Simple device server

You are writing code for a server that act as a go-between a user and a set of devices controlling power converters. The server takes text commands from users, parses them and controls the devices accordingly. A single server can control any number of devices.

Commands syntax

As of now, there are 4 commands the user can issue (but there might be more in the future) with the following syntax:

1. `s name <id> <name>`

   Sets the name of the devices, where `<id>` is a number (>=0) identifying the device,
   and `<name>` is a string of any length (>0) that may contain only lower-case letters, digits, and an underscore.

   Example: `s name 0 pc_magnet_1`

2. `g name <id>`

   Gets the name of the device with `<id>`. If the name was not set, `<id>` is returned as the name.

   Example: `g name 0` → `pc_magnet_1`

3. `s params <id> <param1,...,paramN>`

   Sets parameters of the device with given `<id>`. Parameters is a list of any length (>0) containing numbers from 0-255, in any order.
   Every time the command is issued, it replaces the current set of parameters.

   Example: `s params 0 1,3,2,0,255,67,67`

4. `g params <id>`

   Returns a list of device parameters, separated by a comma, **in an ascending order**.

   Example: `g params 0` → `0,1,2,3,67,67,255`

Repository

Write your solution in the file `solution.cpp`. It contains a little bit of code that you can start working with. The `main()` function has some examples that should make clear how the commands and the server should work. There is also a very simple `Makefile` which you may use to compile your code.

Ignore the file `physical_device.h` - it is there just to make the compilation possible (treat it as if this was an external library, outside of your project).

Tasks

1. **Look at the `Device` class in the file `solution.cpp`.**

   - Suggest what could be improved/added/removed in this class.
   - Decide how to store and add a set of the device parameters.
   - You may rewrite or extend the class as you wish.

2. **Write the `Server` class, that fulfils the following specification:**

   - It stores the `Device` objects. Order of the devices corresponds to their IDs (i.e. the first added device gets ID 0, the second added gets 1, and so on). Once added, the device is never removed.
   - It has a public, **blocking** method `addDevice` that adds passed `Device` object to the stored devices. You decide how to store and pass the device object. Your code does not create a `Device` object, the caller will pass an already-created `Device` instance (*see `main()` function for example*).
   - It has a public, **non-blocking** method `createAction` that takes one parameter `std::string` and returns an `Action` object (*see the description below*). The `std::string` parameter is a user command (*as described above*). As it comes from a human user, the syntax might be erroneous.
   - Both `addDevice` and `createAction` can be invoked from different threads and the Server class has to be thread-safe.
     However, `addDevice` should block the calling thread until the device is added, while `createAction` should be non-blocking and return immediately. (*For simplicity, it is OK to spawn new threads on each invocation of this function – if that's helpful for your solution.*)

3. **Write the `Action` class, that fulfils the following specification:**

   - It has a public, **blocking** method `getResult`, that blocks until the user command is executed. It takes no arguments. The method should allow the calling thread to get the command result as an `std::string` and to check if the command execution was successful or not (e.g., if the command syntax was valid). You decide how, but there is no need to pass any information on why the command execution was not successful – only that it failed. For *set commands* that don't return anything, an empty string shall be returned.

You can implement more classes or methods, where you find it appropriate. Unit tests are not necessary.
Write code that is clean, well-documented, thread-safe, and easy to maintain and extend.

Rules

1. You may use any C++ standard – however, fairly modern code is encouraged.
2. You may use everything that can be found in the C++ Standard Library.
3. Explain your reasoning/comment your code where you find it appropriate.