Documentation

PCL C++ Programming Style Guide

To make sure that all code in PCL is coherent and easily understood by other developers and users, we follow a set of strict rules that everyone should adopt. These rules are not to be broken unless there is a very good reason to do so. Changes to these rules are always possible, but the person proposing and changing a rule will have the unfortunate task to go and apply the rule change to all the existing code.

Table of Contents

1. Naming

1.1. Files

All files should be **under_scored**.

- Header files have the extension **.h**
- Templated implementation files have the extension **.hpp**
- Source files have the extension **.cpp**

1.2. Directories

All directories and subdirectories should be **under_scored**.

- Header files should go under **include/**
- Templated implementation files should go under **include/impl/**
- Source files should go under **src/**

1.3. Includes

Include statements are made with **"quotes"** only if the file is in the same directory, in any other case the include statement is made with <**chevron_brackets**>, e.g.:

```
#include <pcl/module_name/file_name.h>
#incluce <pcl/module_name/impl/file_name.hpp>
```

1.4. Defines & Macros

Macros should all be **ALL_CAPITALS_AND_UNDERSCORED**. Defines for header type files also need a trailing underscore. Their naming should be mapped from their include name: `pcl/filters/bilateral.h` becomes `PCL_FILTERS_BILATERAL_H_`. The `#ifndef` and `#define` lines should be placed just past the BSD license. The `#endif` goes all the way at the bottom and needs to have the define name in its comment, e.g:

```
// the license

#ifndef PCL_MODULE_NAME_IMPL_FILE_NAME_HPP_
#define PCL_MODULE_NAME_IMPL_FILE_NAME_HPP_

// the code

#endif // PCL_MODULE_NAME_IMPL_FILE_NAME_HPP_
```

1.5. Namespaces

Namespaces should be **under_scored**, e.g.:

```
namespace pcl_io
{
  ...
```

```
}
```

1.6. Classes / Structs

Class names (and other type names) should be **CamelCased**. Exception: if the class name contains a short acronym, the acronym itself should be all capitals. Class and struct names are preferably **nouns**: PFHEstimation instead of EstimatePFH.

Correct examples:

```
class ExampleClass;
class PFHEstimation;
```

1.7. Functions / Methods

Functions and class method names should be **camelCased**, and arguments are **under_scored**. Function and method names are preferably **verbs**, and the name should make clear what it does: checkForErrors() instead of errorCheck(), dumpDataToFile() instead of dataFile().

Correct usage:

```
int
applyExample (int example_arg);
```

1.8. Variables

Variable names should be **under_scored**.

```
int my_variable;
```

1.8.1. Iterators

Iterator variables should indicate what they're iterating over, e.g.:

```
std::list<int> pid_list;
std::list<int>::iterator pid_it;
```

1.8.2. Constants

Constants should be **ALL_CAPITALS**, e.g.:

```
const static int MY_CONSTANT = 1000;
```

1.8.3. Member variables

Variables that are members of a class are **under_scored_**, with a trailing underscore added, e.g.:

```
int example_int_;
```

1.9. Return statements

Return statements should have their values in parentheses, e.g.:

```
int
main ()
{
  return (0);
}
```

2. Indentation and Formatting

The standard indentation for each block in PCL is **2 spaces**. Under no circumstances, tabs or other spacing measures should be used. PCL uses a variant of the GNU style formatting.

2.1. Namespaces

In a header file, the contets of a namespace should be indented, e.g.:

```
namespace pcl
{
  class Foo
  {
    ...
  };
}
```

In an implementation file, the namespace must be added to each individual method or function definition, e.g.:

```
void
pcl::Foo::bar ()
{
  ...
}
```

2.2. Classes

The template parameters of a class should be declared on a different line, e.g.:

```
template <typename T>
class Foo
{
  ...
}
```

2.3. Functions / Methods

The return type of each function declaration must be placed on a different line, e.g.:

```
void
bar ();
```

Same for the implementation/definition, e.g.:

```
void
```

```
bar ()
{
  ...
}
```

or

```
void
Foo::bar ()
{
  ...
}
```

or

```
template <typename T> void
Foo<T>::bar ()
{
  ...
}
```

2.4. Braces

Braces, both open and close, go on their own lines, e.g.:

```
if (a < b)
{
  ...
}
else
{
  ...
}
```

Braces can be omitted if the enclosed block is a single-line statement, e.g.:

```
if (a < b)
  x = 2 * a;
```

2.5. Spacing

We'll say it again: the standard indentation for each block in PCL is **2 spaces**. We also include a space before the bracketed list of arguments to a function/method, e.g.:

```
int
exampleMethod (int example_arg);
```

If multiple namespaces are declared within header files, always use **2 spaces** to indent them, e.g.:

```
namespace foo
{
  namespace bar
  {
     void
```

```
      method (int my_var);
    }
}
```

Class and struct members are indented by **2 spaces**. Access qualifiers (public, private and protected) are put at the indentation level of the class body and members affected by these qualifiers are indented by one more level, i.e. 2 spaces. E.g.:

```
namespace foo
{
  class Bar
  {
    int i;
    public:
      int j;
    protected:
      void
      baz ();
  }
}
```

2.6. Automatic code formatting

The following set of rules can be automatically used by various different IDEs, editors, etc.

2.6.1. Emacs

You can use the following PCL C/C++ style file, download it to some known location and then:

- open .emacs
- add the following before any C/C++ custom hooks

```
(load-file "/location/to/pcl-c-style.el")
(add-hook 'c-mode-common-hook 'pcl-set-c-style)
```

2.6.2. Uncrustify

You can find a semi-finished config for Uncrustify here

2.6.3 Eclipse

You can find a PCL code style file for Eclipse here: *http://svn.pointclouds.org/pcl/trunk/doc/advanced/content/files/* Be warned that this can still contain errors, please mention on the user mailing list if this file needs changes. (feel free to patch)

3. Structuring

3.1. Classes and API

For most classes in PCL, it is preferred that the interface (all public members) does not contain variables and only two types of methods:

- The first method type is the get/set type that allows to manipulate the parameters and input data used by the class.
- The second type of methods is actually performing the class functionality and produces

output, e.g. compute, filter, segment.

3.2. Passing arguments

For get/set type methods the following rules apply:

- If large amounts of data needs to be set (usually the case with input data in PCL) it is preferred to pass a boost shared pointer instead of the actual data.
- Getters always need to pass exactly the same types as their repsective setters and vice versa.
- For getters, if only one argument needs to be passed this will be done via the return keyword. If two or more arguments need to be passed they will all be passed by reference instead.

For the compute, filter, segment, etc. type methods the following rules apply:

- The output arguments are preferably non-pointer type, regardless of data size.
- The output arguments will always be passed by reference.