Examination Guide

Contents

1. Objectives 2
2. Course Contents 2
3. Evaluation 3
   3.1. During the semester 3
   3.2. During the examination session 3
   3.3. During the retake session 3
4. Examination dates 4
5. Written Examination 5
   5.1. OOP concepts – C++ 5
   5.2. Inheritance, polymorphism, UML, design patterns – C++ 8
6. Practical Examination 10
   6.1. Problem statement 10
   6.2. Advice for the practical examination 11
7. Bibliography 13

1. Objectives

- Solve small/medium scale problems using Object-Oriented Programming.
- Explain class structures as fundamental, modular building blocks.
- Understand the role of inheritance, polymorphism, dynamic binding and generic structures in building reusable code
- Use classes written by other programmers, use libraries (STL).
- Write small/medium scale C++ programs with simple graphical user interfaces.

2. Course Contents

1. C Programming Language Fundamentals
2. Functions, Modular Programming and Memory Management in C
3. Object-Oriented Programming, Classes and Objects in C++
4. Templates, C++ Standard Template Library
5. Inheritance, UML diagrams
6. Polymorphism
7. I/O Streams, Exception handling
8. RAII, Smart Pointers in STL
9. Qt framework
10. Signals and slots in Qt
11. Model/View Architecture
12. Design Patterns
13. Concurrency, Move semantics
14. Revision

3. EVALUATION

3.1. During the semester

Lab grade – **L (30% of the final grade)**:

- 50%: Lab assignments: weighted arithmetic mean of 9 or 10 (for the optional Lab14) lab assignments, with 0 for the labs you did not present.
- 50%: 3 practical tests during the labs (10%, 15%, 25%)
- Additional work: 0 - 0.5 (laboratory bonus - LB) – **this is added to the final grade**.

The precondition to attend the examination in the regular session is: L ≥ 5 (no rounding).

Seminar bonus – **SB**: 0 – 0.5 – optional bonus for your activity during the seminars. **This is added to the final grade.**

3.2. During the examination session

Written examination – **W (40% of the final grade)**: on your examination date.

Practical examination – **P (30% of the final grade)**: on your examination date, after the written examination. You are encouraged to take the practical examination on your own laptops.

Final Grade: G = 0.3 x L + 0.3 x P + 0.4 x W + LB + SB

To pass the examination all grades (L, W, P) must be ≥ 5 (no rounding).

3.3. During the retake session

- The final grade in the retake session is computed by the same algorithm presented above.
- During the retake session you can hand in laboratory work, but are limited to a maximum laboratory grade (**L**) of 5.00. *All lab assignments should be properly solved in order to receive 5.*
- You can choose to retake the written, practical, or both examinations in case you have failed/not attended during the regular examination session.
- If you want to increase the grade you obtained during the regular session, you may participate to the examinations during the retake session. Your final grade will be the largest one between those obtained.

4. EXAMINATION DATES

16.06.2017 – group 917 (backup date for groups 911, 914)

19.06.2016 – groups 912, 913 (backup date for group 917)

20.06.2016 – groups 915, 916 (backup date for groups 912, 913)

23.06.2016 – groups 911, 914 (backup date for groups 915, 916)

Important observations!

- Make sure you've fulfilled your financial obligations towards the University, otherwise we are not allowed to grade you.
- To take the exam on the backup date, you need a very good reason. In such situations, send me an email (iuliana@cs.ubbcluj.ro) **at least 48h beforehand!**
- Re-check the date/time of the exam beforehand.
- Arrive on time and bring a photo ID.
- Bring your laptop.

