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

Exam for

TNG033: Programming in C++

August 15th, 2012, 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. The examiner will drop in the examination room several times during the exam.

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**.

Delivering your solutions: In the folder **`D:\TENTA\TNG033`** create 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.

Tentavisning occasions will be announced by e-mail and in the course web page.

Good luck!

Exercise 1 [35 p]

In the file **`ex1.cpp`**, you can find the definition of a (simple) class **`List`** to represent a singly linked list such that each node stores an integer. Notice that the list may contain repeated values and it may not be sorted. The only (member) functions available are a destructor and an **`operator<<`**. No dummy nodes are used in the implementation of the lists. Add to the class the functionality described below.

a. A constructor **`List(int A[], int n)`** that creates a new list storing the values given in array **`A`**, where **`n`** is the number of integers in **`A`**. If **`n`** is zero then the constructor should create an empty list. The list should store the values in the same order as given in the array **`A`**, i.e. **`A[0]`** is stored in the first node, **`A[1]`** is stored in the second node, *osv*.

b. A boolean function **`repeated_member`** that returns **`true`** if a given integer **`i`** occurs multiple times in the list. If the integer **`i`** does not occur in the list or occurs only once then the function should return **`false`**.

c. An **`operator -=`** that deletes from the list all repetitions of a given integer i. Thus, if the list initially has more than one node storing value **`i`** then the resulting list should have only one node storing **`i`** (i.e. all nodes storing **`i`** but one are deleted from the list). If integer **`i`** does not occur in the list then the list remains unchanged. Thus, if list
`L = 1 → 4 → 3 → 2 → 4 → 7`
then
`L -= 4;`
results in
`L = 1 → 4 → 3 → 2 → 7` (or, **`L = 1 → 3 → 2 → 4 → 7`**).

Add your code to the file **`ex1.cpp`** that also contains a **`main()`** function to test your code. The **`main()`** function cannot be changed. A possible output can be found in the file **`ex1_out.txt`**.

Exercise 2 [30 p]

Write a program that starts by reading a text file storing information about movies. The file consists of a sequence of lines such that each line contains first an integer representing a score (from 1 to 5) and then a movie name (e.g. "*Blade Runner*"). Assume that each movie occurs only once in the file.

The program then performs, step by step, the actions described below. Notice that you should use the **Standard Template Library** (**STL**), whenever possible. 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), if possible.

Add your code to the file **`ex2.cpp`**. You do not need to create different functions for each of the (small) exercises below. You can write the corresponding code directly in the **`main`** function, as indicated in **`ex2.cpp`**, and do not delete the comments given in the **`main`**.

The file **`data.txt`** can be used to test your program and the expected output is available in the file **`ex2_out.txt`**.

a. 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 print an error message and terminate.

b. Declare a new type **`Table`** representing a **`map`** such that the key is a string (movie name) and the associated value is an integer (movie score).

c. Load the file into a variable **`T`** of type **`Table`**. Note that movie names stored in **`T`** should be converted to upper case letters (e.g. "*Blade Runner*" is converted to "*BLADE RUNNER*"). The file may be garbled and any lines not as described above should be simply ignored.

d. Display, to **`cout`**, the number of movies stored in table **`T`**.

e. Display, to **`cout`**, all the information stored in table **`T`** (i.e. movie names and corresponding scores) sorted alphabetically by movie name.

f. Display, to **`cout`**, all the information stored in table **`T`** (i.e. movie names and corresponding scores) sorted increasingly by score.

g. Request to the user the name of a movie and then display the score for the given movie. If the movie does not exist in table **`T`** then display the message "**`Movie not found!!`**".

Exercise 3 [35 p]

Design a class hierarchy for representing *celestial bodies*, such as *stars* and *planets*. Stars and planets are regarded as two different kinds of celestial bodies. As usual, a planet always belongs to another celestial object. For instance, planet **`Earth`** belongs to star **`Helios`** and planet **`Moon`** belongs to planet **`Earth`**.

`Celestial_Body` shall be base class for other celestial body classes. This class should store the *name* (**`string`**) of the celestial body. When a new **`Celestial_Body`** object is created, its *name* shall be explicitly initialized, and thereafter it should not be possible to change it. Moreover, the class should have the following member functions.

- A function **`get_name()`** that returns the *name* of a celestial body.
- An **`operator<<`** that displays information about a celestial body.

`Star` is a direct subclass of **`Celestial_Body`**, and shall also store the *name of the galaxy* it belongs to (**`string`**). When a new **`Star`** is created, its *name* and the *name of the galaxy* should be explicitly initialized.

`Planet` is also a direct subclass of **`Celestial_Body`** and every planet stores a *pointer to the celestial body* it belongs to (e.g., a **`Star`**) and whether the planet is *populated* (**`bool`**). When a new **`Planet`** is created, its *name* and the *pointer to the celestial body* it belongs to are always explicitly initialized, and thereafter it should not be possible to change them. If no initial value about population is given then by default it is assumed to be a "not populated" planet. Moreover, the class should have the following member functions, in addition to the functionality described for celestial bodies.

- A function **`get_celestial_body()`** that returns a pointer to the celestial body to which the planet belongs.
- A function **`is_populated()`** that returns **`true`,** if the planet is populated. Otherwise, it returns **`false`**.

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

- It should not be possible to create **`Celestial_Body`** objects that are not instances of one of the **`Celestial_Body`** sub-classes (such as **`Star`** or **`Planet`**).
- It should be possible to keep track of the number of existing celestial bodies.

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

Add your code to the file **`ex3.cpp`** that also contains a **`main()`** function to test your code. Add to the **`main`** function the code needed to display the number of celestial bodies. The **`main()`** function cannot be changed, otherwise. The expected output is available in the file **`ex3_out.txt`**.