- Authors: Vadim Pisarevsky
- Sample code and discussion: #17997
- Status: Draft
- Platforms: All
- Complexity: TBD (estimated complexity; certain amount of man-weeks, man-months etc.)

Introduction and Rationale

From time to time we face the following problem in OpenCV – how to add some extra parameter to an algorithm while preserving backward compatibility. Since OpenCV 4.0 we relaxed compatibility requirements. It's source-level compatibility, not binary compatibility. But still, even at the source level it may be difficult to add a new parameter.

Here are the methods that we currently use: 1. add another optional parameter to existing function. 2. create an overloaded function with this extra parameter 3. convert algorithm to a class and add necessary `set<Param1>() set<Param2>()` etc. methods 4. add `vector<int>` or `vector<double>` function argument in which users can put the pairs `param_id1, value1, param_id2, value2 ...` 5. more flexible variant of the previous approach, where instead of vector of numeric values we pass a dictionary of parameters.

Each of the methods has some drawbacks: 1. when the number of parameters becomes large, it's difficult to "decrypt" the function call when studying some code. Need to check the description and carefully identify which parameters get which values. Also, users have to manually specify parameters of all previous optional parameters in order to change the one they need. It may be inconvenient. Also, with a large number of parameters there is higher risk to put parameters in the wrong order (and compiler will not provide any diagnostic in this case when parameters are of the same type). 2. overloaded functions also make the code comprehension more difficult. And also overloaded functions may create a real problem for some bindings, e.g. Python or Javascript bindings, which do not have embedded overloading mechanism. 3. conversion of an algorithm to a class is probably the most poweful solution in w.r.t obtained extendibility. But such classes take more lines of code to use. Instead of a single call, users have to write several: create class, tune parameters one by one and then call the processing method. 'Call chain' approach may somewhat help, but it's still quite a bit of typing. Besides, it's more difficult for developers to convert functions to classes, make a proper future-proof design. And the classes may also be a bit challenging for the wrapper generators. 4. we use vector of parameters for a few functions currently. Most notable example is cv::imwrite function, where user can set jpeg or png compression, for example, using those extra parameters. This approach of using `(property_id, value)` pairs can also be combined with the class approach. `VideoCapture` is well-known example of this approach. This approach is quite flexible for certain scenarios (in the case of `VideoCapture` it's still more or less adequate, ~20 years after its introduction

and ~10 years after migrating this approach to OpenCV's C++ API). But in general it's not convenient to use, not that easy to implement, it's error-prone and not intellisence-friendly. 5. We currently use this dictionary approach in OpenCV DNN, and it seems to be quite suitable for this task; but for most algorithms implementing such approach would be too heavy, the overhead will be noticeable and in this case it's probably better to go with the method 3, which is intellisence-friendly.

Proposed solution

Is there a better solution? The ideal solution should have the following properties:

- be fast, i.e. add just a little overhead to encode, pass and decode the parameters when the algorithm is invoked.
- easy to implement
- should not break source-level compatibility when new parameters are added
- type-safe, i.e. there should be a compile error when parameter of unknown name or wrong type is passed
- as a direct consequence from the previous item, it should be intellisence-friendly, so that IDE could hint which algorithm parameters are available and probably even mark it in the editor when wrong id/value is passed
- the code should be easy to read, i.e. from the algorithm call it should be clear which parameters are set to which values
- crisp. That is, the code should be compact. There should be a mechanism to leave some or all of the parameters at their default values and specify only what you need to change.

It looks like a long list, and since none of the current OpenCV approaches has all those properties, it may seem impossible to implement.

However, if we look at some modern languages, like Python, Ocaml etc., we will see that there is well-known notion of named arguments, which provide everything of the above (well, in the case of Python passing named arguments is not very efficient, but it's rather due to the implementation and dynamic nature of the language).

For example, this is how some hypothetic algorithm can be invoked in Python:

```
H, inliers = my_homography_estim_algorithm(points1, points2, method=RANSAC, eps=3, maxiters=
```

It looks very natural, it's easy to read, easy to write (crisp), easy to implement, it's extendible etc.

Apparently, with the modern C++ standards we can implement such solution in C++ as well. The feature is called designated aggregate initialization. Formally, it's available since C++ 20, but reasonably new versions of clang and GCC support this syntax too when C++ 14 standard or above is used.

The suggested approach

- for each more or less complex algorithm `cv::foo` we make yet another overloaded variant that takes `cv::FooParams` structure as the last parameter. We put there all the flags and optional parameters.
- The structure does not contain explicit constructor or any other explicit method. It just contains some fields that are initialized to some "default" values:

```
struct FooParams
{
     int param1 = -1;
     float params2 = 0.f;
     std::string param3 = "";
     ...
};
rettype foo(type1 arg1, type2 arg2, ..., const FooParams& params=FooParams());
```

- *it's suggested to keep all the parameters in alphabetic order, initially and when adding new parameters*, since C++ requires that the order of parameters in the aggregate initialization construct matches the declaration order.
- that's it. Users can now call the function with simple `foo(real_arg1, real_arg2, ..., { .param2 = 3.14f });` where param1 and param3 will be automatically set to their default values.
- this approach is perfectly compatible with Python, where such functions can be recognized and automatically converted to Python functions with keyword parameters.

Possible issues

What if users have some compiler that does not support aggregate initialization? As long as `C++ 11` is supported (which is the minimum supported version of C++ since OpenCV 4.x), OpenCV headers and source code will still compile, but the function call will take some more lines of code:

```
FooParams params;
params.param2 = 3.14;
foo(real_arg1, real_arg2, ..., params);
```

In the sample in the comment below there is also 'call chain' method demonstrated. In brief, we can add 'set' methods that return the altered class itself and call them sequentially to generate the needed set of parameters:

```
foo(real_arg1, real_arg2, ..., FooParams()._param2(3.14));
```

Note that the old-style structure initialization will not compile:

```
FooParams params = { -1, 3.14, "" }; // compile error will be reported here
```

```
foo(real_arg1, real_arg2, ..., params);
```

because the structure is not a C structure, it provides some default values for its members and thus some default constructor is generated. And this is a good property, since it eliminates possibility that the code starts to misbehave with a newer version of API when some new parameters are inserted in the middle.

Impact on existing code, compatibility

The solution is completely source-level compatible with existing OpenCV.