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

Exam for

TNG033: Programming in C++

May 28th, 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 [25 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. The only (member) functions available are a constructor, to create an empty list, 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 constructor that, given an integer **`i`**, creates a list storing the single value **`i`**. Thus, the declaration **`List L(5);`** creates a list storing value **`5`**.
- An **`operator+`** such that, given two lists **`L1`** and **`L2`**, **`L1+L2`** appends a copy of **`L2`** to the end of list **`L1`**. Moreover, **`L1+L2+L3+`**...**`+`**$\mathbf{L_n}$ appends a copy of **`L2`** to the end of **`L1`**, and then appends a copy of list **`L3`** to the end of the updated list **`L1`**, and so on.

To get full points, operation **`L1+L2`** should visit at most once each node of **`L1`** and each node of **`L2`**.

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

Exercise 2 [30 p]

Write a program that reads integer numbers from a given text file and 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 **`numbers.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 file and open it. If opening the file does not succeed then the program should print an error message and terminate.

b. Read all numbers from the input file and store them in a **`vector`**, called **`vec1`**. Note that the input file may be garbled, i.e. it may contain no numeric values. Any rubbish is simply ignored.

c. Output, to **`cout`**, the number of integers read into **`vec1`**.

d. Sort the numbers in **`vec1`**.

e. Display, to **`cout`**, all numbers stored in **`vec1`**, first in increasing order and then decreasingly. The values should be displayed with one space between them.

f. Assume that **`%`** represents the reminder of integer division. Integers **`i1`** and **`i2`** are *congruent equivalent*, if and only if **`(i1 % 5) = (i2 % 5)`**. For instance, **`1`** is congruent equivalent to **`16`** because **`(1 % 5) = (16 % 5)`**.

Test whether the numbers stored in the first half of the vector **vec1** are congruent equivalent to the numbers store in the second half of the vector. If the vector stores an odd number of values then ignore the first element stored in the vector. If the test succeeds then the program should display to **cout** the message "**Congruence test succeeded!!**". Otherwise, the program writes the message "**Congruence test failed!!**".

For instance, consider the vector

vec1 = <0, 1, 2, 3, 15, 16, 17, 18>.

Then, the values stored in the first half of the vector (**0**, **1**, **2**, and **3**) are congruent equivalent to the values stored in the second half (**15**, **16**, **17**, and **18**), since **(0 % 5) = (15 % 5)**, **(1 % 5) = (16 % 5)**, **(2 % 5) = (17 % 5)**, and **(3 % 5) = (18 % 5)**. But, if

vec1 = <0, 1, 2, 3, 15, 20, 17, 18>

then the values stored in the first half of the vector (**0**, **1**, **2**, and **3**) are not congruent equivalent to the values stored in the second half (**15**, **20**, **17**, and **18**), since **(1 % 5) ≠ (20 % 5)**.

g. Copy the values in vector **vec1** to another vector **vec2**.

h. Replace all values, in the vector **vec2**, larger than **15** by **20**. Then, display the values stored in **vec2**, with one space between them.

i. Test whether the numbers stored in vector **vec1** are congruent equivalent to the values stored in vector **vec2**. If the test succeeds then the program should display to **cout** the message "**Congruence test succeeded!!**". Otherwise, the program writes the message "**Congruence test failed!!**".

Exercise 3 [45 p]

Design a class hierarchy for handling different types of customers of a company, regular **`Customers`**, **`Good_Customers`**, and **`Bad_Customers`**. Design your classes with care and keep with the given specifications.

`Customers` can order products from the company on credit. For every customer there is a credit limit. The following information is stored for every customer: the customer id (an **`int`**); the credit limit (an **`int`**); and, the debt of the customer to the company (a **`double`**).

The class should have the functionality described below.

- A constructor. When a customer is created the credit limit must be explicitly initialized and it should not be possible to change it. Moreover, the new customer's debt to the company should be set to zero.
- A function **`get_ID()`** that returns the customer's id.
- A function **`to_pay()`** that returns how much the customer should pay to the company, i.e. the debt.
- A function **`get_credit_limit()`** that returns the customer's credit limit.
- A boolean function **`update_debt(double d)`** that adds **`d`** to the customer's debt. Since there is a credit limit, the debt is only updated if it does not exceed the credit limit. If it is not possible to update the customer's debt then the function returns false. Otherwise, it returns true.
- An **`operator<<`** that displays the customer's invoice. The invoice should contain the customer id and the amount to be paid. The credit limit should not be shown in the invoice.

`Good_Customers` are customers who have a discount (a **`double`**) in the next invoice. The discount is set when the customer is created and it should be possible to update this value, if needed. To this end, a function named **`set_discount`** can be used. Note that the function **`to_pay()`** should take into accout the discount value offered to the customer. Moreover, the invoices for **`Good_Customers`** should clearly indicate both the debt, the discount offered, and the total amount to be paid (i.e the debt minus the discount).

`Bad_Customers` are customers who pay an interest rate (an **`int`**) over their debt. The interest rate (e.g. 10%) is set when the customer is created and it should be possible to update it, if needed. To this end, a function named **`set_interest_rate`** can be used. Note that the function **`to_pay()`** should take into account the interest rate, i.e. the customer must pay the debt plus 10% of the debt (if the interest rate is set to 10%). Moreover, the invoices for **`Bad_Customers`** should clearly indicate the debt, the interest rate, the extra amount to be paid due to the interest rate, and the total amount to be paid.

The classes should also keep information about the largest discount offered to a (good) customer and the highest interest rate set for a (bad) customer.

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 largest discount offered to a (good) customer and the highest interest rate set for a (bad) customer. The **`main()`** function cannot be changed, otherwise. The expected output is available in the file **`ex3_out.txt`**.