Computer Science Class

Master ENG/ENV/TECH

Lab and Homework 3

2018

General instruction for all lab

See resource Assignment in https://hippocampus.ec-nantes.fr/ "Labs" section of M1_M_ENG_PROGR course for due date and delivery.

Your work consists of sources answering exercises given below. They must be provided as individual text files following the naming convention of your subject. For each exercise a table, called 'requested information table' hereafter, sets precisely what is expected in terms of naming convention, precises order of input data for the programs you create and output format of these programs results (see chapter 3.4 for complete explanations). You are expected to upload an archive containing all your work in resource Assignments of course "Labs" section of Pedagogic server (no individual file will be allowed).

An auto evaluation tool called auto_eval.bash (see chapter 3.1, 3.2, and 3.3 for details) given with this document, has to be used to generate this archive. It can also be used to auto evaluate your archive before submission and check partially the correctness of your work.

The same auto evaluation tool will be used by the teachers to evaluate your work.

Failure to return the sources archive on time on the server will result in a grade of zero for this homework.

Uploading an archive with a wrong structure or name (i.e. archive not readable or not named correctly) will result in a grade of zero for this homework.

Note : Some useful tricks are given in chapter 3.5.

1 A company hierarchy

Here is the declaration of a class hierarchy given in *company.h* reproduced below :

```
1  // Base class
2  class employees
3  {
4   public:
5    void setSalaryAndId(const int i, const double s);
6    double getSalary(void) const;
7    virtual void printEmployee(void) const;
8   private:
9    int ID;
10   double salary;
11 };
12
13 //First derivation
14 class fixed_term : public employees
15 {
16  public:
17   fixed_term(int id,double salary, double period_);
18   double getPeriod(void) const;
19   virtual double totalCost(void) const =0;
20  private:
21   double period;
22 };
23 class permanent : public employees
24 {
25  public:
26   permanent(int id,double salary, double bonus_);
27   virtual double cost(void) const =0;
28  protected:
29   double bonus;
30 };
31
32 //Second derivation
33 class manager : public permanent
34 {
35  public:
36   manager (int id, double salary, double bonus, double extra_bonus_, int
         team_size_);
37   void printEmployee(void) const;
38   double cost(void) const;
39  private:
40   int team_size;
41   double extra_bonus;
42 };
43 class developer : public permanent
44 {
45  public:
46   developer (int id, double salary, double bonus, int
         number_of_active_project_);
47   void printEmployee(void) const;
48   double cost(void) const;
49  private:
50   int number_of_active_project;
51 };
52 class trainee : public fixed_term
53 {
54  public:
```

```
55    trainee (int id, double salary, double period, double imputation_, const
          permanent &in_charge_ );
56    void printEmployee(void) const;
57    double totalCost(void) const;
58   private:
59    double imputation;
60    const permanent &in_charge;
61  };
62  class subcontractor : public fixed_term
63  {
64   public:
65    subcontractor(int id, double salary, double period, double computer_cost_)
          ;
66    void printEmployee(void) const;
67    double totalCost(void) const;
68   private:
69    double computer_cost;
70  };
```

It is a kind of financial database library for an IT development company. This library is more a training tool than a real purpose software. All the employees have an identification number (ID) and a monthly salary. That is what ***employees*** class stores. Two categories are possible in this company :

- Employees with permanent contract. This category has a bonus corresponding to company profits shared. ***permanent*** class represents this category and stores the bonus per month.
- The second category is related to the employees having a kind of fixed-term contract. For them, they only work in the company for a determined period. ***fixed_term*** class represents this second category and stores the period in months for which they are scheduled to stay in the company.

Finally, both categories have subcategories. Managers (who have extra monthly bonus related to the size of the team they manage) stored in ***manager***, software developers (who work on a certain amount of projects) stored in ***developper***, subcontractors (who rent the company computers and this rental has a monthly price) stored in ***subcontractor*** and students in trainee period (who have an extra cost corresponding to some percentage of their follower cost in the company) stored in ***trainee***.

