Linköping Institute of Technology
Campus Norrköping
Department of Science and Technology/ITN
Aida Nordman

Exam for TNG033

April 8th, 2013, 08.00 am - 12.00 pm

Aid: Any book, solutions for exercises, and access to web pages. However, the **exam is individual**. Thus, you are not allowed to use e-mail. If you copy any code from a web page then you must explicitly indicate that fact and write the web page address in a comment line preceding the copied code. TNG033 course webpage requires a **`login = TNG033`** and a **`password = TNG033ht13`**.

Files needed: Files needed for this exam are available in the folder **`D:\TENTA\TNG033`**.

Instructions: Use a computer in the lab room to log in with your username and password. Then, start **`Code::Blocks`**. **Private laptops are not allowed in the exam**.

You must **write your name and personal number in the beginning of every file** (i.e. **`.cpp`**, **`.h`**), in a comment line.

Do not turn off the computer any time. Only log out when you have finished the exam.

To simplify the marking process, for each exercise, all code is given in the same file. Thus, **deliver one source file for each exercise**.

You are advised to keep a copy of your exam in an USB memory.

Delivering your solutions: In the folder **`D:\TENTA\TNG033`** there is a sub-folder for each exercise, called **`Exerc1`**, **`Exerc2`**, *osv*.

Save your source code files (e.g. **`.cpp`**, **`.h`**) for exercise **`1`** in the folder **`D:\TENTA\TNG033\Exerc1`**. For exercise **`2`**, you save the source files in the folder **`D:\TENTA\TNG033\Exerc2`**, *osv*.

Use your LiU student e-mail address and rename the folder **`TNG033`** in **`D:\TENTA`** with it. For example, if your LiU e-mail address is **`magsi007`** then the folder name **`D:\TENTA\TNG033`** should become **`D:\TENTA\magsi007`**.

Results: Results are e-mailed automatically to every student from LADOK.

Good luck!

Information about this exam

This exam has three parts. The table below summarizes the number of points awarded, difficulty level and grade level aimed, by each of these parts. Thus, you may use this information to plan how to use your time during the exam and which exercises you want to focus on.

	Number of Points	***Difficulty Level***	***Aim***
Part I	60 p	Low	Grade 3
Part II	20 p	Medium	Grade 4
Part III	20 p	Medium++	Grade 5

It is important that you have in mind the following.

- To be approved in the exam, you must do the exercises in ***Part I*** and get at least 50 points in this part (including duggor points, if any).
- In addition to ***Part I***, you can do the exercises in ***Part II*** or ***Part III***. ***Part II*** is graded, if ***Part III*** is not enough to award grade 5 to your exam.
- ***Part III*** exercises only award points, if the programs run and produce the correct output for the given test files.
- You can read in the course web page **`http://www.itn.liu.se/~aidvi/courses/12/TNG033.htm`** detailed information about how the final course grade is decided.

Note that if you find any open questions in some of the presented exercises then feel free to make your own decisions. However, your decisions should be reasonable and must not contradict any of the conditions explicitly written for the exercise. Please, write comments in the programs that clarify your assumptions/decisions, if any.

To get full points in an exercise, the following conditions must be satisfied.

- Readable and well indented code.
- No global variables can be used, except when explicitly stated otherwise.
- Respect for the submission instructions.
- Code copied from a web page should be clearly indicated through a commented line indicating the web page address used as source.

Part I

Exercise 1 [20 p]

Write a program that reads and stores 100 user given integers in a 10×10 matrix. The matrix should be filled by lines and you may assume the user input is correct. The program displays then a message saying whether the matrix is symmetric.

Your program should

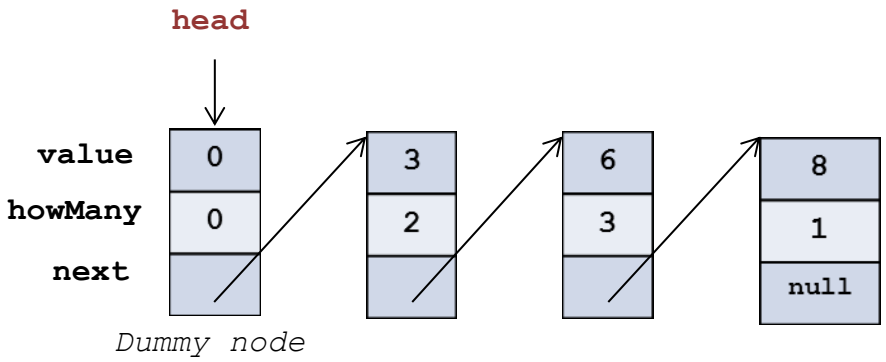
- define a new data type named **`MATRIX`** representing a 10×10 matrix of integers;
- have a Boolean function to test whether a given **`MATRIX`** is symmetric, i.e. the function should have an argument of type **`MATRIX`**.

