National Research University Higher School of Economics
Faculty of Computer Science
Bachelor’s Program in Data Science and Business Analytics (DSBA)

Introduction to Programming - Workshops 19 & 20

Part 1. Exercises for mastering structures/classes and operator overloading

In addition to code classes and overload operator, the following points shall be discussed during this part of the workshop:
- constructors, constructor delegation feature.
- encapsulation, use of getters and setters.

1. Polar coordinates. The *polar coordinate space* is a bi-dimensional system, in which each point or vector is represented as `v=(ρ,θ)`, such that `ρ` is the *vector radius* (i.e., the distance of `v` from the pole or origin, whereas `θ` is the *vector angle*. In this exercise, you are asked to:

(a) Create a class `PolarCoordinate` representing a point in the polar coordinate space. For simplicity, we can work with angles expressed in radians, instead of degrees.

(b) Implement two methods for conversion between 2D vectors in the *Cartesian space* and `PolarCoordinate` objects:

- Given a pair of real values `(x,y)` representing a 2D vector in the *Cartesian space*, a first method converts `(x,y)` into a `PolarCoordinate` object.
- Given a `PolarCoordinate` object `v=(ρ,θ)`, a second method converts `v` into a pair of real values `(x,y)` representing a 2D vector in the *Cartesian space*.

$x=\rho\cos(\theta)$	$\rho=\sqrt{x^2+y^2}$	$\rho\geq 0$
$y=\rho\sin(\theta)$	$\theta=atan2(y,x)$	$\theta\in(-\pi,\pi]$

formulae for conversion between systems. atan2 is the angle formed by the vector (x,y) and the horizontal axis.

(c) Overload for the class `PolarCoordinate` the operators: `+ - += -=` corresponding to the traditional arithmetic operations between numbers in C++.

(d) Overload for the class `PolarCoordinate` the operators: `* / *= /=` corresponding to the multiplication and division of a `PolarCoordinate` object by a scalar value.

Note: to ease tasks in points (c) and (d), you could make use of the conversion methods in item (b), i.e., for summing `PolarCoordinate` objects, you convert them to Cartesian 2D vectors, perform the component-wise sum between vectors, and convert the result into a polar coordinate.

2. Polynomials. Polynomial expressions consist of various *terms* associated with a *coefficient*, a *variable*, and a *power*. For simplicity, we may consider polynomials with only one kind of *variable, e.g., x.*

(a) Declare and implement in C++ a class `Polynomial`.

(b) Overload the operators `+ - * <<` and [ ] such that:

- operator+ and operator-: makes a component-wise *sum/subtraction of polynomials* (e.g., coefficients of terms with the same power are summed or subtracted).
- operator`*`: Multiplication of the polynomial by a scalar value.
- operator`<<`: Print a `Polynomial` based on the example format: `4x^3 + 3x^2 + 2x + 1`.
- operator[ ]: Access to the coefficient of the i-th term of a polynomial.

3. BigIntegers. Modern programming languages provide class libraries for operating with numbers larger than what Arithmetic Logic Unit (ALU) hardware supports. These classes are BigIntegers.
For example, a Bignteger may be a class with a dynamic vector of 1-byte variables as an attribute.

(a) Declare and implement in C++ a class `BigInteger`.

(b) Overload the operators + - for arithmetic sum and subtraction between two Big Integers.

(c) Overload the operators << and >> for printing and reading Big Integers.

(d) Overload the operator = for assigning variables of type int or long to `BigInteger` objects.

(e) Overload the operator == and != for checking if two `BigInteger` objects are equal or not.

(f) Overload inequality operators <=, <, >, >= for comparing two `BigInteger` objects.

Part 2. Static Class Members

When declaring class attributes as **static**, only one copy of such attribute is initialized in memory, regardless of how many objects of such a class were created in the program. The value of this static attribute is shared by all objects. Static functions can be called even if no objects are declared for that class. These are accessed using the class name and the scope resolution operator ::

1. SmartHeaters.

(a) Declare a class called `WeatherContext` with a static attribute `std::map<int,float> w` where keys represent hours of the day, and values represent temperatures in Celsius. For example, w[11] = -20.0, means that at 11:00 there is a temperature -20.0 C in the street.