Write the implementation of this hierarchy in a *company.cc* file. **The given *company.h* must no be changed**. Here are the expected features:

- Obviously the method *setSalaryAndId* sets data member *ID* and *salary* for a ***employees*** instance.
- Method *getSalary* returns data member *salary* for a ***employees*** instance.
- *fixed_term* constructor uses its arguments to set *ID*,*salary* and *period* data members.
- *permanent* constructor uses its arguments to set *ID*,*salary* and *bonus* data members.
- *manager* constructor uses its arguments to set *ID*, *salary*, *bonus*, *extra_bonus* and *team_size* data members.
- *developer* constructor uses its arguments to set *ID*, *salary*, *bonus* and *number_of_active_project* data members.
- *subcontractor* constructor uses its arguments to set *ID*, *salary*, *period* and *computer_cost* data members.

- *trainee* constructor uses its arguments to set *ID*, *salary*, *period*, *imputation* and *in_charge* data members.
- *getPeriod* method returns the *period* member data of ***fixed_term*** hierarchy.
- *cost* method returns the computed cost per month of the employees in permanent category. This must sum the salary and the bonus. And for a manager it must add the product of the extra bonus times the size of the team.
- *totalCost* method returns the computed cost for all their period of the employees in fixed-term category. This must multiply the salary and extra cost by the period. For subcontractors the extra cost is just the rental computer cost. For trainees the extra cost corresponds to a percentage (stored in *imputation*) of what costs the employee who is in charge of the student. This is to consider that a person in charge passes a certain amount of their time to help the student and this has to be added to what costs really the student to the company. It is transcripted into a percentage of the permanent employee cost.
- *printEmployee* method prints all the member data on a line. Follow the example given below.

An extra condition is imposed to *printEmployee* method of class ***employees***. **It should not be empty and it must be used in your implementation (i.e. it must participate to the printing task).**

Test *company.cc* with the program in *test_company.cc* which should without any modification give the following output for the specified entered data :

```
Enter id, salary, bonus, extra bonus and managed team size for Alice manager
2 5000.  400.  50.  7
Enter id, salary, bonus, and active project in charge for Igor developer
1 3000.  500.  3
Enter id, salary, bonus, and active project in charge for Kim developer
8 2500.  100.  1
Enter id, salary, bonus, and active project in charge for Aalap developer
3 2700.  200.  2
Enter id, salary, contract period, and charge for Ling subcontractor
100 2000.  6.  23.
Enter id, salary, contract period, and imputation ration for John student
200 1800.  3.  0.1
OUT01 :ID 2, Salary 5000, Bonus 400, Extra bonus 50, Team size 7
OUT02 :ID 1, Salary 3000, Bonus 500, Project number 3
OUT03 :ID 8, Salary 2500, Bonus 100, Project number 1
OUT04 :ID 3, Salary 2700, Bonus 200, Project number 2
OUT05 :ID 100, Salary 2000, Period 6, Computer cost 23
OUT06 :ID 200, Salary 1800, Period 3, Imputation 0.1
```

```
finished
```

In this question the given *test_company.cc* is in fact with many comments. auto_eval.bash is tuned for this configuration. Your first task is to validate your implementation against this restricted set of instructions. And you can only implement what is needed at this stage.

When it will be done you will try to uncomment as many instructions as you can (and finish the implementation if needed). The idea is that you understand inheritance, polymorphism, virtual methods, abstract classes, ...etc so that you quickly uncomment what is relevant. It is a self training exercise with the compiler playing the role of the teacher somehow.

In fact the teachers will use this special uncommented version of *test_company.cc* with the auto evaluation tool. So your auto_eval.bash will not test those extra outputs. It will be of your own responsibility to make it work properly.

