LINKÖPINGS UNIVERSITET
Institutionen för datavetenskap
Avd. för Programvara och system
Tommy Olsson

2009-08-12

Computer examination in

TDDD38 Advanced Programming in C++

Date:	2009-08-14
Time:	8-13
Place:	IDA, SU-locations, Building B, plan 3 (second floor)
On duty:	Tommy Olsson, 28 1954 **Teacher on duty will visit the locations approximately once every hour, and should otherwise only be called on for really urgent matters.**
Administrator:	Madeleine Häger Dahlqvist, 28 2360
Means of assistance:	An English-* vocabulary may be brought to the exam. An overview of the standard library, "STL Quick Reference", is available during the exam. The system man pages are available (*Note*: some man pages are not fully compliant with the C++ standard. If the compiler and a man page seems to disagree, please check with STL Quick Reference or ask teacher on duty, algorithm for_each is one example, shall return F, not void). Scrap paper and pen is allowed. No other means of assistance (printed or electronic) are allowed.
Grading:	Each theory question is worth 1 credit point, and each programming problem is worth 5 credit points. The maximum credit points is 25, and minimum to pass is 11 points, **0-10** points for grade **U/FX**, **11-14** points for grade **3/C**, **15-18** points for grade **4/B**, and **19-25** points for grade **5/A**.
Instructions:	**NOTE: Don't log out at any time during the exam. Only log out when you have finished.**
Instructions:	Answers may be given in Swedish. Assumptions made, besides those given, must be stated and may not alter the meaning of the problems. All answers and programs are to be written on text files in the login directory (working directory). Please, don't create any subdirectories. Please, use the file names given – upper and lower case is significant. In case names for program entities are given, use such names exactly as given. Code should be written in a highly readable way (e.g. concerning naming, formatting, etc.). Given files are found in the subdirectory `given_files`. **Please delete all backup files, compilation products, etc., before you log out!** Leave only requested files, thank you! *Design with care – use C++ properly!*

Theory questions

Note: Please copy the file THEORY.TXT from directory given_files to the working directory, fill in your personal data at the top of the file, and give the answers to the theory questions in that file.

1. Give at least two different uses of the keyword **static** can be used, and describe the effect of **static** in each case.

❑

2. Which are the so called *special member functions*, and what is their respective purpose/use?

❑

3. Given the following template declaration

```
template<class T> T fun(const T t);
```

Which requirements must an instantiation type fulfill, in general?

Is there anything we can say about requirements on T from the template declaration above?

❑

4. Which is the earliest possible resumption point in a program, after an exception has been thrown in some block?

Which is the latest resumption point, if the program is to, at all, survive?

❑

5. Compare and contrast overloading of a function (name) with the definition of a template for the function (name), e.g. for which types do they work, etc.?

❑

Programming problems

6. Write your code on a single file named program6.cc.

Define a polymorphic class hierarchy for dates (year, month, and day). The hierarchy shall have an abstract class **Date** as common base, which purpose is to define a simple common interface, basically consisting of a member function print() and **operator**<<.

- print() shall take an std::ostream as argument and print the date onto that ostream. How a date is to be printed depends on the specific type of Date, se below.
- **operator**<< shall be overloaded for Date, to print a Date object onto an ostream, in accordance to the specific type of date.

Only required special member functions should be declared in addition to these two functions. Copying Date objects is not to be allowed.

Date shall have two direct, non-abstract subclasses, **Int_Date** and **Cstring_Date**:

- **Int_Date** shall store the date as three **int**s, one for the year (e.g. 2009), one for the month (1-12), and one for the day of the month (1-31).

  Year, month and day shall always be given when a new Int_Date object is created.

  print() shall output an Int_Date on the form “2009-08-14”.

- **Cstring_Date** shall store the date as three dynamically allocated **char**[], one for the year (e.g. “2009”), one for the month (e.g. “August”) and one for the day (e.g. “14”).

  Year, mont and day shall always be given, as **char***, when a new Cstring_Date object is created.

  print() shall output a Cstring_Date on the form “August 14, 2009”.

  Make sure to handle the memory for Cstring_Date correctly, in all respects.

  Hint: Use std::strlen() to determine the length of a **char*** string, and std::strcpy() to copy **char*** string. Please, se the man pages for more information about C string library functions.

Write a test program, which uses a std::vector to store pointers to dynamically allocated Date objects. Store some different Date objects in the vector and then iterate over it. For each object, test **operator**<<, then delete the object and set the pointer in the vector to a null pointer.

Design with care!

