OBJECT ORIENTED PROGRAMMING USING C++ - PROGRAMMING EXERCISES

Jorge A. Silva

1.

Consider the following definition of class **Date** that will be used to represent dates:

```
class Date
{
public:
  Date(unsigned int year, unsigned int month,unsigned int day);
  Date(string yearMonthDay); // yearMonthDay must be in format "yyyy/mm/dd"
  void setYear(unsigned int year) ;
  void setMonth(unsigned int month) ;
  void setDay(unsigned int day) ;
  void setDate(unsigned int year, unsigned int month, unsigned int day) ;
  unsigned int getYear() const;
  unsigned int getMonth() const;
  unsigned int getDay() const;
  string getDate() const;  // returns the date in format "yyyy/mm/dd"
  void show() const;       // shows the date on the screen in format "yyyy/mm/dd"
private:
  unsigned int year;
  unsigned int month;
  unsigned int day;
};
```

a) Write the code of all the methods of the class. For now, consider that the dates represented in objects of class **Date** are valid.

b) Write the **main()** function of a program that tests all the methods of the class.

c) Add the following methods to the class:

- **isValid()** - returns a boolean value indicating whether the date is valid or not; hint: implement a private method that returns the number of days of a given year-month pair;
- **isEqualTo(const Date &date)** - returns a boolean value indicating whether the date represented in an object is equal to the date received as parameter;
- **isNotEqualTo(const Date &date)** - returns a boolean value indicating whether the date represented in an object is not equal to the date received as parameter;
- **isAfter(const Date &date)** - returns a boolean value indicating whether the date represented in an object is after date received as parameter;
- **isBefore(const Date &date)** - returns a boolean value indicating whether the date represented in an object is before the date received as parameter.

Note: in the future you will learn how to overload the operators **==**, **<** and **>**, so that they can be used to compare objects of type **Date**, as well as operator **<<** to output a date.

d) Modify the **main()** function of the program in order to test the new methods.

e) Explain why the following variable declaration is not possible: **Date d1;** .

f) Add a default constructor to class **Date**, that must construct an object representing the current date; you must search for a way of obtaining the current date from your computer.

g) Modify the implementation of the class so that the atributes (**year**, **month** and **day**) are represented in a single string, using the format "**yyyymmdd**". Verify that the **main()** function that you developped in **d)** does not have to be modified.

2.

The following class was defined to store the relevant information about the grades of a student, in a specific course. The final grade depends on the grades obtained in the short exam, the project and the exam of the course.

```
class Student {
public:
  Student();
  Student(const string &code, const string &name);
  void setGrades(double shortExam, double project, double exam);
  string getCode() const;
  string getName() const;
  int getFinalGrade() const;
  // other get and set methods
  bool isApproved() const; // is the student approved or not ?
  static int weightShortExam, weightProject, weightExam;  // weights in percentage (ex:20,30,50)
private:
  string code; // student code
  string name; // student complete name
  double shortExam, project, exam; // grades obtained by the student in the different components
  int finalGrade;
};
```

a) Write the code of the `setGrades()` method. This method should also set the value of the `finalGrade` attribute, taking into account the values of its parameters, and the weights of the `shortExam`, `project`, `exam` components in the `finalGrade` value, which are, respectively, `weightShortExam`, `weightProject`, `weightExam`. The calculated value must be rounded to the nearest integer; when the decimal part of the calculated value is 0.5 the `finalGrade` must be rounded up. Justify the use of `static` qualifier for the attributes `weightShortExam`, `weightProject`, `weightExam` and define them with the values 20, 30 and 50, respectively.

b) Implement the remaining methods of class **Student** and develop a program to test all the methods.

c) Write a function, **readStudentData()**, that reads from the keyboard the code, name and grades of a student (in the short exam, project and exam) and returns an object of type `Student`, having the attributes read from the keyboard. On the right, you can see an example of a dialog with the user, during the execution of this function.