To avoid any misinterpretation of *cost* and *totalCost* computation here are the expected results with the example above :

- For Alice computation give 5750
- For Igor computation give 3500
- For Ling computation give 12138
- For John computation give 6450

A last point not seen in detail in course. If a base class ***A*** has a method *FA* then any derived class may explicitly call this method using this syntaxe : " ... A::FA(..."

	Requested information
Files	company.cc
Input	
Output	

Scaling : 1/3

2 Define and Implement a class to represent polynomials

2.1 Polynomial operators

This work will use components of your previous laboratories works (Lab2 in particular). Take the correction **without modifying anything** (to avoid loosing time on this part and to make sure that you are all using the same interface).

You must create a class called ***Poly*** to store and compute polynomials of any order, using real values (represented by "double") as parameters, with real coefficients. Your class should have a constructor which takes a pointer to double (the coefficients) and an order as arguments to instantiate an object.

Your class should deliver correct instances in any circumstances (construction, operator return, ...etc). By correct instance, we mean an object representing a polynomial where the coefficient of higher order is not null. This aspect for constructor must become, if not respected, an error which will be treated by throwing an exception. Use following instruction "throw −1;" and take a look on "Lab2 Error treatement chapter " for further information.

Your class **must** represent a polynomial with **only one private member** variable (conceptually, polynomial member variable must be seen as something which stores coefficients and is able to give the polynomial order). Use of derivation is **prohibited**

Your class should have a "const" method to evaluate a polynomial at a given point. You will overload parenthesis operator so that for a given instance P of class ***Poly***, *P(x)* returns the evaluation of the polynomial represented by P at point x.

Your class should have a "const" method *print* to print on screen a polynomial (**please follow example below for outputs format of this method**). You should have "const" operations to add (+), subtract(-), and multiply(*) two polynomials. These methods should return an instance of the class ***Poly***.

An assignment operator (=) for class ***Poly*** is impossible to implement without modification of Lab2 work and that is not expected. You must then implement a fake version of this operator just to stop the program (using a "throw −1;") if someone tries to use it.

As usual, create your class ***Poly*** in *Poly.h*, and *Poly.cc* files (where respectively you declare and implement class ***Poly***), and test it with the main program given by file *test_poly_operators.cc*. With the data given below, you should get the following output :

```
Input order of p1 : n = 2
Input coeffient of p1 (from order 0 to order n) :
0
0
0.5
Input order of p2 : n = 3
Input coeffient of p2 (from order 0 to order n) :
1
0
4
2
Input order of p3 : n = 5
Input coeffient of p3 (from order 0 to order n) :
8
6
0
0
2
1
Testing constructor and output
p1 : poly(x)=+0.5*x^2
p2 : poly(x)=1+4*x^2+2*x^3
p3 : poly(x)=8+6*x^1+2*x^4+1*x^5
Testing addition
p2+p3 : poly(x)=9+6*x^1+4*x^2+2*x^3+2*x^4+1*x^5
p3+p2 : poly(x)=9+6*x^1+4*x^2+2*x^3+2*x^4+1*x^5
Testing substraction
p2-p3 : poly(x)=-7-6*x^1+4*x^2+2*x^3-2*x^4-1*x^5
p3-p2 : poly(x)=7+6*x^1-4*x^2-2*x^3+2*x^4+1*x^5
p3-p3=0 : poly(x)=0
Testing multiplication
p2*p3 : poly(x)=8+6*x^1+32*x^2+40*x^3+14*x^4+1*x^5+8*x^6+8*x^7+2*x^8
p3*p2 : poly(x)=8+6*x^1+32*x^2+40*x^3+14*x^4+1*x^5+8*x^6+8*x^7+2*x^8
Testing complicate operation
p3-p1*p2 : poly(x)=8+6*x^1-0.5*x^2
(p3-p1*p2)*p1 : poly(x)=+4*x^2+3*x^3-0.25*x^4
Testing () operator
First try, input a value for x1 :
1.5
p2(x1) : 16.75
p3(x1) : 34.7188
Second try, input a value for x2 :
1
p2(x2) : 7
p3(x2) : 17
Testing exceptions from problematic cases
```

(maybe some messages related to your implementation)

```
Error1 : catching exception for null order coef
```

(maybe some messages related to your implementation)

```
Error2 : catching exception for = usage
```

2.2 Euclidian division

For the next part of the exercise, you will have to implement the euclidian division of two polynomials.