Write your program in the file **`ex1.cpp`** and you can test it with the (symmetric) matrix given in the file **`matrix.txt`**.

Do not use **`STL`** to solve this exercise.

Exercise 2 [40 p]

In the file **`ex2.cpp`**, you can find the definition of a (simple) class **`List`** to represent a singly linked list of **sorted** integer values. It is possible that a value is stored more than once in a list. To this end, each node stores an integer value and the number of times the value is repeated in the list (**`howMany`**). Consequently, a list has one node for each different value stored in the list. For instance, the list **`L = 3 → 3 → 6 → 6 → 6 → 8`** is implemented as the figure below shows. Notice that a dummy node is used in the list implementation.



The implementation of class **`Node`** is given in the file **`ex2.cpp`**, but you may add any class friends, if needed.

For class **`List`**, the only (member) functions available are a default constructor to create an empty list, a destructor, and a copy constructor.

You should write your code in the file **`ex2.cpp`** that also contains a **`main()`** function with some tests. The **`main()`** function cannot be changed. The expected output can be found in the file **`ex2_out.txt`**.

Add to the class **`List`** the functionality described below.

a. Another constructor that given a value **`k`** and an integer **`n`** creates a list storing value **`k`** repeated **`n`** times. Notice that **`n`** should be set to one by default. For instance, the declarations

```
List L3(8);
```

```
List L4(3,2);
```

create the lists **L3 = 8** and **L4 = 3 → 3**, respectively.

Your program should now be ready to pass **Phase 0** of the tests in the **main()**.

b. An **operator<<**. For instance, **cout << L4;** should display **3 → 3**.

Your program should now be ready to pass **Phase 1** of the tests in the **main()**.

c. An **operator+** (one or several functions might be needed) such that

- **L1+L2** returns a new (sorted) list storing all values occurring in any of the lists **L1** or **L2**. For instance, if **L1 = 1 → 4 → 4 → 7** and **L2 = 4 → 8** then **L1+L2** should return the list **1 → 4 → 4 → 4 → 7 → 8.**
- **k+L** returns a new sorted list with the integer constant **k** added to list **L**. For instance, if **L = 1 → 4 → 4 → 7** then **7+L** should return the list **1 → 4 → 4 → 7 → 7** and **2+L** should return the list **1 → 2 → 4 → 4 → 7**.
- **L+k** returns a new sorted list with the integer constant **k** added to list **L**. For instance, if **L = 1 → 4 → 4 → 7** then **L+7** should return the list **1 → 4 → 4 → 7 → 7** and **L+2** should return the list **1 → 2 → 4 → 4 → 7**.

Your program should now be ready to pass **Phase 2** of the tests in the **main()**.

Part II

Exercise 3a [10p]

Write a program that reads a text file and codifies the text in the file in morse code. The input text file contains English words, punctuation signs, brackets, and numbers. Since only characters of the English alphabet, '**a**' to '**z**' and ''**A**' to '**Z**', can be translated to morse code, anything else (like punctuation signs, digits, and brackets) should be simply discarded by the program. For instance, the text file **data.txt** contains a sample text and "**35%**", occurring the file, should be ignored by the program. Note that the input text may have words consisting of a mix of uppercase and lowercase letters.

As usual, each character of the English alphabet is encoded in morse code as a sequence of "**.**" and "**-**" and morse codes are the same for lowercase as for uppercase letters, e.g. both "**y**" and "**Y**" become "**-.--**". When a word is codified, there should be a white space between each character in morse code, as the example below shows.

```
You          -.-- --- ..-
```

More specifically, the program should perform the following sequence of actions.

- Request to the user the name of the input text file and open it. If opening the file does not succeed then the program should display an error message and terminate.
- Load the text file into a vector of words.
- Display a message, if the text file is empty. Otherwise, display the number of words read into the vector.
- Translate every possible word in the vector to morse code. The codified words should be written into a text file. This file should contain one word in morse code per line. Do not forget that there should be a white space between each character in morse code, as in the example above.

Use **STL** to solve this exercise. To get full points, it is also required to use standard algorithms, instead of hand-written loops (such as **for**-loop, **while**-loop, or **do**-loop), whenever possible.

