***roscpp overview (/roscpp/Overview)**: Initialization and Shutdown (/roscpp/Overview/Initialization%20and%20Shutdown) | Basics (/roscpp/Overview/Messages) | Advanced: Traits [ROS C Turtle] (/roscpp/Overview/MessagesTraits) | Advanced: Custom Allocators [ROS C Turtle] (/roscpp/Overview/MessagesCustomAllocators) | Advanced: Serialization and Adapting Types [ROS C Turtle] (/roscpp/Overview/MessagesSerializationAndAdaptingTypes) | Publishers and Subscribers (/roscpp/Overview/Publishers%20and%20Subscribers) | Services (/roscpp/Overview/Services) | Parameter Server (/roscpp/Overview/Parameter%20Server) | Timers (Periodic Callbacks) (/roscpp/Overview/Timers) | NodeHandles (/roscpp/Overview/NodeHandles) | Callbacks and Spinning (/roscpp/Overview/Callbacks%20and%20Spinning) | Logging (/roscpp/Overview/Logging) | Names and Node Information (/roscpp/Overview/Names%20and%20Node%20Information) | Time (/roscpp/Overview/Time) | Exceptions (/roscpp/Overview/Exceptions) | Compilation Options (/roscpp/Overview/Compilation%20Options) | Advanced: Internals (/roscpp/Overview/Internals) | tf/Overview (/tf/Overview) | tf/Tutorials (/tf/Tutorials) | C++ Style Guide*

ROS C++ Style Guide

Contents

This page defines a style guide to be followed in writing C++ code for ROS. This guide applies to all ROS code, both core and non-core.

For Python, see the PyStyleGuide (/PyStyleGuide) and for Javascript, see the ROS JavaScript Style Guide (/JavaScriptStyleGuide)

For general ROS developer guidelines, see the DevelopersGuide (/DevelopersGuide).

1. Motivation

Coding style is important. A clean, consistent style leads to code that is more readable, debuggable, and maintainable. We strive to write elegant code that will not only perform its desired function today, but will also live on, to be re-used and improved by other developers for many years to come.

To this end, we prescribe (and proscribe) a variety of practices. Our goal is to encourage agile but reasoned development of code that can be easily understood by others.

These are guidelines, not rules. With very few exceptions, this document does not completely ban any particular C++ pattern or feature, but rather describes best practice, to be used in the large majority of cases. When deviating from the guidelines given here, just be sure to consider your options carefully, and to document your reasoning, in the code.

Above all else, be **consistent.** Follow this guide whenever possible, but if you are editing a package written by someone else, follow the existing stylistic conventions in that package (unless you are retrofitting the whole package to follow this guide, for which you deserve an award).

Throughout this guide, we will refer to the Google C++ style guide (https://google.github.io/styleguide/cppguide.html), as it is a well-written document on the topic at hand. Here's the first such reference, which gives a longer motivation of the need consistent style:

- See also Google:Background (https://google.github.io/styleguide/cppguide.html#Background)

2. Autoformatting of ROS Code

Why waste your valuable development time formatting code manually when we are trying to build amazing robots? Use a robot to format your code using "clang-format" and these instructions (https://github.com/davetcoleman/roscpp_code_format).

3. What about all this non-conforming code?

A lot of ROS C++ code was written prior to the release of this style guide. Thus, the codebase contains a lot of code that doesn't conform to this guide. The following advice is intended for the developer working with non-conforming code:

- All new package should conform to this guide.
- Unless you have copious free time, don't undertake converting the existing codebase to conform to this guide.
- If you are author of a non-conforming package, try to find time to update the code to conform.
- If you are doing minor edits to a non-conforming package, follow the existing stylistic conventions in that package (if any). Don't mix styles.
- If you are doing major work on a non-conforming package, take the opportunity to re-style it to conform to this guide.

4. Naming

The following shortcuts are used in this section to denote naming schemes:

- **CamelCased**: The name starts with a capital letter, and has a capital letter for each new word, with no underscores.
- **camelCased**: Like CamelCase, but with a lower-case first letter
- **under_scored**: The name uses only lower-case letters, with words separated by underscores. (yes, I realize that *under_scored* should be *underscored*, because it's just one word).
- **ALL_CAPITALS**: All capital letters, with words separated by underscores.