Having A and B, two polynomials, A divided by B gives mathematically 2 polynomials Q (the quotient) and R (the remainder of the division) so that :

$A=B*Q+R$

Your class must, having those two instances A and B of class ***Poly***, give :

- Q the quotient by calling "A.euclidienDivisionQ(B)"
- R the remainder by calling "A.euclidienDivisionR(B)" or "A.euclidienDivisionR(B,A.euclidienDivisionQ(B))"

The second version uses the result of a first computation of the quotient to compute the remainder.

Add *euclidienDivisionQ* and *euclidienDivisionR* (the two versions) "const" method in the same *Poly.h* and *Poly.cc* files that you created in part 2.1.

Test you implementation of euclidian division with the program given in *test_poly_eucl_div.cc*. With the given data below, you should get the following result :

```
Input order of p1 : n = 3
Input coeffient of p1 (from order 0 to order n) :
1
0
4
2
Input order of p2 : n = 5
Input coeffient of p2 (from order 0 to order n) :
8
6
0
0
2
1
Testing euclidian divison, quotient part
Q=(p2*p1)/p1=p2 : poly(x)=8+6*x^1+2*x^4+1*x^5
Q=p2/p1 : poly(x)=+0.5*x^2
Testing euclidian divison, residual part
R=(p2*p1)-Q((p2*p1)/p1)*p1=0 : poly(x)=0
Rdirect=(p2*p1)-Q((p2*p1)/p1)*p1=0 : poly(x)=0
R=p2-Q(p2/p1)*p1 : poly(x)=8+6*x^1-0.5*x^2
Rdirect=p2-Q(p2/p1)*p1 : poly(x)=8+6*x^1-0.5*x^2
```

So at least after completing part 2.1 and part 2.2, you should have 10 member functions (including constructor and operators) and no more than one private member variable.

	Requested information
Files	Poly.cc, Poly.h
Input	
Output	

Scaling : 2/3

3 Appendix

3.1 Auto_eval.bash limits

This tool is just a help to verify mainly that:

- Your program is located in correct source files having correct names.
- No file is missing. If so you will be able to pass through all steps but your points for the missing file will be lost.
- All programs, at least, compile and run correctly (to obtain the 2 points see 3.2).
- With the student parameter setting, it gives correct results (to obtain the 6 points see 3.2).
- The archive given to teacher has the correct structure and name.

What this tool does not verify is that:

- Your programs are correct in absolute. Be careful auto_eval.bash is tuned to have exercises running with a set of parameters but a different setting will be used by the teacher to correct your work. This means that your code may work with auto_eval.bash and the student parameter setting but not with the teacher parameter setting. It is your responsibility to ensure that your program is correct and works with any setting.
- Your programs are correct if you just generate an archive and didn't pass through evaluation step.
- You give your correct group identification number. If you generated an archive following a wrong group ID and upload it as it is then you will have a grade of zero for this work.

3.2 Evaluation rule

The auto evaluation tool auto_eval.bash will be used by teachers (in a slightly modified version) to evaluate your work with the following rule:

- For an exercise the mandatory program doesn't compile => 0 points for the exercise.
- For an exercise the mandatory program compiles but doesn't execute correctly (i.e. crash,stop in the middle ...) => 1 point.
- For an exercise the mandatory program compiles and executes without bug but doesn't give correct results => 2 points.
- For an exercise the mandatory program compiles, executes without bug and gives correct results/behavior => 6 points.

Naturally when you are in position of having only 1 or 2 points, the program will be inspected to see if it is relevant (answering the question). If not (something that has nothing to do with the exercise but compiles and runs correctly) => 0 points.

