The OpenCV Coding Style Guide

The document is a short guide on code style, used in OpenCV. The modules (`core` , `imgproc` , etc) are written in C++, so the document concerns only the C++ code style.

General Comments about Writing OpenCV code

- The code should be written C++ 11. When the patch is prepared for OpenCV 2.4.x or 3.4.x, use C++ 98.
- The code should be cross-platform. Try to minimize or eliminate completely the platform-specific details, especially in the header files.
- If your patch for OpenCV is based on some source code from Internet, please, pay attention to and respect the license. In particular, check that it's not GPL or LGPL. Check that it's not "for research purpose only" license. Check that the source code, supplied documentation, README, or the referenced papers with algorithm description do not mention any patents.
- If your patch modifies existing files, please, try to minimize your patch as much as possible. Do not reformat existing code, do not do other stylistic changes. If the intent is to fix a bug or add a new small feature, just do exactly that.

Files

All the functionality must be put into one or more `.cpp` and `.hpp` files into the appropriate module of OpenCV (`opencv/modules` or `opencv_contrib/modules`). A new module should be created if the contributed functionality is does not fit any existing module.

- All the file names are written in lower case for better compatibility with both POSIX and Windows.
- C++ interface headers have `.hpp` extension
- Implementation files have `.cpp` extension
- The implementation is put into `opencv/modules/<module_name>/src` , interface is added to the header files in `opencv/modules/<module_name>/include/opencv2/<module_name>` . In most cases files will be added to compilation process automatically by CMake during configuration stage. Some modules require explicit listing of source files in theirs root CMakeLists.txt file.
- Each module can have samples in `opencv/modules/<module_name>/samples` directory. Some samples can be added to global locations: `opencv/samples/cpp` , `opencv/samples/python` , etc.
- Documentation is written in `.hpp` files and any additional files (bibliographic reference, images) can be put into `opencv/modules/<module_name>/doc` directory.
- Accuracy tests are put to `opencv/modules/<module_name>/test` directory, performance tests - to `opencv/modules/<module_name>/perf` . If test needs data, it can be added to special repository: http://github.com/opencv/opencv_extra. If possible, try to reuse existing test data (e.g. `lena.png`).

File Structure

- Every source file, except for the samples, starts with license header:

```
// This file is part of OpenCV project.
// It is subject to the license terms in the LICENSE file found in the top-
level directory
// of this distribution and at http://opencv.org/license.html.
```

OpenCV code license (Apache 2.0) can be found here. Below you can also put your copyright.

- All the functionality must be put into `cv::` namespace, or nested namespace, e.g. `cv::vslam::`
- Code lines should not be very long. Normally, they should be limited to **100** characters.
- No tabulation should be used. Set your editor to use spaces instead.
- Indentation is 4 spaces.
- Only English text (ASCII) is allowed. Do not put comments or string literals in other languages.
- Header files must use guarding macros, protecting the files from repeated inclusion:

```
#ifndef OPENCV_module_name_header_name_HPP
#define OPENCV_module_name_header_name_HPP
namespace cv { namespace mynamespace {
// ...
}}
#endif
```

- Source files must include `precomp.hpp` header before other headers, in order to make precompiled headers mechanism in Visual C++ work properly.

Naming conventions

- OpenCV uses mixed-case style identifiers for external functions, types and class methods.
- Class names start with a capital letter.
- Methods and functions names start with a small letter, unless they are named after the author of the algorithm, e.g. `cv::Sobel()`, in which case they can start with a capital letter.
- Macros and enumeration constants are written with all capital letters. Words are separated by underscore.
- All public functions and classes must be marked with the `CV_EXPORTS` macro. `CV_EXPORTS_W` macro should be used to expose class or function to Python and Java bindings.

Designing functions and class interfaces

It is important to design function interface in a way, consistent with the rest of the library. The elements of function interface include:

- Functionality
- Name
- Return value
- Type of arguments
- Order of arguments
- Default values for some arguments

Functionality

The functionality must be well defined and non-redundant. The function should be easily embedded into different processing pipelines that use other OpenCV functions.