5. Written Examination

5.1. OOP concepts – C++

- templates, STL (containers, algorithms, iterators, smart pointers);
- dynamic memory allocation;
- constructors, destructors, inheritance, virtual methods, abstract class, operator overloading, static, friend elements;
- input output streams;
- exceptions;

Given the test function below, implement the class **Stack**. Specify the method which adds an element to the stack.

```
void testStack()
{
	Stack<std::string> s{ 2 };
	assert(s.getMaxCapacity() == 2);
	try	{
		s = s + "examination";
		s = s + "oop";
		s = s + "test";
	}
	catch (std::exception& e) {
		assert(strcmp(e.what(), "Stack is full!") == 0);
	}
	assert(s.pop() == "oop");
	assert(s.pop() == "examination");
}
```

Define the classes “ToDo” and “Activity” such that the following C++ code is correct and its results are the ones indicated in the comments.

```
void ToDoList()
{
	ToDo<Activity> todo{};
	Activity tiff{ "go to TIFF movie", "20:00" };
	todo += tiff;
	Activity project{ "present project assignment", "09.20" };
	todo += project;

	// iterates through the activities and prints them as follows:
	// Activity present project assignment will take place at 09.20.
	// Activity go to TIFF movie will take place at 20.00.
	for (auto a : todo)
		std::cout << a << '\n';

	// Prints the activities as follows:
	// Activity go to TIFF movie will take place at 20.00.
	// Activity present project assignment will take place at 09.20.
	todo.reversePrint(std::cout);
}
```

Determine the result of the execution of the following C++ programs. If there are any errors, indicate the exact place where the errors occur. Justify your answers.

```
class B
{
public:
	B() { std::cout << "B{}"; }
	virtual void print() { std::cout <<
"B"; }
	virtual ~B() { std::cout << "~B()"; }
};

class D : public B
{
public:
	D() { std::cout << "D{}"; }
	void print() override { std::cout <<
"D"; }
	virtual ~D() { std::cout << "~D()"; }
};
```

```
int main()
{
	B* b[] = { new B{}, new D{} };
	b[0]->print();
	b[1]->print();
	delete b[0];
	delete b[1];
	return 0;
}
```

```cpp
class Person
{
public:
	Person() { std::cout << "Person{}"; }
	virtual void print() = 0;
	virtual ~Person() { std::cout <<
"~Person()"; }
};

class Student : public Person
{
public:
	Student() { std::cout << "Student{}";
}
	void print() override { std::cout <<
"Student"; }
	virtual ~Student() { std::cout <<
"~Student()"; }
};
```

```cpp
int main()
{
	Person* p = new Person{};
	delete p;
	Person* s = new Student{};
	s->print();
	delete s;

	return 0;
}
```

```cpp
class E
{
public:
	E() { std::cout << "E{}"; }
	virtual ~E() { std::cout << "~E()"; }
};

class DE : public E
{
public:
	static int n;
	DE() { n++; }
};

int DE::n = 0;



int fct2(int x)
{
	if (x < 0)
	{
		throw E{};
		std::cout << "number less than
0" << std::endl;
	}
	else if (x == 0)
	{
		throw DE{};
		std::cout << "number equal to 0"
<< std::endl;
	}
	return x % 10;
}
```

```cpp
int main()
{
	try
	{
		int res = 0;
		res = fct2(-5);
		std::cout << DE::n;
		res = fct2(0);
		std::cout << DE::n;
		res = fct2(25);
		std::cout << DE::n;
	}
	catch (E&)
	{
		std::cout << DE::n;
	}

	return 0;
}
```