For every exercise an extra scaling is done. After each requested information table (see 3.4) you will have the coefficient used for this scaling.

By using auto_eval.bash you will be able to estimate how many source evaluation points you will have for this lab without scaling.

3.3 Getting and using auto_eval.bash

auto_eval.bash like this pdf files comes from self extracting file labX.bash that you download from resource "Lx subject" in https://hippocampus.ec-nantes.fr "Labs" section of M1_M-ENG_PROGR course, where X corresponds to the current lab number.

Place auto_eval.bash in the folder where you have all exercises sources that have to be given to teachers. Then in this folder just type:

```
··> ./auto_eval.bash
```

The first step is to input your group ID (letter A, B or C and a number):

```
../auto_eval.sh

================================================================================
Setting your lab group
================================================================================

Please what is your group letter ?
Type your choice ( A B C ) :A


Please what is your group number ?
Type your choice ( 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 ) :01
```

Input your group letter and number

If you launch the script for the first time, you have to create an archive of your work. Select "Prepare an archive of your work".

```
Please what is your group number ?
Type your choice ( 01 02 03 04 05 06 07 08 09 10 11 12 13 14 15 16 17 18 19 20 ) :01



//////////////////////////////////////////////////////
///   This is auto_eval version 0.1 for LAB 1 2014
//////////////////////////////////////////////////////
/// Current status
///   No archive file ./archive_lab1_A_01.tar.gz present
//////////////////////////////////////////////////////

What do you want to do ?

    a) prepare an archive of your work.
    q) stop using this program

Type your choice ( a q ) :a



================================================================================
Archiving file for lab 1
================================================================================

File 'bary_triangle.cc' correctly archived
File 'mathfunction.h' correctly archived
File 'mathfunction.cc' correctly archived
File 'test_mathfunction.cc' correctly archived
File 'Poly.h' correctly archived
File 'Vector.h' correctly archived
File 'ex1.cc' correctly archived
File 'Poly.cc' correctly archived
File 'Vector.cc' correctly archived
Compressing archive

================================================================================
Archive ./archive_lab1_A_01.tar.gz ready to be auto evaluate or uploaded on the server.
================================================================================
```

Create archive

A *.gz* archive file is created, and ready to be uploaded on the pedagogic server. If you want, at this point you can stop the program (Type "q"). Note that theses two steps at least are mandatory to create the archive of your work that you will upload on the server. You can create this archive by yourself (for instance using the *tar* command), but if there is any kind of problem when we try to extract it, your grade may be affected with no protestation possible (the most stupid problem

would be that your archive doesn't have the correct name which leads to ... zero for the whole lab evaluation) .

Otherwise, you can use this tool to get an auto-evaluation of your work. Now that your archive is created, you can notice that a new command appeared, "evaluate archive already generated". Type "b".

```
////////////////////////////////////////////////////////
///   This is auto_eval version 0.1 for LAB 1 2014
////////////////////////////////////////////////////////
/// Current status
/// Archive file ./archive_lab1_A_01.tar.gz present
////////////////////////////////////////////////////////

What do you want to do ?

    a) prepare an archive of your work.
    b) evaluate archive already generated.
    q) stop using this program

Type your choice ( a b q ) :b
```