Example of the execution of the function:

```
Student code? up20179999
Student name? Ana Silva
Short exam grade? 13.5
Project grade? 17
Exam grade? 15.7
```

d) Write the code of a function, **showAboveAverageStudents()**, that receives as parameters an ouput stream and a `vector<Student>` and writes to the output stream, all the data about the students whose `finalGrade` is above the average `finalGrade`. The data fields must be vertically aligned.

e) Write a program that reads from the keyboard the data about the students that took a course, storing them into a `vector<Student>`, and shows on the screen the list of "above average" students.

f) Modify the program so that the student's data is read from a text file and the ouput is saved to another text file. The input data about each student must be stored in a single text line, all items being separated by semicolons. For example, the data of the example above should be stored as: `up20179999;Ana Silva;13.5;17;15.7` .

3.

Define a class **Person** for representing persons. Consider that a person has the following attributes: name, gender and birthdate. Use an object of class **Date** (see problem 1) to represent the birthdate. Develop your program using separate compilation.

4.

(*adapted from Big C++ book*) Develop a program to print out an invoice. An invoice describes the charges for a set of products, acquired in certain quantities. Complexities such as dates, taxes, and invoice and costumer numbers may be omitted. The program simply prints the billing address, all the items (description, unit price, quantity ordered and total price), one item per line, and the total amount due. For simplicity, no user interface is required; simply use a test harness that adds items to the invoice and then prints it. Your program must include the classes: `Client` (describes the client name and address), `Product` (describes a product with a description and price), `Item` (describes a line of the invoice; see example below) and `Invoice` (describes an invoice for a set of purchased products).

Example of output:

```
DEI- FEUP
Rua Dr. Roberto Frias, s/n
4200-465 Porto

Description          Price  Qty   Total
-------------------- ------ ----- --------
Computer        999.90    10  9999.00
Printer         149.90     1   149.90

Amount due: 1148.90 euro
```

5.

Develop a small program for the management of a library. The program must allow the management of the books and of the users, as well as the borrowed books. All the information must be permanently saved in files.

For simplicity, you may assume that books and users are never removed from the system, although books may be "lost" and users may be "not active" (use an attribute to store those properties).

Notes:

- use `const` qualifier, both in methods and in method parameters, whenever you find they should be used;
- you may improve your program by adding to each book the date when it was taken by the user, using objects of classe `Date`, from a previous exercise, for that purpose;
- use separate compilation to develop the program.

6.
(*adapted from Big C++ book*) Develop a small program to implement a simple inventory management system. The program must use two classes: **Product** and **Inventory**. A product must have an identification number, a description and the available quantity. Products have unique identification numbers, starting with 1, that must be automatically assigned to each new product. The inventory system has two operations: 1) process an order for a particular product number; products can only be ordered one at a time; if the product is available it is immediately sold, otherwise it is put in a back-order list; 2) process the shipment of a given product; when a product arrives in shipment the back-order list is searched and all the orders that match the product identification number are processed (the corresponding product is sold). Hints: use two classes, **Product** and **Inventory**; in the Inventory class, use a list to represent the back-orders, so that they are processed on a first-come, first-served basis.

7.
Write a template function that determines the minimum and maximum values of a vector of numbers that it receives as parameter.

8.
a) Suppose you didn't have class **vector** from the STL. Define and implement a template class, **Vector**, that emulates the functioning of class **vector** from the STL. Your class **Vector**, defined below, must only implement a limited set of functionalities; the functionality of the methods is the same as in **vector** (from the STL). The memory for the elements of **Vector** must be allocated dinamically. Hints: use function **malloc()** for dynamic memory allocation; to implement method **push_back()** investigate the use of function **realloc()** to increase the space allocated for the buffer.

```
template <class T>
class Vector
{
public:
  Vector();
  Vector(unsigned int size);
  Vector(unsigned int size, const T & initial);
  Vector(const Vector<T> & v);
  ~Vector();
  size_t size() const;
  bool empty() const;
  T & front();
  T & back();
  T & at(size_t index);
  void push_back(const T & value);
  void pop_back();
  void clear();
private:
  T * buffer;
  size_t bufferSize;
};
```

b) In the **vector** class of the STL, the reallocation of memory is not done every time a new element is pushed back; instead, it is allocated in new chunks. The capacity of a **vector** tells us how many elements the vector could hold before it must allocate more space. Modify the definition and implementation of the class methods in order to include a new attribute **bufferCapacity** and two new methods: **capacity()** that tells the current capacity of the **Vector** object, and **reserve()** that requests that the vector capacity be at least enough to contain **n** elements, where **n** is a parameter of the method. Before doing your implementation, try to infer how the capacity of a vector is increased in the STL; for that, start with an empty STL **vector** object, make a lot of push backs, and determine its capacity after each one of the push backs.

