of `std::shared_ptr` unless necessary.
- Every class, member function and member variable should have proper documentation using doxygen in Qt style.

5.1.1 What about really long lines?

Google style says nothing about this as this situation would be rare and maybe people tend to avoid it. `deal.II`never has this problem as it does not have the limit on how many columns to use per row as most people read source code with doxygen and it still has good readability in html. `Libmesh`, on the other hand, does not have very long line code per row. For us, we sometimes has

```
pre_jacobi = std::shared_ptr<PETScWrappers::PreconditionJacobi>
 (new PETScWrappers::PreconditionJacobi)
```

In situations like this, we can create an alias for the type with `typedef`. For instance

```
// Create the alias
typedef PETScWrappers::PreconditionJacobi PJacobi;
...
// when you use it, life is much easier
pre_jacobi = std::shared_ptr<PJacobi> (new PJacobi);
```

In summary, if the type name is longer than 20 characters, alias is adviced to be used.

After using alias, long line would be rare in BART. In those rare cases, such as reporting information on screen using pcout, match the operators

```
pcout << "This␣is␣the␣first␣line␣showing␣breaking␣a␣long␣line"
      << "We␣come␣to␣the␣second␣line" << std::endl;
```

5.2 Useful things (maybe) for used-to-Python developers

C++11 and later simplifies a lot of things from old standards.

Lambda expression. Take the following example sorting a series of pairs of integers. We sort the pairs according to the std::pair<T, T>::second

```
// pairs
std::vector<std::pairs<int, int>> pairs;
...// some assigning-value process
std::sort (pairs.begin(), pairs.end(),
           [](std::pair<int, int>& p1, std::pair<int, int>& p2)
           {return p1.second<p2.second;});
```

Brace initializations.

```
std::vector<int> vec_int = {1, 2, 3};
std::vector<std::vector<int>> id_map = {{1, 2}, {2, 2}};
```

Range based for loop. Beside the index based for loop, range-based (since C++ 11) is also very useful. It works for a lot of data types including maps.

```
for (auto m : mymap)
{
  auto key = m.first;
  auto val = m.second;
}
```

or even more Pythonic if C++ 17 is supported

```
for (auto &[key, val] : mymap)
{...}
```