Auto-evaluate your work

Some information are displayed on the screen, saying that the auto-evaluation worked through well (or not). The most interesting information for you are the ones on the next screen-shot.

```
================================================================================
Compile archive ./archive_lab1_A_01.tar.gz
================================================================================

Program bary_triangle compile correctly
test_mathfunction.cc compile correctly
mathfunction.cc compile correctly
Program test_mathfunction compile correctly
ex1.cc compile correctly
Poly.cc compile correctly
Vector.cc compile correctly
Program ex1 compile correctly

================================================================================
Run archive ./archive_lab1_A_01.tar.gz
================================================================================

Program bary_triangle run correctly with bary_triangle_00.dat
Program test_mathfunction run correctly with test_mathfunction_00.dat
Program ex1 run correctly with ex1_00.dat

================================================================================
Evaluate result of archive ./archive_lab1_A_01.tar.gz
================================================================================

=> For input parameter setting, program 'bary_triangle' give 3 out of 3 correct result(s).
=> For input parameter setting, program 'test_mathfunction' give 2 out of 2 correct result(s).
For ex1_00.dat input parameter setting, program 'ex1' don't give correct result for p2 result.
=> For input parameter setting, program 'ex1' doesn't give any correct result from 1 expected.

////////////////////////////////////////////////////////
///   This is auto_eval version 0.1 for LAB 1 2014
////////////////////////////////////////////////////////
/// Current status
/// Archive file ./archive_lab1_A_01.tar.gz present
/// Archive unpacked successfully
/// 3 out of 3 program compile successfully
/// 3 out of 3 program run successfully
/// 5.0 out of 6 correct results where obtain
////////////////////////////////////////////////////////
```

Results of your auto-evaluation

Here you can see:

- Which of your source codes compile correctly.
- Which of your programs run correctly.
- Which of your programs give the expected results.

This way you can have an estimation of your grade according to the rules detailed in section 3.2. However keep in mind that the grade estimated using this tool may not be your final grade (see section 3.1). Note that you can use this command only if you have generated an archive first.

You could see that in the previous example, one of the results given by program "ex1" is incorrect. To display the difference between your results and the expected one, select the new command "display difference from solution of last generated archive" by typing "c".

```
=> For input parameter setting, program 'bary_triangle' give 3 out of 3 correct result(s).
=> For input parameter setting, program 'test_mathfunction' give 2 out of 2 correct result(s).
For ex1_00.dat input parameter setting, program 'ex1' don't give correct result for p2 result.
=> For input parameter setting, program 'ex1' doesn't give any correct result from 1 expected.

////////////////////////////////////////////////////
///   This is auto_eval version 0.1 for LAB 1 2014
////////////////////////////////////////////////////
/// Current status
/// Archive file ./archive_lab1_A_01.tar.gz present
/// Archive unpacked successfully
/// 3 out of 3 program compile successfully
/// 3 out of 3 program run successfully
/// 5.0 out of 6 correct results where obtain
////////////////////////////////////////////////////

What do you want to do ?

    a) prepare an archive of your work.
    b) evaluate archive already generated.
    c) display difference from solution of last evaluated archive
    q) stop using this program

Type your choice ( a b c q ) :c



================================================================================
Display difference from last evaluation of archive ./archive_lab1_A_01.tar.gz
================================================================================

ex1 00 p2 :
yours    >  poly(x)= +0.5*x^2
solution >  poly(x)=  0.5*x^2
```

Compare obtained results and expected results

Now you can:

- Generate a new archive of your work, if you modified some of your source files.
- Auto-evaluate the current archive.
- Compare the results obtained with your program and the solution.
- Stop using the program.

In the last case, if you decide to quit the program, you will be asked whether you want to clean or save a certain temporary folder. This temporary folder was generated during the auto-evaluation step, and can possibly be used for debugging even if the script is not launched. Those of you who are the most at ease with programming may want to use it, otherwise you can just delete it.

```
//////////////////////////////////////////////////////
///   This is auto_eval version 0.1 for LAB 1 2014
//////////////////////////////////////////////////////
/// Current status
/// Archive file ./archive_lab1_A_01.tar.gz present
/// Archive unpacked successfully
/// 3 out of 3 program compile successfully
/// 3 out of 3 program run successfully
/// 5.0 out of 6 correct results where obtain
//////////////////////////////////////////////////////

What do you want to do ?

    a) prepare an archive of your work.
    b) evaluate archive already generated.
    c) display difference from solution of last evaluated archive
    q) stop using this program

Type your choice ( a b c q ) :q



================================================================================
Warning : Cleaning
================================================================================

Do you want to clean (c) or save (s) temporary folder /home/users/struct/ble/Enseignement/CPP/AutoEval/lab1/lab1/auto
_eval_dir22261 ?
This folder may contain compilation, execution and evaluation of last evaluated archive.
If you save it, it's your responsability to do the cleaning

Type your choice ( c s ) :s
```

