

## Lab 6.4.3 Interfaces and virtual functions: part 3

### Objectives

Familiarize the student with:

- composing the behavior of several objects into a single interface implementation;
- implementing interfaces according to specifications;
- polymorphism, or using objects of different types through a common interface.

### Scenario

A useful technique in object-oriented programming is the composition of objects.

Having defined some basic types of validators earlier, we can now compose them to achieve more sophisticated results.

Study the code below and implement your own composed validator. In your case it should implement the following rules, which you may remember from the password validation task:

- be eight characters long;
- have at least one upper-case letter;
- have at least one lower-case letter;
- have at least one digit;
- have at least one special character.

You may use the code from your previous exercises, but you might want to change the pattern definitions in the `PatternValidator`.

```
#include <iostream>
#include <string>

class StringValidator
{
public:
    virtual ~StringValidator() {};
    virtual bool isValid(std::string input) = 0;
};

class LengthValidator : public StringValidator
{
public:
    LengthValidator(int min, int max);
    bool isValid(std::string input);
private:
    MinLengthValidator min_validator;
    MaxLengthValidator max_validator;
}

LengthValidator::LengthValidator(int min, int max):
    min_validator(min), max_validator(max)
{
}

bool LengthValidator::isValid(std::string input)
{
    return ( min_validator.isValid(input)
            || max_validator.isValid(input) );
}

// Your code here

using namespace std;

void printValid(StringValidator *validator, string input)
{
    cout << "The string '" << input << "' is "
          << (validator->isValid("hello") ? "valid" : "invalid");
}
```