Add the code to the file **Ex3a.cpp**. This file already contains the declaration of an array of constant strings, named **morseCode**, that holds the morse code for each character of the English alphabet. Thus, **morseCode[0]** stores the morse code for '**a**', **morseCode[1]** stores the morse code for '**b**', *osv*.

Test your program with the input file **data.txt**. File **ex3a_out.txt** contains the corresponding expected output of the program.

Exercise 3b [10p]

Write another program that reads the text file, with the words in morse code, created in the previous exercise and decodes it back to English words. Note that there is one word in morse code per line. Moreover, it may not be possible to recover completely the original text because digits, punctuation signs, and brackets have been ignored during the encoding process of the previous exercise.

Your program should use a container of type **`map`** to help to translate morse code to English. This container may be declared as a global variable of the program. The first step in the **`main`** function should then be to initialize the map container.

Use **`STL`** to solve this exercise. To get full points, it is also required to use standard algorithms, instead of hand-written loops (such as **`for`**-loop, **`while`**-loop, or **`do`**-loop), whenever possible.

Add the code to the file **`Ex3b.cpp`**. Test your program with the input file **`ex3a_out.txt`**. File **`ex3b_out.txt`** contains the corresponding expected output of the program.

Part III

Exercise 4 [20p]

Design a class hierarchy for representing employees of an IT-company. The hierarchy consists of classes **`Employee`**, **`Manager`**, and **`Programmer`**.

`Employee` shall be the base class of all classes in the hierarchy. This class should store the employee's name (**`string`**) and id (**`int`**). When a new **`Employee`** is created, it should be possible to define his/her name and id, and thereafter it should not be possible to change this information (i.e name and id). Assume that each employee's id is unique. Moreover, the class should have the following member functions.

- A function **`salary()`** that returns the salary of an employee.
- A function **`get_id()`** that returns the id of an employee.
- An **`operator<<`** to display the information about an employee. For all employees, name, id, salary, and position (e.g. programmer or manager) are displayed first. Extra information may then be displayed depending on the specific type of employee.

`Manager` is a direct subclass of **`Employee`**. Managers have a base salary and a bonus as percentage of the base salary. Thus, a manager with a bonus of 10% has a salary equal to base salary plus 10% of the base salary. Moreover, each manager has a group of employees that he/she is responsible for. Each instance of class **`Manage`**r should therefore keep a list of the employees that belong to the manager's group (possibly, other managers or programmers). **`Manager`** class should have the following member functions.

- A function **`manage`** that adds a new employee to the group of the manager.
- A function **`unmanage`** that removes an employee from the group of the manager.
- A function **`group_size()`** that returns the number of persons in the group managed by the manager.
- When displaying the information of a manager with **`operator<<`**, the id of all employees in the manager's group should be displayed, after displaying the manager's name, id, salary, and position.

`Programmer` is a direct subclass of **`Employee`**. A programmer's salary depends on how much the programmer is paid per hour and different programmers may have a different price per hour. Thus, each instance of this class should store the number of working hours and the price for each hour. The salary is obviously the number of working hours multiplied by the price per hour. Moreover, every programmer is specialized in a certain programming language. When a new programmer is created, it should be possible to define the programming language the programmer is specialized in, the number of working hours, and the price to be paid per hour. When displaying the information of a programmer with **`operator<<`**, the programming language in which the programmer is specialized should be displayed after name, id, salary, and position.

Note that your class hierarchy should also satisfy the following conditions.

- It should not be possible to create **`Employee`** objects that are not instances of one of the **`Employee`** sub-classes (such as **`Manager`** or **`Programmer`**).

- It should be possible to keep a list of all existing employees. This list can only be updated by member functions of the classes forming the hierarchy. Client code, i.e. application programs, should not be able to directly manipulate the employees list.
- There should be a member function **`list_all_employees()`** that displays the information about all existing employees, **sorted increasingly by employees id**.
- There should be a member function **`number_of_employees()`** that returns the number of existing employees.
- There should be a function **`get_employee(int n)`** that returns a pointer to the employee with id number **`n`**.

Design your classes with care and keep with the given specifications.

Add your code to the file **`ex4.cpp`** that also contains a **`main()`** function to test your code. In the indicated points, add to the **`main`** function the code needed to display

- the number of existing employees,
- a list of all existing employees, sorted increasingly by employees id,
- the salary of an employee, whose id is given by the user. If there is no employee with the given id then an error message should be displayed.

The **`main()`** function cannot be changed, otherwise. The expected output is available in the file **`ex4_out.txt`.**

A printed copy of the files **`ex4.cpp`** and **`ex4_out.txt`** is given with this exam.