Stop the auto-evaluation program

3.4 Requested information

Tables given at the end of each exercise entitled 'Requested information' contains the following items:

- Files: this is the list of files which must be present in your archive uploaded in the server. auto_eval.bash harvests these files to generate the mandatory archive.
- Input: describes for each program the precise order in which information must be entered.
- Output: describes for each program the output format of the results. **Results (that teachers want to check easily) must be presented by your program in a rather rigid format described by the following**:
  - A result is present on one single line. Nothing else may be written on that line.
  - A result is first identified by a token which starts the line.
  - After the token a ':' must separate the token from result value.
  - After ':' result value must be written (it could be anything).
  - token, ':' and result value must be separated at least by one blank character.

  In this section you have the list of mandatory results given by token names and in parentheses by result name coming from subject.

Here is an example of a '*Lame*' program made of one file *lame.cc*. It is generating 2 output results called λ and μ in the subject. From the subject, this program should ask for 2 input values called E and ν to compute results. The requested information table would look like:

	Requested information
Files	lam.cc
Input	ν then E
Output	MU(μ), LAMBDA(λ)

Scaling : 1/8

'*Lame*' program execution would look like with $E = 2500.$, $\nu = 0.28$ and computed $\mu = 976.5625$ and $\lambda = 1242.8977$:

```
blablabla
please enter nu :
0.28
blabla
please enter E :
2500.
 LAMBDA : 1242.8977
blabla
 MU : 976.5625
blabla
```

Notice in this example that ν is asked before E as mandatory by "Input" directive. And λ and μ appear in undefined order but with mandatory format (i.e token LAMBDA and MU).

Very important. You should avoid at any cost the use of token for anything else than the result it corresponds to. Consequences may result in a zero grade just because you messed up the tools with false information.

In example above if in any 'blabla' you write MU or LAMBDA you will get a zero for associated results.

Last but not least when you enter information and output first a question message you must always add a new line to your question if it precedes a result. Let's take the example of '*Lame*' above which outputs token 'LAMBDA' right after asking to input E. If you don't follow this last recommendation you may ask for E without inserting a new line after the question as you can see in the example below:

Your executable is Lame and on terminal you type:

```
··> ./Lame
```

And obtain the following bunch of outputs:

```
blabla
please enter E : 2500.
LAMBDA : 1242.8977
blabla
```

This works well on the terminal by typing on the keyboard '2500.' and 'enter'. But auto_eval.bash works differently. It uses what is called redirection mechanisms. **And in fact this leads auto_eval.bash to miss recognizing your results.**

In this example you must add a "end of line" after the question to obtain:

```
··> ./Lame
```

```
blabla
please enter E :
2500.
LAMBDA : 1242.8977
blabla
```

and auto_eval.bash will work correctly in this case.

3.5 Useful

During Labs you will have questions where testing programs ask for many inputs. During debugging stage you may then be obliged to re enter those inputs manually many times. That may be error prone and time consuming. To simplify your work use redirection.

For example lets take a '*prg*' program which runs the following:

```
··> prg
```

```
Enter number :
23
```

```
Enter number :
13
Enter number :
-1
Mean value :
18
```

Here for illustration we just limit inputs to three entries. Put all those inputs in a file, with one input per line. Let us call this file "ip.txt". Its contents for this example will be:

```
23
13
-1
```

Now if you type the following command you get the same execution of '*prg*' without typing anything:

```
···> prg < ip.txt
```

```
Enter number :
Enter number :
Enter number :
Mean value :
18
```

Note that inputs just vanish from terminal display.