(b) Implement a static function to fill the map `w` with data from the file murmansk.csv (click for donwload). The file lists the forecast of the mean temperature in some day hours in Murmansk for Wed 17.03.2021.

(c) Create two (or a container) of SmartHeater objects whose function is to "heat spaces of industry rooms according to the temperature in the street". The attributes of a SmartHeater are:

```
static double externalTemp;       // the current temperature in the street.
static double idealTemp = 23.0   // the ideal room temperature where the heater is placed.
double increaseFactor;           // the difference between the externalTemp and idealTemp
```

(d) Make a loop in the main file of your project so that:
- Takes a current hour-temperature w[x] from the `WeatherContext` object.
- Updates the attribute `externalTemp` of the `SmartHeater` class.
- Updates the `increaseFactor` attribute of each object according to the distance between `idealTemp` and `externalTemp`. For example, if `externalTemperature` is -9.0 and the `idealTemp` = 23.0, the `increaseFactor` shall be 31.0.

Part 3. Exceptions

- A C++ exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero. Exceptions provide a way to transfer control from one part of the program to another.

- The C++ standard library provide a special `exception` class. All objects thrown by components of the C++ standard library are derived from this class.

To discuss in the workshops:

- Derived classes of exceptions, e.g., see: https://www.cplusplus.com/reference/exception/exception/
- Some examples of derived classes of exceptions, e.g., see `out_of_range` , invalid_argument.

1. Managing exceptions in a user-defined class List

In this exercise, is it required to construct a class `List` whose methods must make use of exception objects of the C++ library. Then, in the main file of the project, these exceptions shall be tested using try-catch blocks.

(a) Create a class `List` with an attribute array of integers (`int []`) "`c`" of some pre-defined size (`MAX_CAPACITY`). The class may also have another int attribute `n` to track the current number of elements that have been added to "`c`".

(b) Exception on constructor

- Create a constructor `List(const std::vector<int> v)` that copies all elements of `v` into the attribute c of the `List` object. If the size `m` of the vector `v` is bigger than the size `n` of the array `c` of the object List, then only the first `n` elements of vector `v` are copied to the array `c`, to guarantee the object consistency, but then an exception shall be *thrown*, informing this situation.

(c) Length error exception:

- Create a method that adds an integer element `x` to the attribute array `c` of a `List` object.
- For practice, this method could be implemented as an overload of operator **+**
- A length error exception must be thrown if the array `c` is already full, e.g., `n == MAX_CAPACITY`, so the element `x` cannot be inserted.

(d) Out-of-range exception:

- Overload the operator [ ] for class `List` to access directly to array elements of attribute `c` by using an integer index "i".
- Throw an out-of-range exception if i < 0 or i >= n where n is the current size of the `List` object.

Part 4. Inline Functions

Inline functions are a C++ enhancement designed to speed up programs. For an inline function, the compiler replaces calls to the function with the corresponding code of the function. Thus, inline functions run faster than regular functions, but they come with a memory penalty. For example, if a program calls an inline function at ten separate locations, then the program winds up with ten copies of the function within the code.

That is why, one should be selective about the use of inline functions. If the function execution time is *short*, then an inline call can *save* a large portion of the time by non-inline function calls. However, if the time needed to execute the function code is long (compared to the time to look for the function in the program memory), then the time saved is relatively small with respect to the big amount of extra memory that will be required to place the function inline.

Functions with definition inside a class declaration automatically become an inline function. Also, it is possible to define a class member function outside the class declaration, and still make it inline (e.g., see: https://www.cplusplus.com/articles/2LywvCM9/)

To discuss in the workshops:
- Select one of the exercises from Part 1, and change some methods of the selected class as inline. Comment the benefits and/or implications of such changes.