9.
Write a program to eliminate duplicates in a set of unordered words, stored in a text file. The values must be read into an STL **vector**. The ordered set of words must be written to the same text file. Use STL algorithm **sort()** to sort the words; to eliminate duplicates, implement two versions of the program:
a) Using your own code to remove the duplicates.
b) Using STL algorithm **unique()** to remove the duplicates.

10.
Repeat the previous problem using an STL **list** to store the words read from the file. In this case it not possible to use the STL algorithms **sort()** and **unique()**? Why? Investigate the use of alternative solutions.

11.

Modify the program developed in problem **2.d** so that the output can be sorted either by student name or by student grade. Use STL algorithm **sort()** for this purpose. Note: in STL **sort()**, besides the range of the elements to sort, you can specify a function that will be used to compare the elements to be sorted.

12.

The process of betting on EuroMillions requires you to choose five "main numbers" from 1 to 50 and two "lucky stars" from 1 to 12. Write two versions a program to choose the "main numbers" and the "lucky stars" of a simple bet in EuroMillions and show the numbers on the screen.
a) Use two STL **vectors** to represent the "main numbers" and the "star numbers". Start by filling the "main numbers" vector with all the numbers from 1 to 50 and the "star numbers" vector with all the numbers from 1 to 12. Then suffle the two vectors using STL algorithm **random_shuffle()**. Finally, keep only the first 5 numbers of the "main numbers" vector and the first 2 numbers of the "star numbers" vector.
b) Use two STL **sets** to represent the "main numbers" and the "star numbers". Implement a random number generator function and repeat the number generation until the "main set" contains 5 numbers and the "star set" contains 2 numbers.

13.

a) Extend the program developed in problem **12.a** to do the following:
- automatically generate a lot of N simple bets in EuroMillions, storing them into a **vector<vector<int>>**; N must be specified interactively by the user;
- then, simulate the generation of the key, storing it into a **vector<int>**;
- finally, show the results for each one of the bets, taking into account the simulated key (note: use the STL algorithm **set_intersection()**).

b) Do the same for the program developed in problem **12.b**, using a **vector<set<int>>** to store all the bets and a **set<int>** to store the key.

14.

Suppose that you want to store a word index table that indicates all the lines where each word occurs, in a given text. For example, in the text on the right, the word "programs" occurs in lines 1, 2, 10, 11 and 17, and the word "language" occurs in lines 9 and 13. When a word occurs several times in the same line, only one occurrence must be registered (example: "programs" occurs twice in line 11, but line 11 will appear only once in the table).
a) Declare the data structure that you would use to store (in main computer memory) the word index table. You may use STL containers if you find them useful. Justify your answer, briefly.
b) Using the data stucture that you have chosen in the previous question, write a program that does the following:

```
 1  Programs that will run as console applications
 2  under Visual Studio C++ are programs that
 3  read data from the command line and output
 4  the results to the command line. To avoid
 5  having to dig into the complexities of creating
 6  and managing application windows before you
 7  have enough knowledge to understand how
 8  they work, all the examples that you will write
 9  to understand how the C++ language works
10  will be console programs, either Win32
11  console programs or .NET console programs.
12  This will enable you to focus entirely on the
13  C++ language in the first instance; once you
14  have mastered that, you will be ready to deal
15  with creating and managing application
16  windows. You will first look at how console
17  programs are structured.
```

- reads the names of 2 text files: a file containing a list of simple words (ex: words.txt) and a text file containing a book or a book sample, like in the example above (ex: book.txt);
- reads the list of words and the book into internal data structures;
- builds the internal data structure that stores the index table for the words contained in the list of words and, at the same time, reads the book into a separate data structure (**vector<string>**); hint : copy the book line to a temporary string, replace all the punctuation marks (full stop, comma, semi-colon, …) in this temporary string with space character and the store it into a **stringstream**, then use **operator >>** to extract the words from the **stringstream**.
- repeatedly asks the user for a word (until the user types CTRL-Z), verifies if the word is present in the list of words and, if so, shows the line number where it is present as well as the corresponding line contents.

