Coding Style Guide

Code highlighting made with http://www.tohtml.com/cpp/

```
// Code block test
```

Objective

This document aims to setup rules, guidelines and advice concerning not only the formatting of the code, but also coding style. Thanks to these conventions, we would like to reach consistency across the project. It is very important that any programmer be able to look at another's code and quickly understand it. Maintaining a uniform style will also help with versionning. We don't want Git to become the battleground of a formatting war.
However, we are not part of the Coding Monkeys secret sect, so it's all right if some rules are not respected, you won't be banned from eating bananas when failing to apply a rule. Except maybe the 80 characters 2 spaces indentation rule :D

Table of contents

General guidelines

Line length

Each line of text in your code should be at most 80 characters long.

Use 2-space indents, not tabs

Use only spaces, and indent 2 spaces at a time. You should be able to configure your favorite editor for this kind of indentation when hitting the tab key.

UTF-8 encoding

The UTF-8 encoding is preferred wherever possible.

Comments

While comments are very important, the best code is self-documenting. Giving sensible names to types and variables is much better than using obscure names that you must then explain through comments.

File comments

Start each file with license boilerplate, followed by a description of its contents.

Python style

Google Python Style Guide

C++ style

This section is largely inspired from the Google C++ Style Guide. We tried to make it short here. But don't hesitate to have a look at the original document for further details, pros and cons explanation etc.

Header files

Header files should be self-contained (*autonomes*) and end in .h. Specifically, a header should have guards, should include all other headers it needs, and should not require any particular symbols to be defined.

#define Guard

All header files should have `#define guards` to prevent multiple inclusion. The format of the symbol name should be *<PROJECT>_<FILE>_H_*.

```
// Example for the file located in popart/src/geometry/colored_point.h
#ifndef POPART_COLORED_POINT_H_
#define POPART_COLORED_POINT_H_

...

#endif  // POPART_COLORED_POINT_H_
```

You may **forward declare ordinary classes** in order to avoid unnecessary #includes.

Inline functions

Define functions inline only when they are small, say, 10 lines or less.
If a template or inline function is declared in a .h file, define it in that same file. The definitions of these constructs must be included into every .cpp file that uses them, or the program may fail to link in some build configurations. Do not move these definitions to separate -inl.h files.

Function parameter ordering

When defining a function, parameter order is: inputs, input/output, then outputs.

Names and Order of includes

Use standard order for readability and to avoid hidden dependencies:

1. Related header
2. C library
3. C++ library
4. Other libraries' .h
5. Our project's .h.

Scoping

Namespaces

Unnamed namespaces in .cpp files are encouraged. With named namespaces, choose the name based on the project, and possibly its path. Do not use a *using-directive*. Do not use inline namespaces. More information can be found here.

Comments

Comments syntax

We will prefer `//` over `/* */` syntax.

File comments

Generally a .h file will describe the classes that are declared in the file with an overview of what they are for and how they are used. A .cpp file should contain more information about implementation details or discussions of tricky algorithms.
Do not duplicate comments in both the .h and the .cpp. Duplicated comments diverge.

Function comments

Declaration comments describe use of the function. Comments at the definition of a function describe anything tricky about how a function does its job.

Formatting

Pointer and Reference expressions

No spaces around period or arrow. Pointer operators do not have trailing spaces. When declaring a pointer variable or argument, you may place the asterisk adjacent to either the type or to the variable name.
The following are examples of correctly-formatted pointer and reference expressions:

```
x = *p;
p = &x;
x = r.y;
x = r->y;

// These are fine, space preceding.
char *c;
const string &str;

// These are fine, space following.
char* c;
const string& str;
```

Class format

Sections in public, protected and private order.

```
class MyClass : public OtherClass {
 public:      // Note the 1 space indent!
  MyClass();  // Regular 2 space indent.
```

```
  explicit MyClass(int var);
  ~MyClass() {}

  void SomeFunction();
  void SomeFunctionThatDoesNothing() {
  }

  void set_some_var(int var) { some_var_ = var; }
  int some_var() const { return some_var_; }

 private:
  bool SomeInternalFunction();

  int some_var_;
  int some_other_var_;
};
```

Horizontal Whitespace

Never put trailing whitespace at the end of a line.

Vertical Whitespace

Minimize use of vertical whitespace.

In particular, don't put more than one or two blank lines between functions, and be discriminating with your use of blank lines inside functions.

The basic principle is: The more code that fits on one screen, the easier it is to follow and understand the control flow of the program. Of course, readability can suffer from code being too dense as well as too spread out, so use your judgement.

Some rules of thumb to help when blank lines may be useful:

- Blank lines at the beginning or end of a function very rarely help readability.
- Blank lines inside a chain of if-else blocks may well help readability.