4.1 Packages

ROS packages are **under_scored**.

This is not C++-specific; see the DevelopersGuide (/DevelopersGuide) for more information.

4.2 Topics / Services

ROS topics and service names are **under_scored**.

This is not C++-specific; see the DevelopersGuide (/DevelopersGuide) for more information.

4.3 Files

All files are **under_scored**.

Source files have the extension **.cpp**.

Header files have the extension **.h**.

Be descriptive, e.g., instead of **laser.cpp**, use **hokuyo_topurg_laser.cpp**.

If the file primarily implements a class, name the file after the class. For example the class `ActionServer` would live in the file `action_server.h`.

4.3.1 Libraries

Libraries, being files, are **under_scored**.

Don't insert an underscore immediately after the **lib** prefix in the library name.

E.g.,

```
lib_my_great_thing ## Bad
libmy_great_thing ## Good
```

4.4 Classes / Types

Class names (and other type names) are **CamelCased**

E.g.:

```
class ExampleClass;
```

Exception: if the class name contains a short acronym, the acronym itself should be all capitals, e.g.:

```
class HokuyoURGLaser;
```

Name the class after what it is. If you can't think of what it is, perhaps you have not thought through the design well enough.

Compound names of over three words are a clue that your design may be unnecessarily confusing.

- See also: Google:Type Names (https://google.github.io/styleguide/cppguide.html#Type_Names)

4.5 Function / Methods

In general, function and class method names are **camelCased**, and arguments are **under_scored**, e.g.:

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

Functions and methods usually perform an action, so their name should make clear what they do: checkForErrors() instead of errorCheck(), dumpDataToFile() instead of dataFile(). Classes are often nouns. By making function names verbs and following other naming conventions programs can be read more naturally.

4.6 Variables

In general, variable names are **under_scored**.

Be reasonably descriptive and try not to be cryptic. Longer variable names don't take up more space in memory, I promise.

Integral iterator variables can be very short, such as **i**, **j**, **k**. Be consistent in how you use iterators (e.g., **i** on the outer loop, **j** on the next inner loop).

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

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

Alternatively, an STL iterator can indicate the type of element that it can point at, e.g.:

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

4.6.1 Constants

Constants, wherever they are used, are **ALL_CAPITALS.**

4.6.2 Member variables

Variables that are members of a class (sometimes called fields) are **under_scored**, with a trailing underscore added.

E.g.:

```
int example_int_;
```

4.6.3 Global variables

Global variables should almost never be used (see below for more on this). When they are used, global variables are **under_scored** with a leading **g_** added.

E.g.,:

```
// I tried everything else, but I really need this global variable
int g_shutdown;
```

4.7 Namespaces

Namespace names are **under_scored**.

5. License statements

Every source and header file must contain a license and copyright statement at the beginning of the file.

In the **ros-pkg** and **wg-ros-pkg** repositories, the **LICENSE** directory contains license templates, commented for inclusion in C/C++ code.

See the ROS developer's guide (/DevelopersGuide) for information on permissible licenses and licensing strategy.

6. Formatting

Your editor should handle most formatting tasks. See EditorHelp (/EditorHelp) for example editor configuration files.

Indent each block by 2 spaces. Never insert literal tab characters.

The contents of a namespace are not indented.

Braces, both open and close, go on their own lines (no "cuddled braces"). E.g.:

```
if(a < b)
{
  // do stuff
}
else
{
  // do other stuff
}
```

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

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

Always include the braces if the enclosed block is more complex, e.g.:

```
if(a < b)
{
  for(int i=0; i<10; i++)
    PrintItem(i);
}
```

Here is a larger example:

Toggle line numbers

```
/*
 * A block comment looks like this...
 */
#include <math.h>
class Point
{
public:
  Point(double xc, double yc) :
    x_(xc), y_(yc)
  {
  }
  double distance(const Point& other) const;
  int compareX(const Point& other) const;
  double x_;
  double y_;
};
double Point::distance(const Point& other) const
{
  double dx = x_ - other.x_;
  double dy = y_ - other.y_;
  return sqrt(dx * dx + dy * dy);
}
int Point::compareX(const Point& other) const
{
  if (x_ < other.x_)
  {
    return -1;
  }
  else if (x_ > other.x_)
  {
    return 1;
  }
  else
  {
    return 0;
  }
}
namespace foo
{
int foo(int bar) const
{
  switch (bar)
  {
    case 0:
      ++bar;
      break;
    case 1:
      --bar;
    default:
    {
      bar += bar;
      break;
    }
  }
}
} // end namespace foo

```

6.1 Line length

Maximum line length is 120 characters.

6.2 #ifndef guards

All headers must be protected against multiple inclusion by #ifndef guards, e.g.:

```
#ifndef PACKAGE_PATH_FILE_H
#define PACKAGE_PATH_FILE_H
...
#endif
```

This guard should begin immediately after the license statement, before any code, and should end at the end of the file.

7. Documentation

Code must be documented. Undocumented code, however functional it may be, cannot be maintained.

We use doxygen (http://www.doxygen.org) to auto-document our code. Doxygen parses your code, extracting documentation from specially formatted comment blocks that appear next to functions, variables, classes, etc. Doxygen can also be used to build more narrative, free-form documentation.

See the rosdoc (/rosdoc) page for examples of inserting doxygen-style comments into your code.

All functions, methods, classes, class variables, enumerations, and constants should be documented.

8. Console output

Avoid printf and friends (e.g., cout). Instead, use rosconsole (/rosconsole) for all your outputting needs. It offers macros with both printf- and stream-style arguments. Just like printf, rosconsole output goes to screen. Unlike printf, rosconsole output is:

- color-coded
- controlled by verbosity level and configuration file
- published on **/rosout**, and thus viewable by anyone on the network (only when working with roscpp)
- optionally logged to disk

9. Macros

Avoid preprocessor macros whenever possible. Unlike inline functions and const variables, macros are neither typed nor scoped.

- See also: Google:Preprocessor Macros (https://google.github.io/styleguide/cppguide.html#Preprocessor_Macros)

10. Preprocessor directives (#if vs. #ifdef)

For conditional compilation (except for the #ifndef guard explained above), always use #if, not #ifdef.

Someone might write code like:

```
#ifdef DEBUG
        temporary_debugger_break();
#endif
```

Someone else might compile the code with turned-off debug info like:

```
cc -c lurker.cpp -DDEBUG=0
```

Always use #if, if you have to use the preprocessor. This works fine, and does the right thing, even if DEBUG is not defined at all.

```
#if DEBUG
        temporary_debugger_break();
#endif
```

11. Output arguments

Output arguments to methods / functions (i.e., variables that the function can modify) are passed by pointer, not by reference. E.g.:

```
int exampleMethod(FooThing input, BarThing* output);
```

By comparison, when passing output arguments by reference, the caller (or subsequent reader of the code) can't tell whether the argument can be modified without reading the prototype of the method.

- See also: Reference Arguments (https://google.github.io/styleguide/cppguide.html#Reference_Arguments)

12. Namespaces

Use of namespaces to scope your code is encouraged. Pick a descriptive name, based on the name of the package.

Never use a **using-directive** in header files. Doing so pollutes the namespace of all code that includes the header.

It is acceptable to use **using-directives** in a source file. But it is preferred to use **using-declarations**, which pull in only the names you intend to use.

E.g., instead of this:

```
using namespace std; // Bad, because it imports all names from std::
```

Do this:

```
using std::list;  // I want to refer to std::list as list
using std::vector;  // I want to refer to std::vector as vector
```

- See also: Google:Namespaces (https://google.github.io/styleguide/cppguide.html#Namespaces)

13. Inheritance

Inheritance is the appropriate way to define and implement a common interface. The base class defines the interface, and the subclasses implement it.

Inheritance can also be used to provide common code from a base class to subclasses. This use of inheritance is discouraged. In most cases, the "subclass" could instead contain an instance of the "base class" and achieve the same result with less potential for confusion.

When overriding a virtual method in a subclass, always declare it to be **virtual,** so that the reader knows what's going on.