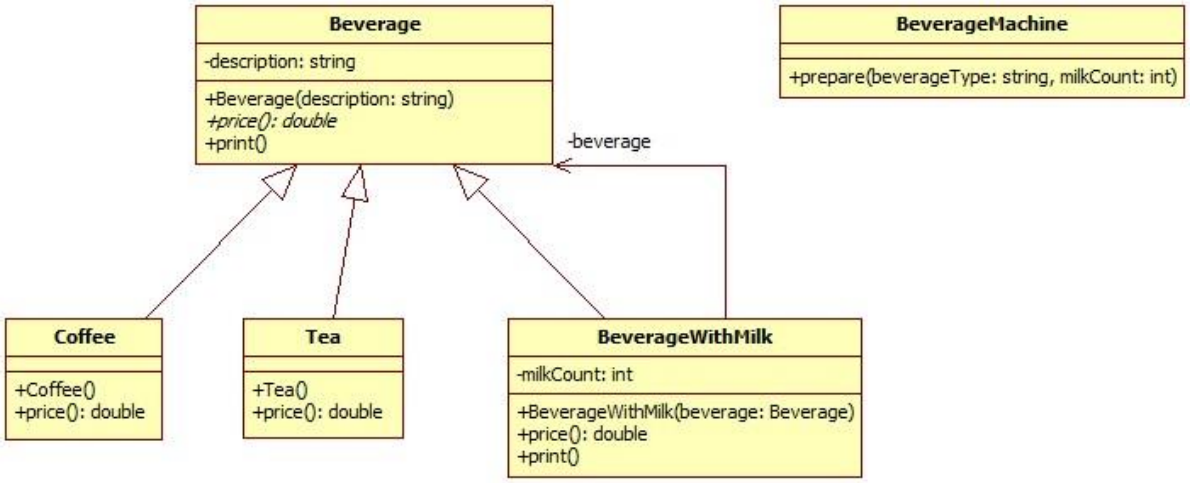
```cpp
#include <deque>
#include <string>
#include <iostream>

int main()
{
	std::deque<std::string> d;
	d.push_back("A");
	d.push_front("B");
	d.push_back("C");
	d.push_front("D");

	auto itBegin = d.begin();
	auto itEnd = d.end();
	itBegin++;
	itEnd--;

	while (itBegin != itEnd)
	{
		std::cout << *itBegin << " ";
		itBegin++;
	}

	return 0;
}
```

5.2. Inheritance, polymorphism, UML, design patterns – C++

Write a program that simulates a beverage machine and corresponds to the UML diagram given below.

- All the beverages (*Beverage*) have a *description* (string) and a *price*, given by the pure virtual function *price()* in the class *Beverage*. The *print()* method in the class Beverage will print at the console the description of the beverage and its price.
- *Coffee* and *Tea* are two concrete beverages which the machine can provide. These two classes will initialize the description by calling the base class constructor with the descriptions “Coffee” and “Tea”, respectively. The prices of these beverages are 2.5 RON for coffee and 1.5 RON for tea.
- The machine allows adding milk to a beverage, thus resulting *BeverageWithMilk*. Each milk costs 0.5 RON. Printing a beverage with milk involves printing the beverage (*beverage.print()*) and then printing the number of portions of milk that were added. The price of a beverage with milk is computed as the price of the beverage, to which the price of all the portions of milk is added.
- When the program starts, a new object of type *BeverageMachine* is created, to simulate buying the following beverages, by calling the method *prepare*, with the given parameters, as follows:
  - “Tea”, 0 – tea, with no milk
  - “Coffee”, 1 – coffee, with milk
  - “Coffee”, 2 – coffee, with double milk
- The *prepare* method will show the prepared beverage, including its price.

- Take memory management into consideration and implement it correctly.



6. PRACTICAL EXAMINATION

Below you will find a problem statement similar to what you can expect to receive during the practical examination. The problem statement will in general follow the requirements set out between Lab 5 and Lab 14, will require a graphical user interface (using Qt) and writing specifications, tests and the implementation of layered architecture.

Observations:

1. Solving the following problem statement completely should be possible for you in a time span of 3 hours.

2. You are encouraged to bring your own laptop to the exam. You are free to use your preferred IDE. Make sure your IDE is set up correctly and it works! Make sure that Qt works!

3. You are allowed to use Qt Designer, if you want to.

4. The problem must be started from an empty workspace. You are allowed to use the following sites for documentation, but nothing else:
- http://doc.qt.io/qt-5/
- http://en.cppreference.com/w/
- http://www.cplusplus.com/

6.1. Problem statement

Write an application for teachers, which simulates shared grading, as follows:

1. The information about the students is stored in a text file, each student having: **an id** (integer), **a name** (string), **a group** (integer), **a grade** (double) and the **name of the teacher** that graded the student (string). The file is manually created and the data about the students is read when the application starts.
2. There is a main teacher, who can add and remove students.
3. Another file contains information about the teachers who can grade students. Each teacher has *a* **name** (string) *(and the groups that he/she has activities with)*. This information is also read from the file when the application starts.
4. When the application is launched, a new window is created for the main teacher, which shows all the students, sorted by their groups and by their names. **(1.5p)**
5. Also, when the application is launched, a new window is created for each teacher who can grade students, having as title the teacher's name. This window will show all the students, sorted by their groups and by their names. (*Each such window will only show the students belonging to the groups that specific teacher has activities with.*) **(1p)**
6. The main teacher can add and remove students. When a student is added, the teacher must input the name and the group. The grade will automatically be set to 1 and the grading teacher to the empty string. **(1p)** When a student is removed, the application must show a confirmation dialog, to

prevent the teacher from deleting students by mistake. If the teacher confirms, the student is deleted. **(1p)**

7. When a modification is made by the main teacher (a student is added or removed), all the other teachers will see the modified list of students in their windows. **(1p)**
8. The teachers can grade students. When a student is graded by a teacher, the main teacher, as well as all the other teachers will see the modifications: the new grade and the name of the teacher that made the modification. **(1p)** If a teacher tries to grade a student that has already been graded (by some other teacher), the operation fails and the teacher is informed with a message. **(0.5p)**
9. When the application is finished, the students' file will be updated. **(0.5p)**

Non-functional requirements

1. Use STL to represent you data structures.
2. Use an object *Student* to represent the necessary data.
3. Use a class *GradingRepository* to manage your teachers and your students.
4. Use a class *GradingController*, which has the role of a controller.
5. Create a custom defined exception class to handle the constraint "If a teacher tries to grade a student that has already been graded (by some other teacher), the operation fails" and use objects of this class. This constraint will be imposed by *the controller*: the function that modifies a grade will have this as a precondition and if it is broken, a custom exception will be thrown.

Observations

1. **1p - of**
2. Specify and test the following functions (repository / controller): **(1.5p)**
    a. Function which adds a student. **(0.5p)**
    b. Function which removes a student. **(0.5p)**
    c. Function which updates the student's grade and grading teacher. **(0.5p)**
3. Use a layered architecture. If you do not use a layered architecture, you will receive 50% of each functionality.
4. If you do not read the data from file, you will receive 50% of functionalities 4, 5 and 6.
5. If you create a console based application, you will receive 40% of each functionality.

6.2. Advice for the practical examination

1. Implement a problem similar to the one in the example (Section **Problem statement**). Time yourself while solving it, make sure you can implement at least some of the functionalities in the allotted time (3 hours). This way you can detect where your difficulties are and you can improve yourself.
2. Build your application incrementally: one step at a time, **compile frequently**.
3. Only add one function in one step, so that you can easily revert to a functional version, in case something doesn't work.
4. **Do not ignore the errors**, solve them before continuing with writing source code.
5. Do not ignore warnings, sometimes these can indicate errors in the program.

6. **Do not implement functionalities that are not required**. By doing this, you might waste valuable time and **the source code will not be graded, only the functional requirements**!
7. If there are issues that you cannot solve, try finding alternative implementations such that you can still test your code.
8. For the practical examination, build a Qt empty project and make sure that it compiles and that you can execute it (+ that you have all the include paths set properly).

7. BIBLIOGRAPHY

1. B. Stroustrup. The C++ Programming Language, Addison Wesley, 1998.
2. Bruce Eckel. Thinking in C++, Prentice Hall, 1995.
3. A. Alexandrescu. Programarea moderna in C++: Programare generica si modele de proiectare aplicate, Editura Teora, 2002.
4. S. Meyers. Effective C++: 55 Specific Ways to Improve Your Programs and Designs (3rd Edition), Addison-Wesley, 2005.
5. S. Meyers. More effective C++: 35 New Ways to Improve Your Programs and Designs, Addison-Wesley, 1995.
6. B. Stroustrup. A Tour of C++, Addison Wesley, 2013.
7. C++ reference (http://en.cppreference.com/w/).
8. Qt Documentation (http://doc.qt.io/qt-5/).
9. E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley Longman Publishing, 1995.