❑

7. Write your code on a single file named program7.cc.

In directory given_files there is a file, exam_result.txt, containing results from an exam. On each line a student's identity number and the number of points the student received is found. Write a program that reads the file and produce a result list as below:

```
Student      Points  Mark

121212-1006    23     A
121212-1238    24     A
121212-1295    21     A
121212-1444    19     A

121212-1014    17     B
121212-1386    18     B
121212-1550    16     B

121212-1055    11     C
121212-1063    14     C
121212-1071    11     C
121212-1147    12     C
121212-1451    12     C
121212-1469    12     C
121212-1584    11     C

121212-1022     7     FX
121212-1220     9     FX
121212-1311     3     FX
121212-1329     5     FX
121212-1519     6     FX
121212-1592    10     FX
```

The mark (A, B, C or FX) is calculated by the program. Points can vary from 0 to 25, where 0-10 points give mark FX, 11-14 give mark C, 15-18 give mark B, and 19-25 give mark A.

As you can se, the list is ordered primarily by mark, and secondly by identity number.

The program shall read the results from the file, and, for each student, calculate the mark and the store the identity number, points, and mark in a suitable standard container.

This is a standard library exercise – here are some requirements:

- Use only *one* standard container as internal storage.
- Use standard library components whenever possible – avoid hand-written loop, such as **for**, **while** or **do**, prefer e.g. standard algorithms instead.
- When needed, define components of your own, e.g. functions or function objects, to use in combination with standard library components.

Note: Non-member functions related to a class type, standard or user defined, should, or must, be declared in the same namespace as the class in question. This because of the *argument dependent lookup*, or Koenig lookup, used in C++). Compile error messages are not always obvious.

❑

8. Copy file program8.cc from subdirectory given_files to your working directory, and write your code on that file.

Class **Wrapper** below, and some instructions is given in program8.cc.

```
class Wrapper {
  public:
    Wrapper(const int);        // initialize value_
    void set(const int);       // assign new value to value_
    int get() const;           // return value_
    std::string str() const;   // return string representation for value_
  private:
    int value_;
};
```

Define the declared member functions, at least str() shall have a separate definition (the other ones may be defined in-class).

Make Wrapper a template and generalize it in two ways using template techniques:

1) make it possible to vary *the stored type*, i.e. the type for value_, so Wrapper can be used for any type T fulfilling needed requirements, e.g. that T have a string representation.

2) make it possible to give a *string representation policy*, to be used by function str() to produce and return a string representation of the stored value. If we have a policy named Hexadecimal, an instance of Wrapper for **int** and Hexadecimal can be declared as follows, and when str() is called, policy Hexadecimal is to be used by str() to produce the string representation of value:

```
Wrapper<int, Hexadecimal> obj(4711);
cout << obj.str();    // the string 0x1267 is to be returned by str()
```

A string representation policy shall have one member function convert(), taking a value of type T and returning the value converted to a string, according to the policy. Define two policy classes, Hexadecimal and Citated:

- **Hexadecimal** is suited for integer types, to produce a string representation according to hexadecimal representation in C++, i.e. starting with 0x and then the value in hexadecimal representation (digits 0-9 and A-F). Hint: stream manipulator hex.
- **Citated** is to put citation marks around the ordinary string representation for the value, i.e. if value is 4711, the string returned by str() shall be ”4711”, not just the string 4711. If the stored type is std::string and the value is foobar, the string ”foobar” shall be returned by str().

Also write a test program in main() for Wrapper and the policy classes, using instances for both **int** and std::string.

❑

9. Write your code on a single file named program9.cc.

Suppose we have a frequently used, heavy-to-compute function taking one argument and returning a result, e.g.

```
long int heavy(int x)
{
  return x * x * x * x;
}
```

and the function is called frequently by the application.

By encapsulating the heavy-to-compute function in a class, we can successively save given arguments and their corresponding result, so, if a certain argument is given several times, the result can be returned directly, without calling the heavy-to-compute function more than once for each unique argument.

Define a function object class named Function_Cache, with two template type parameters, one for the argument type and one for the return type of the heavy-to-compute function.

Function_Cache shall store a pointer to the heavy-to-compute function, and this pointer is to be bound to the function through a constructor argument, i.e. when creating a Function_Cache object we instantiate with the argument and return type of the heavy-to-compute function, and we give the pointer to the heavy-to-compute function as argument to the constructor for Function_Cache.

Arguments and corresponding results are to be stored in a std::map member of Function_Cache.

It shall not be allowed for copying or assigning Function_Cache objects.

Write a test program in main() testing Function_Cache for the given heavy-to-compute function heavy, and with both different and repeated arguments. Put trace printouts at some strategic points to show what is happening when calling the function object.

❑