- See also Google:Inheritance (https://google.github.io/styleguide/cppguide.html#Inheritance)

13.1 Multiple inheritance

Multiple inheritance is strongly discouraged, as it can cause intolerable confusion.

- See also Google:Multiple Inheritance (https://google.github.io/styleguide/cppguide.html#Multiple_Inheritance)

14. Exceptions

Exceptions are the preferred error-reporting mechanism, as opposed to returning integer error codes.

Always document what exceptions can be thrown by your package, on each function / method.

Don't throw exceptions from destructors.

Don't throw exceptions from callbacks that you don't invoke directly.

If you choose in your package to use error codes instead of exceptions, use only error codes. **Be consistent.**

14.1 Writing exception-safe code

When your code can be interrupted by exceptions, you must ensure that resources you hold will be deallocated when stack variables go out of scope. In particular, mutexes must be released, and heap-allocated memory must be freed. Accomplish this safety by using the following mutex guards and smart pointers:

- TODO
- TODO

15. Enumerations

Namespaceify your enums, e.g.:

```
namespace Choices
{
  enum Choice
  {
     Choice1,
     Choice2,
     Choice3
  };
}
typedef Choices::Choice Choice;
```

This prevents enums from polluting the namespace they're inside. Individual items within the enum are referenced by: Choices::Choice1, but the typedef still allows declaration of the Choice enum without the namespace.

16. Globals

Globals, both variables and functions, are discouraged. They pollute the namespace and make code less reusable.

Global variables, in particular, are strongly discouraged. They prevent multiple instantiations of a piece of code and make multi-threaded programming a nightmare.

Most variables and functions should be declared inside classes. The remainder should be declared inside namespaces.

Exception: a file may contain a **main()** function and a handful of small helper functions that are global. But keep in mind that one day those helper function may become useful to someone else.

- See also Google:Static and Global Variables (https://google.github.io/styleguide/cppguide.html#Static_and_Global_Variables)
- See also Google:Nonmember, Static Member, and Global Functions (https://google.github.io/styleguide/cppguide.html#Nonmember,_Static_Member,_and_Global_Functions)

17. Static class variables

Static class variables are discouraged. They prevent multiple instantiations of a piece of code and make multi-threaded programming a nightmare.

18. Calling exit()

Only call **exit()** at a well-defined exit point for the application.

Never call **exit()** in a library.

19. Assertions

Use assertions to check preconditions, data structure integrity, and the return value from a memory allocator. Assertions are better than writing conditional statements that will rarely, if ever, be exercised.

Don't call **assert()** directly. Instead use one of these functions, declared in **ros/assert.h** (part of the rosconsole (/rosconsole) package):

```
/** ROS_ASSERT asserts that the provided expression evaluates to
 * true.  If it is false, program execution will abort, with an informative
 * statement about which assertion failed, in what file.  Use ROS_ASSERT
 * instead of assert() itself.
 */
#define ROS_ASSERT(expr) ...
```

```
/** ROS_BREAK aborts program execution, with an informative
 * statement about which assertion failed, in what file. Use ROS_BREAK
 * instead of calling assert(0) or ROS_ASSERT(0).
 */
#define ROS_BREAK() ...
```

Do not do work inside an assertion; only check logical expressions. Depending on compilation settings, the assertion may not be executed.

20. Testing

See gtest (/gtest).

21. Portability

We're currently support Linux and OS X, with plans to eventually support other OS's, including possibly Windows. To that end, it's important to keep the C++ code portable. Here are a few things to watch for:

- Don't use **uint** as a type. Instead use **unsigned int**.
- Call **isnan()** from within the **std** namespace, i.e.: **std::isnan()**

22. Deprecation

To deprecate an entire header file within a package, you may include an appropriate warning:

```
#warning mypkg/my_header.h has been deprecated
```

To deprecate a function, add the deprecated attribute:

```
ROS_DEPRECATED int myFunc();
```

To deprecate a class, deprecate its constructor and any static functions:

```
class MyClass
{
public:
  ROS_DEPRECATED MyClass();

  ROS_DEPRECATED static int myStaticFunc();
};
```