Name

The name should basically reflect the function purpose. There are a few common naming patterns in OpenCV:

- Majority of function names have form: `<actionName><Object><Modifiers>`, e.g. `calibrateCamera`, `calcOpticalFlowPyrLK`.
- Sometimes the function may be called by the algorithm name it implements or result object name it produces, e.g. `Sobel`, `Canny`, `Rodrigues`, `sqrt`, `goodFeaturesToTrack`.

Return value

It should be chosen to simplify function usage. Generally, a function that creates/computes a value should return it. It is the good practice to do so for the functions returning scalar values. However, in case of image processing function this would lead to frequent allocation/deallocation of large memory blocks. Image processing functions often modify an output image, which is passed as a parameter (by reference) instead of creating and returning result images.

Functions should not use return value for signaling about critical errors, such as null pointers, division by zero, bad argument range, unsupported image format etc. Instead, they should throw an exception, an instance of `cv::Exception` or its derivative class. On the other hand, it is recommended to use a return value to report normal run-time situations that can happen in a correctly working system (e.g. tracked object goes outside of the image).

Types of arguments

Argument types are preferably chosen from the already existing set of OpenCV types: `Mat` for raster images and matrices, `vector<Mat>` for collection of images, `vector<Point>`, `vector<Point2f>`, `vector<Point3f>`, `vector<KeyPoint>` for point sets, contours or collections of key points, `Scalar` for 1- to 4-element numerical tuples (like colors, quaternions etc.) It is not recommended to use plain pointers and counters, because it makes the interface lower-level, meaning more probable typing errors, memory leaks etc. For passing complex objects into functions, methods, please, consider `Ptr<>` smart pointer template class.

A consistent argument order is important because it becomes easier to remember the order and it helps programmer to avoid errors, connecting with wrong argument order. The usual order is: input parameters, output parameters, flags and optional parameters.

Input parameters usually have `const` qualifiers. Large objects are normally passed by a constant reference; primitive types and small structures (`int, double, Point, Rect`) are passed by value.

Optional arguments often simplify function usage. Because C++ allows optional arguments in the end of parameters list only, it also may affect decisions on argument order—the most important flags go first and less important—after.

For the example of function and class declarations, take a look at the core.hpp file.

Using `InputArray`, `OutputArray` and related classes

In 2.4 we introduced new "proxy" datatypes in OpenCV to support multiple array types simultaneously. For example, in some cases it is more convenient to represent a point set as `vector<Point3f>`, in other cases - as a matrix. In some cases it is more convenient to store homography matrix as `Mat`, in other - as `Matx33f`. If a parameter of your function has type `Mat`, `Matx<>`, `vector<Point...>`, `vector<Mat>` or `vector<vector<Point...>>`, please consider using `InputArray`, `OutputArray` and other wrapping types.

- `InputArray` - used for input arrays; you can only read from the arrays and do not modify or reallocate them. For example, parameter of `cv::determinant()` function is `InputArray`.
- `InputOutputArray` - used for both input/output arrays, i.e. arrays which are modified inside the functions. For example, all the drawing functions, such as `cv::line`, `cv::drawContours` etc. accept the image as a `InputOutputArray` type.

- `OutputArray` - used for output arrays. The function can not assume that the array has the proper size and type, or that its content is somehow initialized. Instead, it should call `OutputArray::create()` method to allocate required data buffer if needed. You can also call `create()` on input/output arrays, but normally you do not have to.

These types are called proxy classes because they are not real arrays. They just store pointers to the actual arrays and the "kind" of array (`Mat` , `Matx` , `vector<>` etc.). So when you see some function that takes such a parameter, it means that it can take `Mat` , `Matx` or `vector<>` . Since those Array types are proxy classes, you **should not** declare local variables of those types unless you are an expert in C++ and know exactly what you are doing.

In fact, `InputArray` is the base class for `OutputArray` , and `InputOutputArray` is synonym for `OutputArray` . But you should use the proper names in the function arguments to assist the automatic wrapper generators.

Inside the functions that accept `InputArray/InputOutputArray/OutputArray` objects, one should call `.getMat()` method to get the underlying matrix.

User may want to ignore some input values or omit producing some output arrays. Special value `noArray()` can be passed as function argument in this case.

High-level C++ interface. Algorithms

In some cases it is necessary to represent an algorithm as a class, not a function. For example, an algorithm can have internal state updated with each run, e.g. background subtraction. Or an algorithm can have many parameters. Some algorithms can have include several stages or steps, e.g. training and prediction in machine learning methods.

If you decide to make your algorithm a class, you should follow OpenCV Algorithm concept.

Rationale and principles of the Algorithm-based design

- We want our API to stay stable when the implementation changes.
- We want to preserve source-level compatibility. Binary-level compatibility should be preserved between patch releases.
- We want to keep header files clean to make tracking API changes easier.
- We want to keep our tools that parse OpenCV headers simple and robust (bindings generators).
- We want OpenCV to build fast.

To achieve these goals we create separate interface and implementation classes. Interfaces are classes without constructors, without data members and with only purely virtual methods. Implementation derives from interface and is hidden from library user. Construction of an object is performed by a factory method or a function which should return class instance wrapped into a smart pointer (`cv::Ptr<>`).

Steps to make your class following this style

- *(in public .hpp file)* Inherit your class from `cv::Algorithm` or derivative class, e.g. `cv::StereoMatcher` , declare interface methods and factory methods:

```
namespace cv {
namespace mynamespace {

class CV_EXPORTS MyStereoMatcher : public StereoMatcher
{
public:
```

```
    virtual double getLambda() const = 0;
    virtual void setLambda(double lambda) = 0;

    static Ptr<MyStereoMatcher> create(...);
};

}} // cv::mynamespace::
```

- *(in private .cpp file)* Implement your algorithm:

```
#include "precomp.hpp"

namespace cv {
namespace mynamespace {

class MyStereoMatcherImpl : MyStereoMatcher
{
public:
    MyStereoMatcherImpl(...) { ... }
    virtual ~MyStereoMatcherImpl() { ... }
    double getLambda() const { ... }
    void setLambda(double l) { ... }

    // implement required methods from base StereoMatcher class
    void compute(InputArray _left, InputArray _right, OutputArray _disp) {
... }

private:
    double lambda;
};

// implement factory method
Ptr<MyStereoMatcher> MyStereoMatcher::create(<args>) { return
makePtr<MyStereoMatcherImpl>( ... ); }

}} // cv::mynamespace::
```

Extending/modifying algorithms

- As long as the public interface is not modified, changes are fine.
- If public interface should be changed, but it was not included in any official public release yet, changes are fine.
- If you want to expose a new extended algorithm, it should be done in a way to preserve the source-level compatibility. Create a new interface on top of the existing one, and provide another `create` function to instantiate your algorithm (since OpenCV 4.x it's fine to add new properties to algorithms, because we relaxed compatibility requirements from binary compatibility to source-level compatibility (except for the patch releases)):

```
namespace cv {
namespace mynamespace {

class CV_EXPORTS MyPyrStereoMatcher : public MyStereoMatcher
{
public:
    // more properties ...
    virtual void setNPyramidLevels(int nlevels) = 0;
    virtual double getNPyramidLevels() const = 0;
    // create your algorithm; the implementation is completely hidden, as
usual
    static Ptr<MyPyrStereoMatcher> create( ... );
};

}} // cv::mynamespace::
```

Code Layout

There is a single coding guideline in OpenCV: *each single file must use a consistent formatting style*.

Recommended formatting style is as follows:

```
if (a > 5)
{
    int b = a * a;
    c = c > b ? c : b + 1;
}
else if (abs(a) < 5)
{
    c--;
}
else
{
    printf("a=%d is far too negative\n", a);
}
```

```
namespace cv {
namespace abc {

class TheClass
{
public:
    int getProperty() const;
    void setProperty(int prop);
private:
    int property;
};

}} // cv::abc::
```

Portability, External Dependencies

Code written for OpenCV 4.x (*master* branch) must comply with the C++11 standard. OpenCV 3.x and older versions must comply with the C++98 standard. C++ extensions usage should be avoided and is forbidden in public headers.

One should get rid of compiler-dependent or platform-dependent constructions and system calls, such as:

- Compiler pragma's
- Specific keywords, e.g. `__stdcall`, `__inline`, `__int64`. Use `CV_INLINE` (or simple `inline` in C++ code), `CV_STDCALL` (try to avoid it if possible), `int64`, respectively.
- Compiler extensions, e.g. special macros for min and max, overloaded macros etc.
- Inline assembly
- Unix or Win32-specific calls, e.g. `bcopy`, `readdir`, `CreateFile`, `WaitForSingleObject` etc.
- Concrete data sizes instead of `sizeof`'s (`sizeof(int)` rather than 4), byte order (`*(int*)"\x1\x2\x3\x4"` is 0x01020304 or 0x04030201 or what?), simple `char` instead of `signed char` or `unsigned char` anywhere except for text strings. Use short forms `uchar` for `unsigned char` and `schar` for signed char. Use preprocessor directives for surrounding non-portable pieces of code.

Writing documentation on functions

The documentation for contributed functions is written using inline Doxygen comments. The documentation is built nightly and is uploaded to https://docs.opencv.org.

Use the existing documentation as an example. You are also welcome to provide tutorials for large descriptive chunks of text with pictures, code samples etc.

Implementing tests

- For tests we use GTest framework. Please, check the documentation at the prect site.
- Each test source file should include `test_precomp.hpp` first.
- All the test code is put into `opencv_test` namespace.
- Declare your Google tests as following:

  ```
  TEST(<module_name>_<tested_class_or_function>, <test_type>) { <test_body> }
  ```

  For example:

  ```
  TEST(Imgproc_Watershed, regression) { ... }
  ```

- To access test data, use `cvtest::TS::ptr()->get_data_path()` method. For example, if you put your test file to `opencv_extra/testdata/cv/myfacetracker/clip.avi`, you can use `cvtest::TS::ptr()->get_data_path() + "myfacetracker/cl.avi"` to get full path to the file. To make it work properly, set the environment variable `OPENCV_TEST_DATA_PATH` to `<your_local_copy_of_opencv_extra>/testdata`
- Avoid including C++ standard library headers, like `vector`, `list`, `map`, `limits`, `iostream`, etc.

- Avoid `using namespace std`. Use `std::` if necessary (common types are imported into `opencv_test` namespace).
- Don't use `std::tr1` namespace.
- Don't include OpenCV headers for `core` / `imgproc` / `highgui` modules. These headers are included by `ts.hpp`.

Python samples

Python samples can be executed in two modes:

- standalone
- throught `demo.py`. This helper script imports samples .py files (one file per sample)

Please follow this template for OpenCV samples on Python:

```
#!/usr/bin/env python

'''
Brief description (what is demostrated).
Usage example, parameters
'''

# Python 2/3 compatibility
from __future__ import print_function

import numpy as np
import cv2 as cv

import sys


def main():
    # use sys.argv to parse arguments
    # ...


if __name__ == '__main__':
    print(__doc__)
    main()
    cv.destroyAllWindows()
```

If you need to store global variables (using UI callbacks), then it is better to add application class and store them there:

```
class App():

    def run(self):
        # ... use "self.tuned_parameter" instead of global variable
```

```
if __name__ == '__main__':
    print(__doc__)
    App().run()
    cv.destroyAllWindows()
```