Example of the interaction with the user when the program is executed:

```
WORD? application
RESULTS:
      1: Programs that will run as console applications
      6: and managing application windows before you
     15: with creating and managing application
```

c) For very large books it may not be convenient to read the book into an internal data structure. Propose and implement an alternative solution for this case (note: take into account that the lines in the text file to be indexed may have not the same number of letters, as illustrated above).

15.
Consider the class **Date** from problem 1. Implement the following modifications/additions and develop a program to test them.
a) Overload the operators ==, **!=**, **<**, **<=**, **>**, **>=** , **>>** and **<<**, for class **Date**.
b) Modify the definition of the 'set' methods so that their return value is a **Date &**. Explain the implications of returning a **Date &**.
c) Overload operators **++** and **--** (both postfix and prefix) for objects of class **Date**; the result of this operator should be the date of the next and the previous day, respectively (taking into account the date stored in the object).

16.
The STL template function

template <class *ForwardIterator*, **class** *UnaryPredicate***>**
ForwardIterator **remove_if (***ForwardIterator* **first,** *ForwardIterator* **last,** *UnaryPredicate* **pred);**

transforms the range [**first**,**last**[into a range with all the elements for which **pred** returns **true** removed, and returns an iterator to the new end of that range. **pred** is a unary function that accepts an element in the range as argument, and returns a value convertible to **bool**; the value returned indicates whether the element is to be removed (if **true**, it is removed); the function shall not modify its argument; this can either be a function pointer or a function object.
Given a **vector<int>**, containing positive an negative values, write two programs that use **remove_if()** to:
a) remove all the negative values; use a "common" function to implement the *UnaryPredicate*.
b) remove all the values in a range specified by the user of the program; use a function object to implement the *UnaryPredicate* and explain why a function object is necessary, in this case.

17.
Define two functions: **void cout(const string &s)** and **bool endl(const string &s)**. The first one shows string **s** on standard output, removing all "non-letter" characters from **s**; the second one returns true if **s** ends in a lowercase letter. Place both functions in a namespace whose name **myfuncs**. Write a program to test the developed functions.

18.
Implement and test a program that uses the class hierarchy presented in the theoretical lectures: **FEUP_person**, **Teacher** and **Student**.

19.
Suppose that you want to build a class hierarchy to represent a spreadsheet. For simplicity, let us consider that spreadsheet cells can store only two types of values: **string** or **double**. Below is the definition of the base class, **SpreadsheetCell**:

```
class SpreadsheetCell  // an "abstract class"
{
public:
  SpreadsheetCell() { };
  virtual ~SpreadsheetCell() { };
  virtual void set(const std::string& inString) = 0;
  virtual std::string getString() const = 0;
};
```

Note that, independent of the type of cell, it must be possible to set/get its value as a **string**.
a) Define the classes **SpreadsheetCellString** and **SpreadsheetCellDouble**, derived from class **SpreadsheetCell**, that will be used to store **string** or **double** cell values. For simplicity, assume that **strings** used as arguments of **set()** method to set the value of cells of type double represent valid **double** values (that is the **string** can be converted into a **double**).
b) Write a program that creates a very simple spreadsheet, consisting of an unidimensional array/vector that stores both **string** and **double** values.
c) Modify the definition of the classes so that you can have specific methods to set and get the value of a **SpreadsheetCellDouble** using values of type **double**.
d) Modify the developed program to test the newly developed methods.

20.
Consider the class **Date** of problem 1.
a) Modify it so that it throws an exception when the constructors or any of the set methods receive as parameter(s) some values that would constitute an invalid date. The thrown object must be of class **InvalidDate** that you must define.
b) Modify the program developed in problem 1 to test the exception handling